Felicity Escarzaga
CS 599: HPC
Assignment 2

Question 1: *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?*

8 bytes * 90 dimensions * 100,000 points = 72,000,000 bytes
72,000,000 bytes / (1024 $^{bytes}/_{KiB}$) / (1024 $^{bytes}/_{MiB}$) = **68.7 MiB**

Question 2: *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?*

8 bytes * $100{,}000^2$ points = 80,000,000,000 bytes
$8.0 * 10^{10}$ bytes / (1024 bytes/KiB) / (1024 bytes/MiB) = **76294 MiB**
= 74.5 GiB

Question 3: *Could you store the dataset in main memory on a typical laptop computer? Explain.*

Yes, a typical laptop has about 8G of memory and could easily store a 68MiB dataset.

Question 4: *Could you store the distance matrix in main memory on a typical laptop computer? Explain.*

No, as stated above a typical laptop has about 8G of memory; therefore it would not be able to store the 74.5G distance matrix.

Question 5: When using p = 1 and p = 20 ranks, what is the total memory required (in MiB) to store the distance matrix, respectively?

Regardless of the number of ranks the distance matrix will require a total of **76294 MiB**. However, when running 20 ranks each rank will use about: 76294 Mib / 20 Ranks = 3815 MiB/rank.

Felicity Escarzaga
CS 599: HPC
Assignment 2

Activity 1:
The points are evenly distributed between the ranks using my_start = `(N/nprocs)*my_rank` with the last rank receiving the remaining points if N was not divided evenly.
Each rank creates its own `nrow` by `N` distance matrix and computes the distance for its respective points in a `[row][column]` for loop that starts from it's `my_start`.
Looking at **Table 1.1**, the time decreases and parallel speedup increases as the number of ranks increase; however the parallel efficiency is highest after the first 4 ranks then begins to decrease. This shows that the increase in ranks has diminishing returns. While 20 ranks is significantly faster than 4 ranks, the difference will not be as significant between 20 ranks and 100 ranks.
That being said, 0.8 efficiency is still relatively high which means the code could benefit from additional ranks, but that benefit would be less each time. We can speculate that this means the code would be better optimized by changing the way it accesses memory, as shown in activity 2.

Table 1.1: Ranks

# of Ranks (p)	Time (s)	Parallel Speedup	Parallel Efficiency	Global Sum	Job Script Name (*.sh)
1	1084.392535	1	1	455386000.679019	distance_act1_fhe2
4	270.5274973	4.008437388	1.002109347	455386000.680447	distance_act1_fhe2
8	139.8924163	7.751617729	0.9689522162	455386000.680108	distance_act1_fhe2
12	96.63835333	11.22114044	0.9350950367	455386000.679990	distance_act1_fhe2
16	75.39875367	14.38210159	0.8988813491	455386000.680024	distance_act1_fhe2
20	66.82855833	16.22648403	0.8113242016	455386000.680184	distance_act1_fhe2

Question 6: *Do you think the performance of the distance matrix calculation is good? Explain.*

As stated above when reviewing **Table 1.1**, performance of the code isn't bad but it's not great. Increasing the dataset size increases the time quadratically, meaning that even with 20 ranks it could take over an hour to compute a distance matrix for 500,000 points with this code. While the program parallelizes relatively well, it could be better.

Felicity Escarzaga
CS 599: HPC
Assignment 2

Activity 2:
Similar to activity 1, each rank determines its respective starting point (my_start) and creates a distance matrix that is nrow by N.
Then it calculates the number of tiles (ntiles) given the blocksize and uses this number to determine the beginning and end of a tile row (trow) and tile column (tcol) since this may not be the same for end columns or end rows. The for loop to calculate the distance matrix is executed as shown in *figure 2.1*, traveling for each tile, for each row, then for each column.

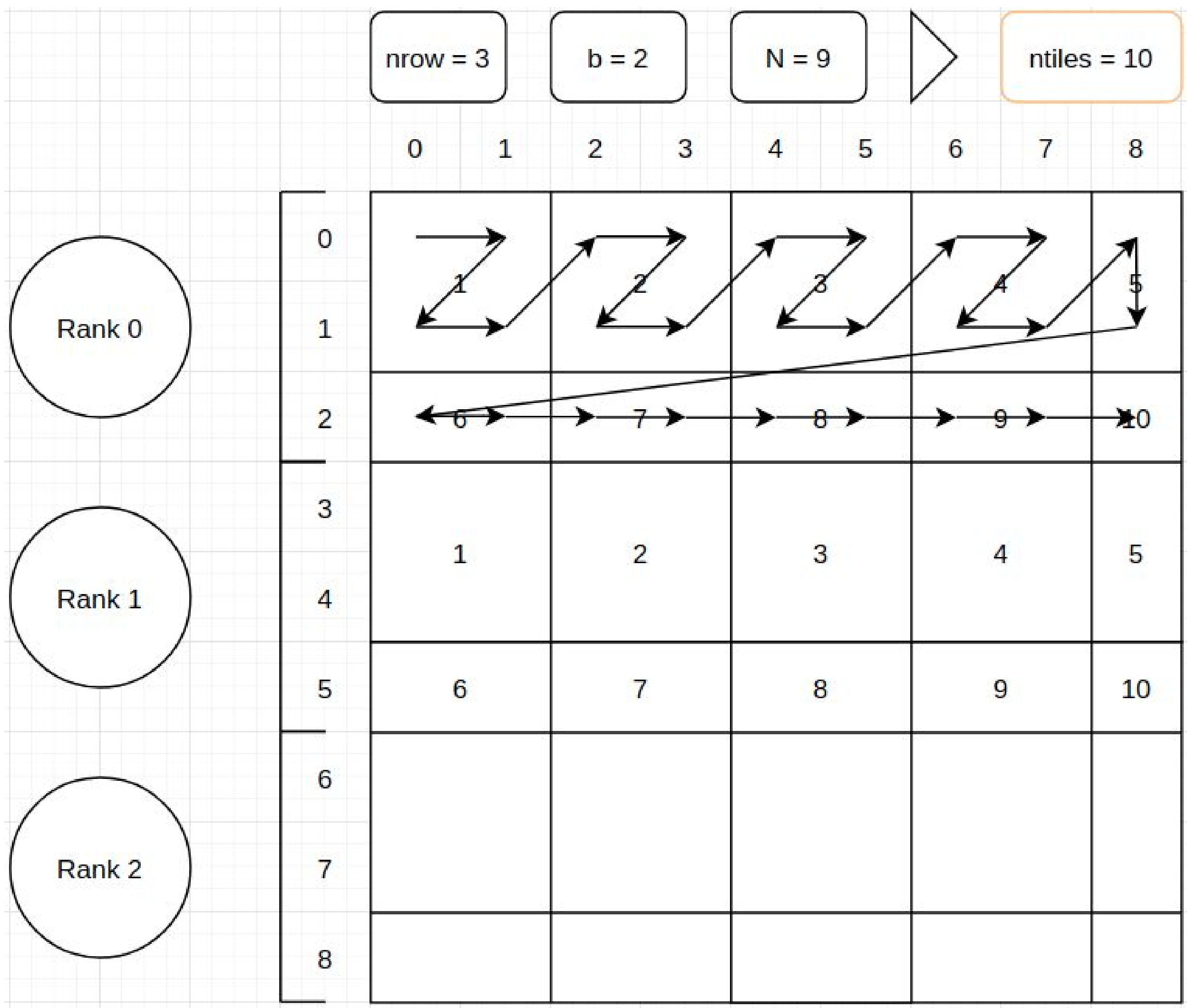

Fig. 2.1: Activity 2 visualization

In **Table 2.1,** it shows that the best block size for 100,000 points is around 100 points. This is likely because bigger tiles means that by the time the loop comes back around the points will no longer be in memory and would be similar to running without tiles at all, whereas too small tiles would be similar to running the loop columns first then rows.

Felicity Escarzaga
CS 599: HPC
Assignment 2

Table 2.1: Block Size

b	Time (s)	Global Sum	Job Script Name (*.sh)
5	43.885868	455386000.680186	distance_act2_fhe2
100	40.29239833	455386000.680205	distance_act2_fhe2
500	41.10490933	455386000.680161	distance_act2_fhe2
1000	41.091574	455386000.680125	distance_act2_fhe2
2000	41.012637	455386000.680143	distance_act2_fhe2
3000	42.25615733	455386000.680156	distance_act2_fhe2
4000	47.12401667	455386000.680101	distance_act2_fhe2
5000	52.428897	455386000.680030	distance_act2_fhe2

Based on **Table 2.2**, tiling improved timing significantly in higher ranks with increased parallel efficiency. Though the efficiency does decrease after 8 ranks, even up to 20 ranks has a parallel efficiency greater than 1. It appears calculating points that may still be in memory greatly improves performance.

Table 2.2: Ranks

# of Ranks (p)	Time (s)	Parallel Speedup	Parallel Efficiency	Global Sum	Job Script Name (*.sh)
1	813.0508585	1	1	455386000.679019	distance_act2_ranks_fhe2
4	201.186013	4.041289185	1.010322296	455386000.680451	distance_act2_ranks_fhe2
8	100.5736665	8.084132624	1.010516578	455386000.680088	distance_act2_ranks_fhe2
12	67.10633067	12.11585927	1.009654939	455386000.679969	distance_act2_ranks_fhe2
16	50.34124367	16.15079007	1.00942438	455386000.680013	distance_act2_ranks_fhe2
20	40.28863867	20.18064858	1.009032429	455386000.680205	distance_act2_ranks_fhe2

Question 7: *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?*

A block size of 5 performs better than a block size of 5000 because a block size of 5000 is unlikely to use cached points by the end making it about as effective as not using tiles at all.

Question 8: *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.*

The speed up of row-wise vs tiled is 66.8 s/ 40.2 s = **1.66** times faster for the tiled version. This means that even using the same resources the tiled version saves time.

Felicity Escarzaga
CS 599: HPC
Assignment 2

Table 2.3: Cache Misses

# of Ranks (p)	% Cache Misses (Row-wise Distance Matrix)	% Cache Misses (Tiled Distance Matrix)	Job Script Name (*.sh)
1	13.275	15.670	distance_act3_fhe2
4	14.643	16.6834	distance_act3_fhe2
8	13.426	16.085	distance_act3_fhe2
12	17.462	17.995	distance_act3_fhe2
16	11.992	17.477	distance_act3_fhe2
20	15.213	19.498	distance_act3_fhe2

Question 9: *Examining the measured percentage of cache misses in the table, does the tiled solution improve cache reuse?*

I really thought that it would, but it appears that the tiled solution did not improve cache reuse. The only rank that came close to being better than the row-wise version was rank 12, and I'm not sure why that is. It's possible that I misunderstand cache misses, or that I did not execute the perf command correctly, or that maybe it was not cache references that improved activity 2 compared to activity 1.