Avaliação de algoritmos de escalonamento de aplicações paralelas em processadores heterogêneos

Breno Corrêa Silva Costa

Universidade Federal de Uberlândia
Faculdade de Computação

Uberlândia
2022

Breno Corrêa Silva Costa

Avaliação de algoritmos de escalonamento de aplicações paralelas em processadores heterogêneos

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, Minas Gerais, como requisito exigido parcial à obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Paulo Henrique Ribeiro Gabriel

Uberlândia
2022

Aos meus pais, Marnilson de Lima Costa e Sônia Maria da Silva Costa,
os meus avós Enilzia de Lima Costa e Marcos Costa, Alice Maria da Silva e Francisco Correa Silva,
minhas irmãs Bianca Silva Costa Queiroz e Brenda Silva Costa Rivelini.

Agradecimentos

Em agradecimento, primeiramente, queria agradecer a Deus pelo que Ele é, por cuidar de mim em cada momento da minha vida e por estar me guiado nos caminhos d'Ele.

Aos meus pais, pela educação e lições de vida que me deram durante toda minha criação e até mesmo na vida adulta. Por sempre terem me apoiado nos meus sonhos, inclusive na minha vida acadêmica, e por todo cuidado e dedicação que tiveram comigo, sempre sendo muito sábios e pacientes.

Minhas irmãs, por toda a força de incentivo que me deram. Principalmente, a Bianca Costa, que me auxiliou diversas vezes e me ajudou bastante na adaptação, quando tive que morar em outra cidade, fora da casa dos pais, para fazer a graduação que tanto queria.

Ao meu amigo Salomão Alves, que esteve comigo em vários momentos da graduação, me ajudando nos estudos e trabalhos em grupo. Agradecer também minha namorada Beatriz Santana, por ter me apoiado durante o desenvolvimento desse trabalho, não me deixando desanimar do que estava construindo.

Aos meu mestres, por todo os ensinamentos teóricos e práticos, além das lições que levarei por toda a vida. Em especial, ao meu orientador, professor Paulo Henrique, por me ter me orientando e me auxiliado bastante durante a realização dessa pesquisa. E também, ao meu tutor do PET, Renan Cattelan, por todos os ensinamentos e base que nos deram no PET, que me preparou muito para o mercado e estudos.

Resumo

A computação paralela tem sido empregada para alcançar melhores desempenhos na execução de aplicações cada vez mais complexas. Entretanto, para uma execução paralela adequada, é necessário um algoritmo de escalonamento eficiente, responsável por organizar toda a execução, e concluir o processamento no menor tempo possível. Este trabalho de conclusão de curso foi desenvolvido com o objetivo de estudar e reproduzir alguns dos algoritmos de escalonamento presentes na literatura, submetendo-os a ambientes de execução diversos, de modo a comparar seus resultados e obter características não citadas em seus artigos de origem. Os resultados são apresentados de forma visual, utilizando gráficos e tabelas, e são discutidos os principais pontos observados nesses algoritmos, em comparação com o que é proposto pelo seus autores.

Palavras-chave: Computação Paralela, Escalonamento de Tarefas, Processador, Aplicações Paralelas.

Lista de ilustrações

Figura 1 – Exemplo de DAG com oito tarefas. 18
Figura 2 – Exemplo de DAG no formato STG. 32
Figura 3 – Padrão de DAG proposto neste trabalho. 32
Figura 4 – DAGs disponíveis no Kasahara Laboratory. 36
Figura 5 – Média e desvio-padrão do makespan para os algoritmos IPEFT, IHEFT, CPOP e HEFT com grafos de 50 tarefas com alto custo de comunicação. 37
Figura 6 – Média e desvio-padrão do load balance para os algoritmos IPEFT, IHEFT, CPOP e HEFT com grafos de 50 tarefas com alto custo de comunicação. 38
Figura 7 – Média e desvio-padrão do makespan para os algoritmos IPEFT, IHEFT, CPOP e HEFT com grafos de 50 tarefas com baixo custo de comunicação. 40
Figura 8 – Média e desvio-padrão do load balance para os algoritmos IPEFT, IHEFT, CPOP e HEFT com grafos de 50 tarefas com baixo custo de comunicação. 41
Figura 9 – Média e desvio-padrão do makespan para os algoritmos IPEFT, IHEFT, CPOP e HEFT com grafos de 50 tarefas com alto custo computacional. 42
Figura 10 – Média e desvio-padrão do load balance para os algoritmos IPEFT, IHEFT, CPOP e HEFT com grafos de 50 tarefas com alto custo computacional. 43
Figura 11 – Média e desvio-padrão do makespan para os algoritmos IPEFT, IHEFT, CPOP e HEFT com grafos de 50 tarefas com baixo custo computacional. 45
Figura 12 – Média e desvio-padrão do load balance para os algoritmos IPEFT, IHEFT, CPOP e HEFT com grafos de 50 tarefas com baixo custo computacional. 46
Figura 13 – Média e desvio-padrão do makespan para os algoritmos IPEFT, IHEFT, CPOP e HEFT com grafos de 50, 100 e 300 tarefas com custo computacional homogêneo. 47

Figura 14 – Média e desvio-padrão do load balance para os algoritmos IPEFT, IHEFT, CPOP e HEFT com grafos de 50, 100 e 300 tarefas com custo computacional homogêneo. 48

Lista de tabelas

Tabela 1 – Média e desvio-padrão do makespan para os algoritmos IPEFT, IHEFT, CPOP e HEFT com grafos de 50 tarefas com alto custo de comunicação. 38
Tabela 2 – Média e desvio-padrão do load balance para os algoritmos IPEFT, IHEFT, CPOP e HEFT com grafos de 50 tarefas com alto custo de comunicação. 39
Tabela 3 – Média e desvio-padrão do makespan para os algoritmos IPEFT, IHEFT, CPOP e HEFT com grafos de 50 tarefas com baixo custo de comunicação. 40
Tabela 4 – Média e desvio-padrão do load balance para os algoritmos IPEFT, IHEFT, CPOP e HEFT com grafos de 50 tarefas com baixo custo de comunicação. 41
Tabela 5 – Média e desvio-padrão do makespan para os algoritmos IPEFT, IHEFT, CPOP e HEFT com grafos de 50 tarefas com alto custo computacional. 43
Tabela 6 – Média e desvio-padrão do load balance para os algoritmos IPEFT, IHEFT, CPOP e HEFT com grafos de 50 tarefas com alto custo computacional. 44
Tabela 7 – Média e desvio-padrão do makespan para os algoritmos IPEFT, IHEFT, CPOP e HEFT com grafos de 50 tarefas com baixo custo computacional. 45
Tabela 8 – Média e desvio-padrão do load balance para os algoritmos IPEFT, IHEFT, CPOP e HEFT com grafos de 50 tarefas com baixo custo computacional. 46
Tabela 9 – Média e desvio-padrão do makespan para os algoritmos IPEFT, IHEFT, CPOP e HEFT com grafos de 50, 100 e 300 tarefas com custo computacional homogêneo. 48
Tabela 10 – Média e desvio-padrão do load balance para os algoritmos IPEFT, IHEFT, CPOP e HEFT com grafos de 50, 100 e 300 tarefas com custo computacional homogêneo. 49

Sumário

1 INTRODUÇÃO 15

2 FUNDAMENTAÇÃO TEÓRICA 17
2.1 Problema de escalonamento de tarefas 17
2.2 Algumas definições 19

3 DESENVOLVIMENTO 21
3.1 Linguagem de programação utilizada 21
3.2 Algoritmos selecionados 22
3.2.1 Heterogeneous Earliest Finish Time 23
3.2.2 Critical Path on a Processor 24
3.2.3 Improved Heterogeneous Earliest Finish Time 26
3.2.4 Improved Predict Earliest Finish Time 28
3.3 Geração dos DAGs 31
3.3.1 Standard Task Graph 31
3.3.2 Padrão proposto 32
3.3.3 Algoritmo de conversão 33

4 EXPERIMENTOS 35
4.1 Métricas de comparação 35
4.2 Geração dos ambientes de execução 36
4.2.1 Cenário 1: Alta variação no custo de comunicação 36
4.2.2 Cenário 2: Variação baixa de custo de comunicação 39
4.2.3 Cenário 3: Variação alta de custo computacional 41
4.2.4 Cenário 4: Variação baixa de custo computacional 44
4.2.5 Cenário 5: Ambiente homogêneos 47
4.3 Discussão 48
4.3.1 Resultados nos ambientes heterogêneos 48

4.3.2 Resultados nos ambientes homogêneos 50

5 CONCLUSÕES 51

REFERÊNCIAS 53

Capítulo 1

Introdução

A computação paralela é uma forma de computação em que vários cálculos são realizados simultaneamente, operando sob o princípio de que grandes problemas geralmente podem ser divididos em problemas menores, os quais, então, são resolvidos concorrentemente (ALMASI; GOTTLIEB, 1989). Porém, esses "pedaços menores" de um problema podem ser distribuídos de diversas formas para serem processados e, dependendo de como eles forem organizados, o desempenho do programa final é afetado.

Diversos estudos ao longo dos anos focaram em desenvolver algoritmos que pudessem obter as melhores soluções de organização das aplicações computacionais nos processadores, a fim de serem executadas no menor tempo possível. Esses algoritmos são conhecidos como algoritmos de **escalonamento de tarefas** (ou *task scheduling algorithms*, em Inglês). Nesse contexto, o termo *tarefas* se refere aos pedaços menores de um problema, no caso uma aplicação computacional. Essas tarefas podem ser *threads*, processos ou fluxos de dados, de forma que essa unidade não pode mais ser divida e, assim, o seu conteúdo deve ser executado sequencialmente (ROBERT, 2011; PLANETA, 2015).

É importante destacar que o problema de escalonamento de tarefas é computacionalmente complexo. De fato, pode-se demonstrar que esse problema é NP-Difícil mesmo para cenários em que as tarefas são independentes (ULLMAN, 1975). Por essa razão, muitos estudos têm como foco o desenvolvimento de algoritmos de escalonamento com baixa complexidade e que, na maioria das vezes, encontram soluções adequadas ou quase ótimas. Nesse contexto, diversas heurísticas têm sido propostas (GRAHAM, 1966; KWOK; AHMAD, 1996; RĂDULESCU; GEMUND, 1999; TOPCUOGLU; HARIRI; WU, 2002; RIOS et al., 2009; PLANETA, 2015; ZHOU et al., 2016).

No entanto, uma análise detalhada desses algoritmos, muitas vezes, não é apresentada na literatura. Muitos desses trabalhos definem um ambiente de testes que não é utilizado em outros. Por exemplo, Kwok e Ahmad (1996) e Rădulescu e Gemund (1999), em seus respectivos trabalhos, geram diferentes conjuntos de tarefas com base nos mesmos algoritmos. Assim, não é possível fazer uma comparação direta entre ambos; seria necessário implementar cada um dos algoritmos apresentados e, somente então, realizar uma série

de experimentos considerando os mesmos dados de entrada.

Problema similar é observado no trabalho de Topcuoglu, Hariri e Wu (2002), que geraram as entradas de dados aleatoriamente em seu trabalho. Assim, embora consigam mostrar empiricamente o bom desempenho dos algoritmos propostos, torna-se inviável construir um ambiente de comparação adequado quando se propõe novos algoritmos. Por exemplo, Rios et al. (2009) estendem um dos algoritmos de Topcuoglu, Hariri e Wu (2002), mas não realizam experimentos sob as mesmas condições. O mesmo se repete nos trabalhos de Planeta (2015), Zhou et al. (2016) e AlEbrahim e Ahmad (2016): todos propõem novos algoritmos mas não realizam comparações sob o mesmo conjunto de instâncias.

Deve-se destacar aqui que os grafos gerados por todos esses autores não foram disponibilizados em repositórios públicos. Além disso, como já mencionado, a complexidade inerente do problema de escalonamento faz com que desenvolvimento de algoritmos ótimos seja impraticável para instâncias grandes. Assim, torna-se interessante criar uma base de casos de teste que possa ser utilizada em diferentes trabalhos, além de se apresentar resultados de execução para diferentes algoritmos já consolidados na literatura.

Este trabalho de conclusão de curso foi guiado por um estudo amplo de vários algoritmos de escalonamento de tarefas e suas características; em seguida, foram escolhidos quatro que teriam mais relevância para serem estudados de forma mais profunda: *Heterogeneous Earliest Finish Time* (HEFT), *Critical Path on a Processor* (CPOP), *Improved Heterogeneous Earliest Finish Time* (IHEFT) e *Improved Predict Earliest Finish Time* (IPEFT). Os dois primeiros foram propostos por Topcuoglu, Hariri e Wu (2002) e os outros dois por AlEbrahim e Ahmad (2016) e Zhou et al. (2016), respectivamente. Todos esses algoritmos foram projetados para ambientes computacionais heterogênicos, ou seja, ambientes distribuídos compostos por processadores distintos entre si (como, por exemplo, grades e nuvens computacionais). Após os estudos e compreensão sobre esses algoritmos, foi desenvolvido todo o código que reproduz o exato comportamento deles, para que seja possível executá-los e obter uma base de comparação sobre seus resultados.

Outra coisa em comum desses quatro algoritmos é o fato de que todos eles requerem como entrada uma estrutura de dados em comum: um grafo acíclico direcionado, cujo objetivo é representar uma aplicação paralela que será distribuída entre os diversos processadores. Para isso, foi necessário desenvolver um programa de geração desses grafos, tornando possível a execução dos algoritmos em diversos cenários. Assim, este trabalho apresenta um conjunto de casos de teste da execução de quatro algoritmos, sendo que tais resultados podem ser empregados como base de comparação para outros trabalho.

O restante desta monografia está organizada como descrito a seguir. No Capítulo 2, são apresentados conceitos fundamentais para se compreender os algoritmos aqui avaliados. No Capítulo 3, são descritos esses algoritmos e as ferramentas utilizadas em sua implementação, bem como o modelo de avaliação proposto. Os resultados são apresentados e discutidos no Capítulo 4, e as conclusões no Capítulo 5.

Capítulo 2

Fundamentação Teórica

Nesse capítulo são apresentados os principais conceitos necessários para o desenvolvimento deste trabalho. Inicialmente, descreve-se o problema de escalonamento de tarefas, onde são definidos os termos empregados no restante do texto. Em seguida, são apresentadas algumas definições de conceitos necessários para compreender o funcionamento dos algoritmos implementados.

2.1 Problema de escalonamento de tarefas

Com o avanço da comunicação de redes de alta velocidade e a evolução de CPUs e GPUs, tornou-se cada vez mais comum encontrar arquiteturas computacionais distintas sendo coordenadas para resolver tarefas computacionais complexas (BITTENCOURT et al., 2018). Esses ambientes computacionais são conhecidos como sistemas distribuídos heterogêneos, ou seja, sistemas cujos nós computacionais não são idênticos, podendo ser distintos em velocidades de processamento, capacidade de recursos ou até mesmo em consumo energéticos. Em oposição a esses ambientes, existem também os sistemas distribuídos homogêneos, em que os nós computacionais são todos idênticos (como exemplo, tem-se os supercomputadores e os *clusters* computacionais).

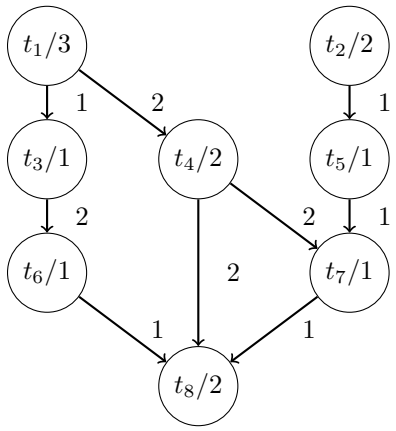
Uma das principais decisões de projeto relacionada ao desenvolvimento desses ambientes computacionais é a escolha do algoritmo responsável pelo escalonamento de tarefas (BITTENCOURT et al., 2018). O problema de escalonamento de tarefas (do inglês, *task scheduling problem*) emerge pela necessidade de organizar previamente como uma aplicação computacional irá ser executada no menor tempo possível em processadores paralelos. Esse paralelismo só é possível pelo fato de que as aplicações podem ser divididas em tarefas menores, que podem ser executadas simultaneamente em processadores diferentes interconectados (ROBERT, 2011). Porém, algumas restrições devem ser respeitadas, como a dependência de dados entre algumas tarefas. Essa relação de dependência funciona de forma que uma tarefa só poderá iniciar sua execução se tiver certos dados, vindos do resultado de outras tarefas; nesse caso, é necessário computar o custo de comu-

nicação entre duas tarefas distintas. Caso ambas as tarefas sejam executadas no mesmo processador, o custo de custo de comunicação entre elas é nulo (ROBERT, 2011).

Nesse contexto, a comunicação entre diferentes tarefas é modelada por meio de grafos acíclicos orientador (DAGs, do inglês *directed acyclic graphs*). Um DAG é um grafo cujas arestas orientadas e sem a presença de ciclos; assim, para qualquer vértice v desse grafo, não há nenhum caminho direcionado que passe por v mais de uma vez. Nesse modelo, cada grafo representa uma aplicação (o programa) paralela e cada vértice (ou nó) representa uma tarefa dessa aplicação. Essa divisão da aplicação em tarefas é que torna possível sua paralelização de forma que as tarefas serão executadas simultaneamente em diversos processadores. Porém, para algumas tarefas, haverá uma ordem de precedência, representada pelas arestas direcionadas, as quais indicam quais tarefas devem ser concluídas para iniciar outras.

Na Figura 1, é apresentado um exemplo de DAG com oito tarefas. Os custos apresentados nos vértices indicam a carga de trabalho da tarefa (por exemplo, o número de instruções de processamento requeridas). Já os rótulos das arestas representam o custo de comunicação, ou seja, o total de dados transferidos de uma tarefa predecessora t_i para uma tarefa sucessora t_j.

Figura 1 – Exemplo de DAG com oito tarefas.



Fonte: Adaptado de Silva e Gabriel (2020).

Formalmente, o problema de escalonamento de tarefas pode ser definido da seguinte maneira. Seja $P = \{p_1, \ldots, p_m\}$ um conjunto de m processadores heterogêneos. Nessa monografia, assume-se que todos os processadores estão interconectados, ou seja, um processador $p_j \in P$ pode se comunicar com todos os demais. Além disso, toda a comunicação entre processadores deve ser executadas sem contenção. Analogamente, $T = \{t_1, \ldots, t_n\}$ representa o conjunto das n tarefa, as quais são organizadas em um DAG, como o ilustrado na Figura 1.

Além disso, vértices que precedem outros recebem nomes específicos (lembrando que todas as arestas são direcionadas). Assim, os vértices de origem são chamados de *pais* e os de destino são os *filhos*. Um vértice que não possui pais representa uma *tarefa de entrada* (t_{entry}) e o que não possui filho é uma *tarefa de saída* (t_{exit}). Na Figura 1 existem duas tarefas de entrada; porém, muitos algoritmos exigem que haja apenas uma. Nesse caso, é criado um vértice artificial, sem custo, ligado a esses dois.

Por ser um ambiente computacional heterogêneo, uma mesma tarefa $t_i \in T$ poderá ter custo computacional diferente em processadores distintos. Devido a isso, define-se a matriz W contendo as dimensões $n \times m$ tal que cada elemento w_{ij} representa o tempo de execução da tarefa t_i no processador p_j. Além disso, pode haver diferentes latências de comunicação entre os processadores. Para isso, define-se a matriz B com dimensões $m \times m$ que armazena a latência de rede entre cada par de processador.

Dessa maneira, o custo de comunicação c_{ij} entre duas tarefas t_i e t_k é definido conforme mostrado na Equação (1). Nesse caso, supõe-se que a tarefa t_i foi atribuída ao processador p_x e que t_j será executada no processador p_y.

$$c_{ij} = L_x + dados_{ij}/B_{xy} \tag{1}$$

Na Equação 1, L é um vetor de dimensão m que representa a latência para iniciar a transferência de dados a partir de um determinado processador (L também é chamado custo de inicialização de comunicação). A variável $dado_{ij}$ representa o total de bytes enviados de t_i para t_j. Destaca-se, ainda, que quando t_i e t_j são atribuídas ao mesmo processador, o custo de comunicação c_{ij} é igual a zero.

2.2 Algumas definições

A partir do formalismo definido na Seção 2.1, são apresentadas, nesta seção, algumas definições necessárias para compreender o funcionamento dos algoritmos estudados neste trabalho.

Definição 2.1. *Insertion-based scheduling policy* é um método que leva em consideração o intervalo de tempo (lacuna) em que um processador permanece ocioso. Caso alguma lacuna seja maior do que o tempo necessário para executar determinada tarefa t_i, então t_i será inserida nesse espaço de tempo, obviamente respeitando todas as dependências dessa tarefa. Algoritmos que não usufruem dessa política levam apenas em consideração o momento depois que o processador conclui todas as tarefas já atribuída a ele.

Definição 2.2. $pred(t_i)$ é o conjunto de todos as tarefas que são predecessoras diretas de t_i, se houver (se não houver predecessores, então $|pred(t_i)| = 0$ e t_i passa a ser conhecida como *tarefa de entrada*. Analogamente, $succ(t_i)$ é o conjunto de todas as tarefas sucessoras diretas de t_i (caso não haja sucessores, então $|succ(t_i)| = 0$ e t_i é uma *tarefa de saída*.

Definição 2.3. $AFT(t_i)$, do inglês *actual finish time*, é o tempo em que a tarefa t_i finalizará a sua execução após ter sido atribuída a algum processador.

Definição 2.4. *Makespan* (ou *scheduling length*) é o tamanho total de um escalonamento de todas as tarefas de uma aplicação e pode ser determinado pelo tempo de finalização da tarefa de saída, conforme mostrado na Equação (2):

$$makespan = \max\{AFT(t_{exit})\} \tag{2}$$

Definição 2.5. Caminho crítico (CP), do inglês *critical path*, é o maior caminho entre a tarefa de entrada até a tarefa de saída. O limite inferior do *makespan* é o tamanho mínimo do caminho crítico ($CP_{\min}$).

Definição 2.6. $EST(t_i, p_j)$, do inglês *earlist execution start time*, é o menor tempo possível para iniciar a execução da tarefa t_i no processador p_j e pode ser computado conforme a Equação (3):

$$EST(t_i, p_j) = \max\left\{ disponível[j], \quad \max_{t_x \in pred(t_i)} \{AFT(t_x) + c_{xi}\} \right\} \tag{3}$$

Nesse caso, $disponível[j]$ representa o menor tempo possível para o processador p_j estar pronto para iniciar execução de t_i. O valor máximo mais interno traz o tempo em que será finalizada a execução de todas as tarefas predecessoras de t_i. Como o processador estará disponível e finalizará a execução dos predecessores requisitos para executar uma tarefa, a Equação (3) trará o menor tempo possível para iniciar a execução. Assim, para tarefas de entrada:

$$EST(t_{entry}, p_j) = 0 \tag{4}$$

Definição 2.7. $EFT(t_i, p_j)$, do inglês *earlist execution finish time*, é o menor tempo possível para finalizar a execução da tarefa t_i no processador p_j. Seu valor é encontrado da seguinte forma (Equação (5)):

$$EFT(t_i, p_j) = EST(t_i, p_j) + w_{ij} \tag{5}$$

Capítulo 3

Desenvolvimento

Nesse capítulo são apresentados os métodos, técnicas, soluções, softwares e tecnologias que foram empregadas para o desenvolvimento deste trabalho. Inicialmente, apresenta-se a linguagem de programação e o ambiente utilizado no desenvolvimento deste trabalho. Em seguida, são descritos os quatro algoritmos que foram implementados. Finalmente, descreve-se o método proposto para gerar os DAGs utilizados como entrada dos algoritmos nos experimentos.

3.1 Linguagem de programação utilizada

Para o desenvolvimento deste trabalho, foi empregada a linguagem de programação Python, na versão 3.7.12. Python é uma linguagem de programação interpretada de alto nível, lançada no ano de 1991 pelo programador Guido van Rossum. Todos os códigos dos algoritmos deste trabalho foram desenvolvidos utilizando Python, pelo motivo de ser uma linguagem já bastante conceituada e uma sintaxe de fácil aprendizado. Além disso, outra motivação é o fato de ser muito rica em bibliotecas disponíveis, o que facilitou muito no desenvolvimento de operações matemáticas e construções de gráficos nos experimentos.Nesse contexto, as bibliotecas Python utilizadas neste trabalho foram:

1. **NumPy:** Foi a principal biblioteca utilizada, pois oferece diversas ferramentas matemáticas para manipulação de vetores e matrizes.

2. **Heapq:** Utilizada para implementar uma fila de prioridade, também conhecida como *Heap Queue* ou *Binary Heap*, uma estrutura de dados que é utilizada por alguns algoritmos.

3. **Operator:** Oferece um conjunto de funções correspondentes aos operadores intrínsecos do Python, sendo utilizada para fazer buscas específicas em vetores.

4. **matplotlib:** Fornece ferramentas para criação de conteúdos visuais animados, estáticos e interativos, importantes para geração dos gráficos apresentados netsa monografia.

5. **pickle:** Traz implementações de protocolos de arquivos binários, possibilitando serializar e de-serializar objetos estruturados do Python, de modo que é bastante útil para carregar e salvar os resultados das execuções dos algoritmos, possibilitando seu uso posterior.

6. **random:** Biblioteca para geração de números pseudo-aleatórios, utilizada em diversos códigos deste trabalho.

Neste trabalho, o desenvolvimento do código foi inteiramente feito utilizando a plataforma *Google Colaboratory*[1], que é um serviço gratuito da Google específico para criação de programas na linguagem de programação Python. A escolha dessa IDE foi motivada por ser um serviço na nuvem e de fácil colaboração.

3.2 Algoritmos selecionados

Após o estudo de algoritmos de escalonamento de tarefas, foi escolhido o ambiente computacional em que o trabalho seria realizado. Esse ambiente consiste em um sistema com um número limitado de processadores heterogêneo e com custo de comunicação entre eles. Em seguida, foram escolhidos quatro algoritmos condizentes com esse ambiente: *Heterogeneous Earliest Finish Time* (TOPCUOGLU; HARIRI; WU, 2002), *Critical Path on a Processor* (TOPCUOGLU; HARIRI; WU, 2002), *Improved Heterogeneous Earliest Finish Time* (ALEBRAHIM; AHMAD, 2016) e *Improved Predict Earliest Finish* (ZHOU et al., 2016).

Esses quatro algoritmos são classificados como métodos heurísticos do tipo *List-based scheduling* (GRAHAM, 1966; SHARMA, 2019), ou seja, todos eles são divididos em duas fases: 1) priorização de tarefas; e 2) seleção de processadores. Todos esses algortimos são inspirados no trabalho original de Graham (1966), que propôs o primeiro algoritmo do tipo *List* e que se tornou uma referência fundamental na área.

Todos os algoritmos trabalhados aqui consideram como entrada um DAG representado as tarefas, um conjunto de processadores e os custos (tanto de processamento quanto de comunicação). Além disso, todos eles utilizam o método *insertion-based scheduling policy*, discutido na Seção 2.2 (Definição 2.1) nos escalonamentos das tarefas. Nas próximas seções, esses algoritmos são descritos em mais detalhes.

[1] Disponível em: <https://colab.research.google.com/>, acesso em 23 mar. 2022.

3.2.1 Heterogeneous Earliest Finish Time

O algoritmo *Heterogeneous Earliest Finish Time* (HEFT) foi proposto originalmente por Topcuoglu, Hariri e Wu (2002) e serviu de base para diversas variações, muitas das quais mantiveram suas principais características. Trata-se de um algoritmo eficiente e que ainda referenciado e utilizado como base de comparação para diversas outras outras heurísticas. Suas duas fases são mostradas no pseudocódigo do Algoritmo 1 e descritas em seguida.

Algoritmo 1: Pseudocódigo do algoritmo *Heterogeneous Earliest Finish Time*.

```
1  Computar a média do custo de computação das tarefas e a média do custo de
     comunicação das arestas;
2  Computar o valor upward rank de cada tarefa, começando pela tarefa de saída;
3  Ordenar a lista de tarefas de forma decrescente com base no valor upward rank;
4  enquanto há tarefas que ainda não foram escalonadas faça
5      Selecione a primeira tarefa t_i da lista de tarefas;
6      para cada processador p_j do conjunto P faça
7          Calcular o valor de EFT(t_i, p_j) usando insertion-based scheduling policy;
8      fim
9      Atribua a tarefa t_i ao processador p_j que tenha o menor valor de EFT para
         essa tarefa e remova t_i da lista de tarefas;
10 fim
```

Fase 1: Priorização de tarefas

Nessa primeira fase, inicialmente, cada tarefa é classificada com base em seu nível de prioridade. Essa classificação é denominada de *upward rank* ($rank_u$), sendo atribuída para uma tarefa t_i seguindo a regra recursiva mostrada na Equação (6). Nessa equação, os valores $\overline{w_i}$ e $\overline{c_{ij}}$ representam, respectivamente, a média do custo computacional de uma tarefa t_i e a média do custo de comunicação da aresta entre duas tarefas t_i e t_j.

$$\begin{cases} rank_u(t_{exit}) = \overline{w_{exit}} \\ rank_u(t_i) = \overline{w_i} + \max_{t_j \in succ(t_i)} \{\overline{c_{ij}} + rank_u(t_j)\} \end{cases} \tag{6}$$

Assim, as tarefas são classificadas de forma ascendente no grafo, começando pela tarefa de saída (t_{exit}), e, dessa forma, $rank_u(t_i)$ é o tamanho do caminho crítico da tarefa t_i até a tarefa de saída. Após a finalização da classificação, a lista de tarefas é ordenada pelo valor *upward rank* de cada tarefa, em ordem decrescente, sendo que, em caso de empate, a escolha é de forma aleatória. Sabendo que o *upward rank* de uma tarefa nunca será maior que seus sucessores, não há risco de uma tarefa predecessora ser classificada após suas sucessoras.

Fase 2: Seleção de processador

Nessa etapa, as tarefas são escalonadas na ordem que estão na lista *upward rank*, seguindo a *insertion-based scheduling policy*. Assim, para cada tarefa t_i busca-se um processador que tenha disponibilidade de executá-la com o menor valor de EFT. Dessa forma, para cada processador p_j será levado em consideração o tempo necessário para que todas as predecessoras dessa tarefa sejam concluídas, acrescido do tempo de comunicação dos dados para esse processador.

Assim, encontra-se o valor de *ready time*, que é o tempo mínimo necessário para iniciar aquela tarefa em p_j. Depois disso, procura-se em p_j uma lacuna de tempo ocioso suficiente para executar a tarefa t_i, lembrando que são consideradas somente as faixas de tempo que estão posteriores ao valor de *ready time*.

3.2.2 Critical Path on a Processor

Assim como o HEFT, o algoritmo *Critical Path on a Processor* (CPOP) também foi apresentado por Topcuoglu, Hariri e Wu (2002). Apesar de ter, segundo os autores, um desempenho comparável ao HEFT, esse algoritmo traz uma política de priorização de tarefas um pouco mais elaborada, motivo pelo qual é menos referenciado que o anterior. Entretanto, considerou-se interessante avaliar o desempenho desse algoritmo frente aos demais, motivo pelo qual foi também escolhido para ser discutido neste trabalho. O CPOP é apresentado, como pseudocódigo, no Algoritmo 2 e descrito logo em seguida.

Fase 1: Priorização de tarefas

Nessa fase, cada tarefa é classificada com base em seus valores de *downward rank* e *upward rank*. A classificação *downward rank* (denotada por $rank_d$) é similar à *upward rank*, explicado no algoritmo anterior, porém seu cálculo é feito de maneira descendente no grafo, começando pela tarefa de entrada (t_{entry}). Assim, o valor de $rank_d$ para uma tarefa t_i é computado, recursivamente, conforme a Equação (7):

$$\begin{cases} rank_d(t_{entry}) = 0 \\ rank_d(t_i) = \max_{t_j \in pred(t_i)} \{\overline{w_j} + \overline{c_{ji}} + rank_d(t_j)\} \end{cases} \tag{7}$$

A prioridade de cada tarefa é a soma dos seus valores de *downward rank* e *upward rank*. Assim, é necessário definir um caminho crítico utilizando o valor das prioridades das tarefas, da seguinte forma: primeiro, seleciona-se a tarefa de entrada como tarefa do caminho crítico; depois seleciona-se entre as sucessoras dessa tarefa a que tem a maior prioridade, e marcando-a também como uma tarefa do caminho crítico. Esse procedimento é repetido até chegar na tarefa de saída, que também será uma tarefa do caminho crítico. Em casos de empates entre as sucessoras, selecione o primeiro que foi encontrado com maior prioridade.

Algoritmo 2: Pseudocódigo do algoritmo *Critical Path on a Processor*.

```
Computar a média dos custos de processamento e de comunicação das tarefas;
Computar o valor upward rank de cada tarefa, começando pela tarefa de saída;
Computar o valor downward rank de cada tarefa, começando pela tarefa de
  entrada;
Calcular prioridade(t_i) = rank_u(t_i) + rank_d(t_i) para cada tarefa t_i;
Atribuir o tamanho do caminho crítico |CP| = prioridade(n_entry);
Iniciar o conjunto das tarefas do caminho crítico C_cp = {t_entry};
t_x = t_entry;
enquanto t_x não é a tarefa de saída faça
    Selecionar t_j tal que ((n_j ∈ succ(t_x)) e (prioridade(t_j) == |CP|));
    C_cp = C_cp ∪ {t_j};
    t_x = t_j;
fim
Escolher o P_cp que minimize Σ_{t_i ∈ C_cp} w_ij,   ∀p_j ∈ P;
Inicializar a fila de prioridade com a tarefa de entrada;
enquanto tiver tarefa não escalonada na fila de prioridade faça
    Selecionar a tarefa t_i com maior prioridade da fila;
    se t_i ∈ C_cp então
        Atribuir t_i no P_cp;
    fim
    senão
        Atribuir t_i para o processador p_j que minimize EFT(t_i, p_j);
    fim
    Inserir na fila de prioridade as tarefas sucessoras de t_i, que estão prontas para
      serem escalonadas;
fim
```

Nota-se que esse algoritmo utiliza uma fila de prioridade contendo todas as tarefas que estão prontas para serem escalonadas. A melhor estrutura para essa fila de prioridade é uma *Binary Heap*, pois o tempo de complexidade para inserção é de $O(\log n)$ e remoção da tarefa de maior prioridade é de $O(1)$.

Fase 2: Seleção de processador

Inicialmente, nessa fase, deve-se encontrar o processador do caminho crítico (P_{cp}), que é o processador que executará em menor tempo todas as tarefas do caminho crítico. Depois disso, as tarefas vão sendo designadas na ordem que aparecem na fila de prioridade, começando pela tarefa de entrada. Para cada tarefa a ser designada, é verificado se ela pertence ao caminho crítico. Em caso afirmativo, ela é atribuída ao processador P_{cp}; caso contrário, ela será atribuída ao processador que permite o menor EFT.

3.2.3 Improved Heterogeneous Earliest Finish Time

O algoritmo *Improved Heterogeneous Earliest Finish Time* (IHEFT) foi proposto por AlEbrahim e Ahmad (2016) a partir de um outro algoritmo descrito por Shetti, Fahmy e Bretschneider (2013). O algoritmo original, porém, não considerava os custos de comunicação entre tarefas, característica que foi incorporada por AlEbrahim e Ahmad (2016) de modo a melhorar a priorização e a seleção dos processadores. Além disso, o IHEFT utiliza um pouco de aleatoriedade em suas decisões, a fim de explorar melhor o espaço das possíveis soluções do problema. Ademais, esse algoritmo foi selecionado para este trabalho por se tratar, dadas as suas características, de um algoritmo que trabalha com a exploração de melhores soluções locais e globais.

Nos artigos citados aqui (SHETTI; FAHMY; BRETSCHNEIDER, 2013; ALEBRAHIM; AHMAD, 2016), esse algoritmo é referenciado apenas como "algoritmo proposto"; porém, por ser um algoritmo proposto para ser uma melhoria do HEFT, ele foi nomeado nesta monografia como IHEFT. Seu pseudocódigo é apresentado no Algoritmo 3 e suas duas fases são descritas em seguida.

Fase 1: Priorização de tarefas

Primeira etapa dessa fase é o cálculo dos pesos de cada tarefa $Peso_{ti}$, que será utilizado na obtenção das prioridades. O peso das tarefas é o valor absoluto da razão da diferença do maior tempo de execução e o menor tempo de execução, definido como:

$$Peso_{ti} = \left| \frac{w_{ij} - w_{ik}}{w_{ij}/w_{ik}} \right| \tag{8}$$

em que w_{ij} representa o maior custo computacional e w_{ik} representa o menor custo computacional da tarefa t_i dentre todos os processadores do ambiente. Com o valor dos pesos de cada tarefa e a média dos custos de comunicação é possível calcular a prioridade de cada tarefa, que nesse algoritmo é nomeada como *proposed rank* ($rank_{proposed}$) e é definida recursivamente da seguinte forma:

$$\begin{cases} rank_{proposed}(t_{exit}) = Peso_{ti} \\ rank_{proposed}(t_i) = Peso_{ti} + \max_{t_j \in succ(t_i)} \{\overline{c_{ij}} + rank_{proposed}(t_j)\} \end{cases} \tag{9}$$

Fase 2: Seleção de processador

O principal diferencial nessa fase desse algoritmo é o uso de uma técnica chamada *non-crossover*, proposta por Shetti, Fahmy e Bretschneider (2013), que o difere do HEFT. Essa técnica não seleciona o processador que produz o menor tempo de finalização para a tarefa (EFT), mas sim o processador que oferece o menor tempo de execução da tarefa (w). Para isso, é calculado um valor chamado *Cross Threshold*, que pode ser entre 0 e 1. Quanto mais perto o valor de 0, maior a chance de ocorrer um *crossover*; assim o

Algoritmo 3: Pseudocódigo do algoritmo *Improved Heterogeneous Earliest Finish Time*.

```
1  Computar o valor de Peso_ti de cada tarefa;
2  Computar a média do custo de comunicação da arestas;
3  Computar o valor proposed rank de cada tarefa, começando pela tarefa de saída;
4  Ordenar a lista de tarefas de forma decrescente com base no valor proposed rank;
5  enquanto houver tarefas que ainda não foram escalonadas na lista faça
6      Selecionar a primeira tarefa t_i da lista de tarefas;
7      para cada processador p_j do conjunto P faça
8          Calcular o valor de EFT(t_i, p_j) usando insertion-based scheduling policy;
9      fim
10     Selecionar p_j que tenho o menor valor EFT(t_i, p_j);
11     se w_ij ≤ min_{p_k∈P} {w_ik} então
12         Atribuir t_i ao processador p_j que minimize EFT(t_i, p_j) e remover t_i da
            lista de tarefas;
13     fim
14     senão
15         Computar o valor de Peso_abstrato da tarefa t_i com os processadores p_j e p_k;
16         Computar o valor de CrossThreshold;
17         Gerar o valor de r aleatoriamente no intervalo [0.1, 0.3];
18         se CrossThreshold ≤ r então
19             Atribuir t_i ao processador p_j que minimize EFT(t_i, p_j) e remover t_i da
                lista de tarefas;
20         fim
21         senão
22             Atribuir t_i para o processador p_k que minimize w_ik e remover t_i da lista
                de tarefas.
23         fim
24     fim
25 fim
```

algoritmo seguiria similarmente ao HEFT. Por outro lado, quanto mais perto do valor de 1, maior a chance de não acontecer o *crossover*. Dessa forma, esse algoritmo combina escalonamentos tanto com menor tempo de execução de uma tarefa (melhor local), como também o menor tempo de finalização de uma tarefa (melhor global).

Focando no funcionamento da fase, são escalonadas as tarefas na ordem que estão na lista de prioridade. Para cada tarefa, primeiro será encontrado qual é o processador com o menor valor de EFT; em seguida verifica-se se o custo computacional da tarefa nesse processador é o menor custo computacional possível entre todos os processadores. Se isso for verdade, a tarefa é atribuída naquele mesmo processador; caso contrario, vai para a validação de *crossover*. Nessa etapa de validação de *crossover*, é calculado o valor de *Cross Threshold*, utilizando os valores de $Peso_{ti}$ e $Peso_{abstrato}$. A fórmula do $Peso_{abstrato}$ é:

$$Peso_{abstrato} = \left| \frac{EFT(t_i, p_j) - EFT(t_i, p_k)}{EFT(t_i, p_j)/EFT(t_i, p_k)} \right| \tag{10}$$

sendo que p_j representa o processador com menor valor EFT e p_k o processador com o menor custo computacional. Assim o cálculo do *Cross Threshold* é dado por:

$$CrossThreshold = \frac{Peso_{ti}}{Peso_{abstrato}} \tag{11}$$

Após esses cálculos, é computado o valor limite r, que é utilizado para comparar com o valor *Cross Threshold* e decidir se ocorrerá o *crossover* ou não. Nesse algoritmo proposto por AlEbrahim e Ahmad (2016), o valor limite r é gerado aleatoriamente entre 0.1 e 0.3, diferente do algoritmo original, cujo valor era fixo (SHETTI; FAHMY; BRETSCHNEIDER, 2013) . Com isso, é verificado se o *Cross Threshold* é menor ou igual a r, em caso positivo, ocorre o *crossover* e a tarefa é atribuída ao processador com menor EFT (melhor global). Caso contrário, não ocorre *crossover* e a tarefa é atribuída ao processador com menor custo computacional (melhor local).

3.2.4 Improved Predict Earliest Finish Time

Outro algoritmo estudado nesta monografia segue uma abordagem conhecida como "preditiva", construindo uma tabela com as possíveis execuções de cada tarefa em cada processador. O algoritmo que seguiu essa abordagem originalmente é denominado *Predict Earliest Finish Time* (PEFT) e foi originalmente proposto por Arabnejad e Barbosa (2014). Posteriormente, Zhou et al. (2016) estenderam esse algoritmo, criando *Improved Predict Earliest Finish Time* (IPEFT), que apresentou resultados melhores (ZHOU et al., 2016), sendo, portanto, implementado neste trabalho.

Para a compreensão do funcionamento deste algoritmo, é necessário definir alguns conceitos além dos já apresentados na Seção 2.2, as quais são apresentadas a seguir.

Definição 3.1. $AEST(n_i)$, do inglês *average earliest start time*, é a média dos menores tempos possíveis para iniciar a execução da tarefa n_i, que pode ser calculada recursivamente descendente no grafo, começando pela tarefa de entrada. Vale notar que seu calculo é idêntico ao *downward rank*, apresentado na Seção 3.2.2:

$$\begin{cases} AEST(t_{entry}) = 0 \\ AEST(t_i) = \max_{t_j \in pred(t_i)} \{\overline{w_j} + \overline{c_{ji}} + AEST(t_j)\} \end{cases} \tag{12}$$

Definição 3.2. $ALST(n_i)$, do inglês *average latest start time*, é a média dos maiores tempos possíveis para iniciar a execução da tarefa t_i. Seu valor é calculado recursivamente ascendente no grafo, começando pela tarefa de saída, da seguinte maneira:

$$\begin{cases} ALST(t_{exit}) = AEST(t_{exit}) \\ ALST(t_i) = \min_{t_j \in succ(t_i)} \{ALST(t_j) - \overline{c_{ij}}\} - \overline{w_i} \end{cases} \tag{13}$$

Definição 3.3. Nó crítico (CN), do inglês *critical node*, é o nó em que os valores de $AEST$ e $ALST$ são iguais.

Definição 3.4. Tabela de custo pessimista (PCT), do inglês *pessimistic cost table*, é uma matriz de dimensões $n \times m$, onde cada elemento $PCT(t_i, p_j)$ é o maior valor do caminho mais longo do sucessor imediato da tarefa t_i até a tarefa de saída, onde a tarefa t_i é escalonada no processador p_j. Os valores de PCT são calculados recursivamente da seguinte maneira:

$$\begin{cases} PCT(t_{exit}, p_j) = 0, \quad \forall p_j \in P \\ PCT(t_i, p_j) = \max_{v_x \in succ(v_i)} \{\max_{p_k \in P} \{PCT(t_x, p_k) + w_{xk} + \overline{c_{ix}}\}\} \end{cases} \tag{14}$$

Na Equação (14), o valor de $\overline{c_{ix}}$ é igual a 0 caso p_k e p_j forem o mesmo processador.

Definição 3.5. Tabela de custo de nó crítico (CNCT), do inglês *critical node cost table*, é uma matriz de dimensões $n \times m$, onde cada elemento $CNCT(t_i, p_j)$ é o valor máximo do caminho mais curto dos sucessores críticos imediatos de t_i até a tarefa de saída, se t_i possuir ao menos um sucessor que é um CN. Caso contrário, é o valor máximo do caminho mais curto entre todos os sucessores imediatos de n_i até a tarefa de saída. O cálculo recursivo de CNCT é dado por:

$$CNCT(t_i, p_j) = \max_{t_x \in succ(t_i)} \left\{ \min_{p_k \in P} \{CNCT(t_x, p_k) + w_{xk} + \overline{c_{ix}}\} \right\} \tag{15}$$

Lembrando que são consideradas apenas as tarefas t_x que são CN, dentro de $succ(t_i)$. Porém, caso não haja nenhuma que seja CN, são consideradas todas as tarefas de $succ(n_i)$.

Definição 3.6. Pai de nó crítico (CNP), do inglês *critical node parent*, uma tarefa t_i é classificada CNP, se t_i não é CN e tem sucessor imediato que seja CN. Caso não cumpra algos dos dois requisitos, a tarefa t_i não é CNP.

Dadas essas definições, o Algoritmo 4 traz o pseudocódigo do IPEFT, cujas etapas de funcionamento são descritas em seguida.

3.2.4.1 Fase 1: Priorização de tarefas

Nessa etapa, são calculados todos os valores da matriz PCT, abrangendo todas as tarefas em todos os processadores. A partir desses valores, calcula-se a prioridade de cada tarefa com base na classificação *PCT rank* (denotada por $rank_{PCT}$), da seguinte maneira:

Algoritmo 4: Pseudocódigo do algoritmo *Improved Predict Earliest Finish Time*.

```
Computar a média do custo de computação das tarefas e a média do custo de comunicação das arestas;
Computar o valor AEST de cada tarefa, começando pela tarefa de entrada;
Computar o valor ALST de cada tarefa, começando pela tarefa de saída;
Computar o valor de CNCT, PCT, CNP, e rank_pct de cada tarefas;
Ordenar a lista de tarefas de forma decrescente com base no valor PCT rank;
enquanto houver tarefas que ainda não foram escalonadas na lista faça
    Selecionar a primeira tarefa t_i da lista de tarefas;
    para cada processador p_j do conjunto P faça
        Calcular o valor de EFT(n_i, p_j) usando insertion-based scheduling policy;
        se CNP(t_i) == true então
            Calcular EFT_CNCT(t_i, p_j) = EFT(t_i, p_j);
        fim
        senão
            Calcular EFT_CNCT(t_i, p_j) = EFT(t_i, p_j) + CNCT(t_i, p_j);
        fim
    fim
    Atribuir t_i ao processador p_j que minimize EFT_CNCT(t_i, p_j) e remover t_i da lista de tarefas;
fim
```

$$rank_{PCT}(t_i) = \frac{\sum_{j=1}^{|P|} PCT(t_i, p_j)}{p} + \overline{w_i} \tag{16}$$

Assim, quanto maior o $rank_{PCT}$ de uma tarefa, maior sua prioridade de escalonamento.

3.2.4.2 Fase 2: Seleção de processador

Nessa fase, as tarefas vão sendo escalonada seguindo suas classificações do $rank_{PCT}$ e conforme ficam prontas para serem escalonadas. Para cada tarefa é computado o seu valor de EFT em todos os processadores; em seguida é calculado seu valor de EFT com base na $CNCT$ (denotado EFT_{CNCT}), também para todos os processadores. Para encontrar o valor de EFT_{CNCT}, primeiro verifica se a tarefa é uma CNP. Em caso afirmativo, o valor de EFT_{CNCT} é igual o valor EFT; caso contrário esse valor é dado por:

$$EFT_{CNCT}(t_i, p_j) = EFT(t_i, p_j) + CNCT(t_i, p_j) \tag{17}$$

Desse modo, o processador com o menor valor de EFT_{CNCT} é selecionado para a tarefa.

3.3 Geração dos DAGs

Como já citado, a entrada para os quatro algoritmos estudados aqui é um grafo acíclico orientado (DAG), que representa uma aplicação computacional com suas devidas tarefas e custos. Porém neste trabalho, houve dificuldade para encontrar grafos que encaixavam no ambiente computacional de interesse, mais especificamente pelo fato de trabalhar com processadores heterogêneos e custos de comunicação. Devido a esse problema, foi decidido desenvolver um algoritmo capaz de transformar grafos para processadores homogêneos e sem custo de comunicação (GHO) em grafos para processadores heterogêneos e com custos de comunicação (GHE).

Para isso, foram utilizados DAGs disponíveis na *Standard Task Graph* (STG), uma biblioteca mantida pela Waseda University (Japão) e que possui diversos DAGs para o problema de escalonamento em processadores homogêneos[2]. Nesse caso, inclusive, tem-se o valor ótimo (ou aproximações do ótimo) para diversos grafos disponibilizados. O formato desses DAGs e as modificações propostas são descritos a seguir.

3.3.1 Standard Task Graph

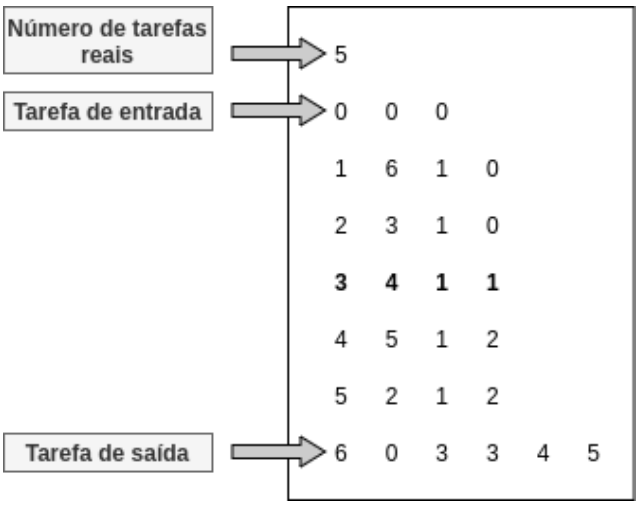
O principal formato utilizado para grafos GHO é o *Standard Task Graph* (STG), que consiste em um arquivo com informações separadas por linhas e colunas. O padrão STG contém sempre grafos com uma única tarefa de entrada e uma única tarefa de saída, as quais servem para a aplicação sempre iniciar e terminar em uma tarefa. Assim, como essas duas tarefas são apenas para representar o inicio e o fim da aplicação, seus custos de computação são nulos e não são consideradas tarefas verdadeiras. A Figura 2 mostra um exemplo de DAG no formato STG.

A primeira linha de um arquivo no padrão STG (Figura 2), representá o número de tarefas "verdadeiras" (desconsiderando a tarefa de entrada e saída). Da segunda linha a diante, é representada uma tarefa por linha, sendo que todas essas linhas são iniciada com o número de identificação dessa tarefa (de 0, para a tarefa de entrada, a $n+1$, para a tarefa de saída). O próximo número de cada linha é o custo ou tempo de processamento da tarefa. Depois, na mesma linha, tem-se o número de predecessores da tarefa, seguido pelo número de identificação dos predecessores.

No exemplo da Figura 2, a primeira linha mostra que se trata de um grafo com cinco tarefas verdadeiras. Sendo assim, serão sete linhas representando tarefas (contando a tarefa de entrada e saída). Utilizando como exemplo a quinta linha, trata-se de uma tarefa com o número de identificação 3, custo computacional igual a 4 e apenas uma tarefa sendo sua predecessora, nesse caso a tarefa de número 1.

[2] Disponível em <http://www.kasahara.cs.waseda.ac.jp/schedule/>, acesso em 23 mar. 2022.

Figura 2 – Exemplo de DAG no formato STG.

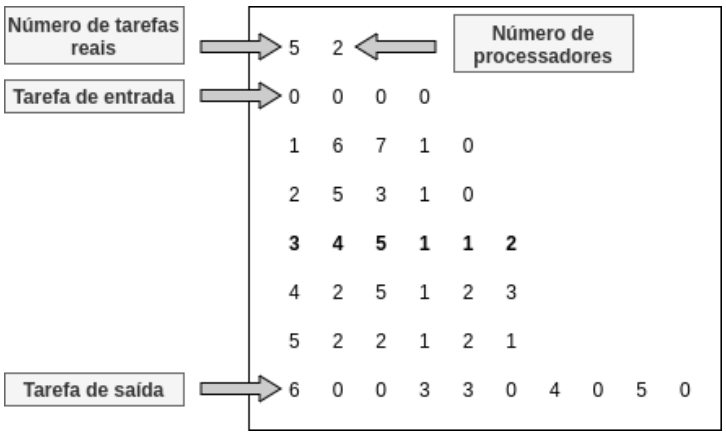


Fonte: autoria própria.

3.3.2 Padrão proposto

Como mencionado, o padrão original do STG não contém informações suficientes para um ambiente heterogêneo que considera custo de comunicação. Assim foi criado, neste trabalho, uma extensão do STG para comportar essas informações, mantendo boa parte da organização e legibilidade do arquivo. A Figura 3 mostra um exemplo de DAG gerado nesse novo formato.

Figura 3 – Padrão de DAG proposto neste trabalho.



Fonte: autoria própria.

A primeira modificação necessária nesse novo padrão é a necessidade de colocar os custo computacionais de cada tarefa para cada processador. Assim, na primeira linha logo após o número de tarefas verdadeiras, há o número de processadores. Além disso, em vez de ter apenas uma coluna em cada tarefa com seu custo de computação, há uma coluna

para cada processador. A segunda modificação é a necessidade de colocar o tamanho dos dados necessários para comunicação entre tarefas. Com isso, na frente de cada número do predecessor será colocado um número de dados necessários para serem transferidos, correspondente a esse predecessor.

Por exemplo, no grafo da Figura 3, na primeira linha logo após o número de tarefas verdadeira, encontra-se o número 2, que representa que esse grafo contém custos computacionais para dois processadores diferentes. Utilizando a quinta linha como exemplo, referente a uma tarefa com o número de identificação 3, tem-se um custo computacional igual a 4 para o primeiro processador e igual a 5 para o segundo processador; ainda, apenas uma tarefa é a sua predecessora (nesse caso, a tarefa de número 1) e trocando um volume de dados igual a 2 nessa dependência. Nota-se que todas as dependências da tarefa de saída contém valor de dados nulo, por se tratar de uma tarefa que não é verdadeira, e, portanto, a transferência é imediata.

3.3.3 Algoritmo de conversão

A execução do algoritmo de conversão de grafo STG para o padrão proposto recebe algumas informação de entrada. Esses dados necessários são o DAG no formato STG, o número de processadores que o grafo irá comportar, a variação do custo computacional (denotada por $\Delta_{compCost}$) e variação do tamanho de dados de comunicação ($\Delta_{commData}$).

Conforme o algoritmo lê o grafo STG, ele também escreve o arquivo no novo padrão. Para cada linha de tarefa t_i, ele gera aleatoriamente o valor de custo computacional para cada processador p_j com base no custo computacional do grafo STG ($wSTG_i$) e a variação do custo computacional, seguindo a fórmula:

$$w_{i,j} = rand[wSTG_i, \quad wSTG_i + \Delta_{compCost}] \tag{18}$$

Sendo $rand[x, y]$ uma função que retorna um valor aleatório entre x e y. Caso for uma tarefa de entrada ou saída manterá o custo computacional nulo em todos os processadores. Além disso, para cada linha de tarefa t_k, é adicionado o volume de dados de comunicação em cada predecessor t_i, com base na variação do tamanho de dados de comunicação, como mostrado a seguir:

$$dados_{i,k} = rand[1, \quad \Delta_{commData}] \tag{19}$$

Se a sucessora t_k for a tarefa de saída ou se a predecessora t_i for a de entrada, o tamanho dos dados de comunicação é nulo. Quando todas as linhas de tarefas são tratadas o arquivo com padrão novo é concluído e pronto para ser utilizado nos algoritmos de escalonamento.

Capítulo 4

Experimentos

Nesse capítulo, ser apresentada as métricas de comparação utilizadas, como foram criados os ambientes para a execução dos experimentos, a forma que os experimentos foram desenvolvidos, os resultados obtidos e, por último, uma discussão sobre esses resultados. Todo o desenvolvimento e execuções dos algoritmos, foram feitos utilizando a plataforma *Google Colaboratory*.

4.1 Métricas de comparação

Para comparar os quatro algoritmos considerados neste trabalho, é necessário utilizar alguma métrica de desempenho. Aqui, optou-se por empregar duas métricas bastante utilizadas na literatura e descritas a seguir:

1. **Makespan:** Dentro dos estudos de escalonamento de tarefas, o *makespan* é a principal medida de comparação, pois, em linhas gerais, consegue mostrar qual algoritmo alcança um escalonamento mais eficiente. Assim, neste trabalho, o *makespan* foi utilizados também como o principal critério de avaliação;

2. **Load Balance:** Diferente do *makespan*, o *load balance* (ou "balanceamento de carga") não é utilizado com tanta frequência, mas ainda gera resultados interessantes, pois ele mostra o quanto os processadores ficarão ocupados ou ociosos no escalonamento. O seu cálculo é dado da seguinte maneira:

   - $|P|$ = número total de processadores
   - $execTempo(p_j)$ = tempo de execução do processador p_j
   - $media = \frac{\sum_{j=1}^{|P|} execTempo(p_j)}{|P|}$
   - $loadBalance = makespan/media$

Quanto menor o valor de *load balance*, mais os processadores se mantém ocupados. Analogamente, quanto maior o valor, maior é ociosidade do sistema.

4.2 Geração dos ambientes de execução

A principal objetivo dos experimentos neste trabalho é explorar os algoritmos em vários ambientes de execução, a fim de encontrar vantagens e desvantagens deles em relação a suas características nos mais diversos cenários. A primeira etapa para geração dos ambientes de execução foi obter os grafos GHO, no padrão STG. Para isso, foram utilizados de grafos do site Kasahara Laboratory da Waseda University. Nesse site (até o momento de desenvolvimento desta monografia), encontram-se bibliotecas de grafos que variam em número de tarefas de 50 até 5000. Na Figura 4, é mostrado um exemplo de como esses grafos são organizados no site. Cada arquivo compactado possui 80 diferentes DAGs.

Figura 4 – DAGs disponíveis no Kasahara Laboratory.

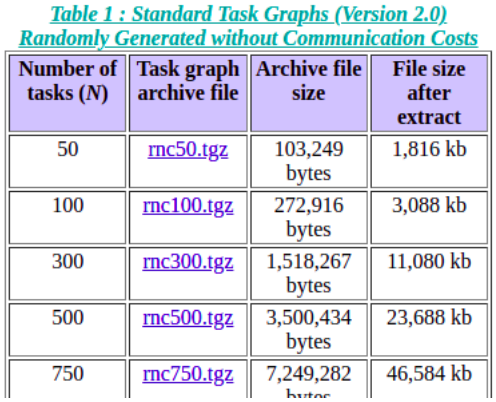


Table 1 : Standard Task Graphs (Version 2.0)
Randomly Generated without Communication Costs

Number of tasks (*N*)	Task graph archive file	Archive file size	File size after extract
50	rnc50.tgz	103,249 bytes	1,816 kb
100	rnc100.tgz	272,916 bytes	3,088 kb
300	rnc300.tgz	1,518,267 bytes	11,080 kb
500	rnc500.tgz	3,500,434 bytes	23,688 kb
750	rnc750.tgz	7,249,282 bytes	46,584 kb

Fonte: Site oficial do Kasahara Laboratory.

Todos os DAGs disponibilizados são do tipo GHO, portanto, foram convertidos para o padrão apresentado, conforme mostrado na Seção 3.3.2. Foram variados quatro parâmetros: 1) número de processadores; 2) número de tarefas; 3) custo de computação; e 4) custo de comunicação. Dessa forma, diversos DAGs foram gerados para criar diferentes tipos de ambientes. Nas próximas seções, são descritos os parâmetros variados e os resultados em cada cenário.

4.2.1 Cenário 1: Alta variação no custo de comunicação

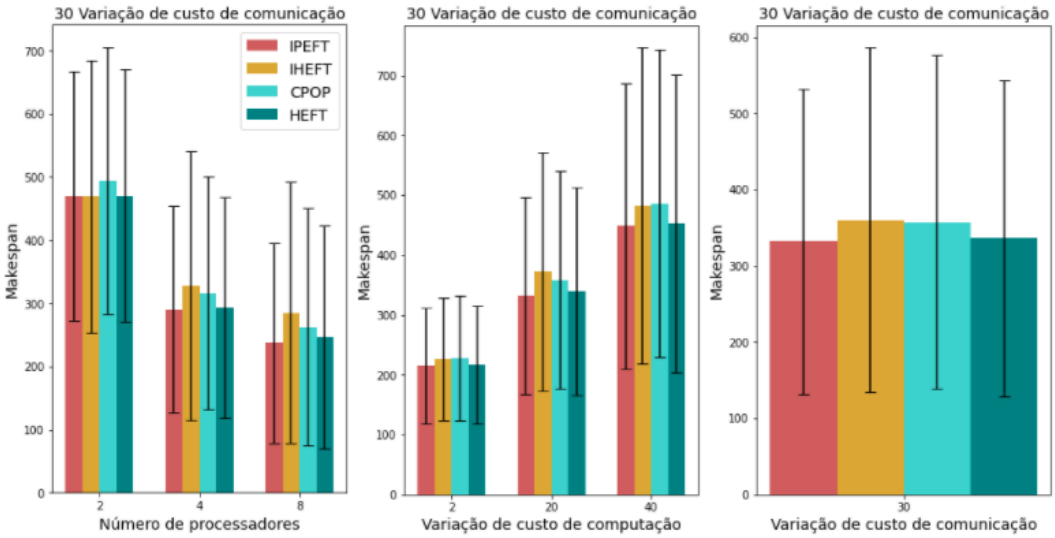
Nesse primeiro cenário de experimentos, a variação de custo de comunicação permanece fixa em 30, que significa que todos os custos de comunicações, foram gerado aleatoriamente no intervalo entre 1 e 30 (seguindo o modelo descrito na Seção 3.3.2). Assim, é

possível avaliar o comportamento dos quatro algoritmos em cenários com grande variação de comunicação. Os parâmetros utilizados nos experimentos foram os seguintes:

- ❏ Número de tarefas: 50
- ❏ Número de processadores: 2, 4 e 8
- ❏ Variação de custo computacional: 2, 20 e 40
- ❏ Variação de custo de comunicação: 30
- ❏ Total de grafos: 324

Nos gráficos da Figura 5, em relação a número de processadores, nota-se que, utilizando apenas **dois** processadores, o desempenho dos algoritmos é muito similar, exceto pelo CPOP que demonstra resultados inferiores. Conforme o número de processadores aumenta, o desempenho do IHEFT vai piorando em relação aos outros algoritmos e o desempenho do IPEFT mantém-se como o melhor, porém com resultados cada vez mais expressivos em relação aos outros.

Figura 5 – Média e desvio-padrão do makespan para os algoritmos IPEFT, IHEFT, CPOP e HEFT com grafos de 50 tarefas com alto custo de comunicação.



Fonte: autoria própria.

Já em relação a variação de custo de computação, pode-se observar os resultados bem piores do IHEFT quando o custo de computação encontra-se próximo a 20. Além disso, foi mantida a predominância dos melhores resultados no IPEFT. Olhando para o gráfico geral, nota-se os melhores resultados no IPEFT, em seguida o HEFT, enquanto os resultados do IHEFT e CPOP foram piores e bem próximos. Na Tabela 1, são sumarizados os valores obtidos na execução dos quatro algoritmos.

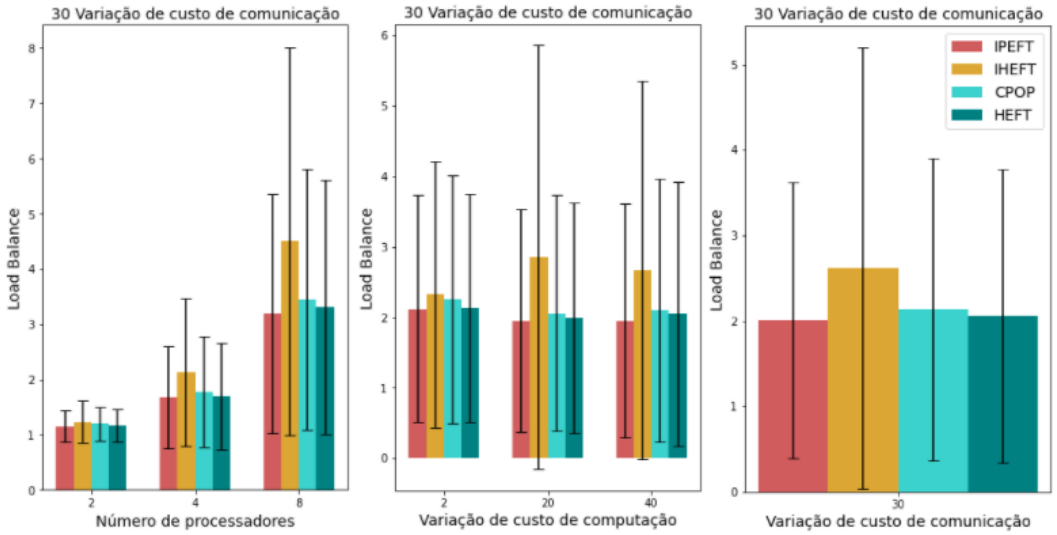
Tabela 1 – Média e desvio-padrão do makespan para os algoritmos IPEFT, IHEFT, CPOP e HEFT com grafos de 50 tarefas com alto custo de comunicação.

$\|P\|$	IPEFT	IHEFT	CPOP	HEFT
2	469.3 ± 197.6	468.8 ± 214.8	493.8 ± 211.2	470.2 ± 199.4
4	290.1 ± 163.6	328.2 ± 212.9	316.1 ± 184.3	292.7 ± 174.4
8	237.1 ± 158.1	284.5 ± 206.6	262.5 ± 188.0	246.4 ± 176.0
$\Delta_{compCost}$	IPEFT	IHEFT	CPOP	HEFT
2	215.5 ± 97.2	226.0 ± 102.1	227.6 ± 104.1	217.1 ± 97.9
20	332.5 ± 164.1	373.0 ± 199.1	358.5 ± 181.6	339.3 ± 173.4
40	448.5 ± 238.5	482.6 ± 263.5	486.4 ± 256.8	452.9 ± 248.2
$\Delta_{commData}$	IPEFT	IHEFT	CPOP	HEFT
30	332.2 ± 200.3	360.5 ± 225.6	357.5 ± 218.5	336.4 ± 207.4

Fonte: autoria própria.

Já nos gráficos da Figura 6, o principal destaque é o IHEFT contendo níveis consideravelmente alto de ociosidade quando executado em **oito** processadores e com valores de variações de custo de computação igual a 20 e 40. Além disso, o desvio padrão do IHEFT para variação de custo de computação igual a 20 é maior o valor de *load balance* obtido. Dessa forma, no gráfico geral, sua média e desvio padrão de *load balance* são maiores quando comparados aos demais algoritmos. Porém, entre os outros algoritmos encontra-se com bem valores próximos, mas o IPEFT obtém o menor valor em todas as comparações desses gráficos. Os valores obtidos estão sumarizados na Tabela 2.

Figura 6 – Média e desvio-padrão do load balance para os algoritmos IPEFT, IHEFT, CPOP e HEFT com grafos de 50 tarefas com alto custo de comunicação.



Fonte: autoria própria.

Tabela 2 – Média e desvio-padrão do load balance para os algoritmos IPEFT, IHEFT, CPOP e HEFT com grafos de 50 tarefas com alto custo de comunicação.

$\|P\|$	IPEFT	IHEFT	CPOP	HEFT
2	1.155 ± 0.286	1.228 ± 0.381	1.197 ± 0.303	1.156 ± 0.295
4	1.668 ± 0.924	2.120 ± 1.334	1.776 ± 1.000	1.697 ± 0.964
8	3.196 ± 2.160	4.496 ± 3.508	3.437 ± 2.358	3.311 ± 2.296
$\Delta_{compCost}$	IPEFT	IHEFT	CPOP	HEFT
2	2.121 ± 1.614	2.325 ± 1.890	2.251 ± 1.758	2.126 ± 1.617
20	1.949 ± 1.578	2.849 ± 3.004	2.061 ± 1.668	1.989 ± 1.634
40	1.949 ± 1.655	2.670 ± 2.683	2.098 ± 1.859	2.050 ± 1.873
$\Delta_{commData}$	IPEFT	IHEFT	CPOP	HEFT
30	2.006 ± 1.618	2.615 ± 2.578	2.137 ± 1.766	2.055 ± 1.713

Fonte: autoria própria.

4.2.2 Cenário 2: Variação baixa de custo de comunicação

Nesse segundo cenário, o custo de comunicação foi gerado aleatoriamente no intervalo entre 1 e 3, ou seja, tem uma baixa variação. Os demais parâmetros permanecem iguais:

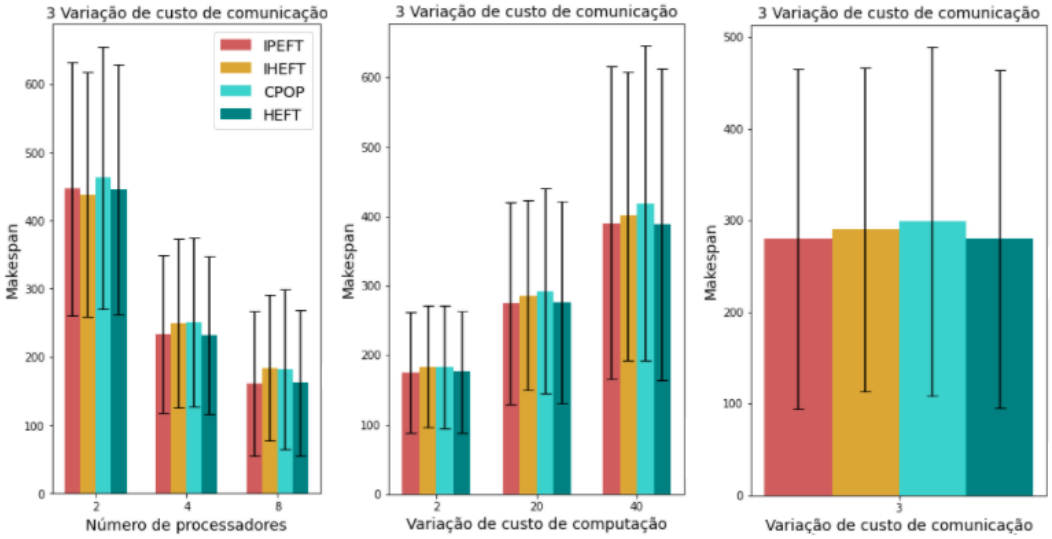
- ❏ Número de tarefas: 50
- ❏ Número de processadores: 2, 4 e 8
- ❏ Variação de custo computacional: 2, 20 e 40
- ❏ Variação de custo de comunicação: 3
- ❏ Total de grafos: 324

Nos gráficos da Figura 7, primeiramente em relação a número de processadores, é observado um resultado muito bom do IHEFT para **dois** processadores, a ponto de ser o melhor entre os algoritmos. Porém, esse resultado vai piorando conforme aumenta-se o número de processadores, até se tornar o pior algoritmo para **oito** processadores. Já os outros algoritmos mantém um padrão nos resultados, com o IPEFT e HEFT próximos e o CPOP com resultados piores.

Sobre o gráfico da variação de custo de computação, é vista uma perda crescente no desempenho do CPOP conforme a variação é aumentada. O IPEFT e HEFT mantêm-se como melhores, variando por diferenças muito pequenas. Com isso, o gráfico geral mostra o HEFT sendo o melhor, por uma distância bem pequena do IPEFT, em seguida o IHEFT e por último o CPOP contendo predominância nos piores resultados nesse ambiente. Os valores obtidos são apresentados na Tabela 3.

Nos gráficos da Figura 8, o IHEFT predomina em todas as classificações como o maior valor de *load balance*, demonstrando resultados altos em **oito** processadores. Já o IPEFT e HEFT encontram-se com resultados muito próximos, porém o IPEFT fica com o menor

Figura 7 – Média e desvio-padrão do makespan para os algoritmos IPEFT, IHEFT, CPOP e HEFT com grafos de 50 tarefas com baixo custo de comunicação.



Fonte: autoria própria.

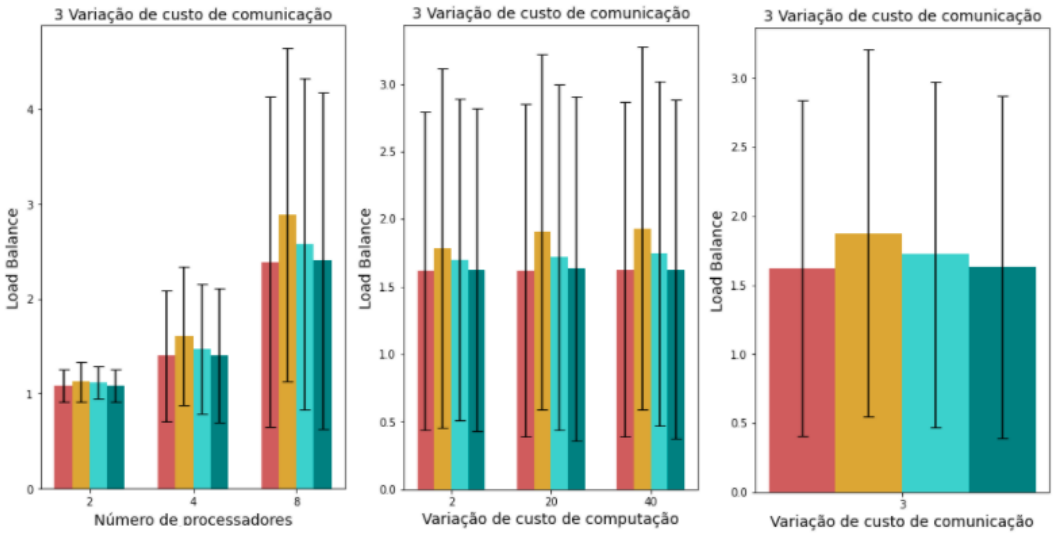
Tabela 3 – Média e desvio-padrão do makespan para os algoritmos IPEFT, IHEFT, CPOP e HEFT com grafos de 50 tarefas com baixo custo de comunicação.

$\|P\|$	IPEFT	IHEFT	CPOP	HEFT
2	446.4 ± 185.3	437.3 ± 178.9	462.2 ± 191.9	445.3 ± 183.1
4	232.7 ± 115.6	249.1 ± 123.5	251.0 ± 123.9	232.2 ± 115.3
8	161.1 ± 105.7	184.0 ± 106.8	181.8 ± 117.0	162.3 ± 106.8
$\Delta_{compCost}$	IPEFT	IHEFT	CPOP	HEFT
2	175.1 ± 87.4	183.6 ± 88.0	183.3 ± 88.1	176.1 ± 87.7
20	274.7 ± 145.8	286.0 ± 136.8	292.7 ± 147.7	275.8 ± 145.5
40	390.3 ± 225.5	400.8 ± 207.9	419.1 ± 226.1	387.9 ± 224.5
$\Delta_{commData}$	IPEFT	IHEFT	CPOP	HEFT
30	280.1 ± 185.2	290.1 ± 176.3	298.4 ± 190.2	279.9 ± 184.1

Fonte: autoria própria.

resultado geral. Além disso, nota-se que os resultados na variação de custo de computação são parecidos para os três valores. Esses valores são mostrados na Tabela 4.

Figura 8 – Média e desvio-padrão do load balance para os algoritmos IPEFT, IHEFT, CPOP e HEFT com grafos de 50 tarefas com baixo custo de comunicação.



Fonte: autoria própria.

Tabela 4 – Média e desvio-padrão do load balance para os algoritmos IPEFT, IHEFT, CPOP e HEFT com grafos de 50 tarefas com baixo custo de comunicação.

$\|P\|$	IPEFT	IHEFT	CPOP	HEFT
2	1.084 ± 0.175	1.126 ± 0.210	1.117 ± 0.169	1.083 ± 0.173
4	1.399 ± 0.690	1.611 ± 0.727	1.470 ± 0.685	1.403 ± 0.705
8	2.385 ± 1.736	2.886 ± 1.752	2.578 ± 1.739	2.402 ± 1.772
$\Delta_{compCost}$	IPEFT	IHEFT	CPOP	HEFT
2	1.618 ± 1.179	1.786 ± 1.328	1.699 ± 1.191	1.627 ± 1.194
20	1.620 ± 1.231	1.904 ± 1.313	1.720 ± 1.279	1.631 ± 1.273
40	1.629 ± 1.239	1.933 ± 1.339	1.745 ± 1.275	1.629 ± 1.253
$\Delta_{commData}$	IPEFT	IHEFT	CPOP	HEFT
30	1.623 ± 1.217	1.874 ± 1.329	1.722 ± 1.249	1.629 ± 1.240

Fonte: autoria própria.

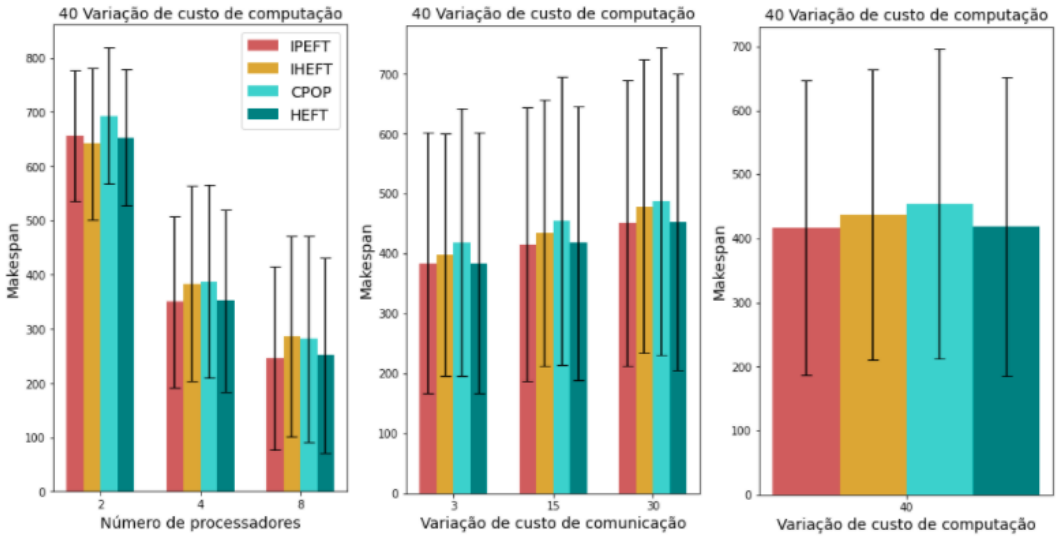
4.2.3 Cenário 3: Variação alta de custo computacional

Neste terceiro cenário, a variação do custo computacional das tarefas foi mantida em 40, ou seja, $rand[wSTG_i, \quad wSTG_i + \Delta_{compCost}]$, sendo $compCost$ igual a 40, seguindo o modelo mostrado na Seção 3.3.2. Os parâmetros utilizados para a criação desse ambiente foram:

- ❏ Número de tarefas: 50
- ❏ Número de processadores: 2, 4 e 8
- ❏ Variação de custo computacional: 40
- ❏ Variação de custo de comunicação: 3, 15 e 30
- ❏ Total de grafos: 324

Nos gráficos da Figura 9, é notado, em relação a número de processadores, um resultado muito bom do IHEFT para dois processadores, a ponto de ser o melhor algoritmo. Entretanto, esse resultado vai piorando conforme é aumentado o número de processadores, tornando-se o segundo pior com quatro processadores e o pior em oito processadores. Porém, os outros algoritmos mantêm um padrão nos resultados, com o IPEFT e HEFT próximos e o CPOP com resultados piores.

Figura 9 – Média e desvio-padrão do makespan para os algoritmos IPEFT, IHEFT, CPOP e HEFT com grafos de 50 tarefas com alto custo computacional.



Fonte: autoria própria.

No gráfico de variação de custo de computação, o desempenho dos quatro algoritmos mantêm um padrão semelhante nos três valores de variação. Onde o IPEFT e HEFT encontram-se com resultados quase que idênticos em variação igual a 3, e nas outras variações, ainda próximos porém o IPEFT sendo o melhor, o IHEFT em terceiro e o CPOP em quarto. Assim, no gráfico geral, o IPEFT obtém os melhores resultados, com o HEFT bem próximo no segundo lugar, em seguida o IHEFT e por último o CPOP com predominância nos piores resultados desse ambiente. Os valores obtidos são mostrados na Tabela 7.

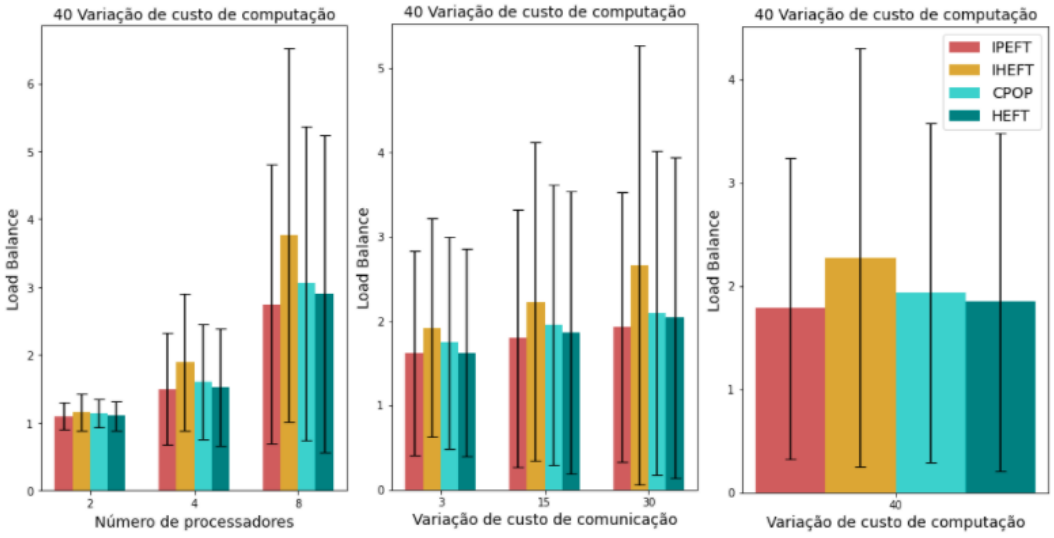
Tabela 5 – Média e desvio-padrão do makespan para os algoritmos IPEFT, IHEFT, CPOP e HEFT com grafos de 50 tarefas com alto custo computacional.

$\|P\|$	IPEFT	IHEFT	CPOP	HEFT
2	655.8 ± 120.3	641.3 ± 140.2	693.2 ± 126.1	652.8 ± 125.3
4	349.7 ± 157.1	383.4 ± 179.5	387.2 ± 177.4	351.5 ± 167.4
8	246.2 ± 168.6	286.4 ± 185.3	281.3 ± 190.2	251.2 ± 179.3
$\Delta_{compCost}$	IPEFT	IHEFT	CPOP	HEFT
2	384.7 ± 217.5	398.3 ± 202.3	419.1 ± 222.5	384.6 ± 217.4
20	415.7 ± 228.2	434.1 ± 222.2	454.9 ± 240.1	417.7 ± 228.7
40	451.3 ± 238.1	478.8 ± 244.9	487.6 ± 256.0	453.3 ± 247.6
$\Delta_{commData}$	IPEFT	IHEFT	CPOP	HEFT
30	417.2 ± 229.7	437.1 ± 226.2	453.9 ± 241.6	418.5 ± 233.3

Fonte: autoria própria.

Nos gráficos da Figura 10, comparando entre os algoritmos, é demonstrado um aumento significativo do IHEFT e uma leve diminuição do IPEFT, conforme é aumentado o número de processadores ou o valor de variação de custo de comunicação. Assim, em todos as comparações desses gráficos, o IPEFT ficou com os menores valores e o IHEFT com os maiores valores. Sendo o HEFT bem próximo do IPEFT, e depois o CPOP. Os valores obtidos são sumarizados na Tabela 2.

Figura 10 – Média e desvio-padrão do load balance para os algoritmos IPEFT, IHEFT, CPOP e HEFT com grafos de 50 tarefas com alto custo computacional.



Fonte: autoria própria.

Tabela 6 – Média e desvio-padrão do load balance para os algoritmos IPEFT, IHEFT, CPOP e HEFT com grafos de 50 tarefas com alto custo computacional.

$\|P\|$	IPEFT	IHEFT	CPOP	HEFT
2	1.101 ± 0.204	1.162 ± 0.274	1.146 ± 0.206	1.101 ± 0.211
4	1.501 ± 0.821	1.889 ± 1.010	1.603 ± 0.850	1.530 ± 0.866
8	2.743 ± 2.056	3.765 ± 2.757	3.057 ± 2.312	2.903 ± 2.337
$\Delta_{compCost}$	IPEFT	IHEFT	CPOP	HEFT
2	1.619 ± 1.213	1.924 ± 1.291	1.749 ± 1.255	1.625 ± 1.227
20	1.796 ± 1.522	2.233 ± 1.893	1.956 ± 1.667	1.868 ± 1.676
40	1.930 ± 1.603	2.659 ± 2.603	2.102 ± 1.919	2.040 ± 1.903
$\Delta_{commData}$	IPEFT	IHEFT	CPOP	HEFT
30	1.782 ± 1.461	2.272 ± 2.025	1.935 ± 1.643	1.845 ± 1.636

Fonte: autoria própria.

4.2.4 Cenário 4: Variação baixa de custo computacional

Finalmente, no quarto cenário, o valor de variação do custo computacional foi mantido em 2, representando, dessa forma, um ambiente com custos computacionais mais reduzidos e próximos. Os demais parâmetros são mostrado a seguir:

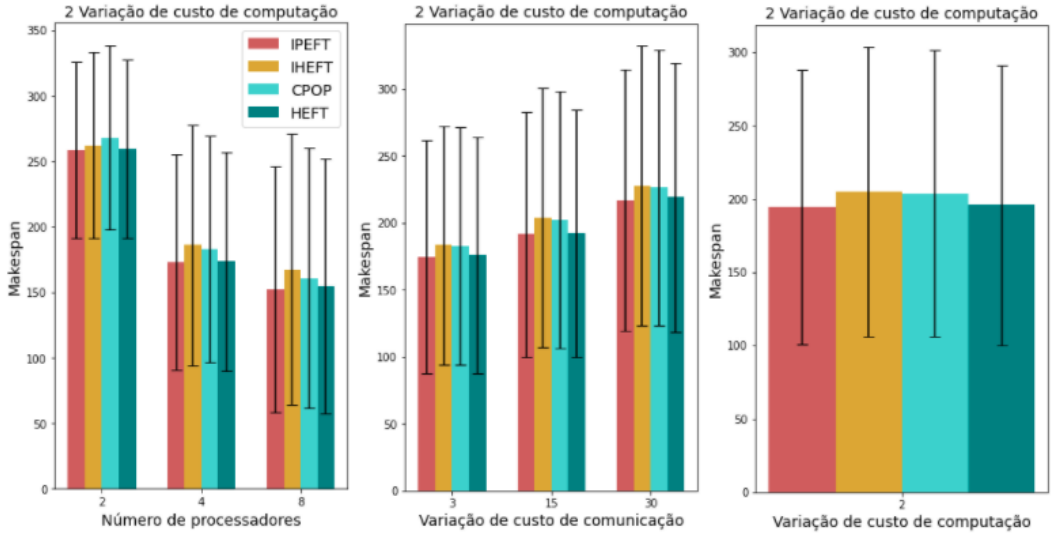
- Número de tarefas: 50
- Número de processadores: 2, 4 e 8
- Variação de custo computacional: 2
- Variação de custo de comunicação: 3, 15 e 30
- Total de grafos: 324

Nos gráficos da Figura 11, é perceptível, em relação a número de processadores, uma piora nos resultado do IHEFT, onde ele permanece no terceiro lugar para dois processadores e em último para quatro e oito processadores.

No gráfico de variação de custo de computação, os quatro algoritmos seguem um padrão, onde o IPEFT e HEFT têm resultados próximos e melhores que os outros algoritmos, porém o IPEFT encontra-se melhor que o HEFT em todos os casos, e entre o CPOP e o IHEFT também observou-se resultados próximos, com o CPOP sendo o melhor que o IHEFT nas três variações. Assim, o gráfico geral não se diferencia muito dos outros gráficos, com o IPEFT em primeiro, depois o HEFT, CPOP e por último o IHEFT. Os valores obtidos são mostrados, também, na Tabela 7.

Nos gráficos da Figura 12, todos os resultados contêm a seguinte sequência crescente, sendo o IPEFT em primeiro, HEFT em segundo, CPOP em terceiro e o IHEFT em último. Além disso, é notável uma crescente nos resultados conforme se aumenta o valor da variação de custo de comunicação. Os valores estão sumarizados na Tabela 8.

Figura 11 – Média e desvio-padrão do makespan para os algoritmos IPEFT, IHEFT, CPOP e HEFT com grafos de 50 tarefas com baixo custo computacional.



Fonte: autoria própria.

Tabela 7 – Média e desvio-padrão do makespan para os algoritmos IPEFT, IHEFT, CPOP e HEFT com grafos de 50 tarefas com baixo custo computacional.

$\|P\|$	IPEFT	IHEFT	CPOP	HEFT
2	258.3 ± 67.3	261.8 ± 70.8	267.8 ± 70.3	259.4 ± 68.1
4	172.7 ± 82.2	185.7 ± 91.9	182.9 ± 86.5	173.5 ± 83.5
8	152.2 ± 93.5	167.4 ± 103.0	160.7 ± 98.9	154.4 ± 97.1
$\Delta_{compCost}$	IPEFT	IHEFT	CPOP	HEFT
2	174.7 ± 87.1	183.2 ± 88.8	182.9 ± 88.3	176.1 ± 88.1
20	191.4 ± 91.3	203.9 ± 96.6	202.2 ± 96.1	192.2 ± 92.4
40	217.1 ± 97.7	227.8 ± 104.3	226.4 ± 102.9	219.1 ± 100.3
$\Delta_{commData}$	IPEFT	IHEFT	CPOP	HEFT
30	194.4 ± 93.8	205.0 ± 98.5	203.8 ± 97.6	195.8 ± 95.4

Fonte: autoria própria.

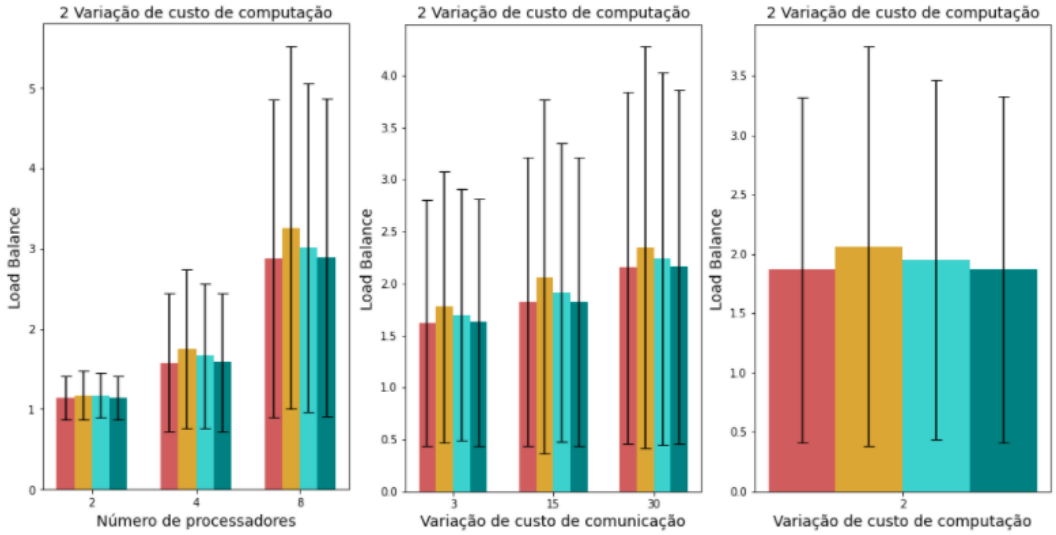


Figura 12 – Média e desvio-padrão do load balance para os algoritmos IPEFT, IHEFT, CPOP e HEFT com grafos de 50 tarefas com baixo custo computacional.

Fonte: autoria própria.

Tabela 8 – Média e desvio-padrão do load balance para os algoritmos IPEFT, IHEFT, CPOP e HEFT com grafos de 50 tarefas com baixo custo computacional.

$\|P\|$	IPEFT	IHEFT	CPOP	HEFT
2	1.137 ± 0.267	1.174 ± 0.308	1.174 ± 0.275	1.141 ± 0.275
4	1.579 ± 0.858	1.750 ± 0.992	1.665 ± 0.898	1.583 ± 0.861
8	2.875 ± 1.971	3.258 ± 2.256	3.011 ± 2.044	2.889 ± 1.977
$\Delta_{compCost}$	IPEFT	IHEFT	CPOP	HEFT
2	1.617 ± 1.183	1.776 ± 1.304	1.699 ± 1.207	1.626 ± 1.191
20	1.824 ± 1.388	2.063 ± 1.700	1.917 ± 1.436	1.823 ± 1.385
40	2.151 ± 1.688	2.344 ± 1.931	2.235 ± 1.790	2.164 ± 1.701
$\Delta_{commData}$	IPEFT	IHEFT	CPOP	HEFT
30	1.864 ± 1.452	2.061 ± 1.682	1.950 ± 1.513	1.871 ± 1.458

Fonte: autoria própria.

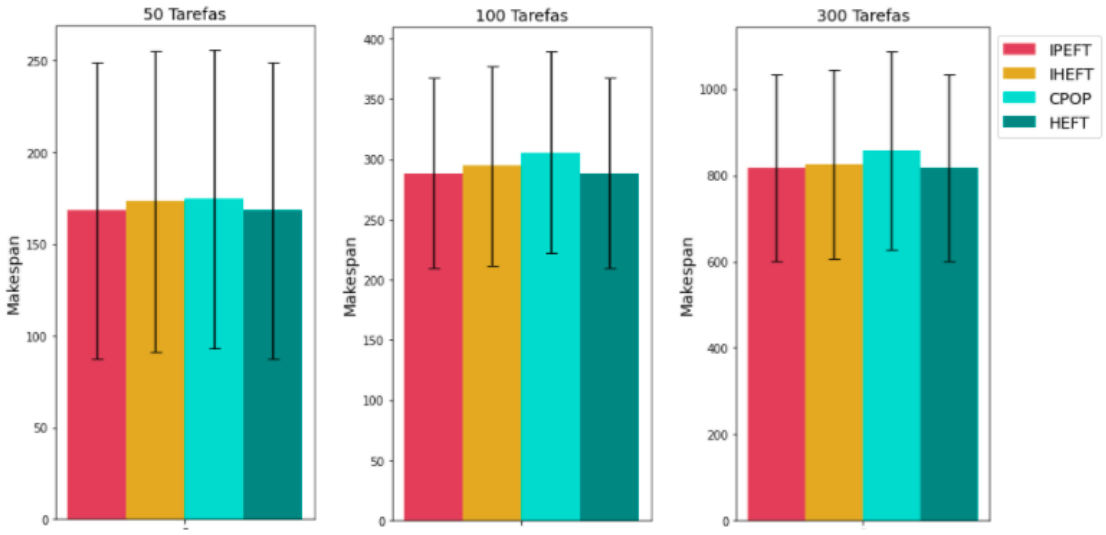
4.2.5 Cenário 5: Ambiente homogêneos

O último cenário avaliado neste trabalho considera que o ambiente computacional é homogêneo, ou seja, todos os processadores são idênticos. Não é o cenário ideal para os quatro algoritmos, visto que todos foram projetados visando ambientes heterogêneos. Porém, optou-se por explorar esse ambiente de modo a observar o comportamento desses algoritmos. Para simular um ambiente homogêneo, basta manter a variação do custo computacional igual a 0; assim, não há diferença em relação ao processador que será escolhido para executar uma tarefa. Os parâmetros adotados aqui são mostrados a seguir:

- ❏ Número de tarefas: 50, 100 e 300
- ❏ Número de processadores: 3
- ❏ Variação de custo computacional: 0
- ❏ Variação de custo de comunicação: 4
- ❏ Total de grafos: 180

Para todos os cenários de Makespan, nesse ambiente, o IPEFT e o HEFT obtiveram resultados idênticos e melhores que o CPOP e IHEFT; em seguida, tem-se os resutados do IHEFT e, por último, do CPOP com a pior média. Esses resultados podem ser observados nos gráficos da Figura 13 e na Tabela 9.

Figura 13 – Média e desvio-padrão do makespan para os algoritmos IPEFT, IHEFT, CPOP e HEFT com grafos de 50, 100 e 300 tarefas com custo computacional homogêneo.



Fonte: autoria própria.

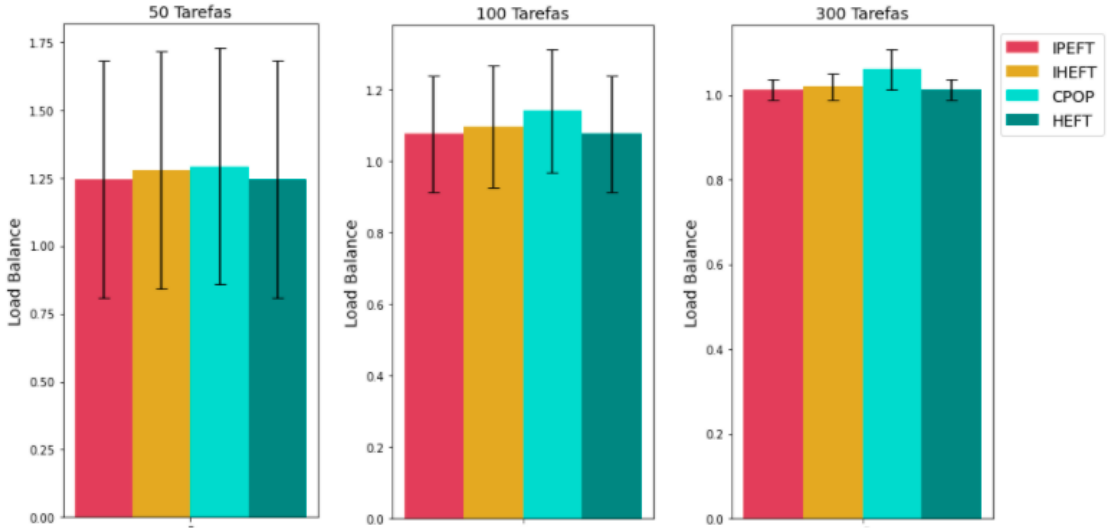
Tabela 9 – Média e desvio-padrão do makespan para os algoritmos IPEFT, IHEFT, CPOP e HEFT com grafos de 50, 100 e 300 tarefas com custo computacional homogêneo.

Número de tarefas	IPEFT	IHEFT	CPOP	HEFT
50	168.4 ± 80.6	173.2 ± 81.8	174.5 ± 81.3	168.4 ± 80.6
100	288.4 ± 78.7	294.6 ± 82.6	305.8 ± 83.8	288.4 ± 78.7
300	818.6 ± 216.1	824.8 ± 824.8	858.0 ± 229.4	818.6 ± 216.1

Fonte: autoria própria.

Finalmente, em relação ao Load Balance, é notável um comportamento similar ao Makespan, com as médias do IPEFT e o HEFT sendo idênticas e melhores que o CPOP e o IHEFT; em seguida os valores do IHEFT e, por último, o CPOP com a pior média (Figura 8 e Tabela 10). Além disso, o desvio-padrão vai diminuindo conforme é aumentado o número de tarefas, pelo fato de aproximar cada vez mais do valor mínimo do Load Balance, valor igual a um.

Figura 14 – Média e desvio-padrão do load balance para os algoritmos IPEFT, IHEFT, CPOP e HEFT com grafos de 50, 100 e 300 tarefas com custo computacional homogêneo.



Fonte: autoria própria.

4.3 Discussão

4.3.1 Resultados nos ambientes heterogêneos

Com foco nos ambientes heterogêneos (Cenários de 1 a 4), o algoritmo IPEFT demonstrou uma consistência de bons resultados de Makespan quando comparado com os

Tabela 10 – Média e desvio-padrão do load balance para os algoritmos IPEFT, IHEFT, CPOP e HEFT com grafos de 50, 100 e 300 tarefas com custo computacional homogêneo.

Número de tarefas	IPEFT	IHEFT	CPOP	HEFT
50	1.246 ± 0.437	1.279 ± 0.437	1.293 ± 0.436	1.246 ± 0.437
100	1.077 ± 0.162	1.098 ± 0.170	1.141 ± 0.172	1.077 ± 0.162
300	1.012 ± 0.025	1.019 ± 0.031	1.060 ± 0.047	1.012 ± 0.025

Fonte: autoria própria.

outros algoritmos. Assim, nos quatro ambientes heterogêneos, utilizando-se como base os gráficos gerais, ele teve a menor média em três ambientes e ficando com a segunda menor média no ambiente com variação baixa de custo de comunicação, perdendo por uma diferença de apenas 0,2 para o HEFT, como mostrado na Tabela 3.

Mesmo com o IPEFT obtendo os melhores resultados de Makespan, o HEFT ainda sim demonstrou um bom desempenho e proximidade com o IPEFT. Calculando-se a média sobre os gráficos gerais nos quatro ambientes, é uma média de 305.9 para o IPEFT e uma média de 307.6 para o HEFT, uma diferença de apenas 0.5% entre essas médias.

Porém, os resultados gerais de Makespan do CPOP e IHEFT, não demonstraram ser bons como o IPEFT e o HEFT. O IHEFT teve a pior média geral de Makespan no ambiente com variação alta de custo de comunicação e no ambiente com baixo custo computacional. Entretanto, o IHEFT alcançou bons resultados quando trabalhava com poucos processadores, isso é notável nas Tabelas 1, 3 e 5, com número de processadores igual a dois. Observa-se que o IHEFT se perde ao focar em melhores locais com muitos processadores, sendo que, na maioria das vezes, o foco no melhor global seria suficiente para obter bons resultados, como é provado pelo desempenho do HEFT.

O algoritmo CPOP conteve a pior média geral de Makespan no ambiente com variação alta de custo computacional e no ambiente com variação baixa de custo de comunicação. Esse algoritmo, diferente do IHEFT, não demonstrou um bom desempenho em nenhum tipo específico de ambiente; ele entregou resultados ruins, comparando com o IPEFT e o HEFT, em todos os ambientes utilizados. O que leva a pensar que o foco de escalonamento do caminho crítico não é a melhor opção.

Outra característica importante de discutir nos resultados de Makespan do IHEFT é seus desvios-padrões, onde seus valores são notavelmente maiores que dos outros algoritmos. Essa característica pode ser explicada por ser um algoritmo com decisões evolvendo aleatoriedade na busca por melhor local ou melhor global, e, assim, seus resultados variam mais do que os outros algoritmos.

Discutindo sobre os resultados Load Balance, são evidentes os valores elevados por parte do algoritmo IHEFT, em comparação com os outros. Um cenário interessante ocorre nos resultados gerais das Tabelas 5 e 6, onde por mais que o IHEFT tenha obtido

valores de média de Makespan menores que o CPOP, o valor de Load Balance do IHEFT é significativamente maior que o CPOP. Assim, isso mostra uma característica importante nas escolhas dos processadores que mais reduzem o custo de computação de uma tarefa (melhor local), de forma que o IHEFT alcança um Makespan menor, e os processadores têm uma ociosidade bem maior que nos outros algoritmos. Porém, os algoritmos CPOP, IPEFT e IHEFT mantêm um padrão entre os valores de Makespan e Load Balance para todas as médias.

4.3.2 Resultados nos ambientes homogêneos

Primeiramente, vale destacar um comportamento similar entre o IPEFT e HEFT nesse ambiente (Cenário 5). Devido ao fato dos custos computacionais de uma dada tarefa serem iguais para todos os processadores, o valor de CNCT do IPEFT, para essas tarefas, será sempre o mesmo para todos os processadores. Com isso, ele não influenciará na escolha do menor EFT entre os processadores. Dessa forma, o algoritmo IPEFT tem o mesmo comportamento no escalonamento que o HEFT, e, assim, seus resultados são idênticos tanto no Makespan, quanto no Load Balance.

Outro ponto válido de citar, refere-se aos resultados do CPOP. Pelo fato do caminho crítico ter o mesmo custo entre todos os processadores, é implicado que o foco no caminho crítico não é uma boa opção para esse ambiente. Isso é mostrado com os resultados do CPOP, que foram os piores em comparação com os outros algoritmos.

Capítulo 5

Conclusões

Algoritmos de escalonamento de tarefas têm sido propostos por diferentes trabalhos, tendo como foco diferentes características desse problema. Neste trabalho, desenvolveu-se um ambiente de avaliação experimentos e comparou-se o desempenho de quatro algoritmos bastante conhecidos (TOPCUOGLU; HARIRI; WU, 2002; ALEBRAHIM; AHMAD, 2016; ZHOU et al., 2016). Devido ao fato dos experimentos terem sido realizados em ambientes bem variados entre si, foram observadas características importantes sobre os algoritmos, que só foram reveladas em ambientes bem específicos.

O IPEFT mostrou-se, em geral, ser o melhor algoritmo em termos de Makespan, tornando assim condizente com o que é citado no seu artigo de origem (ZHOU et al., 2016), que o apresenta como um algoritmo com melhor desempenho que o HEFT, porém com a mesma complexidade de tempo. Portanto, demonstrou validez na sua principal característica preditiva, que o diferencia dos demais algoritmos.

O interessante do algoritmo HEFT (TOPCUOGLU; HARIRI; WU, 2002) é que, por mais que seja um algoritmo relativamente antigo, também demonstra bons resultados de Makespan, a ponto de, em alguns poucos cenários, ser o melhor de todos. Portanto, isso explica muito por que tantos algoritmos foram criados com base nas características do HEFT.

Já o algoritmo CPOP (TOPCUOGLU; HARIRI; WU, 2002), por mais que seu trabalho de origem o descreva como um algoritmo de bom desempenho, com resultados próximos até do HEFT, não se encaixou com os experimentos obtidos nessa pesquisa. O CPOP demonstrou resultados distantes quando comparado com os algoritmos que obtiveram bons resultados, e, por mais variado que tenham sido os cenários, ele não mostrou um bom desempenho em nenhum deles.

Por último, o IHEFT (ALEBRAHIM; AHMAD, 2016) mostrou ser o algoritmo mais peculiar entre os quatro. No geral, ele foi um algoritmo que não entregou bons resultados, o que discorda um pouco de sua literatura, que o propõe como um algoritmo focado em reduzir o valor de Makespan. Porém, em alguns poucos cenários, ele demonstrou ser o melhor algoritmo, superando até o IPEFT. Além disso, foi notável sua grande oscilação

nos resultados (o que pode ser visto pelos seus desvios-padrões), devido a sua busca por melhor local ou melhor global, envolvendo fator aleatório.

Portanto, esses experimentos e resultados podem entregar uma nova perspectiva sobre esses algoritmos, conhecendo assim um pouco mais sobre suas características e até reafirmando algumas conclusões externas sobre eles, principalmente ao executá-los em ambientes homogêneos, já que nenhum deles foi desenvolvido com foco nesse tipo de cenário.

Devido a extensão desse trabalho, outras métricas interessantes não foram utilizadas, como por exemplo o *Scheduling length ratio* (SLR), tempo de espera, *speedup*, robustez, entre outras. Porém, com os quadro algoritmos já desenvolvidos, muitos outros trabalhos podem reutilizá-los e levar em consideração essas outras métricas.

Outro ponto que pode ser aprofundado, futuramente, é quanto aos ambientes de execução, a fim de criar cenários ainda mais abrangentes e, assim, descobrir novas características dos algoritmos envolvidos. Por exemplo, pode-se utilizar cenários reais nos experimentos e ambientes mais extremos, em questão de número de tarefas e variações de custos. Além disso, este trabalho limitou-se a apenas em quatro algoritmos de *List-based scheduling*. Posteriormente, é possível explorar algoritmos baseados em outros paradigmas, incluindo meta-heurísticas (SILVA; GABRIEL, 2020).

O código desenvolvido, referente aos algoritmos de escalonamento de tarefas, geração de grafos e experimentos estão disponíveis publicamente no repositório do *Github*. [1]

[1] Disponível em <https://github.com/Brenocsc/Task-Scheduling>, acesso em 5 mar. 2022.

Referências

ALEBRAHIM, S.; AHMAD, I. Task scheduling for heterogeneous computing systems. **The Journal of Supercomputing**, Springer Science and Business Media LLC, v. 73, n. 6, p. 2313–2338, nov. 2016.

ALMASI, G. S.; GOTTLIEB, A. **Highly parallel computing**. Redwood City, CA: Benjamin–Cummings Publishing, 1989. 519 p. ISBN 9780805301779.

ARABNEJAD, H.; BARBOSA, J. G. List scheduling algorithm for heterogeneous systems by an optimistic cost table. **IEEE Transactions on Parallel and Distributed Systems**, Institute of Electrical and Electronics Engineers (IEEE), v. 25, n. 3, p. 682–694, mar. 2014.

BITTENCOURT, L. F.; GOLDMAN, A.; MADEIRA, E. R. M.; FONSECA, N. L. S. da; SAKELLARIOU, R. Scheduling in distributed systems: A cloud computing perspective. **Computer Science Review**, Elsevier BV, v. 30, p. 31–54, nov. 2018.

GRAHAM, R. L. Bounds for certain multiprocessing anomalies. **Bell System Technical Journal**, Institute of Electrical and Electronics Engineers (IEEE), v. 45, n. 9, p. 1563–1581, nov. 1966.

KWOK, Y.-K.; AHMAD, I. Dynamic critical-path scheduling: an effective technique for allocating task graphs to multiprocessors. **IEEE Transactions on Parallel and Distributed Systems**, Institute of Electrical and Electronics Engineers (IEEE), v. 7, n. 5, p. 506–521, maio 1996.

PLANETA, M. **Simulation of a Scheduling Algorithm for DAG-based Task Models**. 67 p. Dissertação (Mestrado) — Technische Universität Dresden, Dresden, Alemanha, maio 2015.

RĂDULESCU, A.; GEMUND, A. J. C. van. On the complexity of list scheduling algorithms for distributed-memory systems. In: **Proceedings of the 13th international conference on Supercomputing**. [S.l.]: ACM Press, 1999. p. 68–75.

RIOS, R. A.; JACINTO, D. S.; SCHULZE, B.; GUARDIA, H. C. Análise de heurísticas para escalonamento online de aplicações em grade computacional. In: GONÇALVES, P. A. da S. (Ed.). **Anais do VII Workshop de Computação em Grade e Aplicações**. Recife: Sociedade Brasileira de Computação, 2009. p. 13–24.

ROBERT, Y. Task graph scheduling. In: PADUA, D. (Ed.). **Encyclopedia of Parallel Computing**. Boston, MA: Springer US, 2011. p. 2013–2025.

SHARMA, N. **Static list scheduling for DAGs: A comparative study between algorithms**. 2019. Online. Disponível em: <https://github.com/sharma-n/DAG_Scheduling>. Acesso em: 23 mar. 2022.

SHETTI, K. R.; FAHMY, S. A.; BRETSCHNEIDER, T. Optimization of the HEFT algorithm for a CPU-GPU environment. In: **2013 International Conference on Parallel and Distributed Computing, Applications and Technologies**. [S.l.]: IEEE, 2013. p. 212–218.

SILVA, E. C. da; GABRIEL, P. H. R. A comprehensive review of evolutionary algorithms for multiprocessor DAG scheduling. **Computation**, MDPI AG, v. 8, n. 2, p. 26, abr. 2020.

TOPCUOGLU, H.; HARIRI, S.; WU, M.-Y. Performance-effective and low-complexity task scheduling for heterogeneous computing. **IEEE Transactions on Parallel and Distributed Systems**, Institute of Electrical and Electronics Engineers (IEEE), v. 13, n. 3, p. 260–274, mar. 2002.

ULLMAN, J. D. NP-complete scheduling problems. **Journal of Computer and System Sciences**, Elsevier BV, v. 10, n. 3, p. 384–393, jun. 1975.

ZHOU, N.; QI, D.; WANG, X.; ZHENG, Z.; LIN, W. A list scheduling algorithm for heterogeneous systems based on a critical node cost table and pessimistic cost table. **Concurrency and Computation: Practice and Experience**, Wiley, v. 29, n. 5, p. e3944, set. 2016.