Instituto Politécnico do Cávado e Ave

Mestrado em Engenharia Informática

Computação de Alto Desempenho

Carlos Baixo, 16949
Carlos Beiramar, 29988
João Silva, 16951

5 de maio de 2024

Conteúdo

1 Introdução

No presente relatório será abordado um problema muito comum na programação concorrente denominado de Job Shop. Este problema consiste num conjunto de máquinas e trabalhos onde cada trabalho é composto por uma série de operações a serem realizadas numa ordem específica e cada uma delas numa máquina designada.
O objetivo do Job Shop é determinar uma sequência de operações que minimize uma métrica específica, como o tempo total para concluir todos os trabalhos (makespan), ou para otimizar a utilização dos recursos, reduzindo tempos de inatividade ou melhorando o fluxo de produção.
Este relatório tem como objetivo apresentar duas possíveis implementações para este problema, uma sequencial e outra paralela.

2 Descrição do Problema

O problema do *Job Shop* consiste na alocação de recursos, onde múltiplos trabalhos devem ser realizados usando um conjunto de máquinas. Cada trabalho é composto por um conjunto de operações que devem ser realizadas por sequência em várias máquinas. Visto que uma máquina não consegue realizar todas as operações, cada trabalho contém um conjunto de operações onde cada operação tem de ser feita numa máquina específica.
Assim, as restrições deste problema podem ser resumidas em dois pontos importantes:

- **Sequência de operações**; Cada trabalho deve seguir uma ordem específica de operações, ou seja, uma operação só pode começar após a anterior ter sido concluída. Isso significa que, para cada trabalho, há uma cadeia linear de operações que devem ser realizadas.
- **Exclusividade das máquinas**: Uma máquina só pode realizar uma operação de cada vez. Se duas operações necessitarem de usar a mesma máquina, uma deve esperar até que a outra termine.

Com estas restrições, o objetivo é encontrar um escalonamento válido, atribuindo um tempo de início para cada operação, de forma a cumprir as restrições de sequência e exclusividade. Para além disso, o tempo de execução deve estar otimizado.

2.1 Dados de entrada

Este problema recebe como entrada a quantidade de máquinas e de trabalhos e uma tabela com a identificação das máquinas capazes de executarem cada uma das operações de cada trabalho, assim como, a sua duração. Como exemplo:

Trabalho	Operação 1	Operação 2	Operação 3
job0	(M0, 3)	(M1, 2)	(M2, 2)
job1	(M0, 2)	(M2, 1)	(M1, 4)
job2	(M1, 4)	(M2, 3)	(M0, 1)

Tabela 1: Escalonamento de Operações por Trabalho

Na implementação deste projeto, este dados serão inseridos na aplicação através de um ficheiro $*$**.jss**:

```
3 3
0 3 1 2 2 2
0 2 2 1 1 4
1 4 2 3 0 1
```

3 Estrutura de dados

Como estruturas de dados para armazenar toda a informação presente nos ficheiros de *input*, foram implementadas duas estruturas de dados.

- Cada operação contém o número da máquina onde deverá ser efetuada e o seu tempo de duração. A seguinte estrutura representa como serão armazenadas as operações:

```
1 struct Operation{
2     int machine_number;
3     int duration;
4 };
5
```

- Cada *Job* irá conter o seu **id**, o número total de operações e uma lista de operações. A seguinte estrutura representa como serão armazenados os Jobs:

```
1 #define MAX_OPERATIONS 100
2
3 struct Job{
4     int job_number;
5     int total_operations;
6     struct Operation operations[MAX_OPERATIONS];
7 };
8
```

- Foi também implementada uma estrutura que irá guardar toda a informação necessária para imprimir o resultado final, ou seja, irá guardar os tempos iniciais de cada operação para cada *Job*.

```
1 struct Output_time{
2     int job_number;
3     int *start_time_operations;
4 }
5
```

Assim, após a leitura do ficheiro de entrada, será criado um *array* estático da *struct Job* cujo o tamanho será o número de *jobs* necessário. Após a leitura total do ficheiro de *input*, por exemplo o ficheiro referido anteriormente, será possível obter uma estrutura semelhante à seguinte:

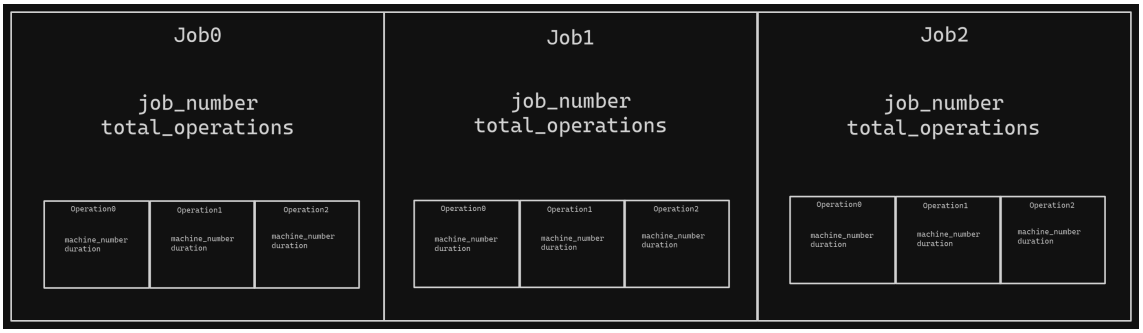

Figura 1: Exemplo de uma estrutura após leitura do ficheiro input

Estas serão as estruturas base implementadas para armazenar toda a informação necessária relativamente ao problema apresentado no enunciado mas, foi necessário implementar uma outra estrutura que nos permite partilhar recursos entre as *threads* de forma a obter o resultado pretendido.

4 Implementação Sequencial

A abordagem para implementar o algoritmo de resolução para este problema de forma sequencial tornou-se menos complexa após a criação das estruturas de dados referidas anteriormente. De seguida irá ser detalhado passo a passo o algoritmo implementado:

- Inicialmente são criados dois *arrays* estáticos que irão armazenar os tempos das máquinas e dos *Jobs*.

```
int machines[number_of_machines];
int job_completion_times[number_of_jobs];
```

- Ambos os array terão as suas posições inicializadas a 0.

```
memset(machines, 0, sizeof(machines));
memset(job_completion_times, 0, sizeof(job_completion_times));

```

- De seguida são criados dois for *loops* que irão iterar o *array* de *jobs* e dentro de cada *job* será iterado o *array* de operações.

```
for (int i = 0; i < number_of_jobs; i++){
    for (int j = 0; j < jobs[i].total_operations; j++){
        int machine_id = jobs[i].operations[j].machine_number;
        int duration = jobs[i].operations[j].duration;
        int start_time = (machines[machine_id] > job_completion_times[i]) ?
    machines[machine_id] : job_completion_times[i];
        int end_time = start_time + duration;
        machines[machine_id] = end_time;
        job_completion_times[i] = end_time;
        output_time[i].start_time_operations[j] = start_time;
    }
}

```

De forma a simular o comportamento deste algoritmo, será apresentado o resultado de todas as iterações após aplicar o algoritmo ao input referido anteriormente.

```
3 3
0 3 1 2 2 2
0 2 2 1 1 4
1 4 2 3 0 1

//i = 0 && j = 0
job_number = 0
machine = 0
duration = 3
start_time = 0
end_time = 3
machines = [3,0,0]
job_completion_times = [3,0,0]
make_span = 3

//i = 0 && j = 1
job_number = 0
machine = 1
duration = 2
start_time = 3
```

```
end_time = 5
machines = [3,5,0]
job_completion_times = [5,0,0]
make_span = 5

//i = 0 && j = 2
job_number = 0
machine = 2
duration = 2
start_time = 5
end_time = 7
machines = [3,5,7]
job_completion_times = [7,0,0]
make_span = 7

//i = 1 && j = 0
job_number = 1
machine = 0
duration = 2
start_time = 3
end_time = 5
machines = [5,5,7]
job_completion_times = [7,5,0]
make_span = 7

// i = 1 && j = 1
job_number = 1
machine = 2
duration = 1
start_time = 7
end_time = 8
machines = [5,5,8]
job_completion_times = [7,8,0]
make_span = 8

// i = 1 && j = 2
job_number = 1
machine = 1
duration = 4
start_time = 8
ent_time = 12
machines = [5,12,8]
job_completion_times = [7,12,0]
make_span = 12

// i = 2 && j = 0
job_number = 2
machine = 1
duration = 4
start_time = 12
end_time = 16
machines = [5,16,8]
job_completion_times = [7,12,16]
make_span = 16

// i = 2 && j = 1
job_number = 2
machine = 2
duration = 3
start_time = 16
```

```
end_time = 19
machines = [5,16,19]
job_completion_times = [7,12,19]
make_span = 19

// i = 2 && j = 2
job_number = 2
machine = 0
duration = 1
start_time = 19
end_time = 20
machines = [20,16,19]
job_completion_times = [7,12,20]
make_span = 20
```

Como é possível verificar, para implementar o algoritmo sequencial, foi criado um *array* de *jobs* e após isso, iterou-se esse *array* e obteve-se o resultado final.

5 Implementação Paralela

Com o objetivo de melhorar a performance do algoritmo descrito anteriormente, foram aplicados os conceitos lecionados na unidade curricular sobre computação paralela e, para isso, o grupo utilizou a biblioteca **pthread.h** o que permitiu fazer as alterações necessária e explorar o uso de múltiplas threads.
Foi implementada uma nova *struct* para encapsular os dados necessários para cada *thread*, permitindo assim haver partilha de memória entre cada *thread*.

```
struct Thread_Args{
    struct Job job;
    int *start_time_operations;
    pthread_mutex_t* mutexes_machines;
    int* machines;
    int* job_completion_times;
    int number_of_jobs;
};

```

Esta nova estrutura permitiu partilhar algumas informações entre as *threads*:

- **struct Job job** - cada thread está associada cada *job*.
- **start_time_operations** - Cada job tera associado um *array* que irá guardar o start_time de cada uma das suas operações.
- **mutexes_machines** - representa um *array* de mutexes para as máquinas.
- **machines** - apontador para um *array* que irá representar os tempos nas máquinas.
- **job_completion_times** - apontador para um *array* que irá representar os tempos dos *jobs*.
- **number_of_jobs** - representa o número de *jobs*.

No desenvolvimento da solução para a implementação paralela foi possível detetar qual seria a possível **secção crítica** no código e que poderia levar a **condições de corrida** e, essa secção crítica seria:

```
int start_time = (machines[machine_id] > job_completion_times[job.job_number]) ?
    machines[machine_id] : job_completion_times[job.job_number];
int end_time = start_time + duration;
machines[machine_id] = end_time;
```

No excerto de código anterior é possível observar que os valores presentes no *array* **machines** são alterados por cada *thread* o que, poderia levar a uma condição de corrida caso duas *threads* tentassem alterar o valor em simultâneo. Assim, a solução passou por aplicar um *lock* no *index* que se pretende alterar.

```
1 pthread_mutex_lock(&mutexes_machines[machine_number]);
2 int start_time = (machines[machine_id] > job_completion_times[job.job_number]) ?
      machines[machine_id] : job_completion_times[job.job_number];
3 int end_time = start_time + duration;
4 machines[machine_id] = end_time;
5 pthread_mutex_unlock(&mutexes_machines[machine_number]);
```

O mesmo procedimento foi aplicado para a variável global **make_span**. Foi então implementada uma função que irá conter todo o código necessário para lidar com os *locks* e *unlocks* e, esta função será chamada em cada uma das *threads*.

5.1 Número de threads

O número de *threads* é passado por argumento pelo utilizador e, houveram alguns cuidados a ter para o código aceitar qualquer uma das seguintes hipóteses:

- O número de *threads* é inferior ao número de *jobs*.
- O número de *threads* é igual ao número de *jobs*.
- O número de *threads* é superior ao número de *jobs*.

No primeiro caso, se o número de *threads* é inferior ao número de *jobs*, o procedimento é o seguinte: é criado um array com uma *thread* em cada um dos índices, executa-se o código tantas vezes quantas *threads* existirem e, após a verificação de que o número de *threads* é inferior ao número de *jobs*, volta-se a iterar o array de *threads* desde o início, espera-se que a *thread* daquele respetivo índice termine e executa-se todo o código para concretizar mais um *job*. Este procedimento é repetido até todos os *jobs* serem executados.
No segundo caso não é necessário nenhum cuidado especial visto que cada *thread* será aplicada a cada um dos *jobs*, pois existem tantas *threads* quanto *jobs*.
Caso o número de *threads* seja superior ao número de *jobs*, antes de executar o código, é feita uma verificação para garantir que o limite de iterações não ultrapassa o número de *jobs* existente.

6 Resultados obtidos

6.1 Implementação sequencial

Neste implementação o resultado será sempre o mesmo visto que, os *jobs* e as operações são iterados por ordem e sequencialmente. Assim, alguns dos resultados obtidos foram:

- Nome do ficheiro input: **ft_03.jss**

Input File	**Output File**	**Result**
3 3 0 3 1 2 2 2 0 2 2 1 1 4 1 4 2 3 0 1	0 3 5 3 7 8 12 16 19	Makespan = 20

Tabela 2: Resultado para o ficheiro com 3 *jobs*

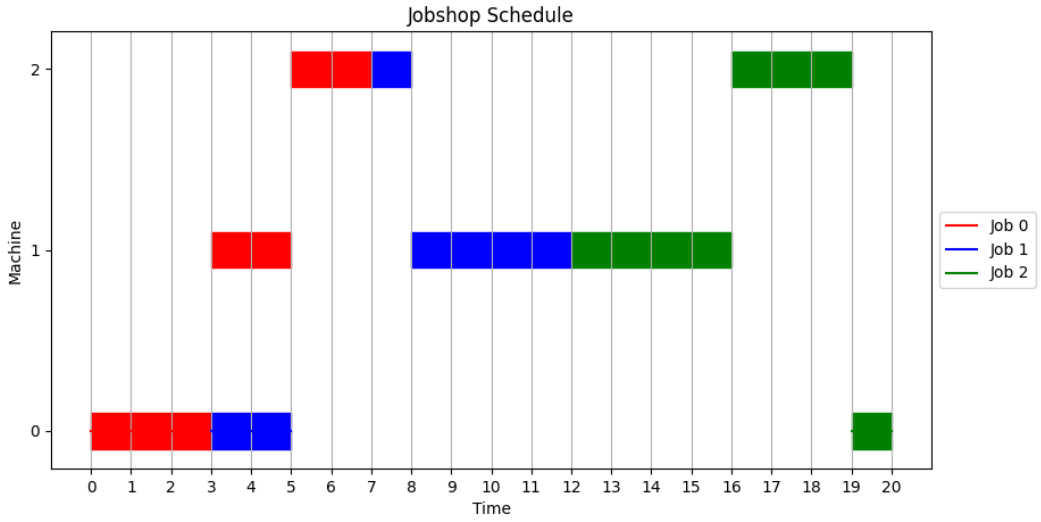

Figura 2: Diagrama do resultado anterior

6.2 Implementação paralela

Nesta implementação o resultado final da makespan varia visto que depende da primeira *thread* a fazer o *lock* da máquina. Assim, o melhor resultado obtido foi:

- Nome do ficheiro de input: **ft_03.jss**

Input File	Output File	Result
3 3 0 3 1 2 2 2 0 2 2 1 1 4 1 4 2 3 0 1	2 5 7 0 2 7 0 4 7	Makespan = 11

Tabela 3: Resultado para o ficheiro com 3 *jobs*

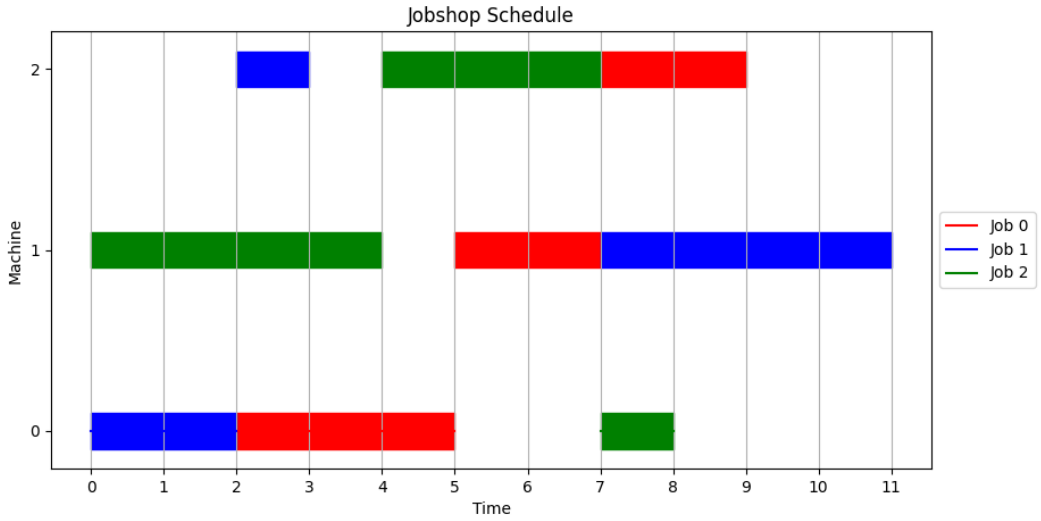

Figura 3: Diagrama do resultado anterior

De realçar que o algoritmo implementado tem a capacidade de correr ficheiros com um maior número de *jobs* apesar de, neste relatório, ser utilizado apenas o ficheiro com 3 *jobs* como exemplo.

7 Análise de Desempenho

Na implementação foram utilizadas duas formas distintas de medir o tempo de execução do programa. A primeira utiliza a função **clock()** da biblioteca **time.h** que irá medir o tempo usado pelo processador para executar o programa, a segunda utiliza a função **clock_gettime()** que pertence à biblioteca POSIX (Portable Operating System Interface) que permitirá medir o tempo real que o programa demorou a executar.

7.1 Computadores utilizados

7.1.1 Windows

- **Processador:** 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz
- **Número de núcleos:** 4
- **Número de threads:** 8
- **RAM:** 16 GB

7.1.2 MacBook

- **Processador:** Apple M1 chip
- **Número de núcleos:** 8
- **Número de threads:** 8
- **RAM:** 16 GB

7.2 Desempenho em função do número de threads

Implementou-se um programa em Python para processar a informação adquirida no job_shop de forma mais fácil e intuitiva. Para efetuar o processamento, criou-se um dataframe com a seguinte estrutura e dados.

	Computer	File	SEQ	PC1	PC2	PC4	PC8	PC16	PC32
0	mac	ft_03	0.00124	0.00133	0.00153	0.00051	0.00056	0.00046	0.00047
1	mac	ft_06	0.00604	0.00511	0.00290	0.00397	0.00158	0.00144	0.00155
2	mac	ft_20	0.01472	0.01508	0.00968	0.00707	0.00305	0.00328	0.00357
3	mac	ft_80	0.19112	0.20712	0.10778	0.06314	0.04863	0.04771	0.05022
4	mac	ft_1000	70.64145	70.85157	36.49682	18.94992	15.47075	15.34645	15.16807
5	windows	ft_03	0.00196	0.00287	0.00212	0.00115	0.00122	0.00146	0.00116
6	windows	ft_06	0.00760	0.00896	0.00605	0.00336	0.00178	0.00199	0.00218
7	windows	ft_20	0.01981	0.02613	0.01350	0.00777	0.00416	0.00389	0.00439
8	windows	ft_80	0.45939	0.44095	0.23632	0.13144	0.07491	0.08647	0.06439
9	windows	ft_1000	267.47008	266.53309	138.53338	94.37913	67.32317	47.65842	47.56434

Figura 4: Matriz de tempos

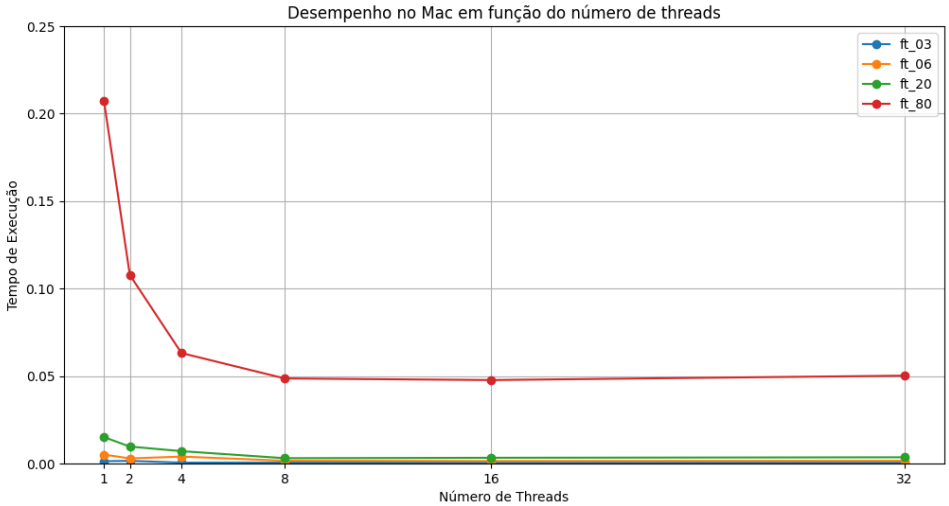

Figura 5: Performance Mac

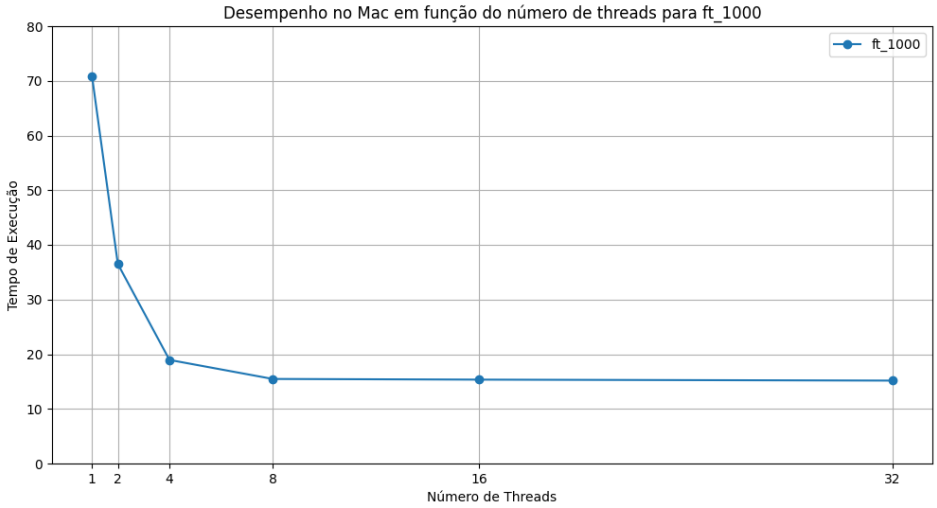

Figura 6: Performance Mac ficheiro 1000 x 10000

Com base nos gráficos apresentados, podemos concluir que, no programa sequencial, há um aumento no tempo de execução à medida que a complexidade dos ficheiros aumenta. Além disso, podemos verificar que, em ficheiros menos complexos, quase não há diferença nos tempos de execução do programa. No entanto, à medida que a complexidade aumenta, começa-se a perceber uma diferença de desempenho com o aumento de threads. Quanto mais threads o programa utiliza, melhor o desempenho, chegando a um limite de 8 threads, que é o limite físico de ambos os computadores.

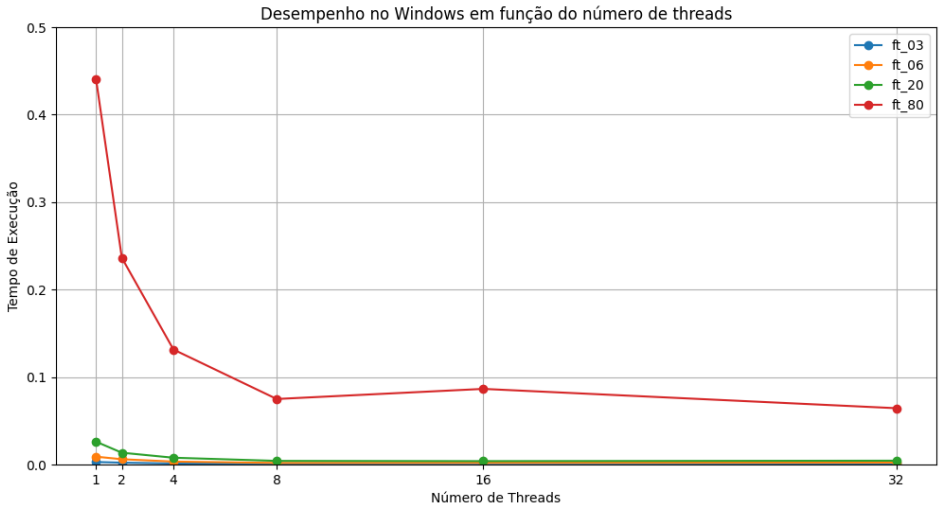

Figura 7: Performance Windows

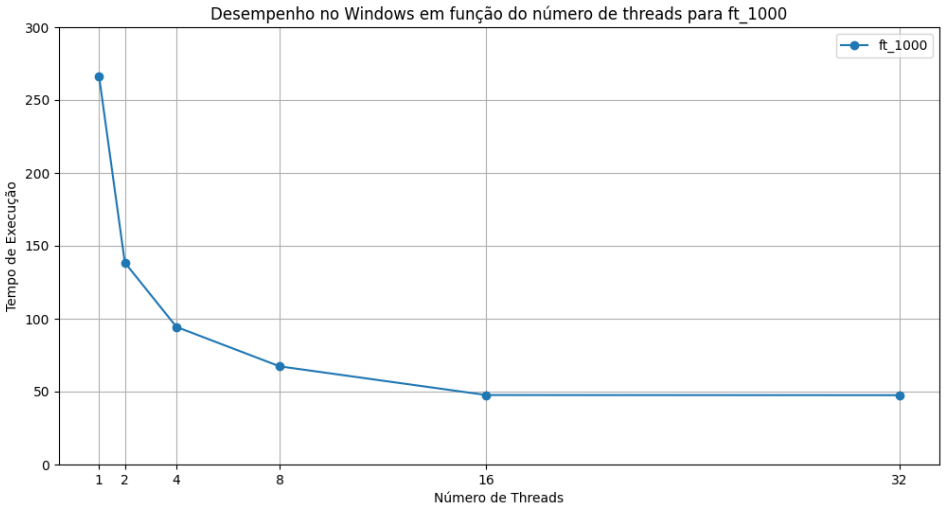

Figura 8: Performance Windows ficheiro 1000 x 10000

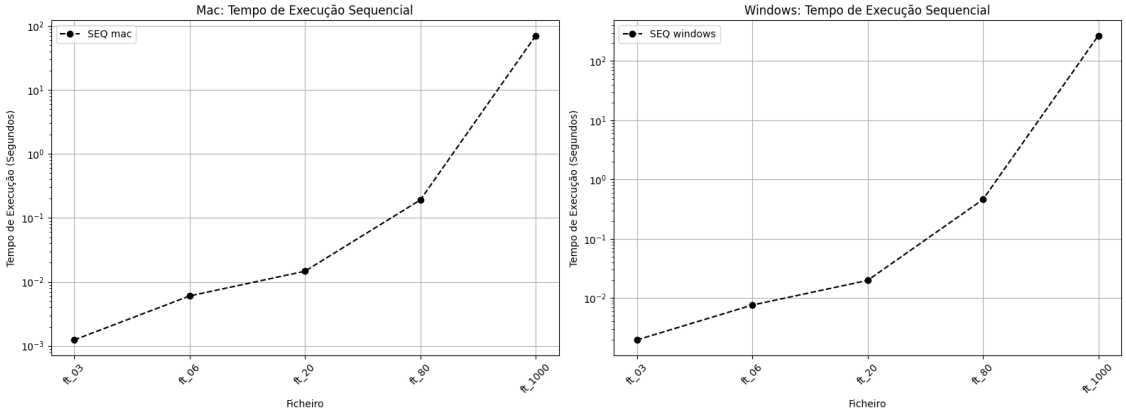

Figura 9: Performance sequencial

7.3 Evolução do Speedup (S) em função do número de threads (P)

Implementou-se um programa em Python para calcular o speedup de cada máquina/ficheiro/número de threads. Para efetuar o processamento, criou-se um dataframe com a seguinte estrutura e dados.

	File	Computer	Speedup_PC1	Speedup_PC2	Speedup_PC4	Speedup_PC8	Speedup_PC16	Speedup_PC32
0	ft_03	mac	0.932331	0.810458	2.431373	2.214286	2.695652	2.638298
1	ft_06	mac	1.181996	2.082759	1.521411	3.822785	4.194444	3.896774
2	ft_20	mac	0.976127	1.520661	2.082037	4.826230	4.487805	4.123249
3	ft_80	mac	0.922750	1.773242	3.026924	3.930084	4.005869	3.805655
4	ft_1000	mac	0.997034	1.935551	3.727797	4.566130	4.603113	4.657247
5	ft_03	windows	0.682927	0.924528	1.704348	1.606557	1.342466	1.689655
6	ft_06	windows	0.848214	1.256198	2.261905	4.269663	3.819095	3.486239
7	ft_20	windows	0.758132	1.467407	2.549550	4.762019	5.092545	4.512528
8	ft_80	windows	1.041819	1.943932	3.495055	6.132559	5.312710	7.134493
9	ft_1000	windows	1.003515	1.930727	2.833996	3.972928	5.612231	5.623332

Figura 10: Matriz para speed up

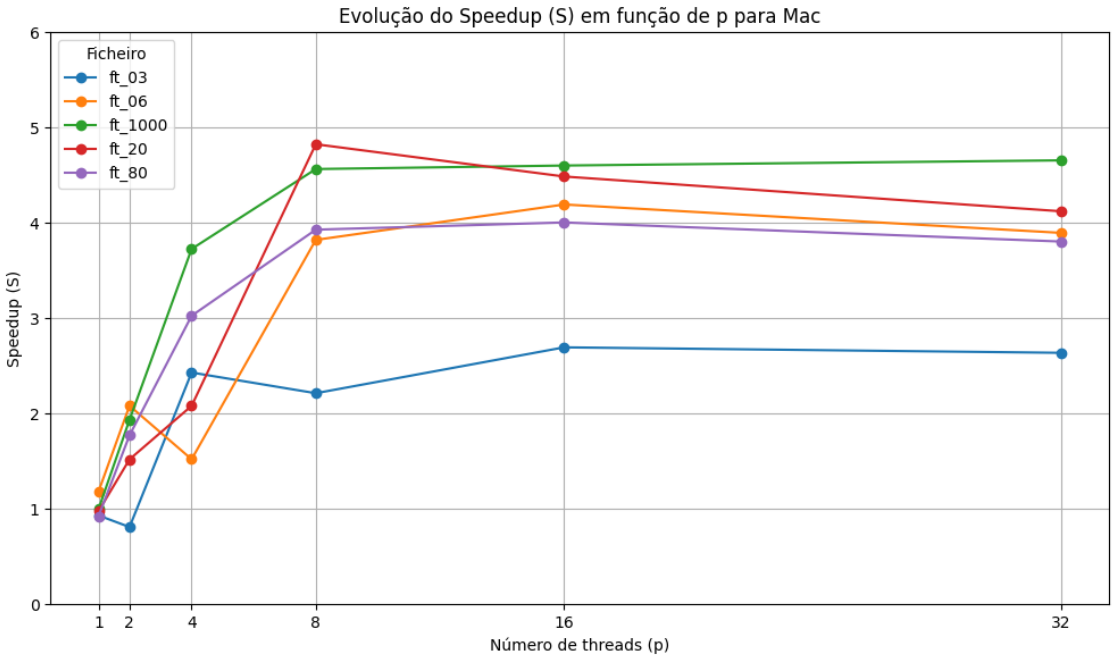

Figura 11: Mac speed up

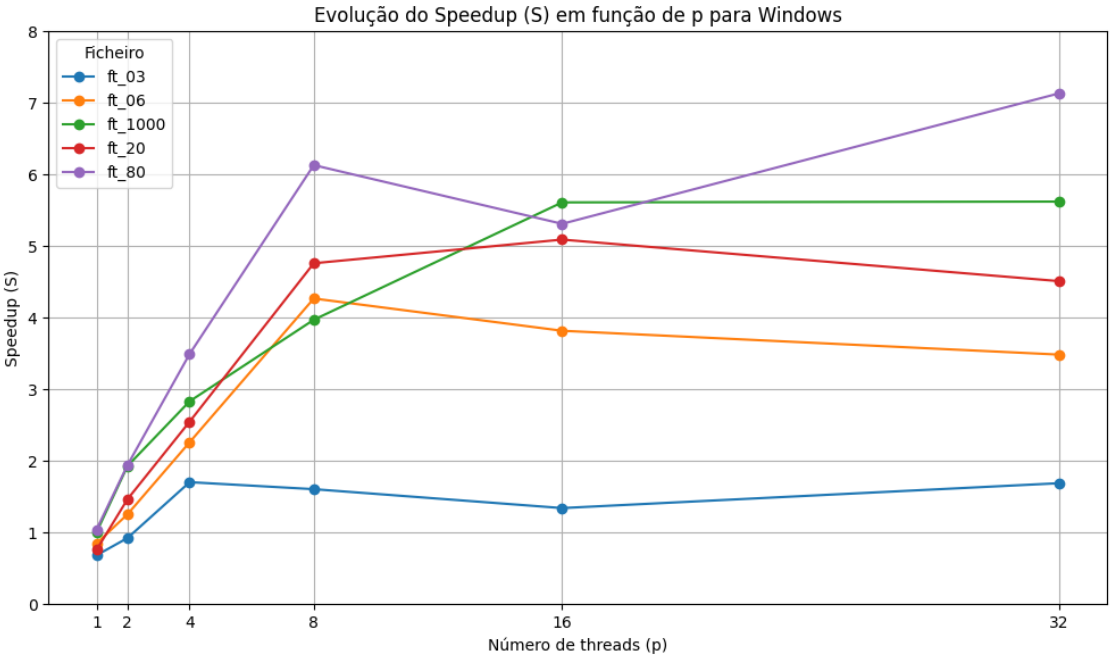

Figura 12: Windows speed up

Com base nos gráficos apresentados, podemos concluir que, com uma única *thread*, o *speedup* é inferior a 1, o que indica que não houve aumento de desempenho comparando o programa sequencial com o paralelo. Além disso, observamos que para mais de 2 *threads*, todos os *speedups* são superiores a 1, o que significa que o desempenho do programa paralelo é superior ao do programa sequencial. Podemos observar que os tempos de execução melhoram aproximadamente para o dobro a cada nova iteração. Por exemplo, com 2 *threads*, o *speedup* do Windows para o ficheiro **ft80** é de 2, e o *speedup* para 4 *threads* é quase 4. Além disso, é possível notar que o *speedup* continua a melhorar continuamente até às oito *threads*, que é o limite de ambos os computadores, atingindo uma estabilidade.

8 Conclusão

Após a realização deste projeto o grupo sentiu que o projeto a ser implementado foi deveras interessante e que contribuiu para expandir conhecimento em várias vertentes. Para além disso, foi possível perceber que houve uma diferença grande nos resultados obtidos pois, o valor do makespan é mais reduzido na versão paralela quando comparado com o valor da versão sequencial.
Porém, numa fase inicial do trabalho, o grupo reparou que o tempo de execução do algoritmo paralelo aumentava em relação ao sequencial, mesmo para ficheiros com vários jobs e várias operações, quando o esperado era precisamente o contrário, isto é, que o paralelismo permitisse uma redução do tempo total de execução, sendo assim mais eficaz. Após consideração, pudemos concluir que isto se devia não só ao facto da complexidade temporal adicionada pela necessidade de criação das threads, mas também porque a complexidade do algoritmo a executar por cada uma delas não ser suficiente para haver uma melhoria significativa no tempo de execução. Essa melhoria de performance só acontece quando a complexidade computacional excede um certo "threshold". Assim que essa complexidade foi adicionada, foi possível verificar que o desempenho melhorou quase diretamente proporcional ao número de threads.
Em suma, a transformação de um algoritmo de job shop sequencial para paralelo demonstrou ser uma estratégia eficaz para melhorar o desempenho, mas requer algum cuidado com a forma que se lida com as secções críticas. A aplicação de técnicas de computação de alto desempenho ao problema de job shop apresenta um campo rico para pesquisa e inovação, com potencial para contribuir significativamente para operações industriais e logísticas.