Relatório Referente ao Projeto 1

Disciplina: Arquitetura de Computadores II

Professor: Alisson Brito

Alunos: Aline Moura Araújo (11512963) e

Luiz Henrique Freire Barros (11514334)

Esse projeto foi desenvolvido na linguagem Python, para a sua realização foi utilizado um programa que realiza operações em uma imagem de entrada para que a sua saída seja uma nova imagem em escalas de cinza, vermelho, verde ou azul. Na metodologia deste trabalho, foram utilizadas cinco versões de um mesmo código, sendo uma delas otimizada, três ineficientes, e uma com paralelismo.

O código pode ser acessado na página do github[1] e executado seguindo as instruções do arquivo *readme.md*.

Na primeira versão (v1) temos o código otimizado, nessa função a imagem de saída foi gerada conjuntamente com as operações de conversão, como é possível ver na imagem abaixo.

```
51      newImage=[]
52      for h in range(height):
53          newImage.append([])
54          for w in range(width):
55              # Verifica se precisa de parametros fora o R,G,B
56              if (parameters != None):
57                  newImage[-1].append(fun(img[h][w][0], img[h][w][1],
58                                          img[h][w][2],  parameters))
59              else:
60                  newImage[-1].append(fun(img[h][w][0], img[h][w][1],  img[h][w][2]))
61
62      return convertArrayToNumpy(newImage)
```

Imagem 1: Trecho de código da primeira versão (**v1**)

[1] https://github.com/luizhenriquefbb/CC-ArqII-ProjetoI

Na segunda versão (v2) temos o código a imagem de saída sendo gerada em um laço de repetição fora do laço da operação de conversão da imagem, além disso, os elementos da matriz são acessados aleatoriamente, como pode-se observar na imagem abaixo.

```
52      newImage=[]
53      for _ in range(height):
54          newImage.append( [None] * width )
55
56      # acessar elementos da matriz aleatoriamente
57      shuffleHeight = (range(height)) # nao aleatorizado ainda
58      shuffleWidth = (range(width)) # nao aleatorizado ainda
59
60      random.shuffle(list(shuffleHeight)) # aleatorizar os acessos
61      random.shuffle(list(shuffleWidth)) # aleatorizar os acessos
62
63      for h in shuffleHeight:
64          for w in shuffleWidth:
65              # Verifica se precisa de parametros fora o R,G,B
66              if (parameters != None):
67                  newImage[h][w]=(fun(img[h][w][0], img[h][w][1],
68                                      img[h][w][2],  parameters))
69              else:
70                  newImage[h][w]=(fun(img[h][w][0], img[h][w][1],  img[h][w][2]))
71
72      return convertArrayToNumpy(newImage)
```

Imagem 2: Trecho de código da segunda versão (**v2**)

Na terceira versão (**v3**) a imagem é criada em um laço de repetição ‘for’ separadamente do laço da operação de conversão da imagem. No laço de conversão foi percorrido primeiramente as colunas e depois as linhas da matriz. Abaixo é possível visualizar o código.

```
51      newImage=[]
52      for _ in range(height):
53          newImage.append( [None] * width )
54
55      # Usar metodo pixel a pixel
56      for w in range(width):
57          for h in range(height):
58              # Verifica se precisa de parametros fora o R,G,B
59              if (parameters != None):
60                  newImage[h][w]=(fun(img[h][w][0], img[h][w][1],
61                                      img[h][w][2],  parameters))
62              else:
63                  newImage[h][w]=(fun(img[h][w][0], img[h][w][1],  img[h][w][2]))
64
65      return convertArrayToNumpy(newImage)
```

Imagem 3: Trecho de código da terceira versão (**v3**)

Na quarta versão (**v4**), a imagem é criada novamente em laços separados como na versão anterior (**v3**) porém, nessa versão, o laço de repetição responsável pela conversão da imagem é percorrido de forma inversa, primeiramente as linhas e depois as colunas, como é possível perceber na imagem abaixo.

```
51      # construir matriz da nova imagem
52      newImage=[]
53      for _ in range(height):
54          newImage.append( [None] * width )
55
56      # Usar metodo pixel a pixel
57      for h in range(height):
58          for w in range(width):
59              # Verifica se precisa de parametros fora o R,G,B
60              if (parameters != None):
61                  newImage[h][w]=(fun(img[h][w][0], img[h][w][1],
62                                      img[h][w][2],  parameters))
63              else:
64                  newImage[h][w]=(fun(img[h][w][0], img[h][w][1],  img[h][w][2]))
65
66      return convertArrayToNumpy(newImage)
```

Imagem 4: Trecho de código da quarta versão (**v4**)

Na última versão (**v5**), foram desenvolvidas três formas diferentes para paralelizar o processo, na primeira forma (*handytrheads*), foram utilizados *threads*, na segunda (*handytrheads2*), foram utilizados processos, e, na terceira (*handytrheads3*), foram utilizados processos com memória compartilhada.

- *handytrheads*: Quando o programa foi dividido em threads, elas se distribuíram pelos processadores, neste trabalho foram divididos em quatro threads. Porém, as threads não são executado com 100% da capacidade pois, um único processo python, elas compartilham o mesmo interpretador, fazendo com o seu desempenho seja sempre perto de 30%. Por causa desse empecilho, o custo acaba piorando porque além de compartilhar o mesmo interpretador, ainda há o custo da troca de threads. Na imagem abaixo é possível ver o desempenho das CPUs no momento da execução, destacado em azul.

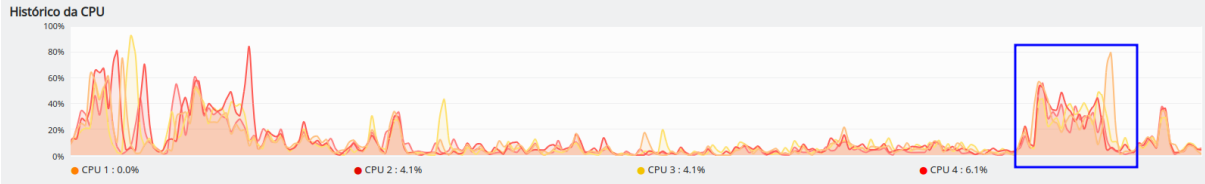

Imagem 5: Monitor de Sistema na execução do *handytrheads* com a imagem “image.jpeg (**v5**)

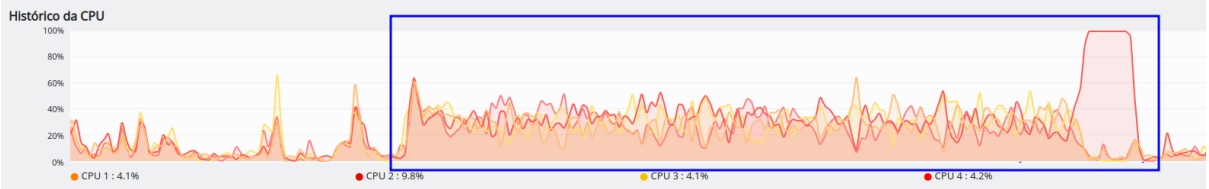

Imagem 6: Monitor de Sistema na execução do *handytrheads* com a imagem “image2.jpg (**v5**)

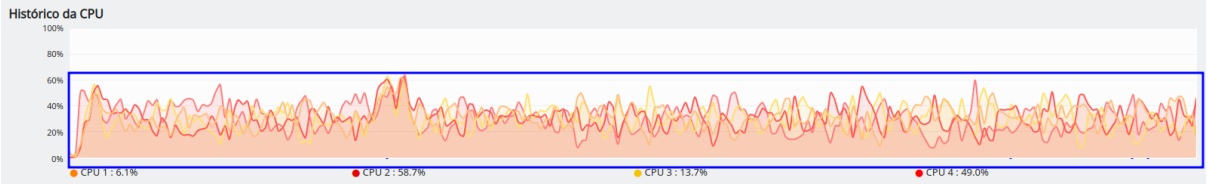

Imagem 7: Monitor de Sistema na execução do *handytrheads* com a imagem “image3.jpeg (**v5**)

- *handytrheads2*: Esse método obteve o melhor tempo de execução. Cada processador atingiu 100% da capacidade de processamento, pois, em vez de dividir o processamento em *threads*, o programa foi dividido em processos, assim, cada processo tem o seu próprio interpretador. Porém, há um grande problema no resultado, pois, cada processo é criado a partir de um *fork* e, dessa forma, o retorno do processamento de cada *work* não é feito, acreditamos que seja um problema da linguagem e foi um problema que não

conseguimos solucionar. Sabe-se que em C/C++ a memória poderia ser compartilhada através de ponteiros. Na imagem abaixo é possível ver o desempenho das CPUs no momento da execução, destacado em azul.

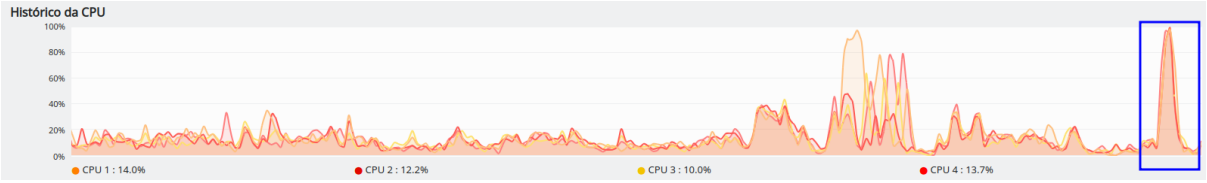

Imagem 8: Monitor de Sistema na execução do *handytrheads2 para image.jpeg* (**v5**)

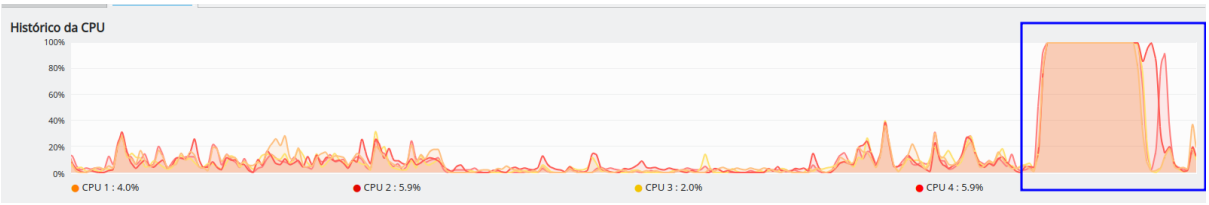

Imagem 9: Monitor de Sistema na execução do *handytrheads2 para image2.jpeg* (**v5**)

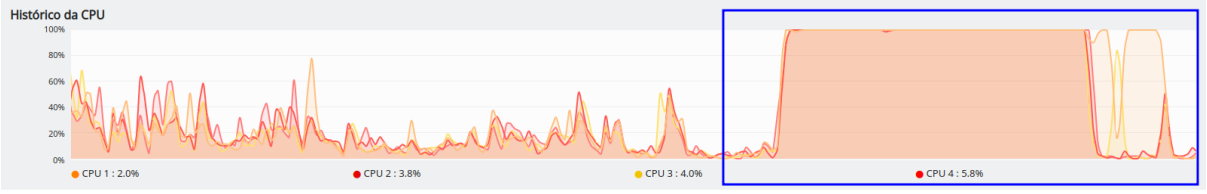

Imagem 10: Monitor de Sistema na execução do *handytrheads2 para image3.jpg* (**v5**)

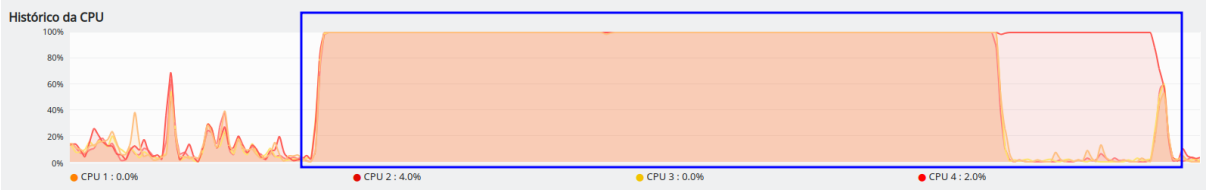

Imagem 11: Monitor de Sistema na execução do *handytrheads2 para image4.jpg* (**v5**)

- *handytrheads3*: Esse método foi desenvolvido para que a memória seja compartilhada entre os processos, fazendo com que cada thread receba uma cópia da lista original (imagem transformada em lista) e, só depois, divida a lista para que cada thread fique com a sua faixa correspondente. O objeto utilizado para compartilhar a memória é uma lista compartilhada nativa do Python (manager.list()). Na imagem abaixo é possível ver o desempenho das CPUs no momento da execução, destacado em azul.

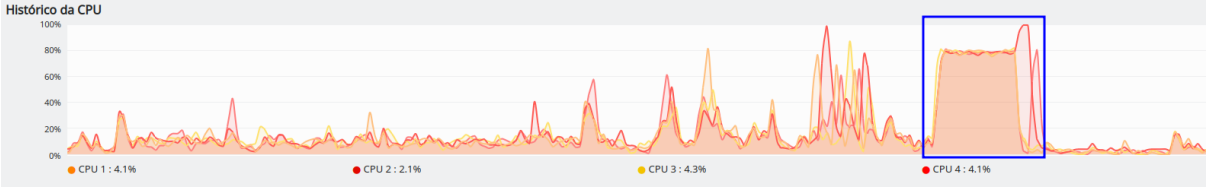

Imagem 12: Monitor de Sistema na execução do *handytrheads2 para image.jpeg* (**v5**)

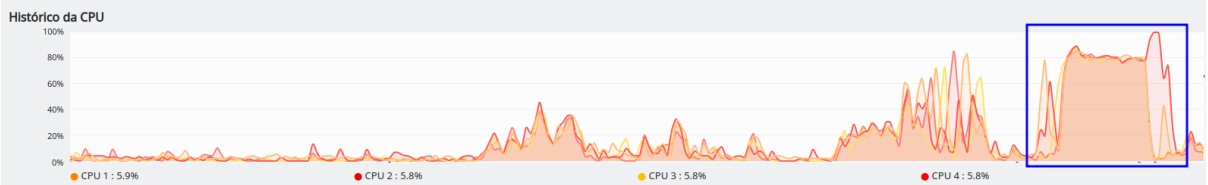

Imagem 13: Monitor de Sistema na execução do *handytrheads2 para image2.jpeg* (**v5**)

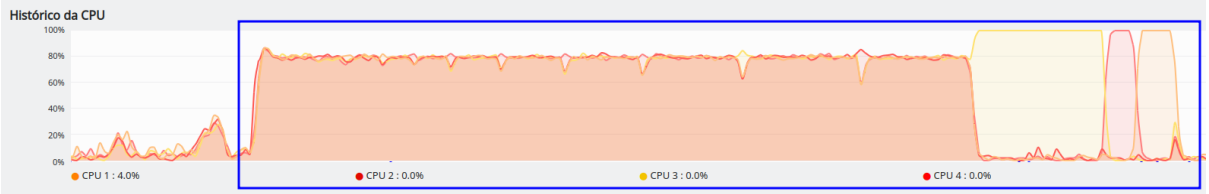

Imagem 14: Monitor de Sistema na execução do *handytrheads2 para image3.jpg* (**v5**)

Resultados

Na elaboração dos resultados, todas as versões foram executadas 5 vezes. Assim, é possível obter uma média do tempo de execução de cada versão para cada imagem. Nas versões v1, v2, v3, v4 e v5(*handytrheads2*) foram utilizadas 4 imagens de tamanhos diferentes (*image.jpeg*, *image2.jpeg*, *image3.jpg*, *image4.jpeg*), no entanto, para as versões v5(*handytrheads)* e v5(*handytrheads3*), foram utilizadas apenas 3 imagens (*image.jpeg*, *image2.jpeg*, *image3.jpg*). As imagens tinham os seguintes tamanhos:

- image.jpeg : 600X375
- image2.jpeg : 1920X1080
- image3.jpg : 3072X2048
- image4.jpeg : 4608X3072

A tabela e o gráfico abaixo mostra os resultados obtidos

versão	tamanho da imagem	média
v1	600X375	5,093838374
v1	1920X1080	46,97701783
v1	3072X2048	141,0684808

v1	4608X3072	307,389293
v2	600X375	4,843168974
v2	1920X1080	46,08059683
v2	3072X2048	152,7276857
v2	4608X3072	332,830218
v3	600X375	5,178227568
v3	1920X1080	47,90230637
v3	3072X2048	147,4672174
v3	4608X3072	355,7144599
v4	600X375	5,172127056
v4	1920X1080	46,41419382
v4	3072X2048	154,983453
v4	4608X3072	322,0114369
v5 - handytrheads	600X375	8,3681409358978
v5 - handytrheads	1920X1080	79,3991839885712
v5 - handytrheads	3072X2048	241,0883660316470
v5 - handytrheads	4608X3072	-
v5 - handytrheads2	600X375	1,4819729328156
v5 - handytrheads2	1920X1080	13,9481158256530
v5 - handytrheads2	3072X2048	42,1154391765594
v5 - handytrheads2	4608X3072	92,2142791748046
v5 - handytrheads3	600X375	11,9307029247284
v5 - handytrheads3	1920X1080	100,9048039913180
v5 - handytrheads3	3072X2048	+400
v5 - handytrheads3	4608X3072	-

Tabela 1 - Média de cada uma das versões em relação ao tamanho da imagem

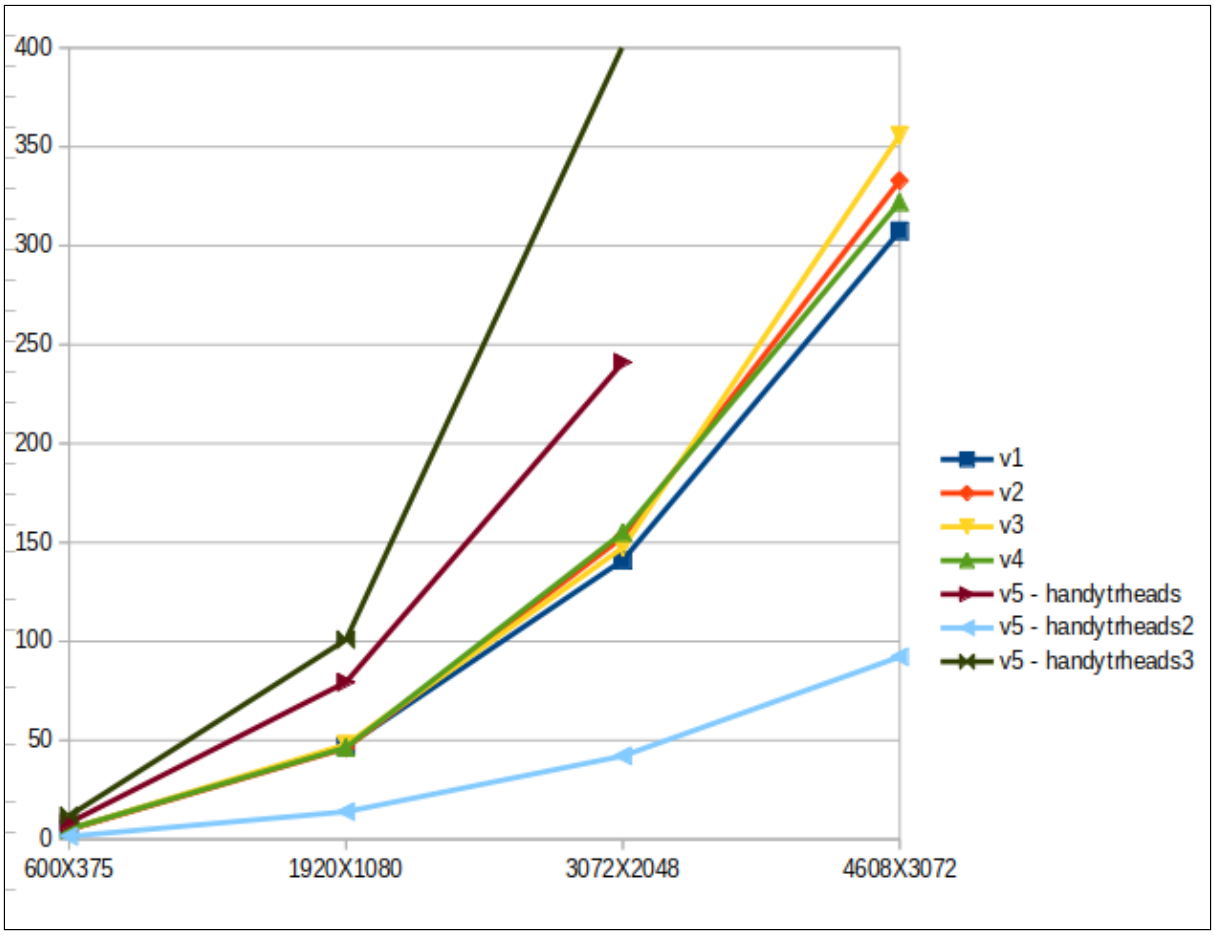

Gráfico 1 - Resultados obtidos

Podemos notar que para imagens relativamente pequenas, os resultados foram muito próximos, o que se pode concluir que a imagem cabia na cache. Logo, as otimizações não se faziam necessárias.

Para imagens maiores, como uma 4608 X 3072, podemos ver que as otimizações mostraram um comportamento diferente. A versão otimizada, como era de se esperar, obteve os melhores resultados, concluindo a execução em menos tempo. Tendo em vista que em vez de realizar dois laços “for”, realizava apenas um.

A terceira versão (percorrendo primeiro as colunas) obteve os piores resultados, o que também era de se esperar, pois é certo que em cada iteração, os dados não estariam na cache, pois desobedece o princípio da localidade espacial dos dados.

Para a **v5**, podemos observar que na execução do *handythread*, obteve melhor resultado comparado ao *handythread3*. No *handythread2* foi possível observar o melhor

resultado, porém a função não cumpriu o objetivo e não retornou a imagem na escala escolhida, apesar do processamento ser feito corretamente.

Era de se esperar que, ao utilizar o paralelismo, o resultado do processamento melhorasse em relação ao **v1**, que é a versão otimizada do código, ainda assim, mesmo o *handythread* tendo mostrado o melhor resultado, não conseguiu ser melhor que a execução do **v1**.