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

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 duas aplicações que recebem imagens como parâmetro de entrada e manipulam essas imagens de modo a criar novas imagens com uma marca de água, um novo tamanho e por fim um thumbnail. As duas grandes diferenças entre estas duas aplicações são, o número possível de threads que podemos utilizar e a quantidade de imagens que cada thread recebe. Enquanto que na primeira aplicação podemos utilizar um número inteiro positivo de threads e cada thread só recebe um conjunto de imagens, na segunda aplicação todas as threads recebem o conjunto total das imagens válidas e são apenas possível de utilizar 3 threads.

2 Funcionalidades implementadas

Tabela 1: *Funcionalidades implementadas*

	Não implementada	Com falhas	Totalmente correta
ap-paralelo-1			
Argumentos da linha de comando			✅
Leitura do ficheiro **image-list.txt**			✅
Criação de *threads*			✅
Distribuição trabalho pelas *threads*			✅
Verificação ficheiro existentes			✅
Produção resultados			✅
ap-paralelo-2			
Argumentos da linha de comando			✅
Leitura do ficheiro **image-list.txt**			✅
Criação de *threads*			✅
Distribuição trabalho pelas *threads*			✅

Verificação ficheiro existentes			✅
Produção resultados			✅

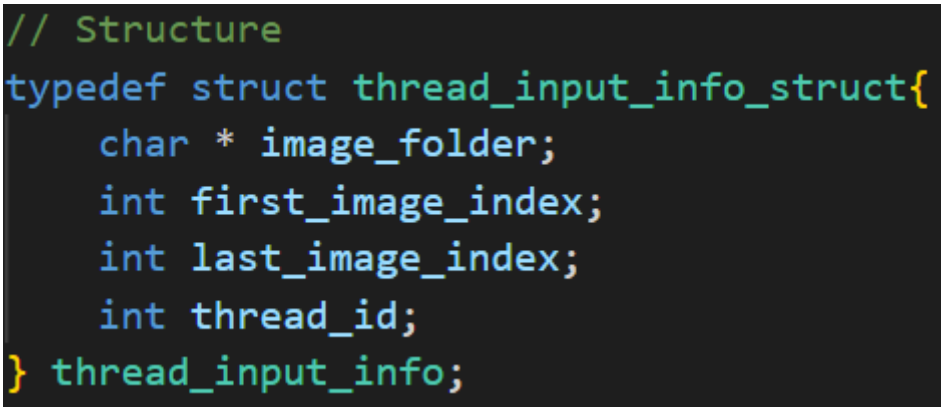
2.1 Descrição das funcionalidades com falhas

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 first_image_index;
    int last_image_index;
    int thread_id;
} thread_input_info;
```

Entrando na explicação de cada parâmetro para as threads, existem 4 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. De seguida, temos dois inteiros, o **first_image_index** e o **last_image_index** que correspondem ao primeiro e último índice, respetivamente, de um array de imagens que cada thread vai processar. A última variável corresponde a 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 em ambas as aplicações (ap-paralelo-1 e ap-paralelo-2). 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 que images_array contém (Para testes apenas)
`void free_image_array(char ** images_array, int array_size);`	Dá free ao images_array
`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-1 foram criadas as seguintes funções:

Tabela 3: *Funções implementadas dos ficheiros* ***image-lib*** *para* ***ap-paralelo-1*** *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

Para a aplicação ap-paralelo-2 foram ainda adicionadas as seguintes funções:

Tabela 4: *Funções implementadas dos ficheiros* ***image-lib*** *para* ***ap-paralelo-2*** *e sua descrição*

Cabeçalho da Função	Breve Descrição
`void add_resize_to_image_with_wm(char * fileName, char * images_folder);`	Adiciona resize numa imagem, sabendo o nome de entrada do ficheiro, no entanto antes aplica watermark no mesmo ficheiro embora só guarde o ficheiro no final da execução.
`void add_thumbnail_to_image_with_wm(char * fileName, char * images_folder);`	Adiciona thumbnail numa imagem, sabendo o nome de entrada do ficheiro, no entanto antes aplica watermark no mesmo ficheiro embora só guarde o ficheiro no final da execução.

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

Para a aplicação ap-paralelo-1 foram criadas as seguintes funções:

Tabela 5: *Funções implementadas dos ficheiros* ***thread*** *para* ***ap-paralelo-1*** *e sua descrição*

Cabeçalho da Função	Breve Descrição
`int get_images_threads_difference(int number_images, int number_threads);`	Compara a relação/diferença que existe entre o número de imagens e o número de threads. Devolvendo **1** se o número de threads for igual ao número de imagens, **2** se o número de threads menor que o número de imagens e **3** se o número de threads for maior que o número de imagens.
`void * thread_function_wm_tn_rs(void * arg);`	Adiciona watermark, resize e thumbnail a uma imagem, gravando sempre o resultado final após cada transformação.

Para a aplicação ap-paralelo-2 foram ainda adicionadas as seguintes funções:

Tabela 6: *Funções implementadas dos ficheiros* ***thread*** *para* ***ap-paralelo-2*** *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_rs_with_wm(void * arg);`	Aplica resize numa imagem, mas antes aplica watermark, guardando apenas o resultado da transformação do resize
`void * thread_function_tn(void * arg);`	Aplica thumbnail numa imagem que tenha recebido watermark e guarda o resultado da transformação
`void * thread_function_tn_with_wm(void *`	Aplica thumbnail numa imagem, mas

`arg);`	antes aplica watermark, guardando apenas o resultado da transformação do thumbnail

Uma nota particular relativa ao **número de funções desenvolvidas** para a aplicação **ap-paralelo-2**, reflete-se no facto de termos feito vários testes e analisado qual o melhor resultado. Por exemplo, dado que foi utilizado um processador com 4 cores, é muito mais rápido correr 3 threads ao mesmo tempo, em que uma faz watermark e guarda o resultado, e outras 2 fazem **thumbnail/resize** com watermark sem gravarem a imagem de watermark, ao invés de ter a thread 1 a gerar os watermarks com a thread 2 e 3 à espera que thread 1 termine para fazerem o thumbnail e resize a partir dos ficheiros watermark gerados.

4 Partição do trabalho (ap-paralelo-1)

Na aplicação **ap-paralelo-1** é utilizada uma função *get_images_threads_difference* que recebe o número de imagens e o número de threads e faz a divisão do número de imagens pelo número de threads. Parte da lógica encontra-se ilustrada na imagem abaixo. No entanto, iremos explicar todo o funcionamento no próximo subcapítulo.

Imagem 3: *Lógica utilizada no main para identificar quantas imagens faz cada thread*

```
// Main Switch case for the project
switch (get_images_threads_difference(numero_imagens_validas,n_threads)) {
        case 1:
            // Numero de Threads igual ao numero de imagens
            printf("DEBUG: Numero de Threads igual ao numero de imagens\n");
            numero_imagens_por_thread = 1;
            break;
        case 2:
            // Numero de Threads menor que o numero de imagens
            printf("DEBUG: Numero de Threads menor que o numero de imagens\n");
            numero_imagens_por_thread = numero_imagens_validas / n_threads;
            extra_images = numero_imagens_validas % n_threads;
            break;
        case 3:
            // Numero de Threads maior que o numero de imagens
            printf("DEBUG: Numero de Threads maior que o numero de imagens\n");
            numero_imagens_por_thread = 1;
            break;
        default:
            printf("ERRO: Existiu um problema com a relacao entre as imagens e as threads\n");
            exit(5);
            break;
}
```

4.1 Algoritmo de partição

Como dito acima, o algoritmo de partição inicia-se com uma função que é a *get_images_threads_difference*. Ao sabermos se existem mais ou menos imagens por threads podemos começar a fazer a divisão de imagens por thread. No caso em que temos um **número de threads igual ao número de imagens**, o problema é bastante simples, cada thread recebe uma imagem e faz todo o seu tratamento. Como a nossa estrutura de dados enviada para a thread, contém dois inteiros índices de início e fim de um image_array a ser processado, neste caso o índice de início irá coincidir com o de fim porque só iremos processar uma imagem.

Para o caso em que existem menos threads do que imagens a resolução deste problema já começa a ser interessante. O que é feito neste caso é dizer que cada thread irá realizar à partida um número de imagens igual ao número de imagens dividido pelo número de threads. É ainda necessário colocar o resto da divisão entre o número de imagens válidas e o número de threads noutra variável, de notar que este resultado só pode variar entre **1** e o número de **threads menos 1**. Assim sendo, o que é feito no no loop de criação das threads é perguntar se existem imagens adicionais, ou seja **extra_images** diferente de 0. Se isto for verdade, então aumentamos o índice até onde as threads vão processar o **image_array** em 1 valor e reduzimos o **extra_images** em 1. Como estamos a criar as threads e a passar índices seguidos do **image_array** para trabalharem, vemos que as primeiras threads são as que irão receber o trabalho adicional, se existir.

Exemplo: Supondo que existem 14 imagens e 4 threads, os resultados das variáveis anteriores seriam **numero_imagens_por_thread igual a 3** e **extra_images igual a 2**. O que iria resultar que a thread 1 e 2 iriam fazer 3 + 1 imagens e as threads 3 e 4 iriam apenas realizar 3 imagens.

Para o terceiro caso em que o número de threads é maior que o número de imagens o que se faz é atribuir inicialmente 1 imagem por thread. Para cada iteração do loop de criação de 1 thread, vamos verificar se ainda existem imagens disponíveis. Se não existirem mais imagens disponíveis, é então colocado um índice negativo do image_array no valor a processar pela thread, ao que a thread interpreta por não ter trabalho a realizar.

Tabela 6: *Distribuição de 10 imagens por 3 threads*

	Imagens a ser processadas por cada *thread*			
	1ª imagem da *thread*	**2ª imagem da *thread***	**3ª imagem da *thread***	**4ª imagem da *thread***
Thread_0	Lisboa-1.png	Lisboa-2.png	Lisboa-3.png	Lisboa-4.png
Thread_1	Lisboa-5.png	Lisboa-6.png	Lisboa-7.png	**Sem imagem**
Thread_2	Lisboa-8.png	Lisboa-9.png	Lisboa-10.png	**Sem imagem**

4.2 Envio de informação para as *threads*

Embora neste código desenvolvido existam mais 2 variáveis globais, existe apenas uma variável global a que as threads acedem, que é o **char ** image_array**. Este array contém os nomes de todas as imagens válidas e existentes para realizar as transformações necessárias. A razão pela qual cada thread acede a este image array é para saberem que imagem irão tratar, dado que as threads recebem o índice de início e fim do **image_array** para os quais estão assignadas a função de watermark, resize ou thumbnail. De notar que na parte 2 como todas as threads recebem a lista de todas as imagens e de modo a mantermos o código consistente, é utilizada a mesma estrutura enviada para a threads que na parte 1. No entanto, o índice de início será o 0 e o índice final será o número de imagens válidas menos um.

Como já foi explicado no **capítulo 3** deste relatório (Organização do código e estrutura de dados), cada thread recebe uma função que irá desempenhar como terceiro parâmetro, e um **thread_information** como último parâmetro.

Imagem 4: *Exemplo de phtread_create*

```
pthread_create(&thread_id,          NULL,          thread_function_a_fazer,
thread_information);
```

Toda a informação relativa à utilização do **thread_information** encontra-se já explicada e exemplificada ao longo deste relatório e por essa mesma razão não entraremos novamente no detalhe de explicação da mesma.

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, realçar que foi enviado, em conjunto com o projeto, um ficheiro .txt chamado *pcinfo* que contém a informação abaixo com mais detalhe. De notar que todos os resultados de ambas as aplicações assim como a descrição do modelo abaixo foram obtidos através do computador do laboratório do SDEEC3.

- **Descrição textual do computado (tipo, marca, localização, ….):** Intel(R) Core(TM) i5-4690 CPU @ 3.50GHz
- **Número de Cores/CPUs:** 4 Cores
- **Velocidade do processador:** 3.5GHz
- **Sistema operativo:** Linux Mint

Foram também realizados outros testes nas nossas máquinas que possuem mais cores. Embora os resultados obtidos nas nossas máquinas fossem melhores (mais rápidos), do que na máquina do laboratório, optámos pelos resultados da máquina do laboratório, de modo a simplificar a comparação de resultados.

5.2 Resultados ap-paralelo-1

Os tempos de execução do **ap-paralelo-1** para os diversos *datasets* e número de *threads* encontram-se na tabela abaixo. Note-se que foram obtidos todos os tempos de execução do serial-basic.

Tabela 7: *Tempos de processamento* ***ap-paralelo-1****(segundos)*

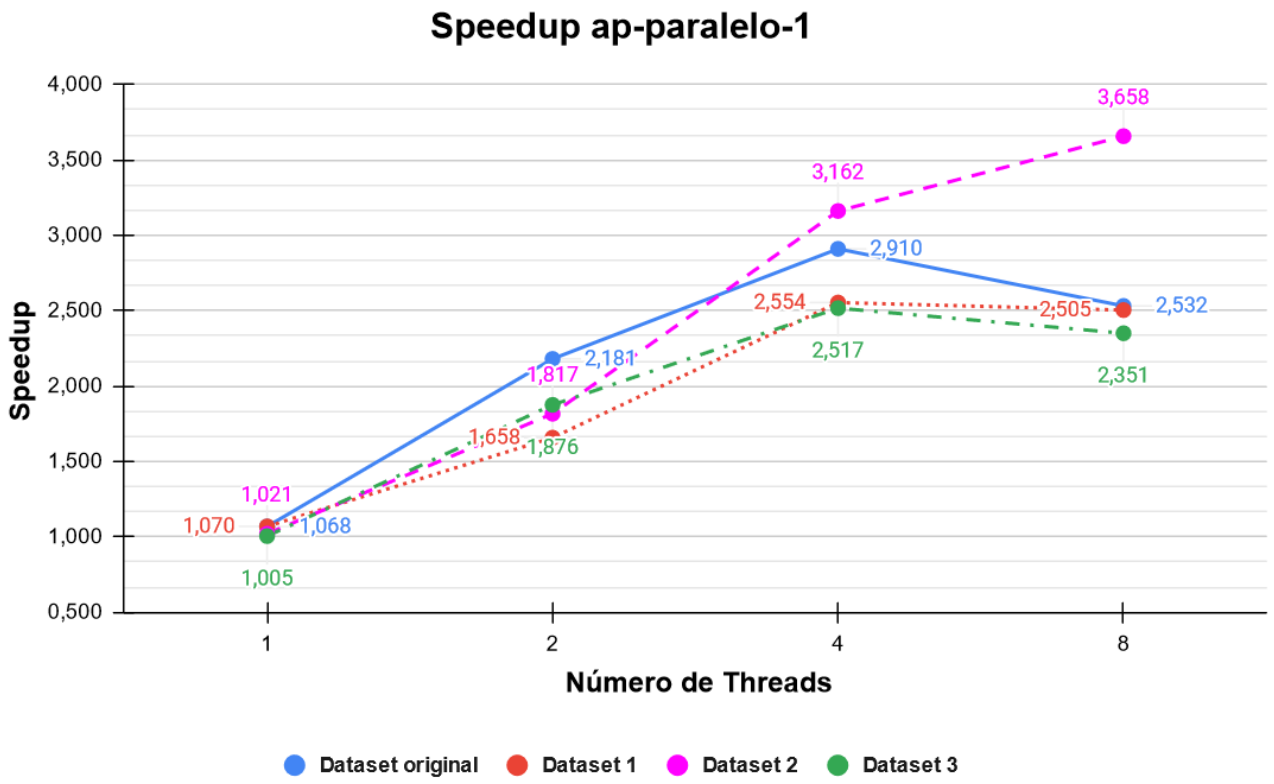
	Número de threads				
	Original / sem threads	**1**	**2**	**4**	**8**
Dataset original	7,436	6,960	3,410	2,555	2,937
Dataset 1	4,105	3,838	2,476	1,607	1,639
Dataset 2	13,352	13,081	7,348	4,223	3,650
Dataset 3	15,560	15,477	8,295	6,181	6,619

Tabela 8: *Valores de Speedup para* ***ap-paralelo-1*** *para todos os datasets*

	SPEEDUP			
	Número de threads			
	1	**2**	**4**	**8**
Dataset original	1,068	2,181	2,910	2,532
Dataset 1	1,070	1,658	2,554	2,505
Dataset 2	1,021	1,817	3,162	3,658
Dataset 3	1,005	1,876	2,517	2,351

Abaixo encontra-se um gráfico com os *speedups* calculados para o programa **ap-paralelo-1**, e diversos *datasets* e números de *threads*. De notar que quanto maior o valor de speedup melhor foi o tempo obtido pela aplicação.

Imagem 5: *Valores de Speedup para* ***ap-paralelo-1*** *para todos os datasets*



O speedup ideal não é obtido pelo facto de existir trechos de códigos onde a sequencialização é a única solução para o programa se poder compor. O facto de os speedups entre os datasets serem diferentes deve-se porque o número e o tamanho das imagens muda de dataset para dataset, logo é de esperar que os speedups sejam diferente entre datasets.

A diferença das 4 para as 8 threads não é significativa porque temos apenas 4 cpus, ou seja, só se vai conseguir realizar 4 tarefas em paralelo, entretanto as outras 4 aguardam disponibilidade.

Por fim, também se chegou à conclusão que ter uma thread é semelhante a ter o código sequencial.

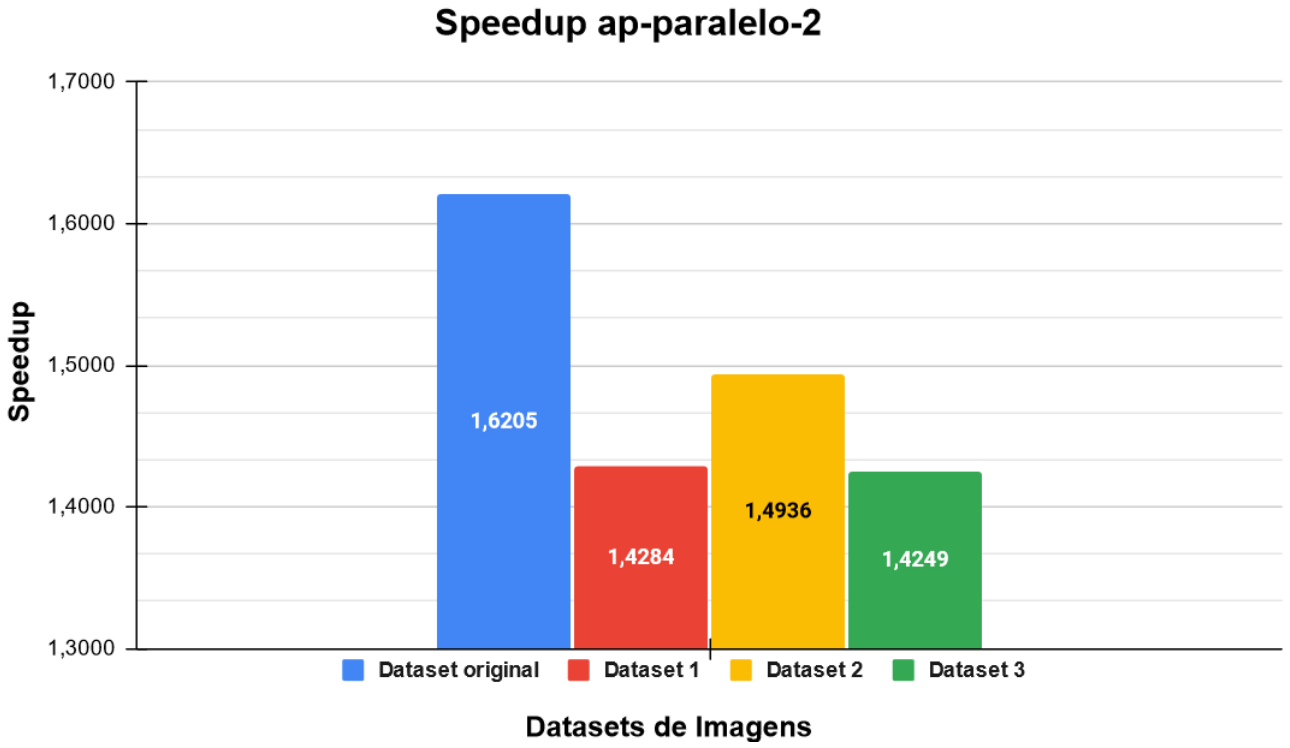
5.3 Resultados ap-paralelo-2

Tabela 9: *Tempos de processamento* ***ap-paralelo-2***

	Tempo de execução (s)		
	ap-paralelo-1 (1 thread)	***ap-paralelo-2***	**Speedup**
Dataset original	6,960	4,295	1,621

Dataset 1	3,838	2,687	1,428
Dataset 2	13,081	8,758	1,494
Dataset 3	15,477	10,862	1,425

Imagem 6: *Valores de Speedup para* ***ap-paralelo-2*** *para todos os datasets*



Relativamente ao speedup ideal e existirem diferentes speedups entre datasets, a explicação é a mesma que apresentada na secção 5.1 Resultados ap-paralelo-1.

Os resultados comprovam um speedup mais elevado para a ap-paralelo-2 com 3 threads em relação à ap-paralelo-1 com 1 thread. Seria de esperar speedups perto de 3 dado que em parte estamos a replicar o que se faz na parte 1 e a multiplicar os recursos para o fazer num factor 3. No entanto, temos algumas limitações relativamente ao que pode e ao que não pode ser paralelizado, daí o speedup não chegar a ser 3. Outra coisa que se consegue observar é a diferença nos valores do speedup entre os datasets, tal se deve à diferença do tamanho entre as imagens.

6 Conclusão

O desenvolvimento deste trabalho permitiu aprender a utilizar threads assim como a manipular tudo o que entra e saí do resultado das mesmas. Foi também possível aprender sobre o funcionamento e comportamento de alguns processadores, nomeadamente dos que têm menos cores e de que maneira se comportam com o paralelismo.