HPCSE II - Exercise 4

Anian Ruoss

April 24, 2019

Task 1

As in Task 2 of Part 2 of Homework 3, we use the head2DSolver to model the temperature distribution on the steel sheets. For every torch we want to determine the optimal beam width, beam intensity and x- and y-coordinates meaning that we have 16 parameters in total for the 4 robotic torches. For all parameters we know the upper and the lower bounds given by:

- $x \in [0.0, 0.5]$ for torches 1 and 2
- $x \in [0.5, 1.0]$ for torches 3 and 4
- $y \in [0.0, 1.0]$ for all torches
- beam intensity $\in [0.4, 0.6]$ for all torches
- beam width $\in [0.04, 0.06]$ for all torches

Since we do not have any additional information about the parameter distributions, we model all parameters as uniformly distributed between within their respective bounds. We employ Korali's CMA-ES solver to maximize the posterior distribution of the parameters (even though we don't really have prior distributions, but the posterior was given in the template) to find the optimal parameter values and we display the results in Listing 1.

Listing 1: Korali output when maximizing the likelihood of the heat distribution given the model with four candles.

```
[Korali] Starting CMAES. Parameters: 17, Seed: 0xFFFFFFFFFFFFFFFF
...
[Korali] Finished - Reason: Object variable changes < 1.00e-06
[Korali] Parameter 'Sigma' Value: 0.937023
[Korali] Parameter 'torch_1_x' Value: 0.242895
[Korali] Parameter 'torch_1_y' Value: 0.240862
[Korali] Parameter 'torch_1_intensity' Value: 0.505957
[Korali] Parameter 'torch_1_width' Value: 0.046153
[Korali] Parameter 'torch_2_x' Value: 0.251464
```

```
[Korali] Parameter 'torch_2_y' Value: 0.741347
[Korali] Parameter 'torch_2_intensity' Value: 0.479662
[Korali] Parameter 'torch_2_width' Value: 0.051501
[Korali] Parameter 'torch_3_x' Value: 0.757994
[Korali] Parameter 'torch_3_y' Value: 0.254470
[Korali] Parameter 'torch_3_intensity' Value: 0.474448
[Korali] Parameter 'torch_3_width' Value: 0.054619
[Korali] Parameter 'torch_4_x' Value: 0.760434
[Korali] Parameter 'torch_4_y' Value: 0.770577
[Korali] Parameter 'torch_4_intensity' Value: 0.475872
[Korali] Parameter 'torch_4_width' Value: 0.059926
[Korali] Total Elapsed Time: 234.156253s
```

Korali performs the optimization in roughly 4 minutes. We know from the lecture that CMA-ES is embarrassingly parallel since we can compute and evaluate every sample independently, which can be achieved by parallelizing the two for-loops at lines 11 and 14. Typically the evaluation is a lot more costly than random number generation, which is why we should already observe a considerable speedup when only parallelizing the loop at line 14.

Task 2

We run the single tasking engine and display its output in Listing 2.

Listing 2: Output from executing the single tasking engine.

```
Processing 240 Samples each with 2 Parameter(s)...
Verification Passed
Total Running Time: 29.717s
```

a)

Since all samples are well-known at the beginning, they can be distributed evenly among all ranks and gathered back to one rank once the evaluations are completed. We implement this divide-and-conquer strategy with UPC++ and MPI and display the results obtained from running the implementations with 24 ranks on an Euler compute node in listings 3 and 4 respectively.

Listing 3: Output from executing the UPC++ tasking engine with the divide-and-conquer strategy.

```
Verification Passed
Total time:           1.37665
Average time:         1.17792
Load imbalance ratio: 0.144362
```

Listing 4: Output from executing the MPI tasking engine with the divide-and-conquer strategy.

```
Verification Passed
Total time:           1.31242
```

```
Average time:          1.17451
Load imbalance ratio: 0.10508
```

We observe a speedup of ≈ 21.5 for UPC++ and ≈ 22.5 for MPI and thus we report efficiencies of $\approx 90\%$ for UPC++ and $\approx 94.5\%$ for MPI. Both implementations suffer from a relatively high load imbalance ratio (≈ 0.145 for UPC++ and ≈ 0.105 for MPI) which results from the fluctuation in evaluation times. The MPI approach is practically identical to the UPC++ code but the collective operations in MPI allow for a much cleaner implementation. In general, MPI feels more natural since it requires very explicit communication.

b)

To solve the load imbalance problem observed in Task 2a) we implement the producer-consumer strategy which takes advantage of the fact that the evaluation times differ and distributes workloads according to rank availability and not according to a fixed scheme. We display the results obtained from running the UPC++ and MPI implementations with 24 ranks on an Euler compute node in listings 5 and 6 respectively.

Listing 5: Output from executing the UPC++ tasking engine with the producer-consumer strategy.

```
Processing 240 Samples each with 2 Parameter(s)...
Verification Passed
Total time:           1.32383
Average time:         1.25486
Load imbalance ratio: 0.0520984
```

Listing 6: Output from executing the MPI tasking engine with the producer-consumer strategy.

```
Verification Passed
Total time:           1.32354
Average time:         1.24246
Load imbalance ratio: 0.0612577
```

We observe for both UPC++ and MPI that the load imbalance ratio drops significantly compared to the divide-and-conquer strategy, although more drastically for UPC++. For UPC++ we observe that the total time decreases slightly compared to the divide-and-conquer strategy (speedup: $\approx$ 22.5, efficiency: $\approx 93.5\%$), whereas for the MPI implementation the total running time increases marginally (speedup: ≈ 22.5, efficiency: $\approx 93.5\%$). Whereas the MPI implementation outperformed UPC++ for the divide-and-conquer strategy, they are now practically identical. The MPI approach differs slightly from the UPC++ approach:

- UPC++ employs a queue of consumers which contain a future among other data. The producer iterates over the queue and checks whether a RPC has completed before distributing another sample to the idling rank.
- The MPI implementation does not require a queue as the producer just sends samples and listens for results until all samples have been evaluated. Unlike the UPC++ implementation, we need to explicitly tell every rank that the evaluation has completed once all samples have been processed.

Even though we were told that the producer-consumer problem would be easier to implement in UPC++ than in MPI, the MPI approach feels cleaner as it is more explicit[1].

Task 3

We run the single tasking engine and display its output in Listing 7.

Listing 7: Output from executing the single tasking engine.

```
Processing 240 Samples (24 initially available), each with 2 Parameter(s)...
Verification Passed
Total Running Time: 29.458s
```

Since not all samples are available at the beginning of the generation it does not make sense to have more ranks than initially available samples and we enforce this constraint with an assert. Apart from that our approaches are similar to those from Task2b) and we display the results obtained from running the UPC++ and MPI implementations with 24 ranks on an Euler compute node in listings 8 and 9 respectively.

Listing 8: Output from executing the UPC++ tasking engine .

```
Processing 240 Samples (24 initially available), each with 2 Parameter(s)...
Verification Passed
Total Running Time: 1.355s
```

Listing 9: Output from executing the MPI tasking engine .

```
Verification Passed
Total Running Time: 1.311s
```

We observe speedups of ≈ 21.5 for UPC++ and ≈ 22.5 for MPI and correspondingly efficiencies of $\approx 90.5\%$ for UPC++ and $\approx 93.5\%$ for MPI. One challenge we faced during the implementation of this exercise is that

[1] From *The Zen of Python, by Tim Peters*: "Explicit is better than implicit.".

getSample() and *updateEvaluation()* have to be called from the root rank and that updateEvaluation has to be called after the evaluation has completed on a consumer rank. This problem can be elegantly solved with the *then()* method from UPC++ which executes a function on root after the RPC has completed. Our MPI implementation is basically equivalent to that of Task2b) and thus we refer to Task2b) for the discussion of the difference of the two approaches. Using the *then*() function is definitely more elegant than sending data back and forth as is required for MPI.