Brief bio. I’m a second-year PhD student in EECS interested in a combination of machine learning methodology (e.g. distributed algorithms for machine learning, fast MCMC methods) and natural language processing (e.g. using large-scale corpora to improve semantic analysis). My strongest background is in mathematics, theoretical computer science, algorithms, and the kind of software building and engineering needed in research in machine learning and NLP. After taking this class, I hope both to understand the principles required to build good parallel/distributed systems and to gain proficiency with the software tools/libraries that are used to write efficient parallel code (e.g. MPI, potentially CUDA, etc.).

Data-parallel Markov chain Monte Carlo (MCMC). A substantial body of recent work in statistical machine learning has focused on making Markov chain Monte Carlo methods fast by parallelizing them to varying degrees. Some of these are mostly asynchronous algorithms that communicate only via a global set of summary statistics; another breed is the so-called “data parallel” kind, which splits the data into pieces, runs MCMC independently on each part of the data, and then finds a way to recombine them. In recent work, a collaborator and I developed a version of this algorithm that optimizes over a parametric family of recombination functions in order to find the best one (by a standard metric of goodness of approximation for distributions).

The strengths of our work are as follows: (1) the algorithm is generic and can be easily applied to any MCMC algorithm operating on continuous variables; (2) the only synchronization required is at the end of sampling; and (3) partly as a result of the second fact, we were able to demonstrate a linear speedup over standard MCMC (with an accuracy penalty smaller than competitors).

Nonetheless, several interesting limitations exist that indicate future work, particularly on the engineering side. One of these is that we relied on a shared memory model. While a shared memory model makes sense for establishing the core accuracy and efficiency properties of the algorithm, a more realistic setting would involve a network of interconnected processors. Building a system implementing our algorithm in that setting would be very interesting since it would allow us to run it on huge problems where the data must be distributed across a cluster. Finally, we implemented our code in Python and believe that
the inherent efficiency limitations of the language likely slowed it down, and that
speedups in the numerical computation parts of the code could likely be attained
by switching to a C/C++ implementation (although we did use the BLAS
libraries through numpy, so the exact speedups remain unclear). Likewise, we
relied on a parallel processing utility library from IPython, which exposes a
very limited interface and whose efficiency could probably be improved upon by
writing custom lower-level parallel code (part of what I hope to be able to do
after finishing this class).