Programação Concorrente – 22/23
Relatório – Projeto – Parte B

Esquema do relatório

Grupo 11

Rúben Miguel Oliveira Tadeia | Nº 75268

Bernardo Silva Costa | Nº 102777

1 Objetivos do trabalho

O objectivo deste projecto, consiste na implementação de uma aplicação que recebe uma diretoria e o número de threads como parâmetro de entrada, de modo a realizar certas transformações em imagens. Transformações essas que podem ser uma marca de água, um novo tamanho, ou um thumbnail. De notar, que o número de threads tem que ser um número inteiro positivo e que se for requisitada apenas uma thread, o resultado será semelhante ao sequencial. A grande diferença entre esta parte do trabalho relativamente à parte A, reside na utilização de Pipes e também, consoante o caso, de mutex. Sobre estas pipes, é de notar que é nelas que é escrito o nome/referência das imagens a tratar, sendo apenas lida essa informação na função das threads.

2 Funcionalidades implementadas

Tabela 1: *Funcionalidades implementadas*

	Não implementada	Com falhas	Totalmente correta
ap-paralelo-3			
Argumentos da linha de comando			✅
Leitura do ficheiro **image-list.txt**			✅
Criação de *threads*			✅
Criação de canais			✅
Envio ficheiros para 1 Estágio			✅
Envio ficheiros para 2 Estágio			✅
Envio ficheiros para 3 Estágio			✅
Reutilização da Watermark			✅
Terminação das Threads			✅
Verificação das imagens existentes			✅
Produção resultados			✅

2.1 Descrição das funcionalidades com falhas

Todas as funcionalidades foram corretamente implementadas, testadas e comparadas com outras versões de código de modo a ser apresentada a melhor solução possível. De notar que todos os testes com valgrind passaram sem erros.

3 Organização do código e estrutura de dados

Antes de elaborarmos sobre as funções que foram adicionadas para a realização deste projecto, vamos ilustrar a nossa estrutura utilizada para as threads, tendo em nota que todas as threads recebem uma estrutura deste tipo como parâmetro de entrada.

Imagem 1: *Estrutura de dados enviada para as threads*

```
// Structure
typedef struct thread_input_info_struct{
    char * image_folder;
    int thread_id;
} thread_input_info;
```

Entrando na explicação de cada parâmetro para as threads, existem 2 variáveis dentro da estrutura **thread_input_info_struct**. Temos um array de caracteres chamado **image_folder** que contém o caminho (pasta) para as imagens que vão ser tratadas pelas threads. Temos também, um inteiro, **thread_id**, que indica o número atribuído a cada thread, que ajuda não só no desenvolvimento como no debug. Porque facilmente identificamos o trabalho que cada thread tem que realizar pelo seu **id**.

Relativamente aos caminhos das directorias, foram definidas as 3 variáveis que se encontram abaixo.

Imagem 2: *Variáveis com as directorias de output*

```
// the directories where output files will be placed
#define RESIZE_DIR "/Resize-dir/"
#define THUMB_DIR "/Thumbnail-dir/"
#define WATER_DIR "/Watermark-dir/"
```

Entrando agora na parte da explicação das funções implementadas é necessário compreender que o código foi estruturado de modo a ser facilmente compreendido e reutilizado, caso necessário. Dito isto, o código encontra-se dividido no **main** file e em **3 ficheiros.c** cada um com suas respectivas **bibliotecas.h**:

- main
- functions
- image-lib
- threads

As funções dos ficheiros **functions**, são funções gerais ao funcionamento do código para a aplicação **ap-paralelo-3**. Apresentamos agora uma breve descrição das mesmas:

Tabela 2: *Funções implementadas dos ficheiros* ***functions*** *e sua descrição*

Cabeçalho da Função	Breve Descrição
`void check_arguments (int argc, char * argv[]);`	Verifica se os argumentos estão na forma correta. Caso exista algum problema, retorna o erro e termina o programa.
`void read_image_file(char * imagesDirectory, char * fname);`	Lê o ficheiro que contém o nome das imagens a serem tratadas. Retira o número de imagens válidas de um array que contém o nome dessas mesmas imagens.
`void print_image_array(char ** images_array, int array_size);`	Escreve no terminal o conteúdo do images_array (Para testes apenas)
`void free_image_array(char ** images_array, int array_size);`	Dá free ao images_array, que é um vector de nomes de imagens
`int check_images(char * image_string);`	Verifica se as imagens existem na pasta fornecida
`void create_directories(char * images_folder);`	Cria as 3 respectivas diretorias onde serão adicionadas as novas imagens

Em seguida, o ficheiro a explicar será o **image-lib**, que contém em grande parte funções que já tinham sido criadas pelo professor. No entanto, de modo a impedir replicação desnecessária de código foram criadas funções que ajudam no tratamento de imagens.

Para a aplicação ap-paralelo-3 foram criadas 3 funções e estas podem ser observadas na tabela seguinte.

Tabela 3: *Funções implementadas dos ficheiros* ***image-lib*** *para* ***ap-paralelo-3*** *e sua descrição*

Cabeçalho da Função	Breve Descrição
`void add_watermark_in_image(char * fileName, char * images_folder);`	Adiciona watermark a uma imagem sabendo o nome e localização do ficheiro
`void add_resize_to_image(char * fileName, char * images_folder);`	Adiciona resize a uma imagem que já tem watermark, sabendo o nome e localização do ficheiro
`void add_thumbnail_to_image(char * fileName, char * images_folder);`	Adiciona thumbnail a uma imagem que já tem watermark, sabendo o nome e localização do ficheiro

Por último, só falta explicar as novas funções implementadas para a paralelização nos ficheiros **thread**.

Para a aplicação ap-paralelo-3 foram ainda adicionadas as seguintes funções:

Tabela 4: *Funções implementadas dos ficheiros* ***thread*** *para* ***ap-paralelo-3*** *e sua descrição*

Cabeçalho da Função	Breve Descrição
`void * thread_function_wm(void * arg);`	Aplica watermark numa imagem e guarda o resultado da transformação.
`void * thread_function_rs(void * arg);`	Aplica resize numa imagem que tenha recebido watermark e guarda o resultado da transformação
`void * thread_function_tn(void * arg);`	Aplica thumbnail numa imagem que tenha recebido watermark e guarda o resultado da transformação

Uma nota particular relativa ao **nome de funções desenvolvidas** para a aplicação **ap-paralelo-3**, embora tenham o mesmo nome das funções utilizadas na aplicação **ap-paralelo-2**, estas novas efetuam a leitura das imagens a processar através de uma pipe em cada função.

4 Descrição do pipeline

Na aplicação **ap-paralelo-3** são utilizadas as pipes, como forma de passar informações para as threads. Isto funciona, porque cada thread está a ler de uma pipe para o respectivo estágio, e apenas são escritas nessa pipe, as imagens que podem iniciar o processamento nesse estágio.

4.1 Envio de informação para as *threads*

Nesta aplicação **ap-paralelo-3**, foram criadas **3 pipes**, uma para estágio do programa. A razão para o número de pipes escolhido, deve-se a querermos uma relação de uma pipe por estágio, o que torna mais fácil para as threads saberem que se lerem de uma certa pipe têm uma certa tarefa para executar. Isto foi possível, porque colocou-se uma pipe diferença em cada uma das funções das threads.

Para cada pipe é apenas enviado um inteiro que se refere ao índice de um vetor de caracteres, que contém o nome de cada imagem a processar.

Imagem 3: *Pipes utilizadas na aplicação*

```
// Pipes
int pipe_watermark[2]; // Pipe onde se vai escrever e ler para watermark
int pipe_resize[2]; // Pipe onde se vai escrever e ler para resize
int pipe_thumbnail[2]; // Pipe onde se vai escrever e ler para thumbnail
```

4.2 Variáveis globais

Abaixo encontram-se ilustradas as nossas variáveis globais.

Imagem 4: *Pipes utilizadas na aplicação*

```
// Variaveis Globais Gerais
int max_word_len = 0; // Tamanho maximo por palavra
char ** images_array; // Array com o nome das imagens
int numero_imagens_validas = 0; // Contem o numero de nomes validos de imagens

// Pipes
int pipe_watermark[2]; // Pipe onde se vai escrever e ler para watermark
int pipe_resize[2]; // Pipe onde se vai escrever e ler para resize
int pipe_thumbnail[2]; // Pipe onde se vai escrever e ler para thumbnail

// Variaveis para saída das threads
int numero_imagens_processadas_watermark = 0; // Numero de imagens processadas - watermark
int numero_imagens_processadas_resize = 0; // Numero de imagens processadas - resize
int numero_imagens_processadas_thumbnail = 0; // Numero de imagens processadas- thumbnail

// Mutex
pthread_mutex_t watermark_mutex;
pthread_mutex_t resize_mutex;
pthread_mutex_t thumbnail_mutex;
```

Existem três variáveis globais gerais: **max_word_len**, que é um inteiro que contém o tamanho máximo do nome de uma imagem, **images_array,** que é um vetor de caracteres que tem em cada entrada, o nome de uma imagem. Por fim, temos o **numero_imagens_validas**, que como o nome sugere contém o número de imagens válidas. No entanto, apenas 2 destas variáveis são acedidas pelas threads, o **images_array** e o **numero_imagens_validas**. A razão pela qual, as threads precisam da primeira variável, é porque, é passado para dentro das pipes, um inteiro que contém um index do images_array, e esse contém o nome da imagem a processar. A segunda variável **numero_imagens_validas**, é passada para as threads, porque a condição de saída das threads é realizada se o **numero_imagens_validas** for igual ao número **nome_do_estágio**. Para finalizar, foram criadas 3 pipes, para ficar cada uma associada a cada um dos 3 estágios.

Em relação aos mutexes, foram criados 3, um para cada estágio da aplicação **ap-paralelo-3**, que têm como funcionalidade garantir que apenas uma thread está a ler e/ou a escrever numa variável comum. Essas variáveis comuns são a **numero_imagens_processadas_(nome_do_estágio)**, existindo portanto 3 variáveis, como se consegue ver pela imagem 4, e têm como funcionalidade garantir que as threads sabem quantas imagens foram processadas para que fácilmente consigam encerrar a pipeline e terminar o programa graciosamente.

4.3 Encerramento do pipeline

Como explicado em capítulos anteriores do relatório, existem 3 variáveis espalhadas pelas threads **numero_imagens_processadas_(nome_do_estágio)** (uma por cada estágio). Esta variável é inicializada a zero e incrementada sempre que uma imagem é processada, quer seja por watermark, quer seja por resize, quer seja por thumbnail. Em cada thread, é comparada esta variável anterior, com o **numero_imagens_validas**, assim quando estes valores foram iguais, atingimos a condição de saída e a thread termina.

4.4 Acesso ao resultado do estágio 1 (Watermark)

Para as pipes é passado única e exclusivamente um inteiro, que contém um index do images_array, e esse contém o nome da imagem a processar. Assim que cada imagem é processada, é escrito para a pipe o mesmo index (número) para ambas as pipes de resize e de thumbnail, como se consegue ver pela **imagem 5** abaixo.

Imagem 5: *Acesso ao resultado do estágio de watermark pelas outras pipes*

```
read(pipe_watermark[0], &image_array_index, sizeof(int));

add_watermark_in_image(images_array[image_array_index],thread_argument->image_folder);

write(pipe_resize[1], &image_array_index, sizeof(int));
write(pipe_thumbnail[1], &image_array_index, sizeof(int));
```

5 Resultados

5.1 Descrição do ambiente de execução

Nesta seção foi utilizado o comando **less /proc/cpuinfo** para saber o modelo exato do processador. Com essa informação foi possível determinar todas as informações abaixo indicadas. É de realçar, que foi enviado, em conjunto com o projeto, um ficheiro **.txt** chamado ***pcinfo*** que contém a informação abaixo com mais detalhe. É também de notar que todos os resultados de ambas as aplicações, assim como, a descrição do modelo abaixo, foram obtidos através do computador pessoal:

- **Descrição textual do computador (tipo, marca, localização, ….):** AMD Ryzen 7 5700U with Radeon Graphics
- **Número de Cores/CPUs:** 8
- **Velocidade do processador:** 1.80 GHz
- **Sistema operativo:** Ubuntu 20.04.5 LTS

5.2 Throughput ap-paralelo-3

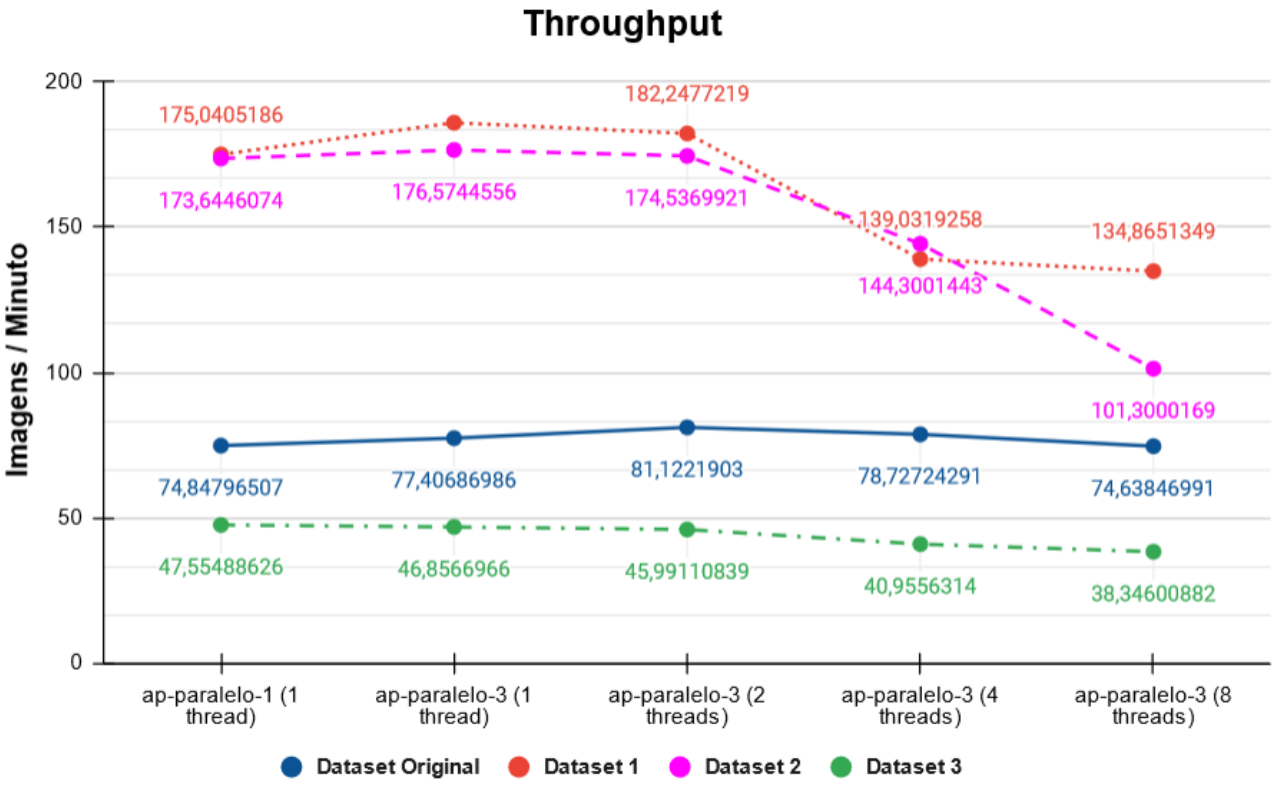
Os tempos de execução do **ap-paralelo-3** para os diversos *datasets* encontram-se na tabela abaixo. Note-se que foram obtidos todos os tempos de execução também para o dataset original.

Tabela 5: *Tempos de processamento* ***ap-paralelo-3*** *(segundos)*

	N. de imagens	***ap-paralelo-1 1 thread***	**ap-paralelo-3**			
			1	2	4	8
Dataset original	8	6.413	6.201	5.917	6.097	6.431
Dataset 1	9	3.085	2.904	2.963	3.884	4.004
Dataset 2	30	10.366	10.194	10.313	12.474	17.769
Dataset 3	10	12.617	12.805	13.046	14.650	15.647

Abaixo encontra-se um gráfico com os *throughputs* calculados para o programa **ap-paralelo-3** para diversos *datasets*.

Imagem 6: *Valores dos throughputs* calculados para o programa **ap-paralelo-3** com diversos *datasets*



Após analisarmos o gráfico anterior, podemos dizer que os resultados estão de acordo com o esperado. Note-se para nós já faz sentido o valor do throughput começar a diminuir a partir do **ap-paralelo-3 (4 threads)**, isto porque, na realidade não estamos a criar 4 threads, mas sim 3 vezes 4 threads, ou seja 4 threads por estágio, o que nos dá um total de **12 threads**. Assim, para o mesmo valor do número de cores do nosso computador (8 cores), o aumento das threads só vai ajudar até um certo limite, que pode ser observado no gráfico. Esse limite pode ser explicado, porque o computador não tem cores disponíveis para fazer as threads em paralelo, portanto, estas vão ter que ficar em stand by, a aguardar que as outras threads terminarem para que o processador consiga “pegar nelas”. Ora a somar a este tempo de espera, temos também o tempo de criação e fecho de threads que podem nem ter chegado a fazer nada. Com isto, consegue-se concluir que o throughput iria diminuir com o aumento do número de threads.

Relativamente ao aumento do throughput ser diferente para os diversos datasets, tal pode ser explicado, porque existem imagens maiores que outras, que levam muito tempo a realizar o watermark e pouco a realizar o resize e thumbnail. No entanto, este aumento do tempo de realização do watermark é suficiente para provocar uma diminuição no valor de imagens por minuto.

6 Otimizações

Relativamente ao tema das otimizações, acreditamos que o nosso código ficou num estado bastante robusto. Quer isto dizer que, embora possa não ser a melhor solução para todos os casos, é sem dúvida uma óptima solução para a maioria deles.

Passando agora a explicar, foram feitos os dois testes: realizar a thumbnail só após a resize ou logo após watermark. Verificou-se que o melhor resultado depende do número de cores do computador assim como do número de threads fornecidos. Isto é, se for um computador recente e fornecermos um número baixo de threads por estágio (~3), então é mais rápido fazer o resize (estágio 2) e o thumbnail (estágio 3) logo após o watermark (estágio 1). No entanto, caso o teste seja na mesma num computador recente mas destas vez se forneça um número alto de threads por estágio (10+), então o resultado será pior, mais lento, se fizermos o resize (estágio 2) e o thumbnail (estágio 3) logo após o watermark (estágio 1). Isto deve-se ao facto de termos todos os cores do processador ocupados, não adiantando nada criar novas threads, porque não existem cores disponíveis que as possam utilizar. Daí estas threads terem que ficar em stand by à espera da sua oportunidade de executar.

Outro aspeto que podia ter sido optimizado no nosso código, seria a não utilização de mutex. Isto porque, a sua utilização para a resolução do projecto não é necessária e aumenta o tempo de execução da aplicação, dado que o mutex bloqueia uma zona do código para que apenas uma thread consiga ler ou escrever o resultado. Assim sendo e de modo a explicar a nossa escolha final dos mutex, podemos dizer que testamos ambas as soluções com sucesso: com mutex e sem mutex, solução na qual era escrito um número default na pipe, para que as threads identificassem a sua condição de paragem.

Em suma, a solução dos mutex acabou por se revelar não só a solução mais compacta e elegante, como uma solução que coloca em prática todos os conhecimentos obtidos nas aulas teóricas.

7 Conclusão

O desenvolvimento deste trabalho permitiu expandir o conhecimento obtido até agora sobre as threads, assim como, a controlar a maneira destas obterem informações de pipes. Para além disto, a melhor aprendizagem que conseguimos levar deste projecto são as múltiplas opções para resolver o problema da “race condition”, isto é, de ter várias threads a acederem e a modificarem um recurso comum em paralelo. As duas melhores opções que resolvem este problema e que foram testadas neste projecto foram:

- A escrita de valores default para a pipe de modo a que as threads saibam a sua condição de saída (sem utilização de mutex). Neste caso, como cada valor ao ser lido pela pipe, sai da pipe, temos a certeza que cada thread só lê um nome/referência de imagem de cada vez.
- A utilização de mutex. Assim, conseguimos garantir que em cada altura de leitura ou de escrita de uma zona crítica (zona que é acedida por mais do que uma thread), temos apenas uma thread a fazer essa tarefa, seja ela ler ou escrever, enquanto essa variável fica bloqueada para esse uso.

Deste modo, terminamos a nossa parte experimental da disciplina que contribuiu bastante para a consolidação dos conhecimentos adquiridos.