P. Koumoutsakos
ETH Zentrum, CLT E 13
CH-8092 Zürich

HW 4 - SSA, Optimization & UPC++

Issued: April 06, 2020
Due Date: April 27, 2020, 08:00am **(new)**

1-Week Milestone: Solve tasks 1 and 2

The skeleton codes for this homework are located in the gitlab repository: https://gitlab.ethz.ch/hpcse20/exercise.

Task 1: Stochastic Simulation Algorithm

The Stochastic Simulation Algorithm (SSA, also known as Gillespie algorithm) is a method that uses Monte-Carlo sampling to generate trajectories according to the chemical master equation.

Here, we consider the evolution of the quantity of N species $\{S_i\}_{i=1}^N$ within a well mixed volume which evolve according to M possible reactions:

$$\sum_{i=1}^{N} w_{i,j}^R S_i \xrightarrow{k_j} \sum_{i=1}^{N} w_{i,j}^P S_i, \tag{1}$$

where $j = 1 \dots M$, k_j are the reaction rates and $w_{i,j}^R, w_{i,j}^P$ are the stoichiometric coefficients for reactants and products.

For large S_i, this system would evolve according to the following ordinary differential equations:

$$\frac{ds_i}{dt} = \sum_{j=1}^{M} \left(w_{i,j}^P - w_{i,j}^R \right) r_j, \text{ with } r_j = k_j \prod_{i=1}^{N} s_i^{w_{i,j}^R}, \tag{2}$$

where $i = 1 \dots N$. The concentrations are defined as $s_i = S_i/\Omega$, where Ω is a scaling parameter dictating the number of particles in the system.

SSA solves the system by explicitly modelling the probability of collisions between reactants. It can therefore only be applied for reactions involving at most two molecules. Reactions involving more molecules can be modelled as sequences of binary reactions.

In the following description, we use $i \in [1, N]$ for the i-th species and $j \in [1, M]$ for the j-th reaction. Before computing the trajectory we need to set initial conditions S_i for our N species and set up reaction channels $RC_{i,j} = w_{i,j}^P - w_{i,j}^R$. We then evolve the species in time as follows:

Step 0: Evaluate propensities a_j and total propensity $a_0 = \sum_j a_j$.

Step 1: Draw $\tau \sim \mathrm{Exp}(a_0)$.

Step 2: Draw reaction $r \in [1, M]$ from a discrete random variable X with $\Pr(X = r) = a_r/a_0$.

Step 3: Advance the process by evaluating $t \to t + \tau$ and $S_i \to S_i + RC_{i,r}$.

Trajectory binning. A trajectory from SSA shows one possible realization of the time evolution of the system. Trajectory binning is a simple way to generate average trajectories over a number of sample trajectories. Since we do not have the same time steps for all samples, we need to average over time and over the samples. We assume that the simulation is run for times $t \in [0, t_{end}]$. We can then compute averaged quantities $\overline{S_{ik}}$ for realizations of S_i in the time interval $[k\delta t, (k+1)\delta t]$ with $k = 0 \dots \lceil t_{end}/\delta t \rceil - 1$. We can therefore set $k = \lfloor t/\delta t \rfloor$ and update $\overline{S_{ik}}$ for each time step of each sample trajectory.

For **Task 1** you have to implement the direct SSA algorithm for the following reaction system:

$$\begin{aligned} S_1 &\xrightarrow{k_1} \emptyset \\ S_2 &\xrightarrow{k_2} \emptyset \\ S_1 + S_2 &\xrightarrow{k_3} 2S_1 \\ \emptyset &\xrightarrow{k_4} S_2 \end{aligned} \tag{3}$$

which for large S_i would evolve according to the following ordinary differential equations:

$$\begin{aligned} \frac{ds_1}{dt} &= -k_1 s_1 + k_3 s_1 s_2, \\ \frac{ds_2}{dt} &= -k_2 s_2 - k_3 s_1 s_2 + k_4. \end{aligned} \tag{4}$$

We choose the rate constants $(k_1, k_2, k_3, k_4) = (1, 1, 1/50, 2)$ which leads to the propensities $(a_1, a_2, a_3, a_4) = (k_1 S_1, k_2 S_2, \frac{k_3}{\Omega} S_1 S_2, k_4 \Omega)$. The initial concentrations are defined to be $(S_1, S_2) = (4\Omega, 0)$.

For this settings the ODEs will reach a steady state solution at $(s_1, s_2) = (15, 5)$. A second steady solution exists at $(s_1, s_2) = (0, 20)$ and some trajectories in SSA might reach those values.

a) [5pts] In preparation for the following subquestion, derive the inverse transformation for the sampling
   - from an exponential distribution with rate τ, i.e. $p(x) = \tau e^{-\tau x}$,
   - and the sampling from a multinoulli distribution with 4 different outcomes $\Pr(X = r) = a_r/a_0$ for $r = \{1, 2, 3, 4\}$ and $\sum_{r=1}^{4} a_r/a_0 = 1.0$.

   Document your solution.

The CDF of the exponential distribution is given by

$$P_X(x) = P(X \leq x) = \int_0^x p(s) = \int_o^x \tau e^{-\tau s} dx = \begin{cases} 1 - e^{-\tau x}, & \text{if } x \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

for $\tau \geq 0$. The inverse of the CDF $P_X^{-1}(y) = \inf\{x | P_X(x) \geq y\}$ is given by

$$P_X^{-1}(y) = \frac{\ln(1-y)}{-\tau}$$

for $y \in [0, 1)$.

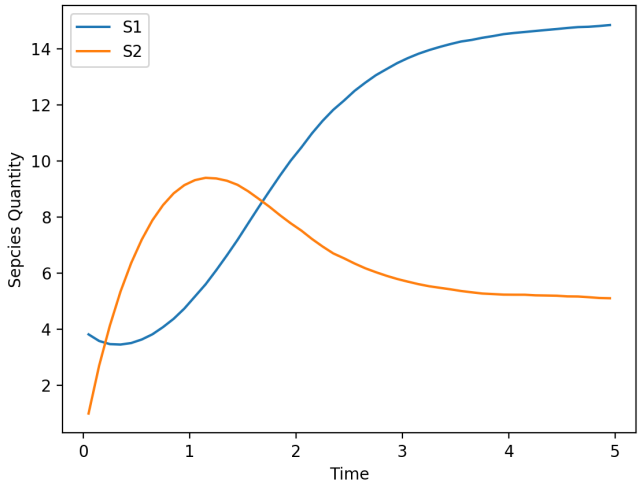

Figure 1: Evolution of Species S_1 and S_2.

The CDF of the multinoulli distribution is given by the non-continuous *step function*

$$P_X(r) = P(X \leq r) = \sum_{i=1}^{r} \Pr(X = i) = \sum_{i=1}^{r} a_i/a_0$$

for $r \in \{1, 2, 3, 4\}$. The inverse of the CDF $P_X^{-1}(u) = \inf\{r | P_X(r) \geq u\}$ is given by

$$P_X^{-1}(u) = \begin{cases} 1, & \text{if } 0 \leq u < \frac{a_1}{a_0} \\ 2, & \text{if } \frac{a_1}{a_0} \leq u < \frac{a_1+a_2}{a_0} \\ 3, & \text{if } \frac{a_1+a_2}{a_0} \leq u < \frac{a_1+a_2+a_3}{a_0} \\ 4, & \text{if } \frac{a_1+a_2+a_3}{a_0} \leq u \leq 1 \end{cases}$$

for $r \in \{1, 2, 3, 4\}$.

b) [20pts] Implement **Step 0**, **Step 1**, **Step 2** and **Step 3** from the SSA algorithm in the file `SSA_CPU.cpp`. In Step 1 and Step 2, use the inverse transform to draw samples from the corresponding probability density function. Please reuse the allocated random numbers from the `r48` array. Other files don't need to be modified.
In order to validate your implementation visualize your results `output.txt` with the python script `plot_output.py` and compare with Figure 1.

c) [5pts] (Optional) Estimate the number of FLOPs, the memory accesses (read and writes) and operational intensity of the algorithm. For this, Implement your estimates in the methods `getFlops` and `getTransfers`. Only consider floating point operations and memory accesses that happen in the while loop given by `while (bNotDone) { ... }`. Note that your result depends on the number of simulations stored in variable `numSamples`, number of bins stored in variable `nBins`, and the number of time steps per simulation, which can be read from the array `ntraj`. What performance in *GFLOP/S* do you achieve? Try different values for the environment variable #OMP_NUM_THREADS. Document your answer.

The FLOP count can be found in Performance measurements can be run with the bash script `performance.sh`. The measurements run with X `OMP_NUM_THREADS` will be dumped to the file `measurements/ssa_X.txt`.

Below you find measurements taken on Euler on a full node job with 36 cores run over interactive shell: `bsub -n 36 -W 04:00 -R fullnode -Is bash`

- 1 OpenMP Threads

  ```
  ================================================================================
  (05th perc.) time: 0.125629, 9.90495e+07 FLOP, Trf: 8.29996e+07 Byte, R: 1.19337 FLOP/Byte, perf: 0.765999 GFLOP/s
  (40th perc.) time: 0.126371, 9.91723e+07 FLOP, Trf: 8.30979e+07 Byte, R: 1.19344 FLOP/Byte, perf: 0.783122 GFLOP/s
  (60th perc.) time: 0.126754, 9.92168e+07 FLOP, Trf: 8.31334e+07 Byte, R: 1.19346 FLOP/Byte, perf: 0.784801 GFLOP/s
  (95th perc.) time: 0.129986, 9.9342e+07 FLOP, Trf: 8.32336e+07 Byte, R: 1.19353 FLOP/Byte, perf: 0.790043 GFLOP/s
  ================================================================================
  ```

- 2 OpenMP Threads

  ```
  ================================================================================
  (05th perc.) time: 0.074227, 9.90524e+07 FLOP, Trf: 8.30019e+07 Byte, R: 1.19337 FLOP/Byte, perf: 1.27302 GFLOP/s
  (40th perc.) time: 0.075908, 9.91614e+07 FLOP, Trf: 8.30891e+07 Byte, R: 1.19343 FLOP/Byte, perf: 1.2994 GFLOP/s
  (60th perc.) time: 0.076346, 9.91947e+07 FLOP, Trf: 8.31158e+07 Byte, R: 1.19345 FLOP/Byte, perf: 1.30761 GFLOP/s
  (95th perc.) time: 0.07807, 9.93423e+07 FLOP, Trf: 8.32339e+07 Byte, R: 1.19353 FLOP/Byte, perf: 1.33926 GFLOP/s
  ================================================================================
  ```

- 4 OpenMP Threads

  ```
  ================================================================================
  (05th perc.) time: 0.041324, 9.90449e+07 FLOP, Trf: 8.29959e+07 Byte, R: 1.19337 FLOP/Byte, perf: 2.15303 GFLOP/s
  (40th perc.) time: 0.042457, 9.91653e+07 FLOP, Trf: 8.30923e+07 Byte, R: 1.19344 FLOP/Byte, perf: 2.31699 GFLOP/s
  (60th perc.) time: 0.042862, 9.92068e+07 FLOP, Trf: 8.31255e+07 Byte, R: 1.19346 FLOP/Byte, perf: 2.33767 GFLOP/s
  (95th perc.) time: 0.046388, 9.93104e+07 FLOP, Trf: 8.32083e+07 Byte, R: 1.19352 FLOP/Byte, perf: 2.40524 GFLOP/s
  ================================================================================
  ```

- 8 OpenMP Threads

  ```
  ================================================================================
  (05th perc.) time: 0.021145, 9.90462e+07 FLOP, Trf: 8.29969e+07 Byte, R: 1.19337 FLOP/Byte, perf: 3.50709 GFLOP/s
  (40th perc.) time: 0.026241, 9.91526e+07 FLOP, Trf: 8.30821e+07 Byte, R: 1.19343 FLOP/Byte, perf: 3.7143 GFLOP/s
  (60th perc.) time: 0.026699, 9.9208e+07 FLOP, Trf: 8.31264e+07 Byte, R: 1.19346 FLOP/Byte, perf: 3.79016 GFLOP/s
  (95th perc.) time: 0.029294, 9.93471e+07 FLOP, Trf: 8.32377e+07 Byte, R: 1.19354 FLOP/Byte, perf: 4.69458 GFLOP/s
  ================================================================================
  ```

- 16 OpenMP Threads

  ```
  ================================================================================
  (05th perc.) time: 0.015998, 9.90594e+07 FLOP, Trf: 8.30076e+07 Byte, R: 1.19338 FLOP/Byte, perf: 5.37838 GFLOP/s
  (40th perc.) time: 0.016812, 9.91674e+07 FLOP, Trf: 8.30939e+07 Byte, R: 1.19344 FLOP/Byte, perf: 5.7016 GFLOP/s
  (60th perc.) time: 0.017415, 9.92237e+07 FLOP, Trf: 8.3139e+07 Byte, R: 1.19347 FLOP/Byte, perf: 5.91965 GFLOP/s
  (95th perc.) time: 0.018575, 9.93393e+07 FLOP, Trf: 8.32314e+07 Byte, R: 1.19353 FLOP/Byte, perf: 6.22829 GFLOP/s
  ================================================================================
  ```

- 32 OpenMP Threads

  ```
  ================================================================================
  (05th perc.) time: 0.010179, 9.90232e+07 FLOP, Trf: 8.29786e+07 Byte, R: 1.19336 FLOP/Byte, perf: 7.42368 GFLOP/s
  (40th perc.) time: 0.010909, 9.91639e+07 FLOP, Trf: 8.30911e+07 Byte, R: 1.19344 FLOP/Byte, perf: 8.60699 GFLOP/s
  (60th perc.) time: 0.011626, 9.92045e+07 FLOP, Trf: 8.31236e+07 Byte, R: 1.19346 FLOP/Byte, perf: 9.10047 GFLOP/s
  (95th perc.) time: 0.013715, 9.93455e+07 FLOP, Trf: 8.32364e+07 Byte, R: 1.19353 FLOP/Byte, perf: 9.76852 GFLOP/s
  ================================================================================
  ```

Note: This code will be reused in Homework 6 as a baseline for your CUDA implementation, please make sure that you don't fall behind. The maximal number of points in Task 1 is 25.

Task 2: Optimization with CMA-ES

In this exercise we will reuse the code from **Task 1**. We will use the CMA-ES solver from the *korali* framework to search the parameter set $(k_1, k_2, k_3, k_4) = (1, 1, 1/5, 20)$ that yields $(\overline{S_1}_{k_{MAX}}, \overline{S_2}_{k_{MAX}}) = (15, 5)$.

a) [0pts] Install the *korali* framework. Installation steps are documented here

- https://www.cse-lab.ethz.ch/korali/docs/using/install.html.

A documentation overview is available here

- https://www.cse-lab.ethz.ch/korali/docs/.

b) [20pts] Implement and maximize the objective function given in `objective.cpp` which returns the negative sum of squared errors (SSE), i.e.

$$f(k_1, k_2, k_3, k_4) = -(\overline{S_1}_{k_{MAX}} - 15.0)^2 - (\overline{S_2}_{k_{MAX}} - 5)^2. \tag{5}$$

Use the getter-methods `getS1` and `getS2` to read the terminal values $\overline{S}_{ik_{MAX}}$ from the `SSA_CPU` class. You can follow the `SSABenchmark.hpp` file on how to initialize and run the *SSA* simulations. Other files don't need to be modified. An exemplary *korali* optimization script and objective function can be found here:

- https://github.com/cselab/korali/blob/master/tutorials/advanced/4.running.cxx/run-cmaes-direct.cpp,
- https://github.com/cselab/korali/blob/master/tutorials/advanced/4.running.cxx/model/direct.hpp.

c) [5pts] Does CMA-ES converge to $(k_1, k_2, k_3, k_4) = (1, 1, 1/5, 20)$? Why (not)? Do we obtain a good fit? Also play with the parameter `Population Size` in the file `main_cmaes.cpp`. Document your findings. If you like to visualize optimization details, you can run the command `python -m korali.plotter`. This will read and visualize values from the `_korali_results` folder (must be present in the current directory).

Task 3: Parallel Tasking in UPC++

[20pts]

You are given a sequential program that approximates the value of π up to any desired accuracy. The approximation is based on a finite series, which is summing up factors according to the formula:

$$\pi \approx 4 \sum_{n=1}^{N} F(n). \tag{6}$$

The more factors N are included in the series, the lower the approximation error, i.e.

$$\lim_{N \to \infty} 4 \sum_{n=1}^{N} F(n) = \pi. \tag{7}$$

The formula $F(n)$ that specifies how the factors are computed, are given to you as a black-box function, to which you have no access. **You are not allowed to alter the formula, or the way the factors are computed in any way**.

Before continuing make sure that you use the following configuration for UPC:

```
$ export UPCXX_GASNET_CONDUIT=smp
$ export UPCXX_THREADMODE=seq
$ export UPCXX_CODEMODE=O3
```

and you have loaded the necessary compiler:

```
module load new gcc/6.3.0
```

Throughout this exercise, the UPCXX tutorial provided in the exercise class as well as the User guide can be very helpful.

a) [1pts] The provided sequential program computes an approximation $\tilde{\pi}$ based on the first $N = 240 \times 10^3$ factors of the series, and then writes the result to two text files in the folder `./Results`. **You have to create the folder before running the scripts, else the results will not be saved.** The program also reports the absolute approximation error $\epsilon = |\pi - \tilde{pi}|$. Run the program and report the absolute error and the speed. Make sure that the textfiles are saved in the results folder after the execution.

The total runtime of the program is approximately $T = 9.13s$ (seconds) and the absoltue error achieved is approximately $\epsilon = 4.16 \times 10^{-6}$.

b) [8pts] In order to parallelize the execution of the sequential program, we follow a divide-and-conquer strategy using UPC++, distributing the factors among ranks and having each rank evaluate it's set of factors in parallel. Build your parallel version of the sequential program by filling the `TODO:` comments on the file `divideAndConquer.cpp` and run it using 24 ranks on an Euler compute node. Request an allocation (in an interactive mode) using:

```
bsub -W 24:00 -n 24 -R fullnode -Is bash
```

For simplicity, assume that the number of ranks divides exactly the number of factors the program computes. Take into account the following steps:

- Initialize `UPCXX`.

- Assuming that the number of ranks is N_{ranks}, each rank needs to compute N/N_{ranks} factors.
- After computing each factor, each ranks needs to save the computed factor on 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 `upcxx::global_ptr<double> factorArray` in the skeleton 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 (see next item), 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 global factor array needs to be initialized by rank zero (`upcxx::new_array<double>`), and broadcasted to all other ranks.
- After broadcasting the array, each rank needs to compute the portion of the factors it is assigned, and then **place** the result back to the `factorArray`. Do not use RPCs in this question, use the `upcxx::rput` command.
- The ranks need to evaluate their factors 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 (composing many futures) in `upcxx`, where we are gathering all futures together and wait for their completion only at the end.
- After the aforementioned step, the time of each rank computation is tracked.
- Finally, rank zero needs to compute the approximate value $\tilde{\pi}$ and save it to the results file, along with the total time. **Downcast** the global pointer to a local one and use it to compute the final approximation. Compute also the absolute error.
- Did you waited for the futures to be completed before rank zero is summing up the series ? Do you need wait and synchronize the ranks ?
- Finalize `UPCXX`.

Run the divide and conquer program with $N_{nodes} = 24$ nodes using the command:

```
upcxx-run -n 24 ./divideAndConquer
```

and report the total runtime as well as the absolute error $\epsilon = |\pi - \tilde{pi}|$. Did you manage to accelerate the sequential program? If yes, what is the speed-up S you achieved? Report the optimal speed-up S^{opt} and the speed-up efficiency S/S^{opt}.

The output at the console is

```
upcxx-run -n 24 ./divideAndConquer
Approximating the value of PI with 240000 series
  coefficients.
PI approximate: 3.1415884869231183
PI: 3.1415926536
Absolute error: 4.1666666747985914e-06
Total Running Time: 1.037551508s
```

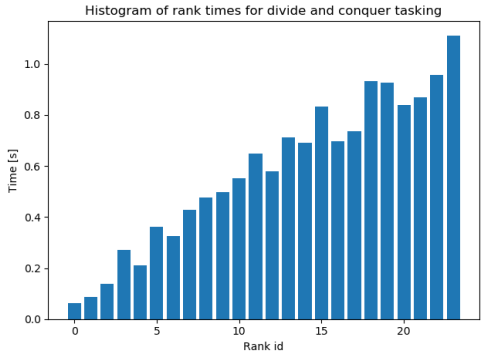

Figure 2: The ranktimes for the divide and conquer strategy

The total error is the same as the sequential version as expected, since we are using the same number of factors to compute the approximation. The total runtime however is much smaller $T = 1.0s$. The associated speed-up is $S = T^{seq}/T^{d-c} = 9.13/1.0 \approx 9.13$. The speed-up efficiency is $9.13/24 \approx 38\%$

c) [1pts] Use the provided script `plot.py` to plot the rank runtimes of the parallel program. You can either copy the results folder locally and plot the figures locally, or load the python module in Euler with:

```
module load python
```

then run the script, and copy the figures to your local machine. The python script, receives two inputs, the number of ranks used by the program and the strategy. It outputs a plot with the runtime of each rank. The program, processes the result files from the folder `./Results` and saves the plot figure in the folder `./Figures`. To run the python script use:

```
python3 plot.py --NUM_RANKS 24 --strategy
  divide_and_conquer
```

What do you observe? Is the load balanced? Calculate the load imbalance ratio. (you can use the python script to calculate the imbalance ratio)

The runtimes of each rank are plotted in Figure Figure 2. We observe that the load assigned to each rank is very imbalanced in the divide and conquer strategy. This implies that the computational cost of evaluating each factor is not equal, i.e. there is high variability in the computational cost of evaluating the factors. 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. In this way, we sacrifice one rank for communication, but we expect to gain in speed by exploiting consumers/workers that would otherwise be idle.

d) [8pts] In the following you decide to alter the parallelization strategy and try a producer consumer tasking strategy. Implement your program by filling the `TODO:` comments in the file `producerConsumer.cpp`. The following steps may assist you in your implementation.

- Initialize `UPCXX`.

- Similarly to the divide and conquer strategy, the master needs to initialize the `factorArray` array with the `upcxx::new_array<double>` command. Then the array needs to be broadcasted so that all ranks have access and point to the shared address space. Note that different to the skeleton code provided in the divide and conquer strategy, the global pointer is already defined in line 24.
- Then, the master rank is calling `master(rankCount)`, while every other rank (consumer rank) k is calling `worker(k)`.
- The master is initializing a queue with all the available workers. This queue is filled all the time **only** with the workers that are available to compute (not the busy ones, that are already computing a factor).
- Until all factors are ready, the master is waiting for any worker to be available. Whenever any worker is available, the master has to get his `workerId`, by popping out the first `workerId` from the queue of the available workers.
- After identifying an available worker, the master needs to send a fire and forget RPC `upcxx::rpc_ff`, the task that the consumer has to complete. The task the master is assigning to the next available worker is of course the evaluation of the next factor.
- Check that the factors are evaluated in reverse order. As we observed from the previous task that the most computationally heavy tasks are the ones with high k assigned to the bigger ranks in the consumer-producer strategy, so we prefer to evaluate them first.
- When all factor evaluations are finished, the master needs to use a RPC to notify the consumers that all evaluations are finished by setting the boolean variable `continueFactorEvaluations` to `False`.
- Complete also the `workerComputeFactor` function. This is the function sent from the master with a RPC to the worker with id `rankId`. The worker, has to compute the associated factor, then use a RPC to update the master's `factorArray` with the computed factor, and notify the master that the consumer is available again to compute by pushing your rankId to the queue.
- Finally, when all factors are evaluated, the master rank needs to compute the approximate value $\tilde{\pi}$ and save it to the results file, along with the total time. **Downcast** the global pointer to a local one and use it to compute the final approximation. (Similarly to the divide and conquer strategy)

Report the total runtime and the associated absolute approximation error. Compute the achieved speed-up and the parallel efficiency. In your computation for parallel efficiency take into account only the ranks devoted for computation and not for communicating the tasks.

The solution is provided in the solution codes. 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 factors to any available worker (consumer). One way to do this, is by creating a FIFO queue with all idle workers. When a worker is idle (not currently evaluating any factor), the master pops the worker out of the queue and assigns an evaluation to the worker. The worker is evaluating the factor. 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 factor to the first available one till all factors are evaluated. Finally, when all factors 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 `upcxx::progress()` command, to allow incoming RPCs to be processed. The output of the code is

```
upcxx-run -n 24 ./producerConsumer
Approximating the value of PI with 240000 series
  coefficients.
PI approximate: 3.1415884869231183
PI: 3.1415926536
Absolute error: 4.1666666747985914e-06
Total Running Time: 0.64935494699999996s
```

The absolute error is the same. The speed-up is $S = 9.13/0.6493 = 14.06$. The theoretical maximum in this case is 23, since the master rank is reserved for managing the communication of the tasks. The parallel efficiency is $14.06/23 = 61\%$.

The following important points need to be noted in the solution:

- Throughout the code, use fire and forget RPCs `upcxx::rpc_ff`. 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 factors to the workers. The master does not need any information returned to him. When all factors are assigned, the master again sends an `rpc_ff`, changing the value of the boolean variable `continueFactorEvaluations` 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 `workerComputeFactor` 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 factor, and calling a `upcxx::rpc_ff` to the master, passing its rank, the factor 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.

e) [2pts] Now, plot the ranktimes of the producer-consumer strategy with the python program. What do you observe? Is the load balanced? Calculate the load imbalance ratio in this case. What is your conclusion, is this producer/consumer strategy preferable over the divide and conquer strategy in this specific example ? When is in general the producer consumer preferable over the divide and conquer strategy? Document your answers.

The runtimes of each rank are plotted in Figure Figure 3. The producer-consumer strategy alleviates the load imbalance problem, and we observe an almost perfect balancing. The total runtime is smaller, so this strategy is preferable in this example. 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 factors), especially as the number of factors that need

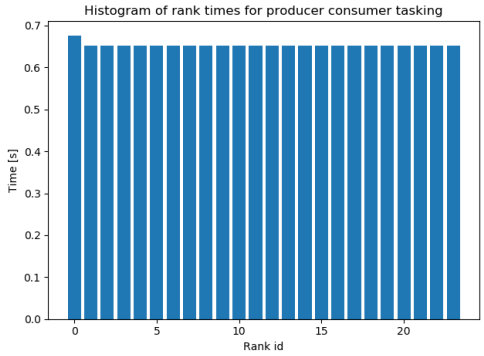

Figure 3: The ranktimes for the producer consumer strategy

to be evaluated increases. (If the number of factors evaluated are increased, the speed-up advantage of the consumer-producer strategy over the divide and conquer method is indeed more evident.)

Guidelines for reports submissions:

- Archive your written pdf and source code (e.g.: .tar, .zip) and submit it via Moodle until April 27, 2020, 08:00am