ETH zürich

High Performance Computing for Science and Engineering for CSE

Spring Semester 2020

P. Koumoutsakos
ETH Zentrum, CLT F 12
CH-8092 Zürich

Mock Exam

Issued: May 25, 2020, 11:15
Hand in: May 25, 2020, 12:00

Last Name:	
First Name:	
Student ID:	
Computer Hostname:	

With your signature you confirm that you:

- have read the exam directives,
- solved the exam on your own and without any external help,
- wrote your answers following the exam directives.

Signature: _____

Grading table

Question	Maximum score	Score	TA 1	TA 2
Question 1	20			
Question 2	20			
Question 3	20			
Question 4	20			

You can solve the questions in *any* order.

NOTE: your final grade will be determined from the grade of *this exam* AND from your homework bonus (when applicable).

Question 1: Conjugacy and Laplace Approximation (20 points)

The Poisson distribution expresses the probability of a given number of events $X \in \mathbb{N}_0$ occurring in a fixed interval of time or space if these events occur with a known constant mean "rate" $\lambda > 0$ and independently of the time. The probability density function of the Poisson distribution is given by

$$P(X = x) = p_\lambda(x) = \frac{\lambda^x e^{-\lambda}}{x!}$$

a) (5 points) Given N observations $\mathbb{X} = (X_1, ..., X_N)$. Assume that the observations are independent and identically distributed $X_i \sim p_\lambda(x)$. Show that the maximum likelihood estimate (MLE) $\hat{\lambda}$ of the parameter λ is given by the sample mean $\bar{X} = \frac{1}{N}\sum_{i=1}^{N} X_i$. Start by writing down the likelihood function $\mathcal{L}(\lambda|\mathbb{X})$.

b) (10 points) A prior is called *conjugate prior with respect to a likelihood function* if the prior and the posterior belong to the same probability distribution family. Show that if the prior of λ follows a Gamma distribution, then the posterior distribution $p(\lambda|\mathbb{X})$ is a Gamma distribution too. In other words, the Gamma distribution is a conjugate prior for the Poisson probability distribution. Also specify the parameter of the resulting Gamma distribution.

c) (5 points)
Approximate the posterior distribution $p(\lambda|\mathbb{X})$ of the previous task using the Laplace approximation method. You can omit time-consuming algebraic manipulations that involve second derivatives.

- The density function of the Gamma distribution with parameters α and β is:

$$f(x; \alpha, \beta) = \frac{\beta^\alpha x^{\alpha-1} e^{-\beta x}}{\Gamma(\alpha)}\,, \quad \alpha, \beta > 0\,,$$

- Where Γ is the Gamma function defined as:

$$\Gamma(z) = \int_0^\infty x^{z-1} e^{-x} \mathrm{d}x\,,$$

Question 2: Domain decomposition & MPI Datatypes (20 points)

In mathematical analysis, a space-filling curve is a one-dimensional curve that passes through every point of the unit square. Space-filling curves can thought of as the limit of a sequence of piecewise linear curves; an example of such a sequence is shown in fig. 1, for a particular space-filling curve called the Hilbert curve.

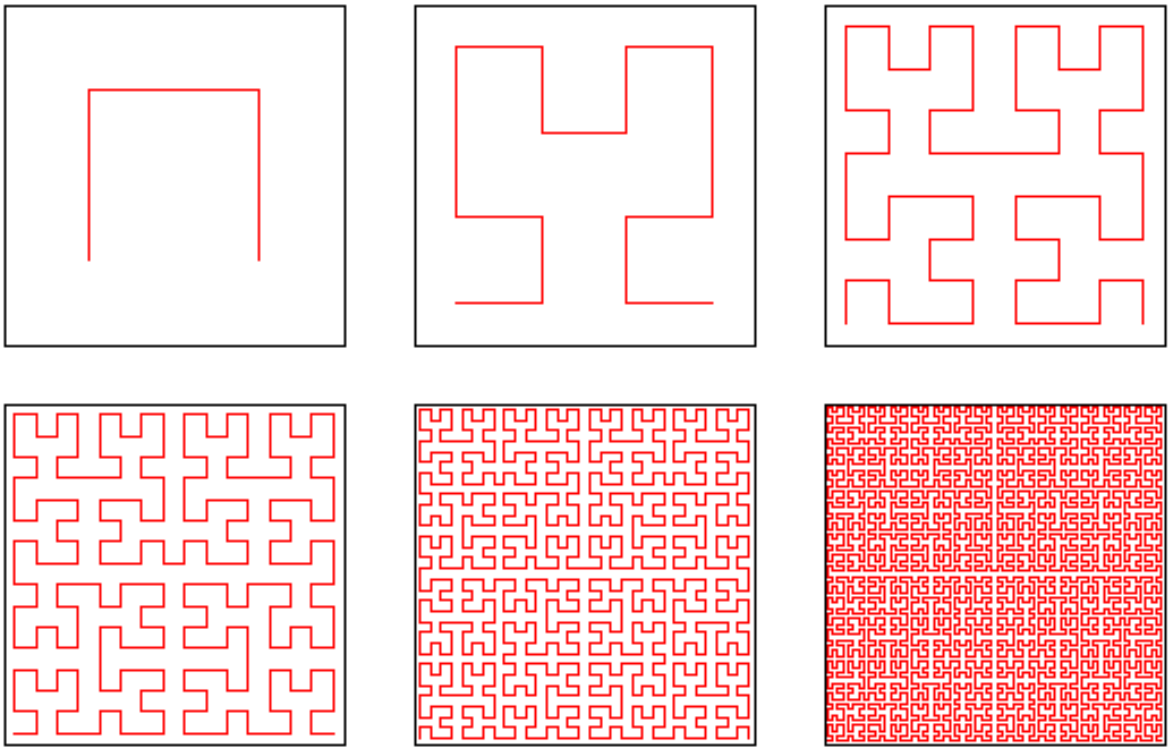

Figure 1: The first six iterations of the Hilbert curve.

Each member of the sequence can be used to assign a number to each point of a two-dimensional grid, as shown in fig. 2, for a 2×2 and a 4×4 grid. An important property of space-filling curves is that they preserve locality. This means that, when they are used to map a two-dimensional grid to a one-dimensional sequence of numbers, numbers that do not differ much correspond to grid points that are close as well (for example, numbers 4 and 7 correspond to two adjacent grid points in fig. 2).

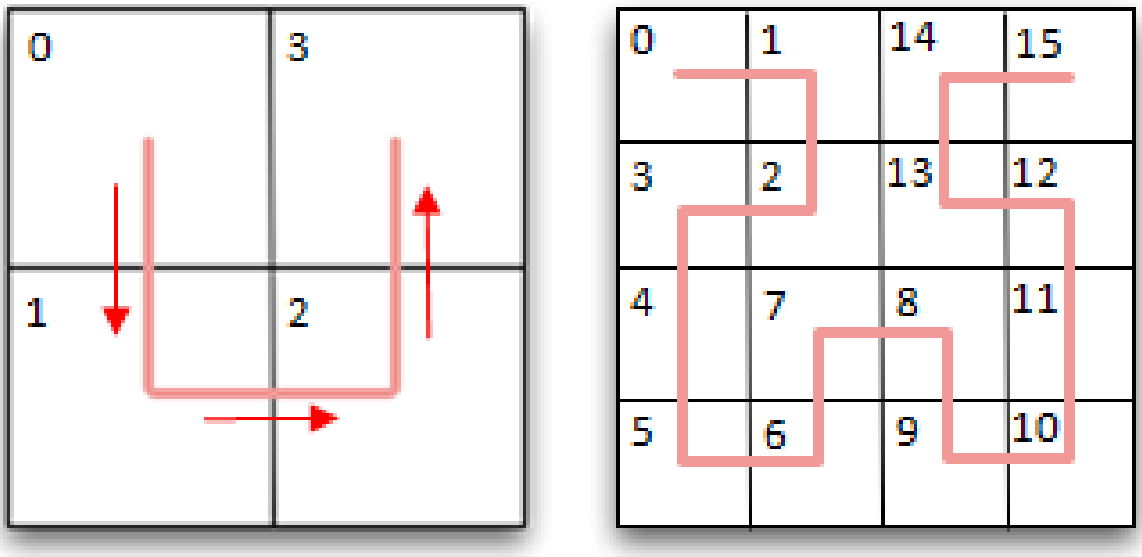

Figure 2: Mapping of a two-dimensional grid to a one-dimensional curve by using the Hilbert curve.

This locality preservation property makes space-filling curves suitable for domain decomposition methods in parallel programming. Referring again to fig. 2, a 4×4 grid (used to solve an equation numerically) could be divided among three parallel processes as follows:

process 0: points $0, 1, 2, 3, 4, 5$ (volume : 6, surface: 12)
process 1: points $6, 7, 8, 9, 10$ (volume : 5, surface: 10)
process 2: points $11, 12, 13, 14, 15$ (volume : 5, surface: 10)

Such decompositions make sure that grid points assigned to every process are clustered together, thus minimizing the surface-to-volume ratio of each process, which in turn reduces the cost of communication. Moreover, they allow for a systematic way to decompose domains, given an arbitrary number of parallel processes.

a) (5 points) Assume an 8×8 grid. Its points are mapped to one-dimensional indices $0, 1, \ldots, 63$ using the Hilbert curve in the top right part of fig. 2. How would this grid be distributed to four parallel processes, i.e. which grid points would each process be assigned? Would such a decomposition be ideal in terms of surface-to-volume ratio of each process (recall that this ratio needs to be minimized, in reduce to save communication costs)?

b) (10 points) Assume that the same 8×8 grid is distributed to three processes. What is the surface-to-volume ratio of each process and which points will each process have?

c) (5 points) Use a custom MPI Datatype that allows a process to send the diagonal of a two-dimensional matrix to another process.

```
1  int main(int argc, char **argv)
2  {
3    int rank;
4    MPI_Init(&argc, &argv);
5    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
6
7    int N = 10;
8    double *A = new double[N * N];
9
10   MPI_Datatype diagonal;
11   // Create the datatype here
12   // ...
13
14   if (rank == 0) {
15     for (int i = 0; i < N * N; ++i)
16       A[i] = SomeFunctionOf(i)
17
18     MPI_Send(A, 1, diagonal, 1, 0, MPI_COMM_WORLD);
19   } else if (rank == 1) {
20     MPI_Recv(A, 1, diagonal, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
21   }
22   MPI_Type_free(&diagonal);
23   MPI_Finalize();
24   delete []A;
25   return 0;
26 }
```

You are given the following list for MPI Datatypes related APIs:

```
MPI_Type_commit(MPI_Datatype *datatype)
MPI_Type_free(MPI_Datatype *datatype)
MPI_Type_contiguous(int count, MPI_Datatype oldtype,
                    MPI_Datatype *newtype)
MPI_Type_vector(int count, int blocklength, int stride,
                MPI_Datatype oldtype, MPI_Datatype *newtype)
MPI_Type_create_struct(int count, int blocklengths[], MPI_Aint offsets[],
                       MPI_Datatype types[], MPI_Datatype *newtype)
MPI_Get_address(void *location, MPI_Aint *address)
MPI_Type_create_resized(MPI_Datatype oldtype, MPI_Aint lb,
                        MPI_Aint extent, MPI_Datatype *newtype)
MPI_Type_get_extent(MPI_Datatype datatype, MPI_Aint *lb,
                    MPI_Aint *extent)
```

Question 3: Parallel tasking theory (20 points)

You work at a newly instituted super-computing center and every morning you receive a queue of 16 jobs that need to be executed. Each job can be executed **independently** and cannot be parallelized. The computational cost of each job is given in the following in terms of a reference computational load C, but you do not know this load a priori to design an optimal parallel tasking strategy. **Communication costs are ignored in this exercise.**

Job order:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Comp. load:	8	1	1	10	1	1	1	8	1	8	8	1	1	1	1	1

We assume for simplicity that this load can be directly translated to a fixed reference computational time, e.g. that a job of load $C = 7$ can be executed by a single rank in $T = 7$ time units. Time measurements in the following need to be provided in terms of these time units. You are provided with a small compute node with **4 cores**, each capable of running a single job at a time. Your task is to propose algorithms that can take advantage of different parallel tasking strategies.

a) (4 points) In the classical divide-and-conquer method the tasks are divided to the available processors. You cannot affect the ordering of the jobs. This leads to the following partition:

Job order:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Comp. load:	8	1	1	10	1	1	1	8	1	8	8	1	1	1	1	1
Exec. by rank:	1	1	1	1	2	2	2	2	3	3	3	3	4	4	4	4

Compute:

1. the total runtime T_{total}
2. the average rank runtime T_{avg}
3. the load imbalance ratio $I = \frac{T_{max} - T_{avg}}{T_{avg}}$
4. the average time that a rank remains idle W_{avg}.

b) (6 points) Now you are asked to apply the producer-consumer strategy. Note that you cannot affect the ordering of the jobs assigned to the ranks. Which rank will execute each job? Please fill in the following array and answer the questions that follow:

Job order:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Comp. load:	8	1	1	10	1	1	1	8	1	8	8	1	1	1	1	1
Exec. by rank:																

1. What is the total runtime T_{total} in this case?
2. What is the average rank runtime T_{avg}?
3. What is the load imbalance ratio I?
4. What is the average rank idle time W_{avg}?
5. What is the disadvantage of the producer-consumer strategy in general?
6. Is the extra effort to implement this strategy worth it in this case?

c) (10 points) The problem with the jobs is that the computational load is not balanced and we do not know the load a priori to design an optimal tasking strategy. A different ordering of **the same** jobs might cause the **worst case** scenario (in terms of runtime) for a **divide-and-conquer** tasking strategy.

1. What is an **example** of an ordering that causes this worst case scenario?
2. To which rank is each job assigned in your example in the **divide-and-conquer** tasking strategy?
3. To which rank is each job assigned in your example in the **producer-consumer** tasking strategy?
4. What are the total runtimes for both tasking strategies in this case?
5. How much faster is the producer-consumer strategy in this case? Report the speed-up compared to the divide-and-conquer method.

Question 4: Prefix sum with CUDA (20 points)

Prefix sum is a useful building block in designing many parallel algorithms. Given an array of N elements

$$[a_0, a_1, a_2, \ldots, a_{N-1}],$$

the *inclusive prefix sum* (also called *inscan*) produces an array

$$[a_0, a_0 + a_1, a_0 + a_1 + a_2, \ldots, a_0 + a_1 + \cdots + a_{N-1}].$$

One way to perform the inclusive prefix sum is shown in the Figure 3.

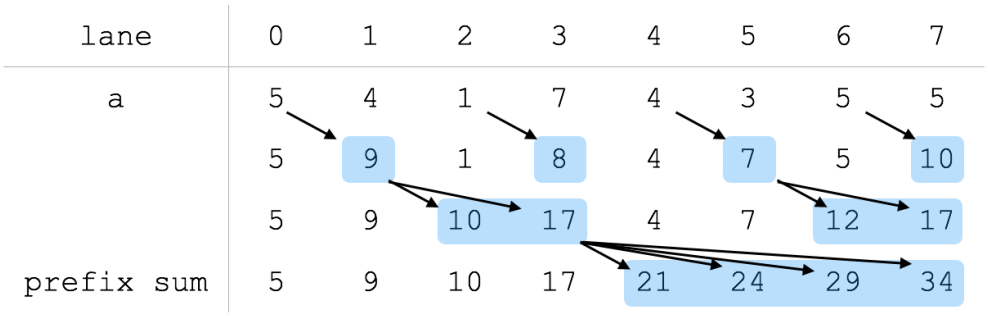

Figure 3: Inclusive prefix sum algorithm for $N = 8$.

Your task is to implement the inclusive prefix sum operation for various different orders of magnitude of N.

a) (8 points) Implement a function `warpPrefixSum` that computes the prefix sum on a level of a warp. The function should return the sum of values a passed to lower lanes of the warp.

```
__device__ double warpPrefixSum(double a) {
  // ...


  return ...;
}
```

Note: You may assume that the warp size is 32, and that the function will be active on all 32 lanes.

b) (7 points) Implement a function `blockPrefixSum` that computes the prefix sum on a level of a block. You may use any approach whose complexity is proportional to $\mathcal{O}(\log N)$.

```
__device__ double blockPrefixSum(double a) {
  // ...




  return ...;
}
```

Note: Assume that the block size is 1024 and that the function is active on all 1024 threads. You may utilize `warpPrefixSum` to simplify the implementation (even if you haven't solved the previous subquestion).

c) (5 points) Describe an algorithm for performing the inclusive prefix scan operation for $N \leq 10^6$. How many kernel invocations does your algorithm do? What does each of the kernels compute?

Note: This question has many possible answers.

Listed are prototypes of warp-level primitives that might be useful:

```
T __shfl_sync     (unsigned mask, T var, int srcLane,    int width=32);
T __shfl_up_sync  (unsigned mask, T var, unsigned delta, int width=32);
T __shfl_down_sync(unsigned mask, T var, unsigned delta, int width=32);
T __shfl_xor_sync (unsigned mask, T var, int laneMask,   int width=32);
int __all_sync(unsigned mask, int predicate);
int __any_sync(unsigned mask, int predicate);
unsigned __ballot_sync(unsigned mask, int predicate);
```