Parallel K-Means

Issues and proposed improvements:

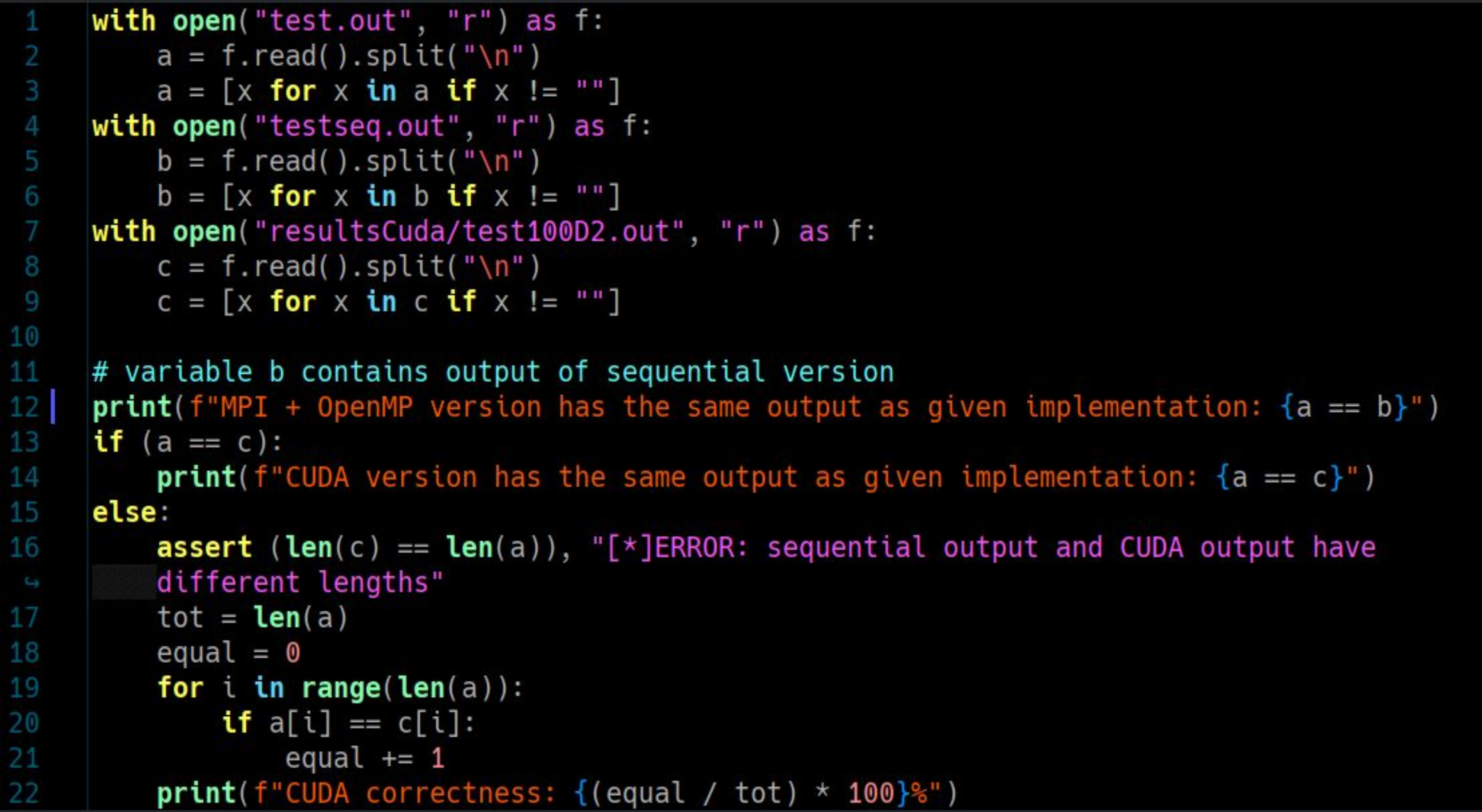
- Correctness
- CUDA maxDist race condition
- Possible CUDA optimizations
- MPI initialization
- MPI Safety
- Pragma correctness

Correctness

The correctness of the proposed implementations is evaluated using the correctness.py script.

We see that OpenMP + MPI version has a 100% accuracy compared to the sequential version, while CUDA only stops at 90%

This is probably because of float precision differences between GPU and CPU, in particular in the difference between results of sqrtf() on CPU and GPU.



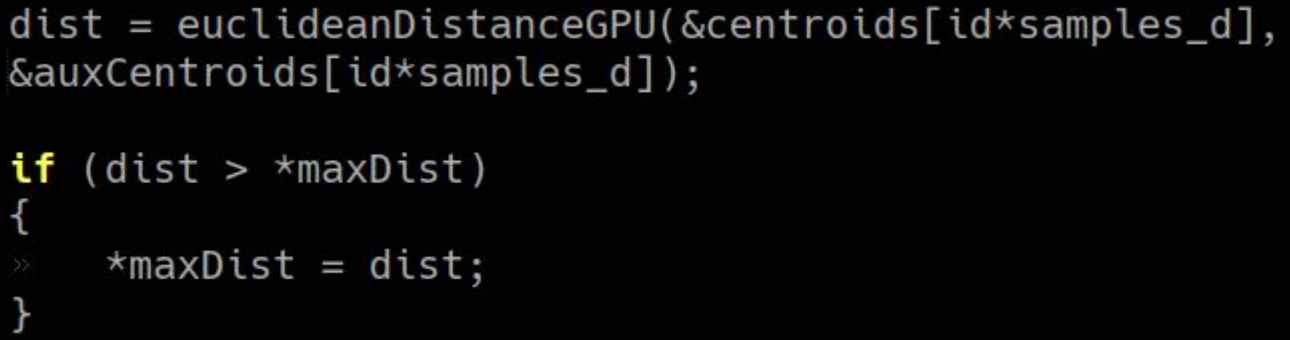
```python
with open("test.out", "r") as f:
    a = f.read().split("\n")
    a = [x for x in a if x != ""]
with open("testseq.out", "r") as f:
    b = f.read().split("\n")
    b = [x for x in b if x != ""]
with open("resultsCuda/test100D2.out", "r") as f:
    c = f.read().split("\n")
    c = [x for x in c if x != ""]

# variable b contains output of sequential version
print(f"MPI + OpenMP version has the same output as given implementation: {a == b}")
if (a == c):
    print(f"CUDA version has the same output as given implementation: {a == c}")
else:
    assert (len(c) == len(a)), "[*]ERROR: sequential output and CUDA output have different lengths"
    tot = len(a)
    equal = 0
    for i in range(len(a)):
        if a[i] == c[i]:
            equal += 1
    print(f"CUDA correctness: {(equal / tot) * 100}%")
```

CUDA maxDist Issue

There is a race condition between threads that update the variable maxDist in one of the kernels of the CUDA version. This is due to the fact that CUDA does not have a native implementation that performs the atomic version of the max operation for floating point numbers.

There are many possible solutions to this problem, while arguably the best solution would be to implement the operation yourself.

maxDist Issue



```
dist = euclideanDistanceGPU(&centroids[id*samples_d],
&auxCentroids[id*samples_d]);

if (dist > *maxDist)
{
»   *maxDist = dist;
}
```

In our code we ignore this problem on purpose, based on the assumption that it does not cause calculation errors and that the worst it could do would be to slow down the program by making it perform more iterations.
However, it would be an issue if the precision of our application was a major concern.

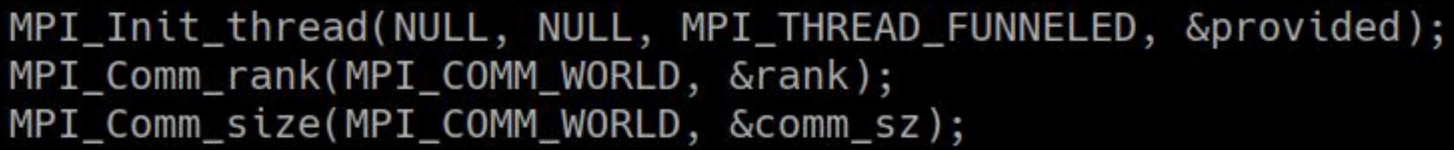
Possible CUDA optimizations

Possible code optimizations could be:

- Aggregating kernels: recalculateCentroidsStep2GPU and Step3 could just be one kernel.
- Move the entire loop logic to one __global__ kernel, and handle the rest with __device__ functions.
- Implement another algorithm that is more suited to a GPU architecture.

MPI initialization

MPI is initialized using MPI_Init(NULL, NULL), but it would be more correct to use the following initialization:

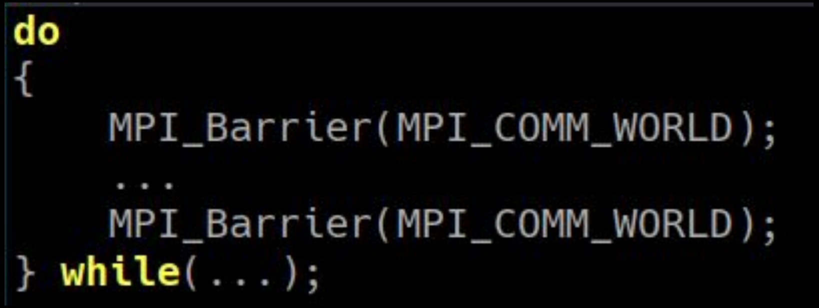


```
MPI_Init_thread(NULL, NULL, MPI_THREAD_FUNNELED, &provided);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
```

Like this we correctly initialize MPI to MPI_THREAD_FUNNELED in order to allow only the main threads to call MPI functions.

MPI safety

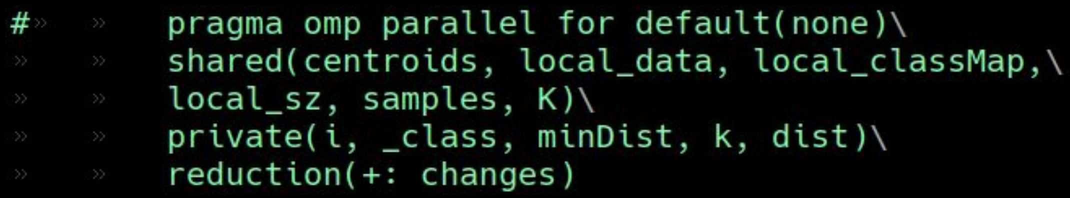
We do not actively check that ranks in MPI are always on the same iteration of the loop.
There are collective operations that mitigate heavily the possible occurrence of this issue, but a more correct approach would be to enclose the logic between two barriers.



```
do
{
    MPI_Barrier(MPI_COMM_WORLD);
    ...
    MPI_Barrier(MPI_COMM_WORLD);
} while(...);
```

Pragma correctness

Some pragmas have redundancy issues.
The first pragma should be slightly changed like so to be more correct:



```
#		pragma omp parallel for default(none)\
		shared(centroids, local_data, local_classMap,\
		local_sz, samples, K)\
		private(i, _class, minDist, k, dist)\
		reduction(+: changes)
```

default(none) needs to be specified if we want to use shared() and private() clauses. Also, atomic addition on the variable *changes* can be transformed in a reduction.

Thank you!