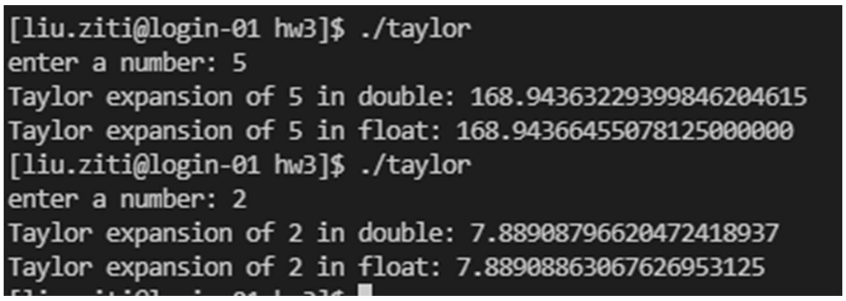
Ziting Liu 002981429
EECE5640
HW3

Q1:
a)



```
[liu.ziti@login-01 hw3]$ ./taylor
enter a number: 5
Taylor expansion of 5 in double: 168.94363229399846204615
Taylor expansion of 5 in float: 168.94366455078125000000
[liu.ziti@login-01 hw3]$ ./taylor
enter a number: 2
Taylor expansion of 2 in double: 7.88908796620472418937
Taylor expansion of 2 in float: 7.88908863067626953125
```

In the first part of the question 1, My code is computing 15 terms of the Taylor series. It will take a user input and output the result in both double and float. Double and Float computation was done separately in the code so that I can observe the difference side by side.

From the screenshot, we can see that double hold more decimal points than floats, and float gives slightly larger result than double. The reason being float (single precision) stores value in 32 bits, whereas double (double precision) stores value in 64 bits. Double can hold more significant values. The reason why float gives larger results than double is that: float is doing more rounding up in the process.

b)

I didn't get the expected result for this part of the question. The output I got with and without the AVX extension was no different. Although I tried many things, including the steps posted on Piazza, changing nodes, try different compiler settings, and switching to COE system. But all of which are giving me the same result.

c)

single precision:

1.1=0 10000000 00011001100110011010000
6200=0 10001011 10000011100000000000000
-0.041=1 01111010 01101000011101000000000

Double precision

1.1=0 01111111111 0001100110011001100110011001100110011001100110011010
6200=0 10000001011 1000001110000000000000000000000000000000000000000000
-0.041=1 01111111010 0100111111011111001110110110010001011010000111001011

Q2:

(The result screenshot is just too big to be included in here)

In this problem, my idea was to divide the number of threads by 2. For instance, we are using 8 threads, I will use 4 threads to remove all the number that are divisible by 3, and the other 4 threads to remove all the number that are divisible by 7.

The running time doubled as I move from 8 to 16 threads, which is a bit of a surprise. The running time of my program is ~0.78sec when running on 8 threads, and ~1.5 sec when running on 16 threads. I believe there are 2 reasons why more threads are slowing down the program. The first one is obvious, creating threads costs time. Modular computation is not a heavy computing task, and the disadvantage of creating 8 more threads will show in this case. The second reason being the synchronization.

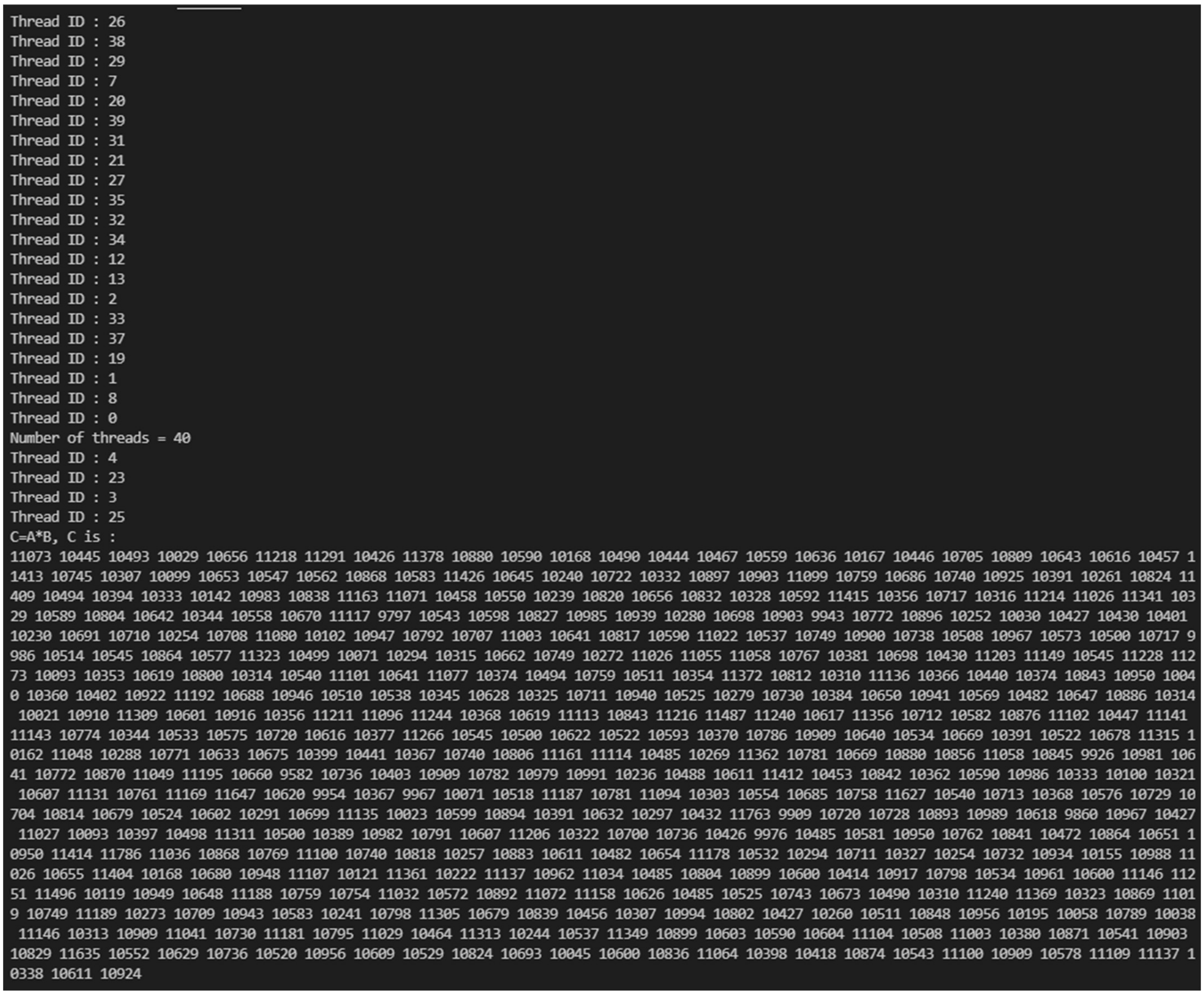
To prove my point, I disabled the synchronization, and changed the range from 100000 to 10000000 numbers, which is 100 times more. Now the 16 threads program is running 3 times faster than 8 threads.

Q3:

Modifying the code using OpenMP was very simple. All I needed to do was modify the loop that spawn threads and deleted the "pthread_join" loop. To observe the change in performance, I changed the eating time from random to 2 second. Another change is all the philosophers will now eat exactly 5 times. I this case, the total number of "eating" is 50. I believe this modification should be good enough to observe the performance changes. I took average time over 5 runs on both Pthread and Open MP. Pthread took 51.217 sec on average, Open MP took 53.012 on average. We can conclude that Open MP philosophers uses a little bit more time than Pthread Philosophers.

Because Open MP is a high-level API to work with threads, the scalability is poor. It might not work with semaphores that well. The time required to synchronize with semaphores might be longer than pthreads.

Q4:



```
Thread ID : 26
Thread ID : 38
Thread ID : 29
Thread ID : 7
Thread ID : 20
Thread ID : 39
Thread ID : 31
Thread ID : 21
Thread ID : 27
Thread ID : 35
Thread ID : 32
Thread ID : 34
Thread ID : 12
Thread ID : 13
Thread ID : 2
Thread ID : 33
Thread ID : 37
Thread ID : 19
Thread ID : 1
Thread ID : 8
Thread ID : 0
Number of threads = 40
Thread ID : 4
Thread ID : 23
Thread ID : 3
Thread ID : 25
C=A*B, C is :
11073 10445 10493 10029 10656 11218 11291 10426 11378 10880 10590 10168 10490 10444 10467 10559 10636 10167 10446 10705 10809 10643 10616 10457 1
1413 10745 10307 10099 10653 10547 10562 10868 10583 11426 10645 10240 10722 10332 10897 10903 11099 10759 10686 10740 10925 10391 10261 10824 11
409 10494 10394 10333 10142 10983 10838 11163 11071 10458 10550 10239 10820 10656 10832 10328 10592 11415 10356 10717 10316 11214 11026 11341 103
29 10589 10804 10642 10344 10558 10670 11117 9797 10543 10598 10827 10985 10939 10280 10698 10903 9943 10772 10896 10252 10030 10427 10430 10401
10230 10691 10710 10254 10708 11080 10102 10947 10792 10707 11003 10641 10817 10590 11022 10537 10749 10900 10738 10508 10967 10573 10500 10717 9
986 10514 10545 10864 10577 11323 10499 10071 10294 10315 10662 10749 10272 11026 11055 11058 10767 10381 10698 10430 11203 11149 10545 11228 112
73 10093 10353 10619 10800 10314 10540 11101 10641 11077 10374 10494 10759 10511 10354 11372 10812 10310 11136 10366 10440 10374 10843 10950 1004
0 10360 10402 10922 11192 10688 10946 10510 10538 10345 10628 10325 10711 10940 10525 10279 10730 10384 10650 10941 10569 10482 10647 10886 10314
 10021 10910 11309 10601 10916 10356 11211 11096 11244 10368 10619 11113 10843 11216 11487 11240 10617 11356 10712 10582 10876 11102 10447 11141
11143 10774 10344 10533 10575 10720 10616 10377 11266 10545 10500 10622 10522 10593 10370 10786 10909 10640 10534 10669 10391 10522 10678 11315 1
0162 11048 10288 10771 10633 10675 10399 10441 10367 10740 10806 11161 11114 10485 10269 11362 10781 10669 10880 10856 11058 10845 9926 10981 106
41 10772 10870 11049 11195 10660 9582 10736 10403 10909 10782 10979 10991 10236 10488 10611 11412 10453 10842 10362 10590 10986 10333 10100 10321
 10607 11131 10761 11169 11647 10620 9954 10367 9967 10071 10518 11187 10781 11094 10303 10554 10685 10758 11627 10540 10713 10368 10576 10729 10
704 10814 10679 10524 10602 10291 10699 11135 10023 10599 10894 10391 10632 10297 10432 11763 9909 10720 10728 10893 10989 10618 9860 10967 10427
 11027 10093 10397 10498 11311 10500 10389 10982 10791 10607 11206 10322 10700 10736 10426 9976 10485 10581 10950 10762 10841 10472 10864 10651 1
0950 11414 11786 11036 10868 10769 11100 10740 10818 10257 10883 10611 10482 10654 11178 10532 10294 10711 10327 10254 10732 10934 10155 10988 11
026 10655 11404 10168 10680 10948 11107 10121 11361 10222 11137 10962 11034 10485 10804 10899 10600 10414 10917 10798 10534 10961 10600 11146 112
51 11496 10119 10949 10648 11188 10759 10754 11032 10572 10892 11072 11158 10626 10485 10525 10743 10673 10490 10310 11240 11369 10323 10869 1101
9 10749 11189 10273 10709 10943 10583 10241 10798 11305 10679 10839 10456 10307 10994 10802 10427 10260 10511 10848 10956 10195 10058 10789 10038
 11146 10313 10909 11041 10730 11181 10795 11029 10464 11313 10244 10537 11349 10899 10603 10590 10604 11104 10508 11003 10380 10871 10541 10903
10829 11635 10552 10629 10736 10520 10956 10609 10529 10824 10693 10045 10600 10836 11064 10398 10418 10874 10543 11100 10909 10578 11109 11137 1
0338 10611 10924
```

In this question, I used the parallel for loop reduction function as requested. Computing matrix*vector require 2 for loops. I used a regular OMP for loop on the outside loop and used a OMP for reduction loop for the inside loop. Because we need to add up $A_{11}*B_1+A_{12}*B_2+A_{13}*B_3+\dots$ Throughout the loop.

Q5:

The sparse matrix format I will be discussing is “Aligned_COO” format. This is an extension to COO format to optimize performance of large sparse matrix having skewed distribution of non-zero elements. (Shah, 2012). The Aligned_COO has two key advantages. It achieves higher performance for matrices with skewed distribution of zeros; It also provide acceptable performance for wide range of sparse matrix patterns.

Like CSR and ELLPACK, it uses segments containing elements for concurrent computation. Segments are executed in parallel fashion to achieve synchronization free execution. Unlike CSR and ELLPACK which are suitable for sparse matrix which have rows with similar row length and DIA format is suitable for diagonal matrix pattern inly. Aligned_COO is suitable for sparse matrix which having large power-law distribution.

Author tested the speedup of using ALIGNED_COO over CSR. The performance gain shows that ALIGNED_COO can be applied to wide range of sparse patterns. It can also gain higher performance over ELLPACK for large and highly skewed matrices. The proposed format gains upto 90x speedup over ELLPACK format for highly skewed data distribution.

M. Shah and V. Patel, "An Efficient Sparse Matrix Multiplication for Skewed Matrix on GPU," 2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems, 2012, pp. 1301-1306, doi: 10.1109/HPCC.2012.192.