Pipe Operator Report

Pol Alvarez

December 30, 2015

1 Method description

The script uses os.fork() and os.pipes() to create a given number of persistent workers (defaults to core count) to which lists of operations are sent by the master to be performed in the different process.

The input is divided into chunks of operations and then sent to each thread through a pipe. Once the process finished all the operations signals the master (with a shared lock) that the results are being written into the pipe. Once available, the master thread reads and prints the results and starts again to send/receive data with next available thread.

The input used is a file of 300003 lines with operations and some 'malicious' lines.

The operations are validated and computed with Python's *ast* module that allows to navigate operations as trees and thus enhancing recursive parsing and computing.1

2 Time comparison

Figure 1 shows the comparison between the sequential time and multiple parallel executions given by the chunks' size (does not apply to the sequential version). It is also shown the time used by the master thread to write and read the operations and results, and the time without the mentioned I/O part.

Best times were achieved with four workers (machine core count was also four) even if one process is used by the master. Probably the high amount of I/O time frees the master process enough time to obtain a performance increase w.r.t executions with a pool of three threads.

Most time-consuming part of the executions are the read/writes to pipes. IO times on the plot correspond to the time spent by the master performing read/write operations. Persistent workers are by default waiting for data so their

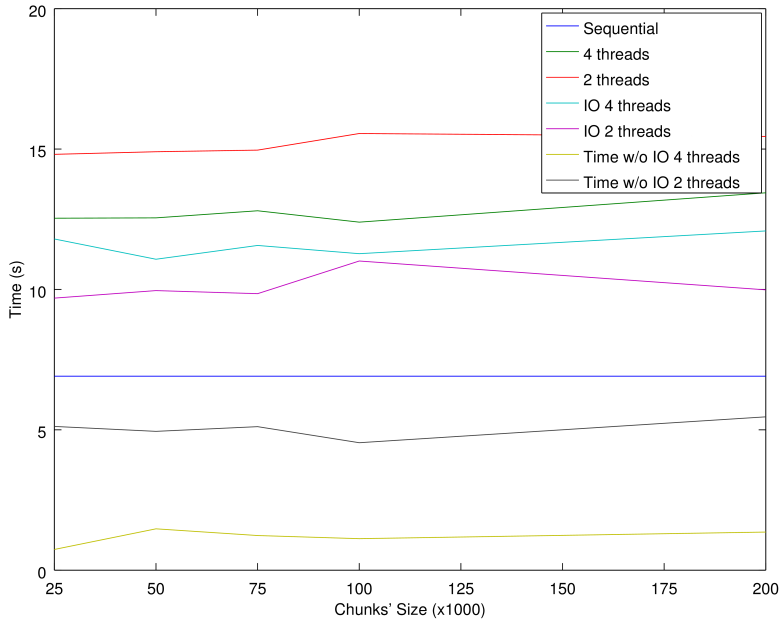

Figure 1: Execution times of each version given by chunk size

read time is shadowed by the master write time because they happen almost simultaneously. The reception of results is asynchronous so it could increase the execution time. Despite that, it is clear that the actual time performing operations is reduced by more than a quarter (with 4 processes).

3 More

To further increase performance, next step would be optimize IO times. One option would be to create transfer threads on the master to send data to more than one thread at once, another to use sockets. Another option that would follow a similar approach would be to use mpi4py module and follow an MPI approach.