Metody Programowania Równoległego
Raport I - OpenMPI

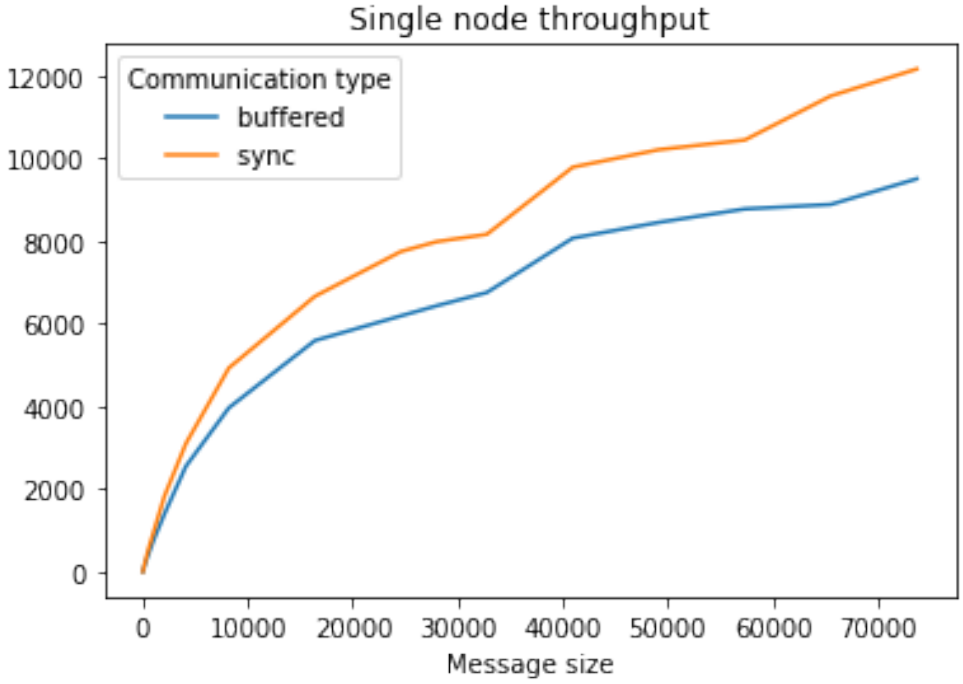
Maciej Trątnowiecki

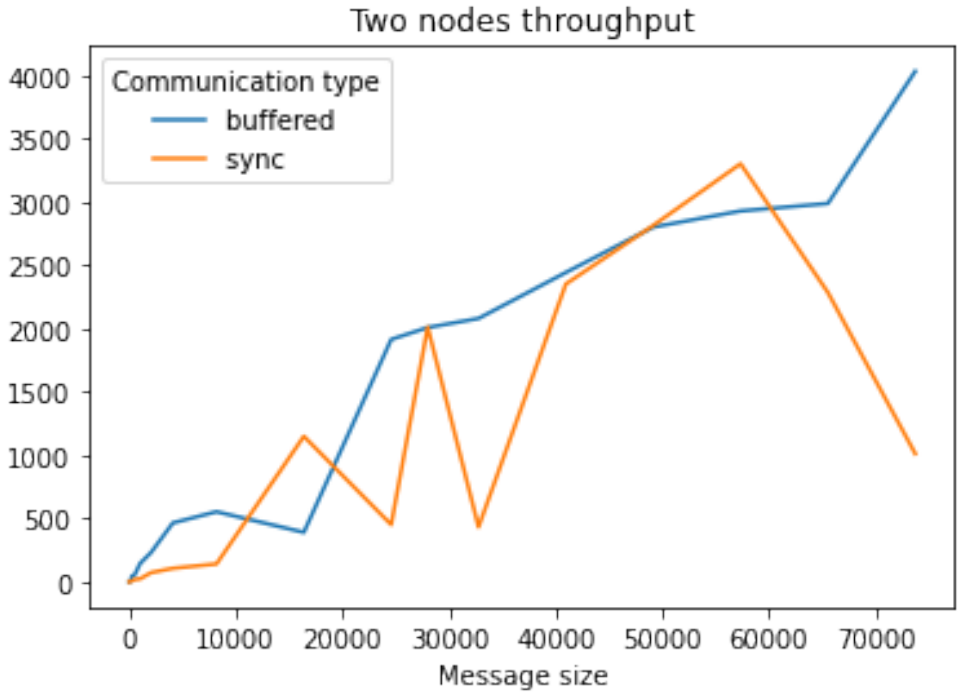
AGH, Semestr Letni, 2022

1 Pomiar przepustowości w komunikacji pomiędzy węzłami klastra

1.1 Pomiary i wyniki

W celu przeprowadzenia doświadczenia przygotowałem skrypt w języku Python, którego kod załączam poniżej. Program przesyła komunikaty pomiędzy dwoma procesami OpenMPI, predefiniowaną ilość razy, następnie liczy przepustowość połączenia. W celu pomiaru przepustowości wielokrotnie uruchomiłem program dla różnych rozmiarów wiadomości i różnych sposobów komunikacji - buforowanej i synchronicznej.
Zebrane wyniki pomiarów przedstawiłem w poniższych wykresach.





W pierwszym przypadku komunikacja odbywała się pomiędzy dwoma procesami uruchamianymi na tym samym węźle klastra, w drugim pomiędzy procesami z różnych węzłów. Zaobserwować możemy, że w przypadku gdy procesy wykonują się na różnych węzłach komunikacja buforowana pozwala uzyskać stabilniejszą przepustowość dla wiadomości o sporym rozmiarze. W przypadku gdy komunikują się procesy z tego samego węzła, komunikacja synchroniczna pozwala na uzyskanie mniejszego narzutu czasowego.

Opóźnienie w przesyłaniu wiadomości, zmierzone jako przepustowość dla wiadomości o rozmiarze jednego bajta wyniosło odpowiednio:

Rodzaj komunikacji	Typ wiadomości	Przepustowość [Mbit/s]
Jeden węzeł	buforowana	1.105
Jeden węzeł	synchroniczna	1.37
Różne węzły	buforowana	0.1468
Różne węzły	synchroniczna	0.0407

Tabela 1: Opóźnienie w przesłaniu wiadomości.

1.2 Kod programu

```
#!/usr/bin/env python
from mpi4py import MPI
import numpy as np
import os
from sys import argv


MSG_N = 1000

if argv[1] == 'buff':
    BUFF_SIZE = 84000
else:
    BUFF_SIZE = 0

MSG_SIZE = int(argv[2])

FIRST_NODE = 0
SECOND_NODE = 1

```

```
def send(comm, *args, **kwargs):
    if BUFF_SIZE > 0:
        comm.bsend(*args, **kwargs)
    else:
        comm.ssend(*args, **kwargs)


def attach_buffer():
    if BUFF_SIZE > 0:
        buf = np.empty((BUFF_SIZE * MSG_N,))
        MPI.Attach_buffer(buf)


def detach_buffer():
    if BUFF_SIZE > 0:
        MPI.Detach_buffer()


def get_message():
    return os.urandom(MSG_SIZE)


def first_node(comm):
    data = get_message()
    for i in range(MSG_N):
        send(comm, data, dest=SECOND_NODE)
        data_rec = comm.recv(source=SECOND_NODE)
        assert data == data_rec


def second_node(comm):
    for i in range(MSG_N):
        data = comm.recv(source=FIRST_NODE)
        send(comm, data, dest=FIRST_NODE)


if __name__ == '__main__':
    comm_world = MPI.COMM_WORLD
    rank = comm_world.Get_rank()

    attach_buffer()
    comm_world.Barrier()
    time=MPI.Wtime()

    if rank == FIRST_NODE:
        first_node(comm_world)
    elif rank == SECOND_NODE:
        second_node(comm_world)
    else:
        print "Expected only two nodes"

    time=MPI.Wtime()-time
    detach_buffer()

    if rank == FIRST_NODE:
        capacity = float(MSG_SIZE*MSG_N*8*2)/time
        print 'buffered' if BUFF_SIZE > 0 else 'sync', ';', MSG_SIZE, ';', MSG_N, ';', capacity

    MPI.Finalize()
```

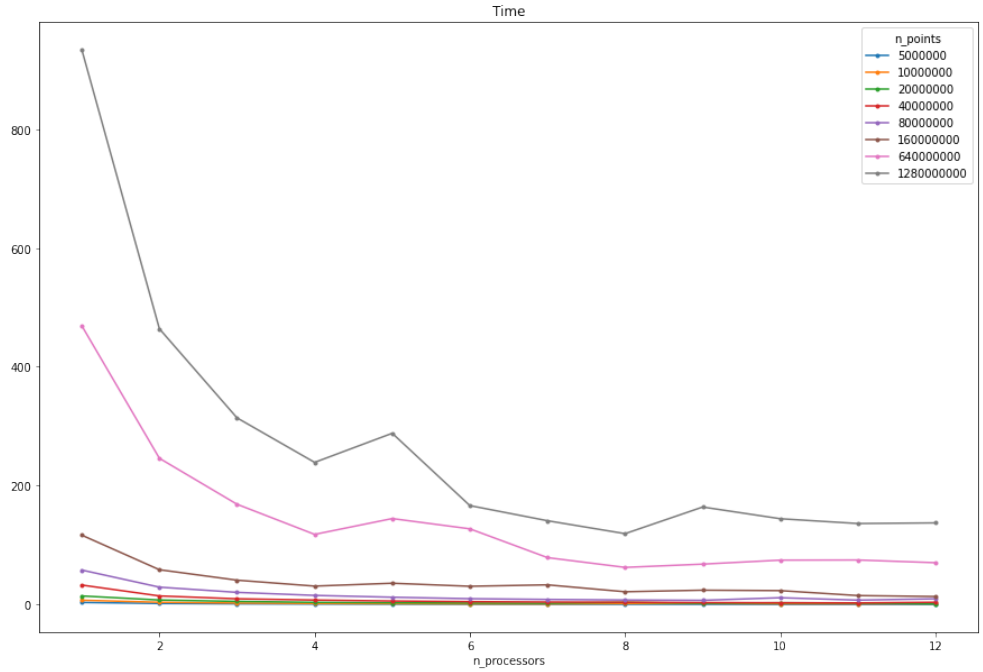
../ex1/ping_pong.py

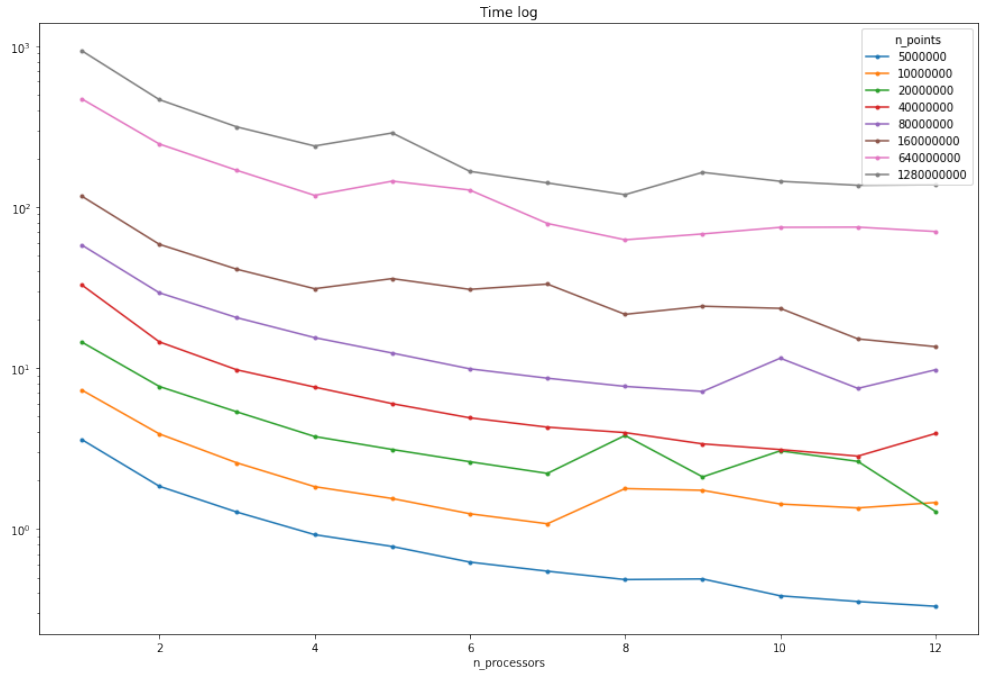
2 Przybliżenie liczby PI metodą Monte Carlo

W ramach eksperymentu przygotowałem skrypt obliczający przybliżenie liczby PI metodą Monte Carlo w sposób równoległy.

2.1 Zależność czasu od liczby procesorów

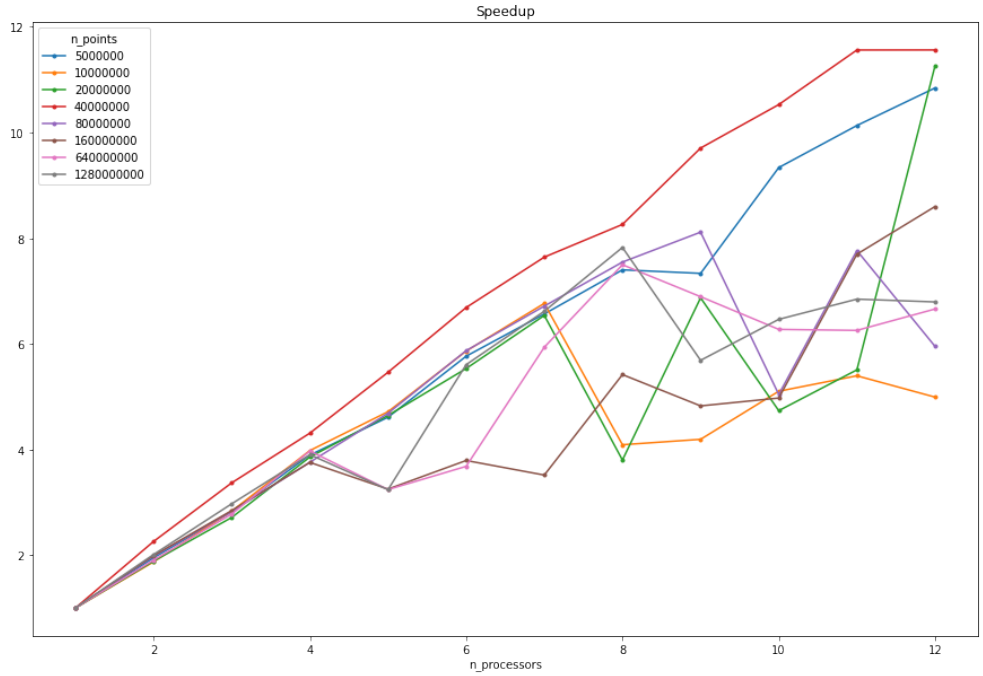
Skrypt uruchomiłem dla problemów o rosnącej złożoności dla liczby procesorów z zakresu 1 do 12 i zmierzyłem czasy wykonania. Poniższe wykresy przedstawiają zmianę czasu wykonania programu w zależności od liczby procesów. Drugi z wykresów przedstawiony został w skali logarytmicznej.





2.2 Przyśpieszenie

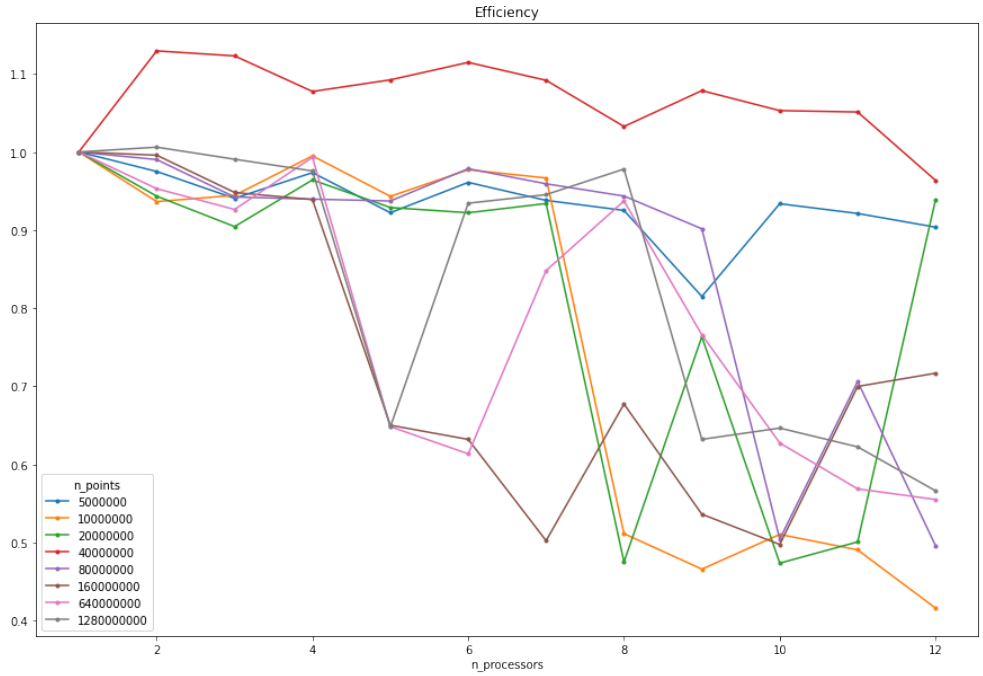
Na podstawie otrzymanych wcześniej pomiarów czasu wykonania programu, przygotowałem wykres przyśpieszenia w zależności od liczby procesorów dla problemów o różnym rozmiarze. Przyśpieszenie obliczyłem jako stosunek czasu wykonania programu bez współbieżności dla otrzymanych wyników dla wyższej liczby procesów.



Zauważyć możemy, że dla większości rozmiarów problemu wykres przyśpieszenia odbiega od teoretycznie idealnego liniowego wzrostu y=x.

2.3 Efektywność

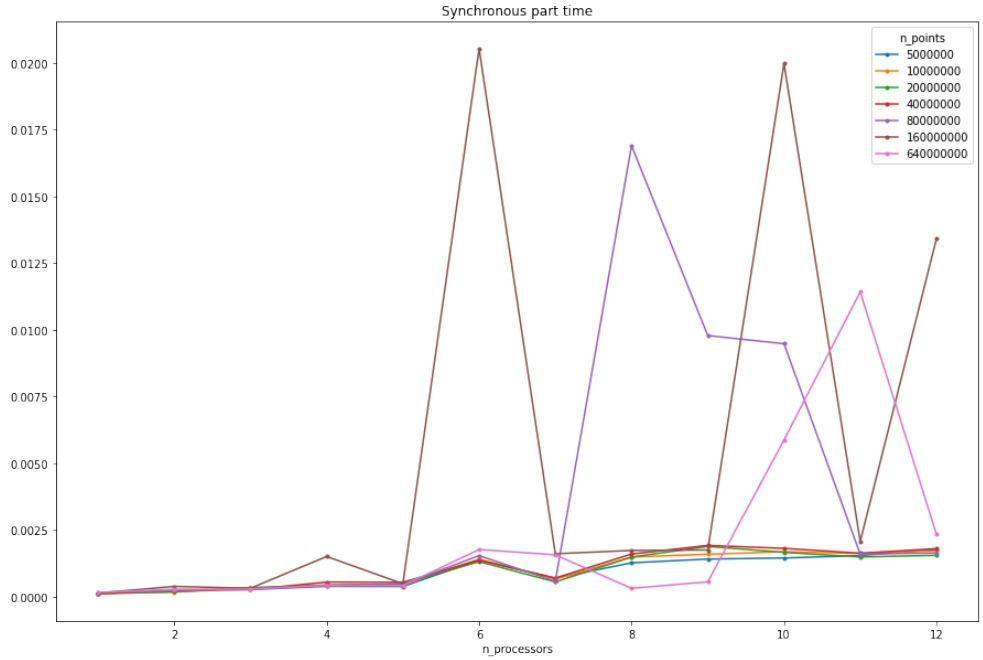
Na podstawie otrzymanych w poprzednim kroku przyśpieszeń, obliczyłem wykres efektywności jako stosunek przyśpieszenia do ilości procesów.



Wbrew oczekiwaniom, zaobserwować możemy spadek wydajności nawet dla problemów o znaczącym rozmiarze.

2.4 Pomiar czasu wykonania części synchronicznej

Wykonałem także pomiar części synchronicznej programu - odpowiedzialnej za zbieranie wyników obliczeń. Wyniki przedstawiłem na poniższym wykresie.



Zauważyć możemy, że czas poświęcony na zbieranie wyników w ogólności nie rośnie wraz ze wzrostem rozmiaru problemu obliczeniowego. Wynika to z faktu, że zbieranym wynikiem jest jedynie liczba punktów wylosowanych wewnątrz koła, dlatego ilość danych nie rośnie wraz ze wzrostem problemu.

2.5 Kod programu

```
#!/usr/bin/env python
from mpi4py import MPI
import numpy as np
import os
from sys import argv
import random


N_POINTS = int(argv[1])
N_PROC = int(argv[2])

def get_message():
    return os.urandom(MSG_SIZE)


def point():
    return (random.random(), random.random())


def point_in_circle(point):
    x, y = point
    return (x * x + y * y) < 1


def node(comm):
    res = 0
    node_points = N_POINTS / comm.Get_size()
    i = 0
    while i < node_points:
        if point_in_circle(point()):
            res += 1
        i+=1
    return res


if __name__ == '__main__':
    ROOT_NODE = 0

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

```
    comm.Barrier()
    time=MPI.Wtime()

    node_res = node(comm)

    comm.Barrier()
    time_calc=MPI.Wtime()-time
    time = MPI.Wtime()

    global_sum = 0
    global_sum = comm.reduce(node_res, global_sum, op=MPI.SUM, root=ROOT_NODE)


    time=MPI.Wtime()-time

    if rank == ROOT_NODE:
        print N_PROC, ';', global_sum, ';', N_POINTS, ';', 4.0 * float(global_sum)/float(
    N_POINTS), ';', time_calc, ';', time

    MPI.Finalize()
```

../ex2/pi.py

```
#!/bin/bash -l
#SBATCH --nodes 1
#SBATCH --ntasks 12
#SBATCH --time=01:00:00
#SBATCH --partition=plgrid
#SBATCH --account=plgmpr22
#SBATCH --sockets-per-node=2

module add plgrid/tools/openmpi
module spider mpi4py
module add plgrid/libs/python-mpi4py/3.0.0-python-2.7

for points in 1000000 36000000 1280000000 ; do
        for n in 1 2 3 4 5 6 7 8 9 10 11 12 ; do
                echo n $n >> res.txt
                mpiexec -np 12 ./pi.py $points >> res.txt
        done
done
```

../ex2/script.sh