ISSN No:-2456-2165

Performance Analysis of Network Port Scanning When Using Sequential Processing, Multithreading and Multiprocessing in Python Programming Language

Ante Projić
University College of Management and Design ASPIRA
Split, Croatia

Frane Marjanica
University College of Management and Design ASPIRA
Split, Croatia

Abstract:- This paper contains an analysis of sequential processing, multithreading, and multiprocessing of a simple application based on Python programming language. Due to the availability of powerful hardware, parallelism and concurrency have become an efficient and powerful solution when considering software performance. Execution times are a significant factor to look at when designing and implementing software. In this paper, we analyze the software performance regarding sequential processing, concurrency and parallelism and its impact on execution times when using network port scanning. The analysis is based on three separate implementations of the same function and its performance on the same network subnet, same hardware, and operating system. We will show that multithreading and multiprocessing have a significant performance impact on software analyzed when using repetitive executions and that we are able to improve the performance of our application. The results obtained will provide insights into the parallel capabilities of Python programming language, and it will show the differences between multithreading and multiprocessing techniques. We will explore both models for parallel computing. The purpose of this paper is to explore the implementation of multithreading and multiprocessing in Python programming language and its potential limitations and implementation suggestions.

Keywords:- *Concurrency, Multiprocessing, Multithreading, Parallelism, Sequential Processing.*

I. INTRODUCTION

Parallel computing fulfills a need for the increased and efficient performance of applications. Parallel computing enables the simultaneous use of multiple computing resources to enable processing that can be executed on multiple CPUs or CPU cores. The processing problem can be broke down into discrete pieces that can be processed simultaneously, and each piece can be further divided into a series of serially executed instructions on different CPUs or CPU cores. Not to be confused with parallel computing, concurrent programming denotes executing several operations concurrently in overlapping periods.

Concurrent programming is not equivalent to parallel programming. Concurrency is a property that specifies that some operations can be run simultaneously, but that is not necessarily the case. Parallelism is a property that specifies that operations are being run simultaneously. Concurrent threads or processes will not necessarily be running in the same instant, wherein parallelism two or more processes or threads literally run at the same time.

II. PARALLELISM AND CONCURRENCY IN PYTHON PROGRAMMING LANGUAGE

Parallel computing fulfills a need for the increased and efficient performance of applications. Parallel computing enables the simultaneous use of multiple computing resources to enable processing that can be executed on multiple CPUs or CPU cores. The processing problem can be broke down into discrete pieces that can be processed simultaneously, and each piece can be further divided into a series of serially executed instructions on different CPUs or CPU cores. Not to be confused with parallel computing, concurrent programming denotes executing several operations concurrently in overlapping periods. The most widely used programming approach for the management of concurrency is based on multithreading. In concurrent programming, an application is made by a single process that is divided into multiple independent threads.

Python programming language has built-in libraries that enable the use of most common programming methods that deal with concurrent or parallel execution, and those are multiprocessing and multithreading.

A process in Python is an independent sequence of execution that has its own memory space. Its memory space is not shared with other processes. Thread is also an independent sequence of execution that shares its memory space with other threads that belong to the program. A thread runs on a single processor, and therefore only one can be run at a time where processes run on separate processors. The most important difference between the two is that concurrency effectively hides latency and gives an illusion of simultaneity where parallelism executes tasks at the same

time. Threads allow dividing the main control flow of a program into multiple concurrently running control streams. Threads allow the creation of concurrent pieces of the program where each piece accesses the same memory space and variables. In contrast, processes have their own memory space and resources. Also, starting threads is computationally less expensive and requires fewer resources than starting processes.

Concurrency and parallelism can be of extreme significance for the performance of software programs, especially those that are CPU and I/O heavy. For example, I/O operations cause a lot of delays in execution due to frequent waits from an external resource like a network.

This paper will demonstrate the impact of sequential and concurrent executions on a port scanning program, and how performance changes with different parameters. Since port scanning spends most of its time waiting for a network request to complete, we can implement concurrency so that in those times that is spent on waiting for another task to complete another task runs.

III. IMPLEMENTING PORT SCANNING IN PYTHON

Our experiment with serial, concurrent and parallel execution is based on a program developed in Python (Python v2.7.5) which implements the basic functionality of port scanning on a single host. For port scanning, we are using socket library in Python. Determinating if the port is open is done via socket.connect_ex(address) function where the return error indicator of 0 means that the port is open and connection has been successfully established. Socket global timeout is defined by using the function socket. Set default timeout(timeout) where it is set at 0.5 seconds. Queue library is used for safely exchanging information between threads. There are three main functions in which is port scanning implemented. Function serial_port_scanner() implements basic serial execution where all port scans are done sequentially. Function multithreading_port_scanner() implements concurrent execution via multithreading by using Python built-in threading library. User can specify the number of threads which will be spawned. Function multiproccesing_port_scanner() implements parallel execution via multiprocessing by using Python built-in multiprocessing library.

A. Source code for application

```
########################################
# port_scanning.py #
Ante Projić #
# PERFORMANCE ANALYSIS OF NETWORK ##
PORT SCANNING WHEN USING SEQUENTIAL ##
PROCESSING, MULTITHREADING AND ##
MULTIPROCESSING IN PYTHON #
# PROGRAMMING LANGUAGE #
# #
########################################
import datetime, socket, time, sys, threading, Queue, multiprocessing

def port_scanner(PortNumber):

    tcp_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    result = tcp_sock.connect_ex((targetIP, PortNumber))
    if result == 0:
        return PortNumber
    tcp_sock.close()

def host_info():
    host_name = socket.gethostname()
    ip_address = socket.gethostbyname(host_name)
    print "Program is run on this host:"
    print "Hostname: %s" % host_name
    print "IP address: %s" % ip_address
    print "Timestamp:"
    print datetime.datetime.now()

def get_target_host():

    print "------------------------------------------------------"
    print "Enter FQDN host address for scanning:"
    target = raw_input(">> ")
    targetIP = socket.gethostbyname(target)
    print "Scanning host %s, IP address: %s" % (target, targetIP)
    print "Enter port range:"
    start = raw_input("Start port >> ")
    end = raw_input("End port >> ")
    return [start, end, targetIP];

def threader():
    while True:
        worker = q.get()
        result_mt.append(port_scanner(worker))
        q.task_done()

def pool_handler(ports, number_of_processes):

    pool = multiprocessing.Pool(processes=number_of_processes)
    pool.map(port_scanner, ports)

def serial_port_scanner():
    print "SERIAL Port scan starting!!!"
    start_time = time.time()
    result_se = []

    for PortNumber in ports:
        result_se.append(port_scanner(PortNumber))
    for port in result_se:
        if port == None:
            pass
        else:
            print "##Port %d is OPEN##" % port

    elapsed_time = time.time() - start_time
    print "SERIAL Port scan finished!!!"
```

```
    print "Time elapsed for SERIAL execution: %s " % elapsed_time

def multithreading_port_scanner():
    print "MULTITHREADING Port scan starting!!!"
    global q, result_mt
    q = Queue.Queue()
    print "Enter number of threads:"
    number_of_threads = int(raw_input(">> "))
    result_mt = []
    start_time = time.time()
    for x in range(number_of_threads):
        t = threading.Thread(target=threader)
        t.daemon = True
        t.start()
    for worker in ports:
        q.put(worker)
    q.join()
    for port in result_mt:
        if port == None:
            pass
        else:
            print "##Port %d is OPEN##" % port
    elapsed_time = time.time() - start_time
    print "MULTITHREADING Port scan finished!!!"
    print "Time elapsed for MULTITHREADING execution: %s " % elapsed_time

def multiprocessing_port_scanner():
    print "MULTIPROCESSING Port scan starting!!!"
    host_cpu_count = multiprocessing.cpu_count()
    print "Number of CPUs in this host is %d" % host_cpu_count
    print "Enter number of proccesses:"
    number_of_processes = int(raw_input(">> "))
    start_time = time.time()
    pool = multiprocessing.Pool(processes=number_of_processes)

    result_mp = pool.map(port_scanner, scanlist)
    for port in result_mp:
        if port == None:
            pass
        else:
            print "##Port %d is OPEN##" % port
    elapsed_time = time.time() - start_time
    print "MULTIPROCESSING Port scan finished!!!"
    print "Time elapsed for MULTIPROCESSING execution: %s " % elapsed_time

if __name__ == '__main__':


    socket.setdefaulttimeout(0.5)
    target_variables = []
    scanlist = []

    host_info()
    target_variables = get_target_host()
    ports = range(int(target_variables[0]), int(target_variables[1]) + 1)
    targetIP = target_variables[2]

    for x in ports:
        scanlist.append(x)

    try:
        serial_port_scanner()
        print "-------------------------------------------------------\n\n"
        multithreading_port_scanner()
        print "-------------------------------------------------------\n\n"
        multiprocessing_port_scanner()
    except KeyboardInterrupt:
        print "You pressed Ctrl+C. Exiting program."
        sys.exit()
```

B. Performance with serial, concurrent and parallel implementations

The application was run on a Linux host (Red Hat Enterprise Linux Server release 7.5 (Maipo)) with 4 CPUs. Port scanning was done on a remote host with a total of 28 ports in listening state. The host was scanned for 1 port, 10 ports, 100 ports, 1024 ports, and all 65535 ports respectively. Every iteration of scanning was using serial, multithreading and multiprocessing techniques. Full results from the experiment are available in the table below.

	Execution time on 1 port	***Execution time on 100 ports***	***Execution time on 1024 ports***	***Execution time on 65535 ports***
Serial	**0.003sec**	0.33sec	3.48sec	218.74sec
Multithreading with 10 threads	0.008sec	**0.04sec**	0.38sec	22.51sec
Multiprocessing with 10 processes	0.039sec	0.07sec	0.37sec	21.57sec
Multithreading with 100 threads	0.029sec	0.06sec	**0.31sec**	**17.81sec**
Multiprocessing with100 processes	0.33sec	0.40sec	1.60sec	71.97sec
Multithreading with 200 threads	0.05sec	0.09sec	0.32sec	18.07sec
Multiprocessing with 200 processes	0.91sec	1.10sec	2.38sec	61.29sec

a. Execution times

IV. CONCULSION

Benchmark results show that in this particular instance multithreading is far superior to serial execution and that it also performs better than multiprocessing. Firstly, multiprocessing has an overhead, it takes longer to spawn a process when compared to multithreading. Multithreading doesn't have as much overhead since the threads can read the same object without creating a copy. Global Interpreter Lock (GIL) in Python is a form of a bottleneck when using multithreading in Python as it allows maxing out only one processor and it prevents execution of multiple threads at once in Python. GIL is a thread-safe mechanism that is used to prevent threads from writing to the same location in memory, and it prevents parallel threads from executing on multiple cores. Therefore, GIL prevents conflicts between threads by executing only one statement at a time.

In this application, multithreading performed significantly better than serial execution which was expected since we are dealing with an I/O (input/output) processing and in time that is spent waiting on a request another thread is spawned and executed. Multiprocessing also cannot scale well in this type of repetitive tasks that are I/O heavy since using more processes than there are available CPUs leads to competition for CPU resources. Contrary, if we were analyzing a processing task that is CPU heavy multithreading might not perform better than serial execution since only one thread could be executed at any given time, and taking into account the time needed to perform a switch between threads, execution time would most likely be worse on multithreading approach than using serial execution. Multiprocessing would be expected to be far superior to serial and multithreading in that case because it would use all available processors in the system.

Multithreading is best for I/O tasks and hiding latency, while multiprocessing performs better for computations. We can conclude that threads are not recommended for use in CPU bound tasks, but they are extremely effective in I/O tasks like network IO as in this example and in filesystem I/O.

REFERENCES

[1]. A. Zaccone, "Python parallel programming cookbook", Packt Publishing Ltd, 2019.
[2]. S. Kumar, "Python : threading vs multiprocessing" unpublished.
[3]. S. Ghosh "Multiprocessing vs. threading in python: what every data scientist needs to know" unpublished.
[4]. J. Palach, "Parallel programming with python," Packt Publishing Ltd, 2019.