HPC Hw2 (Zhe Chen)

Machine configuration

All results are got by running on my office's machine 'blob'. It has a 4 core CPU, Intel Xeon E5-1603, 2.80GHz. Max memory size is 375GB, max memory bandwidth is 31.4GB/s. It uses Sandy Bridge EP, so operations per cycle is 8. Thus, theoretic FLOPS/s is 4 Cores * 2.80GHz * 8 operations/cycle = 89.6 GFLOPs/s

Dubug

Problem 1 and 3 are both to debug the codes. All are solved and renamed as required.

All the bugs I found and solved is commented around with `//BUG:` and explained how that is a bug and why that solution works. Just search the codes with `BUG`.

Makefile

All codes are compiled with g++, "O3" flags, openmp. Command is like:

```
g++  -std=c++11 -O3  -march=native -fopenmp  gs2D-omp.cpp -o gs2D-omp
```

I wrote makefile, so just type make and all codes are compiled automatically.

Problem 2: Optimization Matrix-Matrix Multiplication

Rearrange loops

Intutively, we need to make cache hits more and flop-rate bigger to make it faster. As for loops order, this means we need to read data sequentially as much as possible to lower the cost of reading. Thus, for problem $C = A * B$, it's easy to get the idea that we should loop rows of A first, say i, then rows of B, say p, then column of C, say j.

I took a test to verify my inference. And it turns out that (i,p,j) is indeed the best order. I listed some typical results in the table below. (i,p,j) order is much better, which is consistent with our prediction.

Dimension	Time	Gflop/s	GB/s	Error
(i,p,j)				
500	0.521711	4.312734	17.319940	0.000000e+00
1000	0.938626	4.261547	17.080280	0.000000e+00
1500	2.285623	2.953243	11.828724	0.000000e+00
(p,i,j)				
500	1.560516	1.441831	5.790394	0.000000e+00
1000	2.824688	1.416086	5.675671	0.000000e+00
1500	7.559161	0.892956	3.576587	0.000000e+00
(i,j,p)				
500	0.617410	3.644257	14.635335	0.000000e+00
1000	1.106277	3.615732	14.491854	0.000000e+00
1500	3.096304	2.180019	8.731701	0.000000e+00

Block

I tested many problems with different scales N, and found that flop rate was steady at small N, and then decreases at some point around N=1000. So I think we reached the bottleneck of cache memory. Thus, I implemented block-wise multiplication to optimize this.

The key parameter here is size of block, `BLOCK_SIZE`. If it's too small, we wasted too much efforts on segment matrix and do not take full advantage of whole cache. If it's too big, it would reach the bottleneck. I did some tests to look for the best `BLOCK_SIZE`.

First, I set N=2048, which is greater than the turning point so that we can see the effect of memeory reading, and `BLOCK_SIZE` to be multiples of 4 and divisor of 2048. Results turn out to be `BLOCK_SIZE=256` is the best. I listed the time each test took in the table below.

BLOCK_SIZE	Time	Gflop/s	GB/s	Error
16	5.948961	2.887877	11.562790	0.000000e+00
64	4.718624	3.640864	14.577679	0.000000e+00
128	4.387855	3.915323	15.676585	0.000000e+00
256	4.051400	4.240477	16.978472	0.000000e+00
512	3.927520	4.374228	17.514000	0.000000e+00
1024	5.858231	2.932603	11.741869	0.000000e+00

OpenMP

I also implented parrallel computing to accellerate the code. Generally, I tried two versions, one with OpenMP only, the other one with OpenMP and blocking trick(use optimized best setting `BLOCK_SIZE=512`).

The results I got are as follows,

1.For the OpenMP only version, flop rate increases until it reaches N=512, whcih is bottleneck of memory. Speed is much quicker than non-parallel code.

2.For the OpenMP+block version, when N is smaller than 512, it's similar to version 1. However, block trick helps it get 4 times greater speed when N=4*512, since my machine has 4 cores, which is consistent with theory. And that's where the highest speed we got. After that, it will decreases since we don't have enough cores and we encountered memory reading problem, which makes sense.

Dimension	Time	Gflop/s	GB/s	Error
1. OpenMP only				
512	0.497402	4.317403	17.337072	0.000000e+00
1024	0.518087	4.145024	16.612479	0.000000e+00
1536	2.462841	2.942844	11.786702	0.000000e+00
2048	5.850661	2.936398	11.757062	0.000000e+00
2. OpenMP+Block				
512	0.556471	3.859109	15.496734	0.000000e+00
1024	0.283655	7.570757	30.342173	0.000000e+00
1536	0.639638	11.331023	45.383108	0.000000e+00
2048	1.195404	14.371598	57.542530	0.000000e+00
2560	3.525609	9.517343	38.099114	0.000000e+00

Problem 4: OpenMP Version of 2D Jacobi/GS

We use OpenMP to parallel 2D Jacobi and GS. Jacobi method is naturally parallizable. However, GS method need to be modified to Black-Red point version to make it possible to parallel half grids at one time.

To see the speed of parallel computing, we fix the iteration number of differnt N to be 10000 and list results in table below. General, we got scaling as $O(N^2)$. And we also get good linear scaling with number of threads, which means our parallel implementation is good.

N	# of threads	Jacobi	GS-Black-Red
100	1	1.81787	1.86796
100	2	0.983745	0.99846
100	3	0.68978	0.712587
100	4	0.552354	0.570675
200	1	7.23864	7.30773
200	2	3.74556	3.75039
200	3	2.52869	2.5944
200	4	1.91781	2.01078