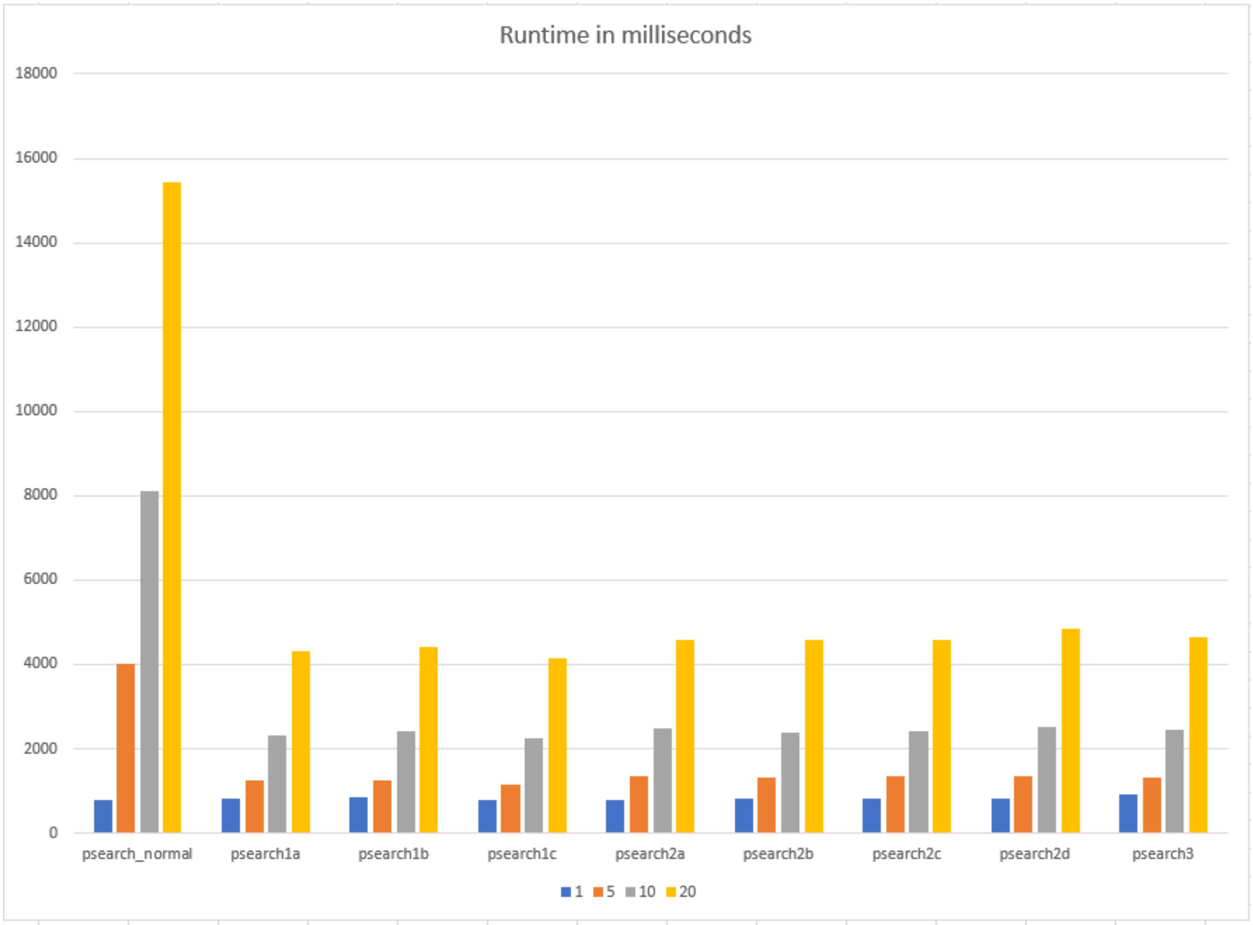

As it can be seen in the graph, making use of multiple processes can speed up programs dramatically. This benchmark was run on a 3-core (6-threads) CPU and as it can be seen in the data, multiprocessing programs run almost 3 times faster than single process `psearch_normal` program. We can see that other multiprocessing programs work in almost same runtime, this can be attributed to process/thread creation costs. Creating processes is an expensive operation, a system call must be issued, the kernel must prepare resources for the new process, schedule each process fairly, so on and so forth. The run time differences between multi processed programs can be attributed to different overheads of communication and synchronization methods, however since they are very small compared to the process creation overhead, they aren't as noticeable.

When we increase the amount of input files, the time increases almost linearly as using multiple processes only means we can run multiple tasks on multiple cores at the same time. If there aren't enough cores for every task, the run time will still be like as if program was consequentially executing, except runtime will be divided by count of cores.

To fix issues given above, we can use process/thread pools, this means we can create a fixed amount of processes/threads (usually one per each CPU thread) and feed them jobs to do from the master process, avoiding process/thread creation overhead for each input and avoiding increased context switches due to there being more processes than CPU threads. To overcome communication performance issues, programs must avoid using locks to synchronize processes, they must use pipes or network sockets to avoid sharing memory and introducing data races. And programs can use asynchronous file operations with kernel support to avoid blocking when input files are too big in size.