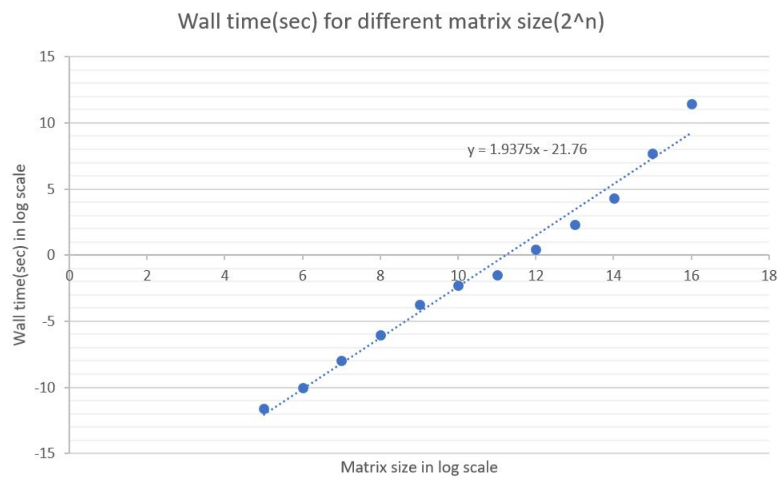
Assignment 1
HPC

By Amirhossein Jafarzadeh Ghazi

February 29, 2020

Matrix size

First of all, I wrote a code for matrix-vector multiplication, and then I found that for the matrix size of 2^{16} with the GNU compiler (gcc), the wall time is about 72 seconds. I did not use any parallelization in this part. You can see the relationship between "Wall Time" and "Matrix Size" on the plot below:



The slop of this graph is about 2 which makes sense because based on order of complexity in double loop, we can assume $WT = 2N^2$ such that WT is wall time and N is matrix size, and then we have:

$$Log(WT) = Log(2) + 2Log(N)$$

GNU versus Intel compiler

At this step I fixed the matrix size to be 2^{16}, and then besides the gcc compiler, I used Intel compiler(icc) for compiling the program. After that I used the no compiler optimization(-O0) and aggressive optimization(-O3). In the table below, you can see wall time for each combination of compilers and optimization modes.

I compiled and ran the program for 10 times and took average, So the results are the average wall time.

Results Using GNU and Intel Compiler	
Compiler	**Double-Loop** Time(s)
gcc	71.659341
icc	63.823759

Table 1: Results Using GNU C compiler (gcc) and Intel Compiler(icc)

Results Using different compilers and optimization criteria	
Compiler/Optimization	**Double-Loop** Time(s)
gcc/O0	73.957342
gcc/O3	59.075633
icc/O0	62.662395
icc/O3	65.119287

Table 2: Results Using different compilers (gcc/icc) and different Optimization criteria (O0/O3)

OMP

Finally, I used parallelization with different number of threads(1, 2, and 4) to compile and run the matrix-vector multiplication code. In addition, for calculating the efficiency and speed-up, I used equations below:

$$Speed-up = S/P \tag{1}$$

$$Efficiency = S/n*P \tag{2}$$

Such that:
S: The wall time for the code compile in series.
P: The wall time for the code compiled in parallel.
n: The number of threads.

Results Using gcc compiler for different optimization and number of threads			
Optimization/Thread	**OMP** Time(s)	**Speed-up**	**Efficiency**
O0/1	72.456783	1.000000	1.000000
O3/1	60.056777	1.000000	1.000000
O0/2	66.359812	1.205306	0.602653
O3/2	62.889564	1.138488	0.569244
O0/4	63.992568	1.232775	0.308194
O3/4	57.002149	1.181462	0.295365

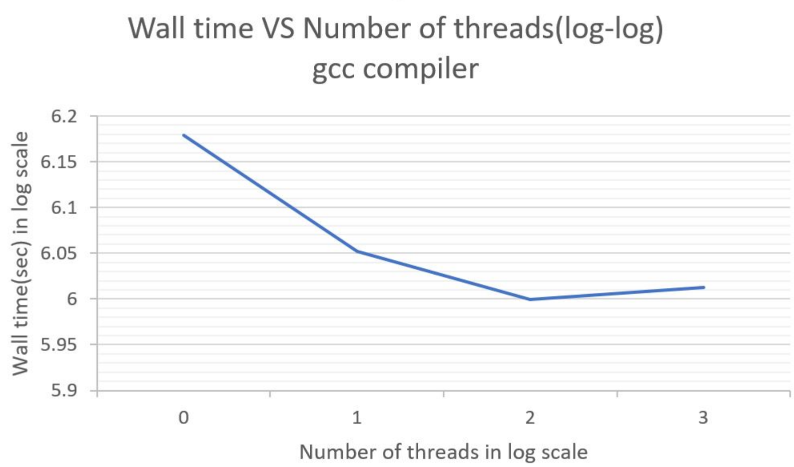
Table 3: Results Using different optimization(O0/O3) and different number of threads(1,2,4) with gcc compiler

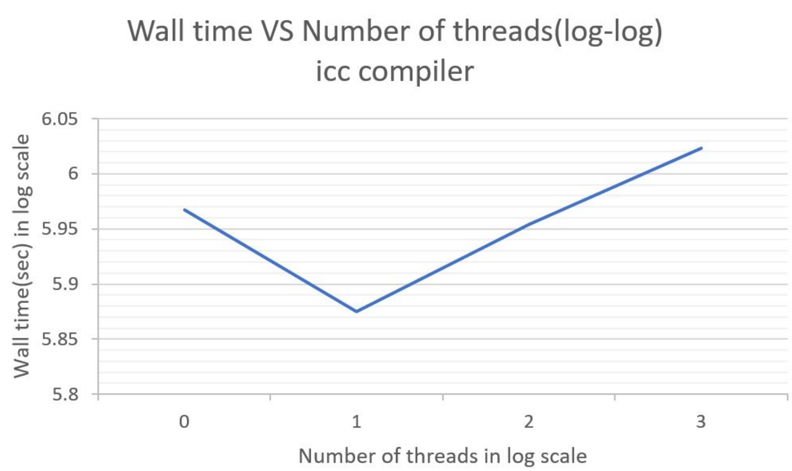
Results Using icc compiler for different optimization and number of threads			
Optimization/Thread	**OMP** Time(s)	**Speed-up**	**Efficiency**
O0/1	62.559635	1.000000	1.000000
O3/1	66.002455	1.000000	1.000000
O0/2	58.669785	1.068052	0.534026
O3/2	57.889564	1.124888	0.562444
O0/4	62.011125	1.010502	0.252626
O3/4	63.987563	1.017687	0.254422

Table 4: ResultsUsing different optimization(O0/O3) and different number of threads(1,2,4) with icc compiler

All in all, it seems we have better efficiency when we use only 2 threads than when we use 4 threads for both gcc and icc compiler. On the other hand, regarding speed-up, in gcc compiler speed-up for 4 threads is better than that of 2 threads. But in the icc compiler, it is different. Speed-up for 2 threads is better than that of 4 threads.

In addition in the graphs below you can compare wall time and number of threads in both compiler:


Wall time VS Number of threads(log-log)
gcc compiler
6.2
6.15
6.1
6.05
6
5.95
5.9
Wall time(sec) in log scale
0
1
2
3
Number of threads in log scale


Wall time VS Number of threads(log-log)
icc compiler
6.05
6
5.95
5.9
5.85
5.8
Wall time(sec) in log scale
0
1
2
3
Number of threads in log scale