IPC

For matrix multiplication, the method used was Strassen's Algorithm. Overall, the architecture of both Pipes and SHM are similar (i.e. no extra optimizations were made for one or the other) so that the differences of the two runtimes are a result of using SHM vs Pipes and not because of design differences – this turns out to be a mistake later.

The hypothesis from the start is that SHM is faster than using Pipes, however as we'll see in the benchmarks (because of architectural similarities) the pipes perform better in both overall runtime and scalability.

Overall Design

- Call fork_strassens.
- Split the two matrices into 4 splits each.
- Create 7 child processes (for the 7 resulting matrices in the algorithm).
- Parent:
    1. Assign child a split id for which split it will handle.
    2. Wait for a child to return its split and then combine them.
    3. Return resulting matrix.
- Child:
    1. Call fork_strassens with its designated splits.
    2. If not at the fork bound, do what parent process does above but for its own children.
    3. If at the fork bound, call regular Strassen's (this also applies to the parent process).
    4. Write the resulting matrix either to pipe or SHM.

The differences are outlined in each section below.

1.1 Pipes

- Parent:
    1. Create a pipe and semaphore before forking.
    2. After creating 7 child processes, read the pipe, waiting for a child process to write its split id, matrix size, and matrix (Do this until all 7 are returned).
    3. Destroy semaphore and close pipe before returning.
- Child:
    1. If forking again, do what the parent does but for its own children.
    2. After getting its resulting matrix, grab the semaphore created by the parent.
    3. Write split id, matrix size, and the matrix to the pipe.
    4. Release semaphore and exit.

1.2 SHM

- Parent:
    1. Create an SHM array[7] that hold shm_ids of child processes.
    2. After creating 7 child processes, loop through the array checking for updated spots.
    3. If a spot is updated, attach to the shared memory location and copy the matrix to its own heap (do this until all 7 splits are copied).

    4. Destroy all shared spaces before returning.
- Child:
    1. If forking again do what the parent does but for its own children.
    2. After getting its resulting matrix, create an SHM and copy the result to the SHM.
    3. Detach from the matrix SHM and attach to the array of shm_ids.
    4. Using the split_id as a designated spot in the SHM array, update the value to the shm_id and detach.

SHM and Semaphore Creation

ftok() was used at “.” or if needed, “IPC-<pipe/shmem>.c”, along with the pid of the calling process to create unique keys. If a unique key isn’t generated, loop until one is found using the pid and a random number.

Limitations

- Matrices that are not powers of 2 are padded with zeros to the next highest power of 2 (This part is not timed since it happens before even calling Strassen’s). The actual limitation is forking for padded sections of the matrices. An optimization here could be to have the child process that is assigned to a split of 0’s to immediately return so that it does not go through with forking more child processes on 0 padded matrices.
- For some reason, IPC-pipe might hang on malloc with errno set to ENOMEM, despite using less than a gigabyte of memory. The guess at what the issue may be is the memory is too fragmented and malloc ends up taking a long time to allocate the space though it is still uncertain.
- To avoid the complexity of semaphores with Strassen’s and SHM, IPC-shmem is severely limited by one of two reasons:
    1. ftok() only using 8 bits to generate a key resulting in multiple keys being the same and causing the program to loop until a unique key is found.
    2. Each child creating its own shared matrix instead of the parent creating one and each child updating the one shared matrix with semaphores. This would reduce the cost of ftok() looping significantly since only 2 shm_ids are necessary per parent instead of 8 with the current design.
- Even without the ftok() limitation, creating 8 SHM spaces causes significant overhead from system calls (updating, copying, reading) when using shared spaces. As we’ll see in the benchmark, the scalability is compromised for SHM.
- Forking can only be controlled in powers of 7. This could be optimized by assigning splits as evenly as possible to each child process if there are less than 7 but with the time constraint, the idea was left out to keep complexity at a minimum.
- Both are limited to 3 Levels of forking (400 child processes – 343+49+7 child processes + 1 root parent) due to the iLab ulimit being set at 2000 processes (a fourth level would go beyond that).

Benchmarks

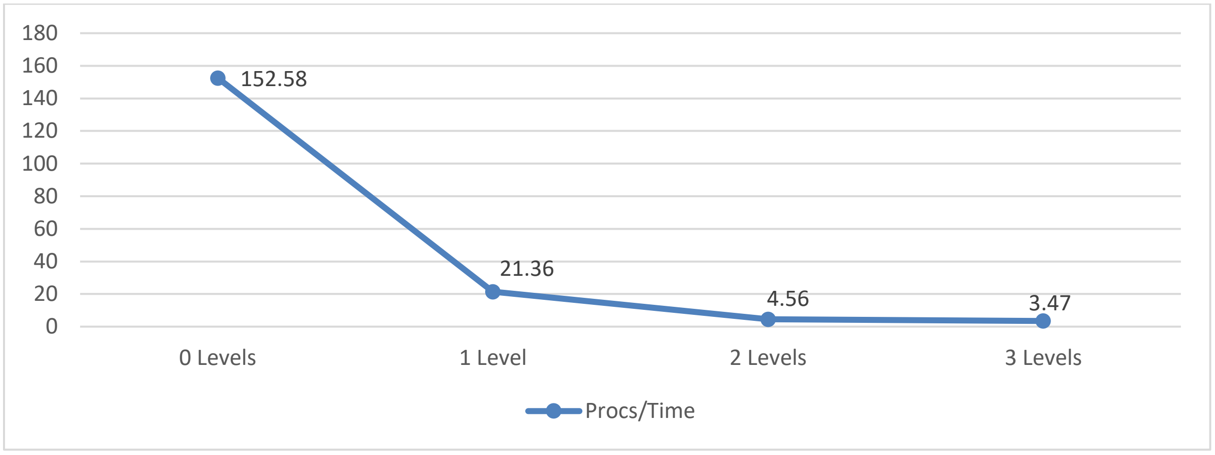
The benchmark was run on aurora.cs at 1000x1000 (padded with zeros to 1024x1024) simple matrices (not random) to observe maximum scalability. Each run contains the average of 5 samples (except for the base run of 1 process because it takes too long). As mentioned earlier, the iLabs limit the number of forking levels so only 4 Levels were tested.

Forking Levels: 0 = 1 proc, 1 = 8 procs (parent + 7 children), 2 = 55 procs, 3 = 400 procs

BASE STRASSEN SIMPLE (1024, AURORA):

0 LEVELS: 152.58 seconds

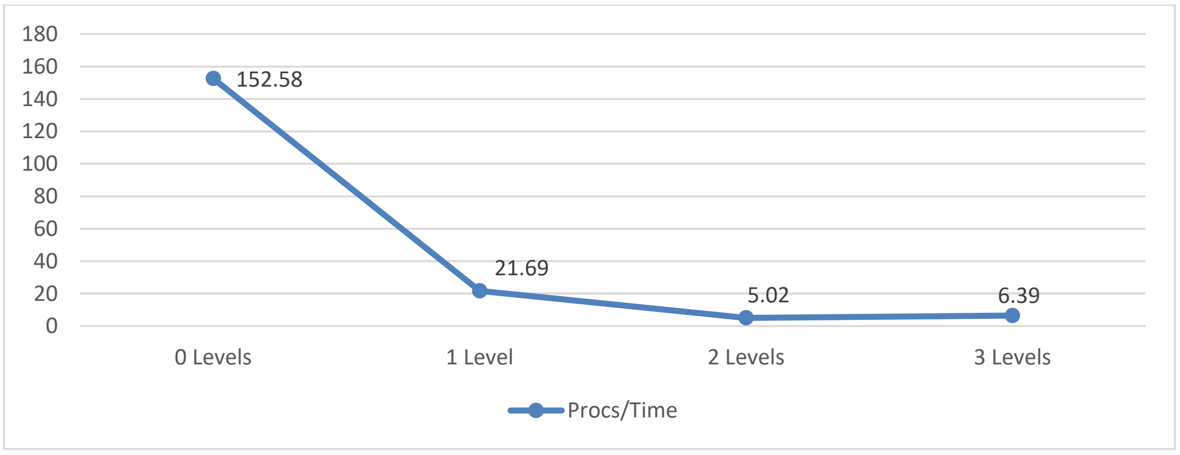
Pipes



PIPE SIMPLE (1024 AURORA, AVG 5 RUNS):

1 LEVEL: 21.36 seconds (20.44, 21.72, 20.7, 21.06, 23.39)
2 LEVELS: 4.56 seconds (4.68, 4.77, 4.52, 4.33, 4.52)
3 LEVELS: 3.47 seconds (3.45, 3.45, 3.44, 3.43, 3.6)

SHM



SHM SIMPLE (1024, AURORA, AVG 5 RUNS):

1 LEVEL: 21.69 seconds (23.05, 20.81 21.16, 22.73, 20.72)
2 LEVELS: 5.02 seconds (5.00, 4.93, 5.08, 4.95, 5.14)
3 LEVELS: 6.39 seconds (6.41, 6.49, 6.63, 5.89, 6.51)

Conclusion

Both methods scale well up to 55 processes reducing runtime by ~77%, from 8 processes, and ~97% from 1 process. Pipes reduces the runtime by about another ~30%. However, SHM fares worse from going to a third level of forking with its runtime increasing by about 1.3 seconds. This is due to the limitation listed earlier about ftok() and looping to find a unique key.

Overall, IPC-shmem performs worse due to the design. Each child process creates and copies its own shared memory space which the parent then must read and copy from. This produces significant overhead from system calls and impacts scalability.

It could be said that with more physical cores, pipes can also scale further at 400 processes. Testing on plastic.cs showed that the difference is marginal at 2 Levels (55 processes), despite the core count being significantly different (i7 8700 [12 vCores] vs. Xeon E5 4870 [80 vCores]). At 400 processes aurora.cs more than halves the runtime of plastic.cs (15.68 seconds).

Out of curiosity, a comparison was done to observe the limitations of Strassen's Algorithm itself compared to a regular splitting of the matrix using naïve matrix multiplication. The comparison was to knb93's implementation. At about 20 processes, a random matrix at size 1000 x1000 completed in less than 1 second. The overhead of Strassen's is significant when combining the splits (O(n^2) in the combining runtime of the recursive relation T(n) = 7T(n/7) + O(n^2) but the constant hidden is above 20) and when adding in communication overhead, it becomes even greater. Strassen's at 400 processes (pipes) on a random matrix of size 1000 x 1000 took 4.05 seconds. It is possible that with much higher matrix sizes Strassen's can perform better (with some optimizations such as using naïve matrix multiplication after reaching splits of 1000 x 1000 instead of using Strassen's) but this was not tested.

Running the Programs

The makefile has an all directive so calling make will compile both programs (including the cache-tlb program).

To run IPC-pipes: ./ipc_pipe <size> <type> <benchmark> <fork> <fork bound>
To run IPC-shmem: ./ipc_shm <size> <type> <benchmark> <fork> <fork bound>

<size> is the matrix size

<type> is the matrix type (1 = simple (A[i][j] = i*j), 0 = random (A[i][j] = rand()%50))

<benchmark> toggle benchmarking (1 = benchmark, 0 = no benchmark)

<fork> toggle forking (1 = fork, 0 = no forking (basic Strassen))

<fork bound> what size to stop forking - this is how to control forking levels (i.e. size = 1024, fork bound = 512 -> 1 Level; size = 256, fork bound = 32 -> 3 levels). This value should always at least be 1 (the program is hard coded to stop forking at size = 2), but forking will always stop at 3 levels, meaning running with ./ipc_shm 1000 1 1 1 1 will fork at 3 levels.