High Performance Computing for Science and Engineering II.

Exercise Set 4

Florian Mahlknecht

2020-04-25

Please note that this report does not contain any source code, see files attached for the implementation details.

1 Stochastic Simulation Algorithm

1.1 Inverse transformation of cumulative distribution function

Exponential distribution Given the following probability distribution

$$p(x) = \tau\, e^{-\tau x} \qquad\qquad x \geq 0 \tag{1}$$

we may calculate its cumulative distribution function:

$$F_x(x) = \int_0^x \tau\, e^{-\tau y} dy = -e^{-\tau y}\Big|_0^x = 1 - e^{-\tau x}$$

Using a random variable u uniformly distributed in $[0,1]$ we can sample x by solving:

$$\begin{aligned} u &= 1 - e^{-\tau x} \\ \log(1-u) &= -\tau x \\ \Rightarrow \quad x &= -\frac{\log(1-u)}{\tau} \end{aligned}$$

Categorical distribution Considering a categorical distribution with 4 possible outcomes we have:

$$P(X = r) = \frac{a_r}{a_0} \qquad\qquad a_0 = \sum_{i=1}^{4} a_i \tag{2}$$

In this case the cumulative distribution function reads:

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

Using the uniformly distributed variable $u \in [0,1]$ again, we get:

$$u = \sum_{i=1}^{r} \frac{a_i}{a_0}$$
$$u\,a_0 = \sum_{i=1}^{r} a_i$$
$$s_r := \sum_{i=1}^{r} a_i$$
$$\Rightarrow r(u) = \begin{cases} 1 & \text{for} \quad u\,a_0 \le s_1 \\ 2 & \text{for} \quad s_1 < u\,a_0 \le s_2 \\ 3 & \text{for} \quad s_2 < u\,a_0 \le s_3 \\ 4 & \text{for} \quad u\,a_0 > s_3 \end{cases}$$

1.2 Sample output

Figure 1 shows an exemplary output of the implementation, which matches the expected figure.

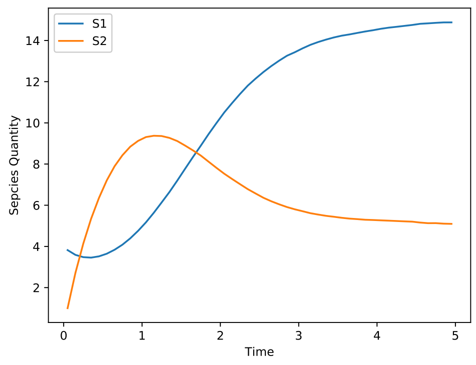


Figure 1: Sample output

1.3 Performance evaluation

Number of threads	40% - Quantil	60% - Quantil
1	0.609	0.610
2	1.161	1.172
4	2.081	2.088
8	3.722	3.974
16	3.233	3.409

Table 1: Scaling performance [GFLOP/s]

As table 1 shows, after 8 threads there is no gain in performance anymore. The flop per byte ratio remains as expected constant throughout the experiment; its value is around 1.25 Flop/Byte.

2 CMA-ES Parameter Search

It is interesting to see that the CMA-ES parameter search does not converge to the expected values. Table 2 illustrates the results for different population size parameters.

	p = 2	p = 4	p = 8	p = 16	p = 32	⋆
k_1	3.331	2.641	2.305	2.619	2.796	1.0
k_2	0.019	1.826	2.832	0.193	0.936	1.0
k_3	2.560	0.535	0.468	0.529	0.566	0.2
k_4	48.931	48.752	48.546	40.079	46.448	20
S_1	14.702	15.072	15.019	**15.002**	15.038	14.837
S_2	1.341	**4.999**	4.990	4.996	4.985	5.133
SSE	13.477	$5.180 \cdot 10^{-3}$	$0.451 \cdot 10^{-3}$	$\mathbf{0.018 \cdot 10^{-3}}$	$1.697 \cdot 10^{-3}$	$44.258 \cdot 10^{-3}$

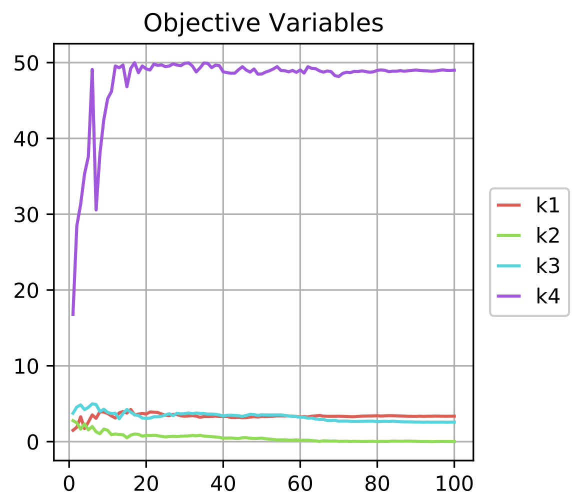
Table 2: Results with varying population size

Except the population size of only 2, the found parameters by CMA-ES perform all better than the expected theoretic onces. This underlines how theoretically expected outcome for a large numbers of reaction cannot exactly be observed in a random experiment with a small amount of reactions. Indeed our theoretical parameters yield an outcome which is off by approximately 0.15.

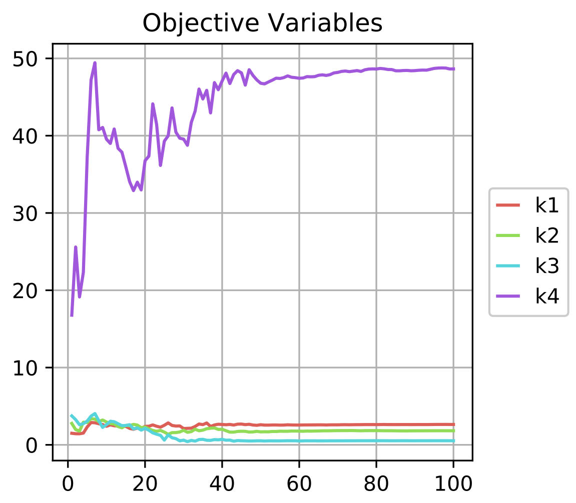
The evaluation was performed by a small test program, i.e. `ssa_test`, which runs the simulation several time with parameters specified as command line arguments and averages the result. The given S_1 and S_2 in table 1 values are in this way an average over 10 test runs. The sum of sqaured errors is calculated on those values. This is important to note, as the random nature of our objective function, allows the CMA-ES algorithm "to get lucky", and get a lower SSE in a specific evaluation step. This is why the calculated value on averaged S_1, S_2 is different from the $F(x)$ output of the korali framework.

The best result in this case is obtained with a population size of 16. The outcomes are just off by $\mathcal{O}(10^{-3})$ from the desired S_1 and S_2. A population size of 32, besides of not yielding better results, takes significantly longer to compute. Note however, that the SSE found by the Korali framework for the best sample is in the same order of magnitude as the one for population size 16, however on the averaged values, $p = 16$ is clearly better.

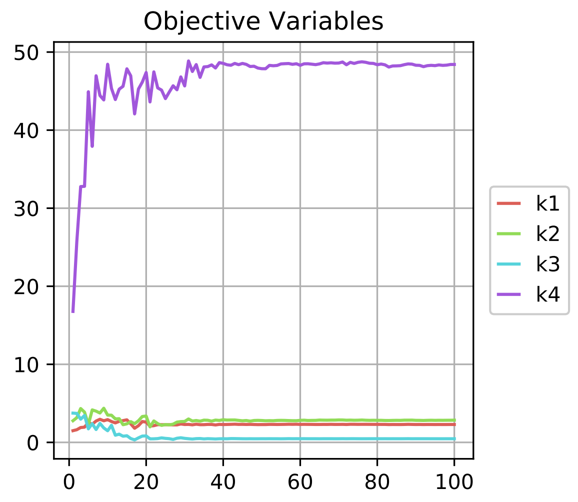
Figure 2 shows how the objective variables converged for the various experiments.



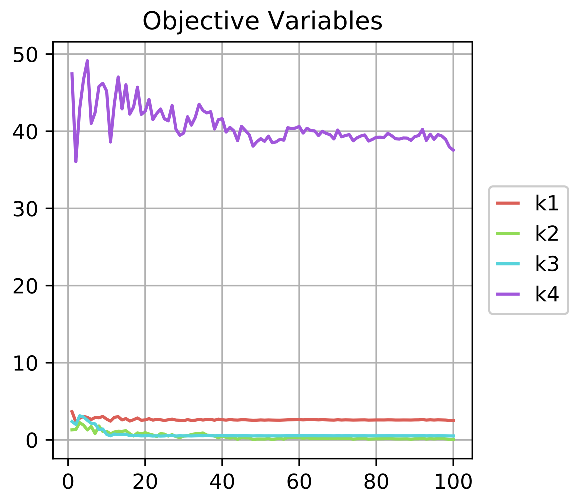
(a) Population size 2



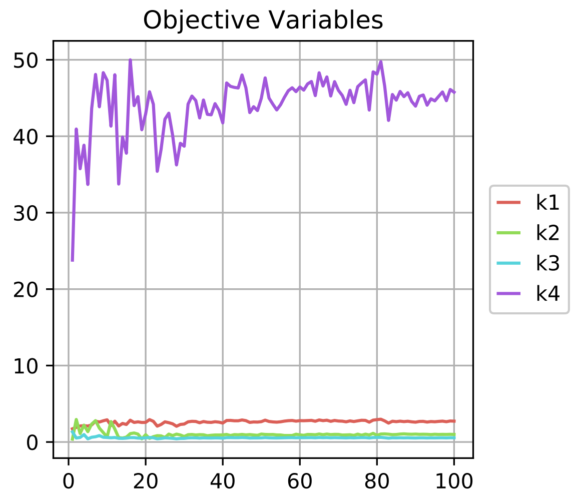
(b) Population size 4



(c) Population size 8



(d) Population size 16



(e) Population size 32

Figure 2: Population size behavior

3 Parallelization with UPC

The sequential approach takes about 9 s on Euler, yielding the following output:

```
./sequential
Approximating the value of PI with 240000 series coefficients.
PI approximate: 3.1415884869231183
PI: 3.1415926535897931
Absolute error: 4.1666666747985914e-06
Total Running Time: 9.1781152220000006 s
```

3.1 Divide et impera

After parallelizing with a *divide et impera* strategy, we get the following output:

```
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: 1.1779726230000001s
```

As expected the absolute error yields the same, which proofs that the execution is equivalent with respect to the sequential approach. From absolute running time numbers (without averaging over more execution times) we can make a rough estimate of the speedup:

$$S = \frac{T_{seq}}{T_{parallel}} \approx 7.79 \tag{3}$$

Given that we have provided 24 times as much resources, the speed up efficiency is rather poor:

$$\eta = \frac{S}{S_{opt}} = 32\,\% \tag{4}$$

3.1.1 Load Imbalance

As fig. 3 shows, the load is not divided equally among the ranks. In particular, the coefficients for larger k take longer to compute. Since we simply divide the computation in linear partitions the difference in computation time between the first and the 24th rank is quite considerable.

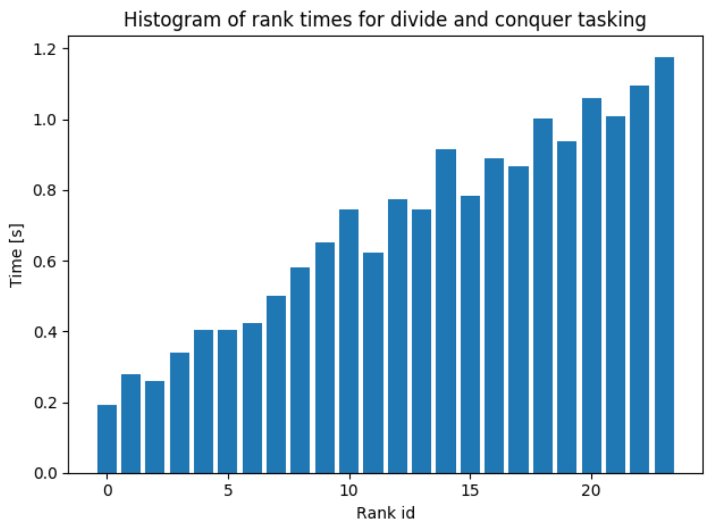


Figure 3: Rank times divide and conquer

Indeed, the load imbalance ratio is significantly high (see modified python script):

$$R = \frac{L_{max} - \bar{L}}{\bar{L}} = 0.70 \tag{5}$$

3.2 Producer Consumer

The second strategy yields the following results:

```
upcxx-run -n 24 ./producerConsumer
Approximating the value of PI with 240000 series coefficients.
PI approximate: 3.1415884869231183
PI: 3.1415926536
Absolute error: 4.1666667479859 14e-06
Total Running Time: 0.8413565499999998s
```

Again the same exact absolute error and approximate PI outcome shows that the code execution is equivalent to the the former approaches.

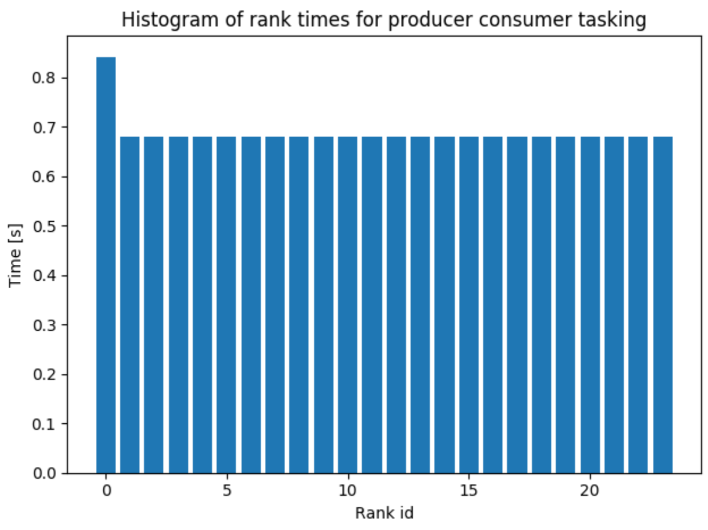


Figure 4: Rank times producer consumer

Figure 4 now effectively illustrates, that the load is much better (almost perfectly) distributed. The actual compute ranks have a computation time of 0.679 s. The speedup, considering the total running time, is:

$$S = \frac{T_{seq}}{T_{parallel}} \approx 10.91 \tag{6}$$

Considering only the ranks that actually carry out the computation, we get a speedup of $S = 13.52$, which yields a considerably better efficiency:

$$\eta = \frac{S}{S_{opt}} = 56\,\% \tag{7}$$

3.2.1 Load imbalance

The load imbalance ratio, as fig. 4 already suggests, is considerably better:

$$R = \frac{L_{max} - \bar{L}}{\bar{L}} = 0.23 \tag{8}$$

3.3 Comparison

Figure 5 shows a final comparison of the scaling performance of the total running times averaged over 5 consecutive executions. It is interesting to note, that with two ranks, the consumer-producer strategy actually performs worse than the sequential approach. This can simply be explained, that with just two total ranks, there is just one computational node, while the other node just communicates with it. This is clearly worse than having just one process alone performing the computation.

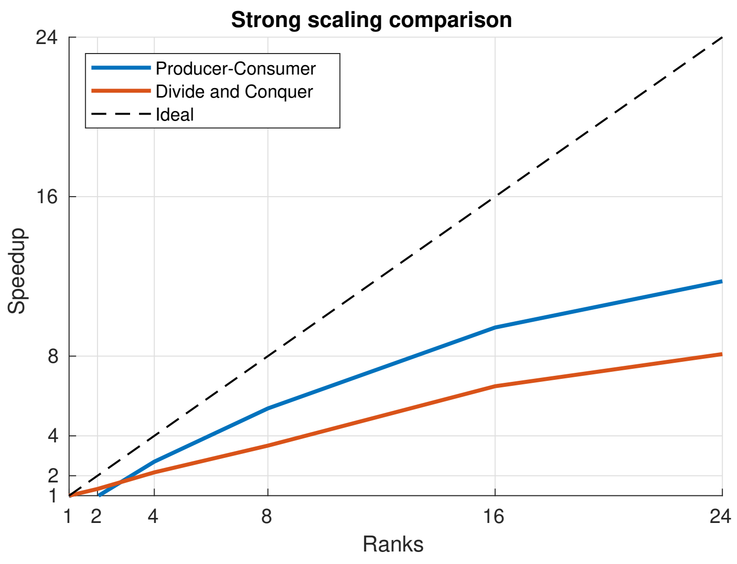


Figure 5: Speedup comparison

3.4 Discussion

As we have seen in this example, the producer-consumer strategy can have advantages when the tasks are not balanced. Considering again figs. 3 and 4 we can appreciate how well this approaches balances out the load. However, this approaches requires that the task can be split up in enough junks, such that the overall load can be divided evenly, but still large enough tasks that communication keeps being worth it. Furthermore, some knowledge about the execution time is preferable, to tackle the big tasks first, avoiding large tasks degrading load balance at the end.

If the tasks are known to be equally heavy on the other side, the divide et impera approach might be better suited, since it shows a much lower communication overhead.