Sheet3
07.Jan.2020 Abraham Gutierrez && Huan Chen

Exercise 1, inner product

(1) run the code and get the following result:

```
g++  main.o mylib.o  -g -flto -fopenmp -o main.GCC_
./main.GCC_
Number of available processors: 64
The return of command 'omp_in_parallel': 1
Checking command line parameters for: -n <number>

N = 40000000

###############################################################################
Code    :  ./main.GCC_
Compiler:  Gnu 9.2.0  C++ standard: 201703
Parallel:  OpenMP 4.5 ---> 64 Threads
Date    :  Dec 15 2019  20:37:52
###############################################################################
C++: Hello World from thread 0 / 64
C++: Hello World from thread 38 / 64
C++: Hello World from thread 25 / 64
C++: Hello World from thread 18 / 64
C++: Hello World from thread 20 / 64
C++: Hello World from thread 14 / 64
C++: Hello World from thread 61 / 64
C++: Hello World from thread 23 / 64
C++: Hello World from thread 8 / 64
C++: Hello World from thread 62 / 64
C++: Hello World from thread 52 / 64
C++: Hello World from thread 26 / 64
C++: Hello World from thread 58 / 64
.
.
.
C++: Hello World from thread 59 / 64

   64   threads have been started.
Memory allocation
0.6 GByte Memory allocated

Start Benchmarking

 <x,y> = 4e+07
```

```
Total time in sec.      :2.6
Time/loop in sec.       : 0.013
GFLOPS                  : 5.7
GiByte/s                : 46

  Try the reduction with an STL-vektor
done
2016  2080  2144  2208  2272  2336  2400  2464  2528  2592  2656  2720  2784  2848  2912  2976
3040  3104  3168  3232  3296  3360  3424  3488  3552  3616  3680  3744  3808  3872  3936  4000
4064  4128  4192  4256  4320  4384  4448  4512  4576  4640  4704  4768  4832  4896  4960  5024
5088  5152  5216  5280  5344  5408  5472  5536  5600  5664  5728  5792  5856  5920  5984  6048
6112  6176  6240  6304  6368  6432  6496  6560  6624  6688  6752  6816  6880  6944  7008  7072
7136  7200  7264  7328  7392  7456  7520  7584  7648  7712  7776  7840  7904  7968  8032  8096
8160  8224  8288  8352
```

(2) Try several schedule types and chunk sizes: **schedule(kind , chunk size)**

N = 4*10^7, NLOOPS = 200.

	Total time (sec)	GFLOPS	GiByte/s
No schedule	2.6	5.7	46
schedule(auto)	2.5	6	48
schedule(static)	2.4	6.2	50
schedule(static, 250)	4.8	3.1	25
schedule(static, 25000)	2.7	5.6	45
schedule(dynamic)	/	/	/
schedule(dynamic, 250)	4.9	3	24
schedule(dynamic, 25000)	2.6	5.8	46
schedule(guided, 250)	2.6	5.7	46
schedule(guided, 25000)	2.6	5.8	46

Conclusion:

- It’s important to find the proper chunk size, e.g. 25000 is better than 250
- For dynamic, if we do not specify chunk size, it sets to defaults (one), which is seriously inefficient in this case.
- In this simple for loop, auto or default schedule is behaving well enough.

(3) Calculate the speedup for different number of cores: **omp set num threads( )**

N = 4*10^7, NLOOPS = 200;
No schedule.
The computer used to do the testing has 32 cores with 2 threads for each core i.e. 64 threads.

Numbet of threads	Timing in sec	GFLOPS	GiByte/s
1	5.534	2.693	21.54
2	4.024	3.703	29.62
4	2.721	5.476	43.8
8	3.014	4.943	39.55
16	3.01	4.951	39.6
32	2.852	5.225	41.8
64	2.946	5.057	40.46

cout << "Number of available processors: " << omp_get_num_procs() << endl;
Number of available processors: 64

The return of command 'omp_in_parallel': 1
The result means: number of nested active parallel regions (active-levels-var) is larger than zero (otherwise returns 0).

Appending without order VS appending with order
Number of Loops: 200
n = 3000000

Number of Threads	Withour order, total time (s)	With order, total time (s)
1	0.6472	0.6385
2	2.568	2.074
4	8.423	10.26
8	25.41	27.22

Exercise 2, data I/O

First all, we produced a file.txt with 2*10^8 double elements inside;

We didn’t parallize the data ‘reading’ process (does that worth it and how), time for that is: 37.8032s
So the following comparison is only for the (max, min, mean values) calculation part, NLOOPS = 20, correct result is: 1 1000 500.5 369.5 133.6 288.7

	Total time in sec	Time per loop in sec
Non-parallized	256.4	12.82
Parallized	8.025	0.4013

Efficiency = 256.4/8.025/64 = 0.4992.

Exercise 3, count_goldbach

Version 1: g++ -O3, number of used threads: nThreads = 64

	Non-parallelization time in sec, t1	Parallelization time in sec, t2	Efficiency= t1/t2/nThreads
n=10,000	0.001289	0.009314	0.0022
n=100,000	0.164	0.02632	0.097
n=400,000	1.361	0.1348	0.16
n=500,000	2.042	0.1808	0.18
n=1,000,000	7.792	0.6222	0.20
n=2,000,000	28.25	3.246	0.14
n=5,000,000	155.6	28.73	0.085

Version 2: g++ -O3

n=10.000.000	-O3
1	555.5 sec
2	273.9 sec
4	136 sec
8	69.51 sec

16	42.7 sec
32	24.29 sec
64	24.33 sec

Exercise 4

Matrix-Vector Multiplication
M times N matrix: M = 60000, N = 9000
Number of Loops: 1.5e+02

Matrix-Matrix Multiplication
Matrix Dimensions (M times L and L times N): M = 2000, L = 2000, N = 2000
Number of Loops: 5

Polynomial Evaluation
Size of vector x: 5000
Polynomial size (N) : 200000
Number of Loops: 10

TimesMatrixVector = [91 48 28 18 18 22 20];
TimesMatrixMatrix = [19 15 9.8 7.3 4.3 2.2 1.6];
TimesPolynomialEvaluation = [14 7.2 3.6 2 1.1 0.61 0.54];

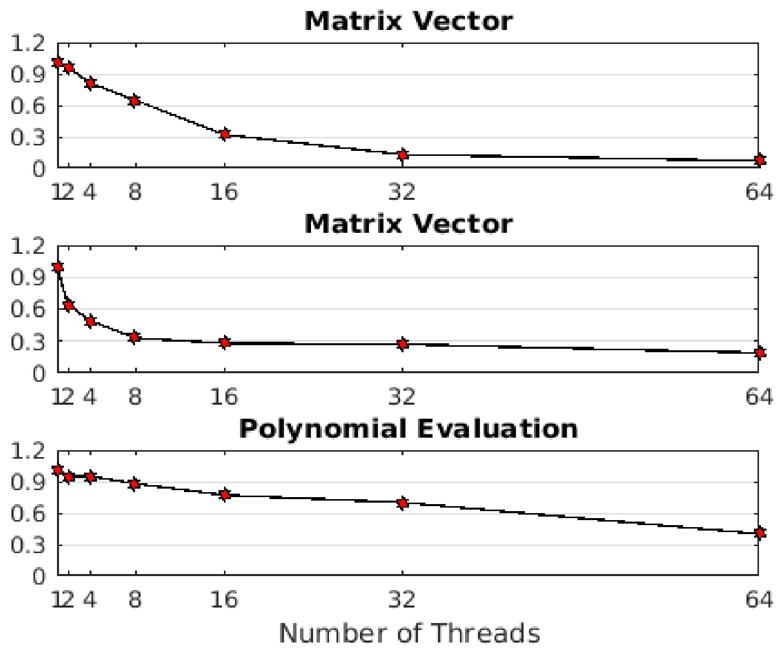
EfficiencyMatrixVector = [1 0.96 0.81 0.65 0.32 0.13 0.07];
EfficiencyMatrixMatrix = [1 0.64 0.49 0.33 0.28 0.27 0.19];
EfficiencyPolynomialEvaluation = [1 0.95 0.95 0.88 0.77 0.7 0.4];

GFlopsMatrixVector = [0.011 0.021 0.036 0.057 0.056 0.046 0.05];
GFlopsMatrixMatrix = [0.77 0.98 1.5 2.1 3.5 6.7 9.4];
GFlopsPolynomialEvaluation = [0.2 0.39 0.77 1.4 2.5 4.6 5.2];

GipPerSecMatrixVector = [0.18 0.34 0.57 0.92 0.9 0.74 0.8];
GipPerSecMatrixMatrix = [12 16 24 33 56 1.1e+02 1.5e+02];
GipPerSecPolynomialEvaluation = [3.8 7.2 14 27 47 86 97];

NumberofThreads = [1 2 4 8 16 32 64];

Efficiency plot



Two versions of Polynomial Evaluation comparison
Size of vector x: 5000
Polynomial size (N) : 200000
Number of Loops: 10

Number of Threads	Parallized outer loop, total time (s)	Parallized inner loop, total time (s)
1	13.4629	15.0908
2	6.79469	7.69313
4	3.47619	4.03413
8	1.97234	2.30301
16	1.20596	1.50045
32	0.672577	1.44844
64	0.514111	2.19934

Exercise 5, Jacobi

Non-parallelized version

Make run, get the following result:

Intervalls: 100 x 100
Start Jacobi solver for 10201 d.o.f.s
aver. Jacobi rate : 0.997922 (1000 iter)
final error: 0.124971 (rel)   0.000194029 (abs)
JacobiSolve: timing in sec. : 0.080958
ASCI file square_100.txt opened
17361 2 34320 3

Start Jacobi solver for 17361 d.o.f.s
aver. Jacobi rate : 0.998401 (1000 iter)
final error: 0.201744 (rel)   0.000265133 (abs)
JacobiSolve: timing in sec. : 0.189638

Parallelized version

Intervalls: 100 x 100, -O3, 1000 iter

Number of Threads	Time1, s	Time2, s
1	0.0774207	0.189079
2	0.0867711	0.1952
4	0.0979033	0.223702
8	0.0935012	0.223959
16	0.12996	0.234558
32	0.151673	0.22109
64	0.264242	0.303877

No improvement, perhaps calculation amount is too small.

Intervalls: 1000 x 1000, -O3, 1000 iter

Number of Threads	Time1, s	Time2, s
1	9.65759	0.188669
2	6.32916	0.175074
4	4.51213	0.120928
8	4.42104	0.0979477
16	3.79665	0.0879493
32	3.63878	0.0947926

64	3.76486	0.203179

When threads number are 2 and 4, the efficiencies are best. When the processes number increase, the communication time between caches of processsors increase for sharing data.