**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 the main memory?**

**Ans:**

The dimension of each of the 100,000 data points is 90. So, the total memory required to store one copy of the entire dataset in the main memory is (90\*100000)\*8 or 72,000,000 bytes or approximately 68.66 MiB (1 MiB = 220 Bytes).

**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 the main memory?**

**Ans:**

Total number of double-precision floating-point numbers = 100,000\*100,000 = 1010

So, the total memory required to store one copy of the distance matrix is 8\*1010 bytes or approximately 74.51 gigabytes.

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

**Ans:**

Given that a typical laptop nowadays has a main memory in the range of 8 to 32 gigabytes, I could easily store the dataset on the main memory of a typical Laptop.

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

**Ans:**

For the same reason mentioned above, it will not be possible to store a copy of the distance matrix on the main memory of a typical laptop.

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

**Ans:**

For both numbers of ranks, the total memory required to store a copy of the entire distance matrix will be the same - 8\*1010 bytes or approximately 74.51 gigabytes. For p > 1 and (100,000 mod p == 0), the distance matrix will be divided equally among the process ranks.

| **Number of Process Ranks** | **Time (seconds)** | **Parallel Speedup** | **Parallel Efficiency** | **Global Sum** | **Job script Name** |
| --- | --- | --- | --- | --- | --- |
| 1 | 965.211207 | **NA** | **NA** | 455386000.679019 | CS599\_HW2\_mh2752\_act1\_nprocs\_1.sh |
| 4 | 239.567063 | 4.02898 | 1.007 | 455386000.680447 | CS599\_HW2\_mh2752\_act1\_nprocs\_4.sh |
| 8 | 120.817908 | 7.98897 | 0.999 | 455386000.680108 | CS599\_HW2\_mh2752\_act1\_nprocs\_8.sh |
| 12 | 81.250253 | 11.879 | 0.9899 | 455386000.679990 | CS599\_HW2\_mh2752\_act1\_nprocs\_12.sh |
| 16 | 61.471607 | 15.7017 | 0.981 | 455386000.680024 | CS599\_HW2\_mh2752\_act1\_nprocs\_16.sh |
| 20 | 49.638656 | 19.445 | 0.972 | 455386000.680184 | CS599\_HW2\_mh2752\_act1\_nprocs\_20.sh |

**Table-1: Row-wise (untiled) implementation**

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

**Ans:**

As the parallel efficiency of the distance calculation across an increasing number of process ranks (hence increasing number of cores) stays very close to 1 as seen in the previous table, the performance of the distance matrix calculation is good. In fact, as the distance calculation by each process rank does not depend on the other process ranks’ calculation operations, we could almost fully parallelize the distance matrix calculation if we had **p = N** (100,000 in this case) number of cores available. In that case, each process rank will calculate exactly one row of the distance matrix which means linear time complexity instead of the quadratic time complexity of the sequential version.

| **Tile Size (b)** | **Time (seconds)** | **Global Sum** | **Job Script Name** |
| --- | --- | --- | --- |
| 5 | 43.089020 | 455386000.680184 | CS599\_HW2\_mh2752\_act2\_tileSize\_5.sh |
| 100 | 41.578737 | 455386000.680184 | CS599\_HW2\_mh2752\_act2\_tileSize\_100.sh |
| 500 | 41.540015 | 455386000.680184 | CS599\_HW2\_mh2752\_act2\_tileSize\_500.sh |
| 1000 | ***41.419449*** | 455386000.680184 | CS599\_HW2\_mh2752\_act2\_tileSize\_1000.sh |
| 2000 | 41.676871 | 455386000.680184 | CS599\_HW2\_mh2752\_act2\_tileSize\_2000.sh |
| 3000 | 43.928598 | 455386000.680184 | CS599\_HW2\_mh2752\_act2\_tileSize\_3000.sh |
| 4000 | 47.828057 | 455386000.680184 | CS599\_HW2\_mh2752\_act2\_tileSize\_4000.sh |
| 5000 | 49.536683 | 455386000.680184 | CS599\_HW2\_mh2752\_act2\_tileSize\_5000.sh |

**Table-2: Tiled implementation (varying tile sizes)**

| **Number of Process Ranks** | **Time (seconds)** | **Parallel Speedup** | **Parallel Efficiency** | **Global Sum** | **Job script Name** |
| --- | --- | --- | --- | --- | --- |
| 1 | 831.239897 | **NA** | **NA** | 455386000.679019 | CS599\_HW2\_mh2752\_act2\_nprocs\_1\_tileSize\_1000.sh |
| 4 | 207.508834 | 4.006 | 1.002 | 455386000.680447 | CS599\_HW2\_mh2752\_act2\_nprocs\_4\_tileSize\_1000.sh |
| 8 | 103.993437 | 7.993 | 0.999 | 455386000.680108 | CS599\_HW2\_mh2752\_act2\_nprocs\_8\_tileSize\_1000.sh |
| 12 | 68.977142 | 12.051 | 1.004 | 455386000.679990 | CS599\_HW2\_mh2752\_act2\_nprocs\_12\_tileSize\_1000.sh |
| 16 | 51.697163 | 16.079 | 1.005 | 455386000.680024 | CS599\_HW2\_mh2752\_act2\_nprocs\_16\_tileSize\_1000.sh |
| 20 | 41.722995 | 19.923 | 0.996 | 455386000.680184 | CS599\_HW2\_mh2752\_act2\_nprocs\_20\_tileSize\_1000.sh |

**Table-3: Tiled implementation (b = 1000) with varying number of ranks**

**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?**

**Ans:**

No, in terms of response time, b = 5 or b = 5000 does not achieve the best performance. This happens probably because b = 5 is too small causing too many cache misses and b = 5000 is too large causing respective ranks to essentially compute row-wise.

**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.**

**Ans:**

Yes, tiling (b = 1000) seems to improve the performance of the program (in terms of response times) quite noticeably compared to the row-wise computation.

For p = 20, tiled solution (b = 1000) speedup is 19.923 and row-wise solution speedup is 19.445.

| **Num of Ranks** | **% Cache Misses (Row-wise Distance Matrix)** | **% Cache Misses (Tiled Distance Matrix)** | **Job Script Name (\*.sh)** |
| --- | --- | --- | --- |
| 1 | 90.196% | 64.601% | CS599\_HW2\_mh2752\_act1\_nprocs\_1\_perf.sh  CS599\_HW2\_mh2752\_act2\_nprocs\_1\_tileSize\_1000\_perf.sh |
| 4 | 91.35% | 19.651% | CS599\_HW2\_mh2752\_act1\_nprocs\_4\_perf.sh  CS599\_HW2\_mh2752\_act2\_nprocs\_4\_tileSize\_1000\_perf.sh |
| 8 | 95.53% | 17.78% | CS599\_HW2\_mh2752\_act1\_nprocs\_8\_perf.sh  CS599\_HW2\_mh2752\_act2\_nprocs\_8\_tileSize\_1000\_perf.sh |
| 12 | 97.24% | 17.71% | CS599\_HW2\_mh2752\_act1\_nprocs\_12\_perf.sh  CS599\_HW2\_mh2752\_act2\_nprocs\_12\_tileSize\_1000\_perf.sh |
| 16 | 97.9% | 18.77 | CS599\_HW2\_mh2752\_act1\_nprocs\_16\_perf.sh  CS599\_HW2\_mh2752\_act2\_nprocs\_16\_tileSize\_1000\_perf.sh |
| 20 | 98.37% | 21.6% | CS599\_HW2\_mh2752\_act1\_nprocs\_20\_perf.sh  CS599\_HW2\_mh2752\_act2\_nprocs\_20\_tileSize\_1000\_perf.sh |

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

**Ans:**

Yes, the tiled solution improve the cache re-use by approximately 26% to 80%.