UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Yuri Cardoso Santamarina

Balanceamento de carga em escalonamento de tarefas baseado em *multi-commodity flow*

Uberlândia, Brasil

2019

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Yuri Cardoso Santamarina

Balanceamento de carga em escalonamento de tarefas baseado em *multi-commodity flow*

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

Orientador: Paulo Henrique Ribeiro Gabriel

Universidade Federal de Uberlândia – UFU

Faculdade de Computação

Bacharelado em Sistemas de Informação

Uberlândia, Brasil

2019

Agradecimentos

Agradeço ao meu orientador Paulo Henrique pelo apoio durante todo o trabalho, e principalmente pela paciência em momentos que precisei deixar um pouco de lado, e também nos momentos que precisei acelerar o desenvolvimento.

Agradeço a minha família, namorada e amigos pelo apoio.

Por fim, agradeço às maratonas de programação. Tudo que foi feito neste trabalho não teria sido possível sem o conhecimento que eu adquiri durante o tempo em que me dediquei a elas.

Resumo

O escalonamento de tarefas é muito comum em sistemas computacionais. A todo momento os sistemas que executam em celulares, computadores e *tablets* utilizam algum tipo de escalonamento. O escalonamento pode ter vários objetivos, dentre eles, o de minimizar o *makespan*, ou então distribuir o mais equitativamente a carga entre os processadores disponíveis, *i.e.*, balancear a carga entre eles. Este Trabalho de Conclusão de Curso tem como objetivo propor um algoritmo *offline* para balanceamento de carga no contexto de escalonamento de tarefas utilizando como base um conceito da área de Fluxo em Redes, o *Multi-commodity Flow*. Objetiva-se também compará-lo a outro algoritmo já existente, o EDF. Para analisar os resultados de ambos, diversos casos aleatórios foram gerados e executados. Algumas métricas foram calculadas a partir disto, no caso, a utilização máxima, utilização média, *makespan* e o tempo de execução. Resultados apresentados pelo algoritmo proposto (Balanceador) foram bons quando comparados ao EDF; além disso, em muitos dos casos, o algoritmo proposto atingiu um balanceamento ótimo, ou seja, todos os processadores executaram a mesma carga de trabalho.

Palavras-chave: Escalonamento de Tarefas, Balanceamento de Carga, Teoria dos Grafos, Fluxo em Redes, *Multi-commodity Flow*.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

U_{med}	Utilização média
U_{max}	Utilização máxima
n	Quantidade de tarefas
m	Quantidade de processadores
K	Soma de todas as demandas
T_{exec}	Tempo de execução
U_j	Utilização de um processador
L_j	Último tempo que o processador j foi utilizado
EDF	Algoritmo *Earliest Deadline First*

Sumário

1 Introdução

1.1 Contextualização e Motivação

Grafos são estruturas compostas por vértices e arestas que representam relações entre objetos (BONDY; MURTY, 2008). Diversos problemas do mundo real podem ser modelados e estudados por meio de grafos. Nesse sentido, a Teoria dos Grafos é uma área interdisciplinar, envolvendo conceitos matemáticos e aspectos computacionais, que busca compreender aspectos teóricos dessas estruturas, bem como sua implementação computacional.

Um grande problema da área de Teoria dos Grafos é o problema de *fluxo em redes* (CORMEN et al., 2012; GOLDBARG, 2014): dado um grafo no qual cada aresta tem uma capacidade máxima, deseja-se transferir certa quantidade de fluxo entre uma origem e um destino, respeitando a quantidade máxima de fluxo que pode passar em cada aresta. Um problema relacionado a esse é o de encontrar o fluxo *máximo* de uma rede, ou seja, a quantidade máxima de unidades de fluxo que podem ser transferidas entre uma origem e um destino em uma unidade de tempo.

Uma variação clássica do problema de fluxo em redes é o chamado *multi-commodity flow problem* (MCFP) (AWERBUCH; LEIGHTON, 1994), segundo o qual existem diversas origens e diferentes destinos (em vez de apenas uma origem e um destino, como no problema de fluxo tradicional) e o objetivo é distribuir a carga entre as arestas do grafo, na tentativa de não sobrecarregar algumas delas. Outro objetivo bastante estudado é a minimização do custo de transportar o fluxo, sendo que, além da capacidade, cada aresta tem um custo associado ao transporte de uma unidade de fluxo.

Diversos problemas do mundo real podem ser modelados por meio de grafos e tratados como problemas de fluxo em redes (MEKKITTIKUL; MCKEOWN, 1998; EVEN; TARJAN, 1975; HAGHANI; OH, 1996). Dentre esses, este trabalho de conclusão de curso foca no problema de escalonamento de tarefas (ALBERS, 2003; GRAHAM, 1969), o qual consiste, em sua ideia original, em atribuir tarefas a máquinas, de modo a minimizar o tempo máximo de execução de todas as tarefas (o chamado *makespan*).

O problema de escalonamento tem sido modelado como um problema de fluxo em redes, geralmente com o objetivo de minimizar o *makespan*. No entanto, diversas outras métricas de desempenho têm sido estudadas na literatura de escalonamento de tarefas. Dentre essas métricas alternativas, destaca-se o balanceamento de carga, cujo objetivo não é minimizar o tempo de execução, mas sim *maximizar* a utilização média das máquinas e, consequentemente, reduzir a ociosidade das mesmas.

Nesse sentido, o *multi-commodity flow problem* pode ser utilizado como base para o desenvolvimento de algoritmos para essa variação do problema de escalonamento de tarefas. No entanto, a literatura na área ainda traz poucos estudos nesse sentido. Assim, esse trabalho visa tratar essa lacuna, buscando uma nova modelagem do problema de balanceamento de carga, bem como sua resolução por meio de algoritmos eficientes.

1.2 Objetivos

O principal objetivo deste trabalho de conclusão de curso é desenvolver um algoritmo para o problema do balanceamento de carga em escalonamento de tarefas, utilizando como base o conceito do *multi-commodity flow problem* para modelar o problema.

Além do mais, objetiva-se comparar o escalonamento baseado em tempo de execução mínimo com o escalonamento baseado em balanceamento de carga, contrastando seus objetivos centrais, ou seja, a minimização do tempo de execução das tarefas e a minimização da ociosidade das máquinas, respectivamente. Cada algoritmo possui suas particularidades e objetivos e também resolvem problemas parecidos, porém, com abordagens diferentes.

1.3 Desenvolvimento

Dois algoritmos foram implementados e diversas situações com tarefas e processadores foram criadas aleatoriamente a partir de um gerador de casos de teste. Esses algoritmos foram avaliados empiricamente em tais situações e os resultados foram tabelados para que se possa compará-los da melhor forma possível, sempre considerando os propósitos de cada um.

O primeiro algoritmo foi criado e implementado especificamente para este trabalho e utiliza como base um algoritmo de fluxo máximo em grafos (DINITZ, 1970; DINITZ, 2006) com uma busca binária e visa balancear a carga entre processadores. Isso é o equivalente a minimizar a maior utilização de um processador.

O segundo algoritmo foi implementado como uma variação do *Earliest Deadline First* (EDF) para multiprocessadores e utiliza basicamente duas filas de prioridade, uma para os processadores e outra para as tarefas e visa minimizar o *makespan*. O EDF é provado ser ótimo para situações com um único processador. No entanto, segundo Srinivasan e Baruah (2002), não há garantias que ele ache a resposta ótima para situações com múltiplos processadores.

Neste trabalho também foi implementado um gerador de casos de teste com o objetivo de auxiliar na criação das situações em que os algoritmos foram avaliados.

Quanto aos resultados, o algoritmo desenvolvido neste trabalho (Balanceador) cumpriu seu objetivo e, em diversas situações, apresentou um balanceamento ótimo, enquanto que o EDF não conseguiu o mesmo resultado.

1.4 Organização da Monografia

Esta monografia está organizada da seguinte forma: no Capítulo 2, são apresentados os conceitos necessários para o desenvolvimento deste trabalho, bem como algumas referências a trabalhos relacionados. O Capítulo 3 trata do desenvolvimento em si, trazendo descrições sobre os dois algoritmos implementados neste trabalho e, também, as métricas utilizadas na comparação entre eles. Os resultados são apresentados e discutidos no Capítulo 4. Por fim, o Capítulo 5 contém as considerações finais, bem como algumas ideias para trabalhos futuros a serem feitos sobre este tema.

2 Referencial Teórico

Este capítulo introduz os conceitos base para o entendimento completo do trabalho, ou seja, são apresentadas definições sobre grafos, sobre o problema de escalonamento de tarefas e dos algoritmos necessários na implementação da proposta. Também são descritos alguns trabalhos correlatos.

2.1 Conceitos Adotados

2.1.1 Grafos

Um grafo é uma estrutura $G(V, E)$, sendo que V é o conjunto dos vértices e E é o conjunto das arestas e pode ser utilizado para representar relações (arestas) entre objetos (vértices) (BONDY; MURTY, 2008). Relações de amizades entre um conjunto de pessoas é um exemplo de utilização de grafos, onde as pessoas são o vértices e as relações de amizade são as arestas, ou seja, uma aresta entre A e B indica que A é amigo de B e B é amigo de A. Nesse caso as arestas são chamadas bidirecionais, pois se há uma aresta de A para B, com certeza tem uma aresta de B para A.

Um outro tipo de aresta são as unidirecionais, que vão apenas em um sentido e podem ser utilizadas para representar ruas de mão única em uma cidade, por exemplo, ou qualquer relação que não seja recíproca. Um grafo onde as arestas são unidirecionais ou direcionadas, é chamado de dígrafo, ou grafo direcionado.

2.1.2 Grafo Bipartido

Dentro do conceito de Grafos, existe um tipo específico de grafo chamado bipartido. Grafos bipartidos são grafos em que é possível separar os vértices em dois conjuntos distintos tal que nenhuma aresta do grafo conecta dois vértices de um mesmo conjunto, i.e., os dois vértices de uma aresta estão sempre em grupos distintos. Tecnicamente falando, todo grafo que não possui um ciclo de tamanho ímpar é bipartido.

A Figura 1 ilustra um grafo bipartido qualquer.

2.1.3 Escalonamento de Tarefas

O problema de escalonamento de tarefas (ALBERS, 2003) é o problema de atribuir tarefas a máquinas de modo a satisfazer algum objetivo central, como o de minimizar o *makespan*, ou então minimizar a utilização máxima de uma máquina, desde que todas as tarefas sejam atendidas.

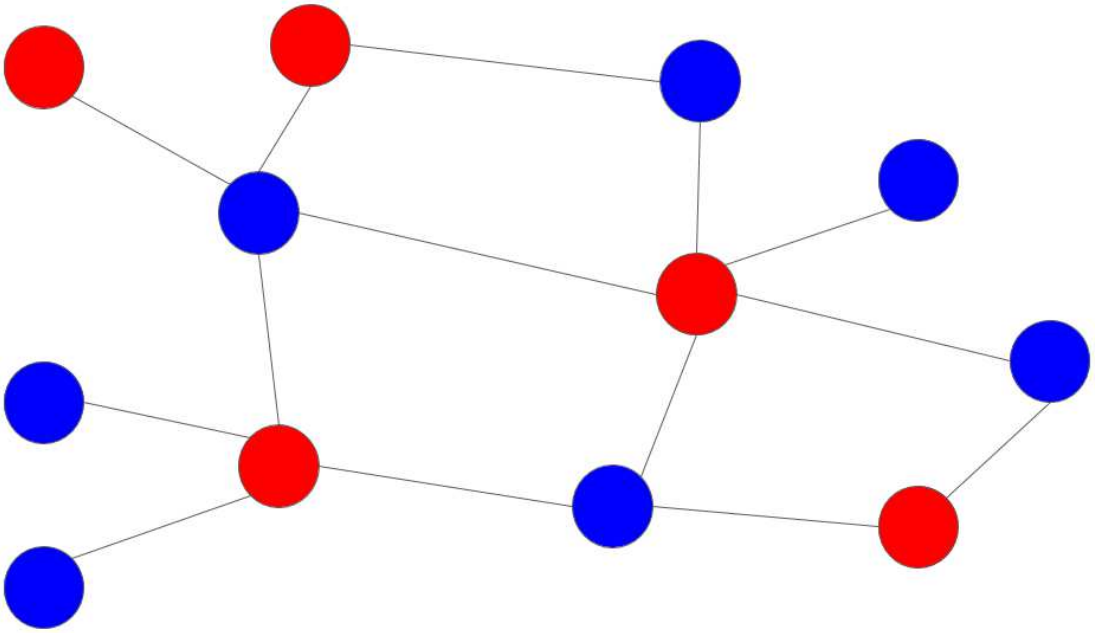

Figura 1 – Grafo bipartido genérico

A ideia do escalonamento de tarefas também pode ser utilizada em outros tipos de situações, como em uma empresa que quer contratar funcionários para determinados cargos. Para isso, os candidatos se inscrevem nas vagas que desejam ocupar e a empresa irá selecionar os candidatos de tal forma que cada um seja alocado a no máximo um cargo, cada cargo seja ocupado por exatamente uma pessoa e que todas as vagas sejam ocupadas.

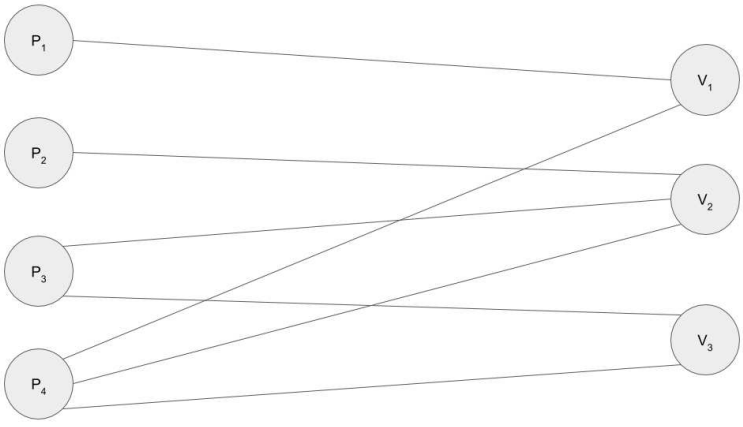

Figura 2 – Ideia do Escalonamento entre pessoas e vagas de emprego

A solução do problema acima é dada pelo *matching* (BONDY; MURTY, 2008) máximo, ou seja, o maior número de pares (pessoa, vaga) que podem ser formados. No caso da Figura 3, a resposta é 3 e os pares são: (P_1, V_1), (P_3, V_3) e (P_4, V_2).

Como o grafo é bipartido, o problema pode ser resolvido utilizando qualquer algoritmo de fluxo máximo. A Figura 4 ilustra como o grafo deve ser construído (todas as

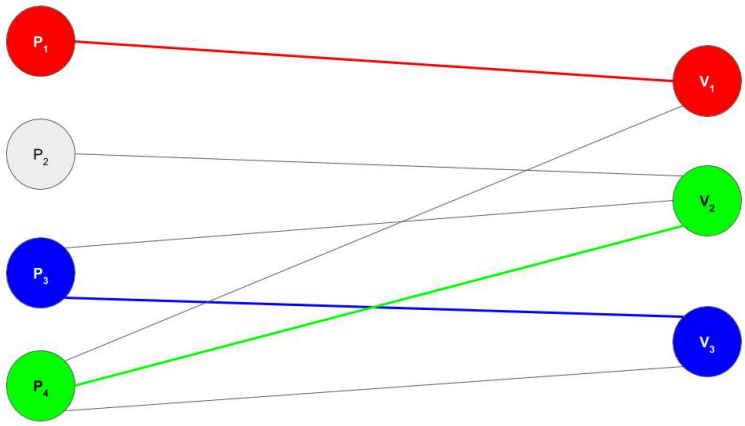

Figura 3 – Atribuição ótima das pessoas às vagas de emprego

arestas possuem capacidade 1, portanto, foram omitidas da imagem). Após a construção do grafo, basta apenas encontrar o fluxo máximo entre *src* e *sink*, que é exatamente o valor do *matching* máximo.

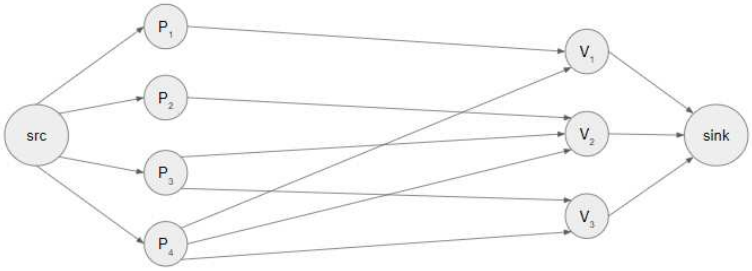

Figura 4 – Grafo modelado para o *matching* máximo

2.1.4 Fluxo em Redes

O problema de Fluxo em Redes (BONDY; MURTY, 2008; CORMEN et al., 2012) é um problema da Teoria dos Grafos em que as arestas além de representarem a relação entre os dois vértices, possui também uma capacidade máxima $c_i(u, v)$ de utilização que deve ser respeitada, ou seja, se $f_i(u, v)$ é o fluxo que passou na aresta i, $f_i(u, v)$ deve ser menor ou igual à $c_i(u, v)$.

Uma aplicação de Fluxo em Redes é a de encontrar o fluxo máximo que é possível passar de uma origem até um destino. Um exemplo dessa aplicação do fluxo pode ser visto em uma rede hidráulica com a origem sendo a central de distribuição de água e o destino sendo alguma casa. O objetivo é calcular a quantidade máxima de litros de água que pode ser transportada entre a origem e o destino por segundo, sabendo que cada cano da rede possui uma capacidade máxima de litros de água que pode suportar sem estourar.

2.1.5 *Multi-commodity flow problem*

O *Multi-commodity flow problem* (AWERBUCH; LEIGHTON, 1994) é uma variação do problema de fluxo em que se tem várias demandas de fluxo, cada uma com origem, destino e quantidade de fluxo a ser transportada.

Dentro do *Multi-commodity flow problem* a variante em que baseia este trabalho é a do balanceamento de carga (*Load Balancing*). O balanceamento de carga é a tentativa de distribuir o mais equitativamente possível algum tipo de carga. No caso deste trabalho a carga é a quantidade de tempo que todas as tarefas gastam para executar, e o objetivo é distribuir tal carga entre os diversos processadores disponíveis.

Idealmente, o balanceamento ótimo é aquele em que todos os processadores são utilizados a mesma quantidade de tempo, porém, nem sempre isso ocorre, restando então, chegar o mais próximo possível deste balanceamento ótimo. O balanceamento de carga é equivalente à maximizar a utilização média dos processadores, o que pode ser atingido minimizando a utilização máxima dos processadores.

2.1.6 Earliest Deadline First - EDF

O EDF é um algoritmo de escalonamento de tarefas com prioridade dinâmica. A prioridade de cada uma das tarefas é atribuída de acordo com o tempo em que a tarefa irá expirar, o chamado *deadline*, garantindo que as tarefas com menor *deadline* tenham maior prioridade, i.e., tarefas mais próximas de expirar devem executar primeiro.

Existem implementações do EDF tanto para uniprocessadores quanto para multiprocessadores em situações que a preempção é permitida, porém, apenas em uniprocessadores o EDF é provado (SRINIVASAN; BARUAH, 2002) que consegue escalonar todas as tarefas, i.e., caso seja possível, de alguma forma, escalonar todas as tarefas, o EDF irá encontrar um meio de realizar, enquanto que para multiprocessadores esse escalonamento nem sempre ocorre.

O EDF também executa em sistemas em que as tarefas são periódicas (SRINIVASAN; BARUAH, 2002). As tarefas periódicas geram sub-tarefas nos instantes de tempo múltiplos do seu período, e entre dois múltiplos consecutivos essa sub-tarefa deve executar uma quantidade de tempo igual à sua demanda de execução.

2.1.7 Busca binária

Busca binária é um algoritmo de busca para encontrar um determinado valor dentro de um conjunto de dados ordenados. O algoritmo sucessivamente diminui o espaço de busca pela metade, até que o elemento seja encontrado ou então até que se constate que ele não existe nos dados. Devido à essa característica, a complexidade de tempo do

algoritmo é de $O(\log(n))$, em que n é o tamanho do conjunto de dados.

O algoritmo 1 resolve o seguinte problema: dado um vetor de inteiros ordenado de forma crescente, cheque se "valor" está presente no vetor.

Algoritmo 1: Pseudocódigo da Busca Binária para encontrar um elemento em um vetor ordenado

```
Entrada: Vetor de inteiros; valor a ser pesquisado
Saída: True caso o valor exista no vetor, False caso contrário
início
    lim_inferior = 0
    lim_superior = vetor.tamanho() - 1
    enquanto lim_inferior <= lim_superior faça
        meio = (lim_inferior + lim_superior) / 2
        se vetor[meio] == valor então
            retorna True
        senão se vetor[meio] > valor então
            lim_superior = meio - 1
        senão
            lim_inferior = meio + 1
        fim
    fim
fim
retorna False
```

O princípio que faz a busca binária funcionar é de que os dados devem estar ordenados de forma crescente ou decrescente, pois assim, é possível saber após um "chute", em qual parte do vetor o elemento que estou buscando possivelmente pode estar.

A aplicação trivial do algoritmo mostrada acima utiliza os dados presentes em um vetor, mas isso não é um requisito, tais dados podem ser "imaginários", como no seguinte problema: dado um valor inteiro positivo T, ache o maior valor inteiro x, tal que $x^2 \leq T$. Tal problema é facilmente resolvido pegando a parte inteira de $\sqrt{T}$, porém, para fins de ilustração, será resolvido com busca binária.

A primeira observação a ser feita é de que se for escolhido um número x e o valor x^2 for maior que T, o x não é a resposta do problema e com certeza nenhum outro valor maior que x poderá ser a resposta, sendo assim, apenas valores menores que x devem ser testados nas próximas iterações. Caso $x^2 \leq T$, o x é potencialmente a resposta do problema, mas ainda não é possível ter certeza se ele é o maior possível, portanto, a busca deve continuar até que o espaço de busca acabe e nesse ponto, o último valor x testado em que $x^2 \leq T$ é a resposta do problema.

A palavra "imaginário" foi utilizada porque no problema acima não há nenhum vetor ou qualquer outra estrutura de dados em que o dado deve ser buscado.

O algoritmo 2 implementa a ideia descrita acima.

Algoritmo 2: Pseudocódigo da Busca Binária para encontrar o valor inteiro de $\sqrt{T}$

```
Entrada: T
Saída: x
início
    lim_inferior = 1
    lim_superior = T
    x = 0
    enquanto lim_inferior <= lim_superior faça
        meio = (lim_inferior + lim_superior) / 2
        se meio * meio <= T então
            x = meio
            lim_inferior = meio + 1
        senão
            lim_superior = meio - 1
        fim
    fim
fim
retorna x
```

A complexidade de tempo do algoritmo é $O(\log(T))$.

2.2 Trabalhos Correlatos

Esta seção trata de alguns trabalhos relacionados com o tema central deste trabalho, o balanceamento de carga em escalonamento de tarefas baseado em *multi-commodity flow*. Serão apresentados trabalhos sobre Escalonamento de tarefas, Fluxo em Grafos, *multi-commodity flow* e EDF.

Haghani e Oh (1996) utilizam o conceito do *multi-commodity flow problem* para formular e resolver o problema de assistência em desastres. Em situações de emergência diversos problemas são enfrentados, como o problema de decidir de que forma serão transportadas as *commodities* (alimentos, remédios, roupas e outros utensílios vitais) de diversas origens para as diversas áreas afetadas pelo desastre de forma a minimizar a perda de vidas e o custo do transporte. Os autores se basearam em heurísticas para apresentarem um solução que ajuda na tomada de decisão em possíveis situações de emergência.

Ford Jr e Fulkerson (2010) publicaram um método guloso para a resolução do problema de fluxo máximos em redes, onde cada aresta tem uma capacidade e o objetivo é maximizar o fluxo que pode ser transportado entre uma origem e um destino predefinidos. O método utiliza o conceito de *augmenting paths*, ou caminhos aumentantes, que consistem em caminhos entre a origem e o destino, no grafo residual, em que todas as arestas do caminho não estão saturadas, ou seja, possuem capacidade positiva.

O método denominado *Ford-Fulkerson Algorithm* consiste em encontrar sucessivos

caminhos aumentantes e a cada iteração, diminuir a capacidade das arestas do caminho em X, onde X é a menor capacidade das arestas do caminho, o chamado gargalo (do inglês, *bottleneck*). O algoritmo termina quando não é possível mais encontrar caminho aumentante, que ocorre quando todos os caminhos entre a origem e o destino no grafo residual possuem pelo menos uma aresta saturada. O fluxo máximo encontrado pelo algoritmo é igual à soma de todos os gargalos encontrados durante a execução.

Srinivasan e Baruah (2002) desenvolveram um algoritmo com base no EDF para escalonar tarefas periódicas em multiprocessadores. Na situação considerada pelos autores, cada tarefa possuía dois atributos: uma exigência de execução C_i (neste trabalho chamaremos sempre de demanda K_i) e um período T_i. Tais atributos significam que a cada tempo múltiplo de T_i, uma nova sub-tarefa era gerada e tinha que executar exatamente C_i unidades de tempo entre esse tempo e o tempo imediatamente anterior ao próximo múltiplo de T_i, que é quando uma nova sub-tarefa será gerada. Os autores alteraram a forma como o EDF atribui prioridades às tarefas para que o algoritmo resolvesse de forma mais eficiente o problema que eles buscavam a solução.

3 Desenvolvimento

Neste trabalho, foram consideradas situações com n tarefas T_i e m processadores P_j idênticos. Uma tarefa $T_i = (B_i, D_i, K_i)$ é caracterizada por três parâmetros: B_i - o primeiro tempo em que a tarefa está disponível para ser executada; D_i - o último tempo que a tarefa pode ser executada, também chamado de *deadline*; K_i - a quantidade de tempo que a tarefa deve executar, também chamado de demanda.

Cada situação considerada seguiu algumas restrições:

- Cada tarefa deve executar exatamente K_i segundos, não importando em qual processador.
- Uma tarefa pode executar em mais de um processador em diferentes instantes de tempo.
- Uma tarefa não pode executar em mais de um processador em um mesmo instante de tempo.
- Uma tarefa pode ter sua execução interrompida (preempção) a qualquer instante inteiro de tempo.
- Cada processador pode executar no máximo uma tarefa em um mesmo instante de tempo.
- Todos os m processadores são idênticos.

Ambos os algoritmos implementados neste trabalho - Balanceador de Carga e o Earliest Deadline First - produzem como saída as métricas que serão utilizadas na comparação entre eles: utilização máxima ($U_{\max}$), utilização média (U_{med}), *makespan* e tempo de execução (T_{exec}).

3.1 Algoritmos Implementados

3.1.1 Balanceador de carga

O Balanceador de Carga desenvolvido para este trabalho realiza o escalonamento das tarefas visando não sobrecarregar um processador, ou seja, ele tem o objetivo de minimizar a maior utilização U_j de um processador. A utilização U_j de um processador é o tempo total que ele foi utilizado para executar as tarefas. Para minimizar o maior U_j, foi utilizado o algoritmo de Dinic (DINITZ, 1970; DINITZ, 2006) para fluxo máximo em grafos juntamente com uma busca binária.

Modelagem

Cada situação foi modelada como sendo um grafo direcionado com um valor em cada uma das arestas. Cada tarefa T_i, processador P_j e instante de tempo Te_k são representados como vértices do grafo. Além deles, dois outros vértices auxiliares chamados de fonte (src) e sumidouro ($sink$), também foram criados pois são necessários para a execução do algoritmo de fluxo máximo.

As arestas foram dispostas da seguinte forma:

- Aresta de src para todas as n tarefas T_i com capacidade K_i.
- Aresta de T_i para cada instante de tempo Te_k no intervalo $[Ii, Fi]$ com capacidade 1.
- Aresta de todos os instantes de tempo Te_k para todos os m processadores P_j com capacidade 1.
- Aresta de P_j para o $sink$ com capacidade X.

As arestas dos processadores P_j para o $sink$ não são importantes agora, portanto foram descritas com capacidade X. Tais arestas terão a capacidade alterada pela busca binária durante a execução do algoritmo.

Todas as arestas descritas acima possuem uma aresta reversa com capacidade 0, ou seja, se foi adicionada uma aresta de A para B com capacidade C, obrigatoriamente deve-se adicionar uma aresta de B para A com capacidade 0. Tais arestas são necessárias para que o Algoritmo de Dinic execute de forma correta. Elas permitem que o algoritmo desfaça alguma decisão ruim já feita e servem para marcar situações do tipo: "já foi passado C de fluxo na aresta que vai de A até B, então vou marcar que posso passar C de fluxo de B para A para desfazer a ação de passar o fluxo de A para B que já foi feita".

A Figura 5 mostra a modelagem um caso com 3 tarefas e 2 processadores, com as seguintes características:

1. $T_1 = (I_1 = 2,\ F_1 = 3,\ D_1 = 2)$
2. $T_2 = (I_2 = 1,\ F_2 = 3,\ D_2 = 1)$
3. $T_3 = (I_3 = 3,\ F_3 = 4,\ D_3 = 1)$

Todas as arestas que não possuem capacidade na imagem são arestas de capacidade igual a 1. Elas foram omitidas pois são fixas independente do caso de teste.

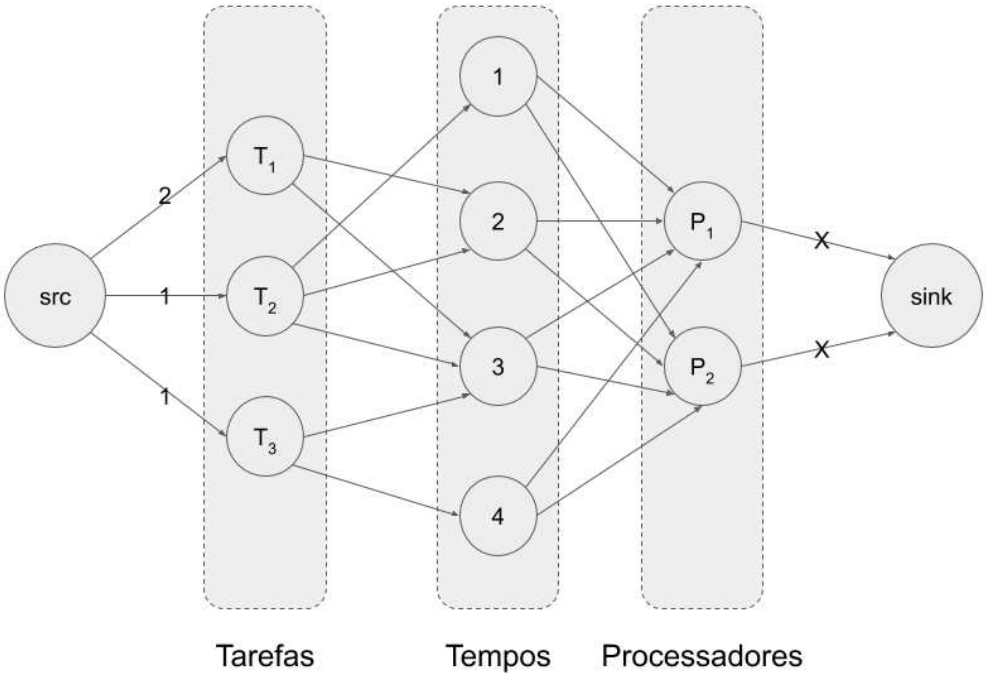

Figura 5 – Modelagem do grafo para a execução do Balanceador

Papel do algoritmo de Fluxo Máximo

Dado que o grafo já está construído, a quantidade de fluxo que passa por um processador é no máximo X, o que faz $U_j \leq X$. O algoritmo de fluxo máximo irá achar a maior quantidade de fluxo possível que pode ser passada entre o *src* e o *sink*, respeitando as capacidades de cada aresta.

É possível escalonar todas as tarefas nessa situação caso a quantidade de fluxo passada por cada tarefa T_i seja igual à sua demanda K_i, ou seja, as arestas de *src* para todas as tarefas T_i devem estar saturadas. Como as arestas entre *src* e T_i possuem capacidade K_i, o fluxo máximo entre *src* e *sink* nunca será maior que soma de todas as demandas K_i pois caso fosse, pelo menos uma aresta teria tido sua capacidade estourada. Chamarei de K a soma de todos as demandas K_i deste ponto em diante.

Portanto, sabe-se que é possível satisfazer as demandas de todas as tarefas caso o fluxo máximo entre *src* e *sink* seja exatamente K. Porém, o algoritmo não visa apenas executá-las, mas também minimizar a maior utilização de um processador, que é essencialmente achar o menor valor X tal que o fluxo máximo seja K. Uma solução seria executar o algoritmo de fluxo máximo para todo X entre 1 e infinito e achar o primeiro que tem K como resposta, porém, esse X mínimo, caso exista, pode ser um valor muito alto, o que faria com que o algoritmo fosse executado muitas vezes, tornando-o inviável em certos casos. Para contornar este problema, foi utilizada uma busca binária.

Papel da Busca Binária

Analisando mais detalhadamente o grafo e o funcionamento de algoritmos de fluxo no geral, percebe-se que dado um grafo com suas capacidades e que seu fluxo máximo já é igual a um valor F qualquer, não existe a possibilidade do fluxo ser menor que F caso sejam aumentadas as capacidades de algumas arestas. Isso ocorre pois o fluxo não será restringido de passar por alguma aresta (na pior das hipóteses o algoritmo de fluxo máximo achará a mesma solução encontrada antes de aumentar as capacidades), visto que nenhuma capacidade foi diminuída, sendo assim, o novo fluxo F' será maior ou igual a F. De forma similar, caso as capacidades sejam decrementadas, o novo fluxo F' será menor ou igual a F.

Considerando o que foi dito acima, para valores crescentes de X, o fluxo máximo não decresce. Note que as únicas arestas alteradas pela busca binária são as que vão de um processador P_j para o *sink*. Tal observação é crucial e permite que a busca binária seja utilizada para achar a menor capacidade X. Como dito anteriormente, o fluxo máximo no grafo modelado para o problema nunca será maior que K, portanto, caso seja possível escalonar as tarefas, com certeza existe uma capacidade X que faça o fluxo máximo entre *src* e *sink* ser K. A solução do problema é então, o menor X que satisfaz a condição acima, ou seja, um X tal que para qualquer capacidade $Y > X$, o fluxo também vai ser K, pois não tem como ele ultrapassar K e nem ser menor, e para qualquer capacidade $Y < X$, o fluxo vai ser menor que K.

Há também os casos que não são possíveis de serem resolvidos: aqueles que dadas as configurações de tarefas e processadores, não tem como satisfazer as demandas de todas as tarefas. Esse caso é fácil de ser detectado, pois não importa o quão grande é o valor de X, o fluxo máximo sempre vai ser menor do que K.

Portanto, ao escolher um X qualquer com a busca binária e executar o algoritmo de fluxo máximo no grafo em que todas as capacidades das arestas de processadores P_j para o *sink* são iguais a X nos da uma informação sobre qual caminho seguir na busca binária. Sempre que o fluxo máximo for menor do que K, os chutes posteriores devem ser maiores do que o chute atual, caso contrário, devem ser menores, até chegar em um momento em que o fluxo máximo para $X - 1$ é menor do que K e o fluxo máximo para X é K. Esse é o ponto de parada do algoritmo e X representa o menor valor, tal que é possível executar todas as tarefas e satisfazer todas as suas demandas, executando no máximo X segundos em cada processador.

A Figura 6 ilustra a resposta encontrada pelo Balanceador de Carga para o caso da Figura 5. A tarefa 1 foi executada no segundo 2 pelo processador 2 e no segundo 3 pelo processador 1. A tarefa 2 foi executada pelo processador 2 no segundo 1. A tarefa 3 foi executada pelo processador 1 no segundo 4. A maior utilização de um processador

foi 2 nesse caso e é a mínima possível pois não tem como satisfazer todas demandas se a maior utilização for menor que 2.

Tarefa / Tempo	1	2	3	4
T_1		P_2	P_1	
T_2	P_2			
T_3				P_1

Figura 6 – Diagrama do resultado apresentado pelo Balanceador para o caso da Figura 5

3.1.2 Earliest Deadline First - EDF

O algoritmo EDF analisa todos os instantes de tempo entre 1 e Max_t, onde Max_t é o maior *deadline* D_i de uma tarefa, visto que após esse tempo, nenhuma tarefa pode ser executada mais, e então decide qual tarefa executar em qual processador.

Para saber qual tarefa deve ser executada e em qual processador, foi utilizado duas filas de prioridade, uma para as tarefas e uma para os processadores. A primeira é ordenada por ordem crescente do D_i, e a segunda por ordem crescente do último tempo em que o processador executou uma tarefa, chamado de L_j, ou seja, de $L_j + 1$ até o tempo atual que o algoritmo está, o processador P_j esteve ocioso.

Inicialmente, todos os processadores foram inseridos na segunda fila de prioridade com $L_j = -1$, visto que ainda não foram utilizados. As tarefas só são inseridas na primeira fila de prioridade quando o tempo atual que o algoritmo está analisando é igual à B_i.

Como foi dito, o algoritmo analisa todos os instantes de tempo viáveis e, após fazer as alterações necessárias em ambas as filas de prioridade (inserir todas as tarefas que devem começar no tempo atual e excluir todas as tarefas que possuem D_i menor do que o tempo atual), ele decide quais tarefas devem ser executadas e em quais processadores. Para isso, o método utilizado é o seguinte: execute a tarefa com o menor D_i no processador que está ocioso a mais tempo, ou seja, o que possui o menor L_j.

Porém, como o algoritmo é uma alteração do EDF para um único processador, esse processo de a cada instante de tempo decidir como escalonar as tarefas deve ser executado enquanto tiver tarefas e processadores disponíveis no tempo atual, para só então, ir para o próximo instante de tempo e continuar o algoritmo. O algoritmo 3 ilustra o funcionamento do EDF.

A Figura 7 ilustra a resposta achada pelo EDF. Ambos os processadores foram usados por 2 segundos cada, igual ao caso do Balanceador de Carga, porém, o *makespan* foi menor: no fim do segundo 3, o caso já estava resolvido, enquanto que no Balanceador de Carga isso aconteceu no fim do segundo 4.

Algoritmo 3: Pseudocódigo do EDF

Entrada: Tarefas e processadores
Saída: Métricas - utilização máxima, utilização média, *makespan*, tempo de execução
início
  para *tempo = 1 até* Max_t **faça**
    Insira todas as tarefas que começam em tempo na fila de prioridade das tarefas.
    Remova todas as tarefas que possuem *deadline* menor que *tempo*.
    enquanto *Existem tarefas e processadores disponíveis no tempo atual* **faça**
      Remova a tarefa com menor deadline.
      Remova o processador que está ocioso a mais tempo.
      Execute a tarefa no processador.
    fim
    Insira novamente na fila de prioridade das tarefas todas as tarefas que executaram no tempo atual, mas ainda não foram totalmente satisfeitas.
    Insira novamente na fila de prioridade dos processadores todos os processadores utilizados no tempo atual, com L_j atualizados.
  fim
fim
retorna *Métricas calculadas*

Tarefa / Tempo	1	2	3	4
T_1	P_1			
T_2		P_2	P_1	
T_3			P_2	

Figura 7 – Diagrama do resultado apresentado pelo EDF para o caso da Figura 5

3.1.3 Diferenças entre os dois algoritmos

Ambos os algoritmos possuem a mesma saída ($U_{\max}$, U_{med}, *makespan* e T_{exec}), porém a execução de cada um prioriza aspectos diferentes, visto que cada algoritmo resolve o mesmo problema, mas com objetivos distintos, portanto, os resultados podem ser diferentes. Além do mais, o EDF é mais flexível que o Balanceador de Carga pois não precisa conhecer todas os dados previamente para poder executar, o que o caracteriza como um algoritmo de escalonamento online, enquanto que o Balanceador de Carga é offline, ou seja, para achar a resposta ótima do problema, o algoritmo precisa conhecer previamente todos os dados sobre tarefas e processadores, para só então, começar a execução.

O EDF busca sempre executar a tarefa que está mais próxima de ficar indisponível no processador que está a mais tempo ocioso. Tal execução no processador com o menor L_j, de certa forma, é uma forma de balanceamento, porém, este não é o foco do algoritmo.

O EDF tem como objetivo primário atender todas as tarefas e minimizar o *makespan*, e tem como consequência um balanceamento de carga aceitável entre os processadores.

O Balanceador de Carga, por sua vez, visa primeiramente balancear a carga, e como consequência, tem um *makespan* aceitável

A seção 4 deste trabalho traz uma comparação mais detalhada entre os algoritmos, levando em consideração os objetivos de cada um e os resultados da execução de ambos em diversos casos de teste gerados aleatoriamente.

3.1.4 Complexidade de tempo dos algoritmos

A complexidade de tempo do Balanceador está relacionada com a complexidade de tempo do algoritmo de fluxo utilizado. Neste trabalho foi utilizado o Dinic, que por sua vez, tem uma complexidade $O(|V|^2 \times |E|)$, em que $|V|$ e $|E|$ são a quantidade de vértices e arestas no grafo, respectivamente. Como a busca binária é $O(\log(D))$, a complexidade final fica $O(|V|^2 \times |E| \times \log(D))$.

Com relação ao EDF, a complexidade do EDF depende da implementação da fila de prioridade utilizada. Neste caso, por ser uma $Heap$, os dois tipos de operações (remoção do menor elemento e inserção) realizadas pelo algoritmo executam em tempo logarítmico do tamanho da fila. Portanto, a complexidade do EDF neste trabalho é $O(D \times (\log(n) + \log(m)) + Max_t)$.

3.2 Métricas de avaliação

Os resultados dos algoritmos serão analisados e comparados seguindo algumas métricas, para que seja possível entender a fundo quando utilizar cada algoritmo.

Utilização máxima - U_{max}

Este valor corresponde ao maior tempo que um processador foi utilizado e nos da informações úteis sobre o balanceamento. Um balanceamento ótimo requer que este valor seja o menor possível e o Balanceador de Carga prioriza esta métrica.

Utilização média - U_{med}

Esta métrica mostra o quão bom um algoritmo é, considerando a distribuição de carga entre todos os processadores. A fórmula a seguir representa a utilização média:

$$U_{\text{med}} = \frac{D}{U_{\text{max}} \cdot m}$$

Quanto maior o valor da fórmula, melhor o algoritmo é nesse quesito. Considerando apenas uma execução do algoritmo em um caso de teste, sabe-se que a K e m são constantes, então para que a U_{med} seja a maior possível, o valor de U_{max} deve ser o menor possível, que é exatamente o principal objetivo do Balanceador de Carga.

U_{max} e U_{med} estão diretamente relacionadas, porém a U_{med} de certa forma padroniza o resultado de um caso de teste, e nos permite analisar o resultado sem olhar para outros atributos como n e m visto que independente deles, o resultado sempre será entre 0 e 1, em que 1 é o balanceamento ótimo.

Makespan

Este valor corresponde ao menor tempo em que todas as tarefas tiveram suas demandas satisfeitas. O EDF prioriza tal métrica, enquanto que para o Balanceador de Carga, ela é uma consequência da execução do algoritmo.

Tempo de execução do algoritmo - T_{exec}

Esta métrica mostra o quão rápido é o algoritmo e seu valor depende diretamente das especificações do computador onde o algoritmo foi executado. Neste trabalho, todos os testes foram feitos em um mesmo computador.

4 Resultados

Neste capítulo serão mostrados os resultados da execução dos dois algoritmos e também serão feitas comparações entre eles considerando as métricas já citadas.

4.1 Ambiente de execução

Todos os testes foram feitos em um computador Lenovo ideapad 330, com um processador Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz 1.80GHz, uma memória RAM de 8GB e um Sistema Operacional Windows 10 Home Single Language de 64 bits.

Todas as implementações deste trabalho foram feitas em C++. O Balanceador de Carga, em específico, é um código com uma complexidade de tempo e memória elevada e custoso para executar até mesmo para o C++ (caso fosse implementado em Python ou Java poderia demorar muito tempo para ser executado; caso fosse feito em C, a complexidade de implementação seria muito grande devido à ausência de certas estruturas de dados que estão presentes na biblioteca padrão do C++, a Standard Template Library - STL). Considerando todos esses fatores, o C++ foi escolhido e foi suficiente para implementar e testar tudo que foi feito neste trabalho com uma performance muito boa.

Foram gerados 400 casos de teste aleatoriamente, sendo que 27 deles se mostraram impossíveis de serem resolvidos por ambos os algoritmos, restando apenas 373.

Nos casos que puderam ser resolvidos, o Balanceador de Carga apresentou bons resultados, obtendo um balanceamento melhor ou igual ao do EDF em todos os casos. Além disso, em 3 dos 373 casos, o Balanceador apresentou um *makespan* inferior ao obtido pelo EDF e, na grande maioria dos outros casos, o *makespan* foi igual ao apresentado pelo EDF.

4.2 Resultados

Utilização máxima

Em todos os experimentos, a $U_{\max}$ de um processador no Balanceador de Carga foi menor ou igual à no EDF. Dos 373 casos, em 217 o Balanceador teve um resultado melhor, e nos outros 156 teve um resultado igual ao EDF. Tal resultado era esperado pois o foco principal do Balanceador é minimizar este valor por meio da busca binária.

A Tabela 1 mostra os casos mais discrepantes quanto ao valor da $U_{\max}$. É possível perceber que quanto menor os atributos n e m, menor é a diferença na $U_{\max}$ dos dois

algoritmos e isso se deve ao fato de que, geralmente, casos menores são mais simples de serem executados e não forçam as situações onde o método guloso do EDF poderia falhar. A medida que n e m aumentam, as possibilidades de execução também aumentam, e em muitas vezes uma escolha que parece ótima no momento para o EDF pode atrapalhar outras decisões mais pra frente, penalizando a $U_{\max}$ no fim da execução.

Tabela 1 – Utilização máxima

Caso	***n***	***m***	***Algoritmo***	U_{max}
11	50	10	Balanceador	230
			EDF	282
13	20	7	Balanceador	144
			EDF	178
39	5	5	Balanceador	60
			EDF	78
48	8	5	Balanceador	68
			EDF	83
51	7	3	Balanceador	135
			EDF	176
214	30	7	Balanceador	229
			EDF	308
327	29	7	Balanceador	182
			EDF	233

Utilização média

Assim como nos resultados da métrica $U_{\max}$ em nenhum caso o Balanceador foi pior que o EDF. Em alguns casos o Balanceador conseguiu o balanceamento ótimo, ou seja, $U_{\text{med}} = 1.0$. Nesses casos, todos os processadores executaram o mesmo tanto de tempo.

Dos 373 casos, o Balanceador atingiu o balanceamento ótimo em 52 deles, enquanto que o EDF em apenas 17. A Tabela 2 mostra os mesmos casos da Tabela 1 porém, substituindo a métrica $U_{\max}$ por U_{med}.

Makespan

Tal métrica não é o que o Balanceador busca otimizar, portanto, era esperado que o EDF fosse bem melhor que ele neste quesito, porém, em 3 (casos 116, 312, e 350) dos 373 casos, o *makespan* do Balanceador foi melhor que o *makespan* do EDF e em apenas 3 (casos 3, 5 e 340) ele foi pior, todos os outros casos eles tiveram o exatamento o mesmo *makespan*. A Tabela 3 mostra esses 6 casos interessantes, bem como alguns outros.

Tabela 2 – Utilização média

Caso	***n***	***m***	***Algoritmo***	U_{med}
11	50	10	Balanceador	0.99696
			EDF	0.81312
13	20	7	Balanceador	1.00000
			EDF	0.80899
39	5	5	Balanceador	0.99667
			EDF	0.76667
48	8	5	Balanceador	1.00000
			EDF	0.81928
51	7	3	Balanceador	0.99753
			EDF	0.76515
214	30	7	Balanceador	0.99813
			EDF	0.74212
327	29	7	Balanceador	0.99608
			EDF	0.77805

Tabela 3 – *Makespan*

Caso	***n***	***m***	***Algoritmo***	***makespan***
3	3	2	Balanceador	18
			EDF	16
5	10	2	Balanceador	408
			EDF	391
22	3	14	Balanceador	380
			EDF	380
116	19	2	Balanceador	610
			EDF	616
312	14	2	Balanceador	455
			EDF	472
340	35	3	Balanceador	653
			EDF	638
350	31	3	Balanceador	559
			EDF	567

Tempo de execução do algoritmo

O Balanceador é um algoritmo com uma complexidade de tempo maior que o EDF, portanto, é visível a diferença no tempo de execução entre os dois algoritmos. No entanto, o maior tempo de execução do Balanceador foi de aproximadamente $1,147$ segundo para o caso 16, contra $0,021$ segundo do EDF no caso 350. Assim, o tempo de execução do Balanceador é viável para para um humano caso haja a necessidade de se encontrar o melhor balanceamento possível e não apenas um balanceamento aceitável (no entanto, para um sistema operacional de tempo real, tal execução pode ser proibitiva). A Tabela 4 a mostra alguns tempos de execução de ambos os algoritmos.

Tabela 4 – Tempo de execução

Caso	***n***	***m***	***Algoritmo***	T_{exec}
2	30	20	Balanceador	0.06100
			EDF	0.00300
16	100	700	Balanceador	1.14700
			EDF	0.01000
17	100	700	Balanceador	1.14100
			EDF	0.01000
350	31	3	Balanceador	0.08400
			EDF	0.02100

Geral

As Tabelas 5 e 6 a seguir mostram uma visão geral de todas as métricas de alguns casos considerados importantes. A Tabela 5 contém alguns parâmetros dos casos, como n, m e K enquanto que a 5 contém os valores das métricas nos mesmos casos.

Tabela 5 – Dados das instâncias avaliadas

Caso	***n***	***m***	***D***
3	3	2	23
5	10	2	524
11	50	10	2293
13	20	7	1008
39	5	5	299
42	5	3	338
48	8	5	340
51	7	3	404
79	9	3	393
116	19	2	916
156	28	6	1388
165	20	6	858
214	30	7	1600
312	14	2	774
340	35	3	1485
350	31	3	1623

Tabela 6 – Métricas de desempenho

Caso	***Algoritmo***	U_{max}	U_{med}	***makespan***	T_{exec}
3	Balanceador	12	0.95833	18	0.00200
	EDF	12	0.95833	16	0.00000
5	Balanceador	262	1.00000	408	0.01700
	EDF	263	0.99620	391	0.00100
11	Balanceador	230	0.99696	627	0.06500
	EDF	282	0.81312	627	0.00500
13	Balanceador	144	1.00000	594	0.03500
	EDF	178	0.80899	594	0.00100
39	Balanceador	60	0.99667	369	0.01300
	EDF	78	0.76667	369	0.00100
42	Balanceador	113	0.99705	596	0.01300
	EDF	134	0.84080	596	0.00100
48	Balanceador	68	1.00000	440	0.01700
	EDF	83	0.81928	440	0.00000
51	Balanceador	135	0.99753	384	0.01400
	EDF	176	0.76515	384	0.00000
79	Balanceador	131	1.00000	539	0.01600
	EDF	159	0.82390	539	0.00000
116	Balanceador	458	1.00000	610	0.03600
	EDF	460	0.99565	616	0.00100
156	Balanceador	232	0.99713	567	0.05200
	EDF	279	0.82915	567	0.00200
165	Balanceador	143	1.00000	553	0.03200
	EDF	174	0.82184	553	0.00100
214	Balanceador	229	0.99813	575	0.04200
	EDF	308	0.74212	575	0.00200
312	Balanceador	387	1.00000	455	0.03200
	EDF	387	1.00000	472	0.00200
340	Balanceador	495	1.00000	653	0.07000
	EDF	538	0.92007	638	0.00300
350	Balanceador	541	1.00000	559	0.08400
	EDF	547	0.98903	567	0.02100

5 Conclusão

Este trabalho teve como objetivo principal desenvolver uma nova abordagem no contexto de escalonamento de tarefas ao focar no balanceamento de carga e não na minimização do *makespan*. Além disso, o trabalho mostrou como conceitos completamente distintos, no caso, fluxo em redes e busca binária, podem ser combinados para resolver um problema complexo.

Assim, os resultados podem ser considerados satisfatórios, visto que o *makespan* não era o objetivo principal do algoritmo proposto neste trabalho. Computacionalmente, o Balanceador é menos eficiente em termos de tempo e memória quando comparado ao EDF; por outro lado, experimentalmente, os tempos de execução dos dois algoritmos são rápidos o suficiente para os ambientes avaliados.

Todas as situações avaliadas neste trabalho tiveram processadores idênticos e seria interessante um trabalho futuro que os considerasse distintos, com velocidades diferentes e até mesmo requisitos diferentes (como, por exemplo, uma limitação no número de segundos que um processador específico pode executar).

Um ponto negativo do Balanceador de Carga é a quantidade de arestas criadas na modelagem do grafo. Para um trabalho futuro, recomenda-se reduzir o número de arestas. Assim, não só a quantidade de memória irá reduzir, como também o tempo de execução do algoritmo, que é muito atrelado à quantidade de arestas presentes no grafo.

Referências

ALBERS, S. Online algorithms: a survey. *Mathematical Programming*, v. 97, n. 1, p. 3–26, 2003. Citado 2 vezes nas páginas 8 e 11.

AWERBUCH, B.; LEIGHTON, F. T. Improved approximation algorithms for the multi-commodity flow problem and local competitive routing in dynamic networks. In: LEIGHTON, F. T.; GOODRICH, M. (Ed.). *STOC '94: Proceedings of the Twenty-sixth Annual ACM Symposium on Theory of Computing*. New York: ACM, 1994. p. 487–496. Citado 2 vezes nas páginas 8 e 14.

BONDY, J. A.; MURTY, U. S. R. *Graph theory*. EUA: Springer, 2008. 650 p. (Graduate Texts in Mathematics, 244). Citado 4 vezes nas páginas 8, 11, 12 e 13.

CORMEN, T. H.; LEISERSON, C. S.; RIVEST, R. L.; STEIN, C. *Algoritmos: Teoria e Prática*. 3. ed. Rio de Janeiro: Campus Elsevier, 2012. 944 p. Citado 2 vezes nas páginas 8 e 13.

DINITZ, Y. Algorithm for solution of a problem of maximum flow in a network with power estimation. *Doklady Akademii Nauk SSSR*, v. 11, p. 1277–1280, 1970. Citado 2 vezes nas páginas 9 e 18.

_____. Dinitz' algorithm: The original version and even's version. In: GOLDREICH, O.; ROSENBERG, A. L.; SELMAN, A. L. (Ed.). *Theoretical Computer Science: Essays in Memory of Shimon Even*. Berlin Heidelberg: Springer, 2006. p. 218–240. Citado 2 vezes nas páginas 9 e 18.

EVEN, S.; TARJAN, R. E. Network flow and testing graph connectivity. *SIAM journal on computing*, v. 4, n. 4, p. 507–518, 1975. Citado na página 8.

FORD JR, L. R.; FULKERSON, D. R. *Flows in networks*. 2. ed. EUA: Princeton University Press, 2010. 216 p. Citado na página 16.

GOLDBARG, M. *Programação linear e fluxos em redes*. São Paulo: GEN LTC, 2014. 520 p. Citado na página 8.

GRAHAM, R. L. Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics*, v. 17, n. 2, p. 416–429, 1969. Citado na página 8.

HAGHANI, A.; OH, S.-C. Formulation and solution of a multi-commodity, multi-modal network flow model for disaster relief operations. *Transportation Research Part A: Policy and Practice*, v. 30, n. 3, p. 231–250, 1996. Citado 2 vezes nas páginas 8 e 16.

MEKKITTIKUL, A.; MCKEOWN, N. A practical scheduling algorithm to achieve 100% throughput in input-queued switches. In: CHOUDHURY, A. K. (Ed.). *Proceedings of Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies*. San Francisco: IEEE, 1998. v. 2, p. 792–799. Citado na página 8.

SRINIVASAN, A.; BARUAH, S. Deadline-based scheduling of periodic task systems on multiprocessors. *Information Pprocessing Letters*, v. 84, n. 2, p. 93–98, 2002. Citado 3 vezes nas páginas 9, 14 e 17.