UNIVERSIDADE FEDERAL DE UBERLÂNDIA

John Vitor da Silva Cunha

Algoritmo genético para escalonamento de tarefas em ambientes paralelos heterogêneos

Uberlândia, Brasil

2024

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

John Vitor da Silva Cunha

Algoritmo genético para escalonamento de tarefas em ambientes paralelos heterogêneos

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, como parte dos requisitos exigidos para a obtenção título de Bacharel em Ciência da Computação.

Orientador: Paulo Henrique Ribeiro Gabriel

Universidade Federal de Uberlândia – UFU

Faculdade de Computação

Bacharelado em Ciência da Computação

Uberlândia, Brasil

2024

Resumo

O aumento do uso de programas de inteligência artificial e de computação de alto desempenho tem aumentado cada vez mais a demanda por computação paralela, a fim de economizar recursos computacionais e tempo na execução de programas cada vez mais exigentes. Para um bom paralelismo e uso dos recursos computacionais, é necessário um bom escalonamento das tarefas do programa a ser executado. Este trabalho visa implementar um algoritmo genético, inspirado em algoritmos conhecidos da literatura, para gerar soluções do problema do escalonamento de tarefas em um sistema multiprocessado heterogêneo e com custo de comunicação entre as tarefas alocadas em diferentes processadores. Neste trabalho foi realizado três cenários de experimentos, um para cada programa de aplicativo real utilizado, onde, em cada cenário, foi realizado o escalonamento de tarefas para o programa específico usando o AG implementado e quatro algoritmos determinísticos muito discutidos na literatura do problema abordado, são eles: IPEFT, IHEFT, CPOP e HEFT. Os resultados de todos os algoritmos foram comparados usando as métricas de *makespan* e *load balance* a fim de investigar se as características dos algoritmos genéticos podem ser benéficas para esta classe de problemas.

Palavras-chave: escalonamento de tarefas, algoritmos genéticos, computação paralela.

Lista de ilustrações

Lista de tabelas

Lista de Algoritmos

Sumário

1 Introdução

A evolução tecnológica fez com que diversas áreas do conhecimento demandassem cada vez mais soluções de alto desempenho computacional. Visando isto, este trabalho abordou o problema do escalonamento de tarefas em ambientes paralelos heterogêneos. Neste tipo de problema, um trabalho computacional (um programa, aplicativo ou processo) é dividido em sub-tarefas que podem ser executadas paralelamente em diferentes nós computacionais para otimizar alguma métrica de desempenho do sistema. Estes nós computacionais podem ser os núcleos de um processador ou um computador interligado a uma rede de computadores, onde cada nó executará paralelamente uma sub-tarefa. Existem diversas abordagens em algoritmos de escalonamento de tarefas, este trabalho explorou os algoritmos genéticos (**AG**) para resolver este problema. Algoritmos genéticos são classes de algoritmos inspirados na teoria da evolução das espécies de Charles Darwin, onde uma população de soluções (indivíduos) é iniciada e a cada geração podem ser aplicados, aos indivíduos, operações genéticas que o modifiquem e ao final da geração a nova população é reavaliada com base em uma função de adaptabilidade. A ideia de AGs é que bons indivíduos consigam se reproduzir e se manter durante mais tempo na população, ao mesmo tempo que não impede que as operações genéticas ocorram em indivíduos mal avaliados e que suas características diversas possam melhorar sua adaptação. Estas características fazem com que a população de soluções ao longo do tempo convirja para uma solução otimizada pela função de adaptabilidade de todos os indivíduos da população.

O tema do escalonamento de tarefas é bastante discutido na academia, mas também tem grande aplicação prática. Cada vez mais indústrias necessitam de computação paralela e de alto desempenho (SANTOS,2021). Além disso, com o aumento do número de dados, até usuários comuns podem necessitar de alto desempenho ao usar sistemas computacionais (PEDERNEIRAS,2019) incentivados principalmente pela nova onda de IA. Visando isso é importante que sistemas computacionais escalonem bem suas tarefas a fim de reduzir o tempo de execução ou a ociosidade dos computadores e/ou núcleos.

O trabalho deSilva(2020) busca discutir as diferentes abordagens e representações de algoritmos genéticos para o problema de escalonamento de tarefas em processadores heterogêneos. Este trabalho é uma revisão sistemática da literatura, foram revisados trabalhos publicados entre 1990 e 2018 sobre o tema, registrando uma grande variedade de representações de cromossomos e operadores genéticos, além das formas diferentes de codificação para estes algoritmos considerando o problema de escalonamento de tarefas. Este artigo foi bastante útil como consulta no desenvolvimento deste trabalho. Já o artigo deSantos(2023) discute os algoritmos genéticos focados na otimização multi objetivo para o problema de escalonamento de tarefas, onde mais de uma métrica é levada em

consideração para o AG otimizar. Neste trabalho também foi discutido o AG deOmara e Arafa(2010), sendo um algoritmo mono-objetivo para o problema do escalonamento de tarefas em um ambiente com processadores homogêneos.

Neste trabalho foi implementado um algoritmo genético para o escalonamento de tarefas paralelas executadas em processadores heterogêneos (ambiente onde o poder computacional é diferente entre os nós do sistema) e com custo de comunicação entre tarefas alocadas em diferentes processadores, onde foram executados experimentos com programas de aplicativos reais e os resultados obtidos foram comparados com a execução de quatro algoritmos determinísticos (IPEFT, IHEFT, CPOP e HEFT) implementados no trabalho deCosta(2022), que também foram executados com os mesmos programas testados no AG. O trabalho deCosta(2022) avaliou vários algoritmos diferentes para buscar soluções para este problema e avaliou e comparou o resultado de todos os algoritmos em diversos experimentos, porém nenhum dos algoritmos utilizados usou a abordagem evolutiva. Portanto, este trabalho busca entender as características dos AGs ao lidar com o problema citado e verificar se esta classe de algoritmo pode produzir boas soluções em comparação com algoritmos muito utilizados e discutidos na literatura sobre o tema. As soluções obtidas de cada algoritmo foram avaliadas pelas métricas de *makespan* e *load balance*, discutidas posteriormente no texto.

Para o desenvolvimento do trabalho foi utilizado a linguagem de programação Python em sua versão 3.12.2, usando o ambiente de desenvolvimento Visual Studio Code. Além disso, no trabalho deCosta(2022) foi proposto um algoritmo de conversão, que converte programas do padrão GHO, que representa as tarefas e suas dependências em um ambiente homogêneo (poder computacional de igual capacidade para qualquer processador), para o padrão GHE, que representa as tarefas e suas dependências em um ambiente heterogêneo e com custo de comunicação entre tarefas dependentes alocadas em diferentes processadores. Este algoritmo de conversão também foi utilizado neste trabalho.

O restante desta monografia está organizado da seguinte maneira: no Capítulo2 está descrito toda a fundamentação teórica do trabalho, descrevendo genericamente os algoritmos genéticos e suas características, além de uma explicação sobre o problema do escalonamento de tarefas e suas variações. No Capítulo3estão descritos trabalhos relacionados ao tema e que serviram como fontes de informação para esta monografia, além disso, este capítulo também contém uma breve descrição de cada um dos algoritmos determinísticos implementados no trabalho deCosta(2022) que foram comparados ao AG implementado neste trabalho. Todo o desenvolvimento do AG proposto, como a conversão de formato dos DAGs (*Directed Acyclic Graphs* ou grafos acíclicos dirigidos), a geração de indivíduos, os operadores genéticos e funções de avaliação utilizados é descrito no Capítulo4. No Capítulo5estão detalhados os programas utilizados para os experimentos e o funcionamento da execução dos cenários dos experimentos junto com os parâmetros

utilizados nas execuções do AG, além das tabelas com os resultados das execuções dos experimentos no AG e nos algoritmos determinísticos testados. Por fim, no Capítulo6 estão as conclusões observadas nos resultados dos cenários dos experimentos, além de recomendações de trabalhos futuros que podem explorar as características do AG para o problema abordado nesta monografia.

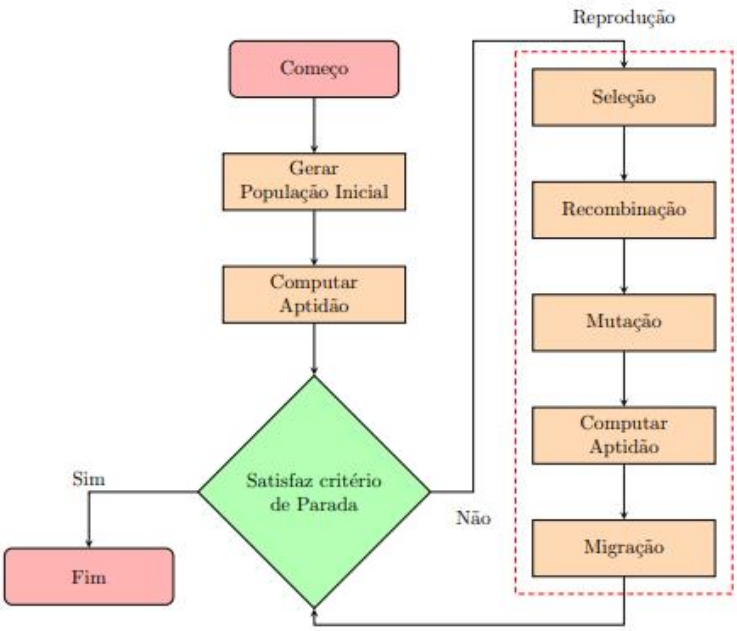
2 Fundamentação teórica

2.1 Algoritmos genéticos

Algoritmos genéticos (**AG**) são classes de algoritmos inspirados na Teoria da Evolução de Charles Darwin, onde uma população de indivíduos é submetida a operadores genéticos inspirados pela reprodução sexual e assexual, visando produzir novos indivíduos e posteriormente selecionar indivíduos para permanecerem na população, levando em conta uma função de aptidão para avaliar o indivíduo. Esta técnica foi inicialmente proposta no trabalho deHolland(1975). Como dito anteriormente, os AGs operam uma população de indivíduos, onde cada indivíduo representa uma possível solução para o problema. Esta característica dos AGs permite explorar várias soluções do espaço de busca em uma iteração, diferente de outros algoritmos de busca que trabalham com apenas uma solução por vez, como a Busca de subida de encosta (*Hill climbing search*) e os algoritmos de busca mais clássicos como a Busca em profundidade (*Depth-First Search*) e a Busca em largura (*Breadth-First Search*). Portanto, AGs implementam uma busca paralela das soluções visando a sobrevivência dos indivíduos mais aptos.

No início do algoritmo, uma população inicial é gerada com os indivíduos sendo possíveis soluções do problema. Após a geração da população inicial, é calculado a aptidão de cada cromossomo usando a função de aptidão para o problema, a fim de classificar cada indivíduo com uma nota. Depois o algoritmo entra em um laço de repetição que terminará ao final de n gerações ou até alcançar algum critério de parada. Em cada geração, n indivíduos são selecionados de acordo com sua nota de aptidão. Em seguida, os indivíduos selecionados passarão pelos operadores genéticos. O primeiro operador é o cruzamento (*crossover*), onde dois ou mais indivíduos (comumente chamados de pais) terão seus materiais genéticos recombinados a fim de gerar um ou mais indivíduos novos (comumente chamados de filhos), por ser um algoritmo estocástico, o cruzamento tem a probabilidade ou não de ocorrer. Após a etapa de cruzamento, os indivíduos selecionados passarão para a etapa de mutação, que também tem a probabilidade de ocorrer ou não, nesta etapa os indivíduos sofrerão uma ou mais modificações aleatórias em seus genes. Depois do cruzamento e mutação, a população de indivíduos aumenta, o que pode levar a ser necessária uma atualização da população para a geração seguinte, pois a ideia dos AGs é sempre ter uma população de tamanho fixo (isso pode mudar a depender da natureza do problema). Essa atualização pode usar vários critérios para selecionar quais indivíduos da população permanecerão para a geração seguinte. Estes passos são repetidos até atingir n gerações ou algum critério de parada. Na figura1é ilustrado o funcionamento geral de um AG.

Figura 1 – Algoritmo genético genérico

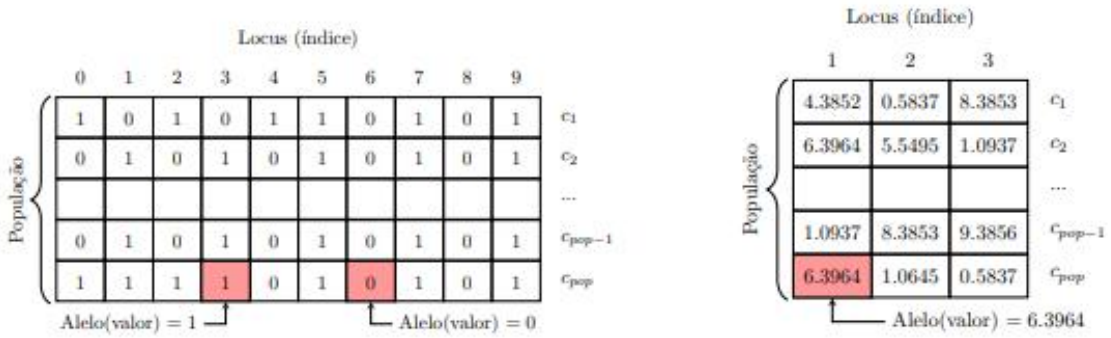


Fonte:Silva(2020)

Nos AGs, um cromossomo é a representação codificada de um indivíduo, onde o indivíduo pode ser visto como uma solução para o problema. A codificação de um indivíduo é um dos maiores desafios em um AG, pois é necessário abstrair o problema que o AG buscará resolver e essa abstração deve representar as características do problema além de ser compatível com os operadores do AG. Um mesmo problema pode comportar várias possibilidades de codificação dos cromossomos, porém diferentes representações podem levar o algoritmo a evoluir de diferentes formas. A codificação do cromossomo é muito importante para o desempenho do AG e diferentes codificações para um mesmo problema levam a um desempenho diferente do algoritmo, tanto na definição do espaço de busca quanto na codificação de indivíduos válidos, como mostrado porJelodar et al.(2006).

Uma população é um conjunto de indivíduos e conforme as gerações do algoritmo passam a população é evoluída pelos operadores genéticos, que podem criar novos indivíduos ou modificar indivíduos já existentes na população. A figura2, mostra duas codificações diferentes para um mesmo problema. A primeira codificação representa o indivíduo como um vetor de valores binários, já a segunda, presenta o indivíduo como um vetor de valores reais. Os alelos são os valores que cada posição do cromossomo pode assumir. Locus define o índice (ou posição) de um alelo no cromossomo. Chamamos de gene um alelo definido no lócus do cromossomo e representa uma unidade individual de informação do indivíduo (COELLO; LAMONT; VELDHUIZEN,2007).

Figura 2 – Exemplos de codificações de indivíduos



(a) Codificação binária. (b) Codificação real.

Fonte:Silva(2020)

2.1.1 Avaliação do indivíduo

Para avaliar a qualidade da solução de um indivíduo em um AG é utilizada uma função de aptidão, como descrito na seção2.1. O valor de aptidão (*fitness*) de um indivíduo mede sua qualidade e determina quão melhor ou pior a solução encontrada é em comparação com outras soluções para o problema. A aptidão pode ser o custo da solução em problemas de otimização. Para problemas de minimização, quanto menor o valor de aptidão, melhor adaptado aquele indivíduo está e mais próximo ele está da solução global para o problema.

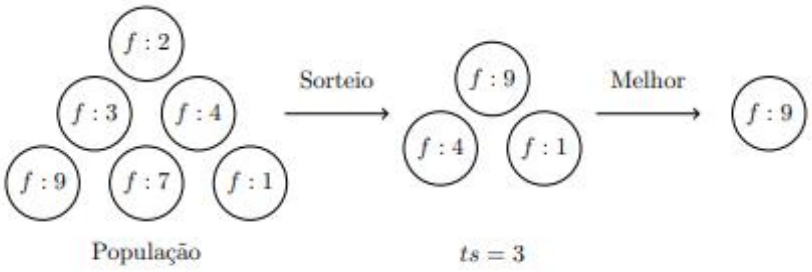
2.1.2 Seleção

A seleção é a etapa do AG onde indivíduos são selecionados para possíveis alterações genéticas, como a mutação e o cruzamento. Existem vários métodos de seleção de indivíduos para AGs e vários deles levam em conta a aptidão do indivíduo, há também a possibilidade da seleção totalmente aleatória, onde k indivíduos são selecionados aleatoriamente da população sem considerar seus valores de aptidão. Os indivíduos selecionados são chamados de pais, enquanto os novos indivíduos gerados pelas operações genéticas sobre os pais são chamados de filhos (ou descendentes). Os métodos de seleção mais popularmente utilizados em AGs são o método do torneio e o método da roleta.

2.1.2.1 Torneio

No método do torneio, n indivíduos são escolhidos aleatoriamente da população e têm seus valores de aptidão comparados, os indivíduos com os melhores valores de aptidão são selecionados para seguir para a fase de cruzamento e mutação (GOLDBERG; DEB, 1991). Na figura3é demonstrado o método de seleção por torneio.

Figura 3 – Método de seleção por torneio



Fonte:Silva(2020)

2.1.2.2 Roleta

A ideia do método da roleta é que os indivíduos com maior valor de aptidão tenham mais probabilidade de serem selecionados, este método é definido como uma probabilidade cumulativa. A ideia principal é representar cada indivíduo da população como um intervalo proporcional a seu valor de aptidão. Após a construção dos intervalos para cada indivíduo, um número aleatório é gerado, respeitando o intervalo da roleta, após isso é selecionado o indivíduo onde seu intervalo na roleta contém o número gerado. No método da roleta todos os indivíduos têm probabilidade de serem selecionados, porém, os de maior aptidão têm mais probabilidade de serem sorteados, o que pode levar a uma perca de diversidade da população. A probabilidade de um indivíduo ser selecionado pelo método da roleta é dado pela expressão abaixo, onde f_i é a aptidão do i-ésimo indivíduo de uma população com N indivíduos:

$$ProbSelec = \frac{f_i}{\sum_i^n f_i}$$

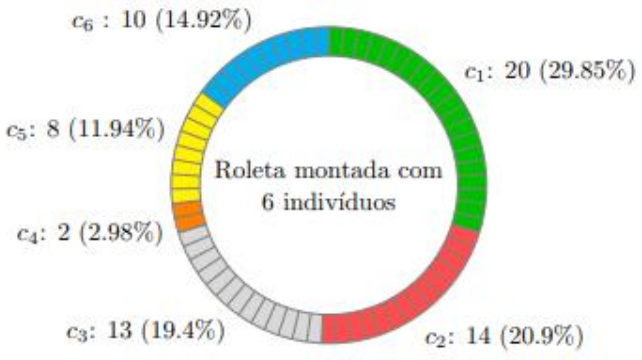
2.1.3 Operadores genéticos

Uma das vantagens já descritas sobre os AGs é sua característica de trabalhar com várias soluções e de evoluir a população de soluções ao longo do tempo. Para explorar ainda mais o espaço de busca à procura de outras soluções candidatas, os AGs utilizam os operadores genéticos, que, como observado na seção2.1, são inspirados nos conceitos biológicos de reprodução sexuada e assexuada.

2.1.3.1 Cruzamento

O cruzamento ou *crossover* é baseado na reprodução sexuada, onde dois ou mais indivíduos são escolhidos como pais e têm seus materiais genéticos misturados a fim de

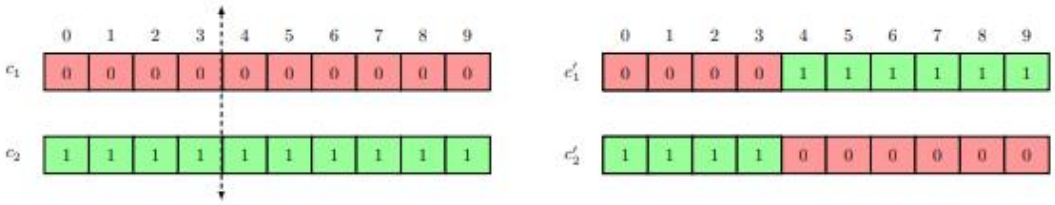
Figura 4 – Método de seleção por roleta



Fonte:Silva(2020)

criar um ou mais filhos. O cruzamento é um método estocástico, ou seja, a probabilidade do cruzamento ocorrer e como será feita a recombinação depende de fatores aleatórios. Um dos métodos de cruzamento mais popular é o cruzamento de ponto único. No método do ponto único, um número aleatório de 0 até o tamanho do cromossomo é gerado e este número será o locus do ponto de corte que será feito nos pais a fim de gerar os filhos, os filhos então serão gerados pelo cruzamento do material genético de ambos os pais. No exemplo mais tradicional, utilizando o método do ponto único, dois pais podem gerar dois filhos, como demonstrado na figura5(ZOMAYA; WARD; MACEY,1999).

Figura 5 – Operador genético de cruzamento



Fonte:Silva(2020)

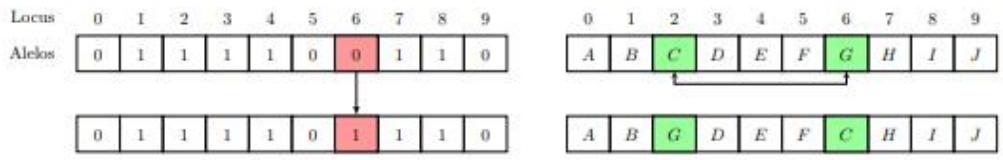
2.1.3.2 Mutação

A mutação é baseada na reprodução assexuada, ou seja, é aplicada em um único indivíduo. A mutação é um operador unário, portando é uma modificação em um gene do cromossomo que pode modificar o indivíduo selecionado ou gerar um novo filho a partir da modificação do gene. Como o operador de cruzamento, a mutação também é um método estático e ocorre a depender de fatores aleatórios, o que causa uma mudança não enviesada (EIBEN; SMITH,2015).

De acordo comGolub e Kasapovic(2002), a mutação é considerada um método para recuperar material genético perdido, permitindo uma exploração maior do espaço de busca e ajudando a prevenir a convergência prematura do algoritmo, aumentando sua diversidade. A mutação proposta porHolland(1975) é um operador que ocasionalmente muda os genes dos indivíduos, ocorrendo ou não mediante um número probabilístico $p_m \in [0, 1]$.

Dois exemplos de operadores de mutação são a mutação binária e mutação permutada. Na mutação binária, um lócus do cromossomo é sorteado aleatoriamente e tem seu gene modificado para um dos valores possíveis de alelo, método semelhante ao introduzido porHolland(1975). Já a mutação permutada é utilizada em cromossomos que não permitem a repetição de valores, neste caso, dois genes sorteados aleatoriamente tem seus alelos trocados, portanto, não gerando um indivíduo inválido. Na figura6há um exemplo de cada tipo de mutação explicado anteriormente.

Figura 6 – Operadores genéticos de mutação



Fonte:Silva(2020)

2.1.4 Reinserção

Ao aplicar os operadores genéticos, o número de indivíduos da população aumenta e é necessário escolher quais indivíduos serão passados para a próxima geração do AG e quais serão descartados. Esta etapa é chamada de reinserção ou migração e há várias formas de execução.

Um dos métodos de reinserção é o elitismo, onde uma porcentagem dos melhores indivíduos da geração atual serão passados diretamente para a geração seguinte, é um

método bastante utilizado, pois permite que bons indivíduos permaneçam por mais tempo na população, porém se esta porcentagem for muito alta pode levar a uma convergência prematura do algoritmo. Há também a possibilidade de utilizar os métodos de seleção explicados na sub-seção2.1.2, que podem ajudar a manter uma boa taxa de diversidade da população, pois não descartam diretamente indivíduos com aptidão baixa. Um método mais agressivo de reinserção é a reinserção pura, onde toda a população será substituída pelos filhos gerados pelos operadores genéticos, porém neste método bons indivíduos pais podem ser substituídos por filhos piores, piorando o resultado do algoritmo.

2.2 Problema do escalonamento de tarefas

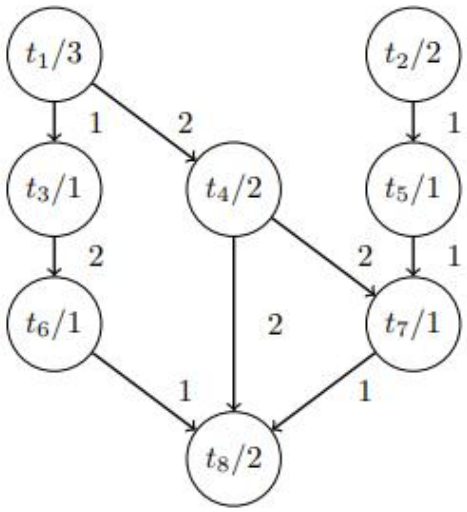
Com o avanço da computação de alto desempenho, viu-se a necessidade de utilizar arquiteturas com vários nós computacionais para resolver tarefas cada vez mais complexas (BITTENCOURT et al.,2018). Existem dois tipos de arquiteturas computacionais para resolver tarefas complexas, a arquitetura homogênea, onde os nós computacionais são idênticos, ou seja, uma tarefa executará da mesma forma em qualquer nó da arquitetura homogênea, exemplo deste tipo de arquitetura são os *clusters* computacionais. O outro tipo de arquitetura, que será abordado neste trabalho, é a arquitetura heterogênea, onde os nós computacionais podem se diferir em velocidade de processamento, capacidade de armazenamento, consumo energético, entre outras características. Os nós computacionais podem ser processadores em um único computador ou computadores interligados entre si. Neste trabalho os nós computacionais estão interligados entre si, ou seja, qualquer nó da arquitetura pode se comunicar com qualquer outro nó desta arquitetura.

Do problema do escalonamento de tarefas surge a necessidade de organizar previamente quais sub-tarefas serão executadas paralelamente nos processadores, a fim de que a tarefa principal (o programa) seja executada no menor tempo possível. Em muitas destas tarefas é encontrada restrições que devem ser respeitadas, por exemplo, é possível que alguma sub-tarefa t_i dependa dos dados de uma tarefa t_j para ser executada, neste caso, a tarefa t_i só poderá ser executada quando a tarefa t_j terminar seu processamento. No exemplo anterior, se as duas tarefas, t_i e t_j, serão executadas em processadores diferentes, então é necessário computar o custo de comunicação para os dados de t_j chegarem ao processador que executará t_i. Se as duas tarefas serão executadas no mesmo processador, então o custo de comunicação é tido como zero (PADUA,2011).

Uma das formas de representar um programa em sub-tarefas para serem executadas paralelamente é utilizando um DAG. Um DAG é um grafo onde as arestas são orientadas e não há ciclos, ou seja, para qualquer vértice v deste grafo, não há nenhum caminho orientado que passe por v mais de uma vez. Cada vértice deste grafo é uma sub-tarefa que pode ser executada paralelamente com outras tarefas, as arestas servem para indicar a

dependência de dados entre duas sub-tarefas e como dito anteriormente, um vértice v_i que tenha arestas chegando nele representa uma sub-tarefa que só poderá ser executada depois que todas as sub-tarefas que originam as arestas apontadas para v_i serem executadas.

Figura 7 – Exemplo de um DAG



Fonte:Costa(2022)

Na figura7é apresentado um DAG de um programa com oito tarefas. O valor contido em cada vértice é o custo para processar aquela tarefa. Os pesos das arestas é tido como o custo de comunicação para os dados de uma tarefa t_i chegar a uma tarefa t_j quando executadas em diferentes processadores. Para o problema de escalonamento de tarefas heterogêneo, como cada processador tem diferentes capacidades de processamento, então é necessária uma formalização um pouco diferente do DAG acima, que considera as capacidades de processamento dos computadores iguais.

Para o problema de escalonamento de tarefas, é definido um conjunto de m processadores dado por $P = \{p_1, p_2, ..., p_m\}$. De forma análoga temos o conjunto de tarefas definido por $T = \{t_1, t_2, ..., t_n\}$. Para arquiteturas heterogêneas é definido uma matriz W de dimensões n x m, onde cada elemento w_{ij} representa o custo de execução da tarefa t_i quando executada no processador p_j. Para definir o custo de comunicação entre os diferentes processadores é criada uma matriz B com dimensões m x m, onde cada elemento b_{ij} representa o custo de comunicação entre o processador p_i e o processador p_j.

Uma das métricas mais utilizadas no problema de escalonamento de tarefas é o *makespan*, definido como a diferença de tempo entre o início e o fim de execução de todas as tarefas do programa, ou seja, o *makespan* é o tempo total necessário para executar todas as tarefas do programa. No problema de escalonamento de tarefas o objetivo geralmente é minimizar o *makespan*, em outras palavras é executar todas as tarefas do programa no

menor tempo possível. Outras métricas também podem ser utilizadas em conjunto com o *makespan*, como o balanceamento de carga entre os processadores (ou *load balance*), que visa minimizar o tempo que os processadores ficam ociosos sem executar nenhuma tarefa.

3 Trabalhos relacionados

No trabalho deCosta(2022) são implementados quatro algoritmos para o problema de escalonamento de tarefas em processadores heterogêneos e com custo de comunicação entre os processadores. São eles: *Heterogeneous Earliest Finish Time* (TOPCUOGLU; HARIRI; WU,2002), *Critical Path on a Processor* (TOPCUOGLU; HARIRI; WU,2002), *Improved Heterogeneous Earliest Finish Time* (ALEBRAHIM; AHMAD,2017), *Improved Predict Earliest Finish* (ZHOU et al.,2017).

O algoritmo *Heterogeneous Earliest Finish Time* (HEFT), primeiramente, calcula uma prioridade para cada tarefa com base no tempo de execução e na comunicação, buscando sempre a ordem que minimize o tempo de conclusão. Em seguida, atribui cada tarefa ao processador que permita o término mais rápido, equilibrando tempo de processamento e comunicação entre as tarefas para otimizar o *makespan* (TOPCUOGLU; HARIRI; WU,2002).

O *Critical Path on a Processor* (CPOP) se concentra na execução do caminho crítico, ou seja, a sequência de tarefas que mais afeta o tempo total de execução. Após identificar o caminho crítico, esse algoritmo atribui todas as tarefas deste caminho a um único processador, sempre que possível, para evitar atrasos de comunicação entre elas. Tarefas fora do caminho crítico são atribuídas aos demais processadores, de acordo com a minimização do *makespan* (TOPCUOGLU; HARIRI; WU,2002).

O *Improved Heterogeneous Earliest Finish Time* (IHEFT) é um aprimoramento do algoritmo HEFT que introduz uma etapa de balanceamento de carga, ajustando a alocação de tarefas entre processadores de maneira mais eficiente para reduzir o tempo total de execução (*makespan*) e minimizando o tempo de ociosidade dos processadores. Outra característica do IHEFT é a inclusão de heurísticas para lidar melhor com tarefas de comunicação intensiva, resultando em um escalonamento mais adaptável e eficiente em relação ao HEFT original, principalmente em ambientes com maior heterogeneidade de recursos (ALEBRAHIM; AHMAD,2017).

Finalmente, o *Improved Predict Earliest Finish Time* (IPEFT) considera tanto os tempos de execução das tarefas quanto os tempos de comunicação entre elas, com o objetivo de minimizar o *makespan*. Além disso, incorpora heurísticas para prever com maior precisão os tempos de comunicação e adaptação de carga, ajudando a distribuir as tarefas de forma mais equilibrada entre os processadores (ZHOU et al.,2017).

Observa-se que estes algoritmos são compostos por duas fases, onde na primeira fase é feita a priorização das tarefas e na segunda, a seleção de processadores que executará cada tarefa. Essa característica os classificam como métodos heurísticos do tipo

List-based scheduling (GRAHAM,1966;SHARMA,2019). No trabalho deCosta(2022), são usadas as métricas *makespan* e *load balance* para comparar o desempenhos dos quatro algoritmos em cinco cenários distintos variando vários parâmetros do problema do escalonamento de tarefas. Este trabalho implementou um algoritmo genético para o problema do escalonamento de tarefas e o executou para programas de aplicativos reais, estes mesmo programas foram executados nos algoritmos determinísticos implementados no trabalho deCosta(2022) e os resultados de todos os algoritmos foram comparados a fim de avaliar o desempenho dos algoritmos inspirados em *List-based scheduling* com o algoritmo genético proposto. Além disso, o trabalho deCosta(2022) trás um algoritmo para converter grafos de tarefas de sistemas homogêneos e sem custo de comunicação para grafos de tarefas de sistemas heterogêneos e com custo de comunicação entre tarefas em diferentes processadores. Este algoritmo também foi implementado e utilizado neste trabalho para converter os grafos dos programas reais, que originalmente foram construídos para processadores homogêneos e sem custo de comunicação entre as tarefas, para o padrão proposto neste trabalho.

A primeira fase da implementação de um algoritmo genético é definir sua representação, isto é, abstrair o problema escolhido e representá-lo nas estruturas de um AG, definindo seu indivíduo, operadores genéticos e sua função de avaliação (*fitness*). Para o problema do escalonamento de tarefas em ambientes multiprocessados há várias formas de representação em estruturas de um AG. O trabalho deSilva(2020) trás uma revisão sistemática da literatura sobre algoritmos genéticos para o problema do escalonamento de tarefas, sumarizando trabalhos publicados entre 1990 e 2018. Neste trabalho foram encontradas 13 representações de indivíduos diferentes, além de 78 representação de operadores genéticos. No trabalho deSilva(2020) também foi comparado quatro diferentes codificações para o problema, são elas: Codificação de Listas Topológicas (TLE), Codificação de Alocação e Escalonamento (MSE), Codificação por Lista Ordenada (OLE) e Codificação por Lista de Processadores (PLE). Para a comparação foram usadas as métricas de *makespan*, *flowtime* e *load balance*. Por fim, o trabalho conclui que a representação OLE teve melhor desempenho comparada às outras, porém a representação OLE se utiliza de uma heurística de alocação de tarefas, o que pode reduzir a diversidade da população ao levar a convergência da busca para um ótimo local. As outras codificações não se diferiram muito em desempenho. O trabalho deSilva(2020) fornece uma ampla biblioteca de implementações de AGs para o problema do escalonamento de tarefas que pode ser utilizada como ponto inicial de uma pesquisa como esta monografia.

No trabalho deHou, Ansari e Ren(1994) é descrito uma representação de AG para o problema do escalonamento de tarefas, definindo o indivíduo, os operadores genéticos e a função de aptidão, além de um algoritmo para a geração da população inicial de soluções. Para representar o indivíduo, usaram uma lista de processadores, onde cada processador contém uma lista das tarefas que executará. Os operadores genéticos no trabalho de

Hou, Ansari e Ren(1994) são baseados no *crossover* de um ponto e mutação de *bit*, operadores bases muito utilizados em algoritmos genéticos. A função de aptidão usada foi o *makespan*. Ao implementar algoritmos genéticos, podem surgir indivíduos inválidos tanto na geração da população inicial, quanto no processamento de novos indivíduos através dos operadores genéticos. No problema do escalonamento de tarefas, um indivíduo inválido pode ser aquele onde uma tarefa T_j, que depende da execução da tarefa T_i, será executada antes da execução de T_i, o que viola as restrições de ordem das tarefas. Para resolver isso, Hou, Ansari e Ren(1994) garantiram que seu algoritmo que gera a população inicial não gera indivíduos inválidos, essa garantia é também estendida para os algoritmos dos operadores genéticos. O trabalho deHou, Ansari e Ren(1994) também compara o AG proposto com outro algoritmo para o escalonamento de tarefas baseado em lista, em cenários com grafos de tarefas gerados aleatoriamente e com grafos de programas reais. Na maioria dos cenários, o AG proposto teve um desempenho melhor do que o algoritmo baseado em lista e em nenhum cenário foi pior. O algoritmo proposto porHou, Ansari e Ren(1994) considera o sistema multiprocessado homogêneo, ou seja, os processadores são considerados idênticos em poder computacional e executam uma mesma tarefa T_i com o mesmo custo. O AG proposto também não considera custos de comunicação entre a execução de tarefas dependentes em diferentes processadores.

4 Desenvolvimento

Neste capítulo serão descritos os métodos, algoritmos e técnicas empregados no desenvolvimento deste trabalho. Inicialmente será apresentado o algoritmo de geração dos grafos acíclicos orientados (DAGs) proposto porCosta(2022). Em seguida, será apresentado o algoritmo para geração da população inicial do AG e, logo após, a descrição do funcionamento do algoritmo, da técnica de seleção por roleta, dos operadores genéticos e estratégia de elitismo adotados.

4.1 Geração dos DAGs

O grafo acíclico orientado (DAG) é uma representação de um programa computacional que descreve as tarefas e custos de computação do programa. Este tipo de representação serve como entrada para vários algoritmos para o problema do escalonamento de tarefas e facilita a visualização das tarefas, seus custos e predecessores. Tantos os quatro algoritmos implementados porCosta(2022) quanto o algoritmo genético proposto neste trabalho utilizam o formato DAG como entrada. Porém, os grafos dos programas reais usados nos experimentos deste trabalho, retirados de uma bibliotecas de DAGs mantida pela *Waseda University* (Japão) [1], eram grafos para processadores homogêneos e sem custo de comunicação (GHO) e o algoritmo proposto neste trabalho trabalha com grafos para processadores heterogêneos e com custo de comunicação entre tarefas dependentes alocadas em diferentes processadores (GHE).

Para resolver isto foi implementado neste trabalho um algoritmo de conversão proposto pelo próprio trabalho deCosta(2022). O algoritmo consiste em ler como entrada um DAG GHO e o número de processadores que se deseja fazer o escalonamento, e para cada tarefa gera os custos de computação para cada processador baseado no custo de computação do grafo original e gera também os custos de comunicação de dados entre tarefas dependentes para cada processador. O resultado será um DAG GHE que servirá de entrada para os algoritmos contidos neste trabalho.

O algoritmo de conversão recebe como entrada um DAG GHO, o número de processadores para alocar as tarefas, um valor inteiro Δ_{comp} que determina o limite superior de custo de computação de uma tarefa em um processador e um valor inteiro Δ_{comu} que determina o limite superior de custo de comunicação entre tarefas dependentes que estão alocadas em diferentes processadores. Tarefas de entrada e saída tem seus custos de computação e comunicação iguais a zero. Para gerar o custo computacional de uma tarefa em

[1] <https://www.kasahara.cs.waseda.ac.jp/schedule/index.html>

um processador é usada a fórmula4.1e para gerar o custo de comunicação ($custoComu$) de uma tarefa é utilizada a fórmula4.2.

$$w_{i,j} = rand[wSTG_i, wSTG_i + \Delta_{comp}] \tag{4.1}$$

$$custoComu_i = rand[1, \Delta_{comu}] \tag{4.2}$$

onde $wSTG_i$ é o custo de computação da tarefa i no grafo original e $rand[x, y]$ é uma função que retorna um valor inteiro aleatório entre x e y.

Os DAGs dos programas reais (*robot*, *sparse* e *fppp*), que foram testados nos experimentos, são originalmente sem custo de comunicação e com custo de computação homogêneo. Com o algoritmo proposto porCosta(2022) foi possível convertê-los em grafos GHE.

4.2 Geração da população inicial

Uma solução em um algoritmo genético é chamada de indivíduo, os AGs trabalham com um conjunto de indivíduos ao mesmo tempo, explorando diferentes locais do espaço de busca em uma única iteração (HOLLAND,1975). Este conjunto de indivíduos representando soluções é chamado de população. Um AG, normalmente, começa sua execução recebendo (ou gerando) uma população de indivíduos inicial e através dos operadores genéticos e de seleção vai modificando suas soluções iniciais em busca de otimizar seu objetivo.

Para o problema do escalonamento de tarefas, o que vai ditar os detalhes da implementação do AG é a representação do indivíduo (ou solução, ou cromossomo). É essa representação que os algoritmos de seleção, elitismo, mutação e reprodução usarão para modificar a população de soluções. Nesta seção será apresentado qual a representação do indivíduo foi utilizada e o algoritmo utilizado para gerar soluções para a população inicial do AG.

4.2.1 Representação do indivíduo

No trabalho deSilva(2020) é apresentado uma revisão sistemática da literatura das características e representações de algoritmos genéticos implementados para o problema do escalonamento de tarefas, revisando trabalhos de 1990 a 2018. Nesta revisão é apresentado quatro categorias de codificações, classificando cada algoritmo revisado. Para este trabalho foi escolhida a representação de Codificação de Alocação e Escalonamento, também chamada de *MSE*, pela facilidade de visualização da solução e implementação do código, além de ser a codificação mais utilizada na literatura analisada porSilva(2020).

A codificação MSE foi proposta porWang et al.(1997), nesta representação o cromossomo (indivíduo) é dividido em duas partes. A primeira parte, chamada aqui de alocação, é uma cadeia que representa a alocação das tarefas sobre os processadores. A segunda parte, chamada aqui de escalonamento, é uma cadeia que define a ordem de execução das tarefas.

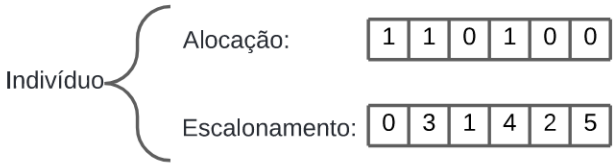
Na cadeia de alocação é definido qual o processador que executará uma tarefa. A cadeia de alocação tem tamanho n, que é o número de tarefas do grafo de entrada, de forma que cada posição $alocacao(i) = p_j$, onde $0 \leq i < n$, $0 \leq j < m$, n o número de tarefas do grafo e m a quantidade de processadores que serão alocados para executar as tarefas do programa do grafo. Ou seja, cada posição da cadeia armazena o processador da tarefa do respectivo índice.

A cadeia de escalonamento define a ordem que as tarefas do grafo de entrada serão executadas. Em uma solução válida, todas as tarefas predecessoras devem ser alocadas antes de suas sucessoras e esta foi a única restrição utilizada na geração da população inicial. Esta cadeia também tem tamanho n e cada posição contém o identificador da tarefa que será executada, a ordenação da cadeia define a ordem de execução das tarefas do programa.

Por consequência da utilização da representação por DAG duas tarefas fictícias são geradas para qualquer grafo. A primeira é a tarefa de entrada e será a primeira a ser executada, antes de qualquer outra. A segunda é a tarefa de saída e será a última tarefa a ser executada, depois de todas as outras. Ambas as tarefas têm custos de computação e comunicação iguais a zero, para não interferirem nos resultados dos algoritmos. A tarefa de entrada não tem predecessores e a tarefa de saída não tem sucessores.

No algoritmo implementado neste trabalho, usando linguagem Python, o indivíduo foi representado como um dicionário (um caso particular de *hash table*) com duas chaves, a primeira chave é uma lista de inteiros representando a cadeia de alocação, a segunda uma lista de identificadores das tarefas representando a cadeia de escalonamento. Na figura8 é possível visualizar um exemplo de representação de um indivíduo para um grafo com 6 tarefas em um sistema com dois processadores. Nesta figura temos a ordem de execução das tarefas na lista de Escalonamento, a primeira a ser executada será a tarefa de identificador “0” e a última será a tarefa de identificador “5”, como dito anteriormente, estas duas tarefas são fictícias. Na lista de Alocação estão contidos os processadores alocados para as tarefas, a tarefa de identificador “0” será executada pelo processador 1, a tarefa de identificador “2” será executada pelo processador 0.

Figura 8 – Representação do indivíduo



Fonte: Autoria própria.

4.2.2 Geração da população inicial

Para gerar a população inicial é preciso implementar um algoritmo de geração de solução e a partir deste algoritmo é possível gerar n soluções que integrarão a população inicial do AG. Para este trabalho foi implementado uma forma simples de gerar uma solução que pode ser visualizada pelo pseudo-código no algoritmo 1.

Algoritmo 1: Algoritmo para geração de indivíduo válido

Entrada: lista de tarefas e lista de processadores
Saída: indivíduo solução para o problema

```
1  início
2      individuo['alocacao'] = [];
3      individuo['escalonamento'] = [];
4      repita
5          se é primeira iteração então
6              tarefa = 0
7          fim
8          senão
9              tarefa = sorteiaTarefa();
10         fim
11         predecessores = pegaPredecessores(tarefa);
12         se predecessores já estão em individuo['escalonamento'] então
13             processador = sorteiaProcessador();
14             individuo['alocacao'].adiciona(processador);
15             individuo['escalonamento'].adiciona(tarefa);
16         fim
17     até todas as tarefas serem alocadas;
18     retorna individuo;
19 fim
```

A ideia do algoritmo consiste em criar o dicionário do indivíduo com as listas de Alocação e Escalonamento vazias, sortear uma tarefa aleatória da lista de tarefas do programa, conferir se os predecessores da tarefa sorteada já estão na lista de escalonamento; se sim então sorteia um processador da lista de processadores para a tarefa, o proces-

sador sorteado será adicionado à lista de Alocação e o identificador da tarefa sorteada será adicionado à lista de Escalonamento; se não repita o processo até todas as tarefas do programa serem alocadas. Se estiver na primeira iteração o sorteio da tarefa não ocorre, ao invés disso, a tarefa de entrada de identificador "0" é selecionada e um processador é alocado para ela. No final da execução do algoritmo será retornado um indivíduo que representa uma solução válida para o problema do escalonamento de tarefas para o grafo passado com entrada.

Para gerar a população inicial de indivíduos de tamanho n, basta executar o algoritmo1 n vezes e armazenar os indivíduos retornados em uma lista. Foi dessa forma a geração da população inicial do AG implementado neste trabalho.

4.3 Estratégia de elitismo

Como explicado na subseção2.1.4, uma das estratégias de condução de indivíduos para a próxima geração é o elitismo. Nesta estratégia, uma fração dos melhores indivíduos da população são selecionados e conduzidos para a próxima geração do AG. Para este trabalho foi implementado uma estratégia de elitismo onde uma porcentagem $p \in [0, 1]$ dos melhores indivíduos da população, avaliados por seu valor de *fitness*, são selecionados e repassados para a próxima geração do algoritmo, nesta estratégia o valor de p é passado como parâmetro de entrada ao algoritmo de elitismo e os indivíduos selecionados como elite ainda podem ser selecionados para a fase de cruzamento e mutação.

Algoritmo 2: Algoritmo para selecionar a elite da população

Entrada: população, tamanho n da população e porcentagem de elitismo p
Saída: lista dos melhores $p * n$ indivíduos da população

```
1 início
2 |  populacaoOrdenada = ordenaPopulacao();
3 |  quantidadeIndividuosElite = n * p;
4 |  retorna populacaoOrdenada[1 .. quantidadeIndividuosElite]
5 fim
```

No algoritmo2é demonstrado o funcionamento da estratégia de elitismo. Primeiro se ordena, em ordem crescente, a lista da população passada de entrada pelo valor de *fitness* dos indivíduos. Depois é calculado a quantidade de indivíduos de elite que serão retornados, com a função $n * p$, onde n é o tamanho da lista da população e $p \in [0, 1]$ é a porcentagem de quantos indivíduos serão selecionados para a elite. No final é retornado os $n * p$ primeiros indivíduos da lista da população ordenada. Um valor de $p = 0$ faz com que o AG execute sem nenhuma estratégia de elitismo, um valor de $p = 1$ faz com que a primeira geração gerada (a população inicial) sempre seja passada adiante e não sofra nenhuma modificação de reprodução ou mutação.

4.4 Estratégia de seleção

Após a fase de elitismo, vem a fase de seleção. Nesta fase, como explicado na subseção2.1.2, os indivíduos da população serão selecionados de acordo com alguma estratégia para passarem por possíveis alterações genéticas (mutação) ou cruzamento (*crossover*). A estratégia utilizada neste trabalho foi a seleção por roleta cumulativa, onde os indivíduos com maior aptidão (menor *fitness*) tem maior probabilidade de serem selecionados, porém, ainda mantendo um elemento estocástico na decisão de qual indivíduo será selecionado, permitindo que indivíduos de menor aptidão possam ser selecionados, aumentando a diversidade da população. Mais detalhes sobre a estratégia de seleção por roleta podem ser encontradas no tópico2.1.2.2.

Algoritmo 3: Seleção por roleta cumulativa em algoritmos genéticos

Entrada: População
Saída: Índice do indivíduo selecionado pela roleta

```
1  início
2      somaFitness ← 0;
3      para cada indivíduo na população faça
4          fitness ← calculaFitness(individuo);
5          fitnessAjustado ← 1/fitness;
6          somaFitness ← somaFitness + fitnessAjustado;
7      fim
8      roleta ← []
9      LimiteSuperior ← 0.0
10     para cada indivíduo na população faça
11         fitness ← calculaFitness(individuo)
12         fitnessAjustado ← 1/fitness
13         probabilidade ← fitnessAjustado/somaFitness;
14         LimiteInferior ← limiteSuperior;
15         LimiteSuperior ← limiteInferior + probabilidade;
16         Adicionar (LimiteInferior, LimiteSuperior) na roleta
17     fim
18     numeroSorteado ← número real aleatório entre 0 e 1;
19     para cada intervalo (LimiteInferior, LimiteSuperior) na roleta faça
20         se LimiteInferior ≤ numeroSorteado < LimiteSuperior então
21             retorna índice do indivíduo correspondente ao intervalo;
22         fim
23     fim
24 fim
```

O algoritmo3descreve a técnica de seleção por roleta cumulativa. Primeiro é calculado para cada indivíduo o valor do *fitness* ajustado (ou normalizado) com a equação 4.3e a soma destes valores também é calculada com a equação4.4.

Em seguida é inicializada uma lista vazia que conterá a roleta e a variável de *LimiteSuperior* com valor zero. Depois, para cada indivíduo, é calculado sua probabilidade de ser selecionado (equação4.5), seu intervalo de seleção definindo seu limite inferior (equação4.6) e superior (equação4.7), assim o par de limite inferior e superior é atribuído àquele indivíduo e adicionado na lista da roleta.

Por fim, um número aleatório $x \in [0, 1]$ é gerado e, para cada par de limites da roleta, é conferido se x está no intervalo do par. Quando o par de limites é encontrado, é retornado o índice do indivíduo da população e este indivíduo é considerado selecionado.

Para este trabalho, dois indivíduos diferentes são selecionados como pais a cada geração do AG para passar por possíveis operações genéticas e é possível que indivíduos que estejam na elite da população sejam selecionados.

$$fitnessAjustado(i) = \frac{1}{fitness(i)} \tag{4.3}$$

$$somaFitness = \sum_{i=1}^{n} fitnessAjustado(i) \tag{4.4}$$

$$probabilidade(i) = \frac{fitnessAjustado(i)}{somaFitness} \tag{4.5}$$

$$LimiteInferior(i) = LimiteSuperior(i-1) \tag{4.6}$$

$$LimiteSuperior(i) = LimiteInferior(i) + probabilidade(i) \tag{4.7}$$

onde n é o tamanho da população e i é o índice de um indivíduo da população

4.5 Operadores genéticos

Após selecionados, os indivíduos pais são potencialmente submetidos aos operadores genéticos. Neste trabalho foram implementados quatro operadores genéticos no AG, dois para o cruzamento (ou *crossover*) e dois para mutação. Um dos operadores, tanto de cruzamento quanto de mutação, atua na parte de alocação do cromossomo, enquanto o outro atua na parte do escalonamento. Por isso é passado como entrada ao AG quatro constantes numéricas representando as probabilidades de cruzamento e mutação para cada uma das partes do cromossomo do indivíduo.

Para haver cruzamento ou mutação, um número aleatório é gerado e comparado à constante numérica que representa a probabilidade de cruzamento ou mutação para uma das partes do indivíduo. Serão feitos no total seis testes. No primeiro é testado a

probabilidade de cruzamento de alocação dos pais; se bem-sucedido ambos os pais terão seu material genético de alocação cruzados para gerar duas novas cadeias de alocação para os novos filhos; se o teste falhar, então as cadeias de alocação dos pais são copiadas para os filhos sem alterações. O segundo teste é para o cruzamento do escalonamento, analogamente ao primeiro teste; se este obtiver sucesso então as cadeias de escalonamento dos pais são misturadas para gerar novas cadeias de escalonamento dos filhos; se falhar, as cadeias de escalonamento dos pais serão copiadas sem modificações para os filhos. No terceiro e quarto testes será avaliado se as cadeias de alocação dos filhos produzidos pela fase de cruzamento passarão por algum processo de mutação, primeiro se testa o primeiro filho e depois o segundo. Os últimos dois testes são análogos ao terceiro e quarto, mas a avaliação da mutação será para as cadeias de escalonamento, primeiro testando o primeiro filho e depois o segundo. O algoritmo4ilustra a primeira parte deste processo, submetendo dois indivíduos pais aos operadores de cruzamento. Logo após a fase de cruzamento, os indivíduos retornados são submetidos como entrada para o algoritmo5, na fase de mutação. Após as fases de cruzamento e mutação, os novos indivíduos produzidos são adicionados à população do AG. Se todos os testes aleatórios falharem, então os novos indivíduos gerados serão exatas cópias dos indivíduos pais selecionados.

Algoritmo 4: Algoritmo de cruzamento (*crossover*)

Entrada: Dois indivíduos pais, probabilidadeCrossoverAlocacao, probabilidadeCrossoverEscalonamento

Saída: Dois indivíduos modificados ou não pelos operadores de cruzamento

```
1  início
2      filhosAlocação ← [];
3      filhosEscalonamento ← [];
4      numeroAleatorio1 ← número real aleatório entre 0 e 1;
5      se numeroAleatorio1 < probabilidadeCrossoverAlocacao então
6          filhosAlocação ←
            Cruzamento_Alocação(pai1['alocação'], pai2['alocação']);
7      fim
8      senão
9          filhosAlocação ← [pai1['alocação'], pai2['alocação']];
10     fim

11     numeroAleatorio2 ← número real aleatório entre 0 e 1 ;
12     se numeroAleatorio2 < probabilidadeCrossoverEscalonamento então
13         filhosEscalonamento ←
            Cruzamento_Escalonamento(pai1['escalonamento'], pai2['escalonamento']);

14     fim
15     senão
16         filhosEscalonamento ←
            [pai1['escalonamento'], pai2['escalonamento']];
17     fim

18     filho1['alocação'] ← filhosAlocação[1] ;
19     filho1['escalonamento'] ← filhosEscalonamento[1] ;
20     filho2['alocação'] ← filhosAlocação[2] ;
21     filho2['escalonamento'] ← filhosEscalonamento[2] ;
22     retorna filho1, filho2;
23 fim
```

Algoritmo 5: Algoritmo de mutação

Entrada: Dois indivíduos, probabilidadeMutacaoAlocacao, probabilidadeMutacaoEscalonamento

Saída: Dois indivíduos modificados ou não pelos operadores de mutação

```
1  início
2      novosIndividuosAlocação ← [];
3      novosIndividuosEscalonamento ← [];
4      numeroAleatorio1 ← número real aleatório entre 0 e 1 ;
5      se numeroAleatorio1 < probabilidadeMutacaoAlocacao então
6          novosIndividuosAlocação[1] ←
             Mutação_Alocação(indivíduo1['alocação']);
7      fim
8      numeroAleatorio2 ← número real aleatório entre 0 e 1 ;
9      se numeroAleatorio2 < probabilidadeMutacaoAlocacao então
10         novosIndividuosAlocação[2] ←
             Mutação_Alocação(indivíduo2['alocação']);
11     fim
12     numeroAleatorio3 ← número real aleatório entre 0 e 1 ;
13     se numeroAleatorio3 < probabilidadeMutacaoEscalonamento então
14         novosIndividuosEscalonamento[1] ←
             Mutação_Escalonemnto(indivíduo1['escalonamento']);
15     fim
16     numeroAleatorio4 ← número real aleatório entre 0 e 1 ;
17     se numeroAleatorio4 < probabilidadeMutacaoEscalonamento então
18         novosIndividuosEscalonamento[2] ←
             Mutação_Escalonemnto(indivíduo2['escalonamento']);
19     fim

20     individuo1['alocação'] ← novosIndividuosAlocação[1] ;
21     individuo1['escalonamento'] ← novosIndividuosEscalonamento[1] ;
22     individuo2['alocação'] ← novosIndividuosAlocação[2] ;
23     individuo2['escalonamento'] ← novosIndividuosEscalonamento[2] ;
24     retorna individuo1, individuo2;
25 fim
```

4.5.1 Recombinação de Ponto Único (**SPX**)

De acordo comSilva(2020), uma das estratégias de cruzamento mais utilizadas é a Recombinação de Ponto Único, ou **SPX** e esta foi a estratégia de recombinação adotada neste trabalho. A ideia do SPX é definir um ponto de corte aleatório de no

máximo o tamanho da cadeia analisada (de alocação ou escalonamento), com este ponto de corte as cadeias dos pais são divididas em duas. A primeira parte da cadeia do primeiro pai concatenada com a segunda parte da cadeia do segundo pai vai dar origem a uma nova cadeia do primeiro filho. Analogamente, a primeira parte da cadeia do segundo pai concatenada com a segunda parte da cadeia do primeiro pai gerará uma nova cadeia do segundo filho. Ao final são gerados dois filhos gerados a partir do cruzamento do material genético dos pais. O algoritmo6demonstra o funcionamento do SPX nas cadeias de alocação.

A recombinação de ponto único pode ser aplicada tanto na cadeia de alocação quanto na cadeia de escalonamento, porém, no caso da cadeia de escalonamento, é preciso garantir que a restrição de precedência entre as tarefas seja respeitada para não gerar indivíduos inválidos. Para garantir esta restrição, foi implementada a estratégia descrita no algoritmo7, explicando que, após pegar a primeira parte da cadeia do primeiro pai e adicioná-la como a primeira parte do primeiro filho, é feito um laço sobre a cadeia do segundo pai, onde é adicionado ao primeiro filho toda tarefa do segundo pai que ainda não esteja contida no escalonamento do primeiro filho. E pela cadeia de escalonamento ser ordenada pela ordem de execução das tarefas, é garantido que o novo escalonamento respeita a restrição de precedência entre as tarefas. Analogamente, a mesma estratégia é usada para gerar a cadeia de escalonamento do segundo filho.

Algoritmo 6: Operador SPX para a cadeia de alocação

Entrada: cadeia de alocação do pai 1 e cadeia de alocação do pai2
Saída: duas novas cadeias de alocação geradas pela combinação das cadeias dos pais

```
1  início
2      pontoCorte ←
         número inteiro aleatório entre 1 e o tamanho da cadeia dos pais;
       //Construir o primeiro filho:
3      primeiraParte ← primeira parte do pai1 até o ponto de corte;
4      segundaParte ← segunda parte do pai2 a partir do ponto de corte;
5      filho1 ← primeiraParte + segundaParte;
       //Construir o segundo filho:
6      primeiraParte ← primeira parte do pai2 até o ponto de corte;
7      segundaParte ← segunda parte do pai1 a partir do ponto de corte;
8      filho2 ← primeiraParte + segundaParte;
9      retorna [filho1, filho2];
10 fim
```

Algoritmo 7: Operador SPX para a cadeia de escalonamento

Entrada: Cadeia de escalonamento do pai 1 e cadeia de escalonamento do pai2

Saída: duas novas cadeias de escalonamento geradas pela combinação das cadeias dos pais

```
1  início
2      pontoCorte ← número inteiro aleatório entre 1 e o tamanho da cadeia dos pais
       //Construir o primeiro filho:
3      tarefasFilho1 ← primeira parte do pai1 até o ponto de corte
4      filho1 ← tarefasFilho1
5      para cada tarefa em pai2 faça
6          se tarefa não está em tarefasFilho1 então
7              adicionar tarefa a filho1
8          fim
9      fim
       //Construir o segundo filho:
10     tarefasFilho2 ← primeira parte do pai2 até o ponto de corte
11     filho2 ← tarefasFilho2
12     para cada tarefa em pai1 faça
13         se tarefa não está em tarefasFilho2 então
14             adicionar tarefa a filho2
15         fim
16     fim
17     retorna [filho1, filho2]
18 fim
```

4.5.2 Operadores de mutação

Após aplicação da fase de cruzamento de alocação e escalonamento, o AG entra na fase de mutação. Para a fase de mutação, dois operadores distintos foram implementados, um para a cadeia de alocação do indivíduo e outro para a cadeia de escalonamento. Para a cadeia de alocação foi utilizado o método de mutação de processador, ou **PM**, utilizado em vários trabalhos comoLee e Chen(2003),Omara e Arafa(2010) eDhingra, Gupta e Biswas(2014). Para a cadeia de escalonamento foi implementado a Mutação de Deslocamento de Tarefas, ou **STM**, algoritmo também proposto nos trabalhos deWang et al.(1997) eGupta, Kumar e Agarwal(2010). Nesta sub-seção ambos os algoritmos serão explicados.

4.5.2.1 Mutação de Processador (**PM**)

Na mutação de processador, uma tarefa escolhida aleatoriamente tem seu processador de execução trocado por algum outro processador também aleatório. Primeiro, um número aleatório no intervalo de zero até o tamanho da cadeia do indivíduo é gerado. Este número é a posição da tarefa na cadeia de escalonamento que terá seu processador mudado. Após isso, um novo número aleatório entre 1 e o número de processadores é gerado e se este for diferente do número do processador da tarefa sorteada, então a tarefa será alocada para o novo processador. Se o número do novo processador for igual ao número do processador que está atualmente alocado para a tarefa sorteada, então um novo número de processador é sorteado até achar um processador diferente.

Algoritmo 8: Mutação por Troca de Processador (PM)

Entrada: Cadeia de alocação de um indivíduo e o número de processadores do sistema

Saída: O mesmo indivíduo com uma tarefa com seu processador modificado

```
1  início
2      posiçãoTarefa ← número inteiro aleatório ∈
         [0, tamanho da cadeia de alocação]
3      repita
4          novoProcessador ← número inteiro aleatório ∈
             [0, número de processadores do sistema]
5          se novoProcessador ≠ processadoratualdatarefa então
6              substituir processador atual da tarefa por novoProcessador na
                 posiçãoTarefa
7              retorna indivíduo modificado
8          fim
9      até para sempre;
10 fim
```

4.5.2.2 Mutação de Deslocamento de Tarefas (**STM**)

A ideia do STM é trocar duas tarefas de lugar na cadeia do escalonamento. Primeira uma tarefa da cadeia é selecionada aleatoriamente, em seguida uma posição da cadeia também é gerada aleatoriamente, o objetivo é colocar a tarefa selecionada na posição gerada aleatoriamente, consequentemente trocando essa tarefa de posição na cadeia com a tarefa que está ocupando atualmente a posição gerada aleatoriamente. O STM deve respeitar a restrição de precedência das tarefas ao fazer a troca de posições das tarefas, também não é possível que a tarefa selecionada aleatoriamente seja a primeira ou a última, pois são as tarefas fictícias de entrada e saída, respectivamente.

Para garantir que soluções inválidas não sejam geradas, primeiro é definido um

limite inferior para a nova posição da tarefa selecionada (tarefa 1). Para definir este limite inferior é feito uma iteração sobre a cadeia de escalonamento verificando se cada tarefa é predecessora da tarefa 1; se a tarefa é predecessora, então o valor do limite inferior é incrementado em um. Logo após é gerada uma nova posição aleatória, entre o limite inferior e o tamanho da cadeia de escalonamento, para a tarefa 1. É analisado então a tarefa que está atualmente nesta nova posição (tarefa 2); se todos os predecessores da tarefa 2 estiverem alocados no escalonamento até a posição original da tarefa 1, então a troca é permitida e as tarefas 1 e 2 trocam de posições na cadeia de escalonamento; senão o processo é repetido até encontrar uma troca válida. Este método garante que a ordem do escalonamento após a mutação seja válida, gerando um indivíduo válido. No algoritmo 9é mostrada a codificação deste método de mutação.

Algoritmo 9: Mutação de Deslocamento de Tarefas (STM)

Entrada: Cadeia de escalonamento de um indivíduo
Saída: A mesma cadeia com duas tarefas trocadas de posição

```
1  início
2      repita
3          tarefa1 ← escolha aleatória de uma tarefa do escalonamento
4          posicaoTarefa1 ← posição da tarefa1 no escalonamento
5          repita
6              tarefa1 ← escolha aleatória de uma tarefa do escalonamento
7              posicaoTarefa1 ← posição da tarefa1 no escalonamento
8          até até tarefa1 não ser a tarefa de entrada ou saída;
9          predecessoresTarefa1 ← predecessores da tarefa1
10         limiteInferior ← 0
11         para cada tarefa no escalonamento faça
12             se predecessoresTarefa1 estiver vazio então
13                 pare o loop
14             fim
15             se tarefa está em predecessoresTarefa1 então
16                 remova a tarefa de predecessoresTarefa1
17             fim
18             limiteInferior ← limiteInferior + 1
19         fim
20         novaPosicaoTarefa1 ← número ∈ [limiteInferior, tamanho do escalonamento − 1]
21         repita
22             novaPosicaoTarefa1 ← número ∈
                [limiteInferior, tamanho do escalonamento − 1]
23         até novaPosicaoTarefa1 ≠ posicaoTarefa1 e novaPosicaoTarefa1 ≠ 0;
24         tarefa2 ← tarefa na novaPosicaoTarefa1do escalonamento
25         predecessoresTarefa2 ← predecessores da tarefa2
26         para i ← 0 até posicaoTarefa1 faça
27             tarefa ← escalonamento[i]
28             se tarefa está em predecessoresTarefa2 então
29                 remova a tarefa de predecessoresTarefa2
30             fim
31             se predecessoresTarefa2 estiver vazio então
32                 troque tarefa1 e tarefa2 de posição no escalonamento
33                 retorna escalonamento modificado
34             fim
35         fim
36     até para sempre;
37 fim
```

Como exemplificado nos algoritmos4e5, após o cruzamento e mutação, as novas cadeias de escalonamento e alocação são concatenadas para gerar os dois novos indivíduos filhos, que serão adicionados à nova população do AG. Após gerar todos os filhos e adicioná-los, os indivíduos da elite também são adicionados à população. Para manter

a diversidade, os indivíduos da população são embaralhados e após este embaralhamento se começa uma nova iteração do algoritmo e se repetirá até atingir o número de iterações passado como entrada. Ao terminar as iterações, a população resultante é retornada.

4.6 Avaliação do indivíduo

Durante as fases de seleção e elitismo, o indivíduo tem seu desempenho avaliado por uma função de aptidão (*fitness*). Esta função vai avaliar a solução considerando as características do problema e medir o quão adaptado está o indivíduo na população. Indivíduos melhores adaptados têm maiores chances de serem selecionados para a elite, e ter seu material genético preservado, ou para passarem pela fase de reprodução do algoritmo. É de fato a função de aptidão que guia os algoritmos genéticos em direção à optimização do problema.

A função *fitness* desenvolvida para avaliar a qualidade dos indivíduos neste trabalho combina as métricas de *makespan* e *load balance*, ponderadas por um parâmetro $\alpha \in [0, 1]$. A expressão4.8descreve a fórmula da função de aptidão. O parâmetro α define a porcentagem de peso que cada uma das métricas assume na função de aptidão. Para um $\alpha = 1$, a função de aptidão ignorará o valor de *load balance* do indivíduo e avaliará somente o *makespan*. De maneira análoga, para um $\alpha = 0$ somente o valor de *load balance* será avaliado.

$$fitness = \alpha \times makespan + (1 - \alpha) \times loadBalance, \alpha \in [0, 1] \tag{4.8}$$

4.6.1 *Makespan*

A implementação da métrica de *makespan* deste trabalho soma, para cada processador, o tempo total de execução das tarefas alocadas. Se os predecessores de uma tarefa não estão alocados no mesmo processador da tarefa sucessora, então também é somado ao tempo de execução do processador, o custo de comunicação para os dados das tarefas predecessoras chegarem ao processador que está executando a tarefa. Por exemplo, se a tarefa A é predecessora da tarefa B, e ambas estão em processadores diferentes, o tempo necessário para transferir os dados de A para o processador que executa B será adicionado ao tempo total de execução do processador que executa B. Ao final do cálculo, cada processador terá um tempo total de execução, já considerando os custos de comunicação entre tarefas dependentes. Como cada processador executa suas tarefas paralelamente, o tempo total de execução de todas as tarefas do programa se dará pelo maior tempo que um processador leva para executar todas as tarefas alocadas a ele. Este valor máximo é retornado como sendo o *makespan* do indivíduo passado.

Algoritmo 10: Cálculo do *makespan* para um indivíduo

Entrada: Matriz de custos de comunicação e execução e indivíduo contendo:

- *escalonamento*: ordem das tarefas
- *alocação*: mapeamento de cada tarefa para um processador

Saída: Makespan: tempo total de execução do escalonamento

```
1  início
       //Inicializa o tempo de processamento para cada processador
2      para cada p de {1, 2, ..., número de processadores} faça
3          tempoProcessamento[p] ← 0;
4      fim
5      para cada i de {1, 2, ..., tamanho indivíduo['escalonamento']} faça
6          processador ← indivíduo['alocação'][i]
7          tarefa ← indivíduo['escalonamento'][i]
8          tempoComunicacaoAcumulado ← 0
9          predecessores ← Pega tarefas predecessores de tarefa
10         para cada j de {1, 2, ..., número de predecessores} faça
11             indicePredecessor ← índice de j no indivíduo['escalonamento']
12             processadorPredecessor ←
                indivíduo['alocação'][indicePredecessor]
13             se processadorPredecessor ≠ processador então
14                 tempoComunicacao ←
                    Custo de comunicação da tarefa em processador
15                 tempoComunicacaoAcumulado ←
                    tempoComunicacaoAcumulado + tempoComunicacao
16             fim
17         fim
18         tempoExecucao ← Tempo de execução da tarefa em processador
19         tempoProcessamento[processador] ←
            tempoProcessamento[processador] + tempoExecucao +
            tempoComunicacaoAcumulado
20     fim
       //Retorna o maior tempo de processamento
21     retorna max(tempoProcessamento)
22 fim
```

4.6.2 *Load balance*

A métrica do balanceamento de carga avalia o quão bem estão distribuídas as tarefas entre os diferentes processadores, avaliando a razão entre o *makespan* e o tempo

médio de execução das tarefas. Algumas soluções guiadas apenas pelo *makespan* podem apresentar situações onde alguns processadores ficam sobrecarregados enquanto outros são subutilizados. A métrica de *load balance* serve para medir a desigualdade entre a distribuição das tarefas dos processadores e ajudar a guiar o AG para soluções onde o balanceamento de carga seja mais equilibrado. Quanto menor o valor de *load balance* mais ocupados os processadores estarão.

Para este trabalho foi implementada a função de *load balance* conforme a expressão 4.9. O pseudo-código11também exemplifica o funcionamento deste algoritmo para o cálculo do balanceamento de carga. E como dito no início da seção4.6, tanto o *makespan* quanto o *load balance* são combinados para gerar o valor de *fitness* final do indivíduo, conforme a expressão4.8.

$$\begin{aligned} Load\ balance &= \frac{makespan}{avg}, \\ \text{onde} \quad avg &= \frac{\sum_{j=1}^{m} ftp[p_j]}{m}, \\ ftp[p_j] &\text{ é o tempo de execução do processador } p_j, \\ m &\text{ é o número total de processadores.} \end{aligned} \tag{4.9}$$

Algoritmo 11: Cálculo do *load balance* para um indivíduo

Entrada: Matriz de custos de execução e indivíduo contendo:

- *escalonamento*: ordem das tarefas
- *alocação*: mapeamento de cada tarefa para um processador

Saída: Load balance: razão entre o makespan e o tempo médio de processamento

```
1  início
       //Inicializa o tempo de processamento para cada processador
2      tempoProcessadores ← [0] * número de processadores
3      para cada i de {1, 2, ..., tamanho indivíduo['escalonamento']} faça
4          tarefa ← indivíduo['escalonamento'][i]
5          processador ← indivíduo['alocacao'][i]
6          tempoExecucao ← Tempo de execução de tarefa em processador
7          tempoProcessadores[processador] ←
             tempoProcessadores[processador] + tempoExecucao
8      fim
9      tempoProcessamentoTotal ← Soma de tempoProcessadores
10     tempoMedioProcessamento ← tempoProcessamentoTotal / número de processadores
11     makespan ← indivíduo['makespan']
12     retorna makespan/tempoMedioProcessamento
13 fim
```

5 Resultados e Discussão

Este capítulo abordará os experimentos realizados no trabalho, apresentando os DAGs utilizados nos testes, como o ambiente de experimentos foi desenvolvido, os parâmetros utilizados no AG e os resultados das métricas de *makespan* e *load balance* para cada um dos experimentos executados. Todo o ambiente de experimentos e execuções foram desenvolvidos utilizando a linguagem de programação Python em sua versão 3.12.2, usando o ambiente de desenvolvimento Visual Studio Code em um ambiente computacional com um processador Ryzen 7 5700X, 32 GB de memória no sistema operacional Windows. Todo o código também pode ser executado no Linux.

5.1 DAGs utilizados

Para este trabalho foram escolhidos três DAGs que foram modelados a partir de programas de aplicativos reais, são eles o *Robot control* (chamado aqui de *robot*), *Sparse Matrix Solver* (aqui chamado *sparse*) e o *SPEC fpppp* (chamado aqui de *fpppp*). Estes grafos podem ser encontrados no repositório de DAGs da *Waseda University*.

Os grafos contidos no site da *Waseda University* são grafos que não levam em consideração os custos de comunicação entre dados de tarefas dependentes e com custos de execução homogêneos (são chamados GHO). Para converter estes grafos GHO em grafos com custos de comunicação e diferentes tempos de execução para diferentes processadores (GHE), foi implementado o algoritmo de conversão mencionado na seção4.1. Os grafos GHE gerados da conversão e os originais estão no repositório do *Github* deste trabalho.

O grafo *robot* consiste em um grafo GHO com 88 tarefas reais e 2 tarefas fictícias, a de entrada e de saída. Para converter este grafo em um GHE, foram passados como argumento para o algoritmo de conversão quatro processadores, com $\Delta_{comp} = 20$ e $\Delta_{comu} = 30$.

O grafo *sparse* consiste em um grafo GHO com 96 tarefas reais e 2 tarefas fictícias. Os parâmetros passados para convertê-lo foram quatro processadores, com $\Delta_{comp} = 100$ e $\Delta_{comu} = 30$.

Por último temos o grafo *fpppp* com 334 tarefas reais, além das duas fictícias. Os parâmetros passados na conversão foram 8 processadores, com $\Delta_{comp} = 20$ e $\Delta_{comu} = 30$.

5.2 Sobre os experimentos

Para os experimentos, foram mantidos os mesmos parâmetros do AG, independente do grafo, a exceção do parâmetro α. A execução dos experimentos foram guiados pelo parâmetro α da função *fitness*. Este parâmetro, como descrito na seção4.6, controla o quanto as métricas de *makespan* e *load balance* influenciarão na evolução da população. Para visualizar os efeitos práticos que a mudança do α tem nas métricas de avaliação, foram escolhidos cinco valores para o parâmetro α, são eles:

- $\alpha = 0$: situação que somente o *load balance* será usado para o valor do *fitness*
- $\alpha = 0,25$: situação que o *makespan* contribuirá com 25% e o *load balance* contribuirá com 75% para o valor do *fitness*
- $\alpha = 0,50$: situação que o *makespan* e o *load balance* contribuirão cada um com 50% para o valor do *fitness*
- $\alpha = 0,75$: situação que o *makespan* contribuirá com 75% e o *load balance* contribuirá com 25% para o valor do *fitness*
- $\alpha = 1$: situação que somente o *makespan* será usado para o valor do *fitness*

Cada um dos grafos reais foi testado com os cinco valores de α descritos acima. Para cada experimento com um grafo e um α, o AG foi executado dez vezes. Em cada uma destas execuções, é armazenado o valor de *makespan*, *load balance* e geração do melhor indivíduo encontrado. Depois da execução do experimento, são tiradas as médias e desvios padrões dos *makespan*, *load balance* e gerações do melhor indivíduo das dez iterações. Em outras palavras, cada grafo foi testado dez vezes para cada α da lista acima, ao final de cada uma das dez execuções foi tirada a média das métricas avaliadas. À exceção do parâmetro α, todos os experimentos do AG foram executados com os seguintes parâmetros:

- Quantidade de indivíduos da população: 30
- Número de iterações do AG: 250
- Probabilidade de acontecer cruzamento de alocação: 0,4
- Probabilidade de acontecer cruzamento de escalonamento: 0,4
- Probabilidade de acontecer mutação de alocação: 0,2
- Probabilidade de acontecer mutação de escalonamento: 0,2
- Porcentagem de indivíduos conduzidos por elitismo: 0,2

Os experimentos foram divididos em três cenários. O primeiro cenário será dedicado para os experimentos do grafo *robot*, o segundo para o grafo *sparse* e o terceiro para o grafo *fpppp*. Para cada um dos cenários foi executado os quatro algoritmos implementados no trabalho deCosta(2022), são eles: IPEFT, IHEFT, CPOP e HEFT. Por serem algoritmos determinísticos, eles foram executados apenas uma vez para cada um dos grafos. Todos foram executados diretamente do código disponibilizado no *Google Colaboratory* do autor. Para avaliar as soluções destes algoritmos foram utilizadas as mesmas funções de *makespan* e *load balance* descritas na seção4.6, o que possibilita comparar os resultados destes algoritmos com os resultados do AG implementado.

5.3 Cenário 1: Grafo *robot*

O primeiro cenário dos experimentos foi com o grafo *robot*. Como descrito na seção 5.1, o GHE deste grafo contém 88 tarefas, 4 processadores, com $\Delta_{comp} = 20$ e $\Delta_{comu} = 30$. Neste cenário O AG foi executado dez vezes para cada valor de α da lista5.2, com os parâmetros descritos na lista5.2.

Tabela 1 – Resultados dos algoritmos determinísticos para o grafo *robot*

Algoritmo	***Makespan***	***Load Balance***
IPFET	1171	**1,49**
IHEFT	**1124**	1,51
CPOP	1284	1,63
HEFT	1369	1,73

Tabela 2 – Médias dos resultados do AG para o grafo *robot*

α	***Makespan***	***Load Balance***
0	849,56 ± 17,06	**1,01 ± 0,0**
0,25	822 ± 6,62	1,01 ± 0,01
0,50	826,78 ± 8,43	$\mathbf{1,01 \pm 0,0}$
0,75	$\mathbf{819,11 \pm 9,76}$	1,02 ± 0,01
1	827,22 ± 6,71	1,02 ± 0,01

Na tabela1estão os resultados das soluções de escalonamento dos programas determinísticos implementados pelo trabalho deCosta(2022). Na tabela2estão as médias dos resultados das dez execuções do AG para cada valor de α, com o respectivo desvio padrão. Nota-se que o AG, independente do α utilizado, teve valores de *makespan* e *load*

balance melhores do que os resultados dos algoritmos determinísticos. O pior resultado, em *makespan*, do AG ($\alpha = 0$) é melhor do que o melhor resultado dos algoritmos determinísticos, neste caso o IHEFT. Mesma coisa ao comparar o resultado do *load balance*. Observa-se também que os valores de *load balance* do AG foram bem menores do que nos algoritmos determinísticos e não variaram muito entre si. É possível verificar que, apesar de $\alpha = 1$ considerar apenas o valor de *makespan* como *fitness*, foram as execuções com $\alpha = 0,75$ e $\alpha = 0,25$ que demonstraram ter melhores resultados, indicando que um bom balanceamento de carga entre os processadores pode melhorar a métrica de *makespan*.

5.4 Cenário 2: Grafo *sparse*

Os experimentos do segundo cenário foram conduzidos com o grafo *sparse*, que contém 96 tarefas, 4 processadores, com $\Delta_{comp} = 100$ e $\Delta_{comu} = 30$. Os parâmetros do AG foram os mesmos utilizados no cenário 1. Abaixo estão as tabelas com os resultados das execuções do grafo com os algoritmos determinísticos e das médias dos resultados das dez execuções feitas para cada α usando o AG.

Tabela 3 – Resultados dos algoritmos determinísticos para o grafo *sparse*

Algoritmo	***Makespan***	***Load Balance***
IPFET	**1572**	1,4
IHEFT	1609	1,48
CPOP	1623	**1,34**
HEFT	1653	1,4

Tabela 4 – Médias dos resultados do AG para o grafo *sparse*

α	***Makespan***	***Load Balance***
0	1738,56 ± 54,48	$\mathbf{1,01 \pm 0,0}$
0,25	1554,56 ± 40,4	1,03 ± 0,01
0,50	1540,89 ± 48,18	1,01 ± 0,01
0,75	$\mathbf{1531,33 \pm 37,96}$	1,02 ± 0,01
1	1534,22 ± 27,46	1,03 ± 0,01

Nas tabelas3e4é possível perceber que os resultados do AG superam os resultados dos algoritmos determinísticos. Mas, diferente do cenário 1, o pior resultado de *makespan* do AG ($\alpha = 0$) foi pior do que todos os algoritmos determinísticos, nota-se também que essa configuração do AG teve o melhor resultado de balanceamento de cargas entre todas

as execuções. Os valores de *load balance* seguiram o que aconteceu no canário 1, com os resultados do AG sendo melhores. Neste cenário também se repetiu o que ocorreu no primeiro cenário, onde o melhor resultado de *makespan* veio de uma execução com $\alpha \neq 1$, neste cenário o melhor resultado veio da execução com $\alpha = 0,75$.

5.5 Cenário 3: Grafo *fpppp*

No último cenário estão os experimentos para o grafo *fpppp*. Este é o maior grafo testado, com 334 tarefas, 8 processadores, $\Delta_{comp} = 20$ e $\Delta_{comu} = 30$. Os parâmetros utilizados no AG foram os mesmos utilizados nos outros dois cenários.

Tabela 5 – Resultados dos algoritmos determinísticos para o grafo *fpppp*

Algoritmo	***Makespan***	***Load Balance***
IPFET	3605	**3,24**
IHEFT	3820	3,7
CPOP	3929	3,51
HEFT	**3569**	**3,24**

Tabela 6 – Médias dos resultados do AG para o grafo *fpppp*

α	***Makespan***	***Load Balance***
0	1352,56 ± 19,07	1,04 ± 0,01
0,25	1321,11 ± 17,15	1,03 ± 0,01
0,50	$\mathbf{1318,56 \pm 15,04}$	$\mathbf{1,02 \pm 0,01}$
0,75	1333,0 ± 25,37	1,04 ± 0,02
1	1326,56 ± 28,71	1,04 ± 0,02

Comparando as tabelas5e6, nota-se que neste cenário os resultados de *makespan* e balanceamento de carga das execuções do AG foram bem melhores que os resultados dos algoritmos determinísticos, com os resultados de *makespan* e *load balance* sendo, em média, 179% e 230% melhores no AG, respectivamente. Diferente dos outros cenários em que, apesar dos resultados do AG também terem sido melhores, não tiveram uma diferença tão expressiva nos resultados entre os dois tipos de algoritmos.

O melhor resultado do AG, tanto em *makespan*, quanto em *load balance*, foi obtido com $\alpha = 0,5$. A execução com $\alpha = 0$, que considera somente o *load balance*, teve um balanceamento de carga pior que as outras execuções e a execução com $\alpha = 1$, que considera apenas o *makespan*, teve um resultado pior que outras execuções. O que confirma

o observado nos cenários 1 e 2, onde um α equilibrado pelas duas métricas teve resultados melhores que os α que consideraram apenas uma das métricas de avaliação. Apesar da ventagem da execução com $\alpha = 0,5$, os resultados do AG não se diferiram muito em nenhuma das duas métricas. Com o pior resultado, em *makespan*, sendo da execução com $\alpha = 0$, fenômeno que também ocorreu nos outros dois cenários.

6 Conclusões

O uso cada vez mais intenso de computação de alto desempenho vem demandando cada vez mais a utilização de algoritmos de inteligência artificial. Este cenário faz com que a demanda por algoritmos de otimização também aumente, já que problemas que não tem solução em tempo polinomial estão cada vez mais frequentes (DOGAN; DOGAN; BOZKURT,2023;BRAGA et al.,2019;CARDOSO et al.,2023;AMARAL; GASPAROTTO, 2021). Para estes tipos de problemas, algoritmos que não garantem o melhor resultado possível, mas resultam em boas soluções em tempo hábil podem ser utilizados.

O problema do escalonamento de tarefas em ambientes multiprocessados heterogêneos é um problema classificado como NP Difícil, ou seja, o escalonamento perfeito levaria muito tempo para ser processado (DARBHA; AGRAWAL,1998). Como achar a solução ótima seria muito custoso, tanto em termos de computação, mas principalmente em termos de tempo, várias técnicas e algoritmos foram propostos para obter uma solução viável para o problema, esta solução não é necessariamente a melhor possível, mas é boa o bastante para economizar recursos computacionais e temporais.

Assim, os algoritmos genéticos surgem como uma boa opção para se aplicar ao problema do escalonamento de tarefas, sua característica de trabalhar com várias soluções ao mesmo tempo, permite que ele explore diferentes soluções do espaço de busca. Além disso, sua evolução guiada por boas soluções adaptadas faz com que o algoritmo tenda a conservar boas soluções. Neste trabalho foi implementado um AG que avalia as soluções por uma função ponderada pelos valores de *makespan* e *load balance* dos indivíduos. O objetivo foi simular as execuções de três DAGs de programas reais no AG implementado e em quatro algoritmos implementados no trabalho deCosta(2022) e comparar os resultados em termos das métricas de *makespan* e *load balance* entre os algoritmos.

Nos cenários descritos no capítulo5, foi possível perceber que os resultados das execuções do AG foram, em média, melhores do que nos algoritmos IPEFT, IHEFT, CPOP e HEFT. Nos cenários 1 e 3, os resultados do AG foram melhores nas duas métricas para todo valor de α testado, principalmente no cenário 3, onde os resultados do AG foram melhores, em média, 179% e 230% em *makespan* e *load balance*, respectivamente. O segundo cenário foi o cenário onde os resultados de *makespan* dos algoritmos determinísticos mais se aproximaram dos resultados do AG, com o IPEFT tendo o melhor resultado dentre os quatro algoritmos, com seu valor de *makespan* próximo aos resultados do AG quando considerado os desvios padrões. Na maioria das execuções do cenário 2 o AG se saiu melhor, porém a execução com $\alpha = 0$ teve um desempenho pior do que o desempenho dos algoritmos determinísticos, apesar de ter um balanceamento de carga

melhor. Algo observado nos três cenários é que execuções do AG com um α com valor equilibrado produz melhores resultados em *makespan* do que as execuções com $\alpha = 1$, que considera apenas o *makespan* para avaliar o indivíduo. Nos cenários 1 e 2, os melhores resultados do AG vierem com $\alpha = 0,75$, enquanto no cenário 3 o melhor resultado foi obtido com $\alpha = 0,5$. Algo que se confirmou observando os resultados dos três cenários foi que o AG produziu soluções muito melhores em termos de balanceamento de carga do que os algoritmos determinísticos, com todas as execuções do AG tendo resultados de *load balance* melhores do que todas as execuções dos quatro algoritmos, isso ficou ainda mais evidente no cenário 3, onde a média de *load balance* das execuções do AG foi de 1,034, enquanto a média dos algoritmos genéticos foi de 3,422.

Com os resultados obtidos nos três cenários, este trabalho demonstrou que algoritmos genéticos podem obter boas soluções para o problema do escalonamento de tarefas em um sistema multiprocessado heterogêneo e com custo de comunicação quando comparados aos algoritmos IPEFT, IHEFT, CPOP e HEFT, que são algoritmos determinísticos muito utilizados na literatura para este problema. Este trabalho também contribuiu com uma comparação de como performa os algoritmos genéticos e os quatro algoritmos determinísticos lidando com programas de aplicativos reais. Os parâmetros do AG foram arbitrariamente escolhidos observando os parâmetros especificados em outros trabalhos presentes na literatura e foram fixos, com exceção do parâmetro α, para todos os experimentos nos três cenários, o que pode ter impactado na qualidade das soluções, já que os parâmetros do AG não foram meticulosamente escolhidos considerando as características de cada grafo testado.

Este trabalho pode ser expandido e usado como referência para vários trabalhos futuros. Pode-se testar empiricamente os grafos reais utilizados variando os parâmetros do AG, a fim de encontrar os parâmetros para taxa de cruzamento, mutação, elitismo, número de iterações e tamanho da população que desempenham melhor em termos das métricas de avaliação. É possível também implementar outras funções de aptidão para avaliar os indivíduos do AG, como as métricas de *flowtime* e confiabilidade, descritas no trabalho deSilva(2020). Pode-se também avaliar o desempenho do AG testando outros grafos, de programas reais ou não, ou variações dos grafos testados neste trabalho, contanto que estejam no formato GHE. Existem diversos operadores genéticos de cruzamento e mutação para o problema do escalonamento de tarefas que não foram implementados neste trabalho, pode-se também implementá-los e testá-los com o conjunto de grafos testados neste trabalho, a fim de comparar os resultados e analisar quais operadores obtiveram resultados melhores. Operadores como a Recombinação de Pontos Aleatórios (RPX), Recombinação de Dois Pontos (TX), Mutação de Deslocamento de Tarefas (STM) e Mutação de Probabilidade Aleatória (RPM) são exemplos possíveis de implementar usando a codificação de alocação e escalonamento (MSE) utilizada neste trabalho. Pode-se também usar resultados e grafos deste trabalho para comparar com outros algoritmos,

como os algoritmos genéticos híbridos, que utilizam heurísticas e meta-heurísticas com objetivo de melhorar a evolução das gerações do AG.

O código desenvolvido do AG e dos experimentos, com os grafos reais originais no formato STG e dos grafos utilizados nos experimentos, pode ser encontrado no repositório do *Github* [1].

[1] Disponível em<https://github.com/johnsigma/IC-AG/tree/tcc>

Referências

ALEBRAHIM, S.; AHMAD, I. Task scheduling for heterogeneous computing systems. **The Journal of Supercomputing**, Springer, v. 73, p. 2313–2338, 2017. Citado na página20.

AMARAL, H. N.; GASPAROTTO, A. M. S. inteligência artificial: o uso da robótica indústria 4.0. **Revista Interface Tecnológica**, v. 18, n. 1, p. 474–486, 2021. Citado na página48.

BITTENCOURT, L. F.; GOLDMAN, A.; MADEIRA, E. R.; FONSECA, N. L. da; SAKELLARIOU, R. Scheduling in distributed systems: A cloud computing perspective. **Computer science review**, Elsevier, v. 30, p. 31–54, 2018. Citado na página17.

BRAGA, A. V.; LINS, A. F.; SOARES, L. S.; FLEURY, L. G.; CARVALHO, J. C.; PRADO, R. S. do. Machine learning: o uso da inteligência artificial na medicina. **Brazilian Journal of Development**, v. 5, n. 9, p. 16407–16413, 2019. Citado na página48.

CARDOSO, F. S.; PEREIRA, N. da S.; BRAGGION, R. C.; CHAVES, P.; ANDRIOLI, M. O uso da inteligência artificial na educação e seus benefícios: uma revisão exploratória e bibliográfica. **Revista Ciência em Evidência**, v. 4, n. FC, p. e023002–e023002, 2023. Citado na página48.

COELLO, C. A.; LAMONT, G. B.; VELDHUIZEN, D. A. V. **Evolutionary Algorithms for Solving Multi-Objective Problems**. Springer New York, NY, 2007. Disponível em:<https://link.springer.com/book/10.1007/978-0-387-36797-2>. Citado na página12.

COSTA, B. C. S. **Avaliação de algoritmos de escalonamento de aplicações paralelas em processadores heterogêneos**. Uberlândia: [s.n.], 2022. 56 p. Citado 8 vezes nas páginas9,18,20,21,23,24,44e48.

DARBHA, S.; AGRAWAL, D. P. Optimal scheduling algorithm for distributed-memory machines. **IEEE transactions on parallel and distributed systems**, IEEE, v. 9, n. 1, p. 87–95, 1998. Citado na página48.

DHINGRA, S.; GUPTA, S. B.; BISWAS, R. Genetic algorithm parameters optimization for bi-criteria multiprocessor task scheduling using design of experiments. **International Journal of Computer, Control, Quantum and Information Engineering**, v. 8, n. 4, p. 661–667, 2014. Citado na página34.

DOGAN, M. E.; DOGAN, T. G.; BOZKURT, A. The use of artificial intelligence (ai) in online learning and distance education processes: A systematic review of empirical studies. **Applied Sciences**, MDPI, v. 13, n. 5, p. 3056, 2023. Citado na página48.

EIBEN, A.; SMITH, J. **Introduction to Evolutionary Computing**. Springer Berlin, Heidelberg, 2015. Disponível em:<https://link.springer.com/book/10.1007/978-3-662-44874-8>. Citado na página16.

GOLDBERG, D. E.; DEB, K. Foundations of genetic algorithms. In: RAWLINS, G. J. (Ed.). **A Comparative Analysis of Selection Schemes Used in Genetic Algorithms**. Elsevier, 1991, (Foundations of Genetic Algorithms, v. 1). p. 69–93. Disponível em:<https://www.sciencedirect.com/science/article/pii/B9780080506845500082>. Citado na página13.

GOLUB, M.; KASAPOVIC, S. Scheduling multiprocessor tasks with genetic algorithms. In: CITESEER. **APPLIED INFORMATICS-PROCEEDINGS-**. [S.l.], 2002. p. 273–278. Citado na página16.

GRAHAM, R. L. Bounds for certain multiprocessing anomalies. **Bell system technical journal**, Wiley Online Library, v. 45, n. 9, p. 1563–1581, 1966. Citado na página21.

GUPTA, S.; KUMAR, V.; AGARWAL, G. Task scheduling in multiprocessor system using genetic algorithm. In: IEEE. **2010 Second international conference on machine learning and computing**. [S.l.], 2010. p. 267–271. Citado na página34.

HOLLAND, J. **Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence**. Oxford, England: U Michigan Press, 1975. Citado 3 vezes nas páginas11,16e24.

HOU, E. S.; ANSARI, N.; REN, H. A genetic algorithm for multiprocessor scheduling. **IEEE Transactions on Parallel and Distributed systems**, IEEE, v. 5, n. 2, p. 113–120, 1994. Citado 2 vezes nas páginas21e22.

JELODAR, M.; FAKHRAIE, S.; MONTAZERI, F.; FAKHRAIE, S.; AHMADABADI, M. A representation for genetic-algorithm-based multiprocessor task scheduling. In: **2006 IEEE International Conference on Evolutionary Computation**. [s.n.], 2006. p. 340–347. Disponível em:<https://ieeexplore.ieee.org/document/1688328>. Citado na página12.

LEE, Y.-H.; CHEN, C. A modified genetic algorithm for task scheduling in multiprocessor systems. In: **the 9th workshop on compiler techniques for high-performance computing**. [S.l.: s.n.], 2003. Citado na página34.

OMARA, F.; ARAFA, M. Genetic algorithms for task scheduling problem. **J. Parallel Distrib. Comput.**, v. 70, p. 13–22, 01 2010. Citado 2 vezes nas páginas9e34.

PADUA, D. **Encyclopedia of parallel computing**. [S.l.]: Springer Science & Business Media, 2011. Citado na página17.

PEDERNEIRAS, G. **Cloud, ou computação em nuvem, na indústria 4.0**. 2019. Disponível em:<https://www.industria40.ind.br/artigo/17984-cloud-ou-computacao-em-nuvem-na-industria-40>. Acesso em: 26 de fevereiro 2024. Citado na página8.

SANTOS, J. F. Algoritmos evolutivos multiobjetivo baseados em tabelas para escalonamento de tarefas em ambientes multiprocessados. Universidade Federal de Uberlândia, 2023. Citado na página8.

SANTOS, S. **Computação de alto desempenho: essencial na vida diária e o futuro da tecnologia**. 2021. Disponível em:<https://canaltech.com.br/colunas/computacao-de-alto-desempenho-essencial-na-vida-diaria-e-o-futuro-da-tecnologia/>. Citado na página8.

SHARMA, N. **Static list scheduling for DAGs: A comparative study between algorithms.** 2019. Acesso em: 12 abril 2024. Disponível em:<https://github.com/sharma-n/DAG_Scheduling>. Citado na página21.

SILVA, E. C. da. **Representações de Algoritmos Genéticos para o Problema de Escalonamento Estático de Tarefas em Multiprocessadores**. Uberlândia: [s.n.], 2020. 231 p. Citado 10 vezes nas páginas8,12,13,14,15,16,21,24,32e49.

TOPCUOGLU, H.; HARIRI, S.; WU, M.-Y. Performance-effective and low-complexity task scheduling for heterogeneous computing. **IEEE transactions on parallel and distributed systems**, IEEE, v. 13, n. 3, p. 260–274, 2002. Citado na página20.

WANG, L.; SIEGEL, H. J.; ROYCHOWDHURY, V. P.; MACIEJEWSKI, A. A. Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach. **Journal of parallel and distributed computing**, Elsevier, v. 47, n. 1, p. 8–22, 1997. Citado 2 vezes nas páginas25e34.

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 Online Library, v. 29, n. 5, p. e3944, 2017. Citado na página20.

ZOMAYA, A.; WARD, C.; MACEY, B. Genetic scheduling for parallel processor systems: comparative studies and performance issues. **IEEE Transactions on Parallel and Distributed Systems**, v. 10, n. 8, p. 795–812, 1999. Citado na página15.