High Performance Computing: Homework 2

Paul Beckman

1 Finding memory bugs

For `val_test01`, we make the following changes

- line 80: change `<=` to `<` to avoid indexing out of bounds
- line 86: change `delete []` to `free` to match original malloc

For `val_test02`, we add the initialization block

```
for ( i = 6; i < 10; i++ )
{
  x[i] = 0;
}
```

to avoid copying and printing uninitialized variables.

2 Optimizing matrix-matrix multiplication

2.1 Loop ordering

The given loop ordering gives the following timings on an Intel(R) Xeon(R) CPU @2.53GHz processor (`crackle1`)

Dimension	Time	Gflop/s	GB/s	Error
16	0.732706	2.729614	43.673817	0.000000e+00
208	0.772579	2.609128	41.746043	0.000000e+00
400	0.769613	2.661077	42.577230	0.000000e+00
592	0.783120	2.649334	42.389349	0.000000e+00
784	1.083208	2.669241	42.707854	0.000000e+00
976	1.373894	2.706800	43.308797	0.000000e+00
1168	1.241814	2.566268	41.060295	0.000000e+00
1360	2.242826	2.243112	35.889800	0.000000e+00
1552	3.530674	2.117618	33.881884	0.000000e+00
1744	5.011761	2.116796	33.868742	0.000000e+00
1936	6.864092	2.114282	33.828515	0.000000e+00

The other ordering where the inner loop is over columns performs similarly. In contrast, other loop orderings give slower timings. For example, if we exchange the `i` and `p` variables so that the inner loop is over the shared dimension `k`, we obtain

Dimension	Time	Gflop/s	GB/s	Error
16	1.318957	1.516352	24.261640	0.000000e+00
208	1.597254	1.262014	20.192218	0.000000e+00
400	2.103392	0.973665	15.578647	0.000000e+00
592	1.986053	1.044658	16.714533	0.000000e+00
784	2.674992	1.080879	17.294057	0.000000e+00
976	3.449245	1.078165	17.250648	0.000000e+00

1168	3.529046	0.903028	14.448448	0.000000e+00
1360	6.411109	0.784718	12.555486	0.000000e+00
1552	10.383032	0.720080	11.521286	0.000000e+00
1744	15.729447	0.674460	10.791354	0.000000e+00
1936	21.918752	0.662110	10.593762	0.000000e+00

This can be explained by the column major ordering of the matrix storage, as having the column variable in the inner loop allows one or more columns to be cashed during computation, reducing memory access requirements.

2.2 Blocking

After blocking (with `BLOCK_SIZE` 16) we see immediate speedups and increased bandwidth, as more arithmetic is done using cached matrix entries

Dimension	Time	Gflop/s	GB/s	Error
16	0.000004	2.278087	36.449388	0.000000e+00
208	0.006020	2.989627	47.834033	0.000000e+00
400	0.042895	2.984037	47.744597	0.000000e+00
592	0.139096	2.983178	47.730855	0.000000e+00
784	0.328163	2.936894	46.990312	0.000000e+00
976	0.629938	2.951764	47.228223	0.000000e+00
1168	1.133763	2.810842	44.973469	0.000000e+00
1360	1.950271	2.579597	41.273548	0.000000e+00
1552	2.978048	2.510576	40.169218	0.000000e+00
1744	4.221113	2.513289	40.212627	0.000000e+00
1936	5.799208	2.502519	40.040300	0.000000e+00

In this regime, smaller block sizes appear to be best for speed. Using `BLOCK_SIZE` 4 gives

Dimension	Time	Gflop/s	GB/s	Error
4	0.000001	0.139891	2.238251	0.000000e+00
204	0.004475	3.794272	60.708348	0.000000e+00
404	0.034631	3.808085	60.929359	0.000000e+00
604	0.117891	3.738184	59.810939	0.000000e+00
804	0.281168	3.696848	59.149565	0.000000e+00
1004	0.548550	3.689902	59.038437	0.000000e+00
1204	0.951046	3.670355	58.725673	0.000000e+00
1404	1.560903	3.546135	56.738164	0.000000e+00
1604	2.343953	3.521229	56.339658	0.000000e+00
1804	3.358989	3.495675	55.930796	0.000000e+00

which shows improvement over `BLOCK_SIZE` 16. If we increase the block size above 16, we see even slower results. This is a bit surprising, as I would expect a larger block size to "just barely fit" in the cache and thus give optimal performance.

2.3 Parallelism

Following the discussion in lecture, I tried reordering the loops in each block so that the shared dimension `k` is the inner loop and using `collapse(2)` on the outer two loops as they are perfectly nested. However, this appears to lead to serious slowdowns.

Alternatively, taking the most naive approach and simply slapping a `parallel for` in front of the first outer loop gives decent speedups. For example, with a bit larger `BLOCK_SIZE` 16 and 16 threads, we obtain

Dimension	Time	Gflop/s	GB/s	Error
16	0.000158	0.051758	0.828133	0.000000e+00
208	0.001721	10.457425	167.318795	0.000000e+00
400	0.010832	11.817296	189.076738	0.000000e+00

```
 592   0.034267  12.109437 193.750996 0.000000e+00
 784   0.078501  12.277328 196.437248 0.000000e+00
 976   0.145595  12.771261 204.340171 0.000000e+00
1168   0.261762  12.174537 194.792598 0.000000e+00
1360   0.454422  11.071019 177.136298 0.000000e+00
1552   0.739173  10.114843 161.837495 0.000000e+00
1744   1.074908   9.869567 157.913073 0.000000e+00
1936   1.463838   9.914097 158.625553 0.000000e+00
```

which is notably faster than the serial blocked implementation, but far from linear strong scaling.

3

See code.

4

2.4 Jacobi

We observe the following timing results for 100 iterations of our Jacobi implementation

SERIAL		OMP_NUM_THREADS=4		OMP_NUM_THREADS=16	
N	Time	N	Time	N	Time
8	0.000025	8	0.000872	8	0.001787
16	0.000053	16	0.000770	16	0.001614
32	0.000176	32	0.000863	32	0.001780
64	0.000762	64	0.001286	64	0.002097
128	0.003575	128	0.003526	128	0.003122
256	0.014993	256	0.010747	256	0.008068
512	0.060531	512	0.042753	512	0.026186
1024	0.518867	1024	0.177029	1024	0.158549
2048	2.126492	2048	1.385724	2048	1.609879
4096	8.621786	4096	6.183262	4096	4.907035
8192	36.595984	8192	21.092580	8192	19.124445

We note that the serial code is faster for $N < 128$, with more notable performance gains using parallelism for larger matrices as expected. We also note very minor performance gains between 4 and 16 threads, and even a slow down with more threads for $N < 4096$, indicating that we are far from a linear strong scaling implementation.

2.5 Gauss-Seidel

Running again on an Intel(R) Xeon(R) CPU @2.53GHz processor (`crackle1`), we observe the following timing results for 100 iterations of our Gauss-Seidel implementation with red-black coloring

SERIAL		OMP_NUM_THREADS=4		OMP_NUM_THREADS=16	
N	Time	N	Time	N	Time
8	0.000021	8	0.000810	8	0.001684
16	0.000075	16	0.000574	16	0.001753
32	0.000253	32	0.000628	32	0.001769
64	0.000933	64	0.000936	64	0.002106
128	0.003881	128	0.001570	128	0.002507
256	0.016518	256	0.004977	256	0.004842
512	0.067685	512	0.021225	512	0.017321
1024	0.392920	1024	0.114471	1024	0.069080
2048	1.824175	2048	1.297420	2048	1.594579
4096	8.255579	4096	4.093789	4096	4.347041
8192	32.163247	8192	17.782354	8192	21.512998

Similarly to Jacobi, we note that the serial implementation is faster for $N < 64$. However in these results there is no clear comparison between 4 and 16 threads; the results are fairly similar. Gauss-Seidel does appear to parallelize slightly better than Jacobi, as we see nearly a factor of two speedup using 4 threads.