Final Report
Parallelizing A Projective Algorithm for Simulating Stochastic Dynamical Systems
CSE 392 - Prof. Biros

Carlos del-Castillo-Negrete
UT EID - cd34775
(Dated: May 20, 2019)

I. INTRODUCTION

Stochastic dynamical systems can often be modeled by the independent, random motion of a large ensemble of particles in phase space. A Standard Monte-Carlo (SMC) algorithm can simulate such a system by numerically integrating independent sets of stochastic differential equations (SDEs) for a large number of particles sampled from an initial distribution. These algorithms are an attractive option for parllalel architectures as they are embarrasingly parallel. Nevertheless simulations with a very large number of particles must be done to obtain accurate results.

In this project we consider a variant on the SMC algorithm that we will call the Projective Monte-Carlo (PMC) algorithm (Figure 1). The PMC algorithm improves on the SMC algorithm by using tensor product decomposition (TDP) techniques combined with projective integration to accelerate convergence. By recognizing that time-dependent MC simulation data can be viewed as a "video" describing the evolution of a probability distribution in time, the PMC algorithm constructs an optimal basis that captures the major variability of the simulation data via a TPD that is traditionally used to compress video data. Using this TPD, the PMC algorithm projects an estimate of the distribution at a later time. By alternating between computationally expensive direct simulation and quick projection steps (Fig. 1), the PMC algorithm has the potential to accelerate convergence to a steady state solution significantly.

The parallelization strategy for the PMC algorithm used in this project was to focus on parallelizing the largest computation bottleneck in the algorithm - the SMC integration (see Table in Figure 2). However unlike

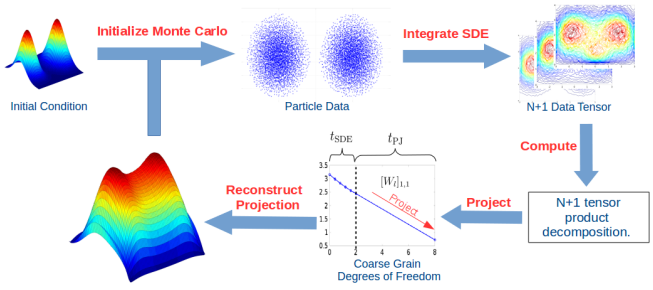

FIG. 1: Schematic diagram of the PMC Algorithm.

	Standard MC	Projective MC		
PJ Ratio	0.00	0.25	0.50	0.75
SMC Runtime (sec)	2391.0	1806.8	1139.9	713.85
PMC Runtime (sec)	0.00	236.29	142.39	89.03
TOTAL RUNTIME (sec)	2391.0	2043.1	1282.2	802.87
Speedup	−−	**1.170**	**1.865**	**2.978**

FIG. 2: Example run-times for serial SMC and PMC on a model problem (diffusive process).

simply parallelizing the SMC integration, the PMC algorithm requires additional communication during particle trajectory integration to capture information about the evolving probability distribution of the system over time to then later use in the projective step. We expect that this additional communication will cause overhead in a parallel implementation, thus there is a trade-off between collecting more data and getting an accurate projection.

For this project I focused on studying the scalability of the embarrasingly parallel SMC algorithm and its variant necessary for the PMC algorithm, which we'll call the SMC+D algorithm (SMC + Data collection). Implenentations were done in Fortran90 and parallelized using MPI. Test runs were done on the Lonestar5 cluster. Results show that, as expected, the SMC algorithm shows great scalability. More surprisingly however, the SMC+D algorithm shows good scalability results as well, even when scaling up the amount of data collected.

II. METHODOLOGY

The core components of the SMC/SMC+D algorithms and a quick work-depth analysis and summary of how they were parallelized follows. Note that the over-all parallelization strategy was to divide n_p particles amongs P processesors and do the work for each indpenendently.

1. *mc_f_to_xy*() - Converts a stastical representation (*stat_rep* type in modules) of the dynamical system, which consists of a probability distribution function defined over a phase space grid (i.e. a matrix since limiting to $d = 2$ dimensional systems), to a particle representation of a system (*particle_rep* type in modules) , which consists of "(x, y)" phase-

```
subroutine mc_fit_grid_to_particles(curr_gd, p_rep)
  ! Find new upper and lower bounds per task
  new_upper(i) = max_serial(p_rep%p_i(i,j))
  new_lower(i) = min_serial(p_rep%p_i(i,j))

  ! All Reduce so that ever task has new lower and upper bounds
  call MPI_ALLreduce(new_lower, curr_gd%lower_bnds, nd, MPI_REAL,
      &
                          MPI_MIN, MPI_COMM_WORLD, ierror);
  call MPI_ALLreduce(new_upper, curr_gd%upper_bnds, nd, MPI_REAL,
      &
                          MPI_MAX, MPI_COMM_WORLD, ierror);

  ! Scale grid to preserve spacing
  call scale_grid(curr_gd)
end subroutine mc_fit_grid_to_particles
```

FIG. 3: Pseudocode for the fit grid method. Note that communication is done via an allreduce operation along each dimension upper and lower bounds so that each process remains with new and upper lower bounds. All processes needed the updated grid since each counts particles in grids seperately and only sends that information (not actual particle coordinates).

```
function mc_xy_to_f (p_rep, gd)
  ! Count number of particles in each cell grid
  do i=1,np
      ! Get index of cell
      x_i(:) = get_index(p_rep%p_i(i))

      ! Add to grid
      np_ij(x_i(1),x_i(2)) = np_ij(x_i(1),x_i(2)) + 1.0
  end do

  ! Reduce np_ij accross processes
  call MPI_Reduce(np_ij, s_rep%f_yx, ny*nx, MPI_REAL, &
                MPI_SUM, 0, MPI_COMM_WORLD, ierror)

  if (rank == 0) then
    s_rep%f_yx(:,:) = build_distribution_fun(np_ij(:,:))
  end if

  mc_xy_to_f = s_rep
end function mc_xy_to_f
```

FIG. 4: Pseudocode for routine that essentially collects data points by estimating the distrbution function from particle coordinates.

space positions of n_p particles. Initialization consists of first integrating over the entire grid to see how many particles end up in each cell and then perturbing each particle randomly within its own cell to initialize the distribution. Assuming $O(1)$ cost of integration over each cell, the work for this algorithm is $O(g_{sz} + n_p)$, where g_{sz} is the size of the grid (number of cells). The depth of this algorithm is the same, since each cell and particle and can be initialized independently. Note no communication is required, and each processos must iterate over the grid (i.e. not parallelized), thus the parallel complexity of the implementation is $T_{mc\_f\_to\_xy}(n_p, g_{sz}, P) = O(g_{sz} + \frac{n_p}{P})$.

2. *mc_step*() - For each time step, calculates new position of each particle by using a user defined function that integrates stochastic differential equations for mechanics of given problem given as input a particles current position and a random number. Work required for this algorithm is $O(n_p * n_t)$, where n_t is the number of time steps that are needed. Depth of this algorithm is $O(n_t)$ since each particle's position depends on its previous value, but not on the other particles' positions. Note that it is assumed that the integration itself, a problem dependent operation, is $O(1)$. Finally, since work is divided equally amongst P processors, the parallel complexity becomes $T_{mc\_step}(n_t, n_p, P) = O(\frac{n_t * n_p}{P})$

3. *mc_fit_grid_to_particles*()− This routine determines how the current grid must be scaled to fit all particles accross all processes. This takes $O(d * n_p)$ work, where d is the dimensionality of the problem (2 in our case), and has $O(log(n_p))$ depth since we have a scan along each dimension which are independent of one another. Refer to Figure 3 for Pseudocode. Communication was implemented via an MPI_Allreduce along each dimension's upper and lower bounds so that each process remains with new and upper lower bounds. Therefore the messages size is simply $d * 2$ for this communication, giving a parallel complexity of $T_{fit\_grid}(d, n_p, P) = O(\frac{d*n_p}{P} + l*log(P) + \frac{2d*P}{b})$, where l, b are the latency and bandwidth respectively.

4. *mc_xy_to_f*()− Does the opposite of *mc_f_to_xy_*() and converts a particle representation of a dynamical system into a probability density function estimate from a particle histogram count over a grid. Note this method assumes each process has an appropriate grid and all these are in synch (i.e. fit grid methods has been called already). The work is thus $O(n_p + g_{sz})$ and the same depth, just like *mc_f_to_xy_*(). Note however communication must be done amonst processes so that the count of each grid cell accross all processes can be determined. This was done via an MPI_Reduce operation (Figure 4), thus giving this part of the algorithm a parallel complexity of $T(n_p, g_{sz}, P) = O(\frac{n_p}{P} + (l + g_{sz}/b) * log(P))$, where l, b are the latency and bandwidth respectively.

5. *mc_solve_and_record*()− Combines all the above into the SMC+D Algorithm (Figure 5). Note that the same routine is used for the SMC and SMC+D algorithm, since the only difference is N, the number of the data points we collect. for SMC we have $N = 1$ - we only care about the final distribution. For SMC+D we have $N > 1$ - multiple data points are collected are equally spaced intervals in time. Combining the total parallel complexity estimates from the subroutines, the overall parallel complexity of the SMC+D algorithm becomes.

```
subroutine  mc_solve_and_record(mc_prob, np, N, out_data_file)
  ! Construct particle representation from given initial s_rep
  p_rep = mc_f_to_xy(init_dist, np_n)

  do i=1,N
      ! Standard MC Integration
      call mc_step(p_rep, nt, mc_prob%delta_t, mc_prob%dX)

      ! Update grid of problem to contain all the particles
      call mc_fit_grid_to_particles(mc_prob%gd, p_rep)

      ! construct and store reconstructed distributuion function
      mc_data%s_rep_i(i+1) = mc_xy_to_f(p_rep, mc_prob%gd)

      ! Project to Next time step
  end do
  ! Destroy particle rep (only used in this method)
  call mc_destroy_prep(p_rep)
end subroutine mc_solve_and_record
```

FIG. 5: Pseudocode for SMC+D Algorithm.

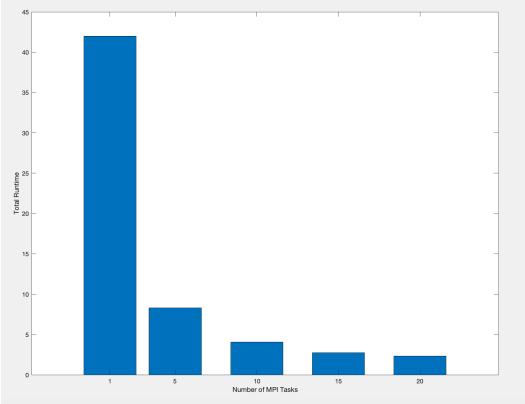

FIG. 6: Total run-time vs. number of MPI tasks for the $n_p = 5e5, N = 5$, test runs. Good speed-up as we increase number of tasks.

$$
\begin{aligned}
T(n_p, g_{sz}, P) &= T_{mc\_f\_to\_xy}(n_p, g_{sz}, P) + \\
&\quad N * [T_{mc\_step}(n_t/N, n_p, P) \\
&\quad + T_{mc\_f\_to\_xy}(n_p, g_{sz}, P) \\
&\quad + T_{fit\_grid}(d, n_p, P)] \\
&= O(g_{sz} + \frac{n_p}{P} + \frac{n_t * n_p}{P} \\
&+ \frac{d * n_p * N}{P} + N * l * log(P) + \frac{2d * P * N}{b} \\
&\quad + \frac{n_p * N}{P} + N * (l + g_{sz}/b) * log(P)) \quad (1)
\end{aligned}
$$

While a bit messy, the important take-away from this estimate are the two terms of $N * log(P)$ that quantify the increased communication costs associated with taking more data points and using more processes to do so.

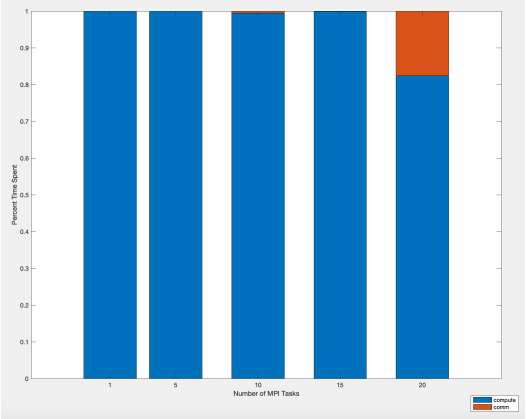

FIG. 7: Run-time break down vs. number of MPI tasks for the $n_p = 5e5, N = 5$, test runs. Note the jump in communication costs doesn't become apparent until $P = 20$ tasks.

III. EXPERIMENTAL SETUP

All routines were implemented in set of Fortran90 libraries and compiled using mpif90 on Lonestar 5. In the root directory of the source code, *$ make all* will build the necessary routines. In the *test* directory there is an example problem implemented in *mc_test.f90*, which was used to run the scalability tests presented. The file test_functions.f90 contains the necessary test function for the model problem. Running *make all* from the *test* directory will build the test examples, from then which an example test run can be run with a command such as *$ ibrun -np 10 $HOME/code/test/test_mc 100000 1 out_file*. Output data will be sent to the specified file. In the matlab directory there will be accompanying matlab scripts that can be used to read in the simulation data and plot the metrics presented. Finally the directory *jobscripts* contains examples of jobscripts submitted to compile the results presented.

The following tests were run to study the scalability of the SMC+D Algorithm. They are presented with hypothesis of what is expected in each case.

1. Strong scaling for small data $N = 5$ - Keeping the number of particles $N = 5e5$ and number of time steps fixed, with $N = 5$ data points being collected, run-time will be measured for $P = 1, 5, 10, 15, 20$ tasks. The time spent computing (*mc_step*) will be presented vs the cost of "communicating" (collecting data from the run, i.e. everything else). The increased cost of communication in terms of fraction of total run-time should be apparant the more procesors used.

2. SMC+D Weak Scaling for large data $N = 20$ - Keeping then number of particles per task fixed at $N = 1e6$, the run-time will be will be measured for $P = 1, 5, 10, 15, 20$ tasks. As before the time spent computing vs communicating will be pre-

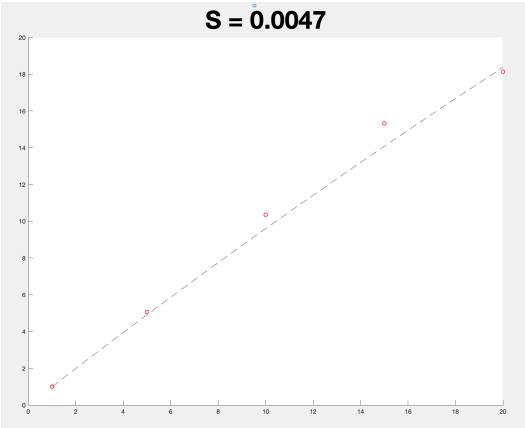

FIG. 8: Fitting Amdahl's Strong Scaling law to data points - $s = 0.0047$. On the x-axis is number of tasks while on the y axis is the speed-up. This low sequential fraction reflects how the algorithm is indeed embarrsingly parallel and nearly completely parallelizable when collecting a small number of data points.

sented. For this experiment the effect of communication on total run-time is expected to be even more apparent than the last case.

3. SMC+D Total Run-time vs. Number of data points N - For $1e7$ particles and $P = 96$ tasks, seperate runs collecting $N = 1, 10, 20, 30, 40, 50$ data points were run. Communication cost is expected to increase as the number of data points collected incerases.

IV. RESULTS

Results for each case studied are summarized in the sections to follow.

A. Experiment 1 - Strong Scaling

The SMC+D algorithm performed very well in strong scaling tests when collecting $N = 5$ data points. First of all net 20X speed-up was achieved in the best case when comparing between the sequential algorithm (first column Figure 6). Also the minimal communication costs is clear in Figure 7. Finally we see that when we fit Amdahl's to our data points we get an observed sequential fraction of $s = 0.0047$. This reflects the fact that the SMC+D algorithm is indeed embarrasingly parallel for small N and has minimal sequential portion that cannot be sped up by parallelization.

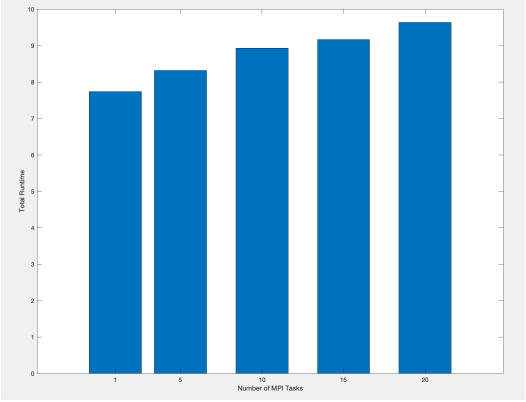

FIG. 9: Total run-time vs. number of MPI tasks for the $n_p/P = 1e5, N = 20$, test runs. Increased run-time for more processes observed as expected as communication costs increase.

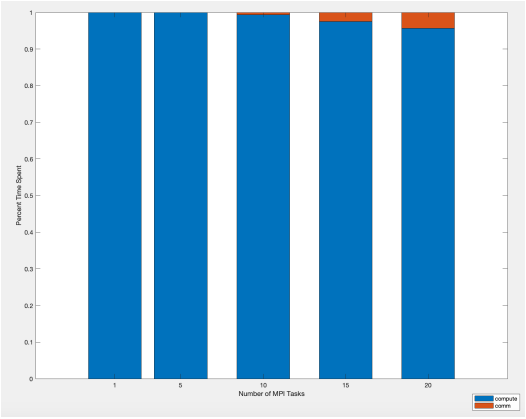

FIG. 10: Run-time break down vs. number of MPI tasks for the $n_p/P = 1e5, N = 20$, test runs. Note incerase in communication costs is minimal.

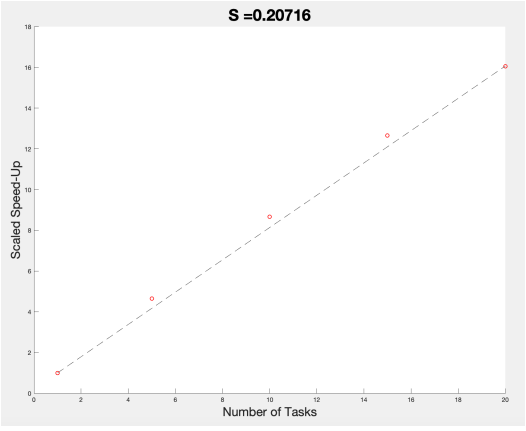

FIG. 11: Fitting Gustafson's Strong Scaling law to data points - $s = 0.20716$. On the x-axis is number of tasks while on the y axis is the sacled-speed-up (speed-up per processor). This higher sequential fraction makes sense since we are taking more snapshots $N = 20$ than in the previous case.

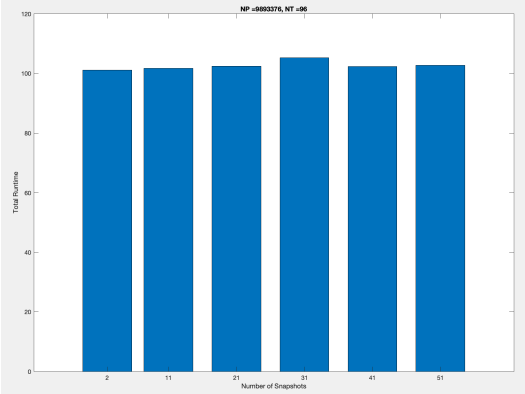

FIG. 12: Total run-time vs Number of data points collected N for $n_p = 1e7$ particles. Note that run-time doesn't scale up.

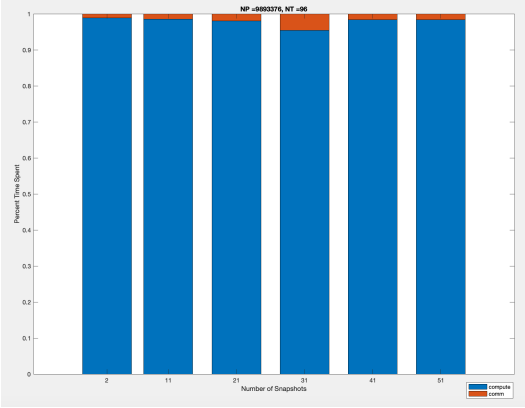

FIG. 13: Run-time breakdown vs. Number of data points collected n for $n_p = 1e7$ particles. Nte increased communication costs at first but then a plateau (and even decline!).

B. Experiment 2 - Weak Scaling

The SMC+D algorithm also performed well in weak scaling tests when collecting $N = 20$ data points and using a constant $n_p = 1e5$ particles per process. Figures 9 and 10 show that the total run-time remains near constant, with only miminal increase due to communication costs for more processes used. In Figure 11 we see the effect of fitting Gustafson's law to th weak-scaling data, resulting in a predicted sequential parameter of $s = 0.207$. note that this predicted sequential paramter is larger in this case as expected as more data points are being collected than in the first experiment.

C. Experiment 3 - Data Points (N) Scan

For the data points scan runs the observation can be made that the amount of communication costs involved with collecting data may not scale poorly as the amount of data collected is increased. The run-time overall didn't really increase as the number of data points increased (Figure 12), and the ration of time spent communicating didn't necessarily scale upwards either (Figure 13). So while some associated cost with collecting data in the SMC+D algorithm is obviously presented, collecting more data points isn't neccessarily that bad!

V. CONCLUSIONS

From the results it can be concluded that the SMC+D algorithm implemented scales well and does a good-job of speeding up the sequential bottleneck of SMC integration within the PMC algorithm. Furthermore it seems as if collecting more data points (up to some limit surely) is beneficial in the sense of performing a better projection since more data is colelcted and, as is apparent from the results presented, not too detrimental to the overall run-time of the parallelized algorithm.