Description of Implementation

This experiment discusses the performance increase in Gauss-Jordan elimination from multithreaded parallelization using OpenMP in order to find linear system solutions.

The parallel multiplication program uses input and output matrices are read from files "data_input" and "data_output", respectively. The program takes two inputs - the number of threads P and the problem size N. The linear system is solved serially, and parallelization constructs from the OpenMP library are used to take repetitive, independent serial execution paths and run them concurrently.

Implementation - Appendix `gaussj.c`

There are several functions starting from the root `gaussj`, which takes float arrays `sys_matr` and `solution` for input and output buffers, respectively. Within the `gaussj` function, an array of integers is created for tracking indices. This array is passed in and mutated through all functions, as it allows performant "swapping" of system matrix rows without actually writing. All support functions operate row-wise or column-wise, usually within loops. Because of swapping and elimination steps, synchronization is necessary because subsequent operations depend upon previous iterations to complete.

Some operations are too lightweight or carry-dependent to make parallelization worthwhile. The actual elimination steps, gaussian elimination and jordan elimination, are trivial enough (and independent enough) to warrant parallelization.

Everything is run by a makefile (See Appendix). Experimental data is gathered using a script (See Appendix Source Experiment Scripts).

Testing and Verification

The provided "serialtester.c" is used as a testing oracle, and its results are compared to the parallel version for the set of static test matrices described above. In the makefile, a `make test_run` or `test_run_async` is equivalent to a normal run, except a correctness test using the testing oracle is run on the results immediately afterward.

Performance Discussion

The controlled variables in this experiment are the number of threads P and the parameter count N. The response variable is computation time. Two problem sizes are chosen, one small (N = 50) and one large (N = 500). Threads are varied from P = 1 to P = 50. Because the number of swaps and writes depends upon the specific parameter coefficients, new matrices are only generated after a complete run from P = 1 to P = 50. There are 5 such runs for each problem size (See Appendix Tables, and the average is charted (See Appendix Charts)

As is evident in Appendix Charts 1 and 2, speedup peaks at 4 and 8 threads for both large and small problem sizes. Interestingly, as compared to the other experiments using pthreads, the performance gains significantly drop after 8 threads. For the other experiments, the slowdown for large numbers of threads was more gradual.

For large problem sizes, speedup peaked at 3.5 (large) and 1.9 (small) - see Appendix Tables. Efficiency is highest for low thread counts, with noteworthy plateaus around 8 threads. This is expected considering the machine has 4 hyperthreaded cores. Above 8 threads, the small problem exhibits a **very** steep slowdown, whereas the large problem slowdown is relatively less drastic beyond 8 threads. Regardless, thread counts higher than 8 saw significant slowdown and decreased efficiency. The small problem parallelized slower than serial execution above 8 threads.

Causes for the sharp slowdown after 8 threads are difficult to reason about, since OpenMP largely hides threading details from the programmer.

For the large problem, peak speedup was 3.5, occurring for problem size N = 500 at P = 8 and at P = 4 threads. So, if we assume that 4.0 is a approximates maximum speedup (S_{MAX}), then we can use Amdahl's law to estimate the fraction of the problem that was parallelized (F):

$$F = 1 - S_{MAX}^{-1}$$
$$F = 1 - (3.5)^{-1}$$
$$F = 0.71$$

This means that about 71% of the Floyd-Warshall algorithm was parallelized, and 29% remained as serial overhead. This overhead may result from many causes, but there are these primary candidates:

- Per-iteration synchronization (both explicit and implicit).
- Thread creation overhead from the OpenMP library

- Memory cache misses. i.e. certain problem patterns result in much more scattered read/write pattern for the arrays, because of how they are swapped or multiplied, which is less likely to take advantage of cache locality.
- Unnecessary thread joins, because the `#pragma omp parallel` directive is used for every `parallel` block, rather than creating the threads with a single parallel directive and reusing them throughout the program.

The most significant cause of serial overhead is simply a lack of parallelization. While some operations had prohibitive loop-carry dependencies or synchronization requirements - there are two places where further parallelization is possible but not done. For both elimination steps, gaussian and jordan, there are fairly simple nested for loops. The iteration spaces are not rectangular, and there exist necessary synchronization points, so the OpenMP `collapse` directive cannot apply. However, with row-wise synchronization in the gaussian elimination and column-wise synchronization in the jordan elimination, further parallelization could have been had.

Conclusion and Experiences

With regards to performance, as is evident from the speedup and efficiency charts (see Appendix Chart 1 and Chart 2), for both problem sizes:

- 4 and 8 threads results in the largest increase in speedup, and 4 threads resulted in the best tradeoff for efficiency
- Small problems were not parallelized very much, suffering slowdown with more than 8 threads.

Using Amdahl's Law for analysis, the estimated maximum speedup of 3.5 translates to a parallelized fraction of 71%. Several sources of the remaining 29% were identified, including synchronization, IO and OS overhead, as well as remaining unparallelized sections..

With regards to experimental quality, the data was gathered from a lab machine that was also running a desktop environment, daemons, and possible other user processes. Processor scheduling may still have caused high timing variance. OpenMP also hides a good deal of the thread management, so there are potential uncontrolled experimental variables within the OpenMP framework.

With regards to software quality, it was very evident that a **clean** serial implementation was necessary to consider using OpenMP. Variable CRUD cycles had to be very obviously laid out for both human and machine in order to see where certain directives applied.

Kirby Banman
1173895

Appendix

Chart 1 - Speedup

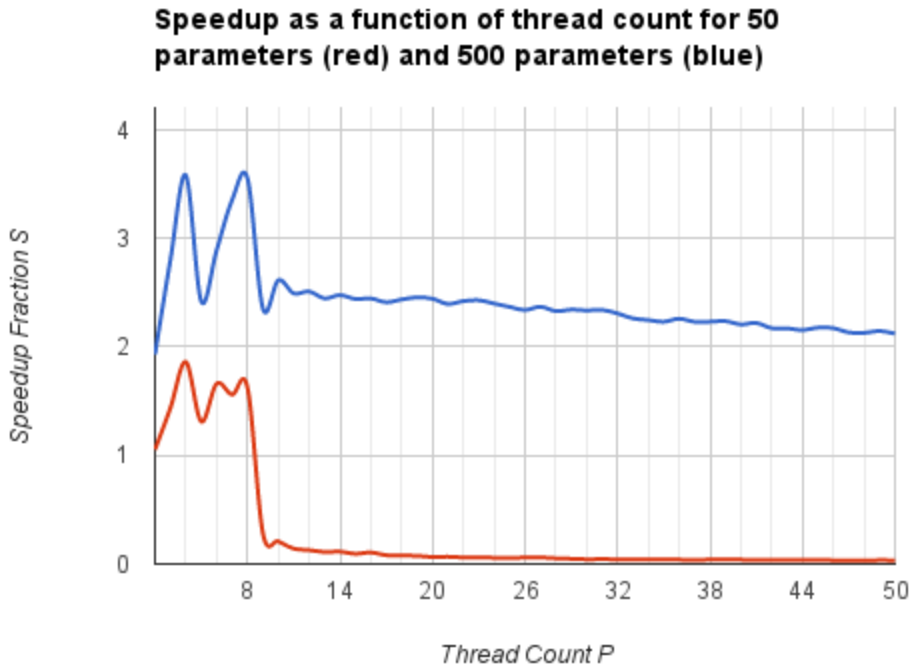

Chart 2 - Efficiency