



# Maratona de Filmes - Relatório Final

**Aluno:** Jamesson Leandro Paiva Santos

<https://github.com/jamessonlps/projeto-supercomp>

## O Problema

Inspirado nos problemas do caixeiro viajante e da mochila binária, o presente relatório explora conceitos e fundamentos de soluções de alto desempenho para buscar soluções para o seguinte problema:

*Você quer passar um final de semana assistindo ao máximo de filmes possível, mas há restrições quanto aos horários disponíveis e ao número de títulos que podem ser vistos em cada categoria (comédia, drama, ação, etc). Qual é o número máximo de filmes que podem ser assistidos de acordo com as restrições de horários e número máximo por categoria? Conseguimos preencher as 24h?*

A entrada de dados do algoritmo consiste em arquivos de texto que seguem a seguinte formatação:

```
10 4
1 3 1 2
11 13 3
14 15 3
10 16 2
10 14 1
...
```

- A primeira linha da entrada contém o número  $N$  de filmes e o número  $k$  de categorias.
- A segunda linha contém o número máximo de filmes por categoria que podem ser assistidos.
- Da terceira linha em diante, segue os filmes com a hora de início, hora de término e a categoria a qual pertence.

Esse problema será explorado aqui com a heurística **gulosa** e a **aleatoriedade**.

## Considerações

Há um gerador de inputs que foi disponibilizado pelos professores da disciplina para criar os arquivos de texto de entrada. A única adaptação feita na geração foi de desconsiderar os casos em que o filme tem hora de início e fim iguais. Isso foi feito porque, para um filme que começa às 17 e termina às 18, por exemplo, considera-se que ele "preenche" o slot das 17, então outro filme que comece às 18 pode ser selecionado. Assim, considerar os filmes com horário de início e fim iguais causaria uma ambiguidade e problemas na implementação do algoritmo.

Além disso, os filmes que começam em um dia e terminam em outro foram considerados para o algoritmo. A restrição imposta é que o tempo total de tela (soma das durações dos filmes selecionados) não ultrapasse 24 e que um não se sobreponha ao outro (por exemplo, é permitido que um filme termine às 10 e outro inicie nessa mesma hora, mas não é permitido que um que comece às 10 e termine às 12 seja inserido se já existe um que começa às 7 e termina às 11, pois o horário de 10 já está preenchido).

Além disso, foi feita a redução do número total de filmes trabalhados na primeira implementação. A redução do número total de filmes na implementação exaustiva com paralelismo usando OpenMP e GPUs foi necessária devido à complexidade computacional do problema da mochila binária e às limitações de recursos computacionais, especialmente quando o número máximo de filmes é restrito a 30.

O problema da mochila binária requer a avaliação de todas as possíveis combinações de itens para encontrar a solução ótima. A abordagem exaustiva implica em avaliar todas as combinações possíveis, o que resulta em um espaço de soluções exponencialmente grande. Com o aumento do número de filmes, o número de soluções possíveis cresce rapidamente, tornando inviável a análise de todas elas em um tempo razoável.

Embora a redução do número total de filmes para 30 possa limitar a representatividade dos resultados em relação a conjuntos de dados maiores, ela é uma estratégia necessária para garantir um tempo de execução razoável e o uso eficiente dos recursos disponíveis. Essa limitação no número máximo de filmes é uma forma de equilibrar a complexidade computacional do problema com a capacidade de processamento e memória dos sistemas utilizados.



Devido a essa mudança do número de filmes da etapa intermediária para a etapa final, a seguir será mantido o relato das abordagens gulosa e aleatória para um grande número de filmes. Você pode ver as novas implementações clicando [aqui](#).

## Heurísticas Gulosa vs Aleatória

### Heurística Gulosa

A implementação da heurística gulosa foi feita tomando a lista de filmes e ordenando-a em ordem crescente de tempo de fim. De posse da lista ordenada, o algoritmo percorre os filmes e popula a lista dos selecionados, levando em conta se o horário está disponível e se ainda pode ser inserido o filme para a respectiva categoria.

O trecho de código a seguir mostra a lógica principal do algoritmo:

```
for (auto &movie : movies)
{
    // Tempo máximo de 24h
    if (total_time >= 24)
    {
        return;
    }
```

```

if (movie.begin >= last_selected.end &&
    max_cat[movie.category] > 0 &&
    has_time_available(times_available, movie))
{
    solution.push_back(movie);
    update_availability_list(times_available, movie);

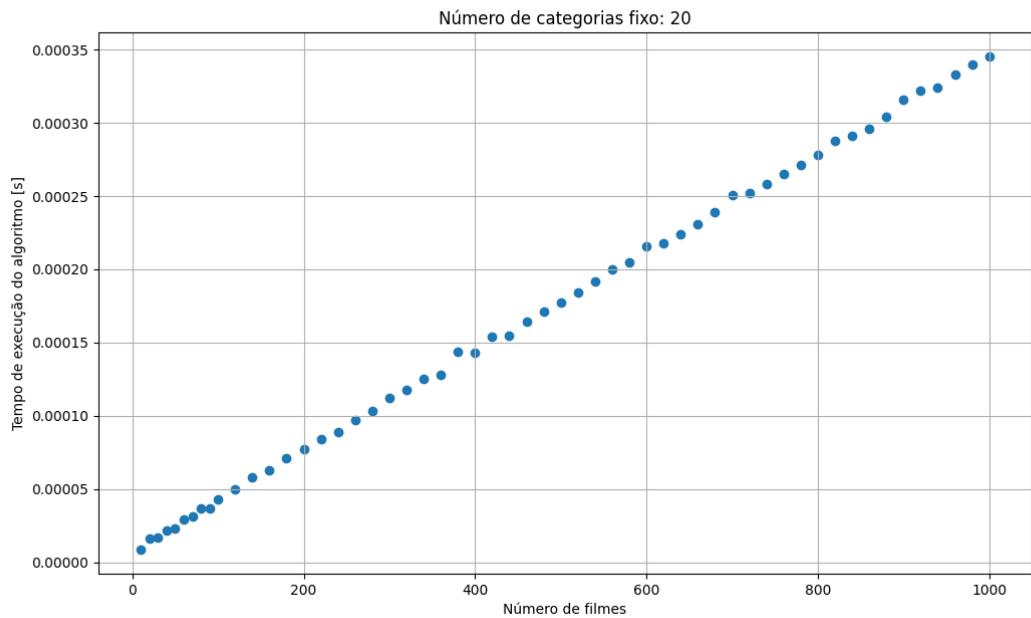
    max_cat[movie.category]--;
    total_time += movie.duration;
    last_selected = movie;
}
}

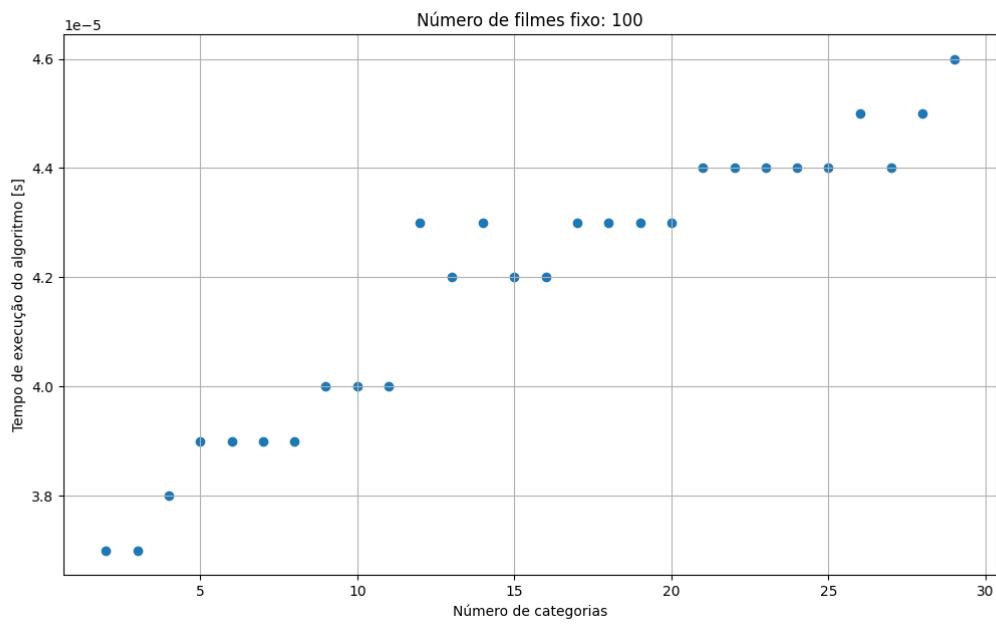
```

Após *inputar* os dados e ordenar o vetor de filmes, esse vetor é percorrido e, se o filme começa a partir do início do anterior, ainda há limite disponível para sua categoria e sua duração não *\*cruza\** com nenhum outro filme já selecionado, então esse filme entra para a lista dos selecionados.

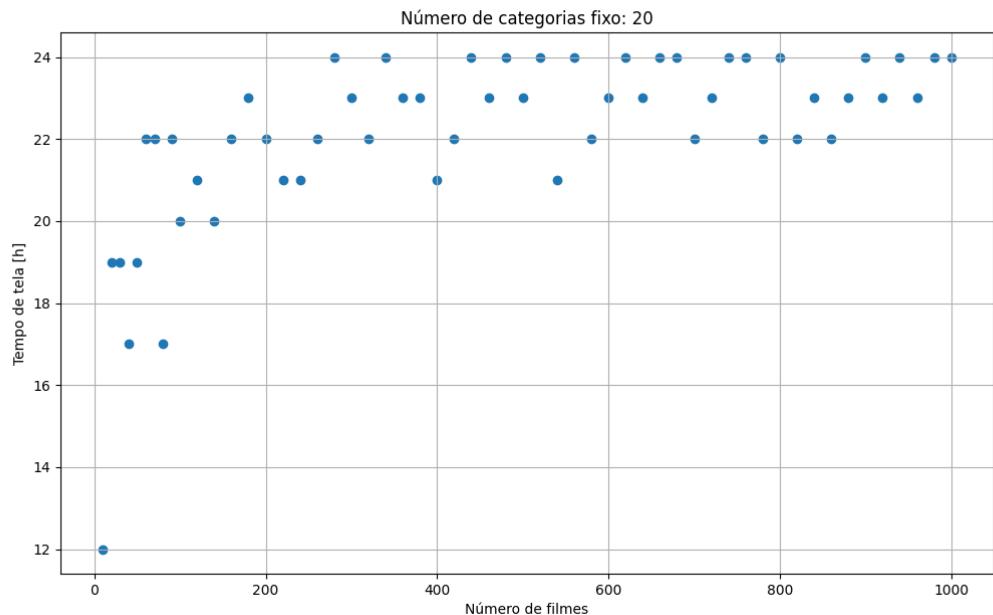
## Resultados

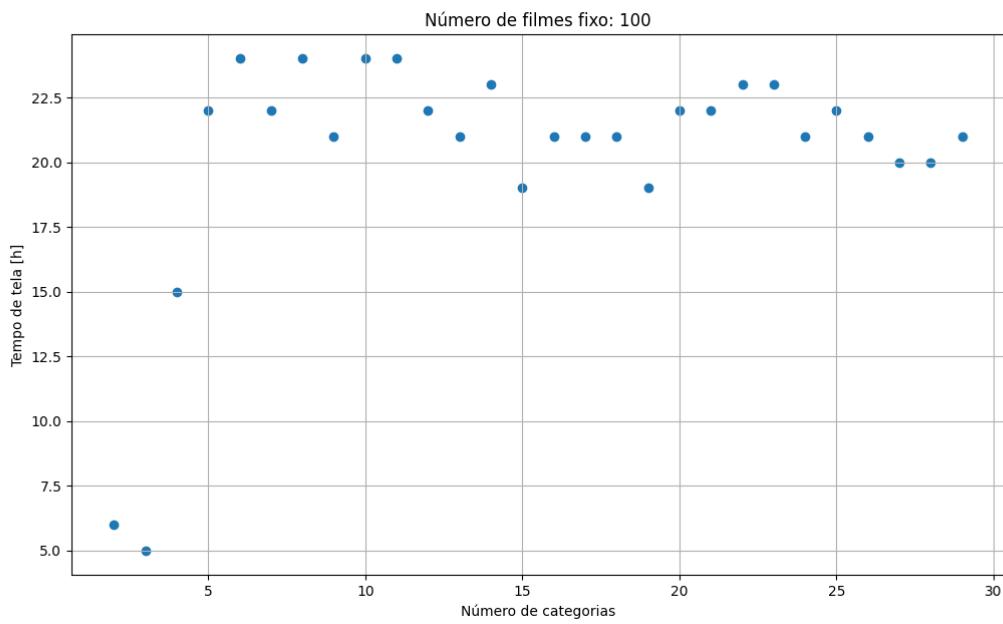
Os resultados a seguir foram obtidos executando o código para as entradas descritas anteriormente.





O tempo de execução do algoritmo quando o número de categorias é fixo é praticamente linear com o número de filmes. Já quando se fixa o número de filmes, o tempo total de execução parece crescer em pequenos degraus, a cada 4 categorias aproximadamente.





Quanto ao tempo de tela, parece que os resultados foram melhores quando se fixou o número de categorias. Com mais filmes, obteve-se mais vezes o tempo de tela em 24h.

## Análise do Valgrind

Ao executar o `valgrind` com o executável `gulosa`, o trecho do resultado (arquivo `callgrind.out.17790`) obtido chama a atenção:

```
1,014,826 (40.03%) ./elf./.elf/dl-lookup.c:_dl_lookup_symbol_x [/usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2]
592,804 (23.38%) ./elf./.elf/dl-lookup.c:do_lookup_x [/usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2]
264,358 (10.43%) ./elf./.sysdeps/x86_64/dl-machine.h:_dl_relocate_object
```

As funções apontadas no resultado do `valgrind` são funções do sistema operacional. Por exemplo, a função `_dl_lookup_symbol_x` está ligada ao carregamento de bibliotecas compartilhadas, enquanto que a `do_lookup_x` cuida da resolução de símbolos em chamadas de função ou variável de bibliotecas compartilhadas.

## Aleatoriedade

A implementação da aleatoriedade é feita a partir da heurística gulosa, mas agora considerando que, em cada iteração na lista de filmes, há 25% de pegar outro filme qualquer que satisfaça a condição de horário e disponibilidade por categoria.

O trecho abaixo contém a lógica principal do algoritmo de aleatoriedade:

```
for (int i = 0; i < num_movies; i++)
{
    // Tempo total atingido
    if (total_time >= 24)
        return;

    double random_prob = distribution_real(engine);

    // Se P > 25%, escolhemos um filme ao acaso
```

```

if (random_prob > prob)
{
    uniform_int_distribution<int> distribution_int(i, num_movies - 1);
    int random_index = distribution_int(engine); // sorteia um índice

    // Se ainda há vaga por categoria e horário, adiciona
    if (max_cat[movies[random_index].category] > 0 &&
        has_time_available(times_available, movies[random_index]))
    {
        solution.push_back(movies[random_index]);
        update_availability_list(times_available, movies[random_index]);

        max_cat[movies[random_index].category]--;
        total_time += movies[random_index].duration;
        last_selected = movies[random_index];
        i--;
    }
}

// Se P < 25%, prossegue com a heurística
else
{
    // Se há vaga na categoria e tempo, adiciona
    if (has_time_available(times_available, movies[i]) &&
        max_cat[movies[i].category] > 0)
    {
        solution.push_back(movies[i]);
        update_availability_list(times_available, movies[i]);

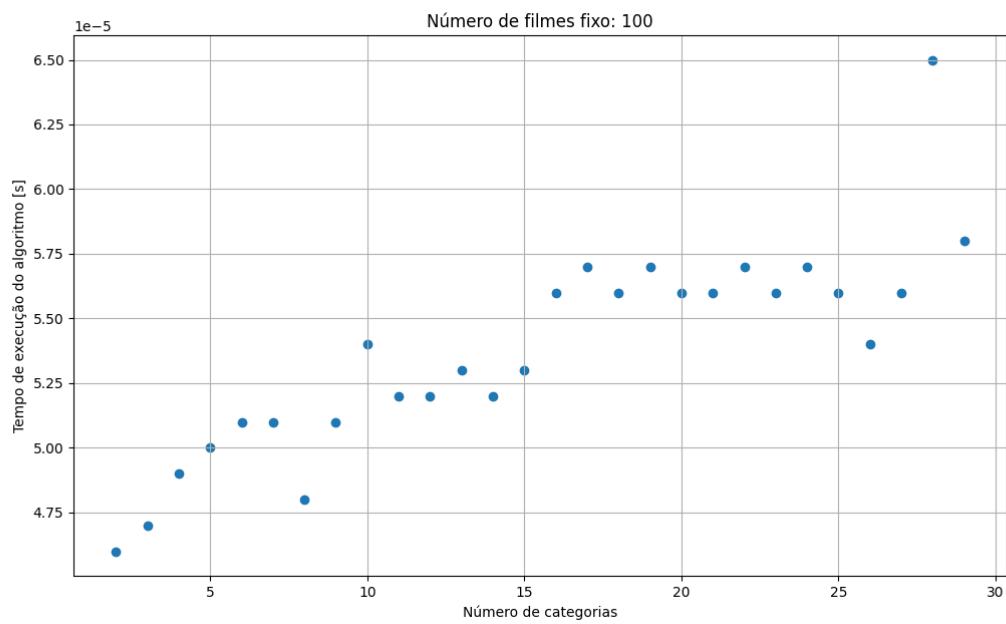
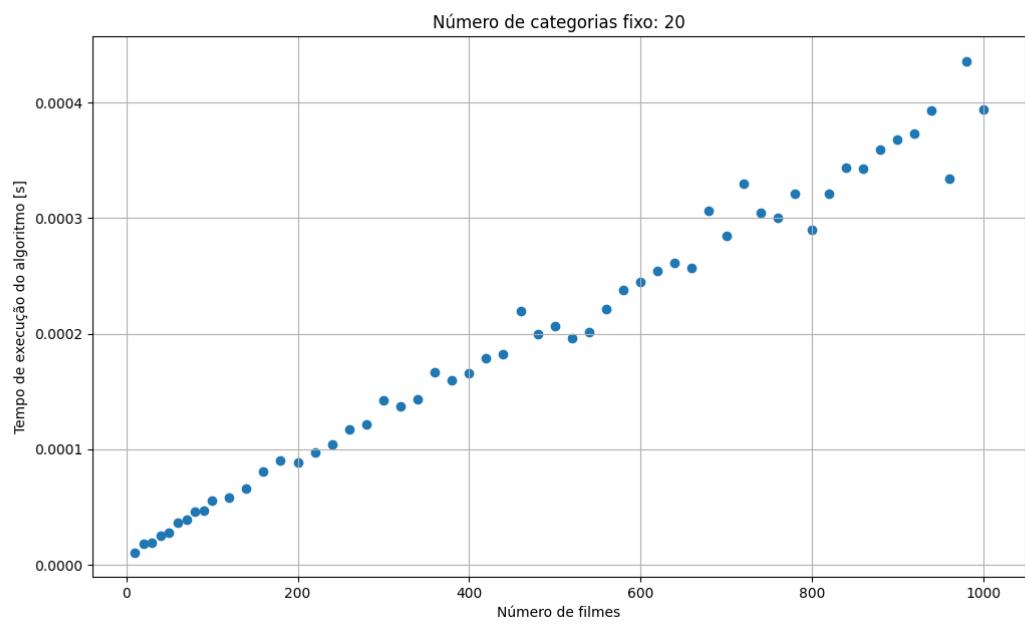
        max_cat[movies[i].category]--;
        total_time += movies[i].duration;
        last_selected = movies[i];
    }
}
}

```

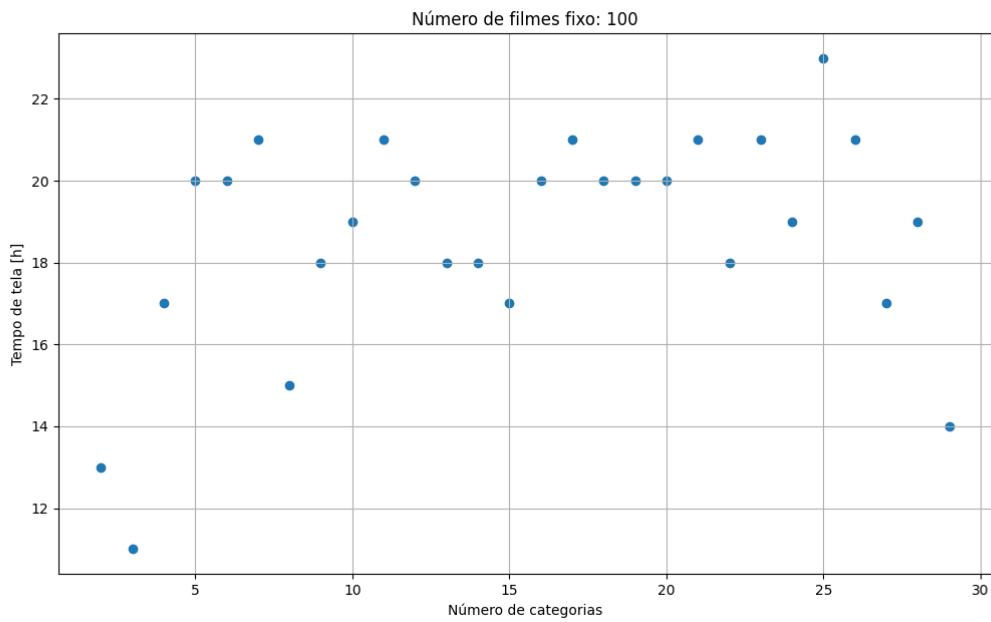
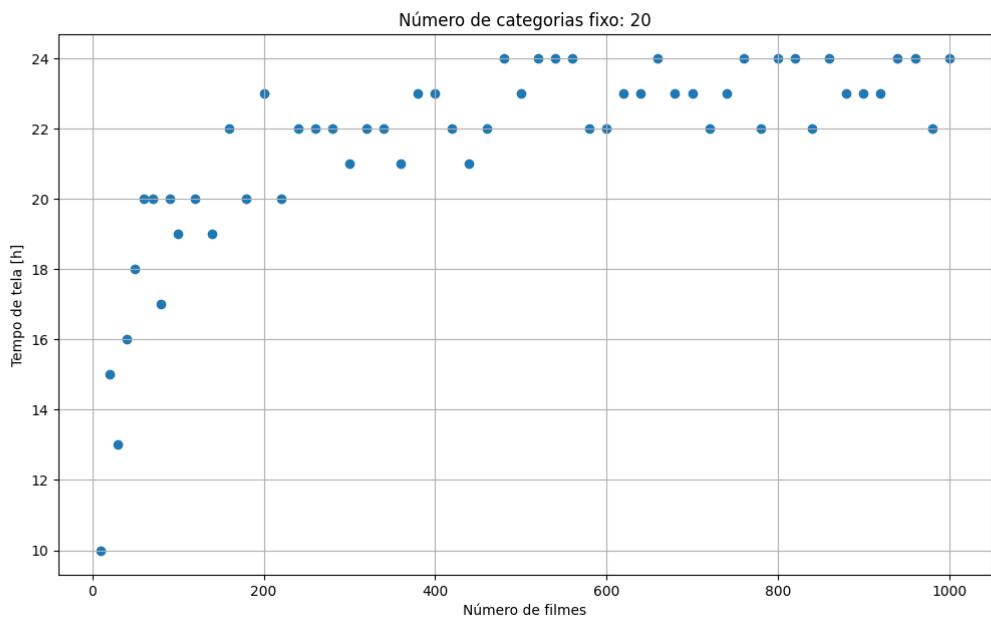
No código acima, se o valor gerado for superior a 0.25, escolhe-se um filme ao acaso e o insere na lista de selecionados, desde que ele cumpra os requisitos já definidos. Caso contrário, segue-se a heurística gulosa.

## Resultados

Os resultados a seguir foram obtidos executando o código para as entradas descritas anteriormente.



Os resultados obtidos para aleatoriedade são semelhantes aos da heurística gulosa, principalmente para o caso em que se fixou o número de categorias. Quando se ficou o número de filmes, não houve grandes mudanças no tempo de execução entre 15 e 30 categorias.



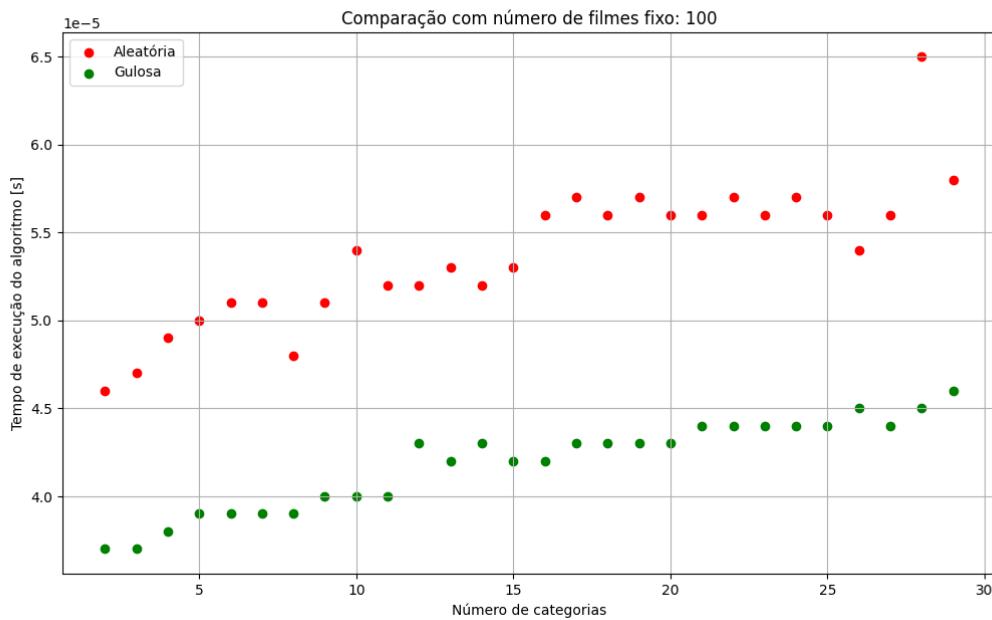
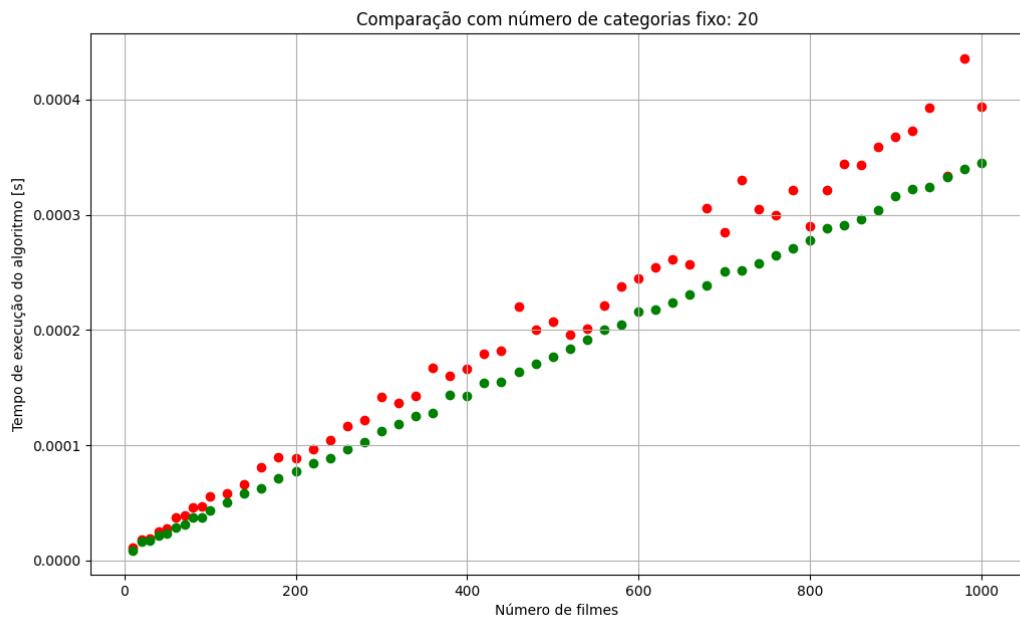
Aqui, o tempo de tela também foi melhor fixando o número de categorias, enquanto que, no caso do número de filmes constante, o tempo total praticamente não chega a 24.

### Análise do Valgrind

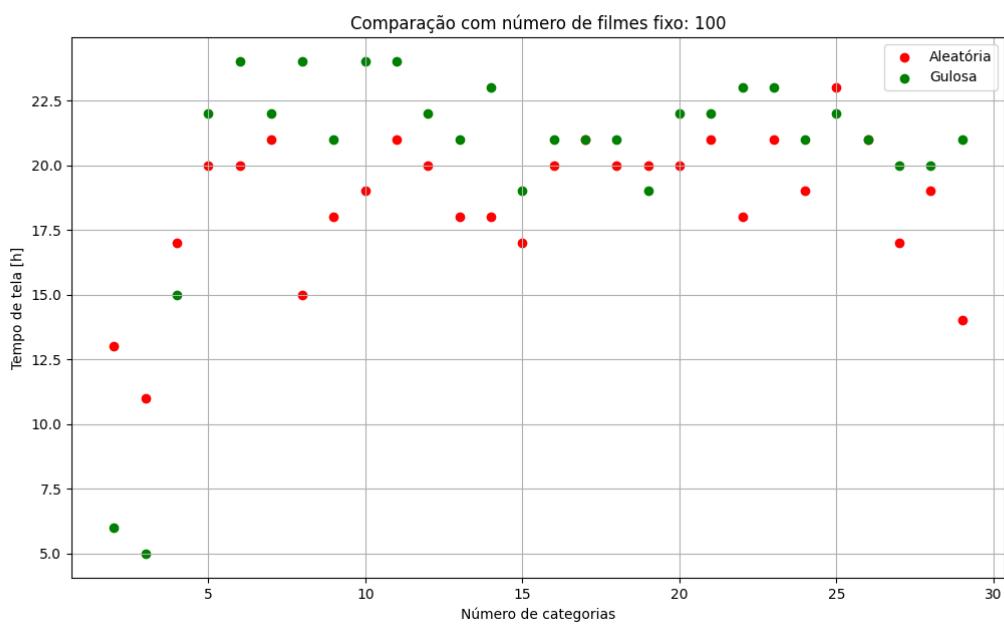
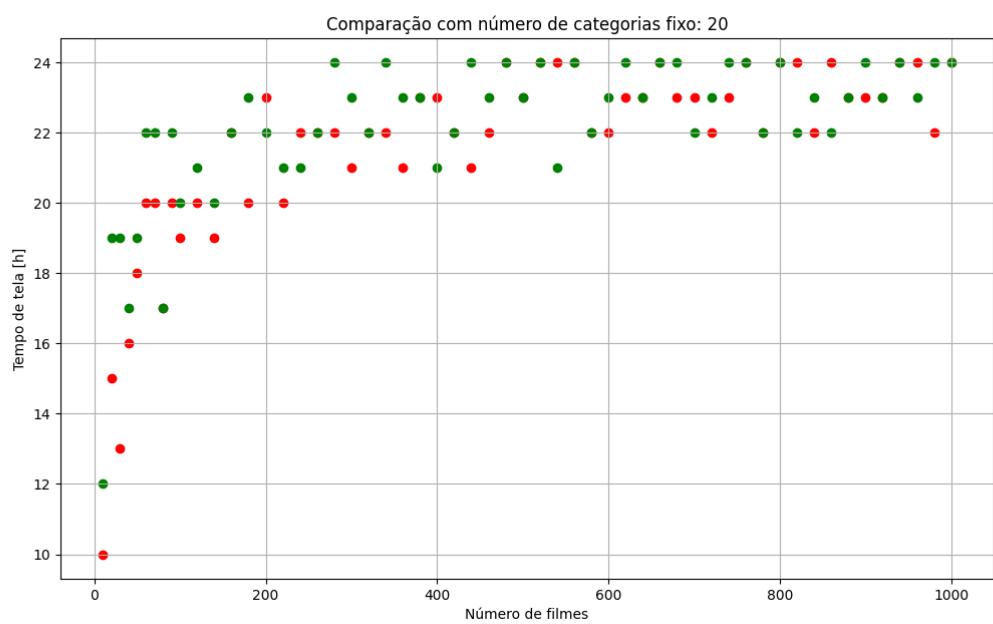
Curiosamente, obteve-se o mesmo resultado obtido para a heurística gulosa aqui, dentre os casos com maior número de execuções. Isso indica que os algoritmos estão consumindo bastante recursos e tempo de execução na parte de carregamento de bibliotecas compartilhadas.

## Comparação

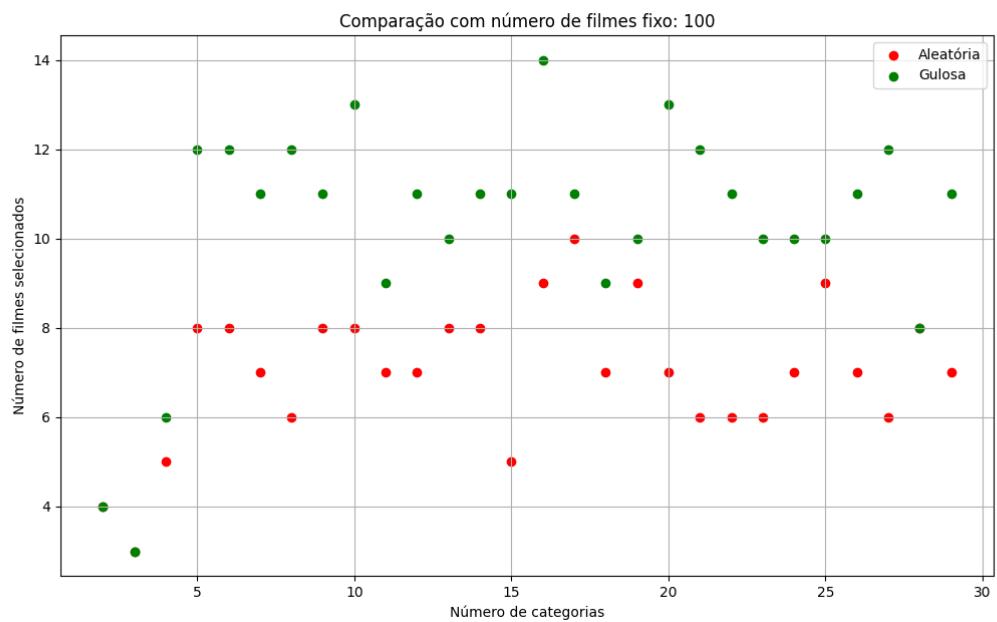
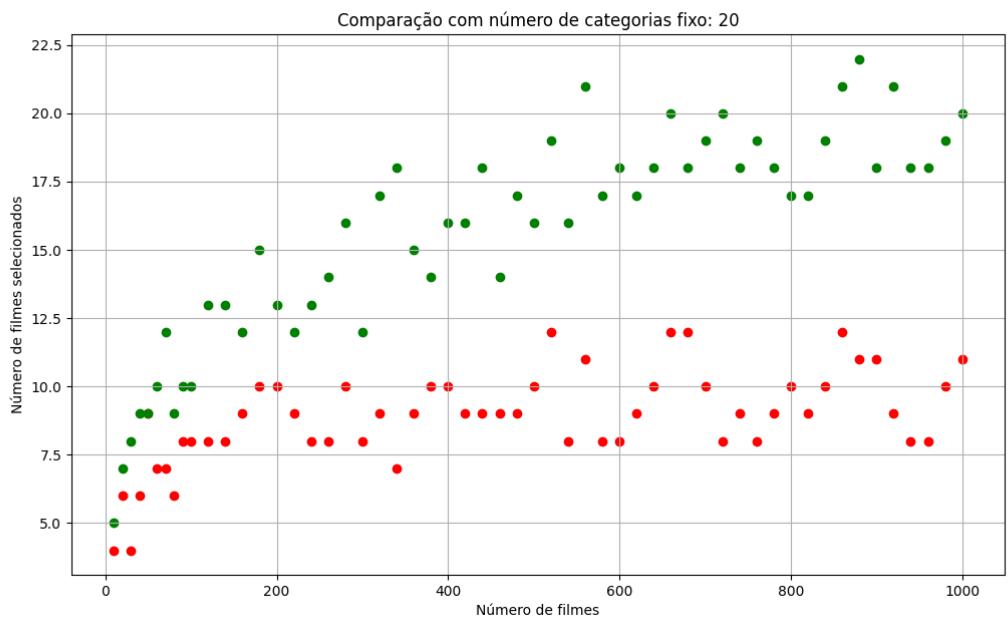
Para visualizar melhor os gráficos anteriores com fins comparativos, os gráficos abaixo são os mesmos anteriores, apenas unindo os dois casos tratados:



Na primeira imagem, o tempo de execução é muito semelhante para ambos, notando algumas pequenas “perturbações” no caso aleatório, o que é, de certa forma, esperado. Já na segunda figura nota-se uma diferença mais clara entre as duas: o tempo de execução para a heurística gulosa é menor.



Em termos de tempo de tela, ambas são muito próximas, mas nota-se que, no caso em que o número de filmes é fixo, a heurística gulosa alcança mais vezes o tempo de tela máximo (24 horas).



Por fim, comparando o número de filmes selecionados em ambos os casos, vemos que a heurística gulosa contemplou melhores resultados, especialmente quando se varia apenas o número de filmes.

## Etapa final - Comparação com Paralelismo

Na nova implementação, o código python ficou assim:

```
def run(self):
    # Descomente a linha abaixo para gerar novos inputs
    # self.generate_input_files()
```

```

# As linhas abaixo geram os outputs para cada algoritmo
# self._generate_outputs(heuristic="gulosa")
# self._generate_outputs(heuristic="aleatoria")
# self._generate_openmp_outputs()
# self._generate_outputs(heuristic="gpu")

# As linhas abaixo geram os gráficos
self.plot_results(heuristic="gulosa")
self.plot_results(heuristic="aleatoria")
self.plot_results(heuristic="gpu")
self.plot_results_openmp()
self.plot_results_comparison()

```

As linhas descomentadas geram os gráficos. A primeira parte contém a linha que irá gerar os inputs e as linhas seguintes comentadas rodam os executáveis com os inputs e geram as saídas no diretório `output`, cada saída na pasta de sua respectiva implementação. Para mais detalhes, acesse o [repositório](#).

## Paralelismo com OpenMP

O paralelismo é uma técnica essencial na área de computação que visa a execução simultânea de tarefas para melhorar o desempenho e a eficiência dos sistemas. Uma abordagem comum para a implementação do paralelismo em sistemas com múltiplos núcleos de CPU é o uso da biblioteca OpenMP. Essa biblioteca oferece diretivas e funções para programação paralela em arquiteturas de memória compartilhada, permitindo a criação de threads que podem executar tarefas em paralelo.

Com o OpenMP, podemos facilmente criar regiões paralelas e distribuir a carga de trabalho entre os núcleos de CPU disponíveis. Isso permite que várias tarefas sejam executadas simultaneamente, acelerando o processamento do problema da mochila binária.

## Algoritmo implementado

```

exhaustive_return exhaustive(
    const vector<movie_item> &mis,
    vector<int> &max_by_cat,
    const int num_movies,
    const int num_threads,
    const int num_categories
)
{
    unsigned long int max_solutions = pow(2, num_movies);

    int best_num_movies = 0;
    int best_screen_time = 0;

#pragma omp parallel
{
#pragma omp parallel for shared(best_num_movies, best_screen_time)
    for (unsigned long int i = 0; i < max_solutions; i++)
    {
        vector<int> max_by_cat_used(num_categories, 0);

        bitset<31> solution(i); // 31 bits (cada bit representa um filme)
        bitset<24> time_available; // 24 bits (cada bit representa um horário)

        int screen_time = 0; // Tempo total de tela da solução
        int total_movies = 0; // Número total de filmes na solução

        // Percorremos cada bit da solução.
        for (int j = 0; j < num_movies; j++)
        {
            // Se o bit for 1, então o filme está na solução.
            if (solution[j] == 1)
            {

```

```

// Verificamos se ainda há filmes disponíveis na categoria do filme.
int category_id = mis[j].m.category - 1;
if (max_by_cat_used[category_id] < max_by_cat[category_id])
{
    // Verificamos se o filme pode ser adicionado na solução.
    bitset<24> is_addable = mis[j].time_used & time_available;

    // Se o filme não puder ser adicionado, ou se não houver
    // mais filmes disponíveis na categoria dele, então não adicionamos.
    if (is_addable.none())
    {
        time_available = time_available | mis[j].time_used;
        max_by_cat_used[category_id] += 1;
        screen_time += mis[j].m.duration;
        total_movies += 1;
    }
}
}

#pragma omp critical
{
    if (total_movies > best_num_movies)
    {
        best_num_movies = total_movies;
        best_screen_time = screen_time;
    }
}
}

exhaustive_return solution;

solution.screen_time = best_screen_time;
solution.total_movies = best_num_movies;

return solution;
}

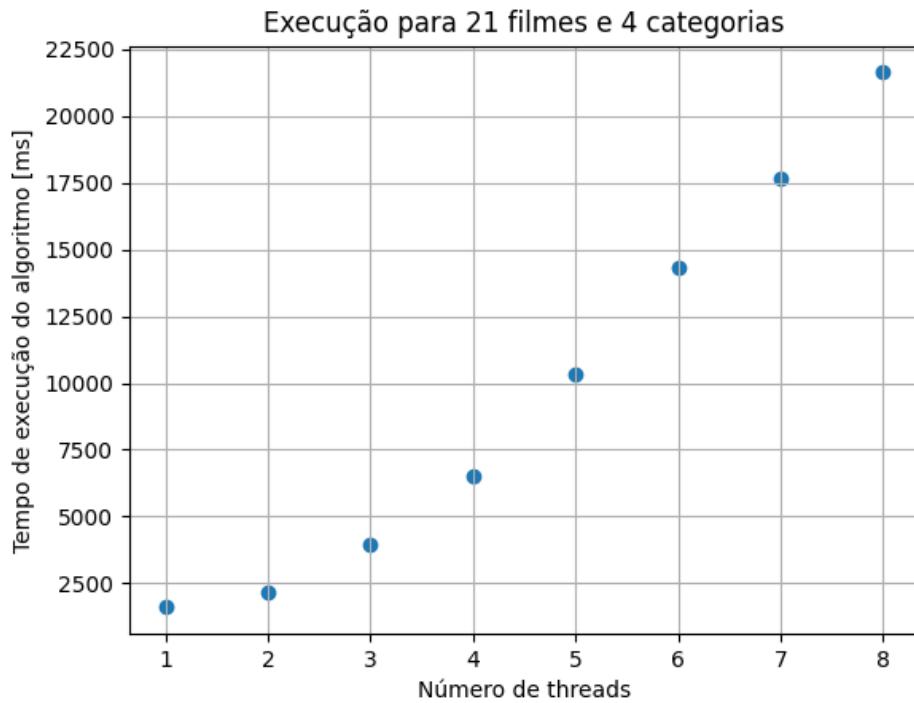
```

No código, a função `exhaustive` implementa o algoritmo exaustivo para resolver o problema da mochila binária. Ela recebe os parâmetros necessários, incluindo um vetor de `movie_item` representando os filmes, um vetor de `max_by_cat` com os limites máximos de filmes por categoria, o número total de filmes, o número de threads a serem utilizadas e o número total de categorias.

Dentro da função `exhaustive`, é calculado o número máximo de soluções possíveis usando o valor `num_movies` e então ocorre a paralelização do loop principal utilizando a diretiva `#pragma omp parallel for`. Cada iteração do loop representa uma solução candidata, onde são feitas verificações e cálculos para determinar se um filme pode ser adicionado à solução. O número total de filmes e o tempo de tela são atualizados a cada iteração e, ao final de cada iteração, ocorre uma seção crítica (`#pragma omp critical`) para atualizar os melhores valores encontrados até o momento.

## Resultados

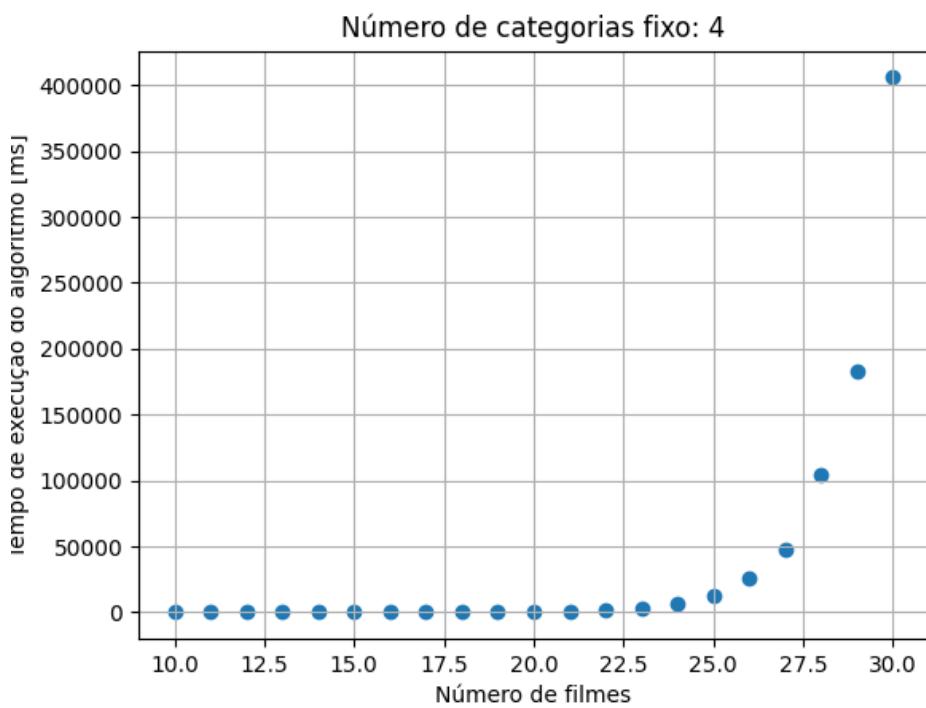
Começando pelos resultados individuais da implementação com OpenMP, foram obtidos os seguintes gráficos:



Ao aumentar o número de threads de 1 a 8, é esperado que o tempo de execução seja reduzido, já que o trabalho é dividido entre as threads e executado em paralelo. No entanto, o desempenho não deve ser linearmente proporcional ao número de threads, devido a fatores como sobrecarga de comunicação e contenção de recursos.

Alguns motivos que podem ter levado a esse resultado incluem:

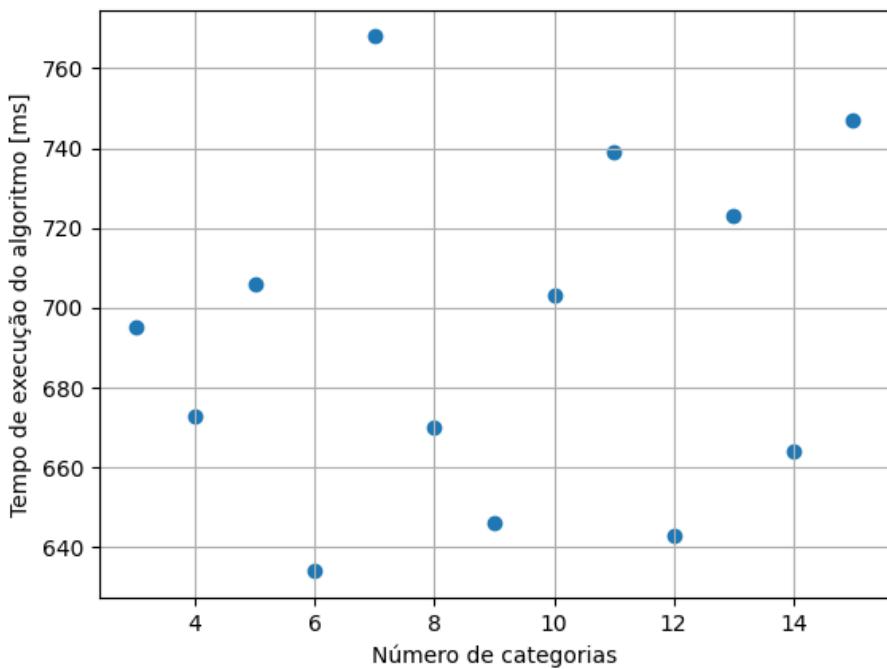
1. **Sobrecarga de comunicação:** A comunicação entre as threads pode levar a um aumento no tempo de execução à medida que o número de threads aumenta. Se a quantidade de comunicação necessária entre as threads for significativa, pode haver um ponto em que o benefício de paralelização seja anulado pela sobrecarga de comunicação.
2. **Contenção de recursos:** Se as threads competirem por recursos compartilhados, como memória ou acesso a dispositivos de E/S, pode ocorrer contenção de recursos. Isso pode resultar em um gargalo e limitar o desempenho do algoritmo, independentemente do número de threads.
3. **Granularidade da tarefa:** Se a tarefa a ser paralelizada não for granular o suficiente, pode haver limitações no desempenho. Por exemplo, se cada iteração do loop principal da busca exaustiva for muito curta, o tempo gasto na criação e gerenciamento de threads pode superar o benefício da paralelização.



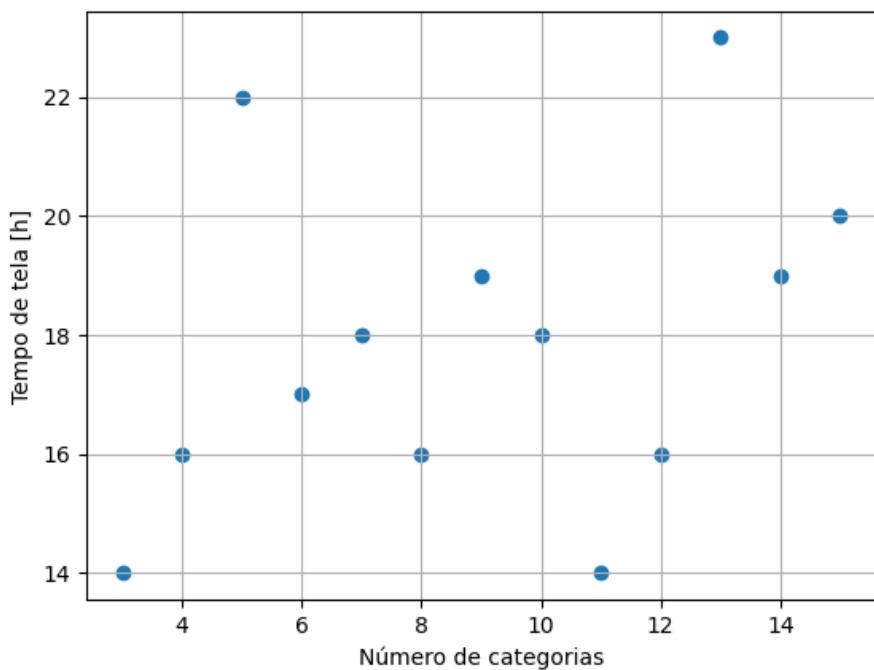
O gráfico acima mostra que, a partir de aproximadamente 25 filmes, o tempo gasto no processo começa a subir a uma velocidade muito elevada. Afinal, para 30 filmes, estamos trabalhando com  $2^{30} = 1.073.741.824$  possíveis soluções!

Os demais gráficos, apresentados abaixo, são mais para visualização e não fornecem grandes informações adicionais. O que se pode notar é que um maior número de filmes acarreta em maior tempo de tela e maior número de filmes selecionados. Mas, de certa forma, isso é esperado, já que o intervalo de filmes com o qual se está trabalhando não é grande.

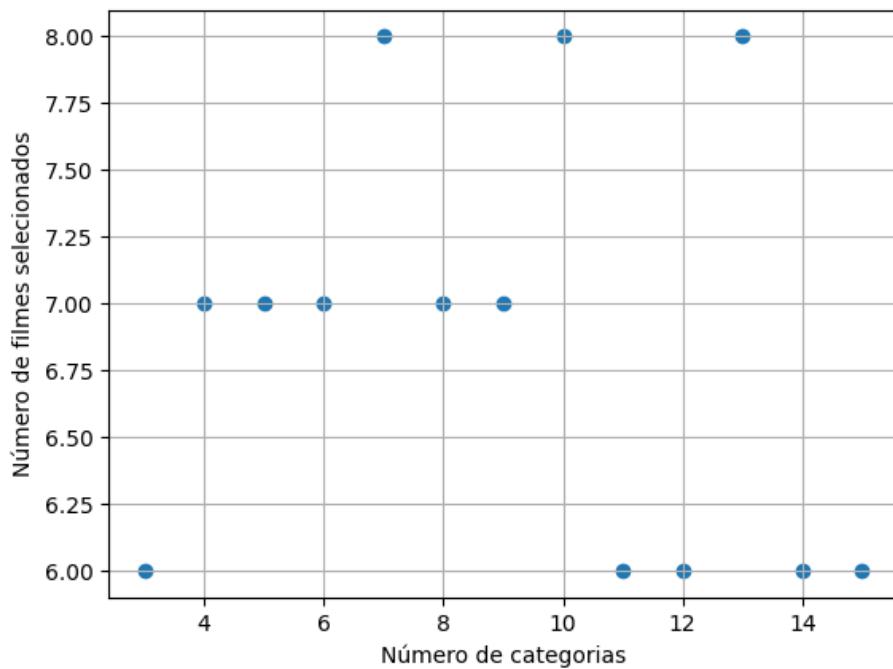
Número de filmes fixo: 21



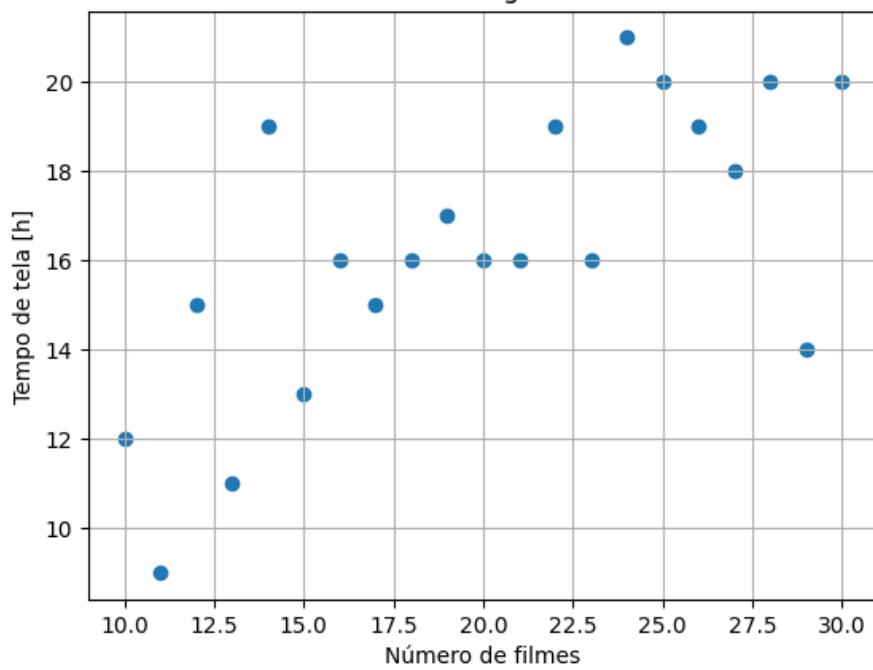
Número de filmes fixo: 21

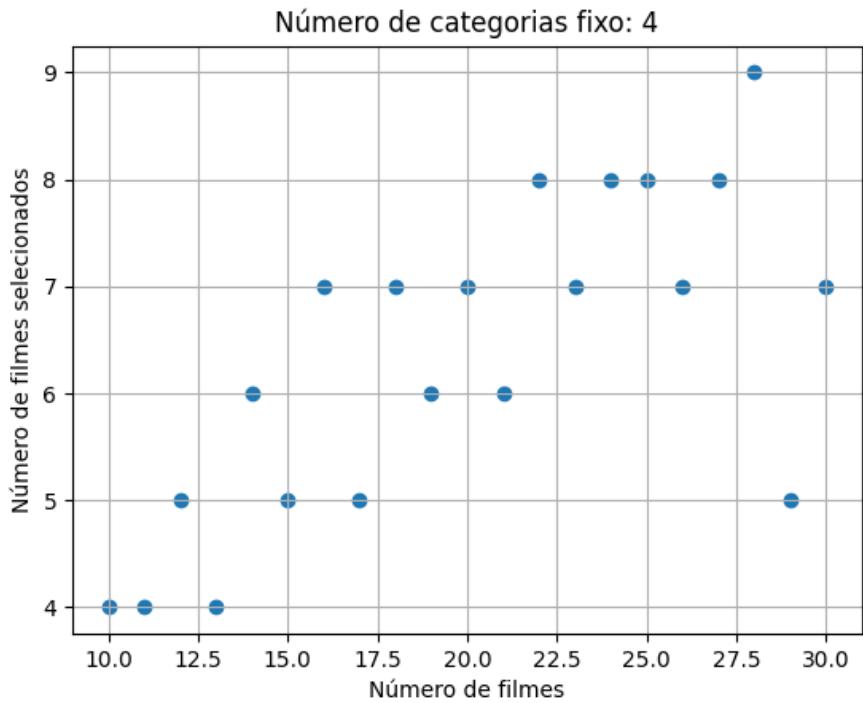


Número de filmes fixo: 21



Número de categorias fixo: 4





## Paralelismo com GPUs

O paralelismo desempenha um papel crucial no campo da computação moderna, permitindo a execução simultânea de tarefas em diferentes processadores ou unidades de processamento gráfico (GPUs). A GPU, originalmente projetada para manipular gráficos, evoluiu ao longo do tempo para se tornar uma poderosa ferramenta de computação paralela. A capacidade de processamento massivamente paralelo das GPUs as torna uma escolha ideal para acelerar algoritmos e resolver problemas complexos em uma fração do tempo necessário pelas CPUs convencionais.

Uma das áreas onde o paralelismo com GPUs tem sido amplamente aplicado é a otimização combinatória, que envolve a busca de soluções ótimas ou aproximadas para problemas de grande escala. Um exemplo comum nesse domínio é o problema da mochila binária, que envolve selecionar um subconjunto de itens com valores e pesos diferentes para maximizar o valor total, sujeito a uma restrição de capacidade. Resolver esse problema de forma eficiente é de grande importância em várias aplicações, como planejamento de recursos, alocação de tarefas e gerenciamento de estoques.

No contexto desse relatório, exploramos a aplicação do paralelismo com GPUs na resolução do problema da mochila binária com base em um conjunto de dados de filmes assistidos. Para implementar essa abordagem, utilizamos a linguagem de programação C++ em conjunto com a biblioteca Thrust, que fornece uma interface de programação de alto nível para a programação paralela em GPUs NVIDIA.

O uso do paralelismo com GPUs para resolver o problema da mochila binária oferece várias vantagens significativas. Primeiramente, a GPU é capaz de lidar com uma grande quantidade de dados simultaneamente, processando várias soluções candidatas em paralelo. Isso resulta em um ganho de desempenho significativo em comparação com a execução sequencial em uma CPU convencional. Além disso, a arquitetura altamente paralela da GPU permite que diferentes threads trabalhem em partes independentes do problema, explorando ao máximo o poder de processamento da GPU e acelerando o tempo de execução.

## Algoritmo implementado

```

void dynamic_program_gpu(
    vector<movie> &movies,
    vector<int> &max_by_cat,
    int num_categories,
    int num_movies,
    return_gpu &solution
)
{
    unsigned long int num_combinations = pow(2, num_movies); // Number of possible combinations

    thrust::device_vector<movie> movies_gpu(movies); // Vector with all movies in GPU
    thrust::device_vector<int> max_by_cat_gpu(max_by_cat); // Vector with max_by_cat in GPU
    thrust::device_vector<int> movie_combinations_gpu(num_combinations); // 2 ^ num_movies

    thrust::counting_iterator<int> counter(0); // 2 ^ num_movies (for movie_combinations_gpu)

    thrust::transform(
        counter, // Start of input
        counter + num_combinations, // End of input
        movie_combinations_gpu.begin(), // Output
        customized_operator(
            num_movies, // Number of movies
            thrust::raw_pointer_cast(movies_gpu.data()), // Pointer to movies in GPU
            thrust::raw_pointer_cast(max_by_cat_gpu.data()), // Pointer to max_by_cat in GPU
            num_categories // Number of categories
        ) // Unary Operator
    );

    // Find the maximum element in movie_combinations_gpu
    auto max_element_it = thrust::max_element(movie_combinations_gpu.begin(), movie_combinations_gpu.end());

    // Calculate the index of the maximum element
    int max_element_index = thrust::distance(movie_combinations_gpu.begin(), max_element_it);

    // Obtain the value of the maximum element (number of movies that can be scheduled)
    int max_element_value = *max_element_it;

    bitset<30> bitset(max_element_index);
    int screen_time = 0;

    for (int i = 0; i < num_movies; i++)
    {
        if (bitset[i])
        {
            screen_time += movies[i].duration;
        }
    }

    solution = {max_element_value, screen_time};
}

```

O código fornecido (o código acima é apenas uma parte da implementação) implementa um algoritmo de programação dinâmica para encontrar a melhor combinação de filmes que maximize o tempo de exibição em uma tela de cinema. Ele usa a biblioteca `Thrust` para realizar cálculos paralelos na GPU, visando melhorar o desempenho.

O algoritmo começa lendo os dados de entrada, que incluem o número de filmes, o número de categorias e os detalhes de cada filme, como o horário de início, o horário de término e a categoria. Em seguida, ele calcula a duração de cada filme com base nos horários de início e término.

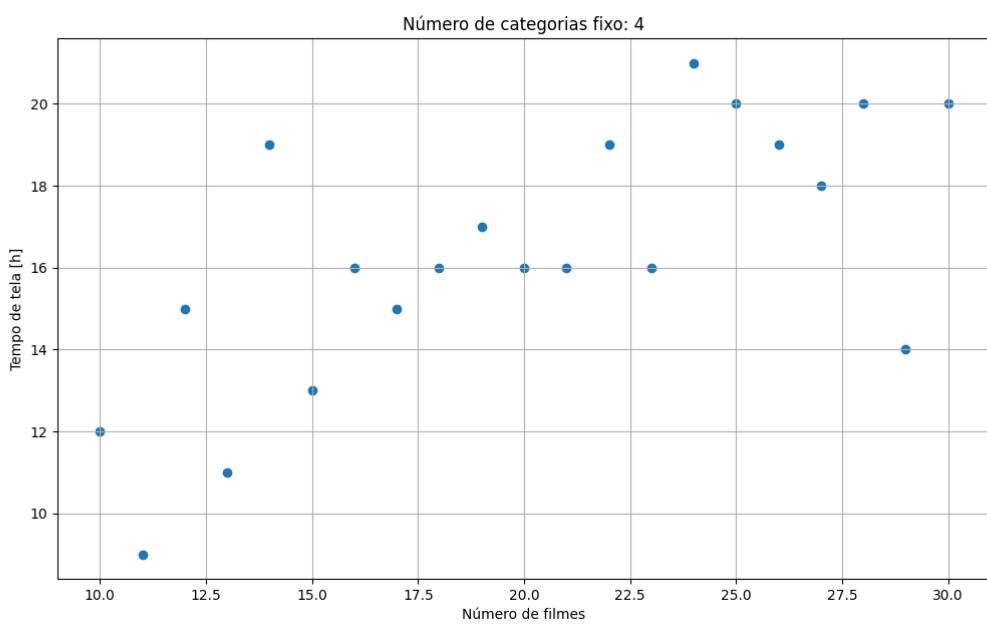
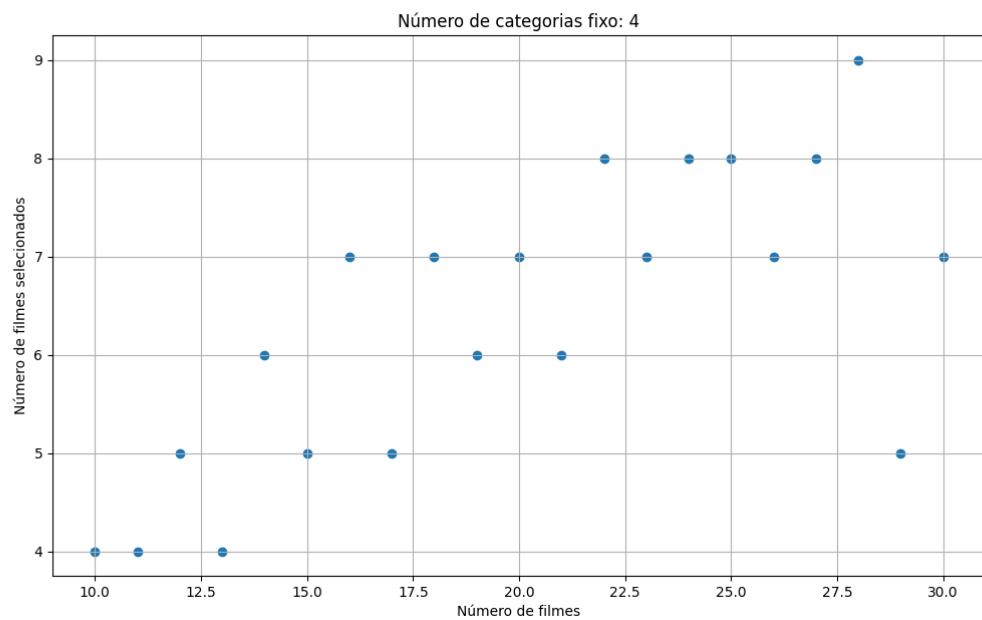
Em seguida, o algoritmo executa a etapa principal de programação dinâmica na GPU. Ele cria uma representação em vetor dos filmes e dos limites máximos de filmes por categoria na GPU. Em seguida, ele gera todas as possíveis combinações de filmes e calcula o número de filmes que podem ser agendados para cada combinação. Isso é feito aplicando um operador personalizado a cada combinação usando a função `thrust::transform`.

Após calcular o número de filmes que podem ser agendados para cada combinação, o algoritmo encontra a combinação que permite agendar o maior número de filmes, usando a função `thrust::max_element`. Ele também calcula a duração total dos filmes selecionados.

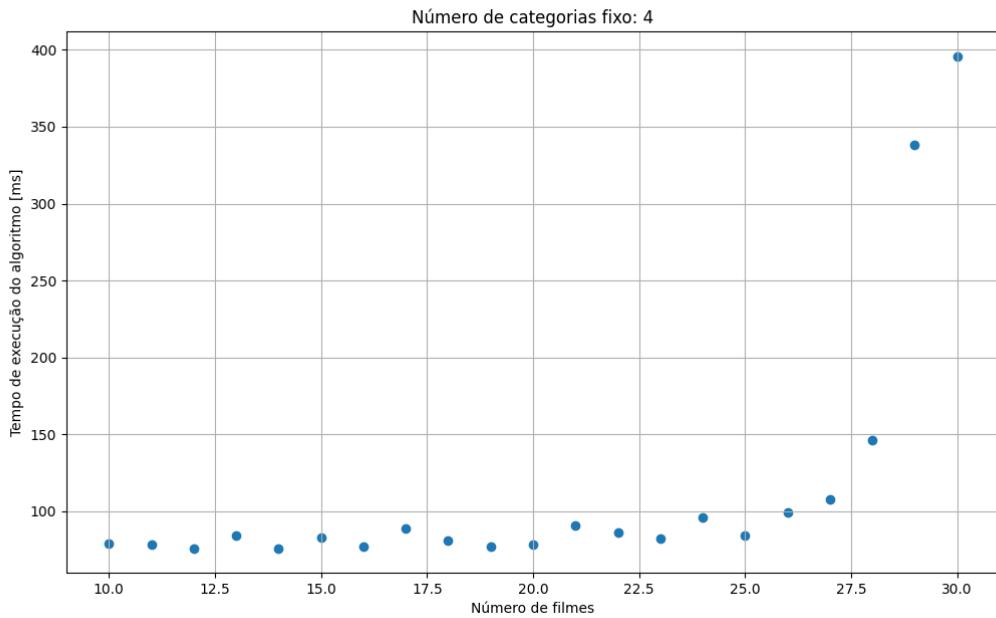
Finalmente, o algoritmo imprime os resultados, exibindo o número de filmes selecionados, o tempo de execução do algoritmo e o tempo total de exibição dos filmes, os quais são capturados pelo código python para geração dos gráficos.

## Resultados

Os resultados obtidos para o paralelismo com GPU podem ser visualizados a seguir:

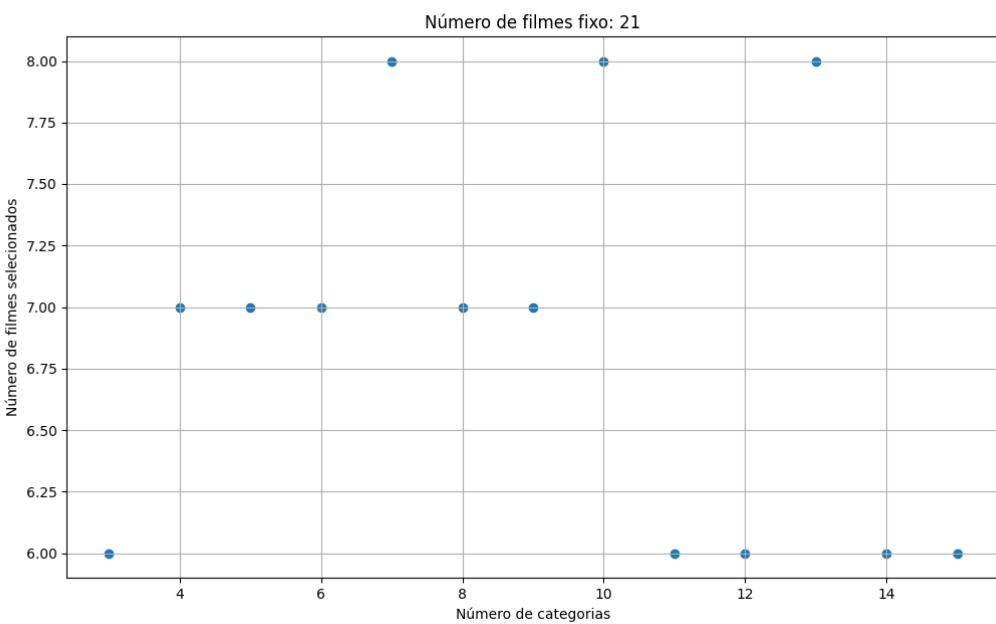


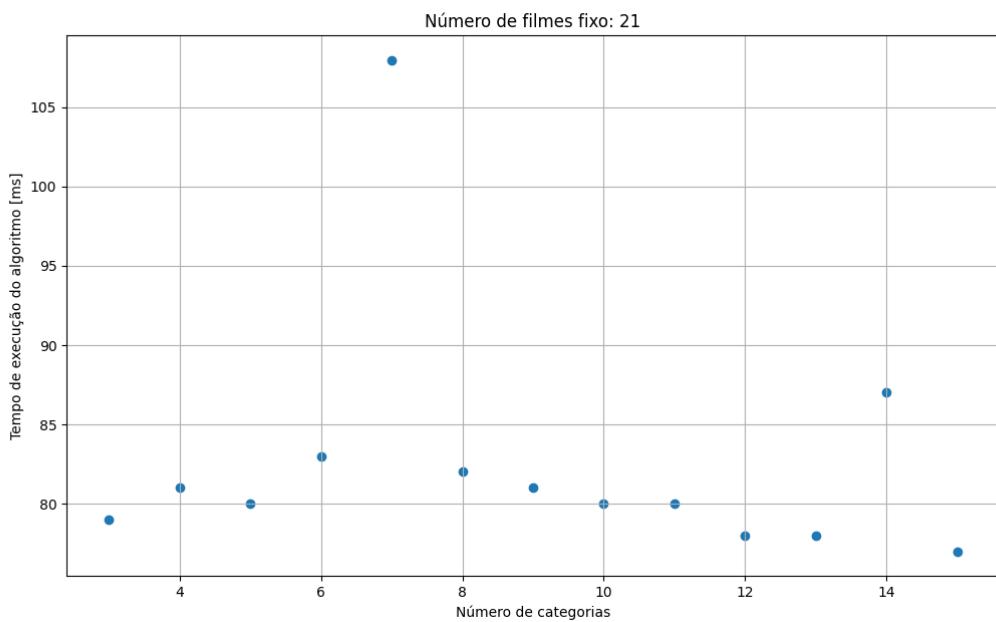
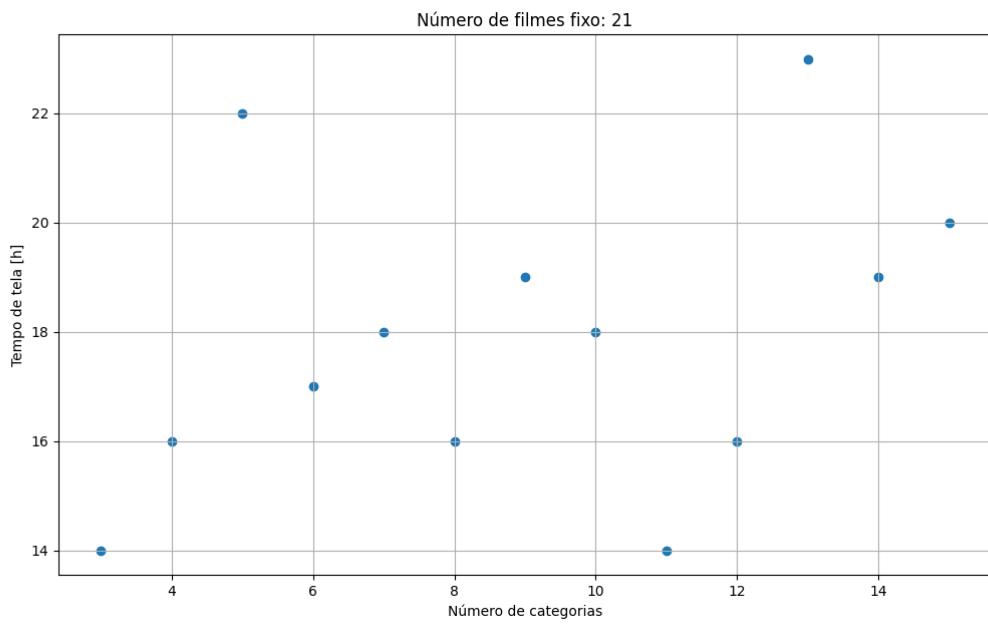
Os dois gráficos acima refletem o que já foi visto na implementação com OpenMP, que o aumento no número de filmes aumenta o tempo de tela e o número de filmes selecionados, visto que se está trabalhando em um intervalo pequeno de filmes.



De forma semelhante ao OpenMP, a partir de um dado número de filmes, o tempo de processamento começa a subir muito rapidamente (ainda que seja mais rápido que o paralelismo feito com OpenMP).

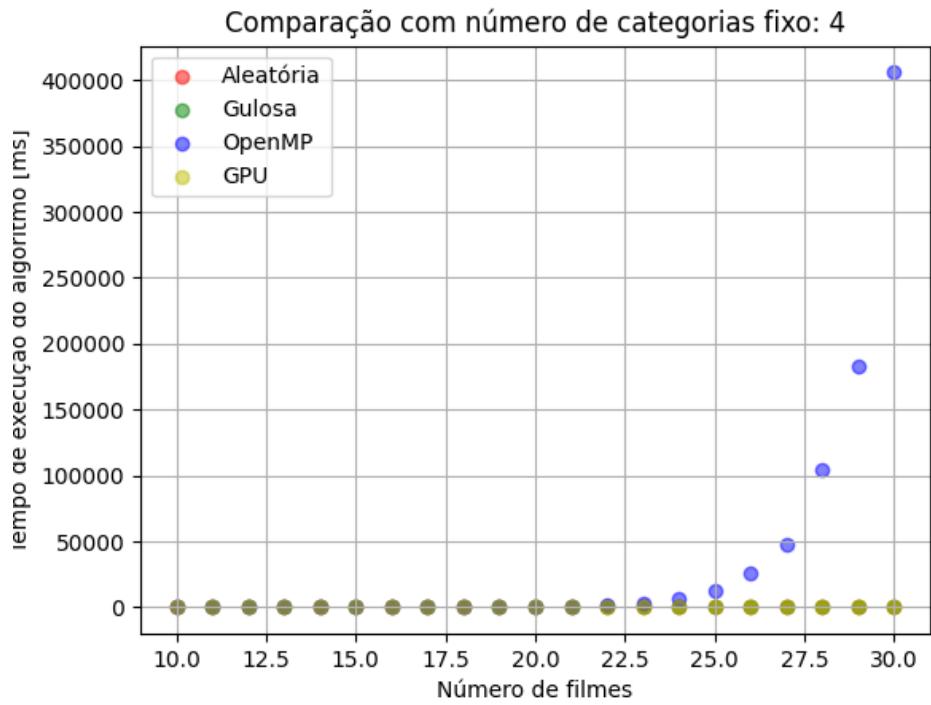
Os demais gráficos são apenas de caráter mais exploratório.





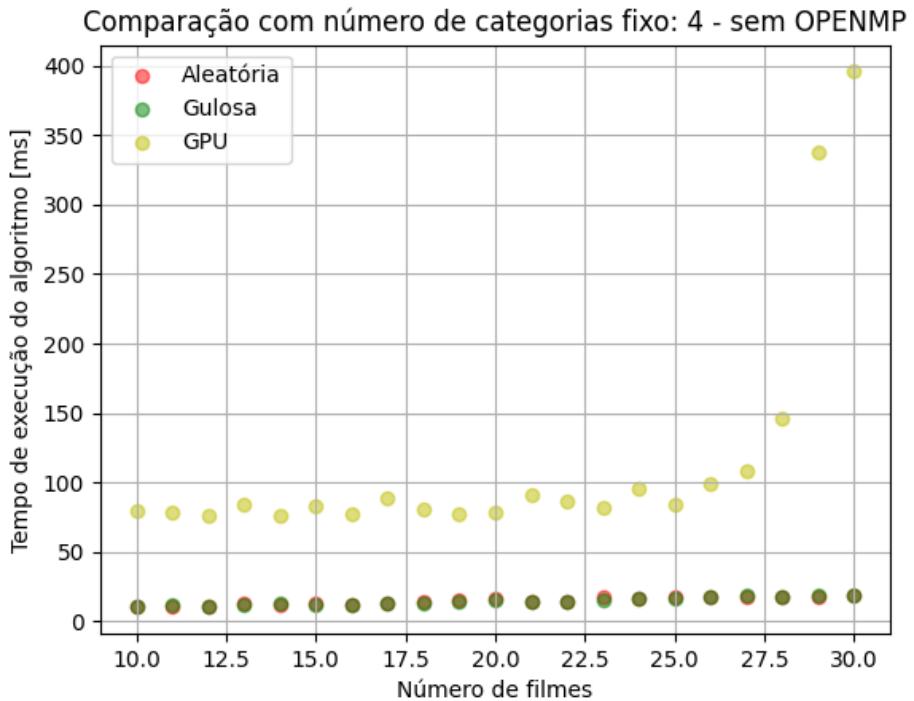
## Comparação entre todos os modelos

A seguir, tem-se a comparação entre todos os modelos trabalhados até aqui (vale lembrar que, nesse ponto, as heurísticas aleatória e gulosa foram alteradas para comportarem um número menor de filmes, com o intuito de se estabelecer uma comparação frente aos modelos de paralelismo).

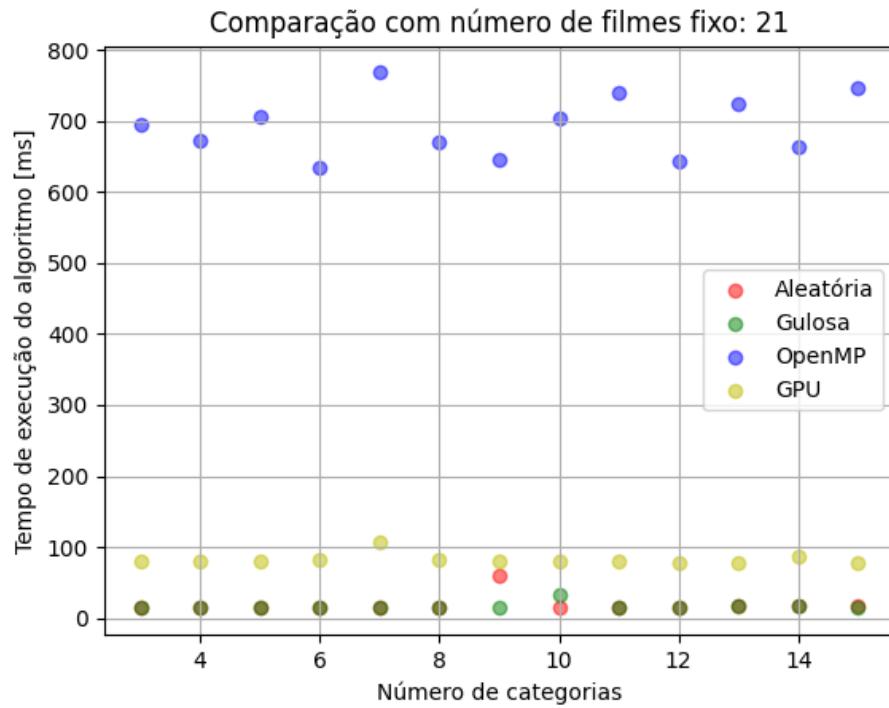


O gráfico acima mostra o desbalanço no tempo de execução da solução implementada com OpenMP das demais. Certamente, dentro dessas condições de infraestrutura e dados, para um grande número de filmes a abordagem paralela com OpenMP mostra-se praticamente impraticável.

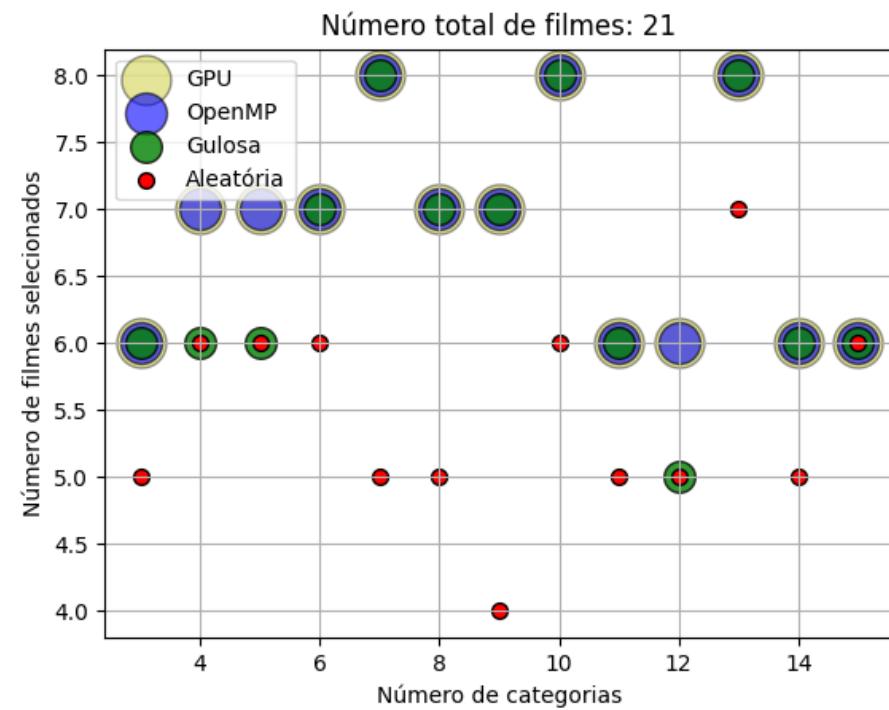
Então, tirando o OpenMP da análise desse caso, o gráfico ficaria assim:



No gráfico acima, vemos que os tempos da implementação paralela GPU é próxima das outras duas, mas a partir de um ponto ela começa a se tornar muito mais demorada, enquanto que as heurísticas aleatória e gulosa crescem no tempo a uma velocidade baixíssima.



Ao variar o número de categorias, o “delta” de tempo entre as implementações não foi tão significativo, embora a abordagem com OpenMP ainda continue demandando mais tempo que qualquer outra.



O gráfico acima é mais de caráter ilustrativo, mostrando o número de categorias vs número de filmes selecionados. Em vários pontos há coincidências de resultados.

## Conclusão

---

Diante de todos esses resultados, é perceptível que a abordagem exaustiva demanda grandes recursos computacionais para ser capaz de apontar a melhor solução, pois devem ser testadas todas as possibilidades. Nesse sentido, pôde-se notar que a abordagem paralela com GPU se saiu melhor nos testes no quesito de tempo de processamento, pois foi executada mais rapidamente. Vale lembrar, contudo, que o problema da mochila binária trata-se de um problema do tipo NP-Completo, portanto ainda não é possível fazer um código C++ para que resolva esse problema em tempo polinomial.

Outro ponto importante visto nos resultados foi que o aumento do número de threads na execução com OpenMP não melhorou a performance da solução; pelo contrário, o processo ficou cada vez mais demorado.

As abordagens paralelizáveis, no final dos experimentos, tiveram um desempenho médio de tempo inferior às heurísticas gulosa e aleatória. Isso é até de se esperar, já que a abordagem exaustiva trabalha com uma quantidade de possibilidades de resposta muito superior às demais por levar em conta todas as combinações possíveis. Para  $n$  suficientemente grande, as abordagens vistas na primeira etapa do relatório são mais praticáveis no quesito de complexidade, embora elas não avaliem todas as possíveis soluções do problema para encontrar a melhor resposta.