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

Terrence Alsup

All runs were done on Prince with 64 cores by running an interactive session with the command:

```
srun --nodes=8 --ntasks-per-node=8 --pty /bin/bash
```

The processors used were Intel(R) Xeon(R) CPU E5-2690 v4 @ 2.60GHz. The Python file `timing_plotter.py` can be run to plot the strong and weak scaling timings for the 2D Jacobi method. This files requires matplotlib which is included in Anaconda3.

0. **Final project update.** Anya and I have finished most of the 1D parallel implementation of KMC using MPI. We have a bug that is causing the program to hang that we are still working on. The serial code for 1D as well as 2D has already been completed. Next we will move to the 2D parallel case which should be a straightforward extension given our 1D implementation. Hopefully by the end of this week we will have moved on to performing scalability tests.

1. **MPI-parallel two-dimensional Jacobi smoother.**
   The 2D Jacobi smoother is implemented in the file `jacobi2D-mpi.cpp`. We assume that the number of cores used is a power of 4 (i.e. 4^j for some $j = 0, 1, 2, \ldots$), and that the size of the matrix is $N \times N$ with $N = 2^j N_l$. Figure 1 below shows the timings as we increase the number of processors with a fixed amount of work per processor. Figure 2 shows the timings as we increase the number of processors for a fixed amount of work.

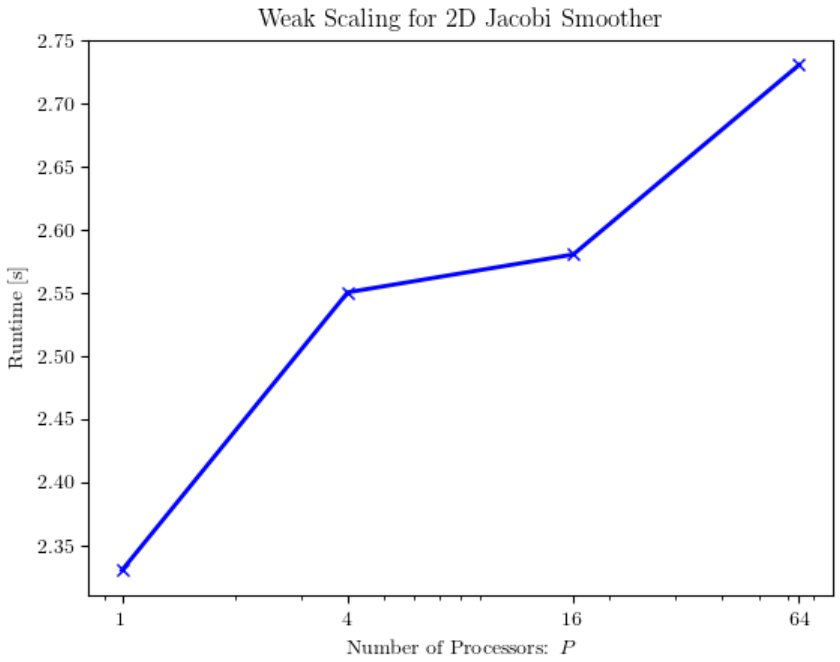

Figure 1: A weak scaling study for the MPI implementation of the 2D Jacobi smoother. We have fixed $N_l = 400$ and do 10^4 Jacobi iterations. We see that increasing the number of processors slightly increases the runtime, mainly due to more memory transfers.

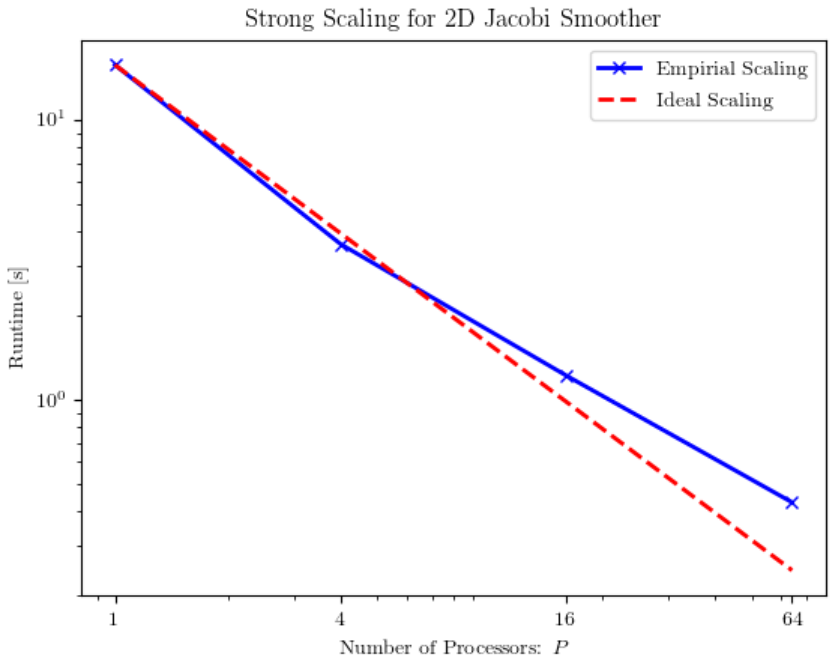

Figure 2: A strong scaling study for the MPI implementation of the 2D Jacobi smoother. We have fixed $N = 1000$ and do 10^4 Jacobi iterations. We see that increasing the number of processors decreases the runtime and the scaling is approximately a factor of $1/4$-th, which is ideal.

2. **Parallel sample sort.**

   The MPI implementation of sample sort is in the file `ssort.cpp`. Table 1 below shows the timings for different N on Prince with 64 cores.

N	Runtime [s]
10^4	0.82
10^5	0.84
10^6	1.13

Table 1: The runtime [s] for sample sort with different N. For $N = 10^4$ and 10^5 the results are roughly the same indicating that most of the time is spent transferring memory between the processors. $N = 10^6$ is large enough so that time for memory transfers is no longer dominant.