CAP 378 Tópicos em Observação da Terra

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

Revisão 2019-12-01

Objetivo

O objetivo deste trabalho da disciplina CAP 378 é demonstrar o aumento de performance em conversão de i
técnicas de PDA, em um ambiente de processamento paralelo.

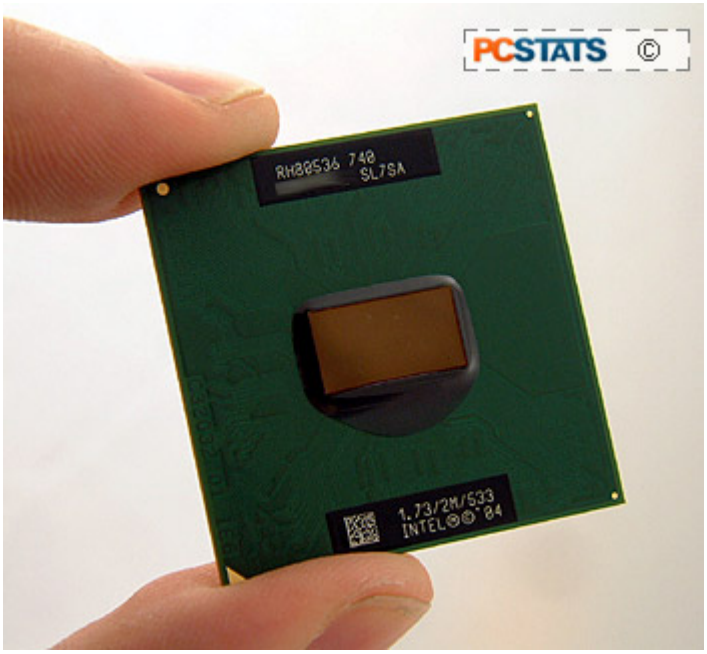
Introdução

PAD (Processamento de Alto Desempenho) pode ser definido como a prática de agregar capacidade de com
desempenho muito maior do que poderia ser obtido normalmente, e possui o objetivo de resolver grandes pr
ou negócios. PAD como conceito não está limitado a supercomputadores ou grandes sistemas, podendo ser
aonde se utilizam técnicas para aproveitar toda capacidade de processamento e aumentar a eficiência.

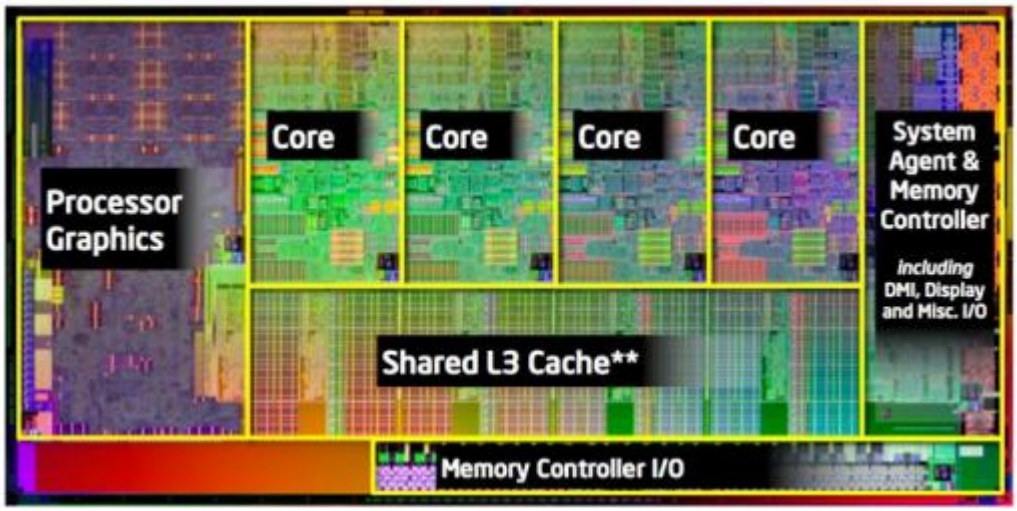
Processamento paralelo

Atualmente os principais sistemas computacionais de alto desempenho utilizam processadores comerciais
para obter desempenho através da computação paralela. Simplificadamente são compostos por muitos proc
trabalhando em conjunto. Programas de alto desempenho são feitos para dividir o processamento entre os v
mesmo tempo, de tal forma a aproveitar a capacidade conjunta do sistema.

Em processamento paralelo um mesmo programa roda em vários núcleos de um processador, que por sua v
execução, e que por sua vez faz parte de um *cluster*.

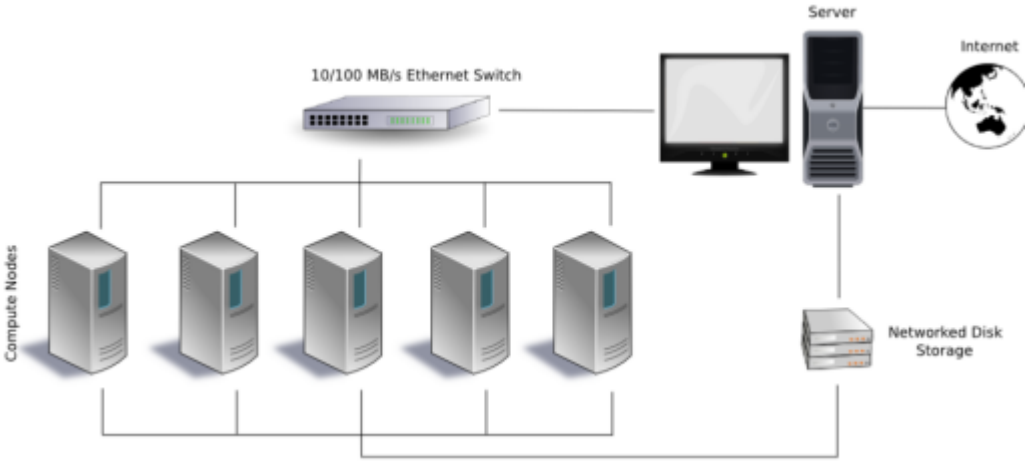


Fonte: http://www.pcstats.com



Fonte: https://images.anandtech.com

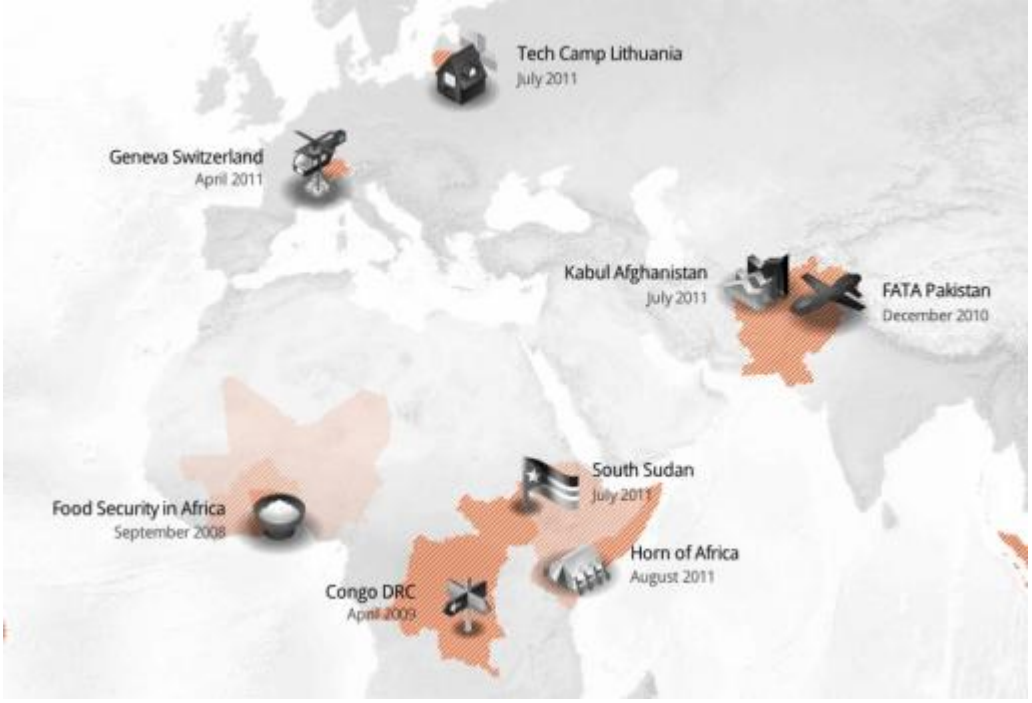
Várias CPUs podem ser unidas, formando um Cluster:



Fonte: https://en.wikipedia.org

Exemplo de aplicação de conversão de imagens

Uma aplicação seria a criação de basemaps em tons de cinza a partir de imagens coloridas.



Fonte: https://gis.stackexchange.com/questions/15146/how-to-emulate-this-grayscale-basemap-lookfeel-in-gc

DESENVOLVIMENTO

O ambiente de programação

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

Python

Python foi escolhido porque é um forte candidato para escrever as partes que são de muito alto nível em apl
processamento de alto desempenho, pois agrega todos os benefícios de uma linguagem de script como rap
desenvolvimento de código e manutenção, além de atuar como uma extensão, integrando bibliotecas escrita
compiladas em um único ambiente fácil de ser usado.

MPI

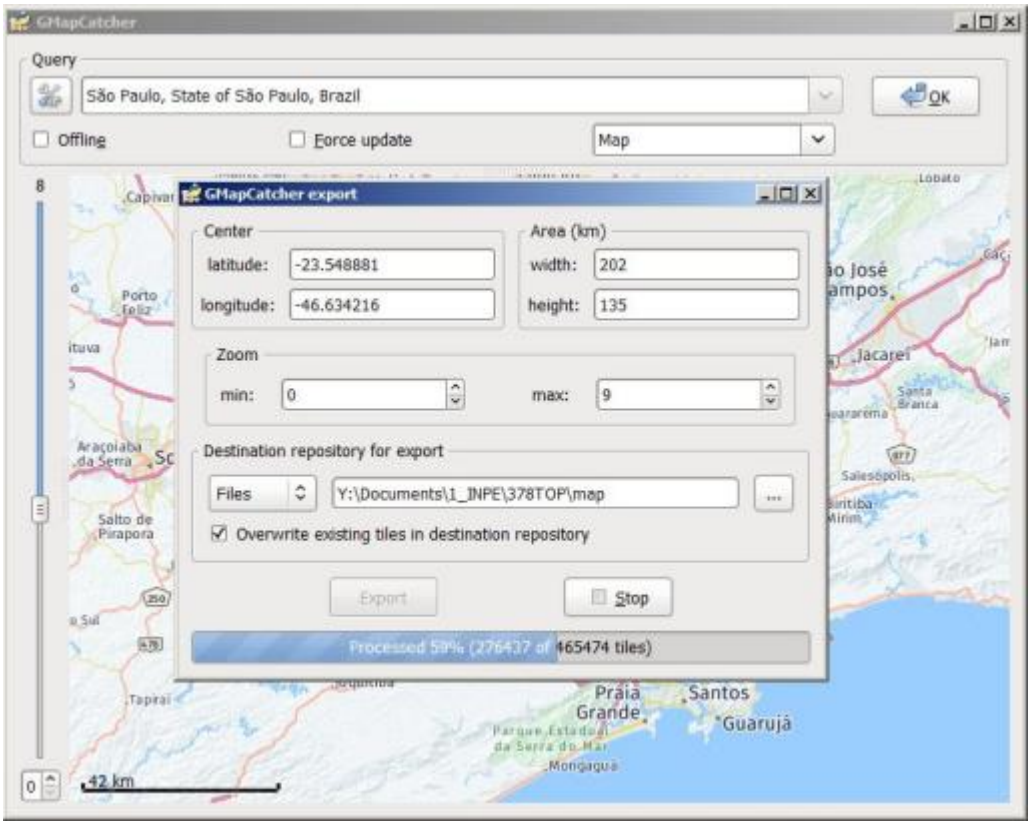
Foi utilizado a biblioteca `mpi4py` que é uma implementação do MPI. O MPI é um padrão aberto para comuni
paralela. O MPI foi escolhido para este trabalho por ter larga aceitação no mercado, com a participação de 4
incluindo vendedores, pesquisadores, desenvolvedores de bibliotecas, e usuários. Suas principais vantagens
flexibilidade.

Slurm

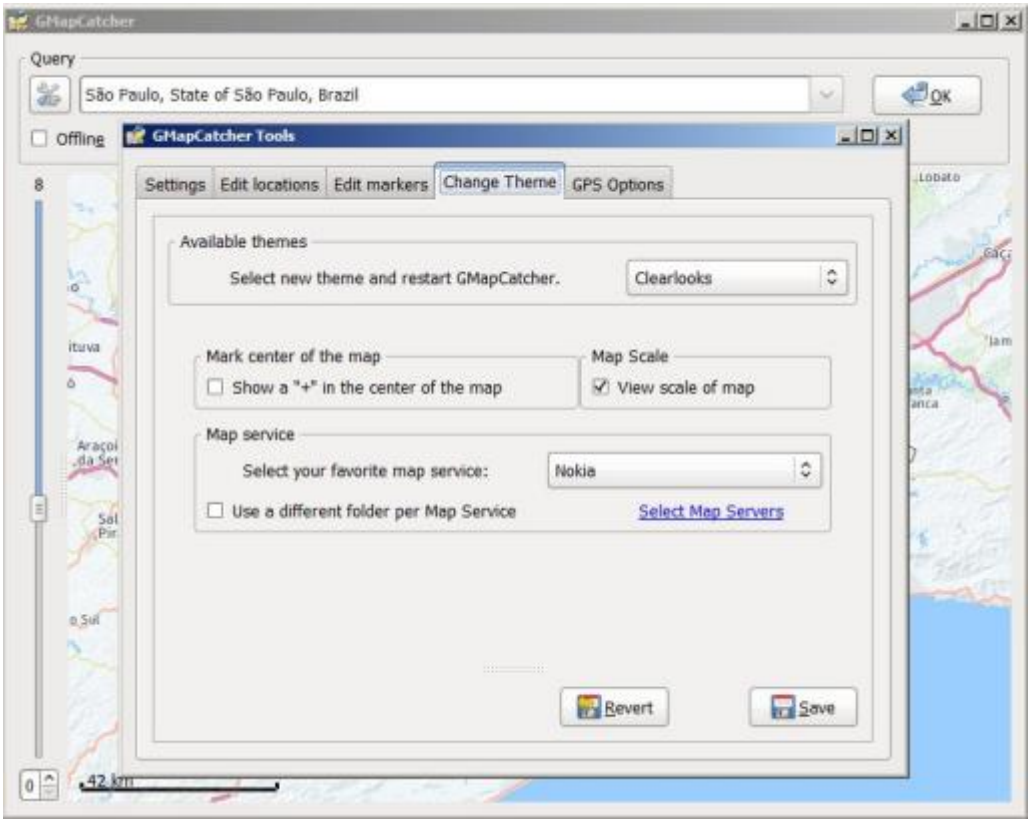
O Slurm é um gerenciador de ambiente de execução, permitindo o escalonamento de tarefas e um gerencian
de *Cluster* e processamento paralelo. Foi escolhido para este trabalho por ser de código aberto, simples de i
utilizado em 60% dos supercomputadores do TOP500 (fonte: Wikipedia).

Download do dataset

O *download* das imagens disponíveis online é feito usando o GMapCather (https://github.com/heldersepu/G



Serviço online Nokia Maps

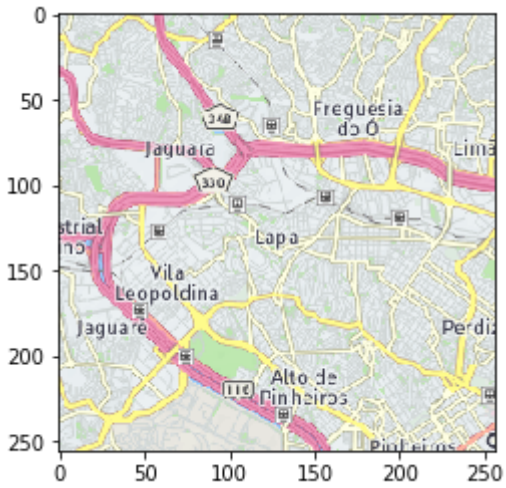


Usando o GMapCather (escrito em Python) navegamos online e as imagens (basemaps) vão sendo baixadas um diretório.

A próxima etapa é converter as imagens para um arquivo `numpy` "map01.npy":

```
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" )
```

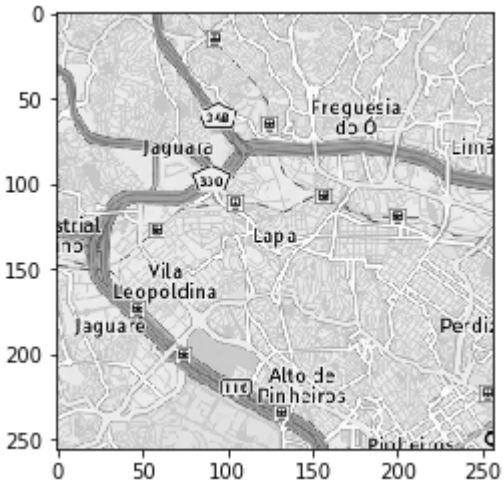



```
Elapsed time: 3.085407018661499 s
```

No programa abaixo é feita a execução serial usando uma função de biblioteca que pega um lote de imagens
Neste caso os parâmetros de conversão (.299, .587, .114) usam o padrão ITU BT.601 https://www.itu.int/rec/
BT.601https://www.itu.int/rec/R-REC-BT.601. Já a normalização melhora o contraste. Escolheu-se apenas es
transformações/conversões, porém caso desejado, pode-se incluir mais.

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

def grayscale_exposure_equalize(batch_X):
    """Processes a batch with images by grayscaling, normalization and histogram equalizati
    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" )
```




```
Elapsed time: 4.093225002288818 s
```

O programa a seguir foi escrito para permitir o processamento paralelo. Ele divide o banco de imagens em pa
processos que o Slurm aloca, processa as partes em cada processador, depois une o resultado em um arqui

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

import numpy as np
import matplotlib.pyplot as plt
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')
```

Alguns testes aleatórios fora do ambiente de testes, somente para verificar se está tudo ok.

```
$ 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
```

Ambiente de testes

Cluster

Nó	Core	Thread	Processador	Instruções	Freq	Cache	RAM	
node01	2	4	Core i7-7500U	avx2, fma3, sse4.2	3.5 GHz	4 MB L3	16 GB	NVIDIA GeFo
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 GeFo
SUBTOTAL	18	30						
master	1	2	Atom N450	ssse3	1.66 GHz	512 kB L2	2 GB	(nó de login)

Rodando o programa

O programa `padempdi.py` é copiado para o diretório "\scratch" do nó de login. Todos os demais nós têm ac

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

Cria-se também um script de submissão `sub_padempdi.sh`. Em `ntasks` coloca-se a quantidade de process
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

```
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 `ntasks` para 2. O Slurm vai alocar dois processos disponíveis

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

cd $SLURM_SUBMIT_DIR

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

Rodando de novo

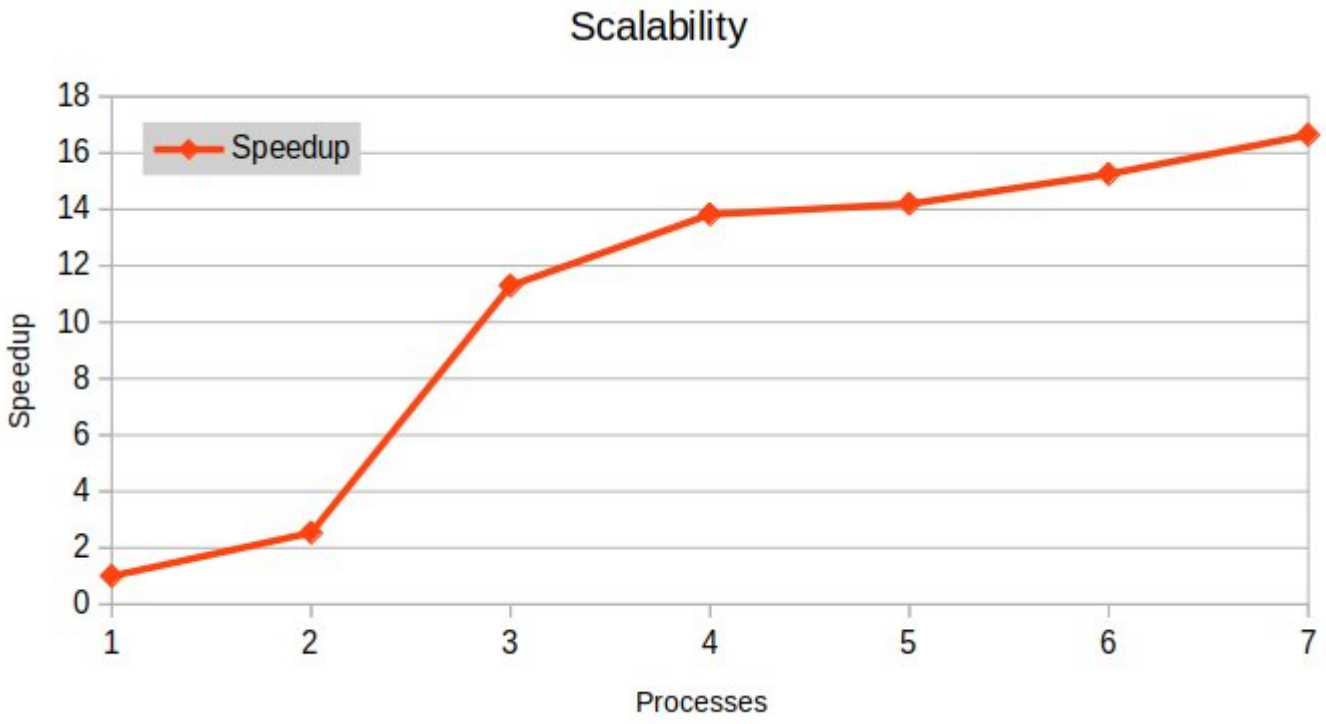
```
x@master:/scratch$ sbatch sub_padempdi.sh
Submitted batch job 187
x@master:/scratch$ cat slurm-187.out
Rank = 0   Elapsed time = 7.49563743697945 s
x@master:/scratch$
```

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
```

Resultados obtidos

O gráfico a seguir mostra os Speedups obtidos rodando com 1, 2, ..., 6, e 7 processos.



Discussão dos resultados

No gráfico acima percebe-se que o aumento do Speedup é grande até 4 processos, aproximadamente, e a pa
mais suave, mostrando uma redução do aumento do Speedup. O motivo pode ser porque o Slurm está usand
de Hyperthreading do processador ativado. Ao invés de alocar um núcleo novo para cada processo, o Slurm
de 4 em um mesmo núcleo utilizando o recurso de Hyperthreading do processador. Como o Hyperthreading
processador, essa seria uma possível razão para o pequeno aumento do Speedup.

Uma possível melhoria neste estudo, e motivo para futura investigação, seria configurar o Slurm para não uti
ele alocaria um novo processador para os ranks a partir do 5. Isso possívelmente melhorará o resultado do S

Conclusão

Demonstrou-se que é possível obter desempenho em PDI utilizando-se técnicas de PAD e processamento pa
ensaio foi obtido um Speedup de até 16x com relação ao processamento serial que normalmente seria feito
sem as otimizações.

O campo de PAD é amplo e poderíamos expandir este estudo utilizando-se outras técnicas e várias melhoria

Referências

- https://support.pawsey.org.au/documAentation/display/US/Parallel+Programming+with+Python
- https://mpi4py.readthedocs.io/en/stable/
- https://insidehpc.com
- https://www.ideals.illinois.edu
- https://numpy.org/
- https://scikit-image.org/
- https://towardsdatascience.com/convnets-series-image-processing-tools-of-the-trade-36e168836f0c
- https://chsasank.github.io/keras-tutorial.html
- https://debuggercafe.com/traffic-sign-recognition-using-neural-networks/