Parallelizing a Projective Algorithm for Simulating Stochastic Dynamical Systems

Carlos del-Castillo-Negrete

CSE 392 - Spring 2019

May 10, 2019

Overview

Outline

- Motivation
  - Stochastic dynamical systems
  - Monte-Carlo Simulations
- Algorithm
  - Standard Monte-Carlo
  - Projective Monte-Carlo
  - Parallelization Strategy
- Implementation
  - Fortran Library
  - Main Routines
- Results/Conclusions
  - Strong/Weak Scaling
  - Problem Parameters

Motivation

Stochastic Dynamical Systems

- Model using independent, random motion of a large ensemble of particles in phase space.
- System governed by a set of Stochastic Differential Equations - Every particle's motion can be determined **independently from the others**.
- Search for **steady state solutions** of system.

Standard Monte-Carlo (SMC)

1. Initialize N_p particles from initial probability distribution.
2. Solve each particle's SDE independently to determine position at some later time.
3. Estimate final distribution function from final particle positions.

Methodology - Algorithm

SMC Analysis

- The Good
  - General algorithm is applicable to wide range of applications.
  - **Embarrasingly parallel**
- The Bad
  - Statistical errors - Need large N_p.
  - Don't care about particles, care about distribution.
  - Curse of dimensionality.
- Parameters
  - T = Number of Timesteps, N_p = Number of particles
  - $cost_{SDE}$ = Cost of solving SDE for one particle at one timestep.
  - Work - $O(N_p * T * cost_{SDE})$
  - Depth - $O(T)$

Methodology - Algorithm

Projective Monte-Carlo (PMC)

- Idea - Take snapshots of distribution during SMC (N_s).
- View snapshots as video of distribution evolving in time.
- Use Tensor Product Decomposition (TPD) techniques to project distribution function at later time using collected data.
- Alternate between SMC and TPD+Projection to accelerate convergence to steady-state.

Parallelization

- SMC Integration bottleneck.
- PMC requires intermediate data gathering $\Rightarrow$ Communication/Synchornization Costs.
- **Goal** - Focus on Parallelizing SMC+data gathering.

Methodology - Algorithm

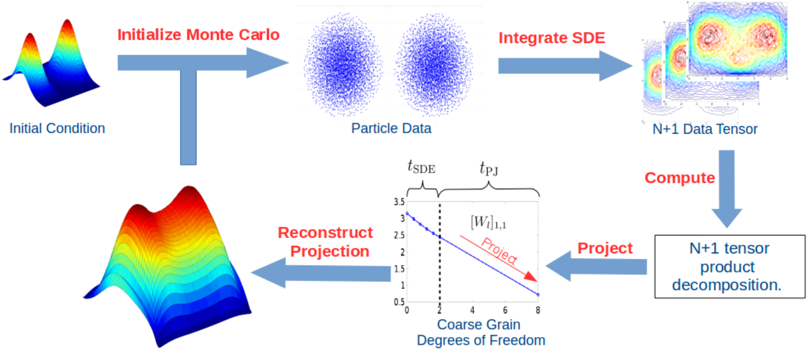


Figure: Schematic diagram of the PMC Algorithm.

Implementation

Fortran Library

- Fortran90 modules.
- User defined problem - User inputs SDEs of system.
- External function pointers - Performance vs Generality.

Monte-Carlo Problem Data Structure

```
type mc_problem
  type(grid) :: gd
  procedure(func_x_i), pointer, nopass :: f_init
  procedure(stoch_diff_dx), pointer, nopass :: dX
  real(8) :: t_i, t_f, delta_t
  type(stat_rep) :: s_rep_init
  type(mc_sim_data) :: mc_data
end type mc_problem
```

Implementation - Data Structures

Particle Representation

```
type particle_rep
  integer :: np
  integer :: nd
  real(8), dimension(:,:), allocatable :: p_i
end type particle_rep
```

Statistical Representation

```
type stat_rep
  type(grid) :: gd
  real(8), dimension(:,:), allocatable :: f_yx
end type stat_rep
```

Implementation - Main Routines

Fitting Grid

```
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
```

Implementation - Main Routines

Statistical → Particle - Snapshot

```
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
```

Implementation- Main Routines

Solve and Record

```
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
```

Experimental Set-Up

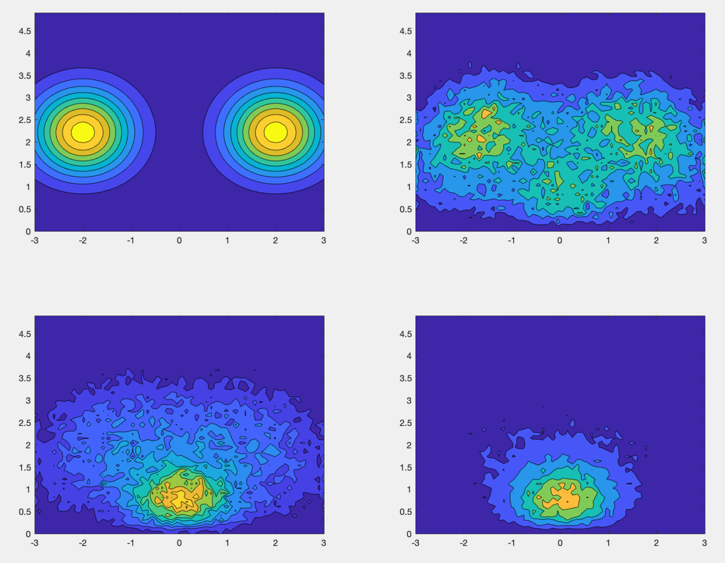
Model Problem

- System - Simple diffusive process - Relaxation to steady-state distribution.
- Performance - Measure run-time for fixed number of timesteps.

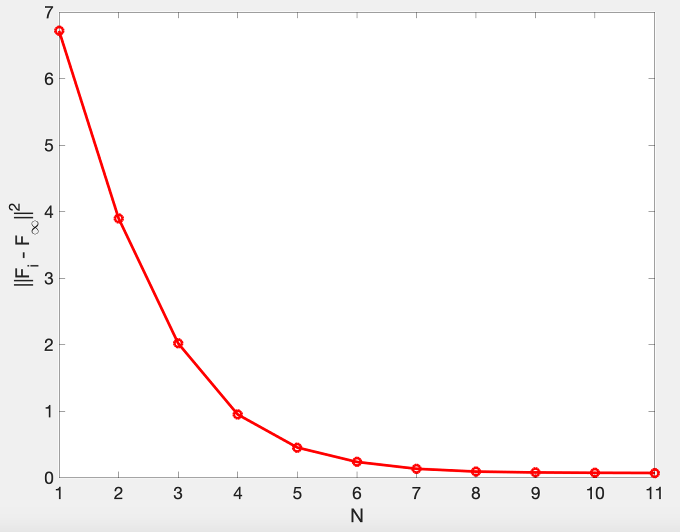
Tests

1. Strong Scaling - SMC should have good strong-scaling. PMC should still have good strong scaling that worsense as we increase N_s.
2. Weak Scaling - Number of particles per task constant - good weak scaling should be achievable.
3. N_s Parameter - Difference in workload as we change N_s - Expect more time to be spent communicating.

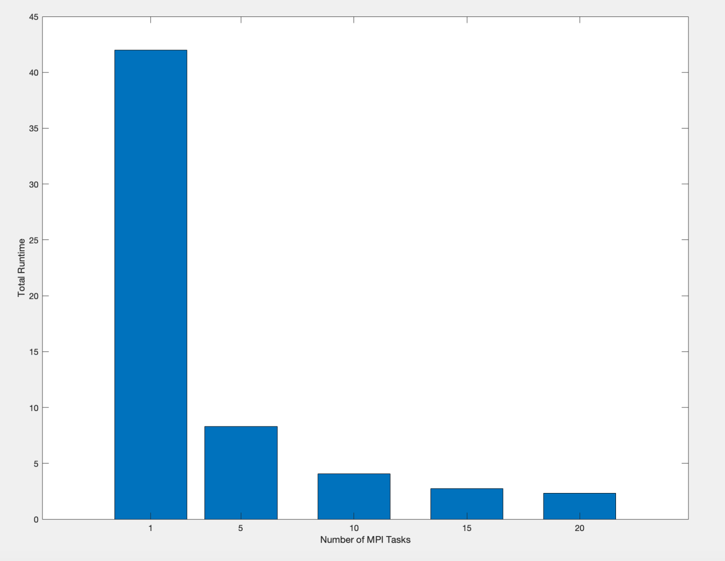
Results - Strong Scaling - $N_p = 5e5, N_s = 5$



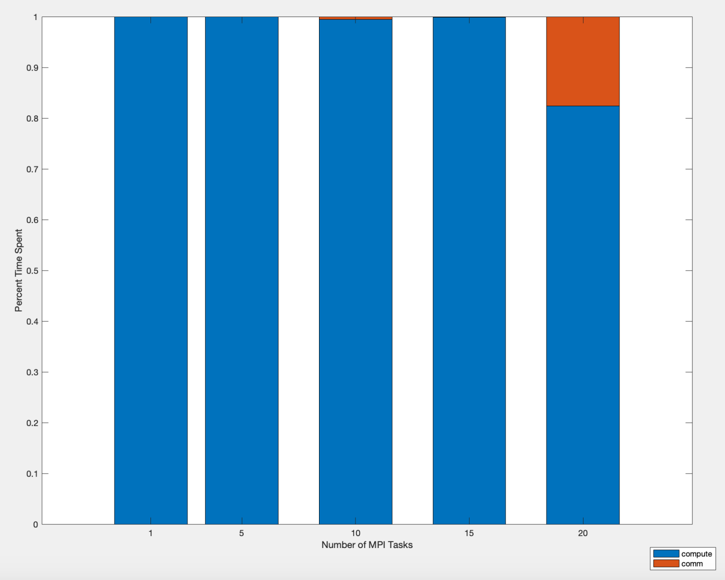
Results - Strong Scaling - $N_p = 5e5$, $N_s = 5$



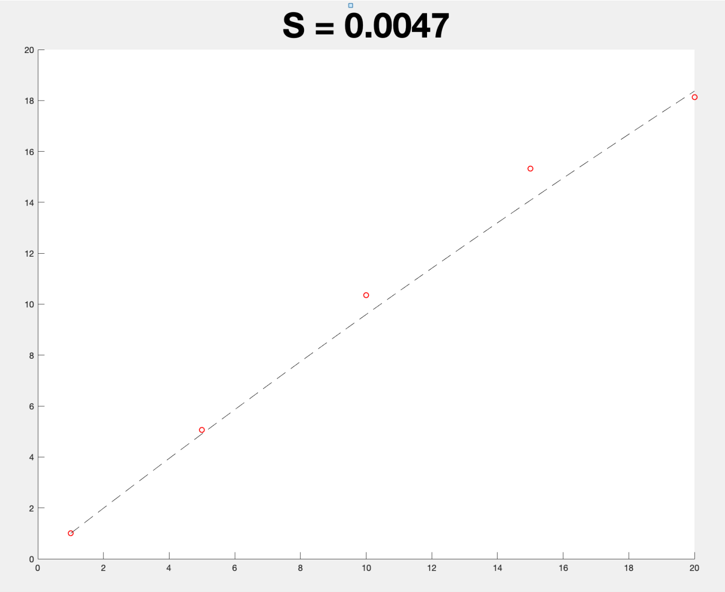
Results - Strong Scaling - $N_p = 5e5, N_s = 5$



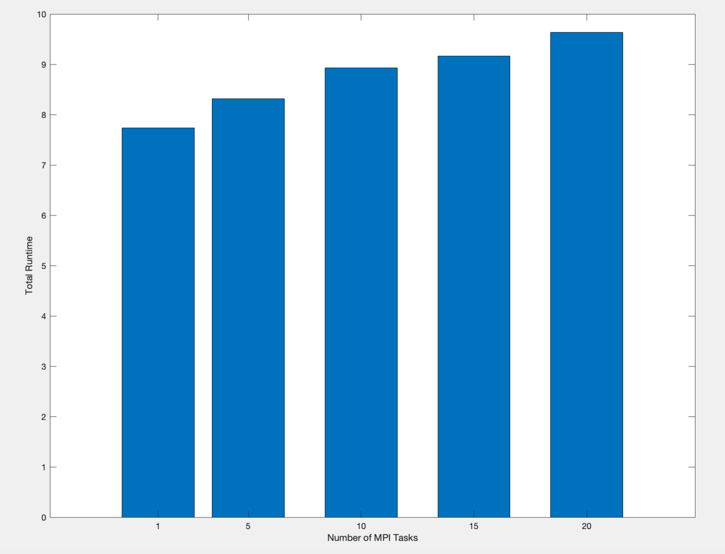
Results - Strong Scaling - $N_p = 5e5, N_s = 5$



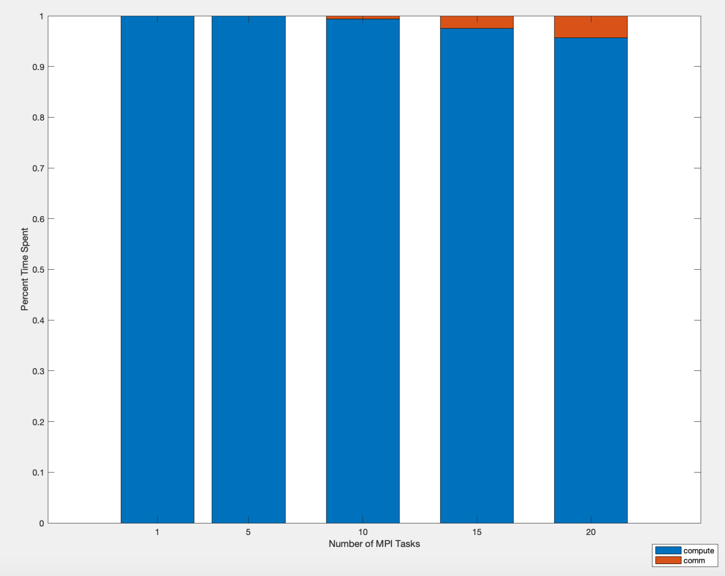
Results - Strong Scaling - Speedup $= \frac{1}{s - \frac{(1-s)}{N}}$



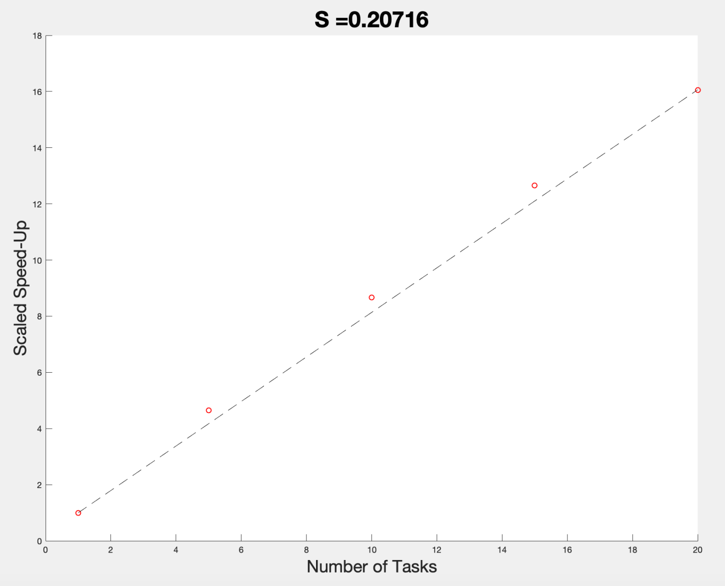
Results - Weak Scaling - $N_p/N_t = 1e6$



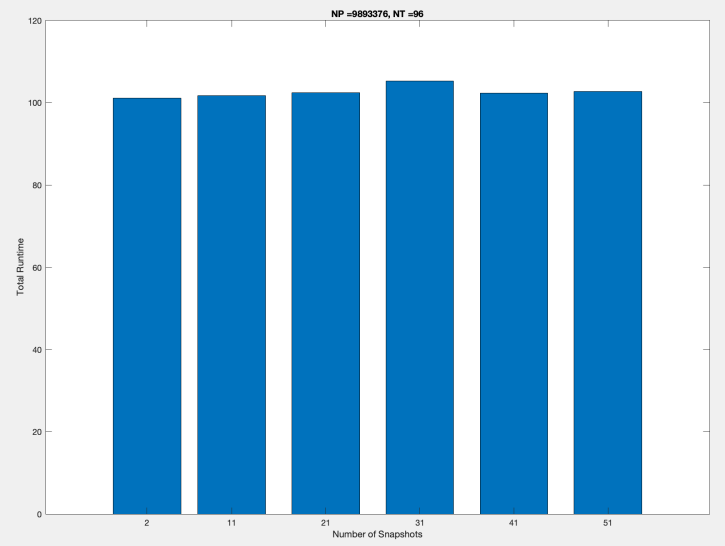
Results - Weak Scaling - $N_p/N_t = 1e6$



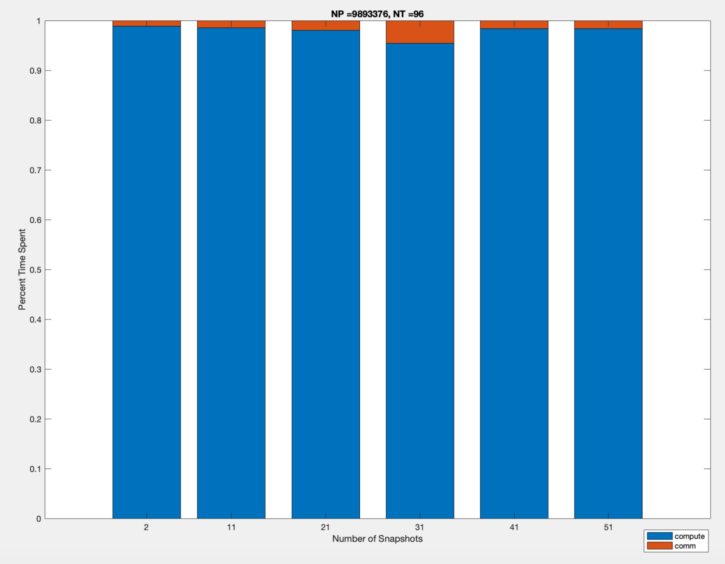
Results - Weak Scaling - $N_p/N_t = 1e6$



Results - N_s Parameter - $N_p = 1e7, N_t = 96$



Results - N_s Parameter - $N_p = 1e7, N_t = 96$



Conclusions

Scaling

- Strong Scaling - Good as expected.
- Weak Scaling - Also good, communication from N_s not that bad.
- Discrepeancy in s estimation - Gustafson's vs Amdahl's Law

Parallelizing PMC

- Good results and scalable - Speed up slowest part of algorithm.
- Data collection overhead not bad and could be made even smaller.

Questions?