Spring 2019: Advanced Topics in Numerical Analysis: High Performance Computing Assignment 2

Yongyan Rao, yr780@nyu.edu
(Dated: March 10, 2019)

. CPU SPECIFICATION

The processor used in the experiment was located in one of Courant's servers, crunchy5.cims.nyu.edu, with the following specs. The servers has 2 Intel Xeon E5630 processors, each processor has 4 cores, and each core supports 2 threads i.e., 2 virtual cores. Therefore, from the software point of view, the server has $2 \times 4 \times 2 = 16$ cores.

processor	: 0 to 15
vendor_id	: GenuineIntel
cpu family	: 6
model	: 44
model name	: Intel(R) Xeon(R) CPU E5630 @ 2.53GHz
stepping	: 2
microcode	: 0x1f
cpu MHz	: 2526.997
cache size	: 12288 KB
physical id	: 0, 1
siblings	: 8
core id	: 0, 1, 9, 10
cpu cores	: 4
apicid	: 32, 0, 34, 2, 50, 18, 52, 20, 33, 1, 35, 3, 51, 19, 53, 21
initial apicid	: 32, 0, 34, 2, 50, 18, 52, 20, 33, 1, 35, 3, 51, 19, 53, 21
fpu	: yes
fpu_exception	: yes
cpuid level	: 11
wp	: yes
flags	: *omitted*
bogomips	: 5053.99, 5053.78
clflush size	: 64
cache_alignment	: 64
address sizes	: 40 bits physical, 48 bits virtual
power management	:

TABLE I: CPU specs.

I. 1. FINDING MEMORY BUGS

Please refer to submitted files. A brief description of the issues is as follows.

- malloc should be paired with free, and new[] should be paired with delete[].
- Dynamically allocated variables need to be initialized before been referred to.

II. 2. OPTIMIZING MATRIX-MATRIX MULTIPLICATION

Three (3) matrix multiplication implementations are presented and compared as below. They were implemented with naive OpenMP parallelism, block/tiling technique, and the combination of OpenMP and block/tiling, respectively.

- The naive OpenMP implementation used the matrix multiplication definition with16 threads, and the outermost (ij) iterations were parallelized.
- The block/tiling implementation used 16 as block size, and the innermost iteration indices were with the kij order.
- The combination implementation used both techniques, where OpenMP was used to parallelized the outermost block iterations (IJ).

Dimension	OpenMP	Block	OpenMP+Block
16	6.227094	0.816691	1.975884
64	0.270586	0.52472	0.139873
112	0.220894	0.709783	0.152585
160	0.409025	0.706535	0.125173
208	0.378959	0.634167	0.116605
256	0.543038	0.631811	0.117298
304	0.471516	0.521371	0.115308
352	0.457483	0.705865	0.117642
400	0.946045	0.862554	0.119515
448	1.687395	0.939999	0.125995
496	1.80889	0.907618	0.12939
544	1.9585	0.846285	0.135635
592	1.818136	0.823583	0.12052
640	1.724149	0.786678	0.119489
688	2.395068	0.927474	0.14833
736	1.066617	0.937604	0.136798
784	1.315492	1.044329	0.164911
832	1.441192	0.796158	0.130011
880	1.544372	0.953048	0.154036
928	1.360137	1.043309	0.179728
976	1.825312	1.207031	0.209551
1024	0.687303	0.773311	0.121235
1072	0.75693	0.843546	0.128064
1120	1.094862	1.074942	0.161223
1168	0.777846	1.074029	0.180988
1216	1.305883	1.214914	0.190664
1264	0.914389	1.331532	0.219023
1312	1.467905	1.597892	0.243967
1360	0.854457	1.608285	0.287719
1408	1.76195	1.795841	0.290001
1456	1.340108	1.97443	0.346057
1504	2.181897	2.16542	0.380633
1552	1.514282	2.361868	0.428185
1600	2.524444	2.654868	0.459159
1648	1.80883	2.766734	0.515133
1696	3.487071	2.952412	0.540911
1744	3.221339	3.282783	0.589136
1792	3.58051	3.572614	0.617847
1840	4.580765	3.897861	0.656165
1888	5.261695	4.18263	0.725827
1936	4.660033	4.399983	0.773285
1984	5.71426	4.726358	0.75702

TABLE II: Time measures (second) for three (3) implementations of matrix multiplication.

From the comparison above, one can conclude that the combination implementation had significantly better performance than the other single technique implementations. The submitted code included the combination implementation and the block/tiling implementation.

III. 3. FINDING OPENMP BUGS

Please refer to submitted files. A brief description of the issues is as follows.

- The thread identification number should be private for each thread.ls
- A synchronization barrier should be reachable to every thread.
- Private pointers should be explicitly initialized by each thread.
- Among threads, the order of applying locks should be the same to avoid deadlock.
- A reduction sum variable should be shared among threads.

IV. 4. OPENMP VERSION OF 2D JACOBI/GAUSS-SEIDEL SMOOTHING

For the OpenMP implementations of both Jacobi and Gauss-Seidel algorithms, the paradigm of spawning and joining threads in each outermost iteration has been used, rather than spawning threads once before the iterations begin and joining them once after iterations end. The reason is that although the second paradigm avoids overhead for multiple times of spawning and joining, it introduces at least one synchronization barrier for each iteration, which balances out its original advantage. In my preliminary experimentation using Jacobi algorithm, the second paradigm performed even slightly worse than the first one.

nthreads \ n	32	64	128	256	512	1024
4	0.054681	0.169285	0.615691	1.604700	5.911598	20.706517
8	0.043413	0.096912	0.194700	0.716959	2.797312	11.014795
16	0.056111	0.108559	0.205713	0.794162	2.762089	10.818951
32	0.583886	0.617420	0.683782	1.623447	5.800639	20.485951
64	1.133700	1.153689	1.320243	2.488874	7.465440	21.053952

TABLE III: Jacobi: Time measures (second) with 4000 iterations.

nthreads \ n	32*	64	128	256	512	1024
4	0.053811	0.193765	0.676608	1.909804	6.419415	24.051801
8	0.045949	0.145819	0.271251	0.821607	3.104475	12.300551
16	0.064752	0.133320	0.243710	0.912089	3.228403	12.217446
32	0.597240	0.825713	0.911695	2.030506	6.655405	24.468986
64	1.154514	1.559127	1.731044	3.093026	8.895311	25.561222

TABLE IV: Gauss-Geisel: Time measures (second) with 4000 iterations.
*Because of early convergence, $n = 32$ cases only had 3029 iterations.