Programação concorrente

2022/2023

Projeto – Parte A

O processamento computacional de grandes conjuntos de dados é normalmente composto por um conjunto de tarefas de elevada complexidade temporal. No entanto, muitas vezes pode-se tirar partido da existência de múltiplos CPUs ou *Cores* para reduzir o tempo total de processamento.

Um desses exemplos é a conversão e processamento de imagens. Normalmente em bancos de imagens todas as imagens sofrem um conjunto de conversões igual: criação de uma copia com menor resolução, criação de um *thumbnail*, ou mesmo aplicação de marcas de água. Estes passos são aplicados de forma igual a todas as imagens ou fotografias.

Neste projeto os alunos irão paralelizar um conjunto de tarefas (processamento de imagens) de modo a tirar partido dos vários *Cores* de um computador e reduzir o tempo total de processamento.

1 Código de exemplo

Em anexo a este enunciado (na diretoria **gd-examples/**), são fornecidas três aplicações que processam um conjunto de ficheiros. Cada aplicação aplica a cada ficheiro uma transformação diferente para produzir novas imagens:

- **generate-resize.c** - redução da dimensão,
- **generate-thumbnail.c** - criação de um *thumbnail* quadrado,
- **generate-watermark.c** - aplicação de uma marca de água.

A quarta aplicação (**serial-process.c**, na diretoria **serial-version/**) implementa de forma sequencial (usando apenas um C*ore*/*cpu*) as funcionalidade descritas neste enunciado e que devem ser replicadas no projeto.

Estas aplicações estão desenvolvidas em C e usam a biblioteca `libGD`[1] para leitura, manipulação e armazenamento das imagens.

As aplicações na pasta **gd-examples/** iteram sobre um vetor de strings (contendo os nomes dos ficheiros) e, para cada ficheiro, efetuam o seguinte:

- lêem do disco as imagens originais através da função **`read_png_file()`**
- **generate-resize.c**
  - cria uma versão reduzida de cada imagem através da função **`resize_image()`**
- **generate-thumbnail.c**
  - cria um *thumbnail* de cada imagem através da função **`make_thumb()`**
- **generate-watermark.c**
  - aplicam o símbolo do IST sobre cada imagem chamando a função **`add_watermark()`**
- todas as transformações a cada uma das imagens são gravadas no disco através da função **`write_png_file()`**

A quarta aplicação (na pasta **serial-version/**) processa de forma sequencial o mesmo conjunto de imagens e aplica a cada uma delas, em sequência, as seguintes transformações:

- aplica o símbolo do IST (marca de água) sobre cada imagem chamado a função **`add_watermark()`**
- cria uma versão reduzida de cada imagem com a marca de água chamando as funções **`add_watermark()`** e **`resize_image()`**

1 https://libGD.github.io/

- cria um *thumbnail* de cada imagem com a marca de água chamando as funções e **`add_watermark()`** e **`make_thumb()`**

Como estas aplicações apenas processam um conjunto pré-definido de imagens (declarado no código numa variável), deverão apenas ser utilizadas para:

- **generate-resize.c generate-thumbnail.c generate-watermark.c**
  - perceber os diversos processamentos e transformações a serem realizadas pelas versões paralelas
  - extrair e reutilizar as funções fundamentais para o processamento das imagens
- **serial-basic.c**
  - referência para verificar o correto funcionamento das verões paralelas
  - referência para comparar os ganhos das versões paralelas

São fornecidos dois ficheiros auxiliares (**image-lib.c** e **image-lib.h**) que contêm todas as funções necessárias à manipulação e transformação das imagens. De modo a simplificar a compilação dos exemplos, são também disponibilizadas duas **Makefiles**, em cada uma das sub-diretoria.

1.1 Instalação da biblioteca libGD

Para execução do programa fornecido e realização do projeto é necessário instalar a biblioteca `libGD`.

Em Ubuntu/Debian basta executar o seguinte comando:

```
sudo apt install libgd-dev
```

Noutras distribuições de Linux o nome do pacote e comando são diferentes:

- centos / RedHat – `sudo dnf install gd-devel.x86_64`
- OpenSuse - `zypper install gd-devel`

Em cygwin também existe um pacote que permite instalar a biblioteca:

- `libgd-devel`

1.2 Compilação com libGD

De modo a compilar programas que usem a biblioteca libGD é necessário fazer o seguinte `include`:

```
#include <gd.h>
```

Aquando da compilação do programa final é necessário adicionar a opção **`-lgd`** para além de todas as opções normalmente usadas:

```
gcc -g programa.c -o programa -lgd
```

1.3 Descrição da biblioteca

A biblioteca libGD usa um tipo de dados (*`gdImagePtr`*) que permite armazenar e representar imagens de diversos formatos (jpg, tiff, png). As variáveis deste tipo são manipuladas por uma série de funções que permitem carregar as imagens para memória a partir do disco, manipulá-las, ou mesmo gravar as imagens de volta para o disco.

1.3.1 Leitura e escrita de imagens a partir do disco

A biblioteca `libGD` tem funções que leem imagens de diversos formatos com o apoio do *`fopen()`*: o programa abre o ficheiro com a imagem usando *`fopen()`*, e usa o *`FILE *`* retornado como argumento da função de leitura. Depois de lida a imagem para a variável do tipo *`gdImagePtr`*, o ficheiro pode ser fechado com *`fclose()`*.

A escrita é feita de forma semelhante, sendo necessário abrir o ficheiro em modo de escrita e a invocação de uma função específica.

Apresentam-se de seguida duas funções que encapsulam estes processos. A função de leitura (*`read_png_file()`*) recebe um nome de um ficheiro do tipo `png` e retorna a *`gdImagePtr`*. Esta função retorna *`NULL`* em caso de erro (impossível abrir o ficheiro ou ler o seu conteúdo).

A função de escrita (*write_png_file()*) recebe um argumento to tipo *gdImagePtr* e um nome e grava essa imagem em disco num ficheiro com esse nome. Esta função retorna 1 em caso de sucesso.

A seguinte tabela apresenta pedaços de código que demonstram a leitura e escrita de imagens. Observe que se abre o ficheiro com a função *fopen()* e que o *FILE** retornado é depois usado para ler a imagem (*gdImageCreateFromPng()*) ou escreve-la em disco (*gdImagePng()*).

```
gdImagePtr read_png_file(
                char * file_name){
   FILE * read_fp;
   gdImagePtr img;
   fp = fopen(file_name, "rb");
   if (!fp) {
      return NULL;
   }
   read_img= gdImageCreateFromPng(fp);
   fclose(fp);
   if (!read_img) {
      return NULL;
   }
   return read_img;
}
```

```
int write_png_file(
              gdImagePtr write_img,
             char * file_name){
   FILE * fp;
   fp = fopen(file_name, "wb");
   if (!fp) {
      return 0;
   }
   gdImagePng(write_img, fp);
   fclose(fp);
   return 1;
}
```

1.3.2 Funções de manipulação

As várias funções de manipulação da imagens recebem, para além do argumento referente à imagem que irá ser manipulada, outros parâmetros relevantes.

Foram desenvolvidas 3 funções que encapsulam as funções do libGD:

```
gdImagePtr  resize_image(gdImagePtr in_img, int new_width)
```

Este função recebe como argumento uma imagem (*gdImagePtr in_img*) e escala-a de modo a ficar com a largura pretendida (*new_width*). Este função retorna uma nova imagem com tamanho diferente, mas com as mesmas proporções.

```
gdImagePtr make_thumb(gdImagePtr in_img, int size)
```

A função *make_thumb()* recebe uma imagem (*in_img*) e cria um *thumbnail* quadrado com largura *size*. Esta função retorna uma nova imagem quadrada.

```
gdImagePtr  add_watermark(gdImagePtr in_img, gdImagePtr watermark)
```

A função *add_watermark()* recebe duas imagens e sobrepõe a imagem *watermark* no canto superior esquerdo da imagem *im_img*. É retornada uma nova imagem.

1.3.3 Gestão de memória

As variáveis do tipo *gdImagePtr* apontam para uma estrutura que armazena toda a informação de uma imagem. A memória é alocada quando se lê do ficheiro ou quando se aplicam funções de transformação.

Quando deixam de ser necessárias, as imagens têm de ser libertadas. Para realizar essa operação deve ser usada a função *gdImageDestroy()*.

2 Descrição da parte A do projeto

Neste projeto os alunos deverão implementar duas versões paralelas da aplicação fornecida (**serial-basic.c**), de modo a reduzir o tempo de processamento de conjuntos de imagens.

As aplicações efetuam três transformações a cada uma das imagens armazenadas numa diretoria (produzindo três novos conjuntos de imagens) e, de modo a acelerar o processo e reduzir o tempo de processamento usarão *threads* de duas formas diferentes.

2.1 Funcionalidades gerais

A lista de imagens a processar será fornecida a cada uma das aplicações paralelas através de um ficheiro de texto que contem em cada linha o nome de uma imagem a ser processada. Depois de lerem os nomes das imagens do ficheiro de configuração, para cada uma das imagens as aplicações paralelas produzem o seguinte:

- imagem original com o símbolo do IST aplicado como marca de água
- versão reduzida da imagem com o símbolo do IST aplicado como marca de água.
- *thumbnail* da imagem com símbolo do IST aplicado como marca de água

As imagens resultantes terão o nome da imagem original, mas serão armazenadas em diretorias específicas tal como descrito na Secção 3.2

2.2 Paralelização 1

A primeira aplicação paralela (chamada **ap-paralelo-1**) tem apenas um tipo de *threads. Cada thtread* processa um sub-conjunto disjunto das imagens e para cada uma das imagens a si atribuídas gera as três imagens transformadas (como descrito na secção

2.1). O número de *threads* criadas é definido pelo utilizador através de um argumento da linha de comando.

Como ilustra a Figura 1 o `main` divide o trabalho pelo número de *threads* definido pelo utilizador e cria-as. Cada *thread* executa as três transformação para os ficheiros que lhe foram atribuídas e, para esperar pelo fim da *threads*, o *`main()`* faz *`pthread_join()`*.

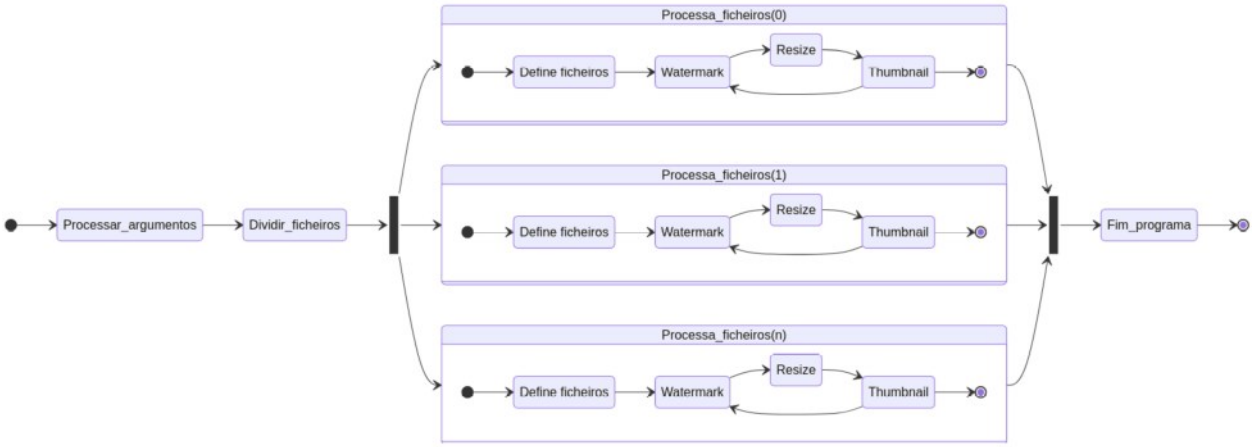


Figura 1: Fluxo de execução da paralelização 1

2.2.1 Paralelização 2

Na segunda aplicação (chamada **ap-paralelo-2**), haverá somente três *threads* distintas. Cada *thread* faz uma transformação específica (como ilustrado na Figura 2), e processa todas as imagens:

- A thread **Processa_watermarks** aplica uma marca de água a todas as imagens
- A thread **Processa_resizes** aplica a marca de água e faz o redimensionamento de todas as imagens
- A thread **Processa_Thumbnails** aplica a marca de água e cria a thumbnail para todas as imagens

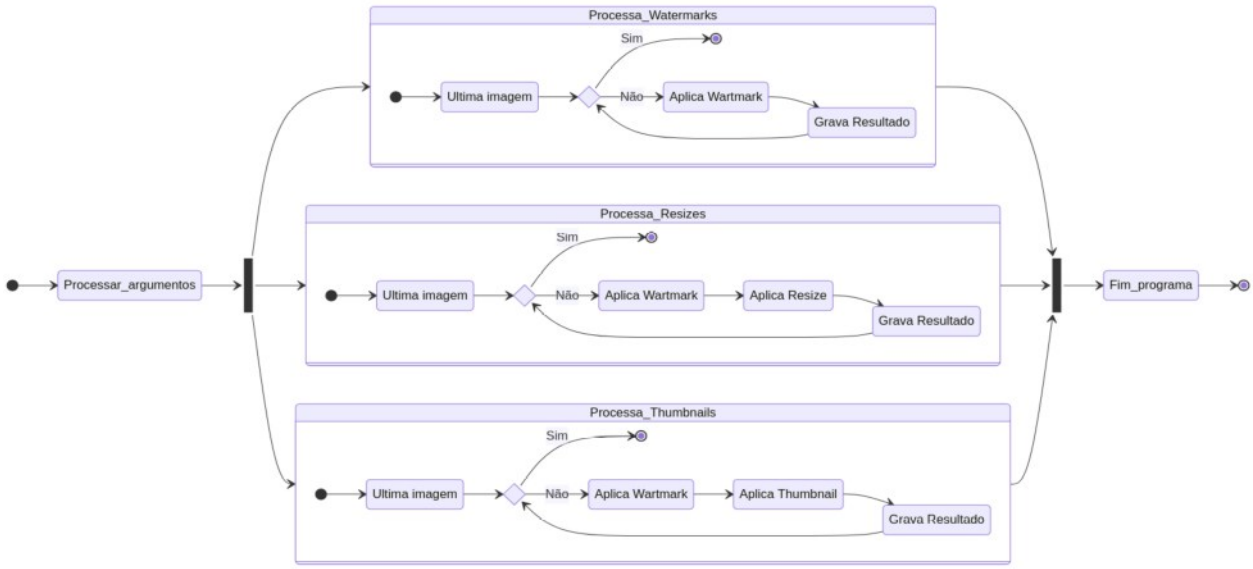


Figura 2: Fluxo de execução da paralelização 2

Na Figura 2 e descrição anterior, as três *threads* executam concorrentemente, mas se os alunos entenderem poderão organizar a sequenciação e execução e responsabilidade destas *threads* de outra forma com a ajuda do `pthread_join()`, de modo a reduzir o tempo de execução.

3 Funcionamento das aplicações

As aplicações serão desenvolvida em **C** e executarão em Linux/WSL (ou Cygwin).

Para cada execução, o utilizador deverá indicar através dos argumentos da linha de comando o seguinte (pela ordem indicada):

- **ap-paralelo-1**
  - a diretoria onde se encontram as imagens originais,
  - o número de *threads.*
- ***ap-paralelo-2***
  - a diretoria onde se encontram as imagens originais,

Os programas aplicarão as transformações atrás descritas, armazenando as imagens num conjunto de pastas pré-definido.

3.1 Dados de entrada

A diretoria onde se encontram as imagens originais e o número de *threads* (no caso do *ap-paralelo-2)* são definidos pelo utilizador na linha de comando aquando da execução das aplicações. Na diretoria deverá existir o ficheiro `image-list.txt` .

3.1.1 Argumentos da ap-paralelo-1

Para a execução da paralelização 1 (**ap-paralelo-1**) o utilizador deverá indicar a diretoria onde se encontram as imagens e o número de *threads* a criar como exemplificado de seguida.

```
ap-paralelo-1 dir-1 4
ap-paralelo-1 dir-2 8
app-parallel-1 . 1
```

O primeiro argumento corresponde à diretoria e o segundo ao número de *threads*.

O número de *threads* deve ser um qualquer número inteiro positivo. Se, por exemplo, o utilizador indicar **1** como o número de *threads* a criar, o programa utilizará apenas uma *thread* **para além** do *`main()`* para efetuar o processamento de todas as imagens.

3.1.2 Argumentos da ap-paralelo-2

Para a execução da paralelização 2 (**ap-paralelo-2**) o utilizador apenas deverá indicar a diretoria onde se encontram as imagens:

```
ap-paralelo-2 dir-1
ap-paralelo-1 dir-2
app-parallel-1 .
```

Neste caso não é necessário indicar o número de *threads*, dado que este valor é fixo, sendo criadas 3 *threads* distintas.

3.1.3 Imagens a processar

O ficheiro `image-list.txt`, contém a lista das imagens que se encontram na diretoria e que devem ser processadas. Este ficheiro contêm um nome de ficheiro por linha. As aplicações deverão ler este ficheiro e só as imagens aí listadas serão processadas. Assim, a diretoria indicada pelo utilizador poderá conter mais imagens do que aquelas a serem processadas.

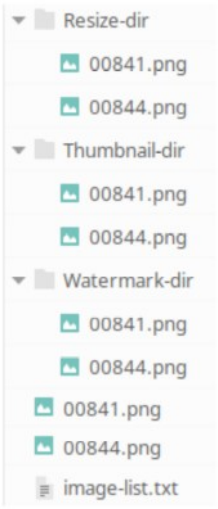
Apenas deverão ser processadas imagens do formato PNG.

Serão fornecidos conjuntos de imagens (com ficheiro `image-list.txt`) de modo que os alunos tenham diversos conjuntos de dados variáveis e comparáveis. Os alunos podem naturalmente utilizar outras imagens durante o desenvolvimento e teste do projeto.

3.2 Resultados

A execução das aplicações produz um conjunto de novas imagens, correspondentes à transformações de cada uma das imagens iniciais.

O nome das novas imagens será o mesmo da imagem original, mas cada imagem será colocada numa sub-diretoria específica:



- ***Resize-dir*** – sub-diretoria que contém os resultado da redução das imagens
- ***Thumbnail-dir*** – sub-diretoria que contém os *thumbnails* produzido
- ***Watermark-dir*** – sub-diretoria que contém a imagens com a marca de água aplicada

Estas sub-diretorias deverão ser criadas pelas aplicações a desenvolver, na diretoria indicada na linha de comando pelo utilizador e onde se encontram as imagens originais.

3.3 Interrupção de execução

Se as aplicações forem interrompidas a meio do processamento dos ficheiros, apenas parte dos resultados terão sido produzidos e guardados no disco.

Se o utilizador voltar a executar a aplicação, não deverá ser necessário voltar a produzir os ficheiros resultado já existentes. A aplicação só deverá processar e gastar tempo na criação dos ficheiros em falta.

Para verificar se um ficheiro existe os alunos poderão usar as funções *access()* como no seguinte exemplo:

```
if( access( nome_fich, F_OK ) != -1){
   printf("%s encontrado\n", nome_fich);
}else{
   printf("%s nao encontrado\n", nome_fich);
}
```

4 Submissão do projeto

O prazo para submissão da resolução da Parte A do projeto é dia **16 de Dezembro às 19h00** no FENIX.

Antes da submissão, os alunos devem criar grupos de 2 e registá-los no FENIX.

Os alunos deverão submeter um ficheiro **.zip** contendo o código de ambas as aplicações. O código para cada uma das aplicações deverá ser colocado numa das seguintes diretorias: **ap-paralelo-1/** ou **ap-paralelo-2/**. Os alunos deverão entregar também uma `Makefile` para a compilação das aplicações e verificar cuidadosamente que a mesma executa corretamente.

Os alunos deverão submeter um pequeno documento/relatório (chamado **pconc-relatorio-A.pdf**). O modelo do relatório será fornecido posteriormente, mas deverá listar as funcionalidades implementadas e apresentar os seguintes resultados para várias execuções:

- Numero de threads
- tempo total de execução da aplicação
- speedup
- características do Computador

Para obter o tempo total de execução, os alunos devem usar os output produzido pelo comando **time**:

time(1) - Linux manual page - man7.org

Estes resultados deverão ser recolhidos executando as duas aplicações nos computadores da SCDEEC3 com as várias combinações de:

- conjuntos de dados (a ser fornecidos pelos docentes),
- 1, 2, 4, 8 número de *threads*.

5 Avaliação do projeto

A nota para esta parte do projeto será dada tendo em consideração o seguinte:

- Número de aplicações e funcionalidades implementadas
- Modo de gestão das *threads* e recursos
- Estrutura e organização do código
- Tratamento de erros
- Comentários
- Relatório