# Spring 2017: Advanced Topics in Numerical Analysis: High Performance Computing Assignment 4 (due April 13, 2017)

**Handing in your homework:** Same method as for the second assignment (send us a link to the git repository). We will use the command

```
git clone YOURPATH/YOURREPO.git
cd YOURREPO
make
mpirun -np 4 ./mpi_solved1
...
mpirun -np 4 ./mpi_solved7
mpirun -np 16 ./jacobi-mpi2D 100 100
mpirun -np 10 ./ssort
```

to test your implementations. The git repository https://github.com/NYU-HPC17/homework4.git contains examples needed for this homework.

1. **Finding MPI bugs.** The above repository contains the files `mpi_bug1.c`, `mpi_bug2.c`, ..., `mpi_bug7.c`. These example codes contain bugs, resulting in hangs or other undesirable behavior. Try to find these bugs and fix them. Add a short comment to the code describing what was wrong and how you fixed the problem. Add the solutions to your repository using the naming convention `mpi_solved{1,2,...}.c`. Each problem should be run with 4 MPI tasks.

2. **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 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, we need to 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, and between which processors this communication must take place.

[1] https://github.com/NYU-HPC17/lecture8.git

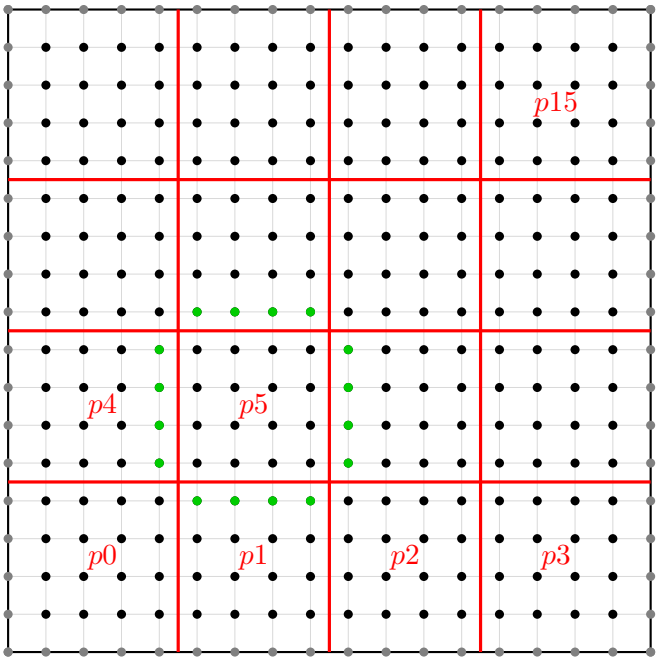


**Figure 1:** Uniform splitting of unknowns for parallel computation with 16 MPI processes, and with $N_l = 4$. Unknowns are shown as black dots, gray dots are domain boundary unknowns. As example, the ghost nodes processor $p5$ requires for updating its values in a Jacobi step are shown in green. $p5$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

- I suggest following my 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 Stampede. 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 on Stampede.[2]

3. **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

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

$i = 0, 1, \ldots, P-2$). The above repository contains a stub called `ssort.c`, which also contains an outline of the algorithm. 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 selects a set of $S$ random entries and communicates these entries to the root processor, who sorts the resulting $SP$ entries and determines $P-1$ splitters $\{S_1, \ldots, S_{P-1}\}$, which are broadcasted.
- *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 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 `pingpong_array.c` example file). Run your implementation of the sorting algorithm on at least 64 cores of Stampede, and present timings depending on the number of $N$ of elements to be sorted per processor.

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