HW 4: High-Throughput Computing with UPC++

Issued: April 1, 2019
Due Date: April 22, 2019 10:00am

Warm-Up & Information

This homework requires the use of Korali, our new Uncertainty Quantification and Optimization framework developed at ETH Zürich. Along with this pdf, you should download the provided code which contains the following files/folders:

- **korali/** contains a ready-to-use, pre-compiled version of Korali. The library and some of the task models will only work on Euler (or compatible ETH systems). The only step required to make it work is to decompress the tar ball: $tar -xzf korali/lib/libkorali.tar.gz

- **plotting/** Contains Python and MATLAB codes required to plot the distribution plots based on the output of Korali’s TMCMC engine.

- **modules.sh** Execute it to load the modules required to run this homework on Euler.

- **task1/** Contains the skeleton code for Task 1.

- **task2/** Contains the skeleton code for Task 2.

- **task3/** Contains the skeleton code for Task 3.

- **task1/conduits/upcxx.cpp** Is the skeleton code for Task 4.

Guidelines for Moodle submission:

- For HW4: at least 4 solution codes: task1.cpp task2a.cpp task2b.cpp task3.cpp upcxx.cpp. You can also include codes for optional items.

- Your report PDF. Do not forget to answer every question in the HWs.
Grading

- CSE Students: Max 170 points to be converted linearly to a 1-to-6 scale.
- Engineering Students: Max 100 points to be converted linearly to a 1-to-6 scale.
- Bonus: Every correct optional item submitted will grant 0.25 bonus points to improve the grade of this or another homework, up to 6 points.

Installing and Using UPC++/Korali Please, check this guide to learn to install, compile, and install a UPC++ application. To select Korali's conduit, run: $export KORALI_CONDUIT=single, for the single-core conduit, and: $export KORALI_CONDUIT=upcxx, for the UPC++ conduit (to be developed by you).
Task 1: n-Candle Problem. (40 Points)

Here we are revisiting the n-Candle problem from Homework 3b. The company is expanding, thanks to the consistently good quality of the produced chassis. The company decides on investing a large amount of capital on the automation of the production and heat treatment of the metal sheets on the way to mass production of the cars’ chassis. The company decides on investing on 4 robotic heat treatment units that will work in parallel on the 2D metal plates. The robotic torches work with a beam of a certain width and intensity and can be placed quite close to each other.

As the head of operations in the factory, your task is to manage the operation of the four torches in order for the metal sheets to pass the structural failure tests. Again, you are armed with the automatic machine that registers the temperature at multiple points of each sheet, just after they leave the treatment facility. The structural engineers of the company managed to create an optimal temperature profile of a metal sheet that would according to their simulations pass all integrity and failure tests. Your task will be to define the position and the technical characteristics of each torch beam (width and intensity) to replicate the optimal temperature profile you obtained by your engineers.

You decide to use your previous knowledge of Koral'i to solve this optimization problem. The metal sheets are still square and 1m long. The specifications for the robotic torches the company acquired are: "The intensity of the beams for metal treatment should be within $0.4 - 0.6$ (a.u.), whereas the width of the beams should be kept between $0.04$ m and $0.06$ m. Due to spatial constraints in the factory, you decide on splitting the torches along the two sides of the heat treatment facility, i.e. two of the torches will be placed in the lower half of the x-axis ($0.0 - 0.5$ m) and the other two torches will be placed in the upper half of the x-axis ($0.5 - 1.0$ m).

Because the positioning of the robotic torches is much more accurate than the human hands, the grid size of the solver has increased to $2^7 + 1$ elements per side (heat2d.cpp), which increases the run time of the heat-2d solver.

- Describe the model and report the number of parameters you will need to represent this problem in Koral'i.
- What is the most likely ($x, y$) position and characteristics (beam width, intensity) of each of the robotic torches?
- How much time did Koral'i take to solve this problem? What could we possibly do to improve this, or which parts would you parallelize first in CMA-ES? (for
Algorithm 1 CMA-ES Pseudo Code

1: procedure Optimize
2: Initialize CMA-ES parameters:
3: \( N \leftarrow \) number dimensions
4: \( \lambda \leftarrow \) number offspring population size
5: \( u \leftarrow \) parent population size for next generation
6: \( \sigma^{(0)} \leftarrow \) initial standard deviation, respectively step size
7: \( C^{(0)} \leftarrow I \) (initial covariance matrix)
8: \( \mu^{(0)} \leftarrow \) initial mean
9: while stopping criterion is not met do
10: \( A \leftarrow \text{chol}(C^{(g+1)}) \)
11: for \( k = 1 : \lambda \) do
12: \( x_k \sim N(\mu^{(g+1)}, A) \) (generate new Sample Population)
13: end for
14: for \( k = 1 : \lambda \) do
15: evaluate \( f(x_k) \)
16: end for
17: sort results, find best \( u \) samples
18: update best ever solution
19: \( C^{(g+1)} \leftarrow \) updated Covariance Matrix (based on \( u \) best samples)
20: \( \mu^{(g+1)} \leftarrow \) updated Mean (based on \( u \) best samples)
21: \( \sigma^{(g+1)} \leftarrow \) updated Step Size
22: end while
23: end procedure

An example solution for the implementation can be found in hw4/solution_codes/task1/task1.cpp.

- The model describes the solution of the heat equation with 4 heat sources. Each heat source has four parameters, i.e. the \( x \) and \( y \) coordinates of the candles and the intensity and width of the beam of the torches. In order to solve this problem, with the prior information given, we need to define the prior distributions for \( 4 \times 4 = 16 \) parameters. The prior distributions for each of the parameters are:

  1. \( x \) coordinate: uniform prior distribution, bound at \( x_{low} = 0.0 \) and \( x_{up} = 0.5 \) for torches 1 and 2; or, bound at \( x_{low} = 0.5 \) and \( x_{up} = 1.0 \) for torches 3 and 4
  2. \( y \) coordinate: uniform prior distribution, bound at \( x_{low} = 0.0 \) and \( x_{up} = 1.0 \) for all torches
  3. intensity of the torch: uniform prior distribution, bound at \( x_{low} = 0.4 \) and \( x_{up} = 0.6 \) for all torches
4. width of the torch: uniform prior distribution, bound at \( x_{\text{low}} = 0.04 \) and \( x_{\text{up}} = 0.06 \) for all torches

In order to solve the optimization problem for all 16 parameters, we set up a problem of type \texttt{Korali::Problem::Posterior} and we run it with the solver \texttt{Korali::Solver::CMAES}.

- The CMA-ES algorithm of Korali is applied to maximize the likelihood and it outputs the following MAP values for the four heat sources:
  1. \( \text{PosX}_1 = 0.242891, \text{PosY}_1 = 0.240871, \text{Inten}_1 = 0.459482, \text{Width}_1 = 0.050817 \)
  2. \( \text{PosX}_2 = 0.760423, \text{PosY}_2 = 0.770744, \text{Inten}_2 = 0.531139, \text{Width}_2 = 0.053721 \)
  3. \( \text{PosX}_3 = 0.251322, \text{PosY}_3 = 0.741702, \text{Inten}_3 = 0.435760, \text{Width}_3 = 0.056847 \)
  4. \( \text{PosX}_4 = 0.758423, \text{PosY}_4 = 0.254739, \text{Inten}_4 = 0.472857, \text{Width}_4 = 0.054764 \)

Notice that the location of candles 1-2 and 2-3 is interchangeable, which is observed after repeated runs.

- It took the Korali CMA-ES engine an average of 317 s to run this optimization problem on an interactive node. We identify that the generation and evaluation of the \( \lambda \) samples in every generation are completely independent of each other. The evaluation of the samples is the computationally most expensive part, therefore, this (lines 14-16 in Algorithm 1) would be the first part of the algorithm to parallelize, in order to improve the run-time of the application.
Task 2: Parallel Tasking (60 Points)

Your recent success as the Factory’s head of operations and Korali expert has made you a prestigious figure in the manufacturing world. Soon, you are approached by the company’s CEO, who tells you:

“You have proven that Korali is an excellent tool to determine the best treatment for car’s metal plates. I am now planning to expand our operations to compete in the airplane industry. Our initial research indicates that we will need to treat many different types and sizes of metal plates. However, your tool still takes too long to provide results for a single plate. We need to find a way to produce results in a much shorter time. If you can show that you can solve our current problem 10 times faster, we will be able to make the move. For this, you will earn a generous raise”.

Determined to prove you are up to the task, you embark yourself in a quest to parallelize Korali, making it much more efficient.

a) (30 Points) We will start by building a simple parallel tasking engine for a problem in which all samples are well-known at the beginning of the generation. Such is the case of CMA-ES in which all the new samples are taken from the previous generation’s best fitting samples. To solve this, we will apply a divide-and-conquer strategy using UPC++, distributing the samples among ranks and having each rank evaluate it’s set of samples in parallel. First, take a look, compile, and execute the single tasking engine. This engine will only run in a single-core, just like the current Korali. Build your parallel version of the engine on task2a.cpp and run it using 24 ranks on an Euler compute node. Answer:

• How much faster did your parallel version run, compared to the single-core version? What can you say about the efficiency of your implementation?

• Compare the total time taken by your engine to the average time taken by each rank. What can you say about load balance in this case?

• <Optional> Build a similar parallel engine using MPI, and answer: Did you take a similar approach? Was it easier or more difficult to implement? Provide the code as task2a_mpi.cpp and justify your answers.

Note: Your submission(s) need to successfully run checkResults() to be considered correct.
Note: This is only a proposed solution. Many different/similar approaches are considered correct.

Note: A full-node is requested in euler.

By entering the directory and running the make command, we compile the scripts. Then we run ./single to run the compiled single task engine that solves the sampling problem. You should get a message that the verification is passed and the simple sampler processes 240 samples in a total running time of approximately 29 seconds (calculated as the average of 6 runs: (28.23 + 29.75 + 29.75 + 28.93 + 28.93 + 28.93)/6).

In order to implement a parallel version of the sampler based on the divide and conquer method we need to pay attention to the following points:

- The master rank has to initialize the sampler and broadcast it then to the rest of the ranks. The broadcast command has to be followed by the .wait() method to wait for completion of the communication.
- In this case, the amount of samples is divided equally among ranks. The amount of samples per rank is given by the total number of samples divided by the total ranks $samples\text{Per\ Rank} = nSamples/ranks\text{Count}$.
- The results of each rank are saved in an array that is accessible by all ranks in the global address space. This can be achieved by defining a global pointer. Each rank has its own private pointer to the shared memory. This pointer is named resultsArray in the solution code provided. The master rank then allocates a new array in the shared memory (global address space). This array is yet invisible to the rest of the ranks. The master rank has to broadcast the address of this array in all ranks, so that all private global pointers point to the same address in the global address space. The main rank creates the global array and broadcasts its address to all other ranks.
- The ranks are then expected to write their results to the respective address space, which for rank $rankId$ is given by $rankId \cdot samples\text{Per\ Rank}$ till $(rankId+1) \cdot samples\text{Per\ Rank}-1$.
- The ranks need to evaluate their samples and update the global result array in an asynchronous fashion, overlapping communication and computation. If we do not overlap communication and computation (e.g. use the .wait() method) then we need to wait on all completions, which can be costly and tricky in case of many asynchronous ranks running in parallel. This can be achieved using a so-called conjoined future in upcxx (make_future()), where we are gathering all future together and wait for their completion.
• Each rank is evaluating its samples, updating the global result with its evaluation and conjoining the future output of rput to all futures with 
  \[ \text{fut\_all} = \text{upcxx::when\_all(fut\_all, fut)}. \]
• Finally the master rank checks the result by first downcasting the global pointer using the \text{.local()} method and calling the \text{checkResults} function.

An example solution for the implementation can be found in \text{hw4/solution\_codes/task2/task2a.cpp}. The total running time for 240 samples when using 24 ranks is approximately 1.4075 seconds (mean over 6 runs, \( (1.369 + 1.418 + 1.402 + 1.436 + 1.413 + 1.407)/6 \)). This corresponds to a speed-up of approximately \( 29/1.4075 = 20.67 \). This is a bit far from the optimal speed-up of 24. The speed-up efficiency is 20.67/24 \( \approx 86\% \). If we compare with the evaluation time for each rank, we can explain this deviation, as the evaluation time for each rank ranges from 0.9 seconds to 1.5 approximately. The average evaluation time is approximately 1.2 seconds. This would correspond to a speed up of 29/1.21 = 23.97 which is close to the optimal. The load imbalance ratio is approximately \( (1.4075 - 1.2)/1.4075 = 0.15, \) or 15\%. Although the sampling procedure is embarrassingly parallel, which explains the high level of parallel efficiency, the load of each rank is not balanced. The time each rank takes to evaluate a sample is not equal for all ranks, which is an indication of load imbalance. In this case it might make sense to try a producer-consumer strategy, where the evaluations are not distributed equally among ranks, but they are assigned to any worker currently available by a master rank.

b) (30 Points) Use task2b.cpp to build an alternative parallel engine that solves the load imbalance problem using the \text{producer-consumer} strategy, and answer:
  • What can you say about load imbalance in the producer-consumer strategy.
  • How much did your new engine improve over the divide-and-conquer strategy?
  • \text{<Optional> Build a similar parallel engine using MPI, and answer: Did you take a similar approach? Was it easier or more difficult to implement? Provide the code as task2b_mpi.cpp and justify your answers. Note: Your submission(s) need to successfully run checkResults()} to be considered correct.

The load imbalance problem can be alleviated using the producer-consumer strategy. In this case, a master rank (the producer) is responsible to assign
the evaluation of the samples to any available worker (slave). One way to do this, is by creating a FIFO queue with all idle workers. When a worker is idle (not currently evaluating any sample), the master pops the worker out of the queue and assigns an evaluation to the worker. The worker is evaluating the sample. When the worker is finished, it needs to inform the master in order to be appended again to the FIFO queue of all idle workers. The master needs to iteratively check the queue for any available workers and assign a sample to the first available one till all samples are evaluated. Finally, when all samples are evaluated, the master needs to inform the workers that the evaluations are completed. One important thing to note, is that the workers have to periodically call the \texttt{upcxx::progress()} command, to allow incoming RPCs to be processed. A sample solution can be found in file \texttt{hw4/solution_codes/task2/task2b.cpp}.

The producer consumer strategy procedure takes in total approximately 1.3 seconds to execute \((1.33 + 1.31 + 1.38 + 1.30 + 1.34 + 1.34)/6 = 1.33\). The mean evaluation time is approximately 1.27 seconds (summing up the evaluations and dividing by 23) and the load imbalance problem is alleviated as the execution times of individual ranks lie between \(1.2 - 1.33\), leading to a load imbalance ration of \(1.3 - 1.27)/1.3 = 0.023\, or \(2.3\%\). The total speed-up of this producer-consumer strategy is approximately \(29/1.33 \approx 21.8\), which is a small improvement over the divide and conquer method. The theoretical maximum in this case is 23, since the master rank is reserved for the managing of the tasks. The speed-up efficiency is \(21.8/24 \approx 91\%\). Results might deviate slightly and averaging many runs can help approximate the speed-ups more accurately, however this is out of scope of the solution. One disadvantage of the producer-consumer strategy is that one rank (the master) is sacrificed to assign the work to the workers. However, the producer consumer strategy is a more appropriate choice in cases of load imbalance (big variation in the evaluation of various samples), especially as the number of samples that need to be evaluated increases. (If the number of samples evaluated are increased, the speed-up advantage of the consumer-producer strategy over the divide and conquer method is indeed more evident.)

The following implementation points are important:

- Throughout the solution, fire and forget RPCs (\texttt{upcxx::rpc_ff}) are used. These do not send any response message to satisfy any future command, so they tend to be faster. In the master loop, the master is assigning
the evaluation of the samples to the workers. The master does not need any information returned to him. When all samples are assigned, the master again sends an `rpc_ff`, changing the value of the boolean variable `continueEvaluations` to inform the workers that all evaluations are assigned, and if they are idle, they are not needed anymore so they track their time and finalize their operation (with the `upcxx::finalize()`).

- The function `workerEvaluateSample` is send from the master to the available worker (after popping it out of the queue). The worker by executing the function, it is evaluating the current sample, and calling a `upcxx::rpc_ff` to the master, passing its rank, the sample it evaluated and the evaluation as arguments to the function. The master needs to push the worker back to the queue (since the evaluation is finished and the worker is idle) and update the pointer to the global address space with the results with the current evaluation (after first downcasting with the `.local()` method).

- **Important:** The worker needs to call the function `upcxx::progress()` iteratively till all evaluations are finished to ensure that the rpc calls are run.
Task 3: Real-Time Sampling (30 Points)

<Mandatory for CSE Students>
(Optional for Engineering Students>

Not every sampling/optimization engine produces all the samples at the beginning of the generation. Instead, other methods like Transitional MCMC, will only evaluate an initial set of samples and use their evaluations to produce a set of new candidate samples to evaluate. Therefore, it is not possible to employ a static distribution of work at the beginning.

In this exercise, we provide a sampler that only produces 24 initial samples, and requires to be updated with their evaluations before being able to produce new samples.

You will have to design an engine that constantly receives the latest results to produce new samples to verify. Check single to gain an idea of what functions are used to update evaluations and obtain new samples to be evaluated. Answer:

- What challenges did you face during this exercise? What UPC++ tools did you use to overcome them?
- <Optional> Build a similar parallel engine using MPI, and answer: Did you take a similar approach? Was it easier or more difficult to implement? Provide the code as task3a_mpi.cpp and justify your answers.

Following challenges have to be overcome in order to solve this task:

1. Master needs to initialize the samples and distribute them to the workers.
2. Work needs to be distributed and the workers should execute processSample.
3. Workers have to return the results, respectively notify the master rank, such that master can updateEvaluation and produce new samples.
4. Master needs to keep track of busy and idle workers in order to assign them more work.
5. Master needs to notify the workers when all samples are processed.

We use the following design and UPC++ tools to do this (please consult the solution code solution_codes/task3/task3.cpp along the documentation below):

1. Let only the master (rankId = 0) call initializeSampler and then we broadcast sampleArray (sampleArray must be of type upcxx::global_ptr, we cannot broadcast pointers to local addresses)
2. Keep the workers in a loop such that they first check for termination/completion and readiness before they pull new data with `upcxx::rget` (from `sampleArray`) and call `evaluateSample`.

3. After evaluation (step 2) each worker notifies the master, which in turn calls `updateEvaluation`, increases the finishedSamples count and marks the working as idle. This three instructions can be packed in an rpc call (`upcxx::rpc_ff`). After that, the worker should check for incoming messages from master (call to `upcxx::progress`).

4. Master initializes a local queue that stores idle workers, respectively their `rankId`. If there are idle workers in the queue, master calls `getSample` and sends an rpc call to the first worker messaging the `sampleId`. In the rpc call the worker must also update its readiness state. Whilst waiting for idle workers in a while-loop, the master should check for incoming messages from the workers (call to `upcxx::progress`).

5. In step 3 each worker asks to increase the finishedSamples count. If the total number of samples (`NSAMPLES`) is reached, master notifies all workers (using `upcxx::rpc_ff`) about completion.

6. There were many submissions that were iteratively checking the workers in order, checking if they are available and then assigning the evaluation of the next sample. So they were not assigning the evaluation to the next available worker, but rather looping over the workers. In the context of this exercise, this method, although producing the same result, is suboptimal in terms of speed and scalability.

Note: Your submission(s) need to successfully run `checkResults()` to be considered correct.
Task 4: Parallel Korali (40 Points)

<Mandatory for CSE Students>
<Optional for Engineering Students>

By now you have become familiar with UPC++ and are ready to put your new skills to use in parallelizing Korali and obtain your well-deserved raise.

Use the principles you applied in Task 3 to implement the UPC++ conduit for Korali. First, analyze the implementation of the Single conduit in task1/conduits/single.cpp. Then, modify task1/conduits/upcxx.cpp to create Korali’s UPC++ conduit.

- Briefly mention the challenges you faced during this point and what UPC++ tools did you use to overcome them.

The parallelization of the Korali conduits makes use of UPC++. The proposed solution can be found in solution_codes/task1/conduits/upcxx.cpp. The upcxx implementation for the korali solver uses a master - slave (worker) approach, like in task3. The following challenges have to be overcome in order to solve this task:

1. Master needs to distribute the samples to the workers, start the upcxx korali solver and finalize processes on the worker ranks.
2. Work needs to be distributed. Each worker will execute processSample. This is done within the korali upcxx solver.
3. Workers have to return the results and notify the master, such that the master can updateEvaluation.

- The tasks for the master and the worker ranks are defined in functions master() and worker() respectively.
- Again, as in task3, we create a queue of workers and variables for the worker employed and the finished samples. The master rank creates the workers queue and fills it with the rank id’s of the remaining ranks, initializes the run of the Korali upcxx solver and, when finalized, the master notifies all workers of completion (with a upcxx::rpc_ff, setting the boolean variable continueEvaluations to false.
- In the processSample function, each worker is assigned a readiness state to start evaluating.
- Each worker, once it acquires a readiness state, evaluates its respective Sample array and notifies the master with a upcxx::rpc_ff, which in its turn calls _solver->updateEvaluation and marks the worker as idle.
• Run your new conduit on 24 cores of an Euler computing node. How much faster did your parallel version run, compared to the single-core version for Task 1? What can you say about the efficiency of your implementation? Did the CEO of the company give you a raise?

The parallelized version of the Korali engine with UPCXX results in a runtime of 39.4 s for Task 1, which is \( \approx 8 \) times faster than the serial version. The perfect parallelization of the korali solver would result to a speed up of 23 (out of 24 ranks, one is the master, only distributing work and not computing). This discrepancy is due to the fact that the korali CMAES engine runs only 8 samples per generation, according to our original settings. However, if we increase this number to e.g. \( > 100 \) we can achieve almost perfect parallelization and get the well-deserved raise.

• \textbf{<Optional>\textbf{}} Build a similar parallel engine using MPI. Work it out on top of the upcxx.cpp file, but provide the code as mpi_conduit.cpp. Answer: Did you take a similar approach? Was it easier or more difficult to implement? Justify your answers.

Note: Your submission(s) need(s) to achieve a similar result to that of \texttt{./single}, and at least 10x faster to be considered correct.