UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

NATHANA CAROLINE DONINI CEZARIO

COMPARAÇÃO DO TEMPO DE PROCESSAMENTO ENTRE HEURÍSTICAS SIMULADAS SELECIONADAS EM PROBLEMAS DE *FLOW SHOP* NAS LINGUAGENS DE PROGRAMAÇÃO PASCAL E C

PONTA GROSSA

2022

NATHANA CAROLINE DONINI CEZARIO

COMPARAÇÃO DO TEMPO DE PROCESSAMENTO ENTRE HEURÍSTICAS SIMULADAS SELECIONADAS EM PROBLEMAS DE *FLOW SHOP* NAS LINGUAGENS DE PROGRAMAÇÃO PASCAL E C

Comparison of processing time between selected simulated heuristics in flow shop problems in Pascal e C programming languages

Trabalho de conclusão de curso de graduação apresentado como requisito para obtenção do título de Bacharel em Engenharia de Produção da Universidade Tecnológica Federal do Paraná (UTFPR).
Orientador(a): Prof. Dr. Fábio José Ceron Branco.

PONTA GROSSA

2022

4.0 Internacional

Esta licença permite compartilhamento, remixe, adaptação e criação a partir do trabalho, mesmo para fins comerciais, desde que sejam atribuídos créditos ao(s) autor(es). Conteúdos elaborados por terceiros, citados e referenciados nesta obra não são cobertos pela licença.

NATHANA CAROLINE DONINI CEZARIO

COMPARAÇÃO DO TEMPO DE PROCESSAMENTO ENTRE HEURÍSTICAS SIMULADAS SELECIONADAS EM PROBLEMAS DE *FLOW SHOP* NAS LINGUAGENS DE PROGRAMAÇÃO PASCAL E C

Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Bacharel em Engenharia de Produção da Universidade Tecnológica Federal do Paraná (UTFPR).

Data da aprovação: 12/dezembro/2022

___

Fábio José Ceron Branco
Doutorado
Universidade Tecnológica Federal do Paraná

___

Ana Maria Bueno
Mestrado
Universidade Tecnológica Federal do Paraná

___

Shih Yung Chin
Doutorado
Universidade Tecnológica Federal do Paraná

PONTA GROSSA

2022

Dedico este trabalho à minha família e amigos, pelos
momentos de ausência.

AGRADECIMENTOS

Este momento é de grande alegria e agradecimento por todos esses anos nesse caminho percorrido, à todos que contribuíram para o meu crescimento profissional e pessoal.

Imensa gratidão principalmente pelo meu pai e pela minha mãe, que, apesar da distância, se mantiveram pacientes, pela ajuda nos momentos difíceis, assim como nas alegrias desde a minha aprovação no curso até a minha graduação.

Agradeço também ao meu noivo, Rafael Dantas, por estar do meu lado por todos esses anos, me apoiando e incentivando, desde o momento em que nos conhecemos na faculdade.

Aos meus colegas de faculdade e amigos em que compartilhei grandes momentos e superações durante o curso, agradeço pela união e apoio.

Ao meu professor orientador Fábio José Ceron Branco pelos ensinamentos e dedicação, a fim de tornar possível a realização deste trabalho.

RESUMO

Este trabalho tem como objetivo o estudo comparativo dos tempos de processamento através da simulação das heurísticas *Shortest Processing Time* (SPT), *Longest Processing Time* (LPT) e NEH adaptadas para a resolução de problemas para os sistemas de produção clássico, *no idle* e *no wait*, para encontrar uma minimização de *makespan* e de *flowtime*, em um ambiente de sequenciamento de tarefas *flow shop*. Os problemas de sequenciamento de n tarefas em m máquinas, possuem como objetivo miniminizar o tempo total das tarefas nas máquinas, a fim de se obter uma maior eficiência no processo produtivo de empresas, além dessa otimização do tempo total nas instâncias, buscou-se também a otimização dos tempos de processamento de cada método implementado e comparando-as em duas linguagens de programação diferentes, Pascal e C, utilizando técnicas de boas práticas. Após as análises medidas por meio de sucesso e de desvios relativos médios, constatou-se um melhor desempenho nos métodos em que o NEH se encontrava. Além disso, foi analisado por meio de variância percentual dos tempos de processamento de cada método nas duas linguagens de programação, no qual podemos concluir que o código implementado na linguagem de programação em C obteve uma redução de 43% nos tempos de processamento em relação aos métodos, e em relação às instâncias, se obteve uma diminuição de 36%.

Palavras-chave: heurísticas; *makespan*; *flowtime*; tempo de processamento; *scheduling*.

ABSTRACT

The objective of this work is the comparative study of processing times through the simulation of the Shortest Processing Time (SPT), Longest Processing Time (LPT) and NEH heuristics adapted to solve problems for classic production systems, no idle and no wait, to find a minimization of makespan and flowtime, in a flow shop task sequencing environment. The sequencing problems of n tasks in m machines, have as objective to minimize the total time of the tasks in the machines, in order to obtain a greater efficiency in the productive process of companies, in addition to this optimization of the total time in the instances, it was also sought the optimization of the processing times of each implemented method and comparing them in two different programming languages, Pascal and C, using best practice techniques. After the analyzes measured by means of success and mean relative deviations, a better performance was found for the method in which the NEH was. In addition, it was analyzed using the percentage variance of the processing times of each method in the two programming languages, in which we can conclude that the code implemented in the programming language in C obtained a 46% reduction in processing times in relation to the methods, and in relation to systems, there was a decrease of 36%.

Keywords: heuristics; makespan; flowtime; processing time; scheduling.

LISTA DE FIGURAS

Figura 1 - Gráfico de Gantt em um Sistema de Produção Clássico....................17
Figura 2 - Gráfico de Gantt em um Sistema de Produção *No Wait*.....................18
Figura 3 - Gráfico de Gantt em um Sistema de Produção *No Idle* 19
Figura 4 – Gráfico de DRM no sistema clássico com função objetivo *makespan* 29
Figura 5 – Gráfico de DRM no sistema clássico com função objetivo *flowtime*29
Figura 6 – Gráfico de DRM no sistema *no idle* com função objetivo *makespan*30
Figura 7 – Gráfico de DRM no sistema *no idle* com função objetivo *flowtime* ..30
Figura 8 – Gráfico de DRM no sistema *no wait* com função objetivo *makespan* 31
Figura 9 – Gráfico de DRM no sistema *no wait* com função objetivo flowtime .31
Figura 10 – Médias das variâncias percentuais no sistema clássico por instâncias..37
Figura 11 – Médias das variâncias percentuais no sistema *no idle* por instâncias..37
Figura 12 – Médias das variâncias percentuais no sistema *no wait* por instâncias..38
Figura 13 – Médias das variâncias percentuais nos sistemas por método 38

LISTA DE TABELAS

Tabela 1 - Exemplo de um Sistema de Produção 17
Tabela 2 – Tabela da sequência dos problemas de Taillard (1993) 25
Tabela 3 – Dados de tempo de processamento na linguagem Pascal para o sistema clássico 32
Tabela 4 – Dados de tempo de processamento na linguagem Pascal para o sistema *no idle* 32
Tabela 5 – Dados de tempo de processamento na linguagem Pascal para o sistema *no wait*.. 33
Tabela 6 – Dados de tempo de processamento na linguagem C para o sistema clássico 33
Tabela 7 – Dados de tempo de processamento na linguagem C para o sistema *no idle*... 34
Tabela 8 – Dados de tempo de processamento na linguagem C para o sistema *no wait* 34
Tabela 9 – Cálculo da variação percentual de tempo de processamento entre as linguagens de programação no sistema clássico 35
Tabela 10 – Cálculo da variação percentual de tempo de processamento entre as linguagens de programação no sistema *no idle* 35
Tabela 11 – Cálculo da variação percentual de tempo de processamento entre as linguagens de programação no sistema *no wait*.. 36

LISTA DE ABREVIATURAS E SIGLAS

LPT	*Longest Processing Time*
NEH	Nawaz, Enscore, Ham
NWFS	*No-Wait Flow Shop*
NIFS	*No-Idle Flow Shop*
SPT	*Shortest Processing Time*
ANSI	*American National Standards Institute*
DRM	Desvio Relativo Médio
Tc	Tempo de processamento na linguagem de programação C
Tp	Tempo de processamento na linguagem de programação Pascal

SUMÁRIO

1 INTRODUÇÃO 12
1.1 OBJETIVOS 13
1.2 PERGUNTA DE PESQUISA 13
1.3 GERAL 13
1.4 ESPECÍFICOS 13
1.5 JUSTIFICATIVA 13
1.6 ESTRUTURA DO TRABALHO 14
2 REFERENCIAL TEÓRICO 15
2.1 SISTEMAS DE PRODUÇÃO 15
2.2 PROGRAMAÇÃO DA PRODUÇÃO 16
2.2.1 SISTEMA CLÁSSICO 17
2.2.2 SISTEMA *NO WAIT* 18
2.3 FUNÇÕES OBJETIVO 19
2.3.1 *MAKESPAN* 19
2.3.2 *FLOWTIME* 19
2.3.3 *TARDINESS* 20
2.3.4 *EARLINESS* 20
2.4 MÉTODOS HEURÍSTICOS 20
2.4.1 *LONGEST PROCESSING TIME* (LPT) 20
2.4.2 *SHORTEST PROCESSING TIME* (SPT) 21
2.4.3 NEH 21
2.5 LINGUAGENS DE PROGRAMAÇÃO E BOAS PRÁTICAS 21
2.5.1 A linguagem Pascal 22
2.5.2 A linguagem C 22
2.5.3 *Clean Code* 23
3 METODOLOGIA DE PESQUISA 24
3.1 CLASSIFICAÇÃO DA PESQUISA 24
3.2 LEVANTAMENTO DE DADOS 25
4 DESENVOLVIMENTO 27
4.1 ANÁLISE DOS RESULTADOS 27
4.1.1 COMPARAÇÃO DOS MÉTODOS 28
4.1.1.1 GRÁFICOS DE SUCESSO 28
4.1.1.2 GRÁFICOS DOS DESVIOS RELATIVOS MÉDIOS 28

4.1.2 COMPARAÇÃO DOS TEMPOS DE PROCESSAMENTO 31
4.1.2.1 NA LINGUAGEM PASCAL 32
4.1.2.2 NA LINGUAGEM C 33
4.1.2.3 CÁLCULO DA PORCENTAGEM DE MELHORIA 35
5 CONCLUSÃO 40
REFERÊNCIAS 41
APÊNDICE A - CÓDIGO EM PASCAL 44
APÊNDICE B - CÓDIGO EM C 51

1 INTRODUÇÃO

Um sistema pode ser definido como um conjunto de elementos que interagem a fim de desempenhar uma função. Nos sistemas de produção, não podia ser diferente, sendo um conjunto de elementos interligados relacionados a produção de um produto ou a um serviço. Para se chegar em um objetivo final, é preciso de uma programação de tarefas para esses elementos.

Por meio da resolução de problemas de programação da produção estão sendo encontradas maneiras eficazes com melhores alternativas e soluções, desenvolvendo métodos para a redução de tempos de produção e de seus custos, a fim de aumentar a produtividade e qualidade de seus produtos. Uma atitude estratégica e competitiva, pois com o desenvolvimento da automação, as linhas de produção estão cada vez mais próximas da Indústria 4.0 e em busca de produtos cada vez mais padronizados.

Por meio da aplicação de heurísticas, a fim de otimizar as funções objetivos dos sistemas produtivos, visto que cada sistema produtivo apresenta valores distintos para ordens de produção e tempos de processamento. Este estudo visa trabalhar com problema *flow shop*, ou seja, quando todas as tarefas têm as mesmas ordens de processamento para todas as máquinas. O sequenciamento de produção, ou *scheduling*, busca resolver os gargalos da linha de produção através de maneiras eficientes a fim de otimizar a utilização dos recursos disponíveis para sua transformação e consequentemente, obter se os produtos finais.

As heurísticas podem ser aplicadas nos mais variados tipos de sistema de produção, estudadas nesse trabalho, o sistema clássico, *no idle* e *no wait.* Em cada sistema existem distintas funções objetivo que incorporam na programação como, por exemplo, o tempo total gasto na produção (*makespan*), o tempo total de fluxo da produção (*flowtime*), entre outras funções.

Além da minimização do tempo total gasto no fluxo da produção, é importante que o tempo de processamento das tarefas sejam também minimizadas, uma vez que uma quantidade grande de tarefas pode gerar um tempo de processamento muito lento.

1.1 OBJETIVOS

Este tópico possui como finalidade apresentar quais são os objetivos que o presente trabalho deverá atingir para a sua conclusão, a pergunta de pesquisa, que será respondida na sequência e também a estrutura do trabalho.

1.2 PERGUNTA DE PESQUISA

É possível obter um melhor tempo de processamento na execução de códigos de diferentes linguagens que executam as heurísticas selecionadas em um conjunto de grande número de tarefas por máquinas?

1.3 GERAL

Fazer uma implementação em duas linguagens com as heurísticas *shortest processing time* (SPT), *longest processing time* (LPT) e *NEH* pelos sistemas clássicos, *no idle* e *no wait*, minimizando o *makespan* e o *flowtime*, para a obtenção de uma máxima eficiência no processo produtivo e uma minimização do tempo de processamento do algoritmo.

1.4 ESPECÍFICOS

- Para o conjunto de heurísticas selecionadas, simular um conjunto de instâncias de tarefas por máquinas (NxM);
- Obter o tempo total de produção e do fluxo nos ambientes simulados;
- Analisar através da construção de gráficos e tabelas, a fim de se obter a heurística de melhor desempenho.

1.5 JUSTIFICATIVA

A fim de se reduzir gastos e de otimizar a produção da melhor maneira, problemas de *scheduling* são o principal meio de se atingir tais objetivos. Por ser bastante utilizado nos meios de produção, existem diversos estudos sobre esse problema, porém, não há na literatura um método de solução ótima.

Problemas de alta complexidade possuem apenas soluções de alta qualidade para cada função objetivo, este trabalho se dedica a fim de se encontrar uma solução para um conjunto de heurísticas para redução do tempo de processamento com o uso de diferentes linguagens de programação.

1.6 ESTRUTURA DO TRABALHO

O presente estudo foi estruturado da seguinte maneira:

• Capítulo 1: Introdução, onde consta a apresentação do tema a ser estudado; a pergunta de pesquisa, assim como os objetivos gerais e específicos do estudo;

• Capítulo 2: Referencial teórico, apresentação da base de pesquisa necessária para a realização do trabalho.

• Capítulo 3: Metodologia, especificação da classificação de pesquisa e a maneira em que foi realizada;

• Capítulo 4: Desenvolvimento do trabalho, realização das análises para a obtenção dos resultados.

• Capítulo 5: Conclusão para a finalização deste estudo e sugestão de trabalhos futuros.

2 REFERENCIAL TEÓRICO

Este capítulo tem como objetivo apresentar conteúdos relevantes relacionados ao tema deste estudo, iniciando com os sistemas de produção (2.1), funções objetivo (2.2), explicando sobre os métodos heurísticos (2.3) e finalmente sobre as linguagens de programação e boas práticas (2.4).

2.1 SISTEMAS DE PRODUÇÃO

Um sistema de produção é o conjunto de atividades e operações inter relacionadas envolvidas na produção de bens ou serviços, segundo Moreira (2008), integrando entre si e formando um sistema. Para Chiavenato (1983) pode ser definido como um conjunto de partes interagentes e interdependentes, ou seja, podem-se relacionar dinamicamente e se unificar, e efetuam uma atividade ou função para atingir um ou mais objetivos ou propósitos.

São classificados de diferentes formas sob o enfoque de diversos autores, desde uma perspectiva mais ampla, por exemplo, Pires (2004) classifica os sistemas produtivos com base nas atividades econômicas: primária (extrativismo), secundária (transformativa) e terciária (serviços), até a mais estreita e específica, como classificada por Russomano (2000) como sistema contínuo, intermitente e construção de projetos.

Em um sistema contínuo, o tempo de disposição das máquinas é pequeno em comparação com o tempo total da produção, geralmente na operação de produtos padronizados. O sistema intermitente é destacado por sua flexibilidade, ou seja, pela capacidade de produzir uma variabilidade grande de produtos, sendo sub classificado por, conforme Maccarthy e Liu (1993):

Flow shop: a linha de produção possui o mesmo fluxo de processamento nas máquinas, sendo permutacional, quando possui uma mesma ordem operacional em diversas máquinas, onde é importante determinar as ordens possíveis para que se possa encontrar uma otimização da função objetivo desejada, porém, só é possível fazer a simulação em produções de pequeno porte, devido ao tempo de processamento.

Job shop : a ordem operacional se modifica de um produto para o outro, onde é possível diversificar a produção, mas em pequenas quantidades personalizadas. Os

tempos de processamento em cada máquina se torna praticamente imprevisíveis por causa da variabilidade nas formas de produção.

Na construção de projetos, possui como produção de itens de grande porte, com alta complexidade e na maioria das vezes, itens únicos. Sendo assim, cada empresa tem seus respectivos produtos ou serviços e possuem sistemas de produção distintos, de acordo com o modo que trabalham. Mas a principal função de um sistema de produção é a transformação de matérias-primas em um produto ou serviço, na qual o valor do produto é agregado.

2.2 PROGRAMAÇÃO DA PRODUÇÃO

Para um sistema de produção ser eficiente, esta precisa estar vinculada a diversos fatores gerenciais e decisores, um desses fatores é a programação da produção ou *scheduling*, que segundo Baker (1974), se dá resumidamente como a alocação de recursos por meio do tempo.

A programação da produção pode ser definida como a determinação de quando e onde cada operação necessária para fabricação de um produto deve ser realizada ou a determinação de datas nas quais se deve iniciar e/ou completar cada evento ou operação que compõe um procedimento (FUCHIGAMI, 2005). Portanto, a programação da produção é responsável por definir o processo de compra, fabricação e montagem de cada elemento para se chegar no produto final.

A sequência da programação dos produtos é capaz de definir e reduzir os tempos de *setup* das máquinas, a organização pode ter vantagens a curto, médio e longo prazo. Nas atividades rotineiras a programação deve atender à demanda de produção, no médio prazo deve-se planejar os recursos com o volume de produtos previstos, e pôr fim, a longo prazo cabe à empresa investir em inovações, pessoas e estratégias a fim de possuir uma melhoria contínua em seus processos e na posição de mercado.

Um problema de programação da produção pode ser definido pelo número de tarefas e operações a serem procedidas, através da quantidade e tipo de máquinas disponíveis e pelo fluxo das mesmas, onde se determina o número de tarefas e de máquinas seja finito, tendo como critério de otimização uma solução objetivo (CONWAY, 1967).

O gráfico de Gantt, inventado em 1917 por Henry Laurence Gantt, é uma ferramenta que representa graficamente através de barras por tempo de processamento das operações, ilustrando o avanço das diferentes sequências, onde o tempo de início e de final das atividades são indicadas no gráfico. As vantagens dessa ferramenta é que faz uma representação visual simples e de fácil entendimento do que ocorre em cada operação da produção (SLACK, 1999).

Os intervalos de tempo representam o início e o fim de cada *Mi* (máquinas) em cada *Ji* (tarefas) aparecendo como barras horizontais, para exemplificar e diferenciar cada tipo de sistema de produção apresentado neste trabalho, conforme o exemplo na Tabela 1.

Tabela 1 - Exemplo de um Sistema de Produção

	J1	***J2***	***J3***
M1	5	4	*3*
M2	3	2	*3*
M3	*4*	*1*	*6*

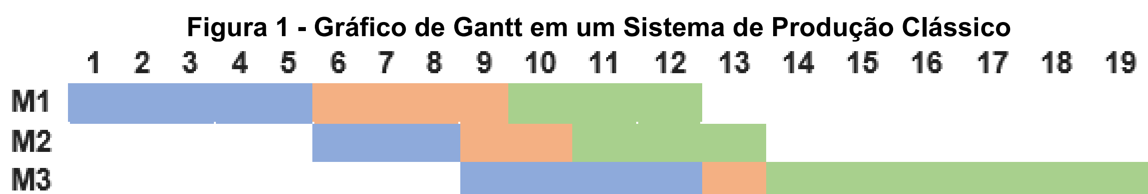
Fonte: Autoria Própria (2022)

Neste estudo é considerado o problema de programação de *flow shop* permutacional, aplicado com diferentes critérios, sendo eles: no sistema clássico, no sistema *no wait* e no sistema *no idle*, sendo detalhados a seguir, com exemplos aplicados nos gráficos de Gantt.

2.2.1 SISTEMA CLÁSSICO

Em um sistema clássico, as tarefas são alocadas nas máquinas, assim que elas estiverem disponíveis, não possuindo restrições, onde cada máquina realiza uma tarefa por vez, sendo cada tarefa representada pelas respectivas cores conforme na Tabela 1. Sendo assim, dá para se obter uma linha do tempo de acordo com cada máquina, como se observa na Figura 1.

Figura 1 - Gráfico de Gantt em um Sistema de Produção Clássico



Fonte: Autoria Própria (2022)

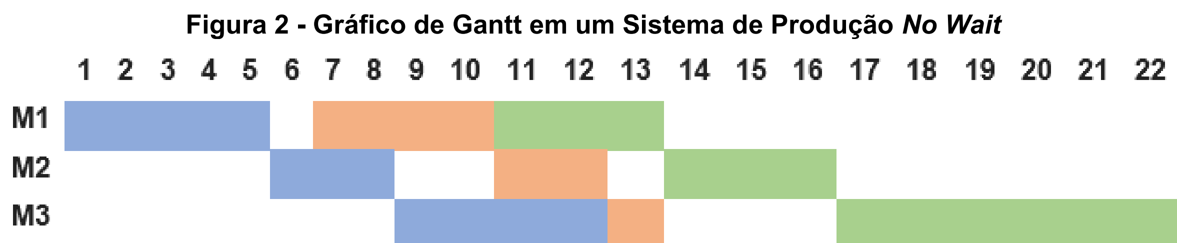
Como podemos observar, no sistema clássico, as tarefas só podem ser processadas somente uma máquina por vez, onde, cada máquina só pode processar uma operação simultaneamente.

2.2.2 SISTEMA *NO WAIT*

Deman e Baker (1974) foram os primeiros autores a estudarem o problema de *No-Wait Flow Shop* (NWFS), com a minimização do tempo total do fluxo das tarefas.

Na programação da produção em sistema de produção *No-Wait Flow Shop* (NWFS), uma vez que as operações de uma determinada tarefa são iniciadas, devem ser processadas sem interrupções nas consecutivas máquinas, ou seja, não permite que ocorra tempo de espera no processamento da tarefa de uma máquina para a outra (*No-Wait*). O único tempo de espera tolerado é no início do processamento da primeira operação das tarefas, na primeira máquina, conforme a Figura 2.

Figura 2 - Gráfico de Gantt em um Sistema de Produção *No Wait*



Fonte: Autoria Própria (2022)

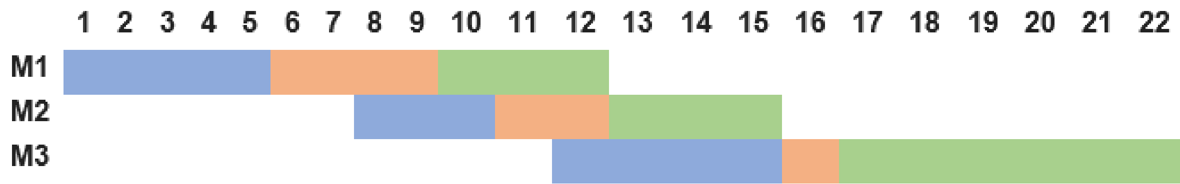
Portanto, no sistema *No Wait*, assim que a tarefa é concluída em uma máquina, deve-se passar imediatamente para a próxima, não podendo ter esperas entre os processamentos de duas máquinas.

2.2.3 SISTEMA *NO IDLE*

Na programação da produção em sistema de produção *No-Idle Flow Shop* (NIFS), não são permitidas ociosidades nas máquinas, o que, consequentemente, pode atrasar o início do processamento das tarefas na próxima máquina, e nas subsequentes, refletindo no aumento do tempo total de programação (BRANCO, 2011).

Em um sistema *No Idle*, as tarefas são iniciadas assim que são finalizadas na máquina anterior, não existindo lacunas entre as tarefas, já que não tem paradas ao longo do processo produtivo, como ilustrado na Figura 3.

Figura 3 - Gráfico de Gantt em um Sistema de Produção *No Idle*



Fonte: Autoria Própria (2020)

Esse sistema é geralmente utilizado para reduzir os custos de utilização ou diminuir o tempo de preparação da mesma, evitando que os tempos de *setup* seja elevado.

2.3 FUNÇÕES OBJETIVO

Uma função objetivo é criada para maximizar ou minimizar um problema, buscando o significado da qualidade da solução em função das variáveis presentes. No *scheduling*, a principal finalidade é a otimização da função objetivo, podendo assumir várias formas, como, por exemplo, segundo Pinedo (2008), *makespan*, *flowtime*, *tardiness* e *earliness*. Considerando estes exemplos, o estudo presente utilizou-se do *makespan* e o *flowtime*. A escolha foi dada devido a ser frequentemente usada na literatura e de possuir maior comunicação com a utilização de programações *flow shop*.

2.3.1 *MAKESPAN*

O makespan é representado pelo cálculo do tempo total gasto até a última tarefa finalizada, sendo a soma de todas as tarefas. Por exemplo, como ilustrado na Tabela 1, podemos afirmar que o *makespan* do sistema de produção clássico exemplificado é 19. Essa função demonstra a utilização total da linha e em consequência, maior confiabilidade nos prazos de entrega.

2.3.2 *FLOWTIME*

O *flowtime* é obtido através da soma dos tempos de conclusão de todas as tarefas, e demonstra redução das matérias-primas no processo. Por exemplo, na Tabela 1, o *flowtime* é representado pela soma (12+13+19) sendo no total 44. Essa função pode ser estudada a fim de reduzir de custos e estoque.

2.3.3 *TARDINESS*

O *tardiness* representa o tempo excedido na entrega, ou seja, quando é finalizado com atraso em relação à sua data prometida.

2.3.4 *EARLINESS*

O *earliness* é dado como o adiantamento do término das tarefas em relação à entrega, ou seja, pelo adiantamento da entrega em relação à sua data prometida.

2.4 MÉTODOS HEURÍSTICOS

Um método heurístico, de acordo com Fuchigami (2005), é um procedimento de busca a solução de um problema apoiado em critérios racionais ou computacionais para selecionar um caminho entre vários existentes, sem se preocupar de examinar todas as possibilidades ou de atingir a melhor opção. Mas deve-se achar uma solução viável, ou próxima da ótima, e que o tempo de computação seja aceitável.

Porém, na maioria dos casos, é considerado mais viável encontrar uma solução heurística do que uma solução ótima, pois essa solução exige muito tempo de processamento computacional ou sendo até mesmo inacessível nos dias de hoje.

Neste trabalho irão ser estudados e comparados, as heurísticas escolhidas: LPT, SPT e NEH, as mais utilizadas e conhecidas na literatura, porém, diferente de outros trabalhos, leva em conta a otimização dos tempos de execução.

2.4.1 *LONGEST PROCESSING TIME* (LPT)

Koulamas e Kyparisis (2008) foram responsáveis pela modificação na heurística LPT a fim de minimizar o *makespan* para duas máquinas. O algoritmo programa, por exemplo, primeiramente as três tarefas de maior duração e depois dessa ordenação, as demais tarefas continuam com a sequência de programação com base na regra LPT.

O *Longest Processing Time* (Maior Tempo de Processamento) prioriza as tarefas mais longas do processo, colocando as tarefas em ordem decrescente, de modo geral apresenta valores inferiores para *makespan*. Por exemplo, como ilustrado na Tabela 1, somando as tarefas separadamente (J1=12, J2=7, J3=12), portanto, a ordem seria J1/J3/J2.

2.4.2 *SHORTEST PROCESSING TIME* (SPT)

O *Shortest Processing Time* (Menor Tempo de Processamento) é outro método de sequenciamento que prioriza as tarefas mais curtas do processo, colocando as tarefas em ordem crescente, de modo geral apresenta valores inferiores para *flowtime*. Por exemplo, como demonstra na Tabela 1, somando as tarefas separadamente (J1=12, J2=7, J3=12), portanto, a ordem seria J2/J1/J3.

2.4.3 NEH

O método NEH, desenvolvido por Nawaz, Enscore e Ham em 1983, é dividido em duas partes: a primeira é a ordenação de tarefas de acordo com a soma de cada uma, e a segunda, um sequenciamento onde são comparados os dois primeiros números para definir o sequenciamento final, escolhendo o de menor valor, sendo esse passo realizado com os demais pares restantes.

O método tem bastante referência na criação de novas heurísticas que utilizam como parâmetros as ordenações LPT e SPT. Como se trata de um método construtivo, este possui duas fases, onde na primeira é baseado na ordenação dos métodos de sequenciamento LPT ou SPT; e em seguida, na segunda fase, o NEH procede construindo uma sequência nova à medida que a ordem das tarefas vai sendo alterada através dos resultados da função objetivo. Ao explorar novas sequências depois da ordenação inicial LPT/SPT, o método se certifica de encontrar as soluções de qualidade.

2.5 LINGUAGENS DE PROGRAMAÇÃO E BOAS PRÁTICAS

Para poder implementar os métodos heurísticos, é recomendado a utilização da implantação com linguagens de alto nível, que são aquelas em que podem ser implementadas para a realização de tarefas numerosas, realizadas através de únicas instruções. Seus compiladores são programas que traduzem a linguagem de máquina para a linguagem de alto nível, demandando uma quantidade de tempo de processamento para fazer este trabalho, dependendo da complexidade do programa, pode exoigir um maior tempo para execução do código.

Através dos compiladores, podemos adicionar recursos, bibliotecas, manipular arquivos, a fim de que se possa desenvolver um algoritmo para atingir resultados.

Segundo Dasgupta, Papadimitriou e Vazirani, "algoritmos são procedimentos precisos, não ambíguos, padronizados, eficientes e corretos", sendo assim, são usados há séculos para a resolução de problemas, sobretudo os matemáticos.

Sendo assim, a escolha das linguagens de programação para a implementação das heurísticas escolhidas neste trabalho, Pascal e C, foi devido à grande popularidade em trabalhos similares na literatura, além de serem linguagens comuns para o aprendizado em ambiente universitário pelo fácil entendimento e acesso.

Além da implementação das heurísticas nas diferentes linguagens, foi posto em prática a importância das boas práticas no desenvolvimento de algoritmos ou softwares, que são muito visadas nas organizações (MARTIN, 2008). Elas são encarregadas em trazer vantagens para a aplicação, tais como: agilidade, simplicidade, manutenabilidade e até mesmo redução de recursos. Um código é considerado bom, quando é eficiente no uso de seus recursos, como, por exemplo, na usabilidade de memória disponível, na otimização do tempo de processamento e na usabiliade de pouca banda de rede.

2.5.1 A linguagem Pascal

Pascal é uma linguagem de programação criada em 1970 pelo suíço Nicklaus Wirth, para ensinar programação estruturada, ou seja, uma programação que pode ser definida pelas configurações: sequência, decisão e iteração; sendo influenciada pela linguagem ALGOL. Trata-se de uma linguagem orientada por objetos, padronizada pela ANSI, utilizada em larga escala. Além de ser uma linguagem didática utilizada para o desenvolvimento da lógica de programação, é utilizada no Delphi, um dos ambientes mais usadas em todo o mundo para o desenvolvimento de sistemas.

2.5.2 A linguagem C

A linguagem C foi originada pela empresa *AT&T Bell Labs* para o desenvolvimento de um sistema operacional chamado Unix, criada em 1972 por Dennis Ritchie. Se trata de uma linguagem flexível, podendo ser utilizada nos mais diversos tipos de projeto, capaz de gerar programas rápidos em se tratando de tempos de execução, pois possui uma sintaxe simples, padronizada pela ANSI.

É uma das linguagens mais usadas para o ensino para iniciantes em programação e portabilidade, tanto que dela se originou uma outra linguagem, a C++, utilizada atualmente por grandes empresas, implementando assim um novo paradigma de programação, orientada a objeto, adicionando novas funcionalidades que limitavam a linguagem C.

2.5.3 *Clean Code*

A decisão de utilizar a técnica de nomenclatura para código, *Clean Code*, é vantajosa, pois, melhora a compreensão da leitura de desenvolvedores para além do próprio autor. Utilizando nomes das variáveis, classes, métodos, funções, a fim de que se possa entender seu significado (MARTIN, 2008). Seguindo algumas regras, pode-se criar um programa de fácil entendimento e manutenção, palavras com significados únicos para cada função estrutural, transformando o código em um programa autoexplicável, otimizando linhas e repetições, mas sem perder seu objetivo. Segue algumas orientações para a implementação e boas práticas (MARTIN, 2008):

- **Seguir padrões de projeto**: para a criação de uma padronização das nomenclaturas utilizadas no código, utiliza-se de um nome de variável que se inicie em minúsculo, e caso tenha um complemento, diferencie com a inicial maiúscula.
- **Evitar repetições em excesso**: a fim de se obter um código com o número de linhas reduzido para melhor desempenho, prefira funções curtas e diretas, retornando apenas um valor.
- **Redução do uso de comentários**: por se tratar de um método de entendimento mais rápido na leitura do código, orienta-se comentar somente se for algo relevante e necessário.
- **Execução de testes limpos:** os testes devem ser realizados em pequenos blocos independentes, quantas vezes for necessário, a fim de se validar cada passo realizado para se chegar no objetivo, evitando testar o programa somente no final da aplicação.

3 METODOLOGIA DE PESQUISA

Tópico dedicado para abordagem da metodologia de pesquisa aplicada neste trabalho.

3.1 CLASSIFICAÇÃO DA PESQUISA

Segundo Gil (1999), a pesquisa pode ser diferenciada quanto à natureza, aos métodos ou abordagens metodológicas, quanto aos objetivos e quanto aos procedimentos. Sendo assim, essa pesquisa se classifica da seguinte maneira:

- Quanto à sua natureza: a pesquisa aplicada é dedicada para a solução de problemas específicos, em busca da verdade para aplicação prática. O presente estudo se dá pela análise de heurísticas escolhidas perante uma programação de tarefas em máquinas, onde poderá ser aplicado em ambientes fabris.
- Quanto aos métodos: se trata de uma abordagem quantitativa, pois emprega medidas sistemáticas, facilitando a comparação e a análise das heurísticas através de parâmetros mediante *makespan* e *flowtime*.
- Quanto aos objetivos: o estudo pode ser tanto exploratório, quanto descritivo. Exploratório, pois serão usadas pesquisas bibliográficas e estudos de casos para a compreensão do problema, de acordo com Vergara (2000) a pesquisa exploratória é realizada em área na qual há pouco conhecimento acumulado e sistematizado, ou seja, tem como objetivo aprimorar ideias, levantar hipóteses sobre assuntos pouco explicados; e descritivo, pois irá buscar a correlação entre variáveis, abordando diferentes heurísticas e analisando o comportamento das mesmas diante de funções objetivo.
- Quanto aos procedimentos: se trata de procedimentos técnicos como pesquisa bibliográfica, que segundo Gil (1999), é um trabalho de natureza exploratória, que propicia bases teóricas ao pesquisador para auxiliar no exercício reflexivo e crítico sobre o tema em estudo existentes na literatura (artigos, livros e periódicos). Pode-se dizer também que a pesquisa é experimental, pois

também haverá a experimentação computacional afim de se analisar os resultados.

3.2 LEVANTAMENTO DE DADOS

Para a realização deste estudo, escolheu-se os 120 problemas de Taillard (1993), reconhecidos amplamente na literatura, utilizados, por exemplo, no artigo "*A heuristic algorithm for scheduling in a flow shop environment to minimize makespan"* (Gupta, 2015), em que ele se utiliza dos dados para comparar heurísticas diferentes das tratadas nesse estudo. Esses problemas se referem tanto a tempos de processamento quanto a tempos de setup em cada atividade em determinada máquina. Esses problemas consistem em um conjunto de instâncias que possuem entre 20 e 500 tarefas e 5 à 20 máquinas, conforme a Tabela 2, foi separado de acordo com os seguintes arquivos salvos em formato de texto.

Tabela 2 – Tabela da sequência dos problemas de Taillard (1993)

MÁQUINA/TAREFA	ARQUIVOS SALVOS
5X20	1-10
10X20	11-20
20X20	21-30
5X50	31-40
10X50	41-50
20X50	51-60
5X100	61-70
10X100	71-80
20X100	81-90
10X200	91-100
20X200	101-110
20X500	111-120

Fonte: Autoria Própria (2022)

Após a numeração dos arquivos, foi criado através do software Dev-Pascal (versão 1.9.2) um código de ordenação, na linguagem de programação Pascal, disponibilizado no Apêndice A, e um segundo código de ordenação através do software Clion (versão 2022.2.4), na linguagem C, disponibilizado no Apêndice B. Foi realizado a programação e os testes com um computador Windows 10, Sistema Operacional 64 bits, Processador Intel® Core™ i5 e de memória RAM de 8,00 GB), realizando a ordenação dos dados com os três métodos apresentados (LPT, SPT e NEH), com os três sistemas também apresentados anteriormente (clássico, *no wait* e

no idle), resultando os valores em um arquivo de texto de saída representando as duas funções objetivo (*Makespan* e *Flowtime*), e respectivamente, o tempo de processamento de cada uma. Foi feito a execução de ambos os códigos, sendo cada um deles salvos em dois arquivos diferentes em Excel, totalizando um total de 48 execuções com os 120 valores.

Com os dados armazenados em ambos arquivos em Excel, um para os resultados do código em Pascal, e um segundo para o código em C, foram realizados os cálculos a fim de descobrir a eficiência de cada método, por meio de suas porcentagens de sucesso e desvios relativos com a construção de gráficos para melhor visualização, e por fim uma comparação dos tempos de processamento entre as duas linguagens.

4 DESENVOLVIMENTO

Após a obtenção dos resultados dos respectivos algoritmos Pascal e C, os dados foram analisados com o objetivo de comparação do comportamento de cada método, onde nessa comparação, ambos possuem o mesmo resultado. Posteriormente, foi feita a comparação dos tempos de processamento de cada linguagem de programação, em que há diferenças no resultado em relação aos desempenhos relacionados ao tempo.

Para a análise foram agrupados os mesmos arquivos que possuem a mesma quantidade de tarefas e máquinas, sendo 10 (dez) arquivos de cada agrupamento, sendo o total 12 (doze) agrupamentos, conforme a Tabela 2.

As estatísticas de desempenho se deram por meio de Porcentagem de Sucesso e pelo Desvio Relativo Médio (DRM), feitos com a ajuda da ferramenta Excel 365, para melhor visualização dos resultados.

A Porcentagem de Sucesso é definida pela comparação entre o número total de problemas tratados e pelo menor valor de *makespan* ou *flowtime* apresentado por cada método, ou seja, se ambos valores são iguais é considerado como verdadeiro, igualando à 1 (um), caso contrário, é considerado como falso, igualando à 0 (zero).

Quanto ao Desvio Relativo Médio se dá pela subtração do valor do método proposto pelo menor valor encontrado, e dividido pelo menor valor encontrado, conforme a equação (1):

$$DRM = \frac{(Hm - Hv)}{Hv} \quad (1)$$

Onde, Hm é o valor do resultado encontrado pelo método tratado, e Hv é o menor valor encontrado dentre os métodos, para determinada função objetivo *makespan* ou *flowtime*.

4.1 ANÁLISE DOS RESULTADOS

Os gráficos que serão apresentados na sequência mostram os dados obtidos pelos cálculos descritos anteriormente, a fim de se comparar os métodos LPT, SPT e NEH de acordo com os valores das funções objetivos e em relação a cada sistema, sendo eles: clássico, *no idle* e *no wait*; assim como o tempo de processamento de cada análise realizada.

Os resultados estão divididos pelas diferentes linguagens e posteriormente irá se comparar os tempos de processamento de ambos.

4.1.1 COMPARAÇÃO DOS MÉTODOS

Conforme foi descrito anteriormente, foi realizado a comparação dos resultados por meio dos gráficos de sucesso e de desvio relativo médio para um melhor entendimento dos desempenhos de cada método proposto.

4.1.1.1 GRÁFICOS DE SUCESSO

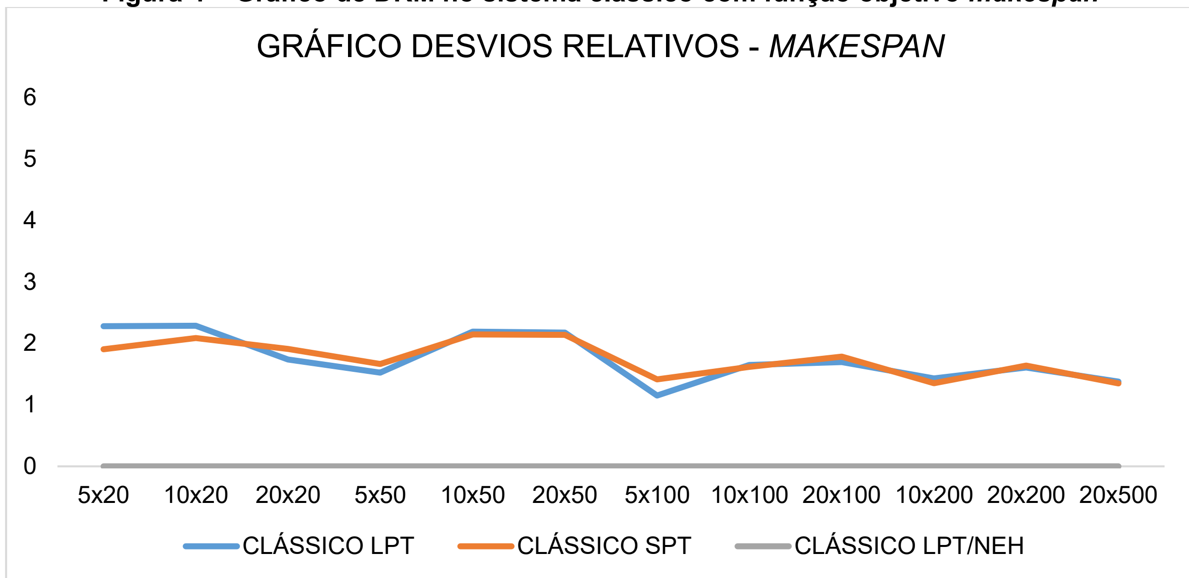
Ao comparar todos os resultados através dos gráficos de sucesso, conforme pode-se constatar o sucesso equivalente a 100% para os métodos em que o NEH é incrementado como segunda parte da ordenação, onde faz conjunto respectivamente com as funções objetivo *makespan*, sendo o conjunto LPT/NEH, e com a função objetivo *flowtime*, com o conjunto SPT/NEH.

Em todos os sistemas analisados, clássico, *no idle* e *no wait*, o método NEH atingiu o mesmo resultado, sendo assim todos os gráficos referentes ao sucesso seriam iguais, portanto não se faz necessário colocá-los.

4.1.1.2 GRÁFICOS DOS DESVIOS RELATIVOS MÉDIOS

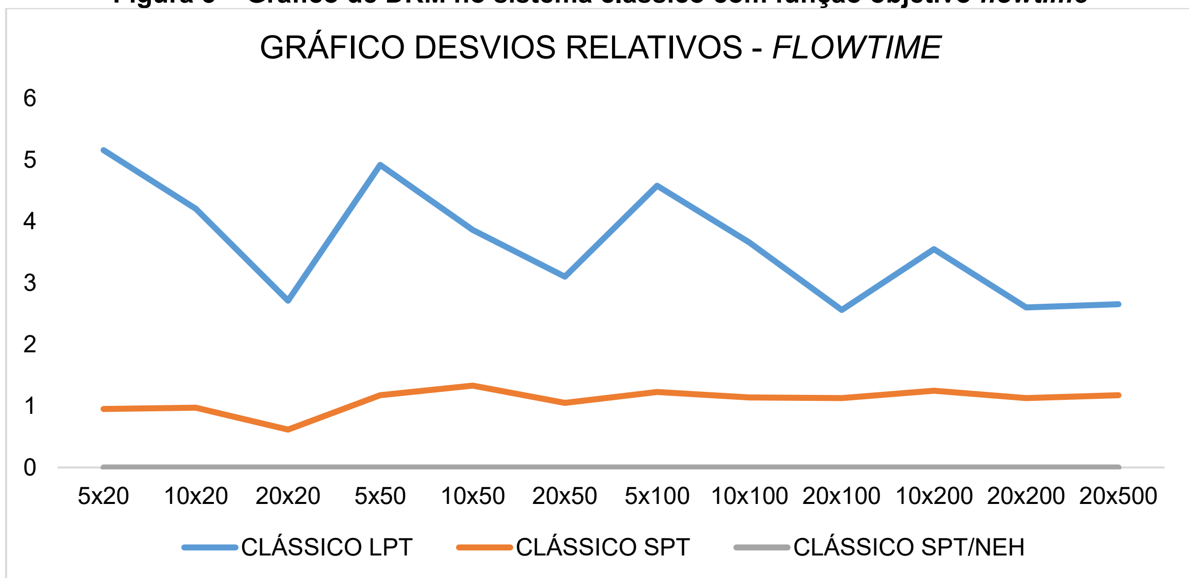
Analisando os gráficos das Figuras 4 à 9, é constatado o destaque do método NEH que possui um desvio relativo igual à zero para todos os sistemas em relação aos demais métodos. Já na variância dos desvios relativos médios, os métodos LPT e SPT se revezam em termos de melhor performance.

Figura 4 – Gráfico de DRM no sistema clássico com função objetivo *makespan*



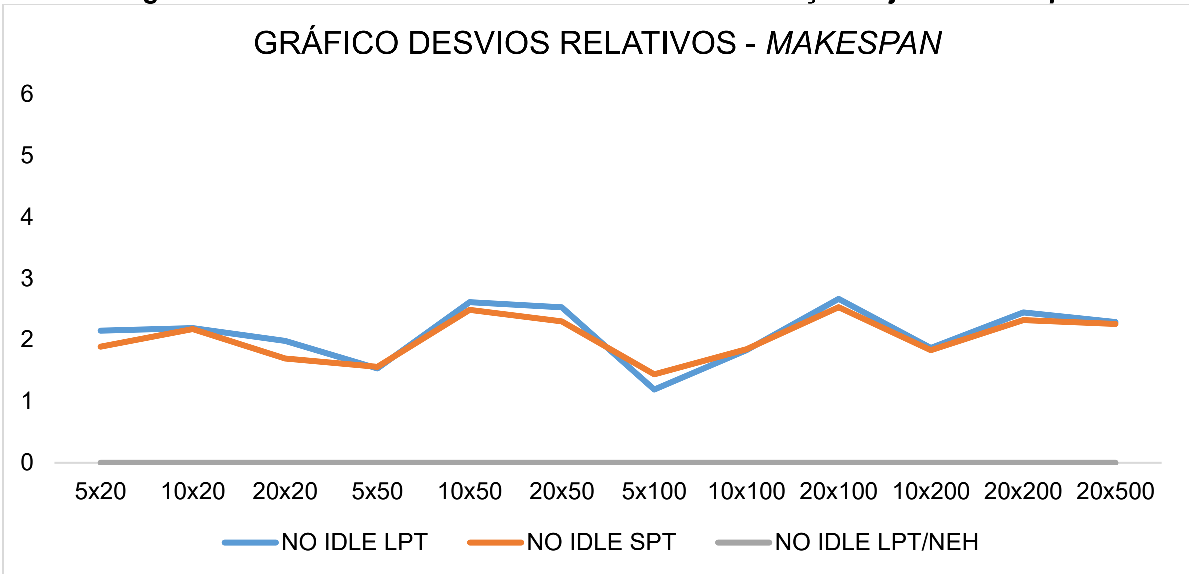
Fonte: Autoria Própria (2022)

Figura 5 – Gráfico de DRM no sistema clássico com função objetivo *flowtime*



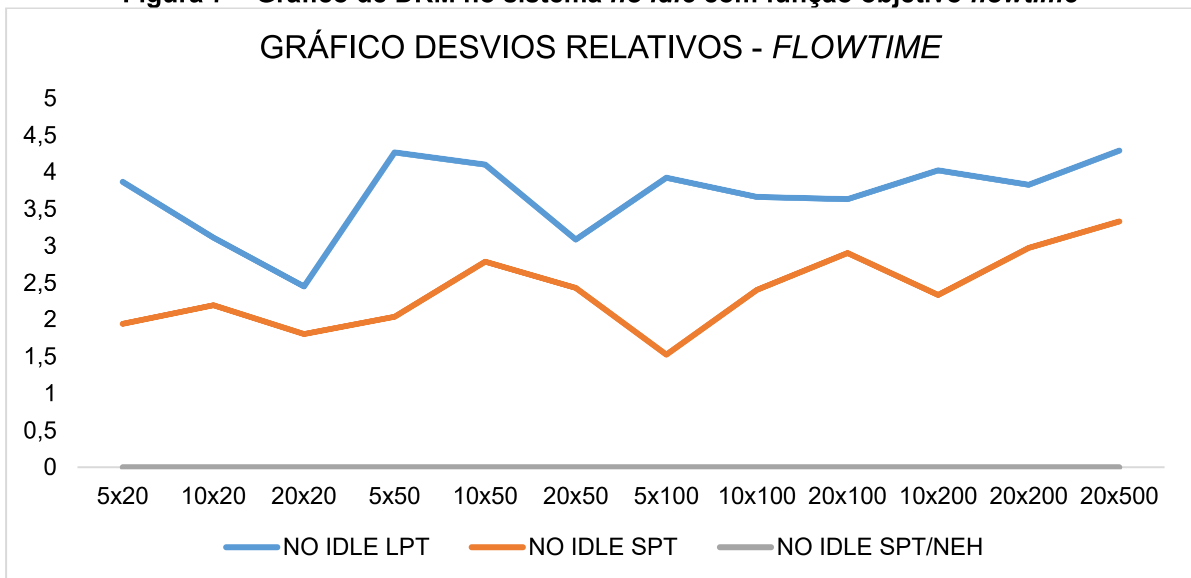
Fonte: Autoria Própria (2022)

Figura 6 – Gráfico de DRM no sistema *no idle* com função objetivo *makespan*



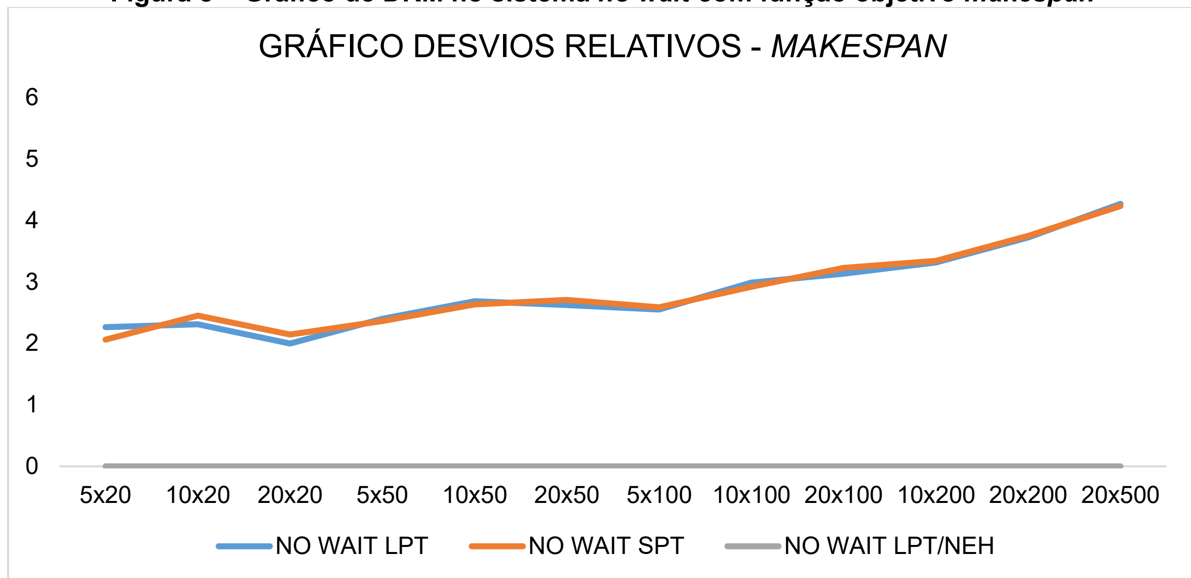
Fonte: Autoria Própria (2022)

Figura 7 – Gráfico de DRM no sistema *no idle* com função objetivo *flowtime*



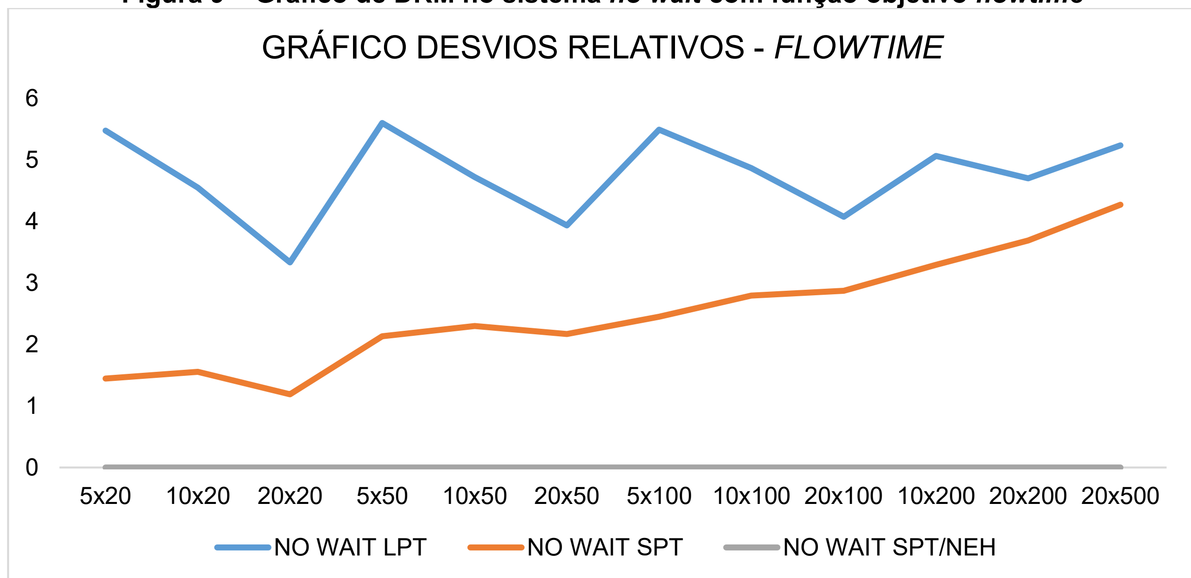
Fonte: Autoria Própria (2022)

Figura 8 – Gráfico de DRM no sistema *no wait* com função objetivo *makespan*



Fonte: Autoria Própria (2022)

Figura 9 – Gráfico de DRM no sistema *no wait* com função objetivo *flowtime*



Fonte: Autoria Própria (2022)

De forma geral, os desvios relativos representam os tempos das funções objetivo, nos quais os gráficos de *flowtime* mostram uma alteração maior nos valores entre os métodos, e nos gráficos de *makespan*, indicam o oposto.

4.1.2 COMPARAÇÃO DOS TEMPOS DE PROCESSAMENTO

Em problemas relacionados ao *flow shop* permutacional, todas as tarefas possuem um mesmo fluxo de processamento nas respectivas máquinas. Como este processo pode levar uma grande quantidade de tempo de processamento durante a

execução, foi desenvolvido dois programas de linguagem de alto nível já citados anteriormente, nos quais pode-se analisar se houve uma melhora nesse tempo de execução entre os dois códigos. Para a contagem de tempo, utilizou-se em ambos os códigos funções de contagem referentes a execução do programa em milissegundos, determinando a eficiência dos códigos em questão.

4.1.2.1 NA LINGUAGEM PASCAL

O código implementado, foram gerados os métodos de acordo com as funções objetivo em cada sistema, em cada bloco de instância, tiraram-se também as médias do tempo de processamento dos arquivos, por meio destes cálculos, pode consolidar nas Tabelas 3 (três), 4 (quatro) e 5 (cinco) separadas por cada sistema.

Tabela 3 – Dados de tempo de processamento na linguagem Pascal para o sistema clássico

CLÁSSICO	*MAKESPAN*				*FLOWTIME*		
	LPT	**SPT**	**LPT/NEH**		**LPT**	**SPT**	**SPT/NEH**
5x20	0	1,5	0		0	1,5	1,6
10x20	1,6	0	1,6		1,5	1,6	6,2
20x20	1,5	1,6	1,6		1,6	1,6	3,2
5x50	1,6	1,6	6,2		1,5	3,1	6,2
10x50	26,6	3,1	6,3		3,2	4,7	9,4
20x50	1,5	3,1	12,5		3,1	3,1	15,6
5x100	4,7	4,7	18,7		4,7	6,3	23,5
10x100	3,1	3,1	32,8		3,1	4,6	40,6
20x100	4,7	4,7	64,1		4,7	6,3	75
10x200	7,8	7,8	212,5		7,8	9,4	260,9
20x200	7,8	7,8	434,4		7,8	10,9	489,1
20x500	21,9	20,4	6401,5		21,9	26,6	7410,9

Fonte: Autoria Própria (2022)

Tabela 4 – Dados de tempo de processamento na linguagem Pascal para o sistema *no idle*

NO IDLE	*MAKESPAN*				*FLOWTIME*		
	LPT	**SPT**	**LPT/NEH**		**LPT**	**SPT**	**SPT/NEH**
5x20	1,5	1,6	1,6		1,6	1,5	1,6
10x20	1,6	0	1,6		1,6	1,6	7,8
20x20	1,5	1,5	1,5		0	1,5	3,1
5x50	4,7	4,7	4,7		4,6	4,7	6,3
10x50	3,2	3,2	7,8		4,7	4,7	9,4
20x50	4,6	3,1	12,5		3,2	3,1	14
5x100	6,3	4,7	18,8		6,2	6,3	21,9
10x100	4,7	6,2	31,2		6,3	6,2	35,9
20x100	6,2	4,7	70,3		6,2	6,3	70,3
10x200	11	9,4	207,8		12,5	10,9	212,5

20x200	10,9	10,9	518,8		10,9	12,5	517,2
20x500	26,6	25	9503,1		25	29,7	9182,8

Fonte: Autoria Própria (2022)

Tabela 5 – Dados de tempo de processamento na linguagem Pascal para o sistema *no wait*

NO WAIT	*MAKESPAN*				*FLOWTIME*		
	LPT	**SPT**	**LPT/NEH**		**LPT**	**SPT**	**SPT/NEH**
5x20	0	3,1	3,1		1,6	1,5	1,5
10x20	1,6	1,6	1,5		1,5	1,6	1,6
20x20	1,6	1,5	0		0	1,6	1,6
5x50	3,1	4,7	6,3		3,1	1,5	6,2
10x50	3,1	4,7	3,1		3,2	4,7	3,1
20x50	3,1	4,7	4,7		3,1	3,1	4,7
5x100	4,7	6,2	9,4		4,7	6,3	9,4
10x100	6,3	7,9	10,9		4,7	6,2	10,9
20x100	6,2	7,8	12,5		6,2	7,8	11
10x200	12,5	17,2	37,5		11	14,1	39
20x200	15,7	17,1	40,6		14	17,2	40,7
20x500	54,6	62,5	407,9		54,7	62,5	368,7

Fonte: Autoria Própria (2022)

De acordo com os tempos apresentados, quanto mais tarefas são colocadas no cálculo, tende a ser maior o tempo de execução, sobretudo nos métodos em que o NEH está presente, onde os tempos de processamento aumentaram drasticamente.

4.1.2.2 NA LINGUAGEM C

A mesma análise é feita para o código implementado em C, dividido nas mesmas instâncias e funções objetivos para cada método analisado conforme as Tabelas 6 (seis), 7 (sete) e 8 (oito):

Tabela 6 – Dados de tempo de processamento na linguagem C para o sistema clássico

CLÁSSICO	*MAKESPAN*				*FLOWTIME*		
	LPT	**SPT**	**LPT/NEH**		**LPT**	**SPT**	**SPT/NEH**
5x20	3,2	3,2	3,1		4,8	1,6	3,1
10x20	1,6	1,6	3,1		6,3	4,6	3,1
20x20	4,7	4,6	4,7		4,7	1,5	3,2
5x50	3,1	3,2	6,3		3,1	3,2	6,2
10x50	4,7	12,5	6,2		3,2	4,6	7,8
20x50	1,5	4,6	12,5		12,5	3,1	11
5x100	3,1	6,2	18,8		4,7	1,6	17,1
10x100	1,6	4,7	37,5		9,3	1,5	31,3
20x100	4,7	6,3	89		4,7	4,7	62,5
10x200	4,7	6,2	195,3		15,7	4,6	200
20x200	4,7	4,7	500		10,9	7,9	421,8

20x500	9,4	18,7	5659,4		15,5	9,4	6117,2

Fonte: Autoria Própria (2022)

Tabela 7 – Dados de tempo de processamento na linguagem C para o sistema *no idle*

NO IDLE	*MAKESPAN*				*FLOWTIME*		
	LPT	**SPT**	**LPT/NEH**		**LPT**	**SPT**	**SPT/NEH**
5x20	10,9	4,7	3,1		4,7	7,8	3,2
10x20	6,1	4,6	6,2		14,1	14,1	4,6
20x20	4,7	4,8	6,3		7,8	3,2	6,3
5x50	6,3	4,7	6,2		7,8	6,2	6,2
10x50	9,3	4,8	9,3		6,3	12,5	11
20x50	4,7	6,3	3,2		6,3	4,7	6,2
5x100	6,4	3,1	21,9		6,2	1,6	15,7
10x100	3,2	3,2	31,3		4,7	4,7	6,2
20x100	4,7	6,2	32,9		4,6	1,5	32,8
10x200	3,2	4,7	150		3,1	1,6	153
20x200	3,2	4,8	417,3		4,8	9,4	476,6
20x500	4,8	4,7	6507,7		4,6	4,8	6491,1

Fonte: Autoria Própria (2022)

Tabela 8 – Dados de tempo de processamento da linguagem C para o sistema *no wait*

NO WAIT	*MAKESPAN*				*FLOWTIME*		
	LPT	**SPT**	**LPT/NEH**		**LPT**	**SPT**	**SPT/NEH**
5x20	6,2	4,7	6,3		3,1	1,6	1,5
10x20	4,8	3,1	4,7		1,6	6,3	3,1
20x20	3,1	1,5	6,3		12,5	4,7	3,2
5x50	1,6	1,5	7,8		6,3	6,3	4,7
10x50	4,7	6,3	4,6		1,5	10,9	3,2
20x50	4,8	1,6	3,2		1,5	3	4,5
5x100	3,1	7,8	3,1		3,1	3,1	6,2
10x100	3,1	1,6	9,4		3,1	6,2	12,5
20x100	6,3	6,3	11		9,4	4,8	6,2
10x200	9,3	7,9	4,8		9,3	4,7	14,2
20x200	9,4	4,8	26,5		9,3	10,9	18,7
20x500	7,9	9,4	176,6		12,5	6,3	196,9

Fonte: Autoria Própria (2022)

Assim como apresentado anteriormente pela outra linguagem, os valores dos tempos de processamento também demonstram um aumento nos métodos em que o NEH se encontra. Porém, nas tabelas com os tempos gerados na linguagem C, pode-se notar visualmente uma redução nos tempos apresentados.

4.1.2.3 CÁLCULO DA PORCENTAGEM DE MELHORIA

A fim de se obter mais precisamente o quanto teve de diferença entre os tempos de processamento nas execuções dos códigos implementado em Pascal e em C, obteve-se o seguinte cálculo da porcentagem em cada sistema, e respectivamente para cada instância, conforme a equação (2):

$$\text{Variação percentual} = \frac{Tc - Tp}{Tc} \times 100 \qquad (2)$$

Onde, Tp seria o tempo de processamento de execução em Pascal, e Tc o tempo de processamento de execução em C.

Após feitos os cálculos, é feita a média de cada instância e de cada método, a fim de se observar melhor as variações percentuais, conforme mostra nas Tabelas 9 (nove), 10 (dez) e 11 (onze) para cada tipo de sistema.

Tabela 9 – Cálculo da variação percentual de tempo de processamento entre as linguagens de programação no sistema clássico

CLÁSSICO	*MAKESPAN*			*FLOWTIME*			
	LPT	**SPT**	**LPT/NEH**	**LPT**	**SPT**	**SPT/NEH**	**MÉDIA**
5x20	100%	53%	100%	100%	6%	48%	68%
10x20	0%	100%	48%	76%	65%	-100%	32%
20x20	68%	65%	66%	66%	-7%	0%	43%
5x50	48%	50%	2%	52%	3%	0%	26%
10x50	-466%	75%	-2%	0%	-2%	-21%	-69%
20x50	0%	33%	0%	75%	0%	-42%	11%
5x100	-52%	24%	1%	0%	-294%	-37%	-60%
10x100	-94%	34%	13%	67%	-207%	-30%	-36%
20x100	0%	25%	28%	0%	-34%	-20%	0%
10x200	-66%	-26%	-9%	50%	-104%	-30%	-31%
20x200	-66%	-66%	13%	28%	-38%	-16%	-24%
20x500	-133%	-9%	-13%	-41%	-183%	-21%	-67%
MÉDIA	-55%	30%	21%	39%	-66%	-22%	

Fonte: Autoria Própria (2022)

Tabela 10 – Cálculo da variação percentual de tempo de processamento entre as linguagens de programação no sistema *no idle*

NO IDLE	*MAKESPAN*			*FLOWTIME*			
	LPT	**SPT**	**LPT/NEH**	**LPT**	**SPT**	**SPT/NEH**	**MÉDIA**
5x20	86%	66%	48%	66%	81%	50%	66%
10x20	74%	100%	74%	89%	89%	-70%	59%
20x20	68%	69%	76%	100%	53%	51%	70%
5x50	25%	0%	24%	41%	24%	-2%	19%
10x50	66%	33%	16%	25%	62%	15%	36%

20x50	2%	51%	-291%	49%	34%	-126%	-47%
5x100	2%	-52%	14%	0%	-294%	-39%	-62%
10x100	-47%	-94%	0%	-34%	-32%	-479%	-114%
20x100	-32%	24%	-114%	-35%	-320%	-114%	-99%
10x200	-244%	-100%	-39%	-303%	-581%	-39%	-218%
20x200	-241%	-127%	-24%	-127%	-33%	-9%	-94%
20x500	-454%	-432%	-46%	-443%	-519%	-41%	-323%
MÉDIA	-58%	-39%	-22%	-48%	-120%	-67%	

Fonte: Autoria Própria (2022)

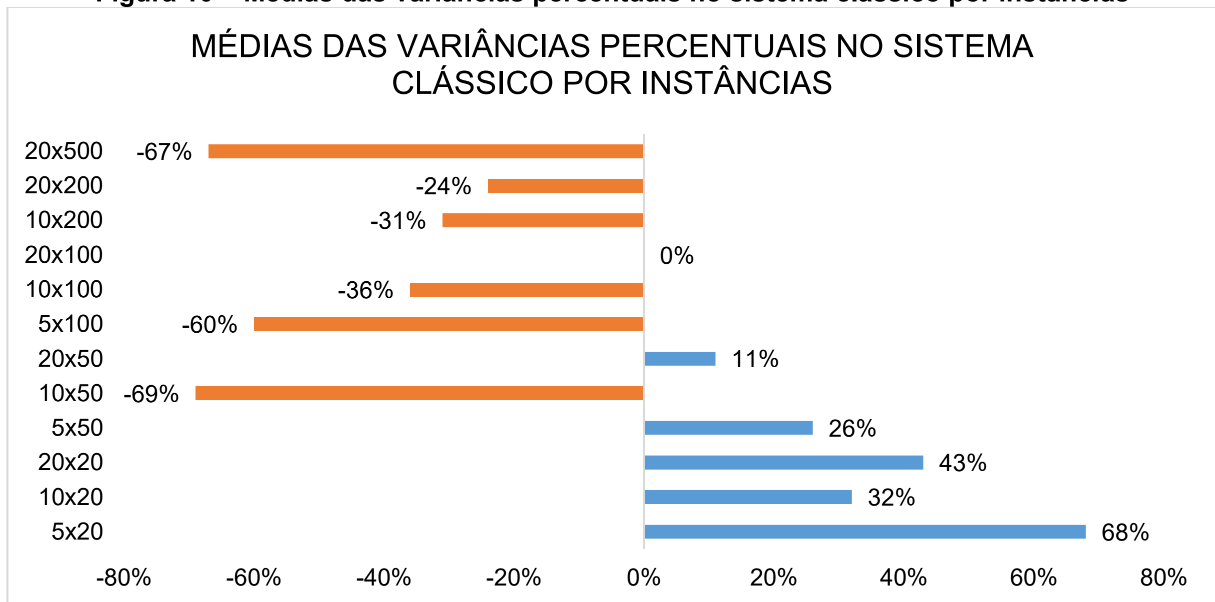
Tabela 11 – Cálculo da variação percentual de tempo de processamento entre as linguagens de programação no sistema *no wait*

NO WAIT	*MAKESPAN*			*FLOWTIME*			
	LPT	**SPT**	**LPT/NEH**	**LPT**	**SPT**	**SPT/NEH**	**MÉDIA**
5x20	100%	34%	51%	48%	6%	0%	40%
10x20	67%	48%	68%	6%	75%	48%	52%
20x20	48%	0%	100%	100%	66%	50%	61%
5x50	-94%	-213%	19%	51%	76%	-32%	-32%
10x50	34%	25%	33%	-113%	57%	3%	7%
20x50	35%	-194%	-47%	-107%	-3%	-4%	-53%
5x100	-52%	21%	-203%	-52%	-103%	-52%	-74%
10x100	-103%	-394%	-16%	-52%	0%	13%	-92%
20x100	2%	-24%	-14%	34%	-63%	-77%	-24%
10x200	-34%	-118%	-681%	-18%	-200%	-175%	-204%
20x200	-67%	-256%	-53%	-51%	-58%	-118%	-101%
20x500	-591%	-565%	-131%	-338%	-892%	-87%	-434%
MÉDIA	-55%	-136%	-73%	-41%	-87%	-36%	

Fonte: Autoria Própria (2022)

Feito os cálculos das variâncias percentuais dos tempos, são observadas oscilações nos tempos, e para que se possa visualizar melhor, são feitas as médias dessas variâncias, tanto por método, quanto por instâncias, conforme os próximos gráficos das Figuras 10 (dez) a 13 (treze). Nas figuras, são representadas pela cor azul um aumento nos tempos, e na cor laranja, uma redução dos tempos entre uma linguagem Pascal e a linguagem C.

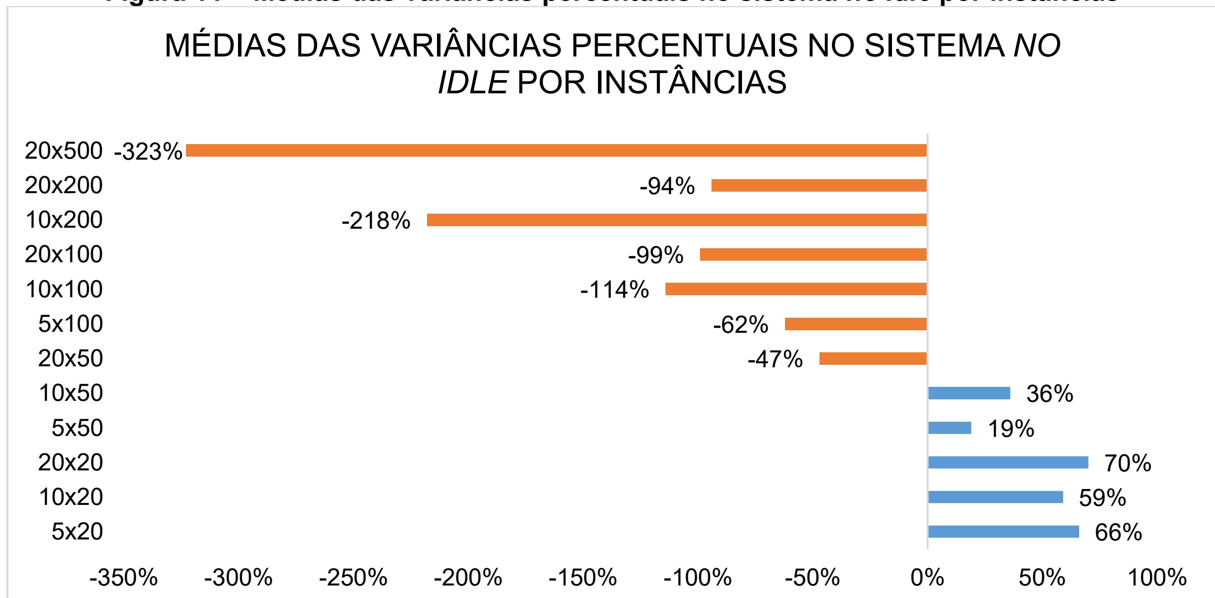
Figura 10 – Médias das variâncias percentuais no sistema clássico por instâncias



Fonte: Autoria Própria (2022)

No sistema clássico, conforme a Figura 11 (onze), houve uma redução para as instâncias que possuem mais tarefas a serem processadas, o que indica uma melhora na questão na linguagem de programação em C. Já nas outras instâncias, com o menor número de tarefas, houve um aumento da variância percentual.

Figura 11 – Médias das variâncias percentuais no sistema *no idle* por instâncias

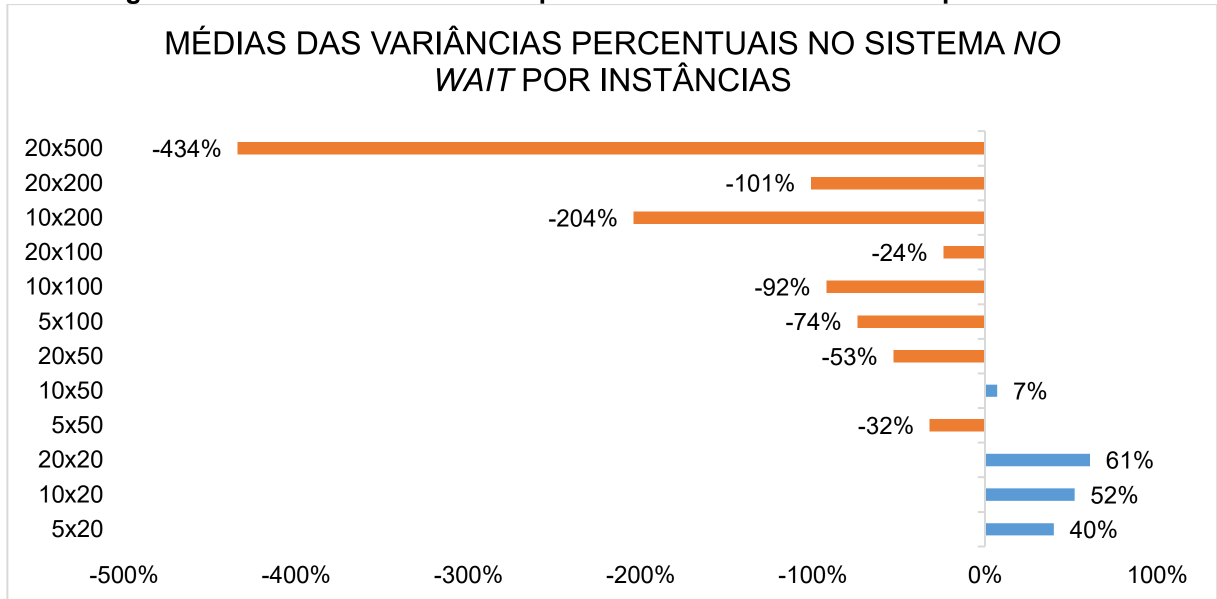


Fonte: Autoria Própria (2022)

Para o sistema *no idle*, conforme pode-se observar na Figura 11 (onze), acerca das instâncias, assim como no sistema clássico, se dá uma melhora nos tempos de

processamentos nas que contém elevado número de tarefas, com destaque para a última instância que possui mais de quinhentas tarefas, com redução acima de 300%.

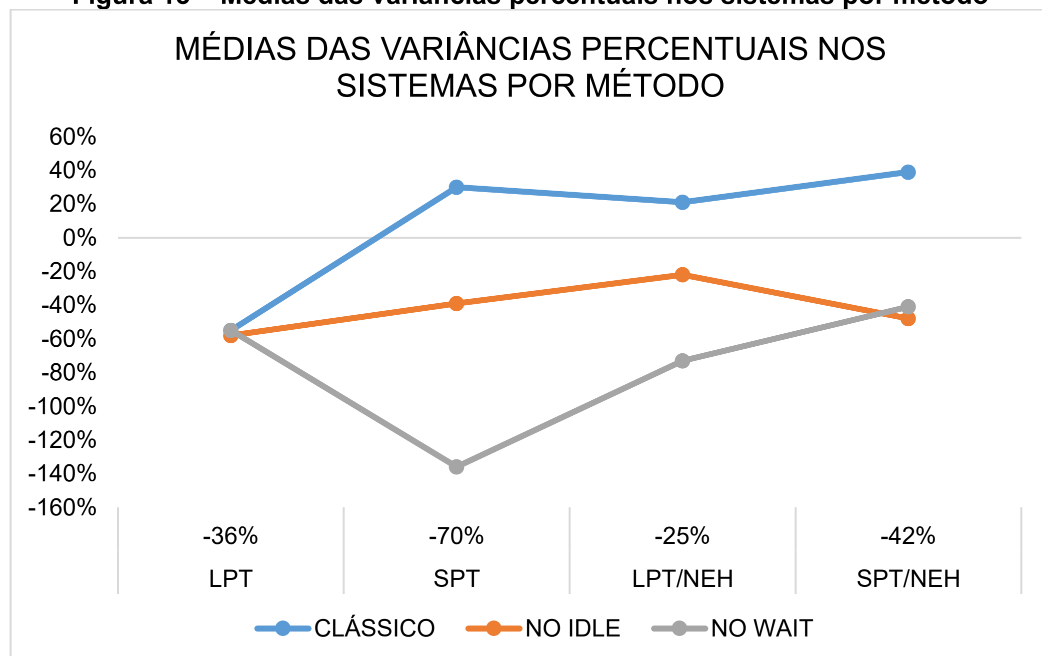
Figura 12 – Médias das variâncias percentuais no sistema *no wait* por instâncias



Fonte: Autoria Própria (2022)

No que se refere ao sistema *no wait*, posto na Figura 12 (doze), se atinge os mesmos resultados que no sistema *no idle*, em que se constata uma redução nos tempos de processamento em todas as instâncias com maiores números de tarefas.

Figura 13 – Médias das variâncias percentuais nos sistemas por método



Fonte: Autoria Própria (2022)

Em relação aos métodos, conforme a Figura 13 (treze), pode-se observar uma importante variação entre as duas linguagens. No método LPT, se observa uma redução de 36% e de 25% em conjunto com o NEH, indicando a menor melhora entre os métodos. Já para o método SPT, se percebe a maior queda entre os tempos, tendo a melhor melhora em relação à execução da heurística, no valor de 70% e 42% em conjunto com o NEH.

De modo geral, pode-se concluir que, ao se calcular a média ampla das variâncias percentuais por instâncias, chega-se a uma redução de 36% com destaque para as instâncias com as maiores tarefas.

Ao avaliar uma média ampla das variâncias percentuais em relação aos métodos, pode-se chegar em uma redução de 43% nos tempos de processamento, com destaque para o método SPT e NEH.

5 CONCLUSÃO

Os problemas de *scheduling* tem como por objetivo a busca pelas soluções e melhorias nas heurísticas utilizadas para a minimização do *maskespan* e do *flowtime*. Este trabalho buscou analisar, além dos tempos demandados nas máquinas com suas respectivas tarefas, os tempos de processamento na execução dos métodos por diferentes linguagens de programação. A otimização desses tempos são importantes para a indústria, assim como a definição do melhor sistema para os seus processos, por isso a importância deste estudo.

Pode-se constatar por meio deste trabalho, que o método construtivo NEH utilizado como segunda fase com os métodos LPT ou SPT, com as melhores taxas de sucesso e de desvios relativos médios iguais à zero, representando uma diminuição significativa no tempo total gasto até a última tarefa, em comparação com os métodos citados com somente uma fase. Com o método SPT também se mostrou uma grande redução nos tempos de processamento.

Com as linguagens de programação propostas, Pascal e C, é possível identificar que existe uma diferença de desempenho entre os tempos de processamento das linguagens, sobretudo para instâncias que possuem muitas tarefas para um conjunto de máquinas. Em relação aos métodos, com exceção do sistema clássico, onde no método LPT demonstrou uma piora no tempo de processamento, pode-se observar pela média da variância percentual uma grande redução no tempo de processamento com a implementação do código na linguagem C.

Com isso, como recomendações de trabalhos futuros, pode-se afirmar a utilização de experimentações com outros problemas envolvendo máquinas e tarefas (simulados ou reais) trabalhar na otimização e redução de tempo de instruções no código proposto por este trabalho a fim de se buscar uma redução nos métodos, utilizando de técnicas avançadas de programação para que novas comparações sejam feitas.

REFERÊNCIAS

BACKES, André **Linguagem C : completa e descomplicada**. - 2. ed. - Rio de Janeiro: Elsevier, 2019. : il.

BRANCO, Fábio José Ceron. **Um novo método heurístico construtivo de alto desempenho para o problema no-idle flow shop.** 2011. Dissertação (Doutorado) – Escola de Engenharia de São Carlos, Departamento de Engenharia de Produção, Universidade de São Paulo. São Carlos, 2011.

BRANCO, Fábio José Ceron; NAGANO, Marcelo Seido; MOCCELLIN, João Vitor. **Avaliação de métodos heurísticos construtivos para o problema de programação de operações no-wait flow shop.** Revista Produção, Florianópolis - SC, 2008.

BRANCO, Fábio José Ceron; NAGANO, Marcelo Seido; MOCCELLIN, João Vitor. Soluções de alto desempenho para a programação da produção flow shop. **GEPROS. Gestão da Produção, Operações e Sistemas** – Ano 4, nº 2, Abr-Jun/2009, p.11-23. Disponível em: <https://revista.feb.unesp.br/index.php/gepros/article/viewFile/743/223> Acesso em: 20 de novembro de 2022.

COELHO, Pedro; SILVA, Cristovão. Parallel Metaheuristics for shop scheduling: enabling industry 4.0. **Procedia Computer Science**, v. 180, p. 778-786, 2021.

DASGUPTA, Sanjoy; Papadimitriou, Christos; Vazirani, Umesh. **Algoritmos.** Porto Alegre: AMGH, 2010.

DEMAN, J. M. V.; BAKER, K. R. **Minimizing mean flowtime in the flow shop with no intermediate queues**. AIIE Transactions, v. 6, pp. 28-34, 1974.

FUCHIGAMI, Hélio Y. **Métodos Heurísticos Construtivos Para o Problema de Programação da Produção em Sistemas Flowshop Híbridos com Tempos de Preparação das Máquinas Assimétricos e Dependentes da Sequência**. 2005.

135f. Dissertação (Mestreado em Engenharia de Produção) – Escola de Engenharia de São Carlos, Universidade São Paulo, São Carlos, 2005.

GIL, Antonio Carlos. (1999). **Métodos e técnicas de pesquisa social**. São Paulo: Atlas, p.42.

GOMES, Guilherme H. G. **Análise De Desempenho De Algoritmo Genético Em Diferentes Linguagens E Ambientes De Programação.** Monografia (Graduação) – Universidade Federal de Ouro Preto. Escola de Minas. Departamento de Engenharia de Produção, Administração e Economia, 2015.

GRIFFITHS, Dawn; GRIFFITHS, David. **Use a Cabeça! C**. Alta Books Editora, 2013.

GUPTA, ARUN & CHAUHAN, Sant. (2015). A heuristic algorithm for scheduling in a flow shop environment to minimize makespan. **International Journal of Industrial Engineering Computations.** 6. 173-184. 10.5267/j.ijiec.2014.12.002.

MACCARTHY, Bart L.; LIU, Jiyin. Addressing the gap in scheduling research: a review of optimization and heuristic methods in production scheduling. **The International Journal of Production Research**, v. 31, n. 1, p. 59-79, 1993.

MARTIN, R. C. Clean Code: **A Handbook of Agile Software Craftsmanship**. [S.l.]: Prentice Hall, 2008. ISBN 0132350882.

NAWAZ, M.; ENSCORE JR., E.E.; HAM, I. **A heuristic algorithm for the m-machine, n-job flow-shop se-quencing problem.** OMEGA – The International Journal of Management Science, v.11, pp. 91-95, 1983.

NETO, Bernardo Henrique Olbertz (2016). **Análise de Métodos Heurísticos para Minimização do Tempo Total da Programação de Operações no Problema Flow Shop Permutacional.** Disponível em: <http://repositorio.roca.utfpr.edu.br/jspui/bitstream/1/7376/1/PG_COCIC_2016_2_03.pdf>. Acesso em: 30 de setembro de 2022.

SLACK, N.; **Administração da Produção**. 1.ed. São Paulo: Atlas. 1999.

TAILLARD, E. **Benchmarks for basic scheduling problems**. European Journal of Operational Research, p. 278-285,1993.

TAILLARD, E. **Problem instances**.
Disponível em: <http://ina.eivd.ch/collaborateurs/etd/default.htm> Acesso em: 20 de novembro de 2022.

THESAURUS, 2016. **Metodologia da Pesquisa Científica: teoria e prática – como elaborar TCC**. Brasília. Disponível em:
<http://franciscopaulo.com.br/arquivos/Classificação%20da%20Pesquisa.pdf>
Acesso em 15 de agosto de 2022.

ZAIED, ABDEL NASSER H.; ISMAIL, MAHMOUD M.; MOHAMED, SHIMAA S. Permutation flow shop scheduling problem with makespan criterion: literature review. **J. Theor. Appl. Inf. Technol**, v. 99, n. 4, 2021.

ZIVIANI, Nivio. **Projetos de Algoritmos Com Implementações em Pascal e C**. 4. ad. -- São Paulo : Pioneira, 1999.

APÊNDICE A - CÓDIGO EM PASCAL

```
program tcc2;
uses Windows, SysUtils;   //bibliotecas

type
arrn = array[0..501] of integer;

var
i,ii,j,jj,jjj,k,m,n,a,aa,bb,cc,dd,ee,xx,mk,ax,ft,fof,tempoinicial,tempofinal,tempo: integer;
   ma,mb,d : array [0..501,0..501] of integer;
   s,t,w,w1,sp : array [0..501] of integer;
   arquivo, arquivo1, arq : text;

{procedure fo(a:arrn;ee:integer);   //No-Idle
  var li,lj,ax1,ax2,ftp : integer;
  begin
    mk := 0;
    for li := 1 to ee do mk := mk + ma[1,a[li]];
    for li := 2 to m do
      begin
        ax1 := 0;
        for lj := 1 to ee-1 do
          begin
            ax2 := 0;
            if ma[li,a[lj]] > ma[li-1,a[lj+1]] then ax1 := ax1 + (ma[li,a[lj]] - ma[li-
1,a[lj+1]]);
            if ma[li,a[lj]] < ma[li-1,a[lj+1]] then
              begin
                ax2 := ma[li-1,a[lj+1]] - ma[li,a[lj]];
                if ax1 < ax2 then ax1 := 0;
                if (ax1 >= ax2) and (ax1 > 0) then ax1 := ax1 - ax2;
              end;
          end;
        mk := mk + ax1 + ma[li,a[ee]];
      end;
```

```
        ft := mk;
        ftp := 0;
        for li := ee downto 2 do
          begin
            ft := ft - ma[m,a[li]];
            ftp := ftp + ft;
          end;
        ft := ftp + mk;
        fof := mk;
  end;
 }
{procedure fo(a:arrn;ee:integer);      //No-Wait
  var lll : integer;
  begin
    mk:=0;
    ft:=0;
    for lll := 1 to ee-1 do mk := mk + d[a[lll],a[lll+1]];
    mk := mk+t[a[ee]];
    writeln('t[a[ee]] = ', t[a[ee]]);
    for lll := 2 to ee do ft := ft + (ee+1-lll) * d[a[lll-1],a[lll]];
    for lll := 1 to ee do ft := ft + t[a[lll]];
    fof := mk;
  end;    }

procedure fo(ax:arrn;b:integer);          //Clássico
begin   //calculo do makespan

 for ii := 1 to m do for jj := 1 to n do mb[ii,jj] := 0;
 mb[1,1] := ma[1,ax[1]];
 for ii := 2 to b do mb[1,ii] := mb[1,ii-1] + ma[1,ax[ii]];
 for ii := 2 to m do
 begin
 mb[ii,1] := mb[ii-1,1] + ma[ii,ax[1]];
 end;
```

```
  for ii := 2 to m do
    for jj := 2 to b do
      begin
        a := mb[ii,jj-1];
        if a < mb[ii-1,jj] then a := mb[ii-1,jj];
        mb[ii,jj] := a + ma[ii,ax[jj]];
      end;
  mk := mb[m,b];
  ft := 0;
  for ii := 1 to b do
  ft := ft + mb[m,ii];
  fof := ft;    //fof = função objetivo final (mudar mk ou ft)
end;

begin
assign(arquivo1,'C:\Users\natha\Desktop\TCC\NOVOS DADOS\saida.txt');
rewrite(arquivo1);
for xx := 1 to 120 do
begin
tempoinicial := GetTickCount;
assign(arquivo,'C:\Users\natha\Desktop\TCC\NOVOS
DADOS\'+IntToStr(xx)+'.txt');
reset(arquivo);
readln(arquivo,m,n);
a := 0;
for i := 1 to m do
  begin
    for j := 1 to n - 1 do read(arquivo,ma[i,j]);
    readln (arquivo,ma[i,n]);
  end;
writeln('colunas: ',n);
writeln('linhas: ',m);
```

```
{Matriz de Delay (só para No-Wait)----------------------------------------------} {
aa := 0;
bb := 0;
cc := 0;
dd := 0;
for jj := 1 to n do
  begin
    for jjj := 1 to n do
      begin
        for i := 1 to m do
          begin
            ma[0,jjj] := 0;
            aa := aa + ma[i,jj];
            bb := bb + ma[i-1,jjj];
            cc := aa - bb;
            if cc > dd then
              begin
                d[jj,jjj] := cc;
                dd := d[jj,jjj];
              end
              else
                d[jj,jjj] := dd;
                if i = m then
                  begin
                    aa:=0;
                    bb:=0;
                    dd:=0;
                  end;
          end;
        if jj = jjj then d[jj,jjj] := 0;
      end;
  end;
 }
{------------------------------------------------------------}
```

```
for i := 1 to n do s[i] := 1;
for i := 1 to n do t[i] := 0;
for i := 1 to n do for j := 1 to m do t[i] := t[i] + ma[j,i];
for i := 1 to n do for j := 1 to n do if t[i] < t[j] then s[i] := s[i] + 1;
//LPT < ou SPT >
for i := 1 to n do for j := 1 to n do if i <> j then if s[i] = s[j] then s[j] := s[j] + 1;
  for i := 1 to n do
    begin
      ee := s[i];
      w[ee] := i;
    end;
for i := 1 to n do
begin
w1[i] := w[i];
end;

{2a. fase--NEH----------------------------------------------------------------}
fo(w,2);
ax := fof;
w1[1] := w[2];
w1[2] := w[1];
fo(w1,2);
if fof >= ax then for i := 1 to 2 do
begin
w1[i] := w[i];
end;
for i := 3 to n do
begin
w1[i] := w[i];
//writeln('w1=', w1[i]);
end;
for i := 3 to n do
   begin
      ax := 0;
```

```
            fo(w1,i);
            ax := fof;
            for k := 1 to i do
            begin
            sp[k] := w1[k];
            end;
            for k := i downto 2 do
              begin
                aa := sp[k];
                sp[k] := sp[k-1];
                sp[k-1] := aa;
                fo(sp,i);
                if fof < ax then
                   begin
                     ax := fof;
                     for aa := 1 to i do
                     begin
                     w1[aa] := sp[aa];
                     end;
                   end;
              end;
  end;
for i := 1 to n do write(w1[i],' ');
fo(w1,n);
writeln;
writeln('funcao-objetivo da sequencia w1: ',fof);
readln;
close(arquivo);
tempofinal := GetTickCount;
tempo := tempofinal - tempoinicial;
writeln(arquivo1,fof,'-',tempo);
end;
close(arquivo1);
end.
```

APÊNDICE B - CÓDIGO EM C

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include <math.h>

//int objectiveFunction(int task, int machine, const int rankedTasks[task], int
(dataFromFile)[][task]);

int classicObjectiveFunction(int task, int machine, const int rankedTasks[task], const
int (dataFromFile)[machine][task], int auxTask);

//int noIdleObjectiveFunction(int task, int machine, const int rankedTasks[task], const
int (dataFromFile)[machine][task],int auxTask);

//int noWaitObjectiveFunction(int task, int machine, const int rankedTasks[task], const
int totalTimeByTasks[task], int (dataFromFile)[machine][task], int auxTask);

#define FILES 120

size_t i, j;

int main() {

   if (remove("saida.txt") == 0) {
      printf("The file is deleted successfully.");
   } else {
      printf("The file is not deleted.");
   }

   int machine, task, classifiedPosition;

   unsigned long initialExecutionTime, finalExecutionTime;

   FILE *entranceFile;
   FILE *outputFile;

   size_t file;

   char filename[7];

   for (file = 1; file <= FILES; file++) {

      initialExecutionTime = GetTickCount(); //começa a contar o executionTime de
execução no computador
      sprintf(filename, "%zu.txt", file); //cria o nome dos arquivos em txt para serem
lidos
      entranceFile = fopen(filename, "r"); //lê os dados dentro dos arquivos txt
      fscanf(entranceFile, "%d %d", &machine, &task); //lê a primeira linha com as
linhas e colunas machine=máquina task=tarefa
```

```
    if (entranceFile == NULL) {
        printf("\n Erro! Arquivo %s inexistente \n ", filename);
        exit(-1);
    }

    int totalTimeByTasks[task], classifiedTask[task], preRankedTasks[task],
tempRankedTasks[task], rankedTasks[task];

    int dataFromFile[machine][task];

    for (i = 0; i < machine; i++) {
        for (j = 0; j < task; j++) {
            fscanf(entranceFile, "%d", &dataFromFile[i][j]);
        }
    }

    printf("\nTarefas: %d\n", task);
    printf("Maquinas: %d\n", machine);


    for (i = 0; i < task; i++) { //inicialização dos vetores
        totalTimeByTasks[i] = 0;
        classifiedTask[i] = 1;
        preRankedTasks[i] = 0;
        rankedTasks[i] = 1;
    }

    for (i = 0; i < task; i++) { //somatória de tempos das tarefas
        for (j = 0; j < machine; j++) {
            totalTimeByTasks[i] = totalTimeByTasks[i] + dataFromFile[j][i];
            // printf("totalTimeByTasks = %d\t", totalTimeByTasks[i]); //OK
        }
    }

    for (i = 0; i < task; i++) {
        for (j = 0; j < task; j++) {
            if (totalTimeByTasks[i] > totalTimeByTasks[j]) {    //LPT < ou SPT >
                classifiedTask[i] = classifiedTask[i] + 1; // ordenação da somatória dos
                //printf("classifiedTask = %d\t", classifiedTask[i]); //OK
            }
        }
    }

    for (i = 0; i < task; i++) {
        for (j = 0; j < task; j++) {
            if (i != j) {
                if (classifiedTask[i] == classifiedTask[j]) {
                    classifiedTask[j]++;//critério de desempate, adiciona 1 a posição
seguinte(j)
                    //printf("classifiedTask = %d\t", classifiedTask[j]); //OK
```

```
            }
        }
    }
}

for (i = 0; i < task; i++) {
    classifiedPosition = classifiedTask[i] - 1;
    preRankedTasks[classifiedPosition] = i;
    //printf("preRankedTasks = %d\t", preRankedTasks[classifiedPosition]); //OK
}

for (i = 0; i < task; i++) {
    rankedTasks[i] = preRankedTasks[i] + 1;
    tempRankedTasks[i] = rankedTasks[i];
    //printf("tempRankedTasks = %d\t", tempRankedTasks[i]); //OK
    //printf("rankedTasks = %d \t", rankedTasks[i]); //OK
}

////----------------------------------------NEH-------------------------------------------------//
    int auxRankedTasks[task], calculatedValue, sp[task], aa;

    for (i = 0; i < task; i++) {
        auxRankedTasks[i] = tempRankedTasks[i];
        //printf("\n auxRankedTasks[%d] = %d",i,auxRankedTasks[i]); //OK
        sp[i] = 0;
        //printf("\n sp[%d] = %d",i,sp[i]); //OK
    }

    //int classic = classicObjectiveFunction(task, machine, rankedTasks,
dataFromFile, 2);
    int noIdle = noIdleObjectiveFunction(task, machine, tempRankedTasks,
dataFromFile, 2);
    //int noWait = noWaitObjectiveFunction(task, machine, tempRankedTasks,
totalTimeByTasks, dataFromFile, 2);

    calculatedValue = noIdle;
    //printf("\n calculatedValue = %d",calculatedValue); //OK

    tempRankedTasks[0] = auxRankedTasks[1];   //os dois primeiros pares de
tarefas são trocados
    tempRankedTasks[1] = auxRankedTasks[0];
    //printf("\n tempRankedTasks[0] = %d",tempRankedTasks[0]); //OK
    //printf("\n tempRankedTasks[1] = %d",tempRankedTasks[1]); //OK

    //chama a procedure do calculo do makespan para calculo do primeiro par de
tarefas
    //classic = classicObjectiveFunction(task, machine, tempRankedTasks,
dataFromFile, 2);
    noIdle = noIdleObjectiveFunction(task, machine, tempRankedTasks,
dataFromFile, 2);
```

```
        //noWait = noWaitObjectiveFunction(task, machine, tempRankedTasks,
totalTimeByTasks, dataFromFile, 2);

        if (noIdle >= calculatedValue) {

            for (i = 0; i < 2; i++) {
                tempRankedTasks[i] = auxRankedTasks[i];
                //printf("\n auxRankedTasks[%d] = %d",i,auxRankedTasks);
                //printf("\n tempRankedTasks[%d] = %d",i,tempRankedTasks[i]);  //OK

            }
        } //se o makespan do par de tarefas trocado for maior do que o makespan da
sequencia inicial, mantem-se a sequencia original;

        for (i = 2; i < task; i++) {
            tempRankedTasks[i] = auxRankedTasks[i];        //o vetor de sequencia final
recebe os mesmo valores do vetor de sequencia inicial;
            //printf("\ntempRankedTasks[%d] = %d", i, tempRankedTasks[i]); //OK
        }

        for (j = 0; j < task; j++) {
            sp[j] = tempRankedTasks[j];      //o vetor auxiliar é alimentado pela vetor de
sequencia final
            //printf("\nsp[%d] = %d", j, sp[j]); //OK
        }

        for (i = 3; i <= task; i++) {

            calculatedValue = 0; //calcula-se o makespan para cada inclusao de tarefa da
sequencia inicial
            //classic = classicObjectiveFunction(task, machine, tempRankedTasks,
dataFromFile, i);
            noIdle = noIdleObjectiveFunction(task, machine, tempRankedTasks,
dataFromFile, i);
            //noWait = noWaitObjectiveFunction(task, machine, tempRankedTasks,
totalTimeByTasks, dataFromFile, i);
            calculatedValue = noIdle;      //a variavel auxiliar recebe o resultado do
calculo

            for (j = 0; j < i; j++) {
                sp[j] = tempRankedTasks[j];      //o vetor auxiliar é alimentado pela vetor de
sequencia final
                //printf("\nsp[%d] = %d", j, sp[j]); //OK
            }

            for (j = i - 1; j >= 1; j--)      //a troca da ordem de tarefas é realizada
            {   // for (i = task - 1; i >= 1; i--) {
                aa = sp[j];
                //printf("\naa[%d] = %d",j, aa); //OK
                sp[j] = sp[j - 1];  //sp[k] := sp[k-1];
```

```
            //printf("\nsp[%d] = %d",j, sp[j]); //OK
            sp[j - 1] = aa;
            //printf("\nsp[%d] = %d", j, sp[j]); //OK

            //classic = classicObjectiveFunction(task, machine, sp, dataFromFile, i);
            noIdle = noIdleObjectiveFunction(task, machine, sp, dataFromFile,i);
            //noWait = noWaitObjectiveFunction(task, machine, sp, totalTimeByTasks,
dataFromFile,i);

            if (noIdle <
               calculatedValue)        //caso o calculo atual seja menor do que o ultimo
calculo armazenado
            {
               calculatedValue = noIdle;           //armazena o calculo de makespan
atual
               //printf("\ncalculatedValue = %d", calculatedValue);
               for (aa = 0; aa <= i-1; aa++) {  //aa := 1 to i

                  tempRankedTasks[aa] = sp[aa];
                  //printf("\nsp[%d] = %d", aa, sp[aa]);
                  //printf("\ntempRankedTasks[%d] = %d", aa, tempRankedTasks[aa]);

               }    //armazena ate o i atual, a sequencia com menor makespan
            }

         }
      }

      int auxTask = task;

      fclose(entranceFile);
      outputFile = fopen("saida.txt", "a+");
      finalExecutionTime = GetTickCount();
      fprintf(outputFile, "%d - %d\n", noIdleObjectiveFunction(task, machine,
rankedTasks, dataFromFile, auxTask),(finalExecutionTime - initialExecutionTime));
   }
   fclose(outputFile);
   return 0;
}

// Sistema Clássico
/*int
classicObjectiveFunction(int task, int machine, const int rankedTasks[task], const int
(dataFromFile)[machine][task],
                   int auxTask) {
   size_t ii, jj;
   int makespanMatrix[machine][task];
   int a, makespan, flowtime = 0;

   for (ii = 0; ii < task; ii++) {
```

```
        //printf("rankedTasks = %d \t", rankedTasks[ii]); //OK
    }

    for (ii = 0; ii < machine; ii++) {
        for (jj = 0; jj < auxTask; jj++) {
            makespanMatrix[ii][jj] = 0;                    //toda a matriz que é alimentada por
valores de makespan, inicialmente é zerada.

        }
    }

    makespanMatrix[0][0] = dataFromFile[0][rankedTasks[0] - 1];

    //printf("makespanMatrix = %d \t", makespanMatrix[0][0]); //OK

    for (ii = 1; ii < auxTask; ii++) {
        makespanMatrix[0][ii] = (makespanMatrix[0][ii - 1] +
dataFromFile[0][rankedTasks[ii] - 1]);
        //printf("makespanMatrix = %d \t", makespanMatrix[0][i]);
    }

    for (ii = 1; ii < machine; ii++) {
        makespanMatrix[ii][0] = (makespanMatrix[ii - 1][0] +
dataFromFile[ii][rankedTasks[0] -1]); // dataFromFile[i]
        //printf("makespanMatrix = %d \t\n", makespanMatrix[i][0]);
        //printf("dataFromFile[%d][rankedTasks[0]-1] = %d \t",i,
dataFromFile[i][rankedTasks[0]-1]);
    }

    for (ii = 1; ii < machine; ii++) {
        for (jj = 1; jj < auxTask; jj++) {
            a = makespanMatrix[ii][jj - 1];
            //printf("a = %d \t", a);
            if (a < makespanMatrix[ii - 1][jj]) {
                a = makespanMatrix[ii - 1][jj];
                //printf("a2 = %d \t", a);
            }
            makespanMatrix[ii][jj] = a + dataFromFile[ii][rankedTasks[jj] - 1];
            //printf("makespanMatrix = %d \t", makespanMatrix[i][j]);
        }
    }

    for (ii = 0; ii < auxTask; ii++) {
        flowtime = flowtime + makespanMatrix[machine - 1][ii];
        makespan = makespanMatrix[machine -1][ii]; //makespan - a variavel de
makespan equivale ao ultimo elemento da tarefa que se deseja na ultima maquina
disponivel.
    }
    //printf("\nmakespan %d", makespan);
    //printf("\nflowtime %d", flowtime);
```

```
    return flowtime;
}
*/

//Sistema No-Idle

int noIdleObjectiveFunction(int task, int machine, const int rankedTasks[task], const
int (dataFromFile)[machine][task], int auxTask) {

    int ax1, ax2, makespan = 0, flowtime = 0, ftp = 0;
    int ii, jj;

    for (ii = 0; ii < task; ii++) {
        makespan = makespan + dataFromFile[0][rankedTasks[ii]];
        //printf("\nmakespan = %d", makespan);
    }

    for (ii = 1; ii <= machine; ii++) {
        ax1 = 0;
        for (jj = 0; jj <= task - 2; jj++) {
            ax2 = 0;
            if ((dataFromFile[ii][rankedTasks[jj] - 1]) > (dataFromFile[ii - 1][rankedTasks[jj
+ 1] - 1])) {
                ax1 = ax1 + (dataFromFile[ii][rankedTasks[jj] - 1] - dataFromFile[ii -
1][rankedTasks[jj + 1] - 1]);
                //printf("\nax1 = %d", ax1);
            }

            if ((dataFromFile[ii][rankedTasks[jj] - 1]) < (dataFromFile[ii - 1][rankedTasks[jj
+ 1] - 1])) {
                ax2 = dataFromFile[ii - 1][rankedTasks[jj + 1] - 1] -
dataFromFile[ii][rankedTasks[jj] - 1];
                if (ax1 < ax2) {
                    ax1 = 0;
                }
                if ((ax1 >= ax2) && (ax1 > 0)) {
                    ax1 = ax1 - ax2;
                    //printf("\nax1 = %d", ax1);
                }
            }
        }
        makespan = makespan + ax1 + dataFromFile[ii][rankedTasks[task-1]];
        //printf("\nmakespan = %d", makespan);
    }
    //printf("\nmakespan = %d", makespan);
    flowtime = makespan;

    for (ii = task - 1; ii >= 1; ii--) {
        flowtime = flowtime - dataFromFile[machine - 1][rankedTasks[ii] - 1];
```

```
        ftp = ftp + flowtime;
    }

    flowtime = ftp + makespan;
    //printf("\nflowtime = %d", flowtime);

    return flowtime;
}

//Sistema No-Wait
int noWaitObjectiveFunction(int task, int machine, const int rankedTasks[task], const
int totalTimeByTasks[task],
                        int (dataFromFile)[machine][task], int auxTask) {

    int delay[task][task], tempTotalTime[task];
    int makespan = 0, flowtime = 0;
    int aa = 0;
    int bb = 0;
    int cc = 0;
    int dd = 0;
    int k, ii, jj;

    for (ii = 0; ii < task; ii++) {
        for (jj = 0; jj < task; jj++) {
            delay[ii][jj] = 0;
        }
    }
    for (ii = 0; ii < task; ++ii) {
        tempTotalTime[ii] = 0;
    }

    for (ii = 0; ii < task; ++ii) {
        tempTotalTime[ii] = totalTimeByTasks[ii];
    }
//{Matriz de Delay (só para No-Wait)-----------------------------------------------}

    for (ii = 0; ii < task; ii++) {
        for (jj = 0; jj < task; jj++) {
            for (k = 0; k < machine; k++) {
                dataFromFile[-1][jj] = 0;
                aa = aa + dataFromFile[k][ii];
                bb = bb + dataFromFile[k - 1][jj];
                cc = aa - bb;

                if (cc > dd) {

                    delay[ii][jj] = cc;
                    dd = delay[ii][jj];

                } else {
```

```
                delay[ii][jj] = dd;
            }
            if (k == machine - 1) {
                aa = 0;
                bb = 0;
                dd = 0;
            }
        }
        if (ii == jj) {
            delay[ii][jj] = 0;
        }
    }
}

for (ii = 0; ii <= task - 2; ii++) {
    makespan = makespan + delay[rankedTasks[ii] - 1][rankedTasks[ii + 1] - 1];
}

makespan = makespan + tempTotalTime[rankedTasks[task-1]];
//printf("\nmakespan = %d", makespan);

for (ii = 2; ii <= task; ii++) {
    flowtime = flowtime + ((task - ii + 1) * delay[rankedTasks[ii - 2] -
1][rankedTasks[ii - 1] - 1]);
}
//printf("\nflowtime1 = %d", flowtime);


for (ii = 0; ii < task; ii++) {
    flowtime = flowtime + tempTotalTime[rankedTasks[ii] - 1];
}
//printf("\nflowtime = %d", flowtime);

return flowtime;

}
```