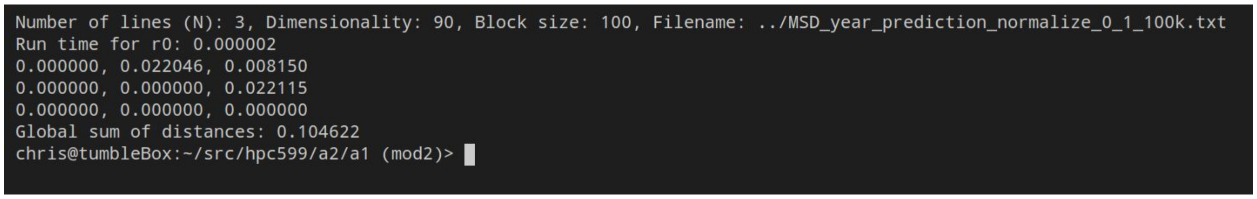
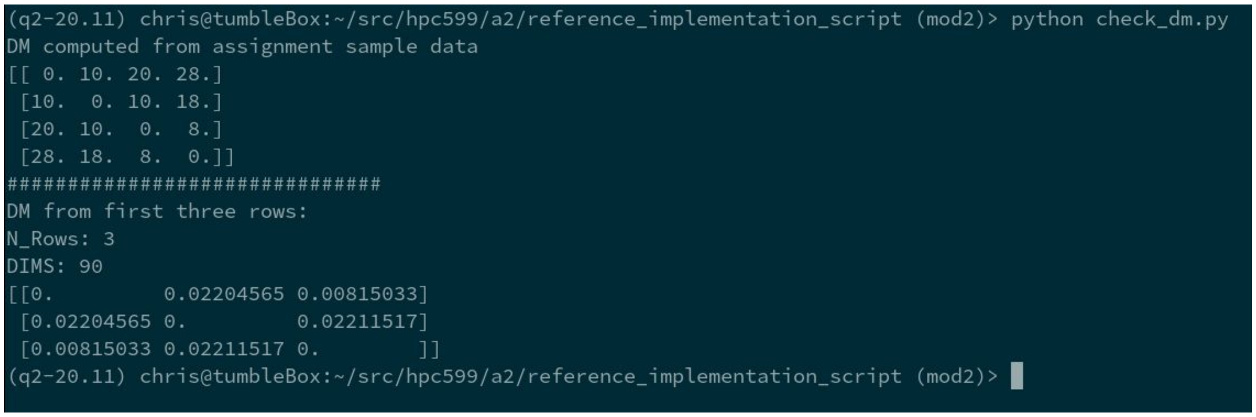
Chris Keefe
Assignment 2

Activity 1: Code description
My portion of the code begins by initializing a bunch of variables, including the number of rows of the data each rank should handle. I populate an array of start indices that get MPI_Scattered and used by each rank to decide what row of the data they should start on. Using scatter here felt super forced (each rank can more efficiently calculate its own start row), and made the assignment more confusing, but I guess practice is practice. I then calloc a 1d array on each rank to hold that rank's chunk of the distance matrix, and start the timer on rank 0. Each rank calculates the upper triangle of the distance matrix only (a small optimization - DM's are symmetrical, and this approach captures all unique values in ½ the calculations). I then calculate and display the run time, and print the distance matrix in the correct order by printing each rank's values and then broadcasting the next rank that should print. I calculate the global sum (multiplying by 2 to compensate for the upper-half-only DM), and print it. I clean up the dynamically allocated data, and close up shop.



```
Number of lines (N): 3, Dimensionality: 90, Block size: 100, Filename: ../MSD_year_prediction_normalize_0_1_100k.txt
Run time for r0: 0.000002
0.000000, 0.022046, 0.008150
0.000000, 0.000000, 0.022115
0.000000, 0.000000, 0.000000
Global sum of distances: 0.104622
chris@tumbleBox:~/src/hpc599/a2/a1 (mod2)>
```

Display of upper triangle DM from first three rows of data with timing and global sum.

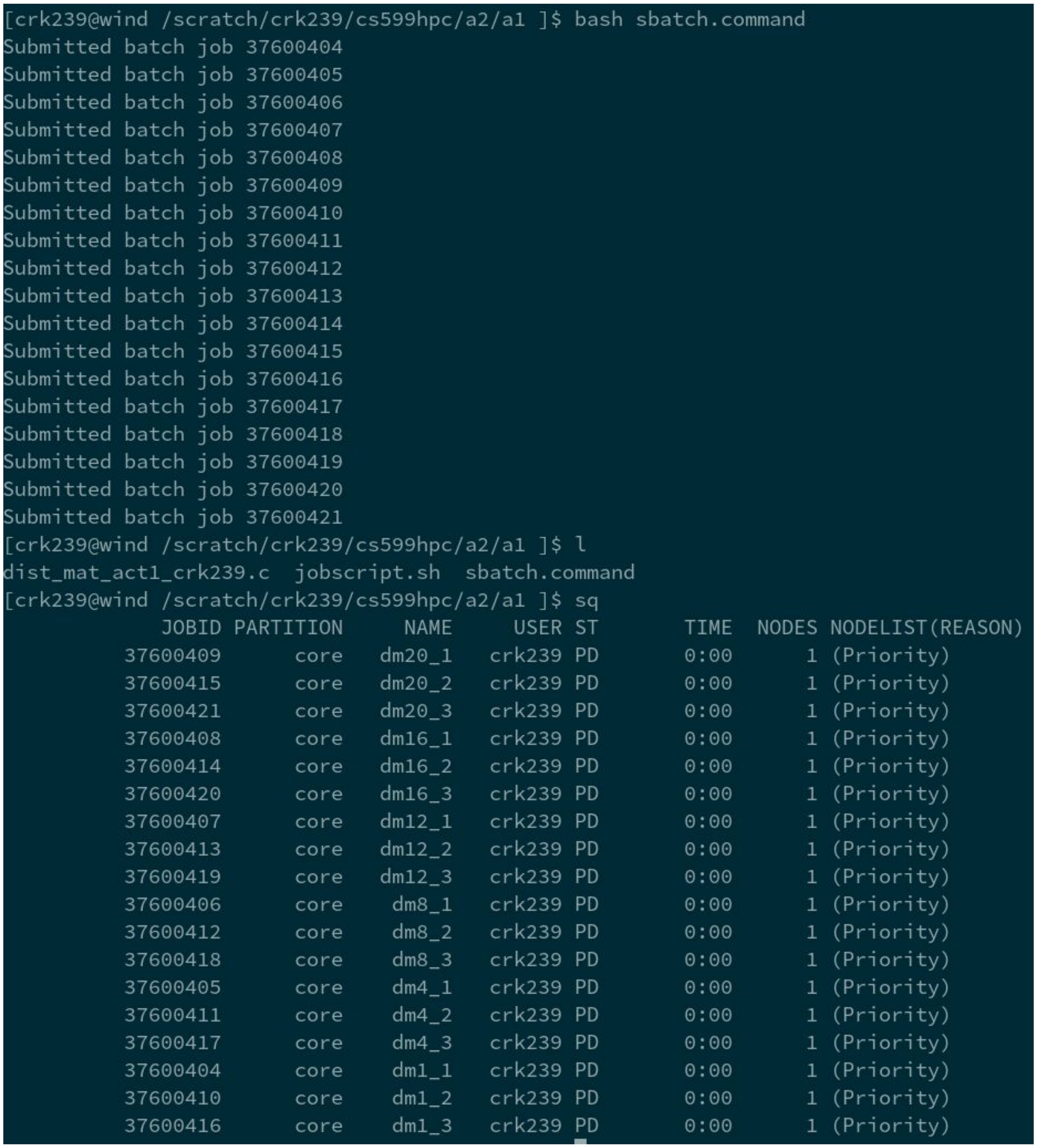


```
(q2-20.11) chris@tumbleBox:~/src/hpc599/a2/reference_implementation_script (mod2)> python check_dm.py
DM computed from assignment sample data
[[ 0. 10. 20. 28.]
 [10.  0. 10. 18.]
 [20. 10.  0.  8.]
 [28. 18.  8.  0.]]
###############################
DM from first three rows:
N_Rows: 3
DIMS: 90
[[0.         0.02204565 0.00815033]
 [0.02204565 0.         0.02211517]
 [0.00815033 0.02211517 0.        ]]
(q2-20.11) chris@tumbleBox:~/src/hpc599/a2/reference_implementation_script (mod2)>
```

Validation of my distance calculations using reference implementation from scipy. Above, see a DM calculated on the sample values in the assignment. Below, a DM calculated on the first three rows of the MSD data set matches the values from my implementation (excepting floating point error). Test script included in the zip file.

Jobscript/run description

In order to cut down on manual effort, I used some Slurm/BASH magic to calculate an appropriate amount of memory for each job, and then sbatch all 18 jobs in one shot. The script passes job-specific output parameters, including --job-name, --mem, --ntasks, and --output. I’ve included the `sbatch.command` script with my code. NOTE: This script was written based on the Slurm documentation, which states that the default memory unit is MB. It looks like monsoon uses MiB, so my script is overbudgeting. This could be easily updated, but time is up.



```
[crk239@wind /scratch/crk239/cs599hpc/a2/a1 ]$ bash sbatch.command
Submitted batch job 37600404
Submitted batch job 37600405
Submitted batch job 37600406
Submitted batch job 37600407
Submitted batch job 37600408
Submitted batch job 37600409
Submitted batch job 37600410
Submitted batch job 37600411
Submitted batch job 37600412
Submitted batch job 37600413
Submitted batch job 37600414
Submitted batch job 37600415
Submitted batch job 37600416
Submitted batch job 37600417
Submitted batch job 37600418
Submitted batch job 37600419
Submitted batch job 37600420
Submitted batch job 37600421
[crk239@wind /scratch/crk239/cs599hpc/a2/a1 ]$ l
dist_mat_act1_crk239.c  jobscript.sh  sbatch.command
[crk239@wind /scratch/crk239/cs599hpc/a2/a1 ]$ sq
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
          37600409      core   dm20_1   crk239 PD       0:00      1 (Priority)
          37600415      core   dm20_2   crk239 PD       0:00      1 (Priority)
          37600421      core   dm20_3   crk239 PD       0:00      1 (Priority)
          37600408      core   dm16_1   crk239 PD       0:00      1 (Priority)
          37600414      core   dm16_2   crk239 PD       0:00      1 (Priority)
          37600420      core   dm16_3   crk239 PD       0:00      1 (Priority)
          37600407      core   dm12_1   crk239 PD       0:00      1 (Priority)
          37600413      core   dm12_2   crk239 PD       0:00      1 (Priority)
          37600419      core   dm12_3   crk239 PD       0:00      1 (Priority)
          37600406      core    dm8_1   crk239 PD       0:00      1 (Priority)
          37600412      core    dm8_2   crk239 PD       0:00      1 (Priority)
          37600418      core    dm8_3   crk239 PD       0:00      1 (Priority)
          37600405      core    dm4_1   crk239 PD       0:00      1 (Priority)
          37600411      core    dm4_2   crk239 PD       0:00      1 (Priority)
          37600417      core    dm4_3   crk239 PD       0:00      1 (Priority)
          37600404      core    dm1_1   crk239 PD       0:00      1 (Priority)
          37600410      core    dm1_2   crk239 PD       0:00      1 (Priority)
          37600416      core    dm1_3   crk239 PD       0:00      1 (Priority)
```

- **Q1: Assume the dataset is stored as double precision floating point values in main memory (each double requires 8 bytes of space). How much memory (in MiB) is required to store the entire dataset in main memory?**

For questions 1-4, I assume we're only storing the dataset in memory once (not, e.g. once per rank).

Assuming data only (no header values):
100, 000 rows x 90 cols = 9,000,000 values
9,000,000 values * 8 bytes/value = 72,000,000 bytes
72,000,000 / 1024 ^ 2 = 68.664550781 MiB to store the data in memory

- **Q2: Assume the distance matrix is stored using double precision floating point values in main memory (each double requires 8 bytes of space). How much memory (in MiB) is required to store the entire distance matrix in main memory?**

Assuming data only (no header values):

100,000 rows x 100,000 columns = 10^10 values
10^10 values x 8 bytes/value = 8 x 10^10 bytes
8 x 10 ^ 10 / 1024 ^2 = 76,293.9453125 MiB

- **Q3: Could you store the dataset in main memory on a typical laptop computer? Explain.**

Yes. If we assume a typical contemporary laptop has between 4 and 16 GiB RAM, 68.664550781 MiB (or .0671 GiB) would fit on any contemporary laptop with plenty of memory left to do some computing.

- **Q4: Could you store the distance matrix in main memory on a typical laptop computer? Explain.**

Nope. Just storing 76,293.9453125 MiB (or 74.51 GiB) would require more RAM than most high-end workstation pcs have available.

- **Q5: When using p=1 and p=20 ranks, what is the total memory required (in MiB) to store the distance matrix, respectively?**

Because the distance matrix remains distributed, the total memory required to store it will be 76,293.9453125 MiB regardless of the number of ranks (give or take the trivial memory overhead of pointers to the distance matrix blocks on each machine, which could be 8 bytes x p ranks x 1 or 2 depending on whether we store the blocks as 1d or 2d arrays).

In case you're looking for the memory required per rank, it's roughly:

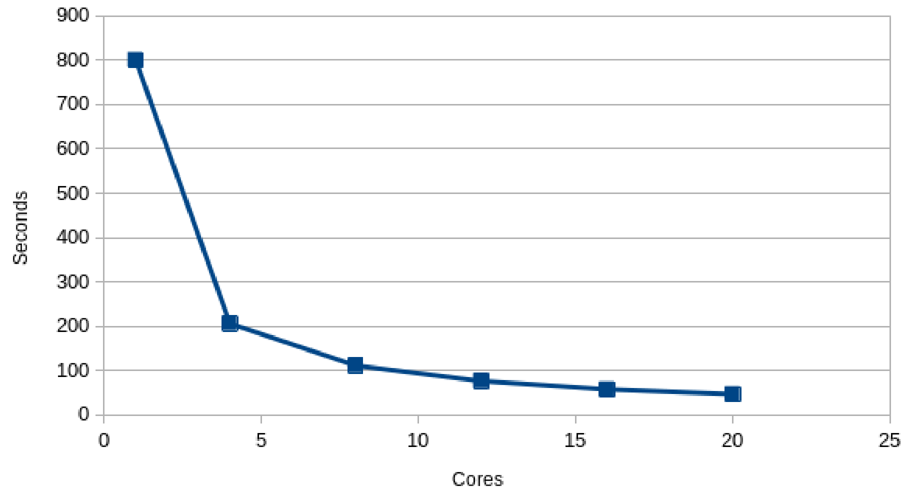
76,293.9453125 MiB per rank for p=1 + pointers
3814.697265625 MiB per rank for p=20 + pointers

# ranks (p)	Time (s)	Speedup	Par Effic	Global Sum	Job script name (.sh)
1	EST: 800	1	1	?	jobscript.sh, sbatch.command
4	205.8668403	3.8860070	0.9715017	455386000.680316	jobscript.sh, sbatch.command
8	110.9087473	7.2131371	0.9016421	455386000.680212	jobscript.sh, sbatch.command
12	76.066647	10.517092	0.8764244	455386000.680004	jobscript.sh, sbatch.command
16	57.54412133	13.902375	0.8688984	455386000.680005	jobscript.sh, sbatch.command
20	46.70446967	17.128981	0.8564490	455386000.680223	jobscript.sh, sbatch.command

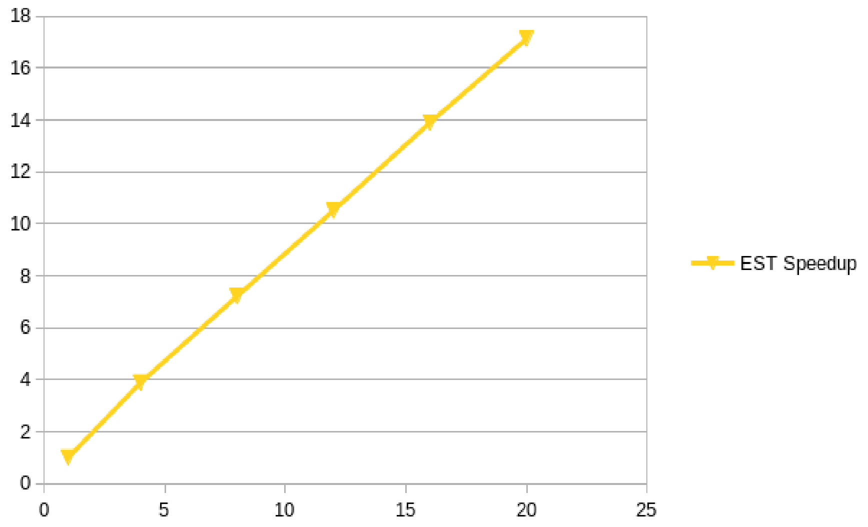
- **Q6: Do you think the performance of the distance matrix calculation is good? Explain.**

My code segfaults consistently when nprocs=1, so calculating speedup and parallel efficiency was not possible. For this table, I've estimated the 1-core run time at 800 seconds (4x as long as four cores, with some wiggle room for overhead). Based on this assumption, the algorithm scales well, with near-linear speedup and a relatively high level of efficiency up to 20 cores.

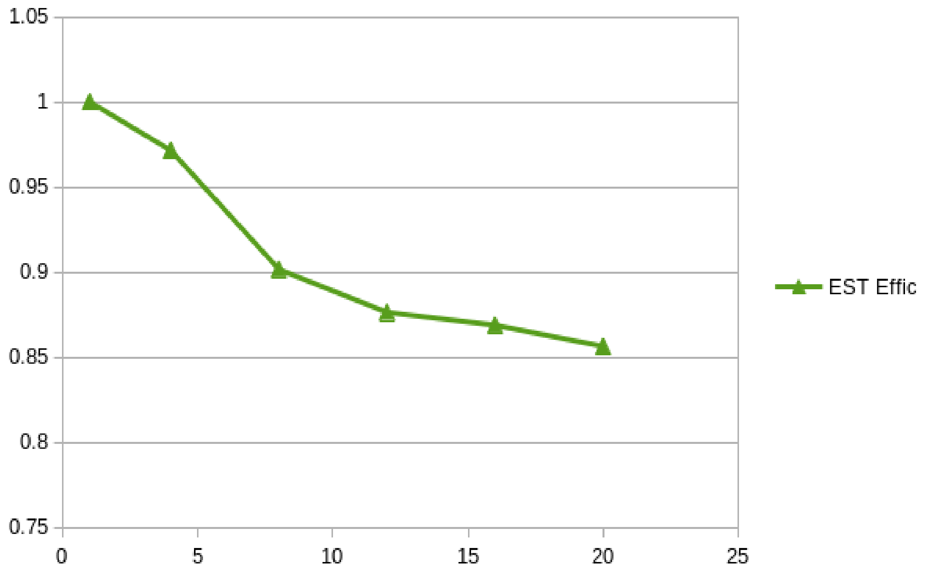
Though it could be much faster, this distance matrix calculation itself isn't the worst possible. I've reduced the number of distances calculated by half, but each calculation still requires floating-point calculations and the use of math.sqrt which is compute-heavy. Row-wise data access isn't terrible, but it's not a particularly cache-aware approach.



Seconds
Cores
0
100
200
300
400
500
600
700
800
900
0
5
10
15
20
25



EST Speedup
0
2
4
6
8
10
12
14
16
18
0
5
10
15
20
25



EST Effic
0.75
0.8
0.85
0.9
0.95
1
1.05
0
5
10
15
20
25

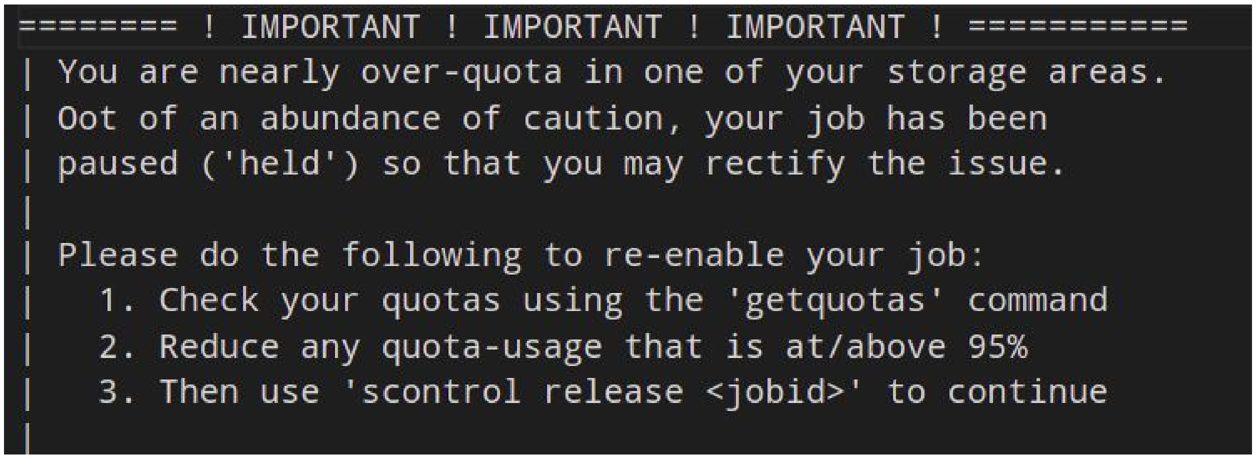
Activity 2: Code description

For readability, I implemented tiling by factoring distance calculations for a two-dimensional tile out into a function. Each rank calculates the number of tiles it will have to compute, and then iterates over them using a while loop. During each iteration, stop indices are ceilinged at the boundaries of the rank's data chunk to prevent overflow, and row-ending tiles are flagged so that indices can be updated correctly at the end of the iteration.

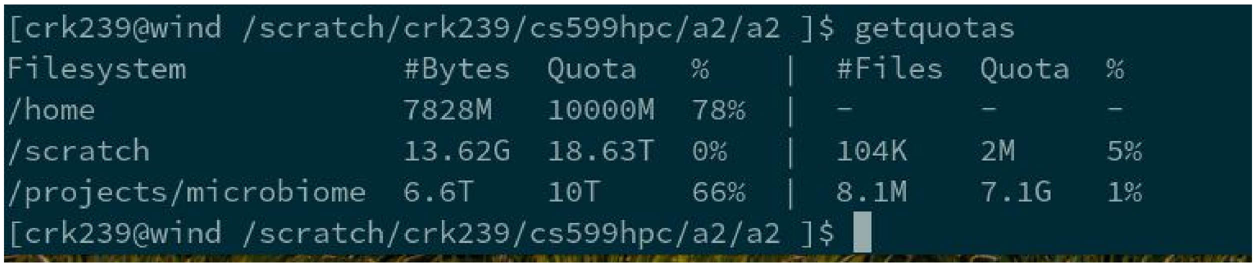
The reduced amount of nesting and intuitive "for tile in tiles" approach gave me a successful first-shot compile and run, which doesn't happen often for me in C, at the possible expense of some efficiency. For C code, this is pretty "Pythonic", and I wonder whether the compiler is inlining the function calls at -O3 and reducing some of the readability-facing assignment and conditional logic, or whether I'm washing out my cache optimizations with bloat. Regardless, working logic would make it easier to refactor for efficiency if that were required.

```
 while (tiles_remaining){
   // Ceiling end indices to prevent overflow, and flag if at end of row
   tile_end_row = (tile_end_row < n_rows_per_rank) ? tile_end_row :
n_rows_per_rank;
   if (tile_end_col >= N){
     tile_end_col = N;
     tile_start_row += blocksize;
     to_next_row = true;
   }
   calculate_tile_dists(rank_start_row, rank_start_col, rank_end_row,
rank_end_col, dataset, local_dm_chunk, N, DIM);
   // adjust loop flag and indices
   tiles_remaining--;
   tile_start_col += blocksize;
   tile_end_col += blocksize;
   if (to_next_row){
     tile_start_row += blocksize;
     tile_end_row += blocksize;
     to_next_row = false;
   }
 }
```

From here on out, I'm guessing. Monsoon is holding all of my jobs for "quota" reasons, despite the fact that my quotas are nowhere near full, and I've been unable to run any of my section 2 or section 3 code on the cluster.



```
======== ! IMPORTANT ! IMPORTANT ! IMPORTANT ! ===========
| You are nearly over-quota in one of your storage areas.
| Oot of an abundance of caution, your job has been
| paused ('held') so that you may rectify the issue.
|
| Please do the following to re-enable your job:
|   1. Check your quotas using the 'getquotas' command
|   2. Reduce any quota-usage that is at/above 95%
|   3. Then use 'scontrol release <jobid>' to continue
|
```



```
[crk239@wind /scratch/crk239/cs599hpc/a2/a2 ]$ getquotas
Filesystem            #Bytes  Quota   %    |  #Files  Quota  %
/home                 7828M   10000M  78%  |  -       -      -
/scratch              13.62G  18.63T  0%   |  104K    2M     5%
/projects/microbiome  6.6T    10T     66%  |  8.1M    7.1G   1%
[crk239@wind /scratch/crk239/cs599hpc/a2/a2 ]$
```

- **Q7: When tiling the computation, comparing all values of b, does b=5 or b=5000 achieve the best performance? Why do you think that is?**

b=5 will probably achieve the best performance. Both block sizes require the same number of calculations (which take up the majority of the run time), but by restricting the number of points we compare *per tile*, we improve the locality, and allow b=5 to produce fewer cache misses.

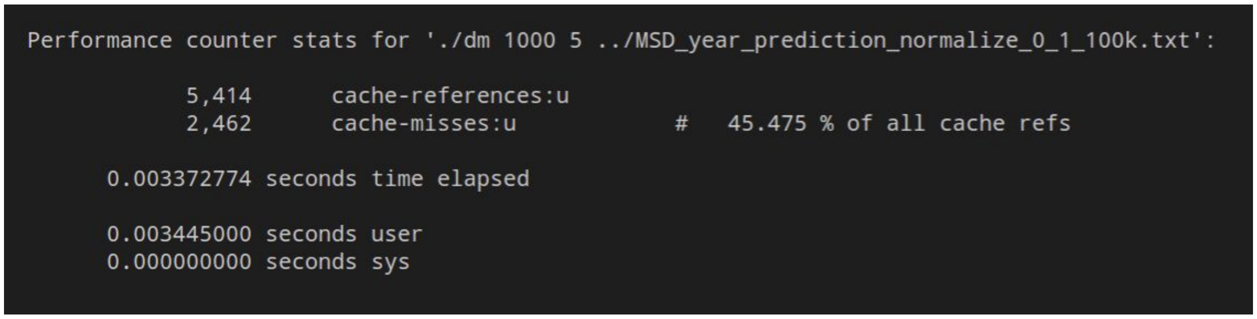
- **Q8: Does tiling the computation improve performance over the original row-wise computation? For p=20 process ranks, report the speedup of the tiled solution using the best value of b over the row-wise solution.**

I'm guessing that tiling the computation will improve performance. This would line up with the worked examples we saw in class, and the significant gains to be had from improving locality would likely more than make up for the micro-optimizations that would make my code more "C-style".

Activity 3: Jobscript Code description
As in the previous activities, I ran a jobscript.sh using a secondary script which programmatically inserted the number of cores while running perf.

- **Q9: Examining the measured percentage of cache misses in the table, does the tiled solution improve cache reuse?**

Again, almost totally guessing here, because I can’t run these jobs on Monsoon. After running some small-data tests on my local machine, I’ve found that the number of cache misses varies by up to 25% for the same binary from run to run. Tiled and untiled versions show no significant difference over ~20 runs. This contradicts my hypotheses above, but I suspect testing on a dedicated machine with larger-scale data might yield different results.



```
Performance counter stats for './dm 1000 5 ../MSD_year_prediction_normalize_0_1_100k.txt':

            5,414      cache-references:u
            2,462      cache-misses:u            #   45.475 % of all cache refs

      0.003372774 seconds time elapsed

      0.003445000 seconds user
      0.000000000 seconds sys
```