1 Task 1

1.1 Subtask 1

As previously seen in previous homeworks, the `heat2DSolver` engine is used to find the torch parameters on the steel sheets.
This time, we have $n = 4$ torches with four parameters (2D coordinates, beam width, beam intensity) each for a total of 16 parameters. The constraints on those are as given in the task description; without any further information, uniform distributions are assumed for all parameters.
Also as done previously, Korali's CMA-ES solver is put to work to maximize posterior distributions of the parameters under consideration while considering our input a non-informative prior.
To be able to compare the runtime to parallel implementations later, we use a population (generation) size of 23.

1.2 Subtask 2

The results are as shown in listing 1.

Listing 1: Task 1: Korali/CMA-ES output: Most likely (x,y) position and characteristics (beam width, intensity) of each of the robotic torches.

```
[Korali] Starting CMAES. Parameters: 17, Seed: 0xFFFFFFFFFFFFFFFF
[Korali] Gen 1 - Elapsed Time: 0.911218, Objective Function Change: 1.14e+01
...
[Korali] Gen 428 - Elapsed Time: 0.896176, Objective Function Change: 2.35e-03
[Korali] Finished - Reason: Object variable changes < 1.00e-06
[Korali] Parameter 'Sigma' Value: 0.935673
[Korali] Parameter 'xpos_1' Value: 0.243243
[Korali] Parameter 'ypos_1' Value: 0.241235
[Korali] Parameter 'beamIntensity_1' Value: 0.437139
[Korali] Parameter 'beamWidth_1' Value: 0.053572
[Korali] Parameter 'xpos_2' Value: 0.251330
[Korali] Parameter 'ypos_2' Value: 0.741884
[Korali] Parameter 'beamIntensity_2' Value: 0.438952
[Korali] Parameter 'beamWidth_2' Value: 0.056372
[Korali] Parameter 'xpos_3' Value: 0.758039
[Korali] Parameter 'ypos_3' Value: 0.254297
[Korali] Parameter 'beamIntensity_3' Value: 0.505204
[Korali] Parameter 'beamWidth_3' Value: 0.051094
[Korali] Parameter 'ypos_4' Value: 0.760546
[Korali] Parameter 'ypos_4' Value: 0.769991
[Korali] Parameter 'beamIntensity_4' Value: 0.504809
[Korali] Parameter 'beamWidth_4' Value: 0.056403
[Korali] Total Elapsed Time: 382.275609s
```

1.3 Subtask 3

The runtime with a population size of 23 comes to about $380s$.
As previously discussed in class and as evident in the given pseudo-code listing, parallelizing the CMA-ES solver should be straightforward as sample generation and evaluation (lines 11ff and 14ff) can be performed independently and thus completely in parallel.

2 Task 2

The reference implementation using a single task achieves the result shown in listing 2.

Listing 2: Task 2: Reference single-task implementation output.

```
Processing 240 Samples each with 2 Parameter(s)...
Verification Passed
Total Running Time: 29.508s
```

2.1 Subtask a

2.1.1 Subtask a.1

The divide-and-conquer parallel version runs the same amount of calculation time about $\frac{29.508}{1.36} \simeq \frac{29.508}{1.38} \simeq 21$ times faster when run with `n=24` processes (listings 3 and 4). The efficiency for both implementations thus is almost identical at $\frac{21}{24} \simeq 0.88$.

Listing 3: Task 2a: UPCXX divide-and-conquer parallel implementation output (n=24).

```
Processing 240 Samples each with 2 Parameter(s)...
Verification Passed
Total time:      28.694s
Average time:    1.196s
Maximum time:    1.360s
Maximum time/avg time = 1.138
Load imbalance ratio = 0.121
```

Listing 4: Task 2a: MPI divide-and-conquer parallel implementation output (n=24).

```
Processing 240 Samples each with 2 Parameter(s)...
Verification Passed
Total time:      29.060s
Average time:    1.211s
Maximum time:    1.380s
Maximum time/avg time = 1.140
Load imbalance ratio = 0.123
```

2.1.2 Subtask a.2

Both implementations also show similar load imbalance ratios of $\simeq 0.12$ which are owed to the static work distribution while there are fluctuating evalutation times.

2.1.3 Subtask a.3

The MPI implementation follows the same design approach. However, `MPI_Scatter` and `MPI_Gather` helps a lot with making data distribution and collection straightforward whereas this seemed more cumbersome with UPCXX's objects. For this task, MPI felt the more natural and thus easier approach.

2.2 Subtask b

2.2.1 Subtask b.1

Both implementations show a much improved imbalance ratios of $\simeq 0.05$ respectively $\simeq 0.06$ which are due to the optimized dynamic workload distribution.

Listing 5: Task 2b: UPCXX producer-consumer parallel implementation output (n=24).

```
Processing 240 Samples each with 2 Parameter(s)...
Verification Passed
Total time:      28.665s
Average time:    1.246s
Maximum time:    1.311s
Maximum time/avg time = 1.052
Load imbalance ratio = 0.049
```

Listing 6: Task 2b: MPI producer-consumer parallel implementation output (n=24).

```
Processing 240 Samples each with 2 Parameter(s)...
Verification Passed
Total time:      29.317s
Average time:    1.275s
Maximum time:    1.360s
Maximum time/avg time = 1.067
Load imbalance ratio = 0.063
```

2.2.2 Subtask b.2

The producer-consumer parallel version runs the same amount of calculation time about $\frac{29.508}{1.311} \simeq 22$ respectively $\frac{29.508}{1.36} \simeq 21$ times faster when run with `n=24` processes (listings 5 and 6). The efficiency for both implementations thus is slightly improved over the divide-and-conquer implementations.
It also becomes evident that the UPCXX implementation is at an efficiency advantage here: This most probably is due to the communication overhead required by the MPI implementation which is less suited to this kind of workload distribution.

2.2.3 Subtask b.3

The MPI implementation follows a somewhat different approach here, as message-based communication doesn't really lend itself to taking over the queue concept chosen for the UPCXX implementation. While the MPI version actually looks quite simple and clean, its performance is less optimal as described above.

3 Task 3

The reference implementation using a single task achieves the result shown in listing 7.

Listing 7: Task 3: Reference single-task implementation output.

```
Processing 240 Samples (24 initially available), each with 2 Parameter(s)...
Verification Passed
Total Running Time: 27.912s
```

3.1 Subtask 1

The producer-consumer parallel versions are very closely following the UPCXX and MPI implementations written for task 2 above. As oversubscribing the samples during the initial ramp-up with only 24 available was tested not to hurt performance nor correctness, no safeguards for the number of consumers were implemented. Other than that, the only other changes from task 2 were around checking out and checking in samples sequentially. In summary, no special challenges were discovered; the UPCXX implementation still relies on futures, RPCs and views. Using UPCXX's `then` tool for chaining the evaluation to the sample check-in could have been used, but didn't seem worth the effort when looking at the performance.

Listing 8: Task 3: UPCXX producer-consumer parallel implementation output (n=24).

```
Processing 240 Samples (24 initially available), each with 2 Parameter(s)...
Verification Passed
Total time:      29.433s
Average time:    1.226s
Maximum time:    1.360s
Maximum time/avg time = 1.109
Load imbalance ratio = 0.099
```

Listing 9: Task 3: MPI producer-consumer parallel implementation output (n=24).

```
Processing 240 Samples (24 initially available), each with 2 Parameter(s)...
Verification Passed
Total time:      28.722s
Average time:    1.249s
Maximum time:    1.309s
Maximum time/avg time = 1.048
Load imbalance ratio = 0.046
```

3.2 Subtask 2

The MPI implementation follows the same design approach as described for task 2b above. While again more cumbersome due to the somewhat unwieldy message passing, its performance remains on par for the given problem size.

4 Task 4

The reference implementation using a single task achieves the result already shown in listing 1 above.

4.1 Subtask 1

The main problem was adapting the approaches developed during the previous tasks above into the hard-wired structure of Korali's conduit system while not being aware of its inner workings. After some header file analysis and running the single task implementation with print statements, the functionality could be deduced sufficiently. Also, reliance on global variables seemed unavoidable while not pretty.

4.2 Subtask 2

As we had run the single task reference version on a population size of 23 to get a useful reference, the UPCXX and MPI implementations were run with the same parameters and `task1_n4.cpp` source file.
The UPCXX implementation ran $\frac{382.28}{22.34} \simeq 17.11$ faster with an efficiency of $\frac{17.11}{24} \simeq 0.71$ while the MPI implementation ran $\frac{382.28}{22.15} \simeq 17.25$ faster with an efficiency of $\frac{17.25}{24} \simeq 0.72$ while

Listing 10: Task 4: UPCXX producer-consumer parallel implementation output (n=24).

```
[Korali] Starting CMAES. Parameters: 17, Seed: 0xFFFFFFFFFFFFFFFF
[Korali] Gen 1 - Elapsed Time: 0.065359, Objective Function Change: 1.14e+01
...
[Korali] Gen 428 - Elapsed Time: 0.052010, Objective Function Change: 2.35e-03
[Korali] Finished - Reason: Object variable changes < 1.00e-06
[Korali] Parameter 'Sigma' Value: 0.935673
[Korali] Parameter 'xpos_1' Value: 0.243243
[Korali] Parameter 'ypos_1' Value: 0.241235
[Korali] Parameter 'beamIntensity_1' Value: 0.437139
[Korali] Parameter 'beamWidth_1' Value: 0.053572
[Korali] Parameter 'xpos_2' Value: 0.251330
[Korali] Parameter 'ypos_2' Value: 0.741884
[Korali] Parameter 'beamIntensity_2' Value: 0.438952
[Korali] Parameter 'beamWidth_2' Value: 0.056372
[Korali] Parameter 'xpos_3' Value: 0.758039
[Korali] Parameter 'ypos_3' Value: 0.254297
[Korali] Parameter 'beamIntensity_3' Value: 0.505204
[Korali] Parameter 'beamWidth_3' Value: 0.051094
[Korali] Parameter 'ypos_4' Value: 0.760546
[Korali] Parameter 'ypos_4' Value: 0.769991
[Korali] Parameter 'beamIntensity_4' Value: 0.504809
[Korali] Parameter 'beamWidth_4' Value: 0.056403
[Korali] Total Elapsed Time: 22.337454s
```

Listing 11: Task 4: MPI producer-consumer parallel implementation output (n=24).

```
[Korali] Starting CMAES. Parameters: 17, Seed: 0xFFFFFFFFFFFFFFFF
[Korali] Gen 1 - Elapsed Time: 0.061893, Objective Function Change: 1.14e+01
...
[Korali] Gen 428 - Elapsed Time: 0.051645, Objective Function Change: 2.35e-03
[Korali] Finished - Reason: Object variable changes < 1.00e-06
[Korali] Parameter 'Sigma' Value: 0.935673
[Korali] Parameter 'xpos_1' Value: 0.243243
[Korali] Parameter 'ypos_1' Value: 0.241235
[Korali] Parameter 'beamIntensity_1' Value: 0.437139
[Korali] Parameter 'beamWidth_1' Value: 0.053572
[Korali] Parameter 'xpos_2' Value: 0.251330
[Korali] Parameter 'ypos_2' Value: 0.741884
[Korali] Parameter 'beamIntensity_2' Value: 0.438952
[Korali] Parameter 'beamWidth_2' Value: 0.056372
[Korali] Parameter 'xpos_3' Value: 0.758039
[Korali] Parameter 'ypos_3' Value: 0.254297
[Korali] Parameter 'beamIntensity_3' Value: 0.505204
[Korali] Parameter 'beamWidth_3' Value: 0.051094
[Korali] Parameter 'ypos_4' Value: 0.760546
[Korali] Parameter 'ypos_4' Value: 0.769991
[Korali] Parameter 'beamIntensity_4' Value: 0.504809
[Korali] Parameter 'beamWidth_4' Value: 0.056403
[Korali] Total Elapsed Time: 22.152529s
```

We observe that the obtained parameter values are identical and thus conclude correctness. Also, as both implementations are more than 10 times faster than the reference implementations, expecting a monetary reward wouldn't seem unreasonable.

4.3 Subtask 3

As discussed and approved on Piazza, the MPI implementation was built off of the `single.cpp` conduit and placed in `single_mod_mpi.cpp`. This helps to avoid having to adapt any header files. Modifying the `Makefile` accordingly allows for straightforward compilation without having to rename any files. Initially, a similar approach to the one used in task 3 was used. It then became evident that keeping the consumers spinning at the end of a generation by sending messages back and forth only worked up to about 14 consumers. With more ranks employed, the spinning ranks would then drown and surpress the last ranks wanting to report their results at the end of a generation and thus slow down progress exponentially. In a somewhat ugly hack, the implementation was adapted to a hybrid where the message passing was supported by a standby queue where unemployed consumers are parked at the end of a generation to be reactivated during the next generation. As seen from the runtime, performance however remains comparable to the UPCXX implementation.