Terrence Alsup

Spring 2019: Advanced Topics in Numerical Analysis: High Performance Computing Assignment 2 (due Mar. 11, 2019)

1. **Finding Memory bugs.**

   The files `val_test01_solved.cpp` and `val_test02_solved.cpp` contain the bug fixes as well as comments about what went wrong in the original programs.

2. **Optimizing matrix-matrix multiplication.**

   The machine used for these computations was Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz (CIMS desktop) with 4 cores. Tables 1, 2, and 3 below shows the timings for two different arrangements of the three for loops. The loop arrangement (i, p, j) means we first loop over $0 < i < m$, then $0 < p < k$, and finally $0 < j < m$. Because the matrix is stored in major column order, and since memory can be read faster sequentially, we would expect that going down each column one by one is fastest. This corresponds to the loop arrangement (j, p, i). On the other hand, if we jump around in memory by going across the rows one by one we would expect worse results. This coincides with the loop arrangement (i, p, j) and indeed we see that it is much worse in every category.

Dimension	(i, p, j)	(j, p, i)
128	3.08	0.24
512	5.54	0.28
1024	11.32	0.38

Table 1: Wall clock time (s) for two different for loop arrangements.

Dimension	(i, p, j)	(j, p, i)
128	0.65	8.46
512	0.39	7.54
1024	0.19	5.61

Table 2: Gflop/s for two different for loop arrangements.

Dimension	(i,p,j)	(j,p,i)
128	10.39	135.34
512	6.20	120.60
1024	3.03	89.73

Table 3: GB/s for two different for loop arrangements.

We can also look at using block matrix multiplication as well as OpenMP to further speed up our computations if the dimension is large. Tables 4, 5, and 6 below show results for different block sizes with and without parallelization. We use 4 threads with OpenMP. Smaller block sizes fit more easily into cache, which is much faster. Parallelizing also allows us to use bigger block sizes. Although the timings are not displayed in the tables below, we note that if the block size is too big then there is a noticeable drop-off of performance, presumably because the blocks no longer fit in cache.

Dimension	128	256	512	128 + OpenMP	256 + OpenMP	512 + OpenMP
512	0.32	0.25	0.24	0.32	0.25	0.24
1024	0.32	0.26	0.26	0.33	0.26	0.25
1536	1.09	0.87	0.88	1.10	0.87	0.84

Table 4: Wall clock time (s) for different block sizes with and without OpenMP.

Dimension	128	256	512	128 + OpenMP	256 + OpenMP	512 + OpenMP
512	6.81	8.64	8.85	6.68	8.63	8.99
1024	6.68	8.38	8.34	6.60	8.38	8.67
1536	6.68	8.37	8.27	6.61	8.38	8.64

Table 5: Gflop/s for different block sizes with and without OpenMP.

Dimension	128	256	512	128 + OpenMP	256 + OpenMP	512 + OpenMP
512	108.88	138.30	141.54	106.92	138.16	143.84
1024	106.93	134.06	133.37	105.54	134.16	138.77
1536	106.88	133.88	132.27	105.68	134.03	138.20

Table 6: GB/s for different block sizes with and without OpenMP.

3. **Finding OpenMP bugs.** The files `omp_solved{2,...,6}.c`, contain the bug fixes as well as comments about what went wrong. They can all be compiled with the makefile provided in the repository.

4. **OpenMP version of 2D Jacobi/Gauss-Seidel smoothing.** The files `jacobi2D-omp.cpp` and `gs2D-omp.cpp` implement the Jacobi and Gauss Seidel methods for solving the discretized version of Poisson's equation with $f \equiv 1$ using OpenMP. Table 7 below shows the timings of these methods when we use a different number of threads. The results appear

to be more or less the same, so perhaps the problem was too small. We do note however that if only one thread is used we get slightly slower results.

Threads	Jacobi	Gauss Seidel
2	16.83	155.77
4	17.62	154.53
8	17.65	153.92

Table 7: Wall clock time (s) for 10000 iterations of Jacobi and Gauss Seidel methods for different numbers of threads. Note that neither came close to converging in any case. Here $N = 100$ for the number of discretization points in each dimension.