Parallélisme

TP

Programmation MPI - Python

Elana Courtines
courtines.e@gmail.com
https://github.com/irinacake

Séance 1 - 12 octobre 2022
Séance 1 - 19 octobre 2022

Nicolas Mellado - nicolas.mellado@univ-tlse3.fr

Disclaimer :
Seul le dernier exercice (exercice 7) de ce TP est évalué.
De ce fait, il ne sera pas partagé dans ce document.
Du moins pas avant la fin des deadlines de tous les groupes...
Disclaimer 2 :
Seuls les exercices "supposément terminés" seront partagés.
Ceux-ci seront ajoutés après que tous les TPs de tous les groupes aient eu lieu.

Exercice 2 : Monte Carlo Simulation to compute π

```python
#! /usr/bin/python3
# mpirun --oversubscribe -n 2 python3 ./mpi_monte_carlo.py

import time
import random
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

if __name__ == '__main__':
    nb = 15000000
    inside = 0
    random.seed(rank) #use the rank as the seed so they don't all do the same thing

    comm.barrier()
    start_time = time.time()
    for _ in range(nb//size): #reduce the amount of calculation depending on the amount of processes
        x = random.random()
        y = random.random()
        if x*x + y*y <= 1:
            inside +=1
    end_time = time.time()

    if rank == 0: #make sure that pi is initialized for every processes
        pi = 0
    else:
        pi = None

    pi = comm.reduce(inside, op=MPI.SUM, root=0)  #p0 retrieves everything in a reduce-sum

    if rank == 0:
        print("[",rank,"] Pi =", 4 * pi/nb, "in ", end_time-start_time, 'seconds')
```

mpi_monte_carlo.py

Exercice 3 : Contrast stretching

```
# mpirun --oversubscribe -n 8 python3 ./mpi_stretching.py
# mpirun -n 4 python3 ./mpi_stretching.py
from mpi4py import MPI
import sys
import random
import math
import time
import numpy as np
import matplotlib.image as mpimg
from scipy import misc
import matplotlib.pyplot as plt
import matplotlib.cm as cm

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()
```

mpi_stretching.py

```
  # load the image
  if rank == 0:
    print("Starting stretching...")
    pixels, nblines, nbcolumns = readImage()
    splitedPixels = split(pixels,size)
  else:
    splitedPixels, pixels = None,None

  comm.barrier()
  start_time = time.time()
  localPixels = comm.scatter(splitedPixels, root=0)

  # compute min and max of pixels
  local_pix_min = min(localPixels)
  local_pix_max = max(localPixels)
  pix_min = comm.allreduce(local_pix_min, op=MPI.MIN)
  pix_max = comm.allreduce(local_pix_max, op=MPI.MAX)
  # compute alpha, the parameter for f_* functions
  alpha = 1+(pix_max - pix_min) / M

  # stretch contrast for all pixels. f_one and f_two are the two different methods
  if rank%2 == 0:
    for i in range(0,len(localPixels)):
      localPixels[i] = f_one(localPixels[i], alpha)
  else:
    for i in range(0,len(localPixels)):
      localPixels[i] = f_two(localPixels[i], alpha)
  pixels = comm.gather(localPixels, root=0)
  end_time = time.time()
  # save the image
  if rank == 0:
    newPixels = []
    for i in range(size):
      newPixels = newPixels+pixels[i]

    saveImage(newPixels, nblines, nbcolumns)
    print("Stretching done in ", end_time-start_time, 'seconds')
```

mpi_stretching.py

Exercice 4 : Maximum number of divisors

Version 1 :

```
# mpirun -n 4 python3 ./mpi_primes1.py

import sys
import time
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

def nb_primes(n):
    result = 0
    for i in range(1, n+1):
        if n%i == 0:
            result += 1
    return result

if __name__ == '__main__':
    upper_bound = int(sys.argv[1])

    current_max = 0

    comm.barrier()
    start_time = time.time()

    debut = rank * (upper_bound // size)
    fin = (rank+1) * (upper_bound // size)

    print("[",rank,"] debut :", debut, " fin ", fin)

    if rank == size - 1:
        for val in range(debut, upper_bound+1):
            tmp = nb_primes(val)
            current_max = max(current_max, tmp)
    else :
        for val in range(debut, fin):
            tmp = nb_primes(val)
            current_max = max(current_max, tmp)

    global_max = comm.reduce(current_max, op = MPI.MAX, root=0)

    end_time = time.time()

    if rank == 0:
        print("[", rank, "] global max :", global_max, " in ", end_time-start_time, 'seconds')
```

mpi_primes1.py

Version 2 :

```
# mpirun -n 4 python3 ./mpi_primes2.py

import sys
import time
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

def nb_primes(n):
    result = 0
    for i in range(1, n+1):
        if n%i == 0:
            result += 1
    return result

if __name__ == '__main__':
    upper_bound = int(sys.argv[1])

    current_max = 0

    comm.barrier()
    start_time = time.time()

    val = rank+1
    while val < upper_bound+1:
        #print("[",rank,"] val :",val)
        tmp = nb_primes(val)
        current_max = max(current_max, tmp)
        val = val + size

    global_max = comm.reduce(current_max, op = MPI.MAX, root=0)

    end_time = time.time()

    if rank == 0:
        print("[", rank, "] global max :", global_max, " in ", end_time-start_time, 'seconds')
```

mpi_primes2.py

Exercice 5 : Unreachable part of graphs

```
# mpirun -n 4 python3 ./mpi_graph_cut.py
from hashlib import new
from matplotlib import pyplot as plt
import networkx as nx
import time
from mpi4py import MPI
import math

# warnings suppression
import warnings
def fxn(): # this function's sole purpose is to supress 'deprecated' warnings
    warnings.warn("deprecated", DeprecationWarning)

def split(x, qt): # provided in a previous exercice
  n = math.ceil(len(x) / qt)
  return [x[n*i:n*(i+1)] for i in range(qt-1)]+[x[n*(qt-1):len(x)]]

def plot_graph(graph, save=False, display=True): # provided
    g1=graph
    plt.tight_layout()
    nx.draw_networkx(g1, arrows=True)
    if save:
        plt.savefig("graph.png", format="PNG")
    if display:
        plt.show(block=True)

# warning suppresion context
with warnings.catch_warnings():
    warnings.simplefilter("ignore")
    fxn()

    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()

    if rank == 0: # only processus 0 initiates the graph
        graph = nx.gnr_graph(10000, .01).reverse()
    else :
        graph = None
    # every processes require the graph to compute
    graph = comm.bcast(graph,root=0)

    new_elements = [0] # We start at the root (node = 0)
    old_elements = []  # We initialize the already seen nodes

    comm.barrier() # time accuracy
    start_time = time.time()
```

mpi_graph_cut.py

```
while len(new_elements) != 0: # as long as we have new node
    if rank == 0: # nodes to be computed have to be dynamically assigned every iteration
        splitted_elements = split(new_elements,size)
    else:
        splitted_elements = None
    # each process has its own set of nodes to compute
    my_elements = comm.scatter(splitted_elements, root=0)

    tmp = []
    if (len(my_elements) != 0 ):
        for node_src in my_elements: # we take all these nodes
            for node in graph.neighbors(node_src): # we take all their descendents
                if not node in old_elements and not node in my_elements and not node in tmp:
                    # If the descendent is not already seen, we keep it
                    tmp.append(node)

    # have all processes update the old elements because they all need it anyway
    old_elements += new_elements

    new_elements = [] # we are appending to an empty list
    all_tmp = comm.allgather(tmp) # every processes MUST have the updated loop condition
    for i in range(len(all_tmp)): # so have them all do this operation
        if all_tmp[i] not in new_elements: # avoid duplicates !!
            new_elements += all_tmp[i] # these are the new node, we will see them on the next iteration
    # with graphs becoming excessively big (>10^10?), it could be interesting to parallelize this loop

end_time = time.time()

if rank == 0:
    print("[", rank, "] Result :", len(old_elements) == len(graph), " in ", end_time-start_time, 'seconds')
    # don't try to plot a graph with thousands of nodes (:
    #plot_graph(graph, save=True, display=True)
```

mpi_graph_cut.py

Exercice 6 : Heat propagation

```
# mpirun -n 4 python3 ./mpi_heat_cut.py
import time
import random
import functools
import matplotlib.pyplot as plt
from mpi4py import MPI

# warnings suppression
import warnings
def fxn():
    warnings.warn("deprecated", DeprecationWarning)

def init_matrix(size_x, size_y):
    result = []
    for _ in range(size_x):
        result.append([0]*size_y)
    return result

def print_matrix(matrix):
    size_x = len(matrix)
    size_y = len(matrix[0])
    for y in range(size_y):
        for x in range(size_y):
            print(matrix[x][y], end=' ')
        print()

def add_hot_spots(matrix, number):
    size_x = len(matrix)
    size_y = len(matrix[0])

    for i in range(number):
        x = random.randrange(1, size_x-1)
        y = random.randrange(1, size_y-1)
        matrix[x][y] = random.randint(500, 1000)


def get_val(matrix, x, y):
    tmp = matrix[x][y] + matrix[x-1][y] + matrix[x+1][y] + matrix[x][y-1] + matrix[x][y+1]
    return tmp // 5


def get_signature(matrix):
    return functools.reduce(lambda a,b: a^b, [sum(col) for col in matrix])
```

mpi_heat_cut.py

```
# warning suppresion context
with warnings.catch_warnings():
    warnings.simplefilter("ignore")
    fxn()

    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()

    random.seed(7)
    n = 1000

    if rank == 0: # only process 0 initiate the matrix
        matrix = init_matrix(n, n)
        add_hot_spots(matrix, 400)
    else:
        matrix = None

    debut = rank * (n // size)      # every process calculates what part of the matrix
    fin = (rank+1) * (n // size)    # it has to compute

    tmp_matrix = init_matrix(n, n)  # every process need its own tmp_matrix
    matrix = comm.bcast(matrix,root=0) # but they all need the base matrix

    init_time = time.time()
    for _ in range(20): # difference cases based on world size a process number
        if size == 1:               # if there is only 1 process
            for x in range(debut+1, fin-1): # skip rows 0 and n-1
                for y in range(1, n-1):     # skip columns 0 and n-1
                    tmp_matrix[x][y] = get_val(matrix, x, y) # compute
        elif rank == size - 1:      # if it is the last process
            for x in range(debut, fin-1):   # skip row n-1
                for y in range(1, n-1):     # skip columns 0 and n-1
                    tmp_matrix[x][y] = get_val(matrix, x, y) # compute
        elif rank == 0:             # if it is the first process
            for x in range(debut+1, fin):   # skip row 0
                for y in range(1, n-1):     # skip columns 0 and n-1
                    tmp_matrix[x][y] = get_val(matrix, x, y) # compute
        else:                       # every other processes
            for x in range(debut, fin):     # don't skip any row
                for y in range(1, n-1):     # skip columns 0 and n-1
                    tmp_matrix[x][y] = get_val(matrix, x, y) # compute
        # every processes Must receive the updated matrix
        matrix = comm.allreduce(tmp_matrix[debut:fin])

    final_time = time.time()
    if rank == 0:
        print('Total time:', final_time-init_time, 's')
        print('Signature:', get_signature(matrix))
        plt.imshow(matrix, cmap='hot', interpolation='nearest')
        plt.colorbar()
        plt.savefig('heat.pdf')
        plt.show(block=True)
```

mpi_heat_cut.py