Parallel Programming SS21 Final Project

Project 03 k-d Tree

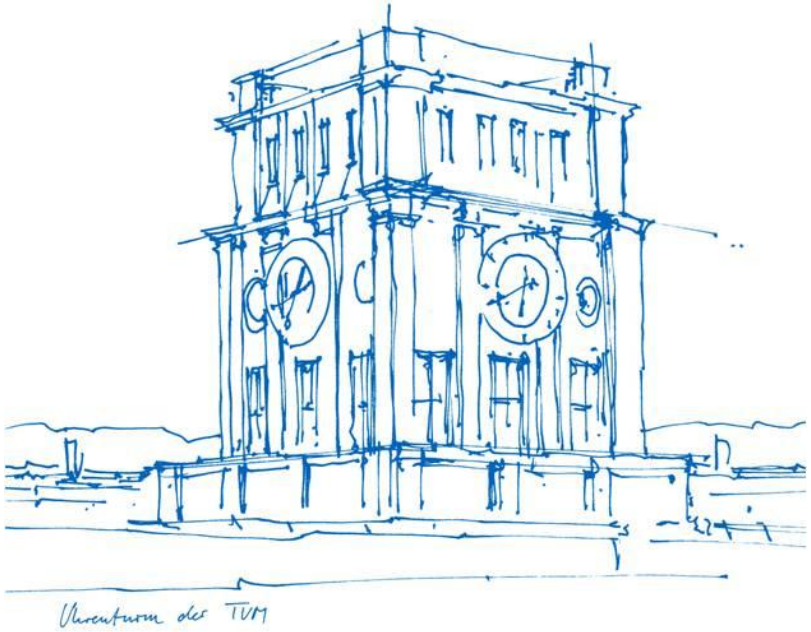
Group 302

09.07.2021

Daniel Baur

Konstantinos Papaioannou

Srivatsa Ravindra



Sequential code analysis - Profiling

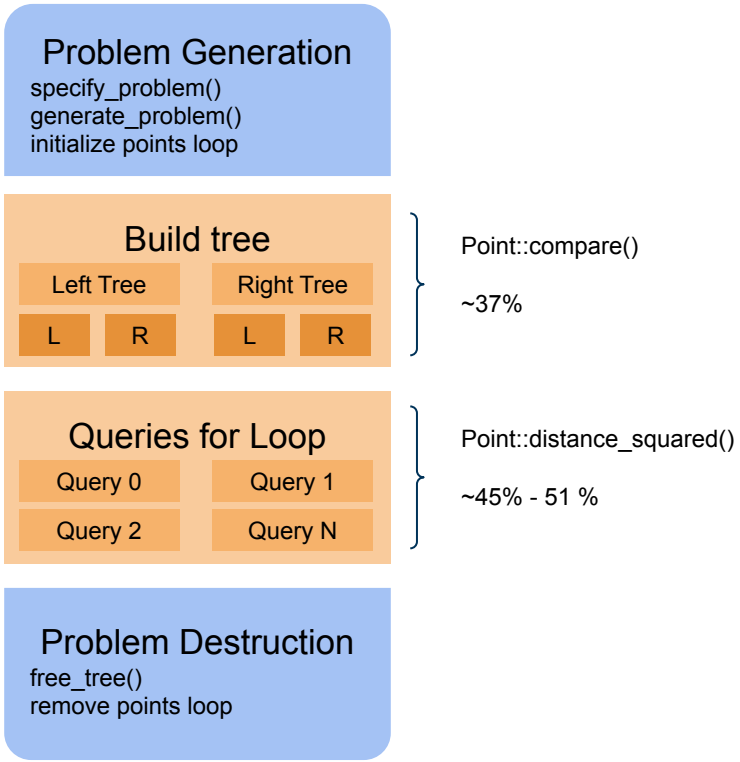
Results measured via perf record ./sequential using the default input value 0

Overhead	Function	Can this be parallelized?
45.88%	Point::distance_squared	yes, can be vectorized.
33.14%	Point::compare	No
7.21%	Utility::generate_problem	yes
4.26%	std::sort	No

Other opportunities for parallelizing:

1. As there are 10 queries and they are independent of each other, each query can be run in parallel.
2. The left and right subtree in the '*build_tree_rec*' can be constructed in parallel.

Sequential code analysis - Amdahl‘s law



Prediction: ~82% - 88% can be parallelized

Maximal Speedup (32 processors):

f = 0.82: SU(32) = 4.86

f = 0.88: SU(32) = 6.78

OpenMP - Parallelized implementation and approach

Initial Ideas:

- left & right subtrees can be generated independently of each other
- queries can run concurrently without interfering with each other

=> parallelize tree generation and NN for-loop

Possible pitfalls:

- Query results need to arrive in requested order
- Possibly large performance overhead for parallelization of tree gen

Implementation concept:

- Use OMP tasks for tree generation, limit number of parallel “splits”
- Use OMP parallel for to split up queries
  - Since it gave us the best results we used default parameters
- Use the Ordered parameter to ensure proper printing order of results

OpenMP - Intermediate Speed-up results, profiling

Intermediate Speed-up: ~3.6
Perf Data:

1. 44.91% Point::distance_squared
2. 33.76% Point::compare
3. 6.19% Point::sort

Idea for Improvement:

- Use SIMD operations for computations in Point::distance squared()
- Parallelize sort operations

Approach:

- use OMP SIMD with reduction to compute distance
- Alternatively: Use intrinsic SIMD to calculate 8 values at a time (see Bonus implementation)

OpenMP - Final Implementation improvements and new speed-up

Calculated Speedup: 3.56 (OMP SIMD)/ 3.93 (Intrinsic SIMD) (Amdahl's law: 6.78)

- Ultimately decided against using intrinsic SIMD operations due to project description
- More time spent in Point::distance_squared with OMP SIMD due to overhead
- No speedup achieved sorting node array with OMP compared to std::sort

Either way, we did not achieve maximum possible speed-up during the time limit

MPI - Parallelized implementation and approach

Initial idea:

- We have 10 queries to solve.
- Solve each query in one mpi process.
- Each MPI process builds its own tree, finds the nearest neighbour and sends the results to the process with rank 0.

Problems with this approach:

- Number of queries to solve remains constant
- If there are more than 10 mpi processes, some of them will have to be idle as we have only 10 queries to solve

MPI - Intermediate Speed-up results, profiling

Achieved speedup: ~1.3 → too low.

@--- MPI Time (seconds) ----------

Task	AppTime	MPITime	MPI%
0	9.31	0.0699	0.75
1	7.12	2.83	39.82
2	7.43	2.85	38.38
3	7.44	2.86	38.43
4	8.97	2.86	31.90
5	7.84	2.83	36.09
6	7.54	2.86	37.94
7	7.3	2.86	39.12
8	7.34	2.86	38.96
9	8.13	2.85	35.07
10	0.0598	0.0597	99.92
11	0.0774	0.0773	99.97
12	0.0746	0.0746	99.96
13	0.0509	0.0509	99.92
14	0.0842	0.0841	99.94
15	0.0372	0.0371	99.90
*	78.8	26.1	33.14

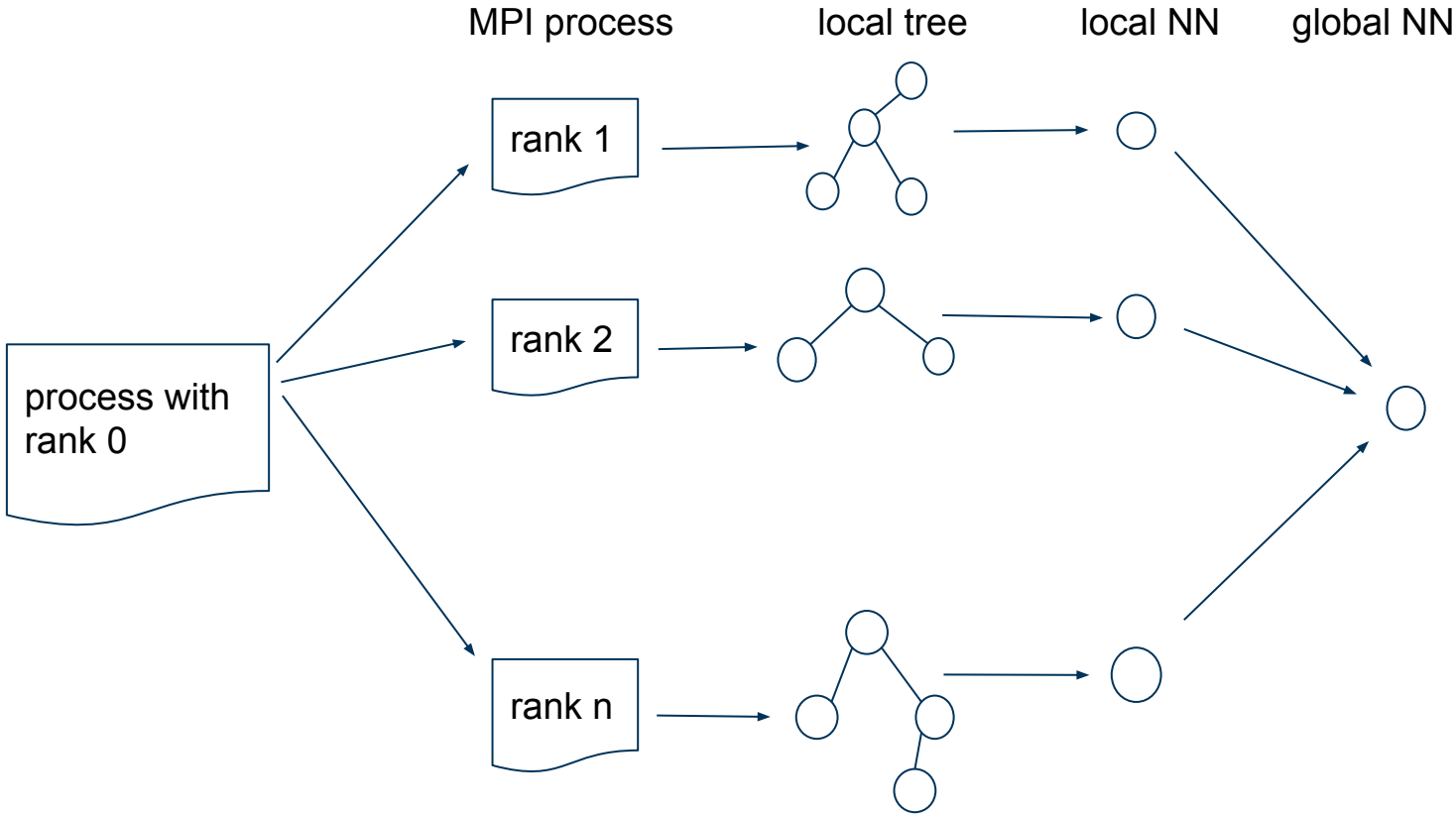
@--- Aggregate Time (top twenty, descending, milliseconds) ----------

Call	Site	Time	App%	MPI%	Count	COV
Bcast	96	2.79e+03	3.53	10.66	1	0.00
Bcast	72	2.79e+03	3.53	10.66	1	0.00
Bcast	22	2.79e+03	3.53	10.66	1	0.00
Bcast	14	2.79e+03	3.53	10.66	1	0.00
Bcast	61	2.79e+03	3.53	10.66	1	0.00
Bcast	30	2.79e+03	3.53	10.66	1	0.00
Bcast	80	2.78e+03	3.53	10.65	1	0.00
Bcast	88	2.78e+03	3.53	10.65	1	0.00
Bcast	38	2.77e+03	3.52	10.61	1	0.00
Comm_create	37	88.8	0.11	0.34	1	0.00
Comm_create	55	83.9	0.11	0.32	1	0.00
Comm_create	87	77.4	0.10	0.30	1	0.00
Comm_create	46	77.2	0.10	0.30	1	0.00
Comm_create	29	74.7	0.09	0.29	1	0.00
Comm_create	66	74.5	0.09	0.29	1	0.00
Comm_create	60	74.4	0.09	0.28	1	0.00
Comm_create	79	73.4	0.09	0.28	1	0.00
Comm_create	5	69.3	0.09	0.27	1	0.00
Comm_create	21	66.9	0.08	0.26	1	0.00
Comm_create	95	66.8	0.08	0.26	1	0.00

Problems:

1. Overhead in splitting the communicator
2. Overhead in Broadcasting the data.

MPI - Final implementation pictorial representation



MPI - Final Implementation improvements and new speed-up

New approach:

- Decompose the data in such a way that each mpi process will work on a subset of points
- Each MPI process receives '*start*' and '*end*' indices which from the process with rank 0.
- '*start*' and '*end*' indices decides which points each mpi process will work on.
- Using these indices, each MPI process constructs a tree.
- Each MPI process will then try to find out a local nearest neighbour for each query.
- MPI_Reduce is then used to find the neighbour with minimum distance.

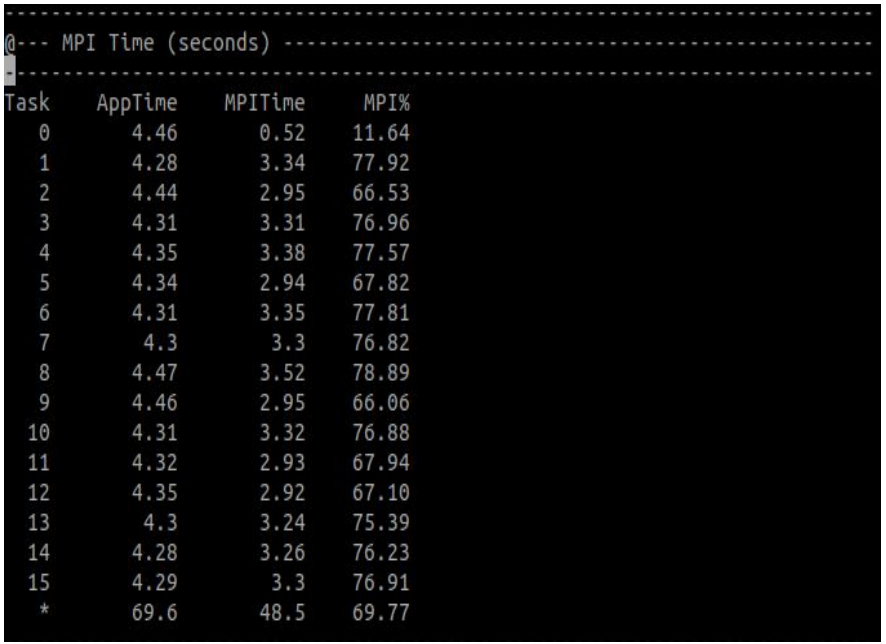
Speed up:

- This approach achieved a speedup of around 7.5

Advantages:

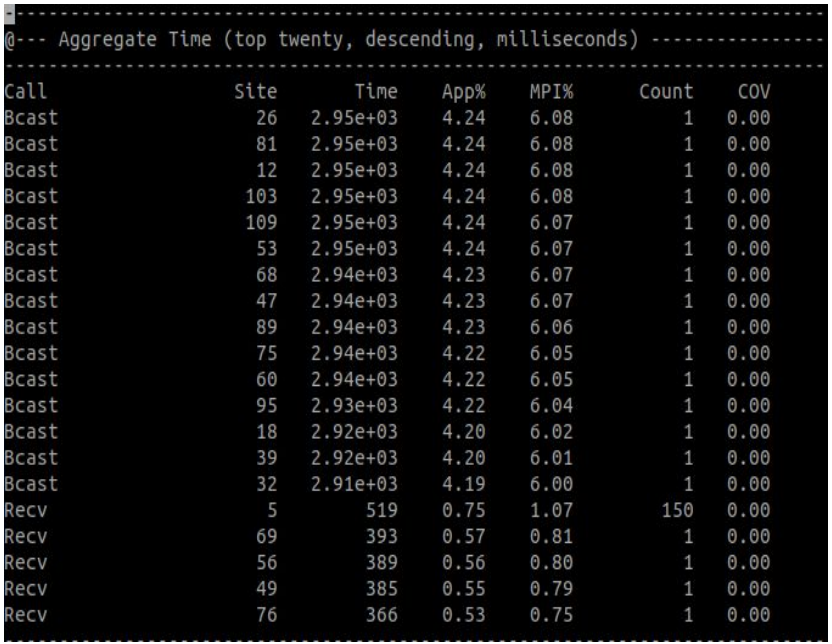
- Works with any number of MPI processes.
- Process with rank 0 decides what each of the MPI process works on.
- Master-Worker pattern.

MPI - Final Implementation improvements,new speed-up, profiling



@--- MPI Time (seconds) ----------

Task	AppTime	MPITime	MPI%
0	4.46	0.52	11.64
1	4.28	3.34	77.92
2	4.44	2.95	66.53
3	4.31	3.31	76.96
4	4.35	3.38	77.57
5	4.34	2.94	67.82
6	4.31	3.35	77.81
7	4.3	3.3	76.82
8	4.47	3.52	78.89
9	4.46	2.95	66.06
10	4.31	3.32	76.88
11	4.32	2.93	67.94
12	4.35	2.92	67.10
13	4.3	3.24	75.39
14	4.28	3.26	76.23
15	4.29	3.3	76.91
*	69.6	48.5	69.77



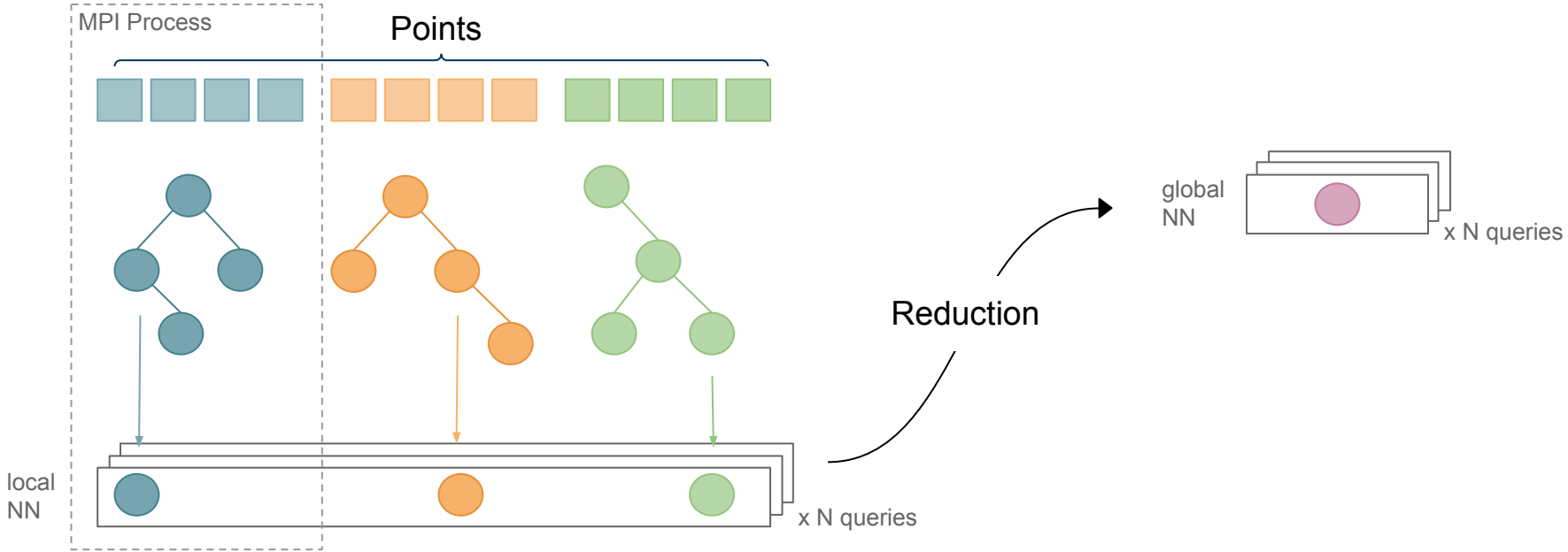
@--- Aggregate Time (top twenty, descending, milliseconds) ----------

Call	Site	Time	App%	MPI%	Count	COV
Bcast	26	2.95e+03	4.24	6.08	1	0.00
Bcast	81	2.95e+03	4.24	6.08	1	0.00
Bcast	12	2.95e+03	4.24	6.08	1	0.00
Bcast	103	2.95e+03	4.24	6.08	1	0.00
Bcast	109	2.95e+03	4.24	6.07	1	0.00
Bcast	53	2.95e+03	4.24	6.07	1	0.00
Bcast	68	2.94e+03	4.23	6.07	1	0.00
Bcast	47	2.94e+03	4.23	6.07	1	0.00
Bcast	89	2.94e+03	4.23	6.06	1	0.00
Bcast	75	2.94e+03	4.22	6.05	1	0.00
Bcast	60	2.94e+03	4.22	6.05	1	0.00
Bcast	95	2.93e+03	4.22	6.04	1	0.00
Bcast	18	2.92e+03	4.20	6.02	1	0.00
Bcast	39	2.92e+03	4.20	6.01	1	0.00
Bcast	32	2.91e+03	4.19	6.00	1	0.00
Recv	5	519	0.75	1.07	150	0.00
Recv	69	393	0.57	0.81	1	0.00
Recv	56	389	0.56	0.80	1	0.00
Recv	49	385	0.55	0.79	1	0.00
Recv	76	366	0.53	0.75	1	0.00

Hybrid - Parallelized implementation and approach

Main Ideas:

1) Split problem to smaller subtrees containing part of the generated points and find local NN (MPI)
2) Solve N queries in parallel using a shared memory model inside each MPI Process (OMP)



Hybrid - Final Performance Results

Perf Data

1) ~36% nearest → Point::distance_squared
2) ~22% Point::compare
3) ~22% Utility::generate_problem → page faults !

Calculated speed-up: 6.05 (Amdahl's law: 6.78)

Calculating distance between 2 points and sorting points in order to create the binary tree are crucial for algorithm's correctness.

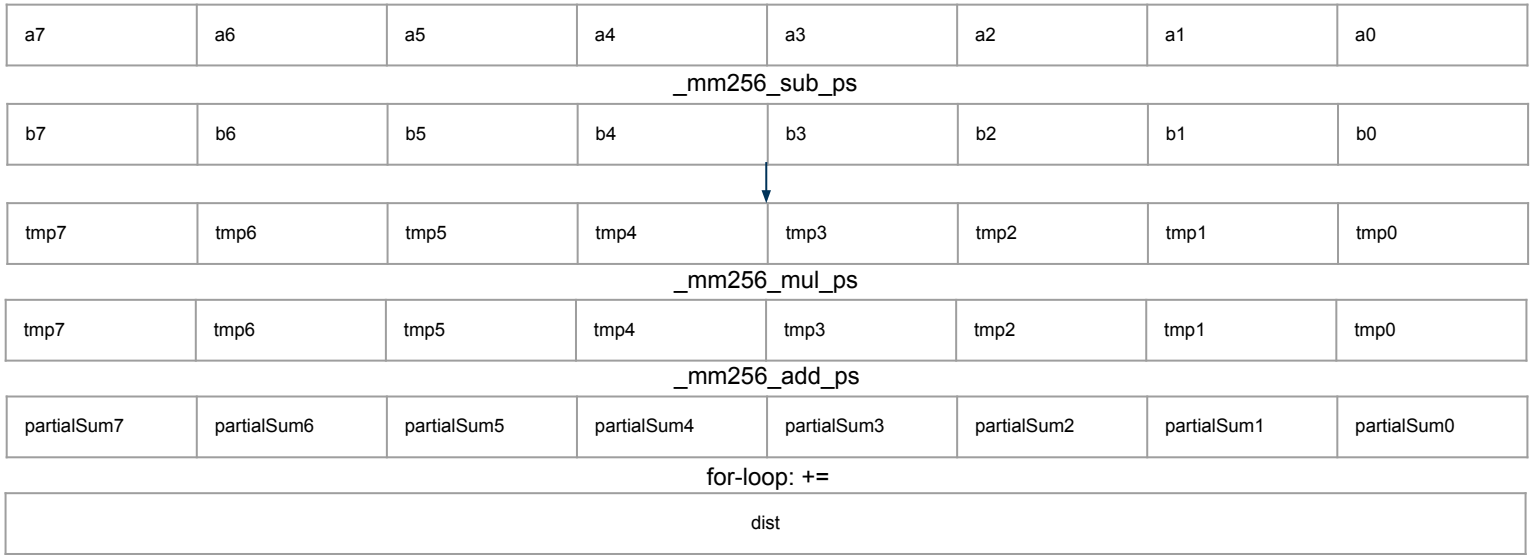
All hardware threads share equally the total amount of work needed to calculate squared distance

Problem generation can't be avoided without changing the given functions

Bonus - Parallelized implementation and approach

Initial Idea:

- Point::distance_squared is ideal for using SIMD operations
- We can use standard SIMD operations to compute 8 values at a time
- Save results as temporary partial sum and compute final distance by adding all partial sums at the end
- No need for remainder loop, since dimension is a multiple of 8



Bonus – Final Performance Results

Perf Data:

1. 16.73% opal_progress
2. 12.62% Point::distance_squared
3. 10.45% Point::compare
4. 9.31% Utility::generate_problem

Calculated speedup: 6.45 (Amdahl's Law 6.78)

Conclusion

Approach	Speedup
OpenMP	3.56
MPI	7.47
Hybrid	6.05
Bonus(SIMD with intrinsics)	6.45