Spring 2019: Advanced Topics in Numerical Analysis: High Performance Computing Assignment 6 (due May 13, 2019)

Handing in your homework: Hand in your homework as for the previous homework assignments (git repo with Makefile), answering the questions by adding a text or a LaTeX file to your repo. Some useful material for this assignment can be found in the repo https://github.com/NYU-HPC19/homework6.

0. **Final project update.** Give us an update on the status of your final project. You can either just write a paragraph or use a table similar as in the previous homework assignment. Keep in mind that by the time this homework assignment is due, a big part of the work on the final project should be done.

1. **MPI-parallel two-dimensional Jacobi smoother.** We implement a distributed memory (i.e., MPI) parallel version of the two-dimensional Jacobi smoother from the second assignment. This is an extension of the one-dimensional case available in the class repository.[1] We will use a uniform domain splitting as sketched in Figure 1 and exchange unknowns corresponding to neighboring points (the so-called ghost points) on different processors. To make our lives easier, we only consider uniform splittings of all unknowns using $p = 4^j$, $j = 0, 1, 2, 3, \ldots$ processors. Additionally we assume that we deal with $N = 2^j N_l$ unknowns in the x and y directions, such that each processor works on N_l^2 unknowns. Before you start coding, figure out a few things:

   - For any p, find which points (and thus unknowns) must be updated by which MPI tasks.
   - Find which points must be communicated (i.e., the ghost nodes), and between which processors this communication must take place.
   - I suggest following our one dimensional example with blocking sends and receives by allocating $(N_l + 2)^2$ unknowns for each MPI task. The "inner" N_l^2 points are processed by each MPI tasks, while the outer points are used to store and update the ghost point copies from neighboring MPI tasks.

   Run your implementation on Prince. For large N_l (e.g., $N_l = 100$), perform a weak scaling study and plot the timings (fix the number of iterations for this study) as you increase the number of points and MPI tasks. Then choose N_l as large as possible to fit on one processor, and perform a strong scaling study, i.e., keep the problem size unchanged while increasing the number of MPI task, and plot the speedup compared to the ideal speedup. **Voluntary bonus question:** Compare a blocking with a non-blocking implementation, in which you overlap computation and computation, and study if you observe a comparison in the run time.[2]

[1] https://github.com/NYU-HPC19/lecture12.git

[2] Recall that the arithmetic intensity of this Jacobi smoother is low and thus the problem is memory-bound.

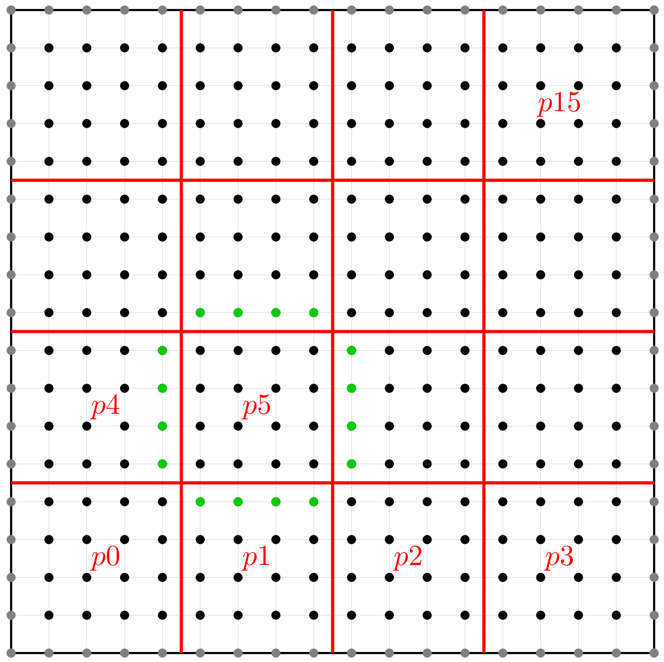

Figure 1: Uniform splitting of unknowns for parallel computation with 16 MPI processes, and with $N_l = 4$. Unknowns are shown as black dots, gray dots are domain boundary unknowns. As example, the ghost nodes processor $p5$ requires for updating its values in a Jacobi step are shown in green. $p5$ needs to obtain these values through communication with $p1, p4, p6, p9$, where they are updated.

2. **Parallel sample sort.** Each of P processors creates an N-vector of random numbers. The target is to sort the union of all these distributed vectors; this union, let's call it $\boldsymbol{v}$, consists of PN numbers and is assumed to be too large to fit into the memory of a single processor—thus, a serial sort algorithm cannot be used. The goal is to sort $\boldsymbol{v}$ such that every processor roughly holds about N sorted numbers, say $\boldsymbol{v}_i$, and that all elements on the processor with rank i are smaller than those on the processor with rank $i+1$ (for all $i = 0, 1, \dots, P-2$). The above repository contains a stub called `ssort.cpp`. This file also contains an outline of the algorithm, which we also discussed in class. For a summary of the sample sort algorithm, see the Wikipedia entry[3] for sample sort, as well as the pages linked under "References" from there. The main steps of sample sort are:

   - *Select local samples:* Each of the P processors sorts the local array and selects a set of S entries[4] uniformly and communicates these entries to the root processor, who sorts the resulting SP entries and determines $P-1$ splitters $\{S_1, \dots, S_{P-1}\}$, which it broadcasts to all other processors.[5]
   - *Distribute to buckets:* Each processor determines the "buckets" to which each of its N elements belong; for instance, the first bucket contains all the numbers $\leq S_1$, the second one are all the entries that are in $(S_1, S_2]$ and so on. The numbers contained

[3] http://en.wikipedia.org/wiki/Samplesort

[4] Choosing $S := P - 1$, as we've used in class, is a reasonable choice.

[5] Note that this step of determining the splitters is not necessary if one knows the statistical distribution of the random numbers. If that is not known, or each processor holds numbers that are distributed very differently, then this step is important to find reasonable bins.

in each bucket are then communicated; processor 0 receives every processor's first bucket, processor 1 gets processor's second bucket, and so on.

- *Local sort:* Each processor uses a local sort and writes the result to disc.

Include the MPI rank in the filename (see the example `file-io.cpp` example file). Run your implementation of the sorting algorithm on at least 64 cores of Prince, and present timings (not including the time to initialize the input array or the time to write output to file) depending on the number of elements N to be sorted per processor (report results for $N = 10^4, 10^5$, and 10^6).

3. **Extra credit: Generalize the Multigrid Implementation.** Generalize the one-dimensional serial multigrid implementation[6] in at least one (you choose!) of the following directions:

   (a) Generalize to the two-dimensional problem. From previous homework, you already have implementation of the two-dimensional Jacobi method. Note that for Jacobi within multigrid, one should use a relaxation parameter ω in the Jacobi update step—see the one-dimensional version.

   (b) Extend either the one- or the two-dimensional version to a shared memory (OpenMP) parallel implementation and run a series of large problems on Prince. Report scalability results.

   (c) Same but for distributed memory (MPI) version.

[6]To be posted at `https://github.com/NYU-HPC19/lecture13`.