INPE

Pós-Graduação em Computação Aplicada

CAP 378 - Tópicos em Observação da Terra

Leticia Meirelles

Eduardo Furlan

Trabalho final: PAD (processamento de alto desempenho) em PDI (Processamento Digital de Imagens)

São José dos Campos - SP

2019

SUMÁRIO

1. INTRODUÇÃO

O crescimento constante na velocidade de cálculos dos processadores tem sido uma grande aliada na evolução de áreas da Ciência que exigem processamento de alto desempenho. Juntamente com os recursos computacionais, faz-se necessário o emprego de técnicas de computação paralela no intuito de explorar ao máximo a capacidade de processamento da arquitetura escolhida, bem como, reduzir o tempo de espera no processamento. No entanto, o custo financeiro para aquisição deste tipo de hardware não é muito baixo, implicando na busca de alternativas para sua utilização. As arquiteturas de processadores multicores e General Purpose Computing on Graphics Processing Unit (GPGPU), tornam-se opções de baixo custo, pois são projetadas para oferecer infraestrutura para o processamento de alto desempenho e atender aplicações que requerem resultados de forma muito rápida, em particular em tempo real. Com o aperfeiçoamento das tecnologias, multiprocessador e GPGPU, a paralelização de técnicas de processamento e análise de imagem tem obtido destaque por viabilizar a redução do tempo de processamento de métodos complexos aplicados em imagens complexas. A indústria de processadores passou a oferecer chips com vários núcleos compondo o processador (KIRK et al, 2016). Estes processadores baseados na arquitetura multicore de uso geral, oferecem recursos de paralelismo, proporcionando ganho de desempenho e processamento mais rápido (VAJDA, 2018). A procura por processamento de alto desempenho, em geral, tem sido atendida por equipamentos ou sistemas computacionais de custo elevado. Diante da popularização das Graphics Processing Units (GPUs) e a aplicação de técnicas consolidadas de programação paralela, diversas áreas de pesquisa como a computação científica (PAGE, 2009), o processamento e a análise de imagem (GABRIEL et al, 2010) e muitas outras, podem conquistar avanços ainda mais significativos, sem a necessidade de grandes investimentos financeiros. Para análise desse tipo de informação aplica-se os conceitos da área de Processamento de Digital de imagens

(PDI). A PDI oferece uma grande variedade de metodologias que permitem a extração de informações com grande utilidade para a análise das imagens de satélites, e o PAD (Processamento de Alto Desempenho) pode ser definido como a prática de agregar capacidade de computação de tal forma a gerar um desempenho muito maior do que poderia ser obtido normalmente, e possui o objetivo de resolver grandes problemas em ciência, engenharia, ou negócios. PAD como conceito não está limitado a supercomputadores ou grandes sistemas, podendo ser aplicado a computadores comuns onde se utilizam técnicas para aproveitar toda capacidade de processamento e aumentar a eficiência.

O objetivo deste trabalho da disciplina CAP 378 é demonstrar o aumento de performance em conversão de imagens, utilizando-se Python e técnicas de PDA (Processamento de Alto Desempenho), em um ambiente de processamento paralelo.

Para uma melhor apresentação do trabalho, este documento está organizado da seguinte forma: no Capítulo 2 é apresentado a contextualização das tecnologias e metodologias utilizadas no projeto, no Capítulo 3 mostra-se o desenvolvimento da aplicação de conversão nas imagens escolhidas, os Capítulos 4 e 5 mostram os resultados e discussão, e por fim no Capítulo 6 apresenta-se a conclusão do trabalho seguido das referências e dos anexos nos quais se encontram os códigos fontes utilizados no projeto.

2. FUNDAMENTAÇÃO TEÓRICA

2.1. PROCESSAMENTO DIGITAL DE IMAGENS

Uma vez a imagem adquirida, seja qual for o meio, esta geralmente necessita de correções, sendo assim, pode-se definir processamento digital de imagem como conjunto de técnicas que tem como objetivo principal remover os principais tipos de degradações e distorções inerentes aos processos de aquisição, transmissão e

visualização de imagens coletadas, facilitando a extração de informações. (Silva, 2003).

O objetivo de utilizar PDI (Processamento Digital de Imagem) é melhorar o aspecto visual, viabilizando para o analista humano subsídios para a sua interpretação, gerando outros produtos que podem ser úteis em análises futuras. As técnicas mais utilizadas de PDI, conforme SPRING podem ser identificadas a seguir:

• **Contraste de imagem** – tem por objetivo aumentar a diferença de tons de cada objeto, melhorando o critério de observação do olho humano;

• **Leitura de pixel** – esta técnica não permite alteração da imagem, somente mostra o valor do nível de cinza ou composição colorida;

• **Filtragem espacial** – o processo consiste na aplicação de uma matriz na imagem original, analisando a leitura de pixel dos "vizinhos" sendo que a imagem resultante é uma imagem nova com a eliminação das linhas e colunas iniciais e finais da imagem original. Dentre os vários métodos de filtragem pode-se identificar alguns, divididos em três classes: filtros lineares (passa-baixa, passa-alta, editor de máscaras), filtros não lineares (operadores para detecção de bordas, filtros morfológicos, editor de elementos estruturantes) e filtro radar;

• **Operações aritméticas** – operações realizadas na imagem ou entre bandas a fim de realçar as similaridades e as diferenças de uma imagem;

• **Principais componentes** – são técnicas que realçam, reduzem ou removem a redundância das bandas, gerando uma nova imagem, onde apresenta informações ainda não encontradas;

• **Modelo de mistura** – esta é uma situação que ocorre quando um pixel representa mais que um tipo de cobertura de terreno. Neste caso, são aplicados cálculos matemáticos para identificar a proporção existente de cada representação do terreno em um pixel;

• **Segmentação de imagem** – consiste em agrupar pixels necessários para representar determinado segmento, isto é, extrair os objetos relevantes para a aplicação desejada;

• **Classificação de imagem** – é um processo de extração de informação em imagens para reconhecer padrões e objetos homogêneos;

• **Estatística de imagens digitais** – esta técnica utiliza cálculos de análises estatísticas como momentos, mediana, matriz de covariância e correlação, matriz de autocorrelação e matriz de correlação cruzada e os resultados podem ser apresentados na forma de gráficos e textos;

• **Restauração** – consiste em corrigir as distorções deixadas pelo sensor óptico, quando as imagens digitais são geradas. O princípio básico é restaurar a imagem e reduzir o efeito de borramento, constituindo assim uma imagem realçada;

• **Eliminação de Ruído** – é considerado um ruído, quando um pixel apresenta níveis muito diferentes de sua vizinhança, descaracterizando a imagem e podem aparecer aleatoriamente ou em listras verticais ou horizontais.

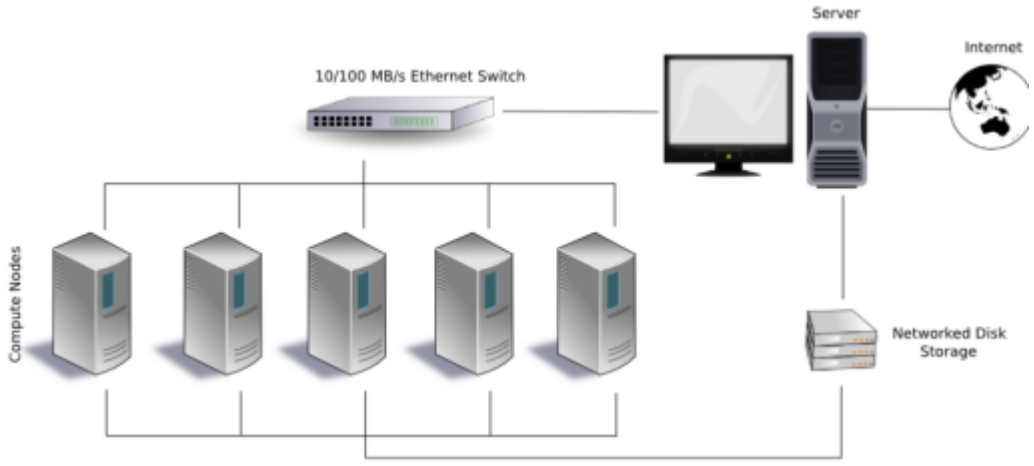
2.2. PROCESSAMENTO DE ALTO DESEMPENHO

O desenvolvimento de soluções computacionais capazes de realizar o Processamento e Análise de Imagem, de maneira geral, tem contribuído no avanço das ciências de geoprocessamento. O presente trabalho procura descrever os procedimentos para aplicar sistemas de processamento imagem acelerados por computação de alto desempenho.

Atualmente os principais sistemas computacionais de alto desempenho utilizam processadores comerciais e possuem arquitetura interligada para obter desempenho através da computação paralela. Simplificadamente são compostos por muitos processadores em paralelo trabalhando em conjunto. Programas de alto desempenho são feitos para dividir o processamento entre os vários núcleos, executando ao mesmo tempo, de tal forma a aproveitar a capacidade conjunta do sistema.

Na atualidade, diversos setores da atividade humana têm demandado por elevado poder de processamento e armazenamento de dados. Embora setores econômicos e governamentais demandem cada vez mais por tais recursos, ainda é na ciência em que esse tipo de demanda se concentra. Os Supercomputadores foram a primeira iniciativa para prover alto poder de computação, todavia são onerosos e pouco escalonáveis. Como alternativa aos Supercomputadores convencionais surgiu o Cluster, que provê computação de alto desempenho (HPC) a baixo custo já que utiliza commodities como componentes. A figura a seguir mostra o esquema de um *cluster* típico.

Figura 1 - Um *cluster* típico



Fonte: https://en.wikipedia.org

2.3. ANÁLISE DE DESEMPENHO

Alguns conceitos que são inerentes ao PAD são o de medição e análise de desempenho para que se possa avaliar a performance computacional obtida, e o de identificação e quantificação de fatores que afetam a escalabilidade de computação paralela.

3. DESENVOLVIMENTO

3.1. AMBIENTE DE PROGRAMAÇÃO

- Python
- MPI (bibliotecas)
- Slurm (gerenciador e escalonador de recursos)

PYTHON (https://www.python.org/)

Python foi escolhido porque é um forte candidato para escrever as partes de muito alto nível em aplicações científicas que utilizam processamento de alto desempenho, pois agrega todos os benefícios de uma linguagem de script como rapidez e economia de desenvolvimento de código e manutenção, além de atuar como uma extensão, integrando bibliotecas escritas em outras linguagens compiladas em um único ambiente fácil de ser usado.

MPI (https://www.mpi-forum.org/)

Foi utilizado a biblioteca “mpi4py” que é uma implementação do MPI. O MPI é um padrão aberto para comunicação de dados em computação paralela. Ele foi escolhido para este trabalho por ter larga aceitação no mercado, com a participação de 40 organizações na padronização, incluindo vendedores, pesquisadores, desenvolvedores de bibliotecas, e usuários. Suas principais vantagens são portabilidade, eficiência, e flexibilidade.

SLURM (https://slurm.schedmd.com/)

O Slurm é um gerenciador de ambiente de execução, permitindo o escalonamento de tarefas e gerenciamento de recursos em ambientes de Cluster e processamento paralelo. Foi escolhido para este trabalho por ser de código aberto, relativamente simples de instalar e configurar, além de ser utilizado em 60% dos supercomputadores TOP500 (fonte: Wikipedia).

3.2. DATASET

O download das imagens foi realizada com GmapCatcher que é uma suíte básica escrita em Python e pode ser executado no Linux, Windows e Mac OSX. Os blocos baixados precisam ser postados - processados e georreferenciados para reconstruir a imagem (https://github.com/heldersepu/gmapcatche).

Figura 2 - Tela principal do GmapCatcher

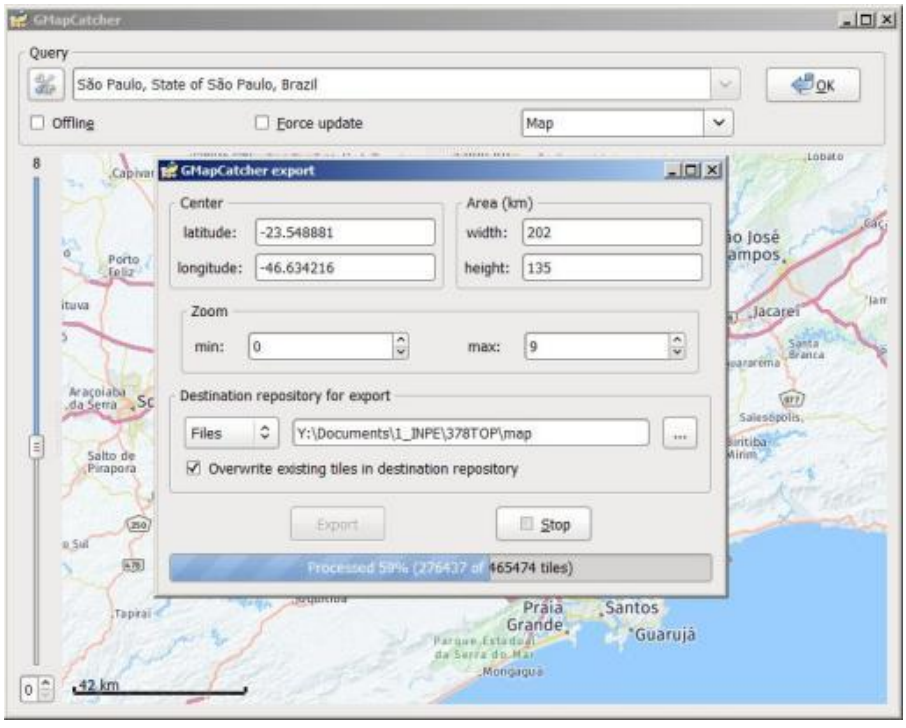
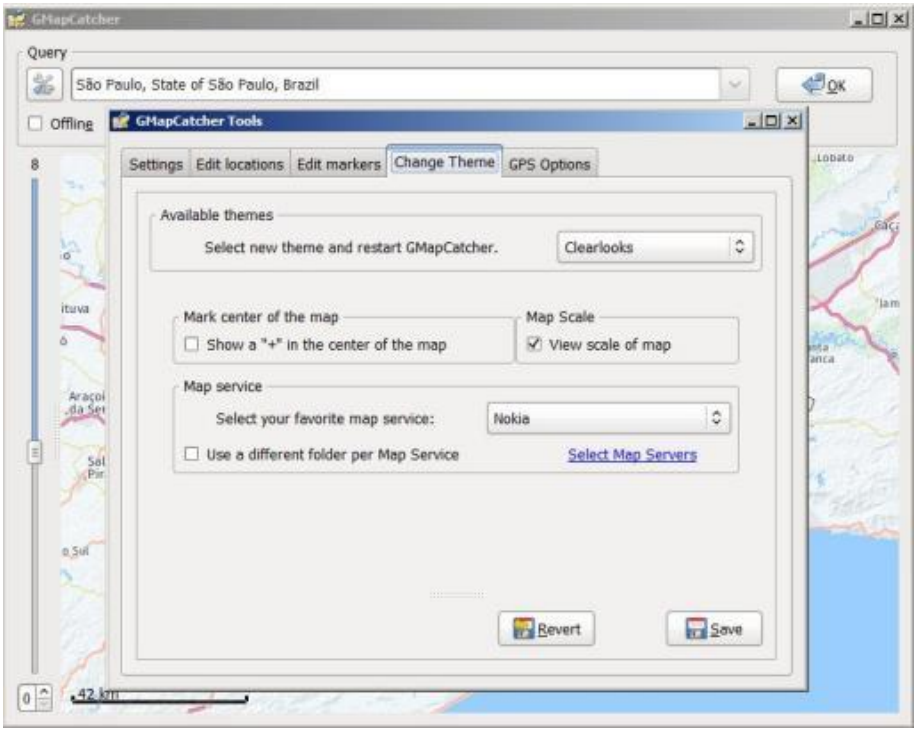


Figura 3 - Serviço online Nokia Maps



Usando o GMapCather (escrito em Python) navegamos online, e as imagens vão sendo baixadas para posteriormente serem gravadas em um diretório.

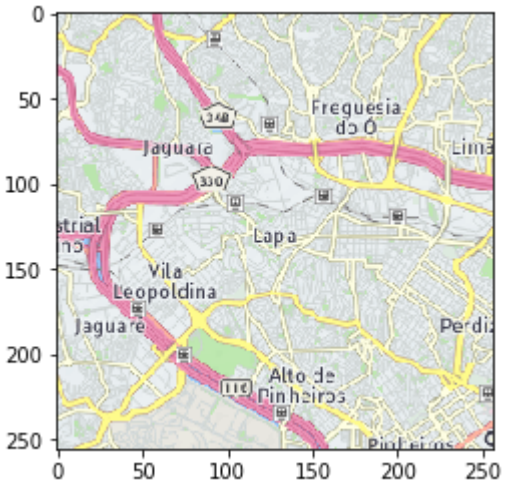
A próxima etapa é converter as imagens que estão em um diretório, para um arquivo *numpy*. Nem todas as bibliotecas do programa abaixo foram utilizadas. Existe uma opção de mudar o tamanho das imagens caso desejado (bastando descomentar a linha correspondente). As imagens que estão em uma estrutura de diretórios são convertidas em um arquivo numpy [n, :, :, :] onde n é o índice da imagem. O arquivo é gravado em "data/map01.npy" para posterior uso

```
import numpy as np
import matplotlib.pyplot as plt
from skimage import io, transform
import os
import glob
import time

def get_class(img_path):
    return int(img_path.split('/')[-2])

# main
time1 = time.time()
root_dir = 'map/tiles/'
imgs = []
```

```
all_img_paths = glob.glob(os.path.join(root_dir, '*/*/*/*/*.png'))
all_img_paths
for img_path in all_img_paths:
    img = io.imread(img_path)
     img = (img / 255.0)  # rescale (senão o n.array float dá mensagem de
aviso)
     #    img = transform.resize(img, (256, 256))
    imgs.append(img)
X = np.array(imgs, dtype='float32')
np.save("data/map01.npy",X)
plt.imshow(X[5, :, :, :].tolist())
plt.show()
print("Elapsed time:", time.time() - time1, "s" )
```




```
Elapsed time: 3.085407018661499 s
```

Programa para execução serial, que converte o arquivo numpy com imagens, para escala de cinza. Existe a opção de gravar um arquivo. A imagem em escala de cinza foi gerada habilitando a opção correspondente. Foi executado serialmente em um nó com

x@node01:/scratch$ python3 gray.py

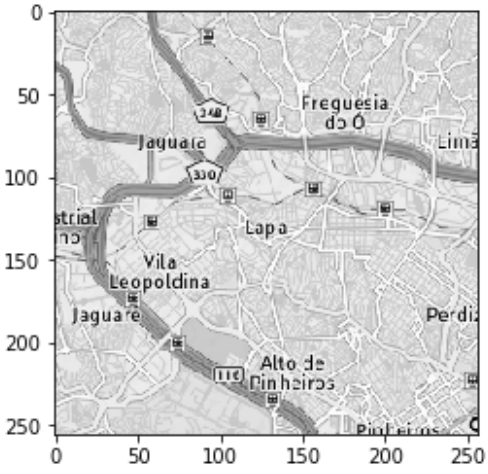
Elapsed time: 10.768640041351318 s

```
import numpy as np
import matplotlib.pyplot as plt
import time

def grayscale_exposure_equalize(batch_X):
```

```
    """Processes a batch with images by grayscaling and normalization
    Args:
        batch_X: a single batch of data containing a numpy array of images.
    Returns:
        Numpy array of processed images.
"""
    X_processed_sub = np.zeros(batch_X.shape[:-1])
    for i in range(len(batch_X)):
        # Grayscale
        img_gray = np.dot(batch_X[i][...,:3], [0.299, 0.587, 0.114])
        # Normalization
        img_gray_norm = img_gray / (img_gray.max() + 1)
        X_processed_sub[i,...] = img_gray_norm
    return (X_processed_sub)

# main
time1 = time.time()
X = np.load("data/map01.npy")
X2 = grayscale_exposure_equalize(X)
# np.save("data/map02.npy",X2)
# plt.imshow(X2[5], cmap='gray')
# plt.show()
print("Elapsed time:", time.time() - time1, "s" )
```



3.3. IMPLEMENTAÇÃO UTILIZANDO PROCESSAMENTO PARALELO

Para a implementação da versão do programa para processamento paralelo utilizou-se a biblioteca mpi4py que fornece ligações Python para MPI (Message Passing Interface) através de uma API baseada em MPI-2 e C++.

Para a compreensão da execução do programa é preciso ter em mente que este mesmo programa vai executar de forma paralela em vários processos diferentes. A comunicação entre os processos será feita através de mensagens MPI. O primeiro processo (Rank 0) fará a leitura do arquivo numpy, divisão em partes, transmissão (via MPI) para os demais processos, e por fim receberá as partes processadas, e unirá tudo em um arquivo final. Existe uma opção de gravar o arquivo, caso desejado, bastando descomentar a linha de código correspondente. Esta forma de divisão das tarefas é relativamente comum, onde um processo (o primeiro, neste caso) assume tarefas adicionais, desde que obviamente não vá interferir muito no tempo de execução, já que o objetivo é dividir as tarefas preferencialmente de forma uniforme.

Como neste caso específico o arquivo de dados é pequeno e pode ser carregado na memória, o Rank 0 faz tudo, lê, envia, recebe, e une as imagens processadas pelos demais Ranks. Já no caso de arquivos maiores uma possível solução seria utilizar os recursos do MPI junto com sistemas de arquivos paralelos como é o caso do Lustre (http://lustre.org/). O Lustre permite configurar acesso paralelo ao arquivo de dados, e junto com o MPI cada processo (Rank) pode acessar o arquivo de forma paralela, tanto para leitura quanto para gravação, e desta forma o Rank 0 não faria mais a comunicação (envio) dos dados, cada Rank faria diretamente a leitura/gravação (via MPI) da sua parte a processar. Dependendo do tamanho do arquivo, é possível também pensar também em subdividir o processamento nos diferentes Ranks, ou seja, cada Rank faria várias leituras/escritas de sub partes a processar.

O Python deve estar instalado nos nós que se pretende executá-lo, assim como as bibliotecas. Por exemplo para instalar as bibliotecas numpy e mpi4py deve-se executar em cada nó:

```
sudo -H pip3 install numpy mpi4py
```

O programa completo encontra-se no Anexo A. A seguir temos trechos extraídos do programa, para explicar o funcionamento. Inicialmente carrega-se as bibliotecas a utilizar

```
import numpy as np
from mpi4py import MPI
```

“MPI.Wtime” serve para medir o tempo de execução, “MPI.COMM_WORLD” é o comunicador que serve por exemplo para definir grupos de execução, “comm.Get_size()” retorna a quantidade de processos (Ranks) em execução, e “comm.Get_rank()” é o número do Rank atribuído ao processo: se por exemplo tivermos 4 processos em execução, os Ranks serão 0, 1, 2, e 3, e a função retornará o Rank correspondente

```
wt = MPI.Wtime()        # "wall time" para cálculo do tempo decorrido
comm = MPI.COMM_WORLD   # comunicador global (pode servir para definir grupos)
cpu = comm.Get_size()   # total de ranks que o mpi atribui
rank = comm.Get_rank()  # rank é o no. que o mpi atribui ao processo
```

“118” é o total de imagens baixadas na etapa anterior, “sseg” vai conter o tamanho de um segmento que um processo vai processar, e “mseg” é o tamanho do maior segmento

```
xlen = 118                              # total de imagens
sseg = int( xlen / cpu )                # tamanho de um segmento
mseg = sseg + ( xlen % cpu )            # tamanho do maior segmento
```

“xsub” é uma área de trabalho temporária, “xprocessed” contém o resultado com as imagens processadas

```
xsub = np.zeros((mseg, 256, 256, 4), dtype=np.float32)     # área de trabalho
xprocessed = np.zeros((xlen, 256, 256), dtype=np.float32)   # resultado
```

A parte inicial desse trecho vai ser executado apenas pelo Rank 0, ele vai carregar o arquivo numpy, dividir e enviar (comm.Send) para cada processo. Para os demais Ranks o “if” desvia para a leitura (comm.Recv) dos dados enviados.

```
if rank == 0 :
    x = np.load("data/map01.npy")       # lê o arquivo com o conjunto de dados
    xbatches = np.array_split(x, cpu)   # divide os dados entre as cpus
    xsub[0:len(xbatches[0])] = xbatches[0]    # segmento que o rank 0 processa
    for i in range(1, cpu) :            # distribui os segmentos
        # quando Send é upper-case usa buffers
        comm.Send(xbatches[i], dest=i, tag=0)    # envia um segmento
else :      # os demais processos (ranks) recebem os segmentos
    comm.Recv(xsub, source=0, tag=0)
```

Aqui cada Rank processa a sua parte e processa

```
# calcula os índices inicial e final de cada segmento, para cada rank
start = 0
if rank == cpu - 1 :            # o último rank
    end = mseg                  # fica com o maior segmento
else :
    end = sseg                  # índice do final do segmento

# todos os ranks processam o seu segmento
# xprocessedsub fica com uma dimensão a menos (mseg, 256, 256)
xprocessedsub = np.zeros(xsub.shape[:-1])
# repete 10x o looping, apenas para fins de medição de tempo
for j in range(0,10) :
    for i in range(start, end) :
        # Grayscale
        ## xsub[i][...,:3] seleciona a imagem (256, 256, 3)
        ## np.dot faz a multiplicação e soma para converter
        img_gray = np.dot(xsub[i][...,:3], [0.299, 0.587, 0.114])
        # Normalization
        img_gray_norm = img_gray / (img_gray.max() + 1)
        xprocessedsub[i,...] = img_gray_norm
```

Após o processamento o Rank 0 não precisa enviar a sua parte pois é ele que está unindo as diversas partes. Os demais Ranks enviam para 0 a sua parte processada

```
# xprocessedsub contém o segmento processado. O shape é (mseg, 256, 256)
# o rank 0 copia direto para o dataset final
if rank == 0 :
    xprocessed[0:len(xprocessedsub)]=xprocessedsub
# os demais ranks retornam o segmento processado para o rank 0
else :
     comm.Send(xprocessedsub, dest=0, tag=rank)      # tag identifica quem
mandou
```

Este trecho final apenas o Rank 0 executa. Ele vai receber as demais partes processadas pelos Ranks, vai unir as partes, e finalmente imprimir a mensagem contendo o tempo decorrido de execução

```
# o rank 0 recebe os segmentos e os combina em um único dataset
#  xprocessedsub do rank 0 já foi copiado e agora serve como armazen.
temporário
if rank == 0 :
    for i in range(1, cpu) :
        status = MPI.Status()
        # recebe um segmento
```

```
            comm.Recv(xprocessedsub, source=MPI.ANY_SOURCE, tag=MPI.ANY_TAG,
status=status)
        rnk_sender = status.Get_source()
             start= rnk_sender * sseg             # índice para a posição
correspondente
        slen = sseg
        # copia para o dataset final
        # essa parte do codigo pode ser melhorada
        xprocessed[start : start + len(xprocessedsub)] = xprocessedsub
    # shape final incluindo o canal, que no caso de grey é 1
    xprocessed.reshape(xprocessed.shape + (1,))
    #grava em um arquivo para uso posterior
    #np.save("data/map03.npy",xprocessed)
    # cada rank mostra o tempo decorrido
    print('Rank =', rank, '  Elapsed time =', MPI.Wtime() - wt, 's')
```

Execução de alguns testes aleatórios fora do ambiente final de testes, para verificar o funcionamento correto do algoritmo implementado

```
$ mpiexec -n 1 python3 pdaempdi.py
Rank = 0   Elapsed time = 18.578750133514404 s
$ mpiexec -n 2 python3 pdaempdi.py
Rank = 0   Elapsed time = 10.840905904769897 s
$ mpiexec -n 3 python3 pdaempdi.py
Rank = 0   Elapsed time = 8.452061176300049 s
$ mpiexec -n 4 python3 pdaempdi.py
Rank = 0   Elapsed time = 1.7179977893829346 s
$ mpiexec -n 10 python3 pdaempdi.py
Rank = 0   Elapsed time = 5.16180682182312 s
$ mpiexec -n 18 python3 pdaempdi.py
Rank = 0   Elapsed time = 15.067718029022217 s
```

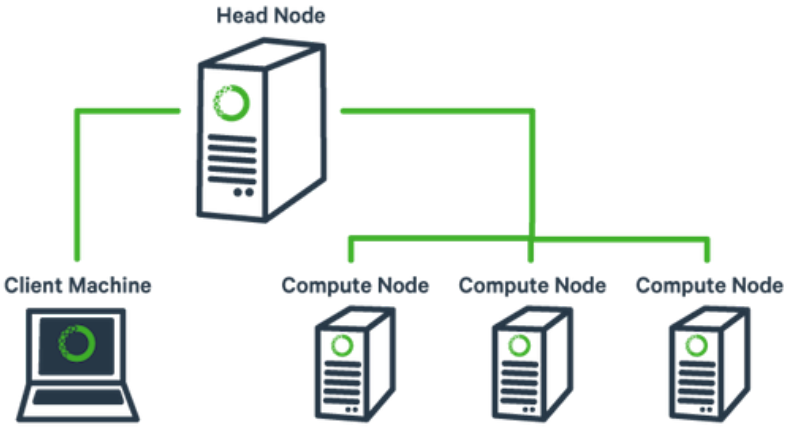
3.4. O AMBIENTE DE TESTES

Para os testes foi configurado um *cluster* com as características:

Nó	Core	Thread	Processador	Instruções	Freq	Cache	RAM	GPU
node01	2	4	Core i7-7500U	avx2, fma3, sse4.2	3.5 GHz	4 MB L3	16 GB	NVIDIA GeForce 940MX 4 GB
node02	4	8	Core i7-2630QM	avx, sse4.2	2.6 GHz	6 MB L3	8 GB	
node03	4	8	Core i7-2630QM	avx, sse4.2	2.6 GHz	6 MB L3	6 GB	

node04	2	2	Core2Duo T7200	ssse3	2 GHz	4 MB L2	2 GB	
node05	2	2	Core2Duo T7200	ssse3	2 GHz	4 MB L2	2 GB	
node06	2	2	Core2Duo T7250	ssse3	2 GHz	2 MB L2	3 GB	
node07	2	4	Core i7-3520M	avx, sse4.2	3.6 GHz	4 MB L3	8 GB	NVIDIA GeForce GT 635M 2 GB
SUB TOTAL	18	30						
master	1	2	Atom N450	ssse3	1.66 GHz	512 kB L2	2 GB	(nó de login)

Os nós rodam Ubuntu 18.04, Python, MPI, NFS, e Slurm.



Fonte: ucdavis-bioinformatics-training.github.io

3.5. EXECUÇÃO DA IMPLEMENTAÇÃO

O programa `padempdi.py` é copiado para o diretório `\scratch` do nó de login. Todos os demais nós têm acesso a este diretório via NFS.

```
$ ssh x@master
x@master's password:
x@master:~$ cd /scratch
x@master:/scratch$
```

Foi criado também um script de submissão `sub_padempdi.sh`. Em `ntasks` colocamos a quantidade de processos desejada. O Slurm vai distribuir entre os nós e processadores:

```
# !/bin/bash
# SBATCH --ntasks=1

cd $SLURM_SUBMIT_DIR

mpiexec -n $SLURM_NTASKS python3 padempdi.py
```

Rodando o arquivo de lote. A saída é direcionada para um arquivo

```
x@master:/scratch$ sbatch sub_padempdi.sh
Submitted batch job 186
x@master:/scratch$ cat slurm-186.out
Rank = 0   Elapsed time = 19.013105583027937 s
x@master:/scratch$
```

Diretório /scratch:

```
x@master:/scratch$ ls -l
total 20
drwxr-xr-x 2 x x 4096 Dec  1 17:43 data
drwxr-xr-x 3 x x 4096 Dec  1 17:41 map
-rw-rw-r-- 1 x x 3879 Dec  1 18:00 padempdi.py
-rw-r--r-- 1 x x   47 Dec  1 18:00 slurm-186.out
-rw-rw-r-- 1 x x   99 Dec  1 18:13 sub_padempdi.sh
x@master:/scratch$
```

Mudando no arquivo de lote o parâmetro para execução com 2 tasks:

```
# !/bin/bash
# SBATCH --ntasks=2

cd $SLURM_SUBMIT_DIR

mpiexec -n $SLURM_NTASKS python3 padempdi.py
```

Executando novamente:

```
x@master:/scratch$ sbatch sub_padempdi.sh
Submitted batch job 187
x@master:/scratch$ cat slurm-187.out
Rank = 0   Elapsed time = 7.49563743697945 s
x@master:/scratch$
```

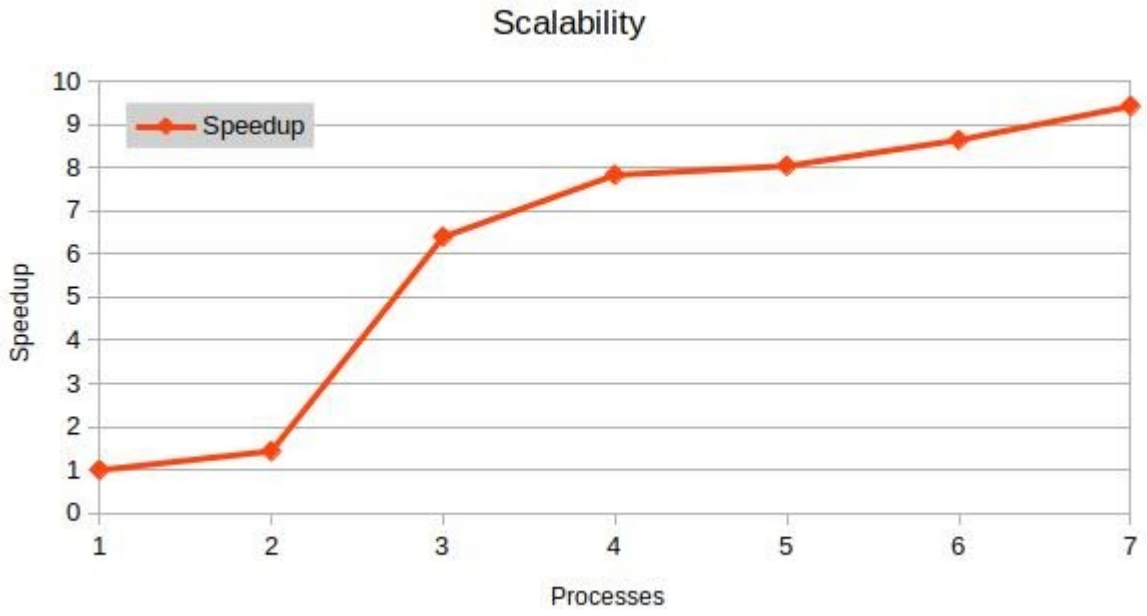
Repetindo os passos acima para 3, 4, 5, 6, e 7 tasks, obtemos:

```
3: 1.6833853269927204 s
4: 1.3756618649931625 s
5: 1.3395628849975765 s
6: 1.2466953669209033 s
7: 1.1426549749448895 s
```

O resultados acima são apresentados no capítulo seguinte.

4. RESULTADOS

O gráfico a seguir mostra os Speedups obtidos rodando com 1, 2, ..., 6, e 7 processos. O Speedup é a relação entre a versão serial e a paralela. A curva mostra o ganho em eficiência com o aumento do número de processos paralelos.



5. DISCUSSÃO

O gráfico acima mostra que o aumento do Speedup é grande até 4 processos, aproximadamente, e a partir daí a inclinação da curva é mais suave, mostrando uma redução do aumento do Speedup. O motivo pode ser porque o Slurm está usando um nó que está com o recurso de Hyperthreading do processador ativado. Ao invés de alocar mais processadores, o Slurm está rodando processos no mesmo processador usando o recurso de Hyperthreading. Como o Hyperthreading

aumenta pouco a capacidade do processador, essa seria uma possível razão para o pequeno aumento do Speedup.

Um ponto importante a considerar seria a leitura de arquivo do disco feita pelo programa, que utiliza buffers do sistema operacional e bibliotecas, e que é tipicamente serial - a princípio, sem utilizar maiores recursos como sistemas de arquivos paralelos, não seria possível paralelizar. Esta parte serial do programa afeta o resultado da paralelização.

Uma possível melhoria neste estudo, e motivo para futura investigação, seria configurar o Slurm para não utilizar o Hyperthreading e com isso ele alocaria um novo processador para os ranks a partir do 5. Isso possivelmente melhorará o resultado do Speedup. Além disso vários outras questões poderiam ser levantadas e investigadas, como por exemplo execução com grandes arquivos de dados.

6. CONCLUSÃO

Foi possível demonstrar que há ganho de desempenho em PDI utilizando-se técnicas de PAD e processamento paralelo em Python. Em um primeiro ensaio obtivemos um Speedup de até 9x com relação ao processamento serial que normalmente seria feito utilizando o programa em Python sem as otimizações.

O campo de PAD é amplo e poderíamos expandir este estudo utilizando-se outras técnicas e várias melhorias no programa.

REFERÊNCIAS

CYTOWSKI, Maciej. Parallel Programing with Python. Disponível em: https://support.pawsey.org.au/documAentation/display/US/Parallel+Programming+with+Python

MPI4PY. Biblioteca MPI para Python. Disponível em: https://mpi4py.readthedocs.io/en/stable/

INSIDEHPC. Site com diversas informações. Disponível em: https://insidehpc.com

ILLINOIS. Site da Universidade de Illinois. Disponível em: https://www.ideals.illinois.edu

NUMPY. Pacote para computação científica. Disponível em: https://numpy.org/

SCIKIT. Coleção de algoritmos para processamento de imagens. Disponível em: https://scikit-image.org/

VAJDA, András. Debugging and performance analysis of many-core programs. In: Programming many-core chips. Springer, Boston, MA, 2011. p. 117-126.

CHILAMKURTHY, Sasank. Keras Tutorial - Traffic Sign Recognition. Sasank's Blog, 5 jan. 2017. Disponível em: https://chsasank.github.io/keras-tutorial.html. Acesso em: 2 dez. 2019.

GABRIEL, Edgar; VENKATESAN, Vishwanath; SHAH, Shishir. Towards high performance cell segmentation in multispectral fine needle aspiration cytology of thyroid lesions. Computer methods and programs in biomedicine, v. 98, n. 3, p. 231-240, 2010.

RATH, Sovit Ranjan. Traffic Sign Recognition using Neural Networks. Debugger cafe, 11 jul. 2019. Disponível em: https://debuggercafe.com/traffic-sign-recognition-using-neural-networks/. Acesso em: 2 dez. 2019.

KIRK, David B.; WEN-MEI, W. Hwu. Programming massively parallel processors: a hands-on approach. Morgan kaufmann, 2016.

CÂMARA, Gilberto; CASANOVA, Marco A.; MAGALHÃES, Geovane C. Anatomia de sistemas de informação geográfica. 1996.

PAGE, Daniel. A practical introduction to computer architecture. Springer Science & Business Media, 2009.

SILVA, Ardemiro de Barros. Sistemas de informações georreferenciadas: conceitos e fundamentos, 2003.

ANEXO A

Códigos fontes dos programas utilizados neste trabalho.

padmpi.py

```
# CAP-378 Trabalho: PAD em PDI
# Uso: mpiexec -n <NTASKS> python3 padempdi.py

import numpy as np
from mpi4py import MPI

wt = MPI.Wtime()        # "wall time" para cálculo do tempo decorrido
comm = MPI.COMM_WORLD   # comunicador global (pode servir para definir grupos)
cpu = comm.Get_size()   # total de ranks que o mpi atribui
rank = comm.Get_rank() # rank é o no. que o mpi atribui ao processo

xlen = 118                                  # total de imagens
sseg = int( xlen / cpu )                    # tamanho de um segmento
mseg = sseg + ( xlen % cpu )                # tamanho do maior segmento

# - 256, 256 é a imagem, e 4 é a qtde de canais.
#  - The different color bands/channels are stored in the third dimension,
such
#   that a gray-image is MxN, an RGB-image MxNx3 and an RGBA-image MxNx4.
# - RGBA = Reg, Green, Blue, Alpha(transparência) → PNG
# - Cinza tem só (256, 256) que corresponde ao tamanho da imagem.
xsub = np.zeros((mseg, 256, 256, 4), dtype=np.float32)     # área de trabalho
xprocessed = np.zeros((xlen, 256, 256), dtype=np.float32)   # resultado

# O processo (rank) 0 lê o arquivo e distribui os segmentos para os ranks
# O rank 0 também processa um segmento
if rank == 0 :
     x = np.load("data/map01.npy")          # lê o arquivo com o conjunto de
dados
    xbatches = np.array_split(x, cpu)    # divide os dados entre as cpus
    xsub[0:len(xbatches[0])] = xbatches[0]   # segmento que o rank 0 processa
    for i in range(1, cpu) :             # distribui os segmentos
        # quando Send é upper-case usa buffers
        comm.Send(xbatches[i], dest=i, tag=0)    # envia um segmento
else :      # os demais processos (ranks) recebem os segmentos
    comm.Recv(xsub, source=0, tag=0)

# calcula os índices inicial e final de cada segmento, para cada rank
start = 0
if rank == cpu - 1 :            # o último rank
    end = mseg                  # fica com o maior segmento
else :
    end = sseg                  # índice do final do segmento
```

```
# todos os ranks processam o seu segmento
# xprocessedsub fica com uma dimensão a menos (mseg, 256, 256)
xprocessedsub = np.zeros(xsub.shape[:-1])
# repete 10x o looping, apenas para fins de medição de tempo
for j in range(0,10) :
    for i in range(start, end) :
        # Grayscale
        ## xsub[i][...,:3] seleciona a imagem (256, 256, 3)
        ## np.dot faz a multiplicação e soma para converter
        img_gray = np.dot(xsub[i][...,:3], [0.299, 0.587, 0.114])
        # Normalization
        img_gray_norm = img_gray / (img_gray.max() + 1)
        xprocessedsub[i,...] = img_gray_norm
# xprocessedsub contém o segmento processado. O shape é (mseg, 256, 256)
# o rank 0 copia direto para o dataset final
if rank == 0 :
    xprocessed[0:len(xprocessedsub)]=xprocessedsub
# os demais ranks retornam o segmento processado para o rank 0
else :
      comm.Send(xprocessedsub, dest=0, tag=rank)      # tag identifica quem
mandou
# o rank 0 recebe os segmentos e os combina em um único dataset
#  xprocessedsub do rank 0 já foi copiado e agora serve como armazen.
temporário
if rank == 0 :
    for i in range(1, cpu) :
        status = MPI.Status()
        # recebe um segmento
            comm.Recv(xprocessedsub, source=MPI.ANY_SOURCE, tag=MPI.ANY_TAG,
status=status)
        rnk_sender = status.Get_source()
             start= rnk_sender * sseg             # índice para a posição
correspondente
        slen = sseg
        # copia para o dataset final
        # essa parte do codigo pode ser melhorada
        xprocessed[start : start + len(xprocessedsub)] = xprocessedsub
    # shape final incluindo o canal, que no caso de grey é 1
    xprocessed.reshape(xprocessed.shape + (1,))
    #grava em um arquivo para uso posterior
    #np.save("data/map03.npy",xprocessed)
    # cada rank mostra o tempo decorrido
    print('Rank =', rank, '  Elapsed time =', MPI.Wtime() - wt, 's')
```

gray. py

```
import numpy as np
# import matplotlib.pyplot as plt
import time

def grayscale_exposure_equalize(batch_X):
    """Processes a batch with images by grayscaling and normalization.
    Args:
        batch_X: a single batch of data containing a numpy array of images.
    Returns:
        Numpy array of processed images.
"""
    X_processed_sub = np.zeros(batch_X.shape[:-1])
    for i in range(len(batch_X)):
        # Grayscale
        img_gray = np.dot(batch_X[i][...,:3], [0.299, 0.587, 0.114])
        # Normalization
        img_gray_norm = img_gray / (img_gray.max() + 1)
        X_processed_sub[i,...] = img_gray_norm
    return (X_processed_sub)

# main
time1 = time.time()
X = np.load("data/map01.npy")
X2 = grayscale_exposure_equalize(X)
# np.save("data/map02.npy",X2)
# plt.imshow(X2[5], cmap='gray')
# plt.show()
print("Elapsed time:", time.time() - time1, "s" )
```

converte.py

```
import numpy as np
import matplotlib.pyplot as plt
from skimage import io, transform
import os
import glob
import time

def get_class(img_path):
    return int(img_path.split('/')[-2])

# main
time1 = time.time()
root_dir = 'map/tiles/'
imgs = []
all_img_paths = glob.glob(os.path.join(root_dir, '*/*/*/*/*.png'))
all_img_paths
for img_path in all_img_paths:
    img = io.imread(img_path)
     img = (img / 255.0)  # rescale (senão o n.array float dá mensagem de
aviso)
     #    img = transform.resize(img, (256, 256))
    imgs.append(img)
X = np.array(imgs, dtype='float32')
np.save("data/map01.npy",X)
plt.imshow(X[5, :, :, :].tolist())
plt.show()
print("Elapsed time:", time.time() - time1, "s" )
```