CS140 Final Project Progress Report

Stefan Seritan (PERM: 5466644), Wei Dai (CSIL: wdai, PERM: 6925747)

March 4, 2015

Serial Results

The Metropolis Monte Carlo simulation for the Potts Lattice Gas (PLG) model was implemented in C++. The melting point and phase diagram for our test system were calculated, as shown below in Figures 1 and 2. The melting point is indicated by the peak in the heat capacity, which is at $kT = 1.35$ for this system. As for the phase diagram, the two phase region ends at $kT = 1.05$. Above this value, the solid can exist as an even mixture, while the solid will split into a species 1 rich phase and a species 2 rich phase within the two phase region (i.e. not perfect mixing). These two values will be used to ensure the algorithm remains formally correct after parallelization.

Figure 1. Heat Capacity ($\frac{dE}{dT}$) vs. Temperature (kT)

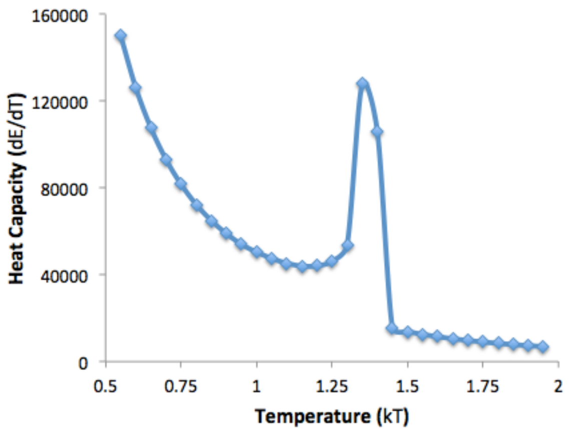
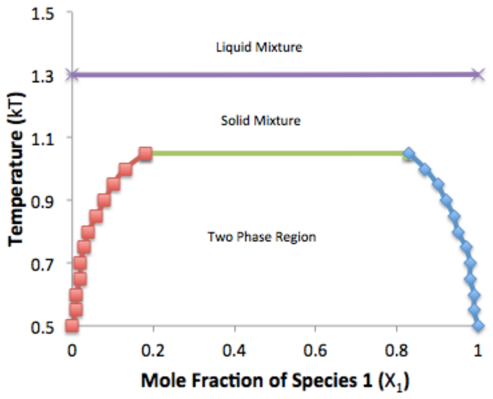


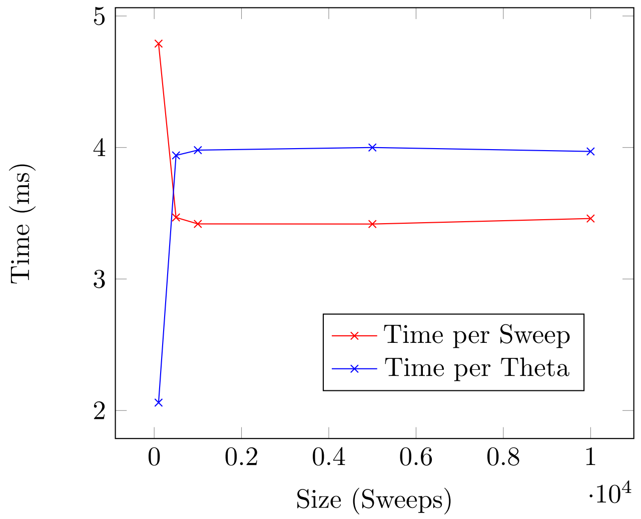
Figure 2. Phase Diagram



Serial Performance

We tested the performance of the serial algorithm on Triton, compiled with icc and -O2 optimization flags. The lattice size we used was $10 \times 10 \times 100$, for $100, 500, 1000, 5000, 10000$ equilibration and data sweeps (with Θ calculated every sweep during data collection). Timing was done using C++11's chrono library. The time needed per sweep and to calculate Theta every sweep is reported below in Figure 3.

Figure 3. Milliseconds for Sweeps and Theta

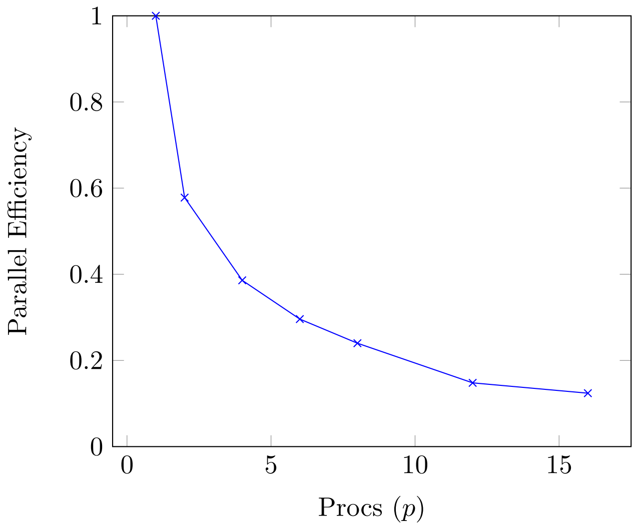


There are a few key insights from the timing information of the serial program. Unsurprisingly, each sweep takes constant time since the lattice size does not change. Therefore, our program is linear in the number of iterations. More importantly, we can see that actually calculating Θ takes more time than one sweep.

Parallelizing Θ Calculation

Parallelizing data collection is theoretically very easy, since it is a 7-point stencil with no data races. We simply wrapped the stencil calculation in cilk_for's, split up in three dimensional blocks. The size ($n = 100, 500, 1000, 5000, 10000$ at 16 procs) and processor ($p = 1, 2, 4, 6, 8, 12, 16$ for 5000 eq. and data sweeps) scaling was studied using the same chrono timing used for the serial version. Since only Θ was parallelized at this stage, only the time for Θ is reported in Figure 4 below.

Figure 4. Parallel Efficiency for Theta Calculation



On average, Θ now only takes over 1 ms to calculate, but it surprisingly scales extremely poorly with increasing processors. We tried several other schemes to iterate over the arrays to increase parallel efficiency, but nothing was very effective. We suspect that perhaps the issue is C++'s internal array implementation. We plan on rewriting the program using a more efficient object oriented approach that will hopefully allow Cilk Plus to more efficiently parallelize the calculation of Θ.

Parallelizing Move Execution

The other bulk of computational time is actually generating and executing moves. Since the canonical ensemble features the particle swap moves, geometric division cannot be used because almost half of the moves will require communication. Our proposed scheme was to centrally generate moves and then distribute them in independent batches to Cilk to be executed in parallel. This can be done so long as a move does not conflict with moves assigned to 2 processors. Best case scenario for a batch is to do half a sweep, in a checkerboard pattern; however, in reality we find that we can only do approximately 3% of a sweep before we have to wait for a sync. Since the moves are so lightweight, even the minimal overhead of our separate queues and conflict checking cancels out the gains we get for using multiple cores.

We have tested version of the parallel move execution with Cilk Plus, TBB, and OpenMP, but only TBB with the newest compilers on our local machine actually beat the serial version. Again, we believe that our data structures are to blame. We hope that shifting to a better data structure will reduce the overhead of assigning moves and making the parallelization actually worth it.

Further Parallelization

In addition to the parallelization described above, we can further parallelize the simulation by splitting data sweeps. Assuming the system is at equilibrium, each data sweep simply serves to collect more data about a system that is no longer changing. We could easily split the data sweeps across nodes (potentially using MPI) and have each node collect data independently, only communicating at the very end to average the results. We have not implemented this yet, but it should provide easy scalability.