Relatório do trabalho da disciplina de Computação de Alto Desempenho

Problema do *Job-Shop*

Carlos Bruno Machado Martins -18836

João Ricardo Pinto Azevedo - 18845

Ricardo Jorge Louro Azevedo - 17632

Mestrado em Engenharia Informática

Julho de 2023



Índice

[1. Introdução 4](#_Toc139269314)

[1.1. Descrição do problema 4](#_Toc139269315)

[1.2. Objetivo 5](#_Toc139269316)

[2. Implementação Sequencial 6](#_Toc139269317)

[3. Implementação Paralela 9](#_Toc139269318)

[3.1. Variáveis globais partilhadas 11](#_Toc139269319)

[3.2. Variáveis locais de cada thread 11](#_Toc139269320)

[3.3. Secções críticas 11](#_Toc139269321)

[3.4. Técnicas de exclusão mútua 11](#_Toc139269322)

[4. Análise de desempenho 12](#_Toc139269323)

[5. Discussão de resultados 12](#_Toc139269324)

[Conclusão 12](#_Toc139269325)

[Figura 1 - Estruturadas de dados utilizadas 6](#_Toc139223908)

[Figura 2 - Leitura do número de jobs e máquinas 6](#_Toc139223909)

[Figura 3 - Leitura das operações dos jobs 6](#_Toc139223910)

[Figura 4 - Implementação sequencial 8](#_Toc139223911)

[Figura 5 - Apresentação dos resultados 8](#_Toc139223912)

[Figura 6 - Implementação paralela inicial 9](#_Toc139223913)

[Figura 7 - Definição do número de threads 10](#_Toc139223914)

[Figura 8 - Alteração do ciclo for exterior por omp parallel 10](#_Toc139223915)

# Introdução

Este trabalho prático tem como principal objetivo a utilização das capacidades de programação paralela através do modelo de partilha de memória, para a resolução de um problema de cálculo computacional.

## Descrição do problema

OJob-Shop é um problema que trata a alocação de recursos para a concretização de um trabalho. No problema de Job-Shop existem várias máquinas que conseguem realizar operações. Para se produzir um trabalho (job) é necessário realizar um conjunto de operações, por sequência, em várias máquinas. Como as máquinas não conseguem realizar todas as operações, cada trabalho é uma sequência de operações, em que cada operação tem de ser feita numa máquina específica.

Adicionalmente, porque cada trabalho é distinto dos restantes, a sequência de máquinas de cada trabalho pode não ser a mesma dos restantes trabalhos, i.e., cada trabalho tem a sua própria sequência de máquinas. Contudo, a ordem das máquinas para cada trabalho tem de ser respeitada, i.e., uma operação de um trabalho que deve ser executada numa máquina específica só pode começar quando a operação anterior desse mesmo trabalho já terminou; assim como a operação seguinte do trabalho só pode começar depois da operação atual terminar. Finalmente, realizar uma operação numa máquina demora tempo. Cada operação de cada trabalho demora um tempo específico, que pode ser distinto para cada operação. Uma máquina só consegue fazer uma operação de cada vez, assim, quando começa uma operação, só fica livre para novas operações após esta terminar.

No entanto, identificam-se assim as duas restrições deste problema:

* As operações de cada trabalho têm de ser realizadas por ordem. A segunda operação de um trabalho só pode começar depois da primeira operação do mesmo trabalho estar concluída, e assim sucessivamente para todas as operações seguintes do mesmo trabalho;
* Uma máquina só consegue executar uma operação de cada vez, pelo que havendo duas operações para a mesma máquina, uma só pode começar depois da outra ter terminado.

## Objetivo

O objetivo do problema é apresentar um escalonamento válido, isto é, definir um tempo de início para cada uma das operações (de todos os *jobs*), de modo a cumprir as restrições apresentadas anteriormente como, por exemplo, uma operação só poder começar após a anterior ter terminado.

# Implementação Sequencial

A abordagem escolhida para atribuição de tempos de início das operações passou pela utilização de um conjunto de ciclos *for* e utilização de *structs.* As *structs* utilizadas podem ser visualizadas no excerto de figura 1.



Figura 1 - Estruturadas de dados utilizadas

De modo a receber o número de *jobs* e de máquinas de um ficheiro de *input* recebido no primeiro argumento, após a invocação do script, foi desenvolvido o excerto de código apresentado na figura 2.



Figura 2 - Leitura do número de jobs e máquinas

Após a leitura do número de jobs e de máquinas, era necessário, também, ler as operações que compunham os jobs. Para tal foi desenvolvido o excerto de código apresentado abaixo, na figura 3.



Figura 3 - Leitura das operações dos jobs

Como referido anteriormente, foram criadas estruturas de dados para as máquinas, operações e conjuntos de operações (*jobs)*. Posto isto, foram inicializadas todas as máquinas como não-ocupadas e atribuído o seu identificador, utilizando um ciclo *for*.

Seguidamente, foi utilizado um *nested for* com três outros ciclos *for* dentro:

* O exterior iterava sobre as máquinas, que correspondem às linhas do ficheiro.
* O primeiro interior ciclo *for* é responsável pela gestão das operações. Começa por verificar se a máquina pretendida para a realização da operação atual (correspondente à iteração), está ocupada:
* Caso não esteja, o tempo inicial do *job* é definido com o tempo máximo e o tempo de finalização da operação é equivalente ao tempo de início da operação atual mais o tempo necessário para finalizar a operação atual. Por fim, a máquina responsável pela operação é marcada como ocupada e o tempo até ao qual a máquina irá estar ocupada é equivalente ao tempo de finalização referido anteriormente.
* Caso esteja, o tempo inicial da operação é equivalente ao tempo até ao qual a máquina estaria ocupada e o tempo final da operação seria equivalente ao tempo inicial da operação mais a respetiva duração. Por fim, a máquina é marcada como ocupada e o tempo até ao qual a máquina estaria ocupada é igual ao tempo de finalização da operação.
* O segundo ciclo *for* interior é responsável pelo *reset* das propriedades das máquinas, isto é, quando as operações de um certo *job* já foram definidas o estado das máquinas passa a livre e o tempo de finalização das mesmas também.
* O terceiro ciclo *for* interior é responsável por atualizar a variável que guarda o tempo máximo de finalização da operação. Esta variável é utilizada para definir o tempo de início de uma operação quando a máquina está livre.

No excerto de código da figura 4, apresentado abaixo, está presente o código da abordagem descrita anteriormente.



Figura 4 - Implementação sequencial

Por fim, de modo a apresentar os resultados foi realizado um mero ciclo *for* que percorria as estruturas pretendidas e apresentava os seus valores. Além disso, como era requisito a criação de um ficheiro de *output* com os resultados, o segundo argumento após a chamada do *script* é o nome do ficheiro onde queremos guardar o resultado.



Figura 5 - Apresentação dos resultados

# Implementação Paralela

De modo a realizar a paralelização do problema foi utilizada a biblioteca *OpenMP* tendo em conta que a adaptação de aplicações sequenciais é relativamente fácil utilizando a mesma.

Durante o processo de conversão do problema de sequencial para paralelo tivemos alguns problemas, nomeadamente, de condição de corrida. Para combater os mesmos, utilizamoso *pragma critical* e o *pragma barrier*, mas nem assim foi possível resolver a condição de corrida.



Figura 6 - Implementação paralela inicial

De modo a dar a volta a este problema foi decidido que, por defeito, cada *thread* será responsável por um *job*. Caso o utilizador defina um número de *threads* em específico, esse mesmo número será utilizado. Para tal, deve simplesmente introduzir o número de *threads* no terceiro argumento aquando da invocação da *script*.



Figura 7 - Definição do número de threads

Relativamente à *thread* responsável por *job*, foi definida uma secção paralela em que o compilador tem conhecimento que o bloco de código subsequente deve ser executado em paralelo com as várias *threads*. Para definição desta secção foi tirado proveito do *pragma omp parallel* com especificação do número de *threads* a utilizar.

Tendo então, a secção paralela criada, procedemos com a utilização da função omp\_get\_thread\_num(), que retorna o identificador da *thread* atual. Posto isto, foi criada uma variável que guarda este identificador que vai atuar como o número do *job.*

De seguida, mantivemos apenas o ciclo *for* que percorre todas as máquinas existentes e o que percorre os *jobs* foi substituído pelo código da figura 8, devido ao problema exposto anteriormente.



Figura 8 - Alteração do ciclo for exterior por omp parallel

Findada a execução do ciclo *for* que percorre as máquinas, é então invocada uma barreira, utilizando o *pragma omp barrier*, de modo a sincronizar as *threads*. De seguida, é realizado o *reset* ao estado das máquinas e é calculado o tempo máximo de finalização das operações. Tendo em conta que foi removido um ciclo *for* que percorria os *jobs*, este processo também sofreu uma alteração mínima. Em vez de ter dois ciclos *for*, passou a ter apenas um e recebe o valor da *thread* atual no seu lugar.



Código 8 - Cálculo do tempo máximo de finalização da operaçao - paralelo

## Variáveis globais partilhadas

Ao longo do código implementado na componente paralela, podemos identificar um conjunto de variáveis globais partilhadas, sendo elas:

* ***Machines*** – Representa uma estrutura que armazena as informações sobre as máquinas como o id, estado (ocupada ou liberta) e o tempo que demorou a processar a informação;
* ***InitialTime*** - Representa uma matriz bidimensional que armazena os tempos iniciais das operações para cada job e máquina;
* ***EndTime*** - Representa numa matriz bidimensional os tempos de finalização das operações para cada job e máquina;
* ***MaxEndTime*** - Uma variável inteira que armazena o tempo máximo de término entre todas as operações.

## Variáveis locais de cada thread

Ao longo do código implementado na componente paralela, podemos identificar uma variável local de cada thread, sendo ela:

* ***jobThread***– Representa uma variável local que armazena o índice da linha (job) atribuída a thread.

## Secções críticas

A secção crítica acontece quando uma thread verifica a condição se a máquina está ocupada e atualiza os tempos iniciais e de finalização. Esta verificação e atualização terá de ser feita com precaução para evitar condições de corrida. O código *#pragma omp critical* é usado para criar uma seção crítica ao bloco de código que realiza operações de leitura e escrita. As variáveis envolvidas nessa seção são:

* ***machines[jobs[row].operations[m].machineId].isWorking:*** a variável *isWorking* da estrutura *Machine* indica se uma máquina está ocupada ou livre.
* ***machines[jobs[row].operations[m].machineId].endTime:*** a variável *endTime* da estrutura *Machine* que armazena o tempo que levou a realizar a última operação na máquina.

## Técnicas de exclusão mútua

A técnica de exclusão mútua utilizada para impedir que múltiplas *threads* executem secções críticas em simultâneo esta salvaguarda através do código *#pragma omp critical*. Este garante que apenas uma *thread* de cada vez pode realizar operações de leitura ou escrita nas variáveis. Isto evita condições de corrida e garante que os tempos finais e iniciais sejam atualizados corretamente.

# Análise de desempenho

No âmbito da análise de desempenho do algoritmo desenvolvido foram realizados um conjunto de testes tanto para a versão sequencial e para a versão paralela. Cada teste foi realizado sete vezes e o valor final apresentado para o teste é a média de duração de cada um dos mesmos.

Na tabela 1, apresentada abaixo, é possível verificar os tempos de cada iteração, juntamente com a duração média de cada teste. Para efeitos de compreensão, P1 representa uma execução da implementação paralela com 1 *thread*.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Threads\**  **Tempo(s)** | **Iteração 1** | **Iteração 2** | **Iteração 3** | **Iteração 4** | **Iteração 5** | **Iteração 6** | **Iteração 7** | **Média** |
| **Sequencial** | 0.0140 | 0.0120 | 0.0170 | 0.0170 | 0.0160 | 0.0090 | 0.0170 | 0.0160 |
| **P1** | 0.0070 | 0.0030 | 0.0020 | 0.0110 | 0.0150 | 0.0030 | 0.0020 | 0.0030 |
| **P2** | 0.0050 | 0.0100 | 0.0080 | 0.0100 | 0.0070 | 0.0080 | 0.0060 | 0.0080 |
| **P3** | 0.0150 | 0.0060 | 0.0150 | 0.0140 | 0.0110 | 0.0040 | 0.0170 | 0.0140 |
| **P4** | 0.0090 | 0.0020 | 0.0030 | 0.0080 | 0.0070 | 0.0030 | 0.0060 | 0.0060 |
| **P5** | 0.0170 | 0.0140 | 0.0100 | 0.0100 | 0.0030 | 0.0030 | 0.0040 | 0.0100 |
| **P6** | 0.0030 | 0.0120 | 0.0040 | 0.0150 | 0.0150 | 0.0030 | 0.0030 | 0.0040 |

Tabela 1 - Testes realizados ao algoritmo

Como é possível, verificar pela tabela 1 as execuções existem algumas médias que levantam algumas suspeitas como é o caso de, por exemplo, o primeiro teste (P1). Isto acontece pelo simples facto de termos definido que cada *thread* seria responsável pelo escalonamento de um *job* e, sendo que temos apenas uma *thread* o algoritmo trata, apenas, do escalonamento de um *job*.

Tabela 2 - Análise de desempenho das execuções

## Cálculo do *speedup*

# Discussão de resultados

A abordagem apresentada anteriormente tem alguns problemas nomeadamente o não respeito da ordem das tarefas, isto é, não há nada que garanta que a ordem de execução das operações vá ser respeitada. Embora a leitura e procura no ficheiro de input seja feita ordenadamente e a ordem, à partida, esteja garantida, este pode ser um problema a ter em consideração.

# Conclusão