James Kerwin

Professor Kannan

Operating System Theory

4 March 2025

Project 1, Part 2 - Shared memory Scalability

Code Description and Use

IMPORTANT: To run the code, first simply compile using the makefile; you can optionally modify parameters, like the following: “make MATRIX_SIZE=10000 NUM_CORES=12 NUM_CHILD_PROCS=15”. You must “make clean” before running “make” again later. To run the rest of the programs, simply do one of the following: “./gen-mat”, “./IPC-pipe”, “./IPC-shmem”, “./verify”.

I have provided 4 .c files for use, these are: gen-mat.c, IPC-pipe.c, IPC-shmem.c, and verify.c. Gen-mat.c generates two matrices, each of MATRIX_SIZE rows and MATRIX_SIZE columns, and writes them to “mat1.csv” and “mat2.csv”. The matrices have elements of the following form: floats from -99.99 to +99.99, that is, of 4 significant figures (for example, 33.26 or -97.81). IPC-pipe.c and IPC-shmem.c immediately start a timer upon execution; then, they perform the matrix multiplication (using child processes and IPC), and do not finish the timer until the result has been written again to disk, in “mat-out-pipe.csv” and “mat-out-shmem.csv”, respectively. They each print their stats as was required by the project description. Finally, verify.c performs the matrix multiplication within a single process and verifies that the result matches what the other programs returned. I found that semaphores weren’t necessary within IPC-shmem.c since on these machines the entire output array could be put into shared memory and no process modified the same spot in memory as another process.

Results

All testing done on m510 machine on cloudlab (specifically ms0804.utah.cloudlab.us).

My scalability scoreboard submission:

10000 x 10000 matrices, 8 cores, 8 child processes, 624.439877 seconds.

```
jfk149@node-0:~/os-ipc-analysis$ ./IPC-shmem
Input size: 10000 columns, 10000 rows
Total cores: 8
Total runtime: 624.439877
```

Here are some of my results:

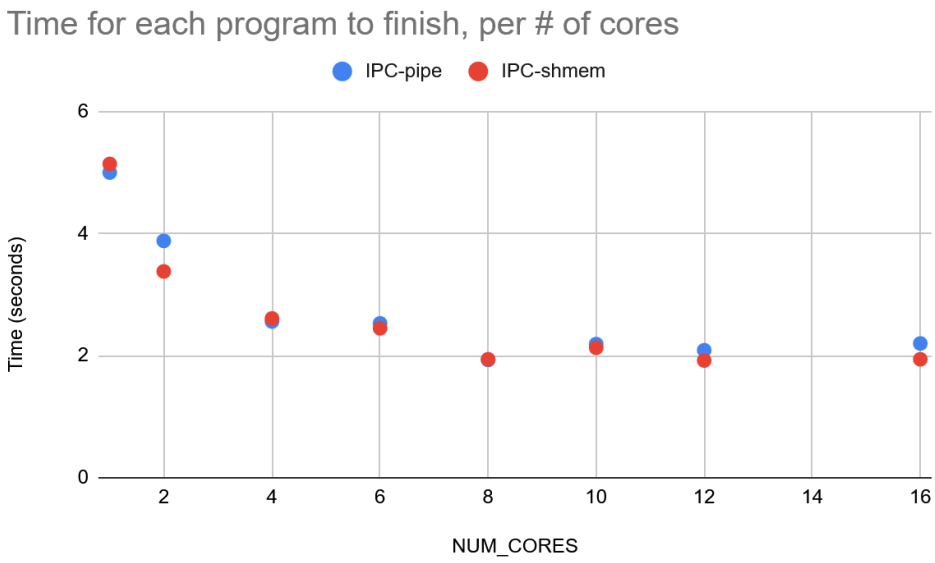


Figure 1: Number of child processes kept constant at 8, matrix size kept constant at 1000x1000.

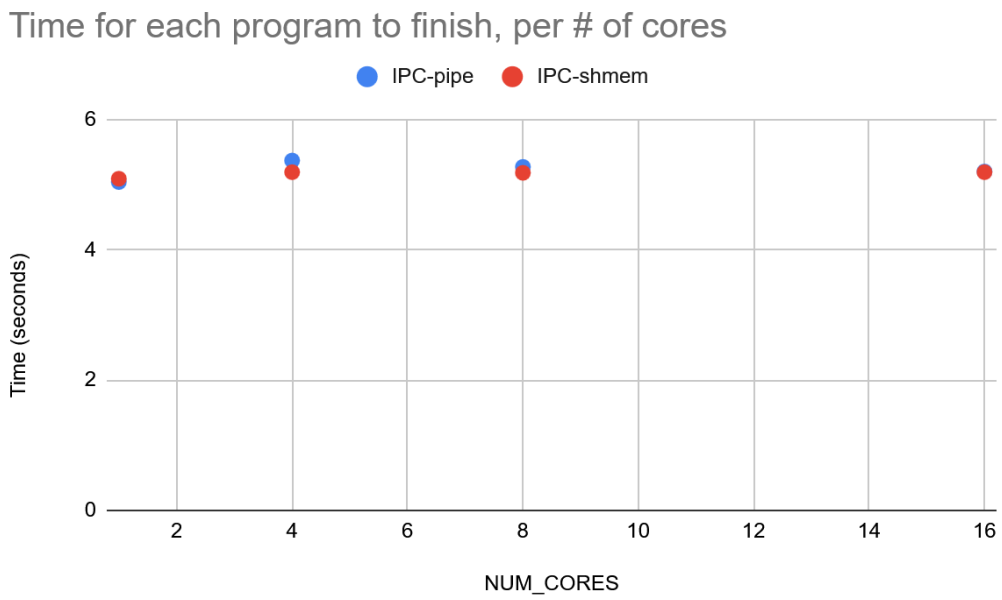


Figure 2: Number of child processes kept constant at 1, matrix size kept constant at 1000x1000

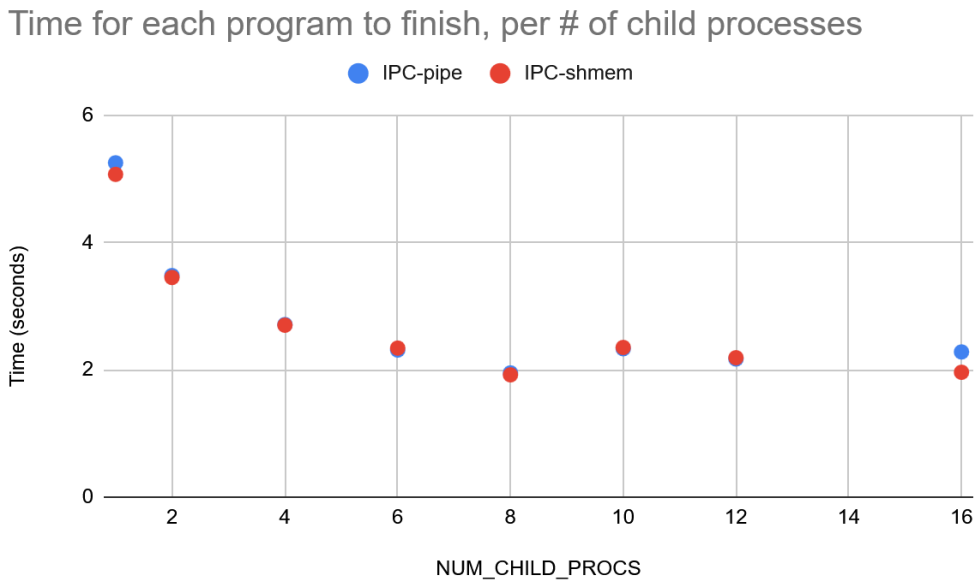


Figure 3: Number of cores kept constant at 8, matrix size kept constant at 1000x1000

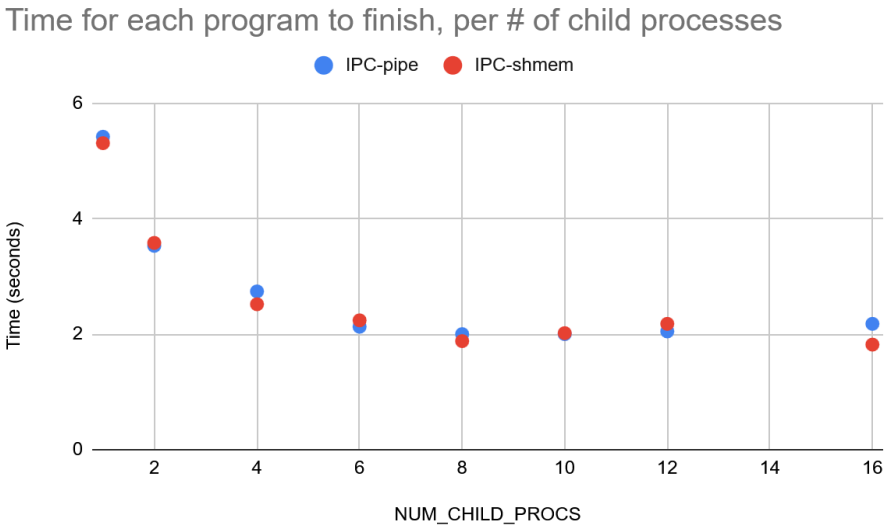


Figure 4: Number of cores kept constant at 16, matrix size kept constant at 1000x1000

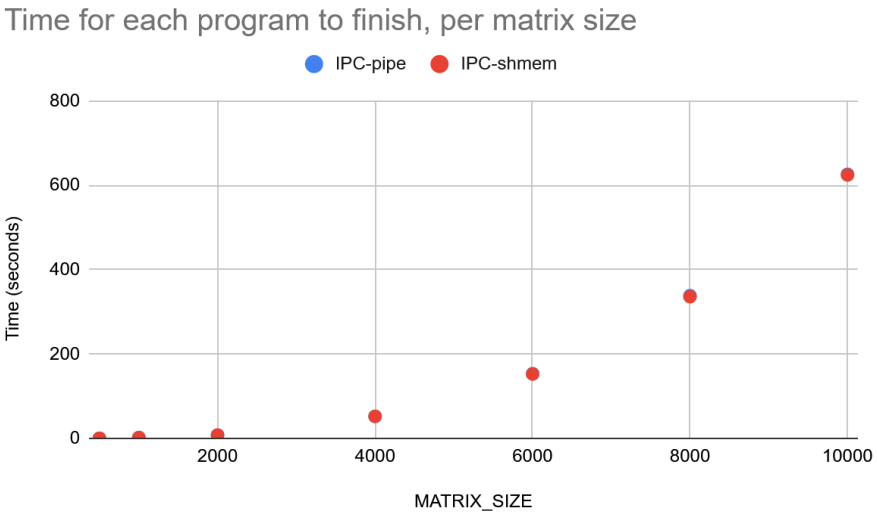


Figure 5: Number of cores kept constant at 8, number of child processes kept constant at 8.

Discussion

So these results are interesting; in some ways they are expected, in some other ways they're unexpected (but understandable in hindsight). From figure 1 we can see that the programs become more performant with more cores, but seem to hit a performance asymptote at 8 cores and don't improve much past that. This was unexpected at first because I figured the performance would increase as you increase cores up to the limit. From figure 2 we can see that with just one child process, the performance doesn't increase with more cores. This makes sense as you don't really get an improvement from more cores if you aren't performing some sort of code parallelism. So perhaps the reason why you hit a performance asymptote at 8 cores is because you're only running 8 child processes? Before investigating that further, figure 3 shows that with 8 cores you also hit a performance asymptote at around 8 child processes. Ok, interesting, maybe it's because you're only using 8 cores so if you introduce more child processes you don't get any more parallelism. If we run 16 cores, we should see performance increases up to 16 child processes, right? Wrong; figure 4 shows us that even with 16 cores we see a performance increase up to just 8 child processes before we hit that pesky asymptote. So what's going on here? Lscpu shows us the answer:

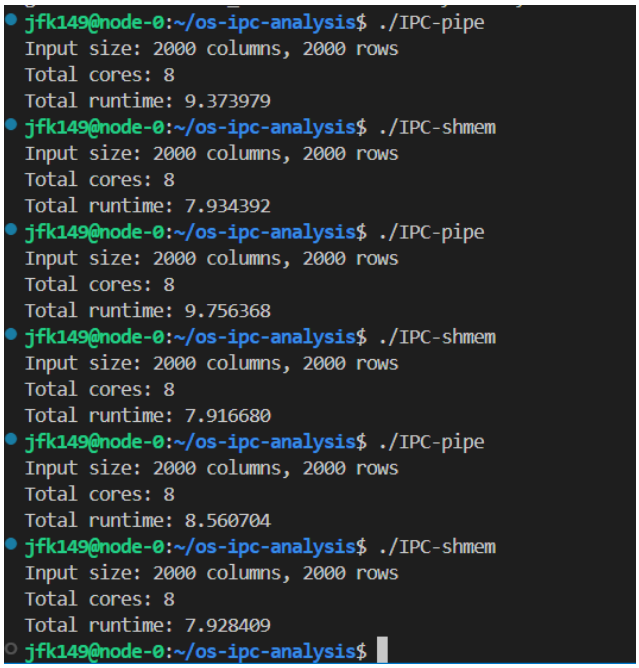
Thread(s) per core: 2

Core(s) per socket: 8

So the m510 machine has only 8 actual physical cores on the machine, but with 16 logical cores provided due to hyperthreading. Hyperthreading can be useful when we want more logical cores to do more jobs at the same time, but here the job is always going to be the same, so we're really limited by the number of physical cores we have. So that's why we're hitting that performance asymptote at 8 processes and using 8 logical cores; using any more than this is really just going

to make us waste time by context switching away on one of those physical cores to another process that's just going to do the same thing.

Another interesting thing is that the IPC-shmem program is barely more performant than the IPC-pipe program, despite not having any semaphores and thus having no synchronization. On its face this should be absurd, since synchronization should always slow you program down. I figured that maybe this was only the case at small matrix sizes, like 1000x1000, but even up to 10000x10000 you see barely any difference. One thing to note is that these results are just some test cases, and do not represent the average results. I expect that if you took the overall average, there would be a statistical difference in means, albeit small (if this assignment required this I would do it, but this is more like personal inquiry). So why is there perhaps just a small performance increase? I suspect this is due to the way I implemented the IPC-pipe program. When I ran perf, I saw similar results between my IPC-shmem program and my IPC-pipe program on how they consume CPU time, both with quite low % of kernel time. I think since I implemented IPC-pipe to have quite large buffers, and since writing is an O(n^2) task while performing calculations is an O(n^3) task, there's just very little contention between processes (there's also very few processes); that is, the lock is held by any given process for just a short amount of time, and they only rarely need the lock. Semaphores are best used in low contention environments, so that's why the semaphores are performing so nicely here. For demonstration's sake, I ran it with 100 child processes at 2000x2000 with 8 cores, and here are the results:



. So it seems like the performance of the IPC-pipe here is consistently worse than IPC-shmem. Go figure!

Conclusion

In conclusion I enjoyed the project, it was fun trying to optimize my code. I'd love to know what I could have done differently in order to further optimize it, as I tried my best. Thank you.