High Performance Computing Assignment 2

Yijian Xie

March 11 2019

The processor I use is i5-7267U @ 3.10GHz. It has 2 processors and 4 threads.

1 Finding Memory Bugs

1.1 val_test_01

Should malloc memory space for $n+1$ integers; `free()` matches with `malloc()`.

1.2 val_test_02

Should initialize all the data in the array.

2 Optimizing Matrix-Matrix Multiplication

2.1 Different Loop Arrangements

`MMult0()` (j, p, i) and the arrangement(p, j, i) has the best performance among all 6 possibilities of rearrangement. The performance of each loop arrangement mainly depends on the innermost loop. As the three array index in the loop are $(i+p*m)$, $(p+j*k)$, $(i+j*m)$, i needs to be the innermost loop to guarantee spatial locality of all three variables.

2.2 Blocked Multiplication

I implemented the code refering to the pseudocode[1].
Each row represents different size N of the matrix, and each column represents the block size(1 means no blocking). We can see that when block size = 32, the algorithm has the best overall performance.

[1] `http://web.cs.ucdavis.edu/~bai/ECS231/optmatmul.pdf`, Page 20

	1	4	8	16	32	64	128	256
256	0.01	0.03	0.02	0.02	0.00	0.00	0.00	0.01
512	0.04	0.27	0.12	0.13	0.02	0.02	0.02	0.04
768	0.14	1.05	0.41	0.43	0.04	0.06	0.09	0.10
1024	0.34	2.01	1.22	1.03	0.11	0.16	0.17	0.18
1280	0.62	3.64	2.60	2.46	0.22	0.40	0.40	0.34
1536	1.12	9.63	3.80	3.58	0.44	0.43	0.42	0.63
1792	2.12	12.81	6.37	6.32	0.64	0.65	0.67	1.18

Table 1: Performance of different block sizes

2.3 OpenMP Blocked Multiplication

This version add `#pragma omp parallel for` before the outermost `for` loop. We can see there is significant improvement of the performance. (1x faster when block size = 32)

	1	4	8	16	32	64	128	256
256	0.01	0.01	0.01	0.01	0.00	0.00	0.00	0.01
512	0.04	0.10	0.08	0.06	0.01	0.01	0.01	0.02
768	0.14	0.78	0.18	0.20	0.02	0.03	0.05	0.07
1024	0.34	0.95	0.45	0.83	0.06	0.10	0.08	0.10
1280	0.62	2.02	1.67	0.94	0.11	0.14	0.15	0.26
1536	1.12	4.47	1.53	1.92	0.19	0.23	0.28	0.33
1792	2.12	6.41	2.45	3.75	0.36	0.37	0.48	0.63

Table 2: Performance of different block sizes(OpenMP version)

3 Finding OpenMP Bugs

3.1 omp_bug2

`tid` and `total` should be private. `total` should be type `double` to guarantee precision.

3.2 omp_bug3

Only one thread can reach the `#pragma omp barrier` in function `print_results()` at one time, should not add barrier here.

3.3 omp_bug4

The array `a[N][N]` is too large to fit in the thread stack, so I changed `N` to a smaller value.

3.4 omp_bug5

If the first thread gets `locka` in the first section and the second thread gets `lockb` in the second section, deadlock will occur. So I changed the order of acquiring lock in the second section.

3.5 omp_bug6

The variable `sum` should be shared by all threads, so I removed all the declaration of `sum` and set it as a global variable.

4 OpenMP Version of 2D Jacobi/Gauss-Seidel Smoothing

Each row represents different N and each column represents different thread number. The iteration is executed 2000 times.

4.1 Jacobi

	1	2	4
100	0.03	0.11	0.17
200	0.12	0.16	0.21
500	0.88	0.75	0.72
1000	3.88	3.25	3.01
2000	30.65	27.60	26.56

Table 3: Performance of different number of threads

4.2 Gauss-Seidel

	1	2	4
100	0.03	0.09	0.15
200	0.13	0.17	0.21
500	0.83	0.67	0.77
1000	3.66	2.67	2.71
2000	16.85	15.30	12.34

Table 4: Performance of different number of threads

4.3 Conclusion

From the timing we can see that for larger N, there is minor improvement (about 25%) when we use more threads. However, when N is small, the overhead of scheduling several threads is quite large compared to the running time.