CPEN 512 Assignment 2

Parallel Gaussian Elimination using Shared Memory

Muchen He, 44638154, Oct 12, 2020

1. Introduction

In continuation of the previous assignment [1], we proceed to explore ways to parallelize the forward-elimination step of Gaussian Elimination [2], this time using POSIX threads (*pthread*). The objective of the assignment is to explore shared-memory parallelization using *pthread*, and observe its performance benefits over the serial program. We also make note of performance between message-passing versus shared memory. Single-precision floating-point values are used, and all metrics are measured form the cpen512.ece.ubc.ca computer.

2. Background

Recall from assignment 1 [1], the forward elimination (FE) involves reducing a given matrix A into its row-echelon form (REF). In this assignment we keep the reference algorithm identical. That is, we also assume that the matrix reducible such that we don't need to perform row-swapping.

The POSIX threads is an IEEE 1003.1c standard built into the *gcc* compiler that enables us to do thread programming. We can create multiple threads of the same program, sharing the same program memory, to perform computations concurrently. In our code, we simply `#include <pthread.h>` then enable `-pthread` compiler flag to use *pthread*. Each thread, upon initialization, receives a function pointer as an entry point for its routine. The thread executes the function, and we can pass each thread's parameters through function parameter.

3. Parallel Implementation

The parallelized algorithm is based on a naïve implementation from CoffeeBeforeArch [3] and assignment 1 [1] with other help from online resources [4]. Similar to the MPI implementation, we divide each row operation equally amongst C different threads. As *pthread* do not utilize message-passing, and that each thread has access to the same memory space, we simply pass the matrix pointer `*MAT`, and the starting row and ending row that this thread is responsible for reducing through the function arguments.

Similar to MPI parallelization, all threads still loop through M, the total number of rows in the Matrix. For example, when thread #0 finishes fully reducing row R_1, not only that it will start partially reducing the following rows that thread #0 is responsible for, other threads (1,2,...) will begin computing other rows R_m to R_M, where R_m is the first row not assigned to thread #0. This is synchronized using `pthread_barrier_wait`, on a shared `pthread_barrier_t` struct. Once all threads finish their tasks, we don't need to *gather* the result unlike MPI. The manipulated matrix `MAT` is immediately available for verification.

Limitations of this implementation is similar to MPI implementation from assignment 1, that the division of labour is unequal. The threads responsible for later rows end up doing more work than threads that process the upper rows. There are also significant synchronization overheads due to the barriers. Lastly, there are overheads in memory contention due to multiple threads attempting to access the same memory location, but only one thread has the mutex lock. There are techniques such as cyclic mapping and

Table 1. Execution Times in Seconds

N	1	2	4	8	16	32	64	128
128	4.25e-04	1.53e-03	2.35e-03	5.37e-03	1.01e-02	1.96e-02	3.90e-02	7.84e-02
512	3.26e-02	2.48e-02	1.94e-02	4.22e-02	5.25e-02	8.81e-02	1.53e-01	3.28e-01
1024	2.52e-01	1.86e-01	1.13e-01	1.76e-01	2.06e-01	2.53e-01	3.62e-01	7.44e-01
2048	3.17e+00	2.90e+00	1.44e+00	1.60e+00	1.03e+00	1.08e+00	1.61e+00	1.87e+00
4096	2.65e+01	2.27e+01	1.47e+01	8.84e+00	8.35e+00	1.01e+01	7.70e+00	9.85e+00

pipelined pipelined communication as shown in [2] that reduces overheads but because of lack of time, I did not implement them.

4. Performance

In this section, I evaluate the performance benefits of parallelized code compared to the serial case. The program is compiled using gcc using a set of custom compiler flags[1]. For testing I used a permutations of matrix sizes $M = \{128,512,1024,2048,4096\}$, and number of threads $C = \{2,4,8,16,32,64,128\}$, each combination runs 5 trials. Then the median execution time is taken. Table 1 shows the total execution time (in seconds) of FE algorithm using square matrix with various matrix sizes and number of processes. Figure 1 shows the average normalized execution time, T_M (execution time $\div M^3$ for a square matrix) and we observe that serial has the worst normalized execution time. For small matrices (small M), T_M increases linearly w.r.t. to C. As we increase M, serial T_M (at $C = 1$) slowly shifts up and the slope for increasing concurrent (increasing C) decreases. This is most noticeable in $M = 2048,4096$ in Figure 1. As we increase C, regardless of matrix size, eventually, we observe a universal shift up of T_M.

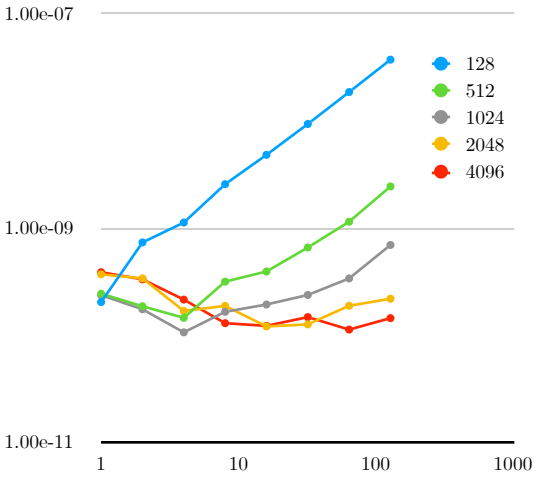

Figure 1. Log-Log plot of Normalized execution time (s) vs. Concurrency (number of threads)

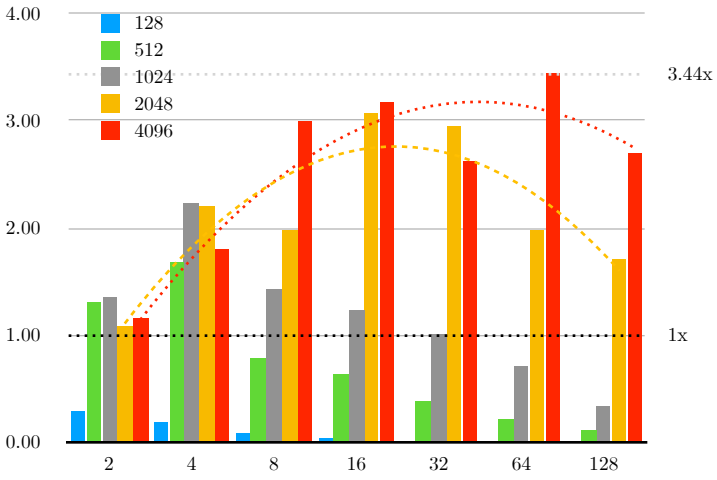

Figure 2. Speedups relative to serialized program

[1] Custom compiler flags: —O3 -m64 -pthread -march=barcelona -mtune=barcelona -ftree-vectorize -fmerge-all-constants -fmodulo-sched -faggressive-loop-optimizations -floop-interchange -funroll-all-loops -ftree-loop-distribution -funswitch-loops -finline-functions

Figure 2 shows the speedup of the FE algorithm compared to serial base case. we get the most speedup when using more processes for larger matrices. For $M < 512$, any concurrency will cause a slow down. For larger M, we observe a "hump" where there's an optimized number of concurrent threads for that matrix size M such that we get maximum speed up. This is outline in dashed line for $M = 2048,4096$. In this case, the best speedup for $M = 2048$ is using 16 threads at 3.1x speedup, and for $M = 4096$, the best speedup of 3.4x is gained from using 64 threads.

5. Conclusion

In this assignment, we explored using pthread as an alternative to parallelize FE algorithm to perform Gaussian elimination. We observe similar characteristics of speedups and execution time compared to MPI, that for large matrices, parallel programs with large number of concurrent processes excels, while suffers for small matrices due to communication overheads. However, because of shared memory overheads it is evident that *pthread* parallel programs don't perform as well as MPI programs. The best speedup is observed for a square matrix with $N = 4096$ and using $C = 64$ concurrent threads with a **3.44x** speedup.

References

(1) M. He, "CPEN 512 Assignment 1: Parallel Elimination using OpenMPI", 2020 [Accessed: 10-Oct-2020].

(2) "CoffeeBeforeArch - Overview," GitHub. [Online]. Available: https://github.com/coffeebeforearch. [Accessed: 05-Oct-2020].

(3) "Gaussian elimination," Wikipedia, 12-Sep-2020. [Online]. Available: https://en.wikipedia.org/wiki/Gaussian_elimination. [Accessed: 05-Oct-2020].

(4) B. Barney, "POSIX Threads Programming". Lawrence Livermore National Laboratory. [Online]. Available: https://computing.llnl.gov/tutorials/pthreads [Accessed: 07-Oct-2020].