Sistemas Distribuídos – COS 470
2016/1
Trabalho Prático 3

Débora Barbosa Pina
Patrícia de Andrade Kovaleski

Decisões de Projeto e Implementação

A linguagem de programação escolhida para a realização do trabalho foi *Python*. Optamos pela utilização de uma linguagem de mais alto nível, ao contrário dos trabalhos anteriores, implementados em C, buscando exatamente uma comparação de desempenho, além do benefício das bibliotecas disponíveis em *Python* para a execução deste trabalho, muito mais simples que as bibliotecas disponíveis em C, por exemplo.

Os algoritmos desenvolvidos nos exercícios descritos a seguir foram testados em dois sistemas operacionais: OSX e Windows. Neste relatório os resultados apresentados para o **primeiro exercício** foram obtidos no Windows, enquanto que os resultados para o **segundo exercício** foram obtidos no OS X. Suas implementações encontram-se disponíveis no *GitHub*: https://github.com/patykov/SD_2016.1 e também anexadas a este relatório.

1. Operações aritméticas em vetores com RPC

Para a implementação de *multithreaded* necessária no primeiro exercício, utilizamos o módulo ***threading***, nativo da linguagem. Já para a implementação das chamadas RPC (*Remote Procedure Call*) utilizamos, a princípio, o módulo **SimpleXMLRPCServer**, que fornece um *framework* de servidor básico para servidores XML-RPC. Porém, ao implementarmos o exercício, nos deparamos com dificuldades internas ao *Python*, descritas a seguir:

i) A expectativa ao utilizarmos a abordagem *multithreading* é que o desempenho seja melhorado, porém, logo em nossos primeiros testes verificamos que ocorria o oposto do esperado. A queda do desempenho ao aumentarmos o número de *threads* é devido a um processo conhecido como *Global Interpreter Lock* (GIL), presente na implementação principal da linguagem *Python*, o *CPython*. O GIL é um *mutex* que impede que as múltiplas *threads* em *Python* executem ao mesmo tempo. Isto é necessário pois o controle de memória do C*Python* não é *thread safe*. Para contornar o GIL seria preciso utilizar outras implementações da linguagem, como o *Jython* ou o *IronPython*, ou utilizar o pacote *multiprocessing* ao invés do módulo *threading*. Explorar outra implementação se mostrou custoso pela falta de documentação adequada, enquanto utilizar *multiprocessing* fugiria a especificação do trabalho. Assim, continuamos com a utilização nativa de *threading*.

ii) Para implementação do *in place* tivemos problemas com a alteração do vetor diretamente no servidor. Assim, para resolver o problema, utilizamos a sugestão do professor, fazendo da seguinte forma:

- Na chamada do servidor feita pelo cliente, passamos parte do vetor por parâmetro para a função no servidor e o retorno da função alterará imediatamente essa parte no vetor principal
- No servidor recebemos o vetor enviado, que é parte do vetor principal, são realizadas as operações no vetor e ele é retornado para o cliente

iii) Ao utilizamos o serviço SimpleXMLRPCServer para as chamadas RPC, o programa desenvolvido foi normalmente executado para um vetor de tamanho N = 10^7, porém, ao considerarmos o vetor de tamanho 10^8, tivemos problemas com a conexão.

Buscando outras possibilidades, encontramos o RPyC (*Remote Python Call*), que é uma biblioteca *Python* para realização de *remote procedure calls*, *clustering* e computação distribuída. RPyC faz uso de *object-proxying*, para superar os limites físicos entre processos e computadores, de modo que os objetos remotos podem ser manipulados como se fossem locais. A documentação dessa biblioteca pode ser encontrada em https://rpyc.readthedocs.io/en/latest/.

Com essa biblioteca, conseguimos implementar o segundo exercício corretamente, que será explicado mais a frente, bem como possibilitar uma melhora de desempenho em nosso programa *multithreading*. Isso se deve, provavelmente, ao suporte à computação paralela presente na biblioteca, que considera todos os processos conectados a ela como uma grande único processo, tendo controle sobre seu escalonamento e sincronização.

iv) Suportar e passar como parâmetros vetores de inteiros em *Python* é consideravelmente complicado uma vez que estes ocupam 40 *bytes* por posição. Assim, utilizamos *arrays* da biblioteca *NumPy*, que são rápidos, eficientes e ocupam menos espaço na memória. Porém, o RPyC só suporta os tipos primitivos da linguagem em suas chamadas, sendo necessário passar nosso *array* para *string,* enviá-lo, passá-lo novamente para *array* e enfim executarmos as operações. Para realizarmos essa de maneira correta, utilizamos o algoritmo sugerido em:
https://mail.scipy.org/pipermail/numpy-discussion/2008-September/037815.html.

Além disso, visando otimizar o número de execuções a serem realizadas, utilizamos a técnica de blocagem. Partindo de estudos realizados anteriormente, utilizamos a melhor configuração para o *hardware* em questão, oito operações por *loop* ao *array*, para melhorarmos o desempenho durante a execução das chamadas RPC.

Para o estudo de caso, consideramos um vetor de tamanho $N = 10^8$, preenchido de forma aleatória (com números no intervalo [0, 100]), com valores de K = 1; 2; 4; 8. 16; 32; 64; 128 para as *threads*. Para cada valor de K, obtivemos o tempo médio de execução do programa rodando-o 10 vezes. Porém, não obtivemos sucesso nas operações para K=1, 2, 4, 8 e 16. Em todas essas execuções o cliente retornava um *timeout*, que não identificamos a razão. Assim, nossos resultados para N=10ˆ8 são para K=32, 64 e 128, apresentados na Tabela 1 e na Figura 1, que nos mostram que o tempo de execução melhora significativamente com o aumento do número de *threads*, o que é o contrário do que obtivemos inicialmente, considerando os problemas com GIL, mas que foi possível com a utilização do RPyC.

Nº de *threads*	Adição	Potenciação	Multiplicação
32	94,1368	107,8053	87,2474
64	73,7812	74,1540	68,8122
128	58,5666	62,8601	57,7304

Tabela 1: Tempo de execução para operações aritméticas em vetores (N=10^8) com RPC

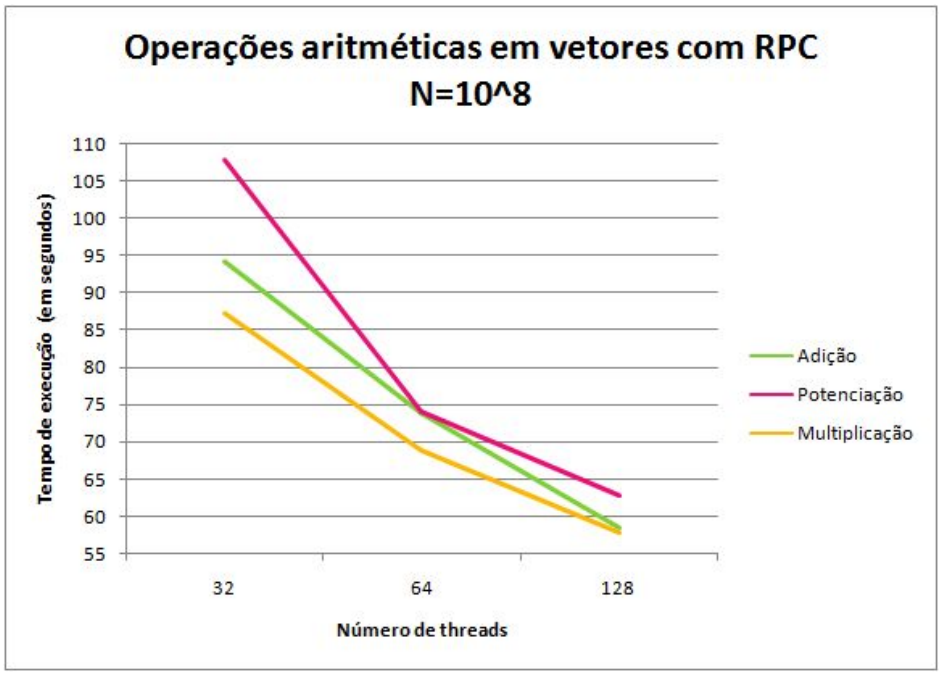

Figura 1: Tempo de execução para operações aritméticas em vetores (N=10^8) com RPC

De forma a avaliar de forma mais ampla e apropriada a influência das múltiplas *threads* no problema, analisamos os resultados obtidos na execução para N=10^7, que foi possível para todos os valores de K. A Tabela 2 e a Figura 2 apresentam esses resultados. Podemos perceber que, igualmente ao caso de N=10^8, o tempo de execução foi diminuído com o aumento do número de *threads*, fortalecendo assim, a conclusão obtida anteriormente.

Nº de *threads*	Adição	Potenciação	Multiplicação
1	18,755	19,016	18,477
2	10,091	12,183	12,144
4	7,843	8,816	8,020
8	6,199	7,161	6,5672
16	5,299	6,525	5,420
32	5,088	5,9236	5,1686
64	4,982	5,9426	4,9976
128	4,959	5,821	4,9028

Tabela 2: Tempo de execução para operações aritméticas em vetores (N=10^7) com RPC

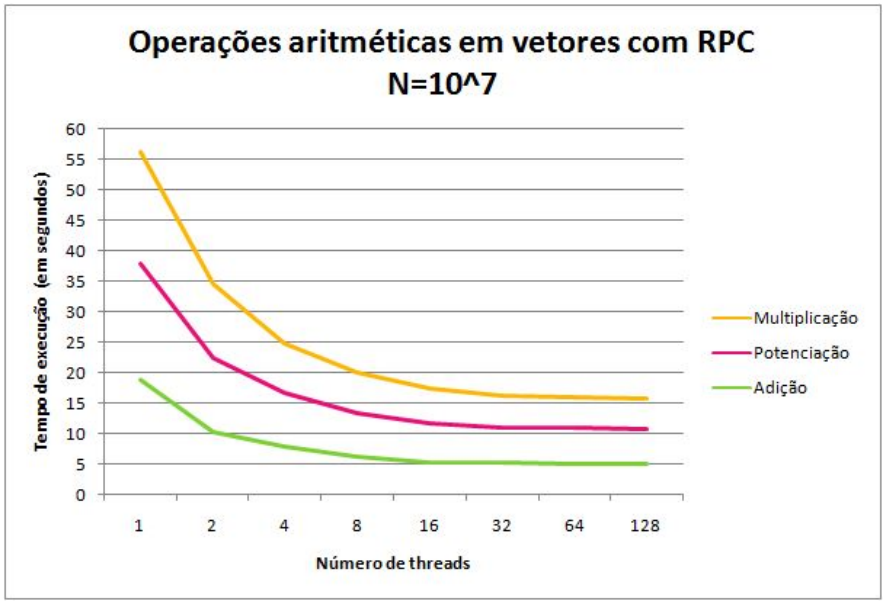

Figura 2: Tempo de execução para operações aritméticas em vetores (N=10^8) com RPC

2. Exclusão mútua entre processos

Para a criação de diversos processos no **segundo exercício**, utilizamos o pacote nativo do *Python*, ***multiprocessing***. Esse pacote oferece concorrência local e remota, contornando o GIL, mencionado anteriormente, usando subprocessos ao invés de *threads*. Já para a implementação do "coordenador", responsável pela liberação do acesso à região crítica, utilizamos chamadas RPC implementadas, a princípio, pelo módulo **SimpleXMLRPCServer**. Porém, novas dificuldades surgiram:

i) Assim como no exercício anterior, tivemos problemas com a funcionalidade do SimpleXMLRPCServer. Nesse exercício, era preciso que o servidor aceitasse diversas conexões, para os diversos processos criados, e mais importante, que os mantivessem presos enquanto estes não recebessem acesso à região crítica. Porém, na implementação utilizando o SimpleXMLRPCServer, detectamos que o servidor não aceitava mais de uma conexão ao mesmo tempo, agindo como um *lock* e, assim, executando um segundo processo apenas quando o primeiro fosse completamente fechado.

Este problema foi o principal motivo de aderirmos à biblioteca RPyC, também utilizada no exercício anterior. Com essa biblioteca, o coordenador é capaz de aceitar mais que um processo ao mesmo tempo, permitindo a implementação do algoritmo.

Para o estudo de caso, consideramos o número de processos K=1, 2, 4, 8, 16, 32, 64 e 128, e duas maneiras do novo processo entrar no sistema: *bulk arrival* e *sequential arrival*. Para cada combinação de K e maneira de entrar no sistema, obtivemos o tempo médio de execução do programa rodando-o 10 vezes. Os resultados dessas execuções são mostrados na Tabela 3 e complementarmente na Figura 3.

Em ambos os casos vemos que o tempo de execução aumenta com o número de processos, o que é esperado pois a região crítica será acessada mais vezes. Porém, nota-se que para o *sequential arrival* esse tempo aumenta drasticamente. Isso se deve ao intervalo fixo de 1 segundo existente entre a entrada dos processos, que gera um "tempo ocioso" entre a execução dos mesmos, até que sejam todos inicializados. Ou seja, diferente do *bulk arrival*, quando o primeiro processo termina de executar, os demais não estão todos aguardando na fila pelo acesso à memória; o que faria com que a região crítica estivesse sempre sendo acessada. Assim, o "tempo ocioso" entre um processo terminar de executar e outro pedir acesso à região gera um *delay* geral no tempo de execução do problema.

Um script em *Python* foi responsável pela verificação de consistência do arquivo gerado pelo programa. Nele, conferimos para cada linha as palavras presentes, bem como suas posições. Além disso, também avaliamos o número de linhas escritas por cada processo.

Nº de processos	***Bulk Arrival***	***Sequential Arrival***
1	49,966771	51,621624
2	52,087718	53,761424
4	53,980090	55,555811
8	54,157381	57,851109
16	55,088474	67,405887
32	56,536338	83,900289
64	59,223219	115,201403
128	62,897650	179,138115

Tabela 3: Tempo de execução para exclusão mútua entre processos no cenário de *bulk arrival* e *sequential arrival*

Figura 3: Tempo de execução para exclusão mútua entre processos no cenário de *bulk arrival* e *sequential arrival*

Anexo

1 - Operações aritméticas em vetores com RPC

```
Servidor:
from datetime import datetime
from cStringIO import StringIO
from numpy.lib import format
import copy
import rpyc
import math

rpyc.core.protocol.DEFAULT_CONFIG['allow_pickle'] = True

def from_string(s):
    f = StringIO(s)
    arr = format.read_array(f)
    return arr

def to_string(arr):
    f = StringIO()
    format.write_array(f, arr)
    s = f.getvalue()
    return s

class MyServer(rpyc.Service):
   def exposed_add(self, str_vec, x):
          vector = from_string(str_vec)
          size = len(vector)
          blocksize = int(math.ceil(size/8))
          for i in xrange(0, (blocksize*8), 8):
                 vector[i] += x
                 vector[i+1] += x
                 vector[i+2] += x
                 vector[i+3] += x
                 vector[i+4] += x
                 vector[i+5] += x
                 vector[i+6] += x
                 vector[i+7] += x
          if((blocksize*8) != size):
                 for i in range((blocksize*8), size):
                        vector[i] += x
          return to_string(vector)

   def exposed_pow(self, str_vec, x):
          vector = from_string(str_vec)
          size = len(vector)
          blocksize = int(math.ceil(size/8))
          for i in xrange(0, (blocksize*8), 8):
                 vector[i] = pow(vector[i], x)
                 vector[i+1] = pow(vector[i+1], x)
                 vector[i+2] = pow(vector[i+2], x)
                 vector[i+3] = pow(vector[i+3], x)
                 vector[i+4] = pow(vector[i+4], x)
                 vector[i+5] = pow(vector[i+5], x)
                 vector[i+6] = pow(vector[i+6], x)
                 vector[i+7] = pow(vector[i+7], x)
          if((blocksize*8) != size):
                 for i in range((blocksize*8), size):
                        vector[i] += x
          return to_string(vector)

   def exposed_mul(self, str_vec, x):
          vector = from_string(str_vec)
          size = len(vector)
          blocksize = int(math.ceil(size/8))
          for i in xrange(0, (blocksize*8), 8):
                 vector[i] = vector[i] * x
                 vector[i+1] = vector[i+1] * x
```

```
                vector[i+2] = vector[i+2] * x
                vector[i+3] = vector[i+3] * x
                vector[i+4] = vector[i+4] * x
                vector[i+5] = vector[i+5] * x
                vector[i+6] = vector[i+6] * x
                vector[i+7] = vector[i+7] * x
        if((blocksize*8) != size):
                for i in range((blocksize*8), size):
                        vector[i] += x
        return to_string(vector)

if __name__ == "__main__":
    from rpyc.utils.server import ThreadedServer
    t = ThreadedServer(MyServer, port = 18862)
    t.start()
```

Cliente:

```
from datetime import datetime, timedelta
from cStringIO import StringIO
from numpy.lib import format
from threading import Thread
import numpy as np
import rpyc
import sys

rpyc.core.protocol.DEFAULT_CONFIG['allow_pickle'] = True

def from_string(s):
    f = StringIO(s)
    arr = format.read_array(f)
    return arr

def to_string(arr):
    f = StringIO()
    format.write_array(f, arr)
    s = f.getvalue()
    return s

def divide_vector(n, k):
   sizes = []
   splitsize = (1.0/k*n)
   for i in range(k):
          sizes.append([int(round(i*splitsize)), int(round((i+1)*splitsize))])
   return sizes

def rpc_call(inicio, fim, x):
   string_vec = to_string(vector[inicio:fim])
   if(OP == 1):
          vector[inicio:fim] = from_string( server.root.exposed_add(string_vec, x) )
   elif(OP == 2):
          vector[inicio:fim] = from_string( server.root.exposed_pow(string_vec, x) )
   elif(OP == 3):
          vector[inicio:fim] = from_string( server.root.exposed_mul(string_vec, x) )
   else:
          print "Operation not valid! Please select: 1 = Add; 2 = Pow; 3 = Mul."

def save_file(time):
   with  open("/Users/admin/Documents/UFRJ/SD/SD_2016.1/TrabalhoPratico3/Q1/log.txt",  "a")  as
file:
          file.write("Numero de threads: " + str(K) + " Operacao: " + str(OP) + " Tempo: " +
str(time) + "\n")


N = 10000000 #10^7
vector = np.random.randint(100, size=N) #Initialize vector with random numbers
server = rpyc.connect("localhost", 18862)

if (len(sys.argv) != 4):
```

```
   print "Verify the functions parameters.\n"
   exit(1)
else:
   K = int(sys.argv[1])   #number of threads
   OP = int(sys.argv[2])  # 1 = Add; 2 = Pow; 3 = Mul
   X = int(sys.argv[3])   #int parameter to the functions


if __name__ == "__main__":
   sizes = divide_vector(N, K)
   threads = []
   i = 0

   #Calculating time
   mdelay = timedelta(0)
   tempo1 = datetime.now()

   #Call rpc function for each thread
   for k in range(K):
          size = sizes[k]
          t = Thread(target=rpc_call, args=(size[0], size[1], X))
          t.start()
          threads.append(t)
   for t in threads:
          t.join()

   tempo2 = datetime.now()
   mdelay = (tempo2 - tempo1)
   save_file(mdelay)
   exit(0)
```

2 - Exclusão mútua entre processos

Coordenador:

```
import rpyc

queue = []

class MyServer(rpyc.Service):

   def exposed_ask_request(self, myTime, myId):
          queue.append(myId)
          while True:
                 if((len(queue) == 0) or (queue[0] == myId)):
                        return 1 #GRANT

   def exposed_send_release(self, myTime, myId):
          del queue[0]
          return 1 #SLEEP


if __name__ == "__main__":
    from rpyc.utils.server import ThreadedServer
    t = ThreadedServer(MyServer, port = 18860)
    t.start()
```

Clientes:

```
import rpyc
import sys
import time
import random
from datetime import datetime, timedelta
import multiprocessing
```

```
if len(sys.argv) != 3:
    print "Verify the functions parameters.\n"
    exit(1)
else:
   K = int(sys.argv[1]) #number of client processes
   order = int(sys.argv[2])  # 1 = Bulk arrival; 2 = Sequential arrival;


def write_file(myTime, myId):
   with  open("/Users/admin/Documents/UFRJ/SD/SD_2016.1/TrabalhoPratico3/Q2/output.txt",  "a")
as file:
          file.write("Meu id: " + str(myId) + ", meu tempo: " + str(myTime) + "\n")

def write_final_time(Time):
   with  open("/Users/admin/Documents/UFRJ/SD/SD_2016.1/TrabalhoPratico3/Q2/log.txt",  "a")  as
file:
          file.write("Numero  de  processos:  "  +  str(K)  +  ", ordem: " + str(order) + "\nMeu
tempo: " + str(Time) + "\n")
          file.write("_____________________________________________________________________\n")


def send_to_coordinator(myTime):
   coordinator = rpyc.connect("localhost", 18860)
   myId = p.pid
   myCount = 0
   while (myCount < 100):
          #REQUEST
          action = coordinator.root.exposed_ask_request(myTime, myId)
          if(action == 1):
                 #GRANT received
                 write_file(myTime, myId)
          myCount+=1
          #RELEASE
          action = coordinator.root.exposed_send_release(myTime, myId)
          if (action == 1):
                 time.sleep(round(random.random(),1))
                 myTime = str(datetime.now())



if __name__ == "__main__":

   mdelay = timedelta(0)
   tempo1 = datetime.now()
   processes = []
   for i in range(0, K):
          p=multiprocessing.Process(target=send_to_coordinator, args=((str(datetime.now()),)))
          processes.append(p)

   #Bulk Arrival
   if (order == 1):
          for p in processes:
                 p.start()

   #Sequencial Arrival
   elif (order == 2):
          for p in processes:
                 p.start()
                 time.sleep(1)
   for p in processes:
          p.join()


   tempo2 = datetime.now()
   mdelay = mdelay + (tempo2 - tempo1)
   write_final_time(mdelay)
```