Replica-Exchange Molecular Dynamics on Hadoop

18.337 Final Report

Zachary W. Ulissi

December 16, 2011

1 Background

Replica exchange molecular dynamics (REMD) is a commonly used technique to accelerate sampling rates of molecular dynamics simulations by performing a number of parallel replica simulations at different temperatures (see Figure 1). Periodically, the temperatures between pairs of replica simulations are switched with a probability

$$p = \min\left(1, e^{(E_i - E_j)\left(\frac{1}{kT_i} - \frac{1}{kT_j}\right)}\right).$$

In most implementations, swaps are only considered for simulations with the nearest temperature. If the higher temperature simulation has lower energy than the lower temperature simulation, the exchange is automatically accepted. However, there is also a chance for the exchange even if this is not the case (the exponential term). These concept are similar to Monte-Carlo based optimization techniques, and allow for the efficient sampling of the simulation potential energy surface without getting trapped in local minima (see Figure 2).

This problem is highly parallel and hierarchical in nature: each replica can be simulated using a number of processors, and each replica is independent until replicas need to be exchanged. The optimal number of processors for a small MD system can vary from O(1) to O(100), and the number of replicas is usually > 10 (setting the number and distribution of temperatures determines the probability of exchanges being accepted, and is thus a simulation size dependent problem).

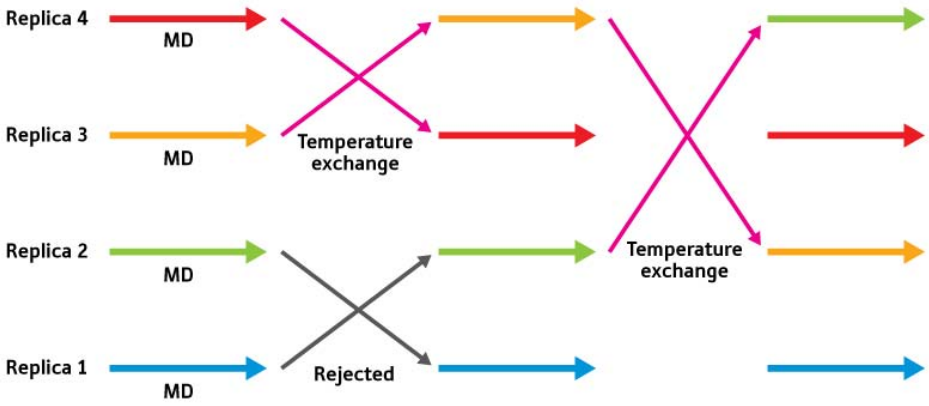

Figure 1: Illustration of the parallel REMD algorithm (reproduced from http://www.rikenresearch.riken.jp/eng/frontline/6290).

A number of implementations currently exist for performing REMD, with most of the production MD codes offering a simple scripting interface. In addition, a number of papers have been published proposing complicated manager/slave systems for performing REMD. However, these solutions are not amenable to large heterogeneous computing environments, such as those used by the distributed Folding@Home effort. Instead, this project enables the REMD simulations using the Apache Hadoop implementation of the Map/Reduce framework, which separates the simulation design from the underlying simulation framework. Hadoop handles all distributed input/output and load balancing. Furthermore, using Hadoop allows for the simple handling of hardware and software failures, which become increasingly disruptive with large-scale MD simulations.

2 Implementation

The REMD algorithm was implemented for Hadoop using the streaming Hadoop interface. The streaming interface for Hadoop allows Hadoop programs to be written using simple console input and output. The simulation was implemented using Python wrappers to implement the map and reduce steps. For the core MD simulations, the highly optimized academic MD software NAMD was used. The essential algorithm is illustrated in Figure 3. The state (and input to the Hadoop program) was the state of each temperature simulation. The state of the simulation consisted of:

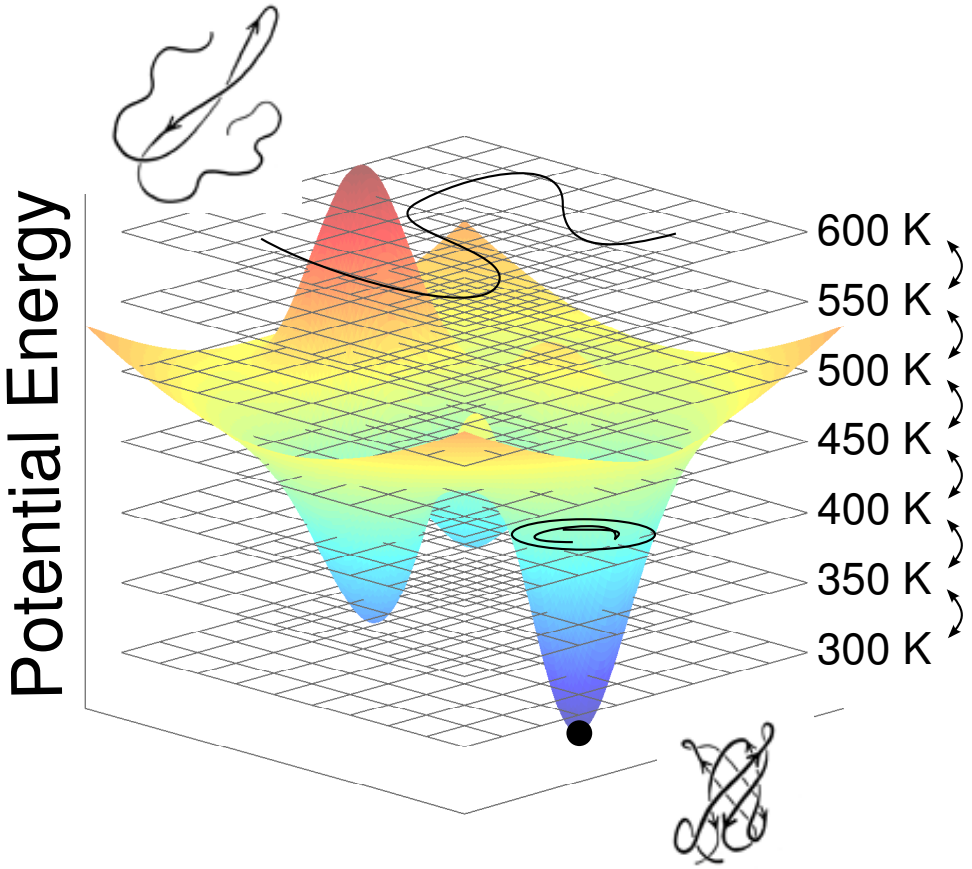

Figure 2: Illustration of simulations at multiple temperatures exploring a potential energy surface. Higher temperature simulations can cross potential barriers more quickly, and when a lower energy region is found these results are transferred to lower energy simulations.

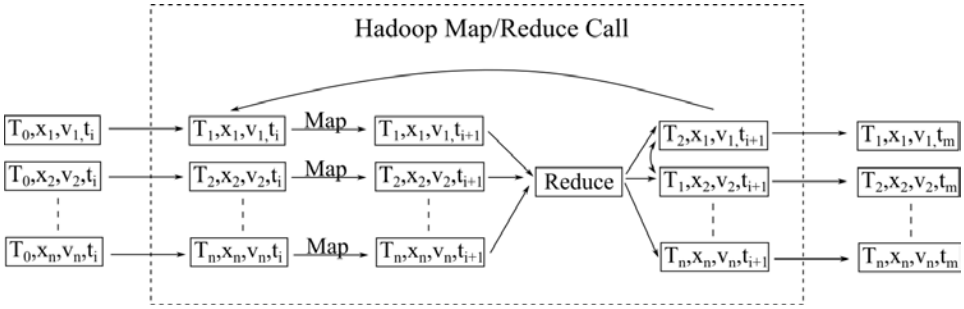

Figure 3: Illustration of REMD simulations in the Map/Reduce framework.

- The setpoint temperature of the simulation
- The energy of the simulation
- The coordinates of each atom
- The velocities of each atom

Each line of input/output corresponded to all of the state information for a single temperature. A simple python program was written to initialize the REMD using an initial guess for the atomic coordinates. A full REMD simulation thus consisted of a series of map/reduce calls.

During the Map step, each temperature simulation was progressed by 100,000 MD time steps (approximately 100 ps). The mapping program was written as a python wrapper for the NAMD MD program. For each temperature, the current coordinates and velocities were written to the appropriate NAMD input files and NAMD called through the shell. The resulting output files were then read, and the updated state sent as output. Since NAMD was required for the simulation, it was packaged with every map/reduce call as a 2MB executable.

During the reduce step, the updated states were read and, for each set of neighboring temperature simulations, the temperatures were exchanged according to the selection criteria listed above. The updated simulation states and temperatures were then emmitted as output, which could then be used for another map/reduce call to further progress the REMD simulation.

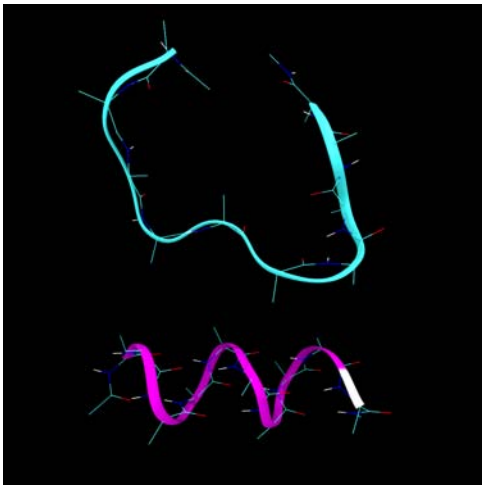

Figure 4: Example used for simulation testing: the unfolded and folded deca-alanin helix.

3 Results and Performance

The Hadoop-based REMD method was tested using a simple example simulation: the folding of a deca-alanin helix, illustrated in Figure 4 in both its folded and unfolded states. The starting configuration was deca-alanin in an unfolded state, and REMD was performed using 50 temperatures from 300 K to 800 K (i.e. 10 degree increments). The simulation was carried out on 25-node and 50-node clusters on Amazon EC2.

After 15 calls of the Hadoop-based REMD code, the simulation had captured the native state at a high temperature, illustrated in Figure 5, and upon further calls this low-energy conformation was passed to lower temperatures.

The performance scaling of the solution was investigated on a 25-node Amazon EC2 Hadoop cluster, shown in Figure 6. Linear performance scaling was observed between the usage of 1 and 25 nodes. Using 50 map tasks (twice as many as the number of nodes) resulted in nearly the same performance as using 25 tasks, suggesting that load-balancing was not an limitation on performance. Specifying more map tasks than the number of simulations resulted in increased simulation times, even though the extra map tasks had no work to perform.

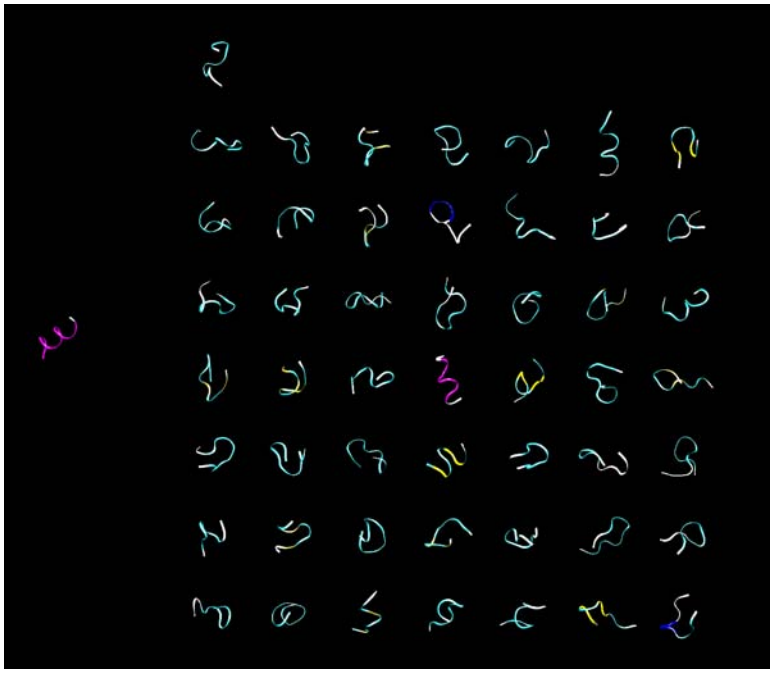

Figure 5: Simulation state after 15 calls of the Hadoop map/reduce REMD program. The native state is shown to the left, and the state for each temperature is shown on the right, starting from 300 K in the lower right-hand corner.

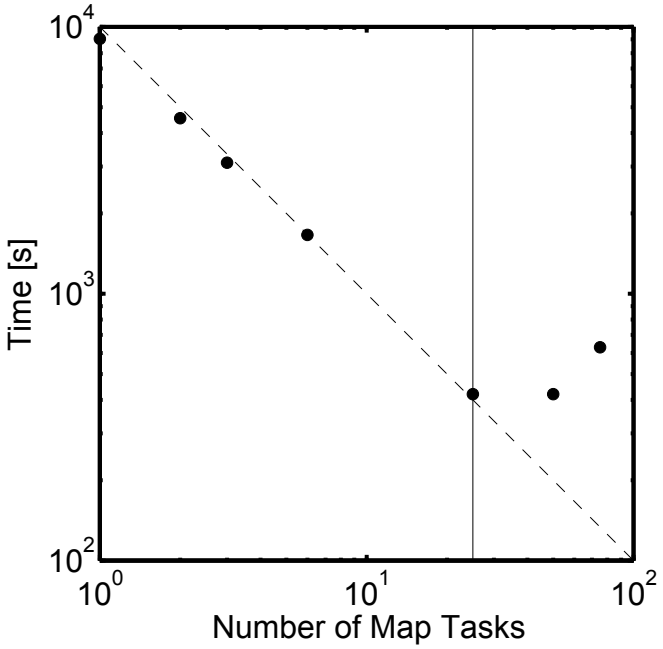

Figure 6: Scaling of the REMD code on a 25-node Amazon EC2 cluster.

4 Conclusion and Future Work

This project was successful in developing a Hadoop-based REMD simulation code. However, a number of factors suggest that this approach will not be adopted for any significant simulations in the near future:

The startup time and general overhead of the Hadoop framework was significant. To achieve reasonable efficiency in usage, the number of map/reduce steps had to be minimized. A reduced number of temperature exchanges limited the rate at which the potential surface could be explored. Furthermore, the time for a low-energy structure identified at high temperature to propagate to low temperature simulations is directly related to the time between exchange steps. Using 50 temperatures, at least 50 map/reduce steps would be necessary. The overhead of each map/reduce call could be perhaps be reduced by writing the wrappers in Java and integrated the program directly into the Hadoop framework.

The broad availability of computation time on NSF and DOE funded supercomputers to academic research labs generally means that a map/reduce framework is not necessary since all machines are collocated and tightly coupled. Using REMD could perhaps be useful on large distributed problems, but each distributed computer needs to have Hadoop installed. Map/reduce based REMD may be most appropriate for private companies that wish to do large-scale simulations on heterogeneous clusters (i.e. drug companies wishing to do virtual drug screening).

A number of possibilities exist for future work. Reducing the overhead of the map/reduce calls would increase the overall simultion efficiency. Secondly, communication time during the accumulation steps in the map/reduce call could be reduced by using binary representations of the coordinate and velocity lists (rather than the character-based ones). Finally, the reduce step took a significant fraction of the computation time even though it was one of the simplest steps, suggesting that communication between the map and reduce steps was a limiting factor.

5 Appendix: Simulation Code

5.1 mapper.py

```
#!/usr/bin/python
import sys
import subprocess
import os

enindex=11

# input comes from STDIN (standard input)
```

```
for line in sys.stdin:
    z=line.split(' NEWFIELD ')
    newtemp=float(z[0])
    oldtemp=float(z[1])
    coord=z[2].replace('NEWLINE','\n')
    vel=z[3].replace('NEWLINE','\n')

    #Set the necessary NAMD configuration inputs
    f=open('set_temp.tcl','w')
    f.write('set temp %f \n' % newtemp)
    f.write('set rsv %f \n' % (newtemp/oldtemp))
    f.write('set oname alanin_%d\n' % newtemp)
    f.write('set ncoor newcoor_%d\n' % newtemp)
    f.write('set nvel newcoor_%d\n' % newtemp)
    f.close()

    #Write the coordinates
    f=open('newcoor_%d' % newtemp,'w')
    f.write(coord)
    f.close()

    #Write the velocity file
    f=open('newvel_%d' % newtemp,'w')
    f.write(vel)
    f.close()

    #Call NAMD, using python 2.7
    #subprocess.check_output(["namd2 alanin.namd > alanin.namd_%d.log" % newtemp],shell=True)
    subprocess.call(["chmod +x namd2"],shell=True)
    subprocess.call(["./namd2 alanin.namd > alanin.namd_%d.log" % newtemp],shell=True)

    #Clean-up the temporary coordinates and velocities
    os.remove('newcoor_%d' % newtemp)
    os.remove('newvel_%d' % newtemp)

    #Parse the log-file and get the final energy
    f=open("alanin.namd_%d.log" % newtemp,'r')
    for line in f:
sline=line.split()
if line!='\n':
            if sline[0]=='ENERGY:':
                energy=float(sline[enindex])
    f.close()

    #print('%d,%f' % (newtemp,energy))

    #Read the final coordinates
    f=open('alanin_%d.coor' % newtemp,'r')
    newcoord=f.read().replace('\n','NEWLINE')
    f.close()

    #Read the final velocities
    f=open('alanin_%d.vel' % newtemp,'r')
    newvel=f.read().replace('\n','NEWLINE')
    f.close()

    #Emit the processed state info
    print(str(newtemp)+' NEWFIELD '+str(oldtemp)+' NEWFIELD '+newcoord+' NEWFIELD '+newvel+' NEWFIELD ' + str(energy))
```

5.2 reducer.py

```
#!/usr/bin/python
import sys
import subprocess
import os
import math
import random

Snewtemp=0
kb=1.38*10**(-23)

for line in sys.stdin:
    if Snewtemp==0:
#retrive the temperatures and energy
```

```
        Sz=line.split(' NEWFIELD ')
        Snewtemp=float(Sz[0])
        Soldtemp=float(Sz[1])
        Senergy=float(Sz[4])
    else:
#retrive the temperatures and energy
        Nz=line.split(' NEWFIELD ')
        Nnewtemp=float(Nz[0])
        Noldtemp=float(Nz[1])
        Nenergy=float(Nz[4])

        #Calculate beta=1/k/T for the current and saved versions
        Nb=1/Nnewtemp/kb
        Sb=1/Snewtemp/kb

        delta=(Nb-Sb)*(Senergy-Nenergy)

        #Swap configurations
        if delta<0:
            print(str(Nnewtemp)+' NEWFIELD '+str(Snewtemp)+' NEWFIELD '+Sz[2]+' NEWFIELD '+Sz[3])
            Sz=Nz
            Senergy=Nenergy
            Soldtemp=Nnewtemp
        #Swap configurations
        elif math.exp(-delta)<random.random():
            print(str(Nnewtemp)+' NEWFIELD '+str(Snewtemp)+' NEWFIELD '+Sz[2]+' NEWFIELD '+Sz[3])
            Sz=Nz
            Senergy=Nenergy
            Soldtemp=Nnewtemp
#emit without swapping
        else:
            print(str(Snewtemp)+' NEWFIELD '+str(Snewtemp)+' NEWFIELD '+Sz[2]+' NEWFIELD '+Sz[3])
            Sz=Nz
            Senergy=Nenergy
            Snewtemp=Nnewtemp
            Soldtemp=Nnewtemp

print(str(Snewtemp)+' NEWFIELD '+str(Snewtemp)+' NEWFIELD '+Sz[2]+' NEWFIELD '+Sz[3])
```

5.3 geninput.py

geninput.py sets up the input for the map/reduce simulation based on initial configuration data.

```
#!/usr/bin/python

#read initial atom coordinates
f=open('alanin.coor','r')
startpdb=f.read().replace('\n','NEWLINE')
f.close()

#read the initial atomic velocities
f=open('alanin.vel','r')
startvel=f.read().replace('\n','NEWLINE')
f.close()

#specify the temperatures for simulation
temps=[300,325,350,375,400,425,450,475,500,525,550,575,600]
oldtemp=300

towrite=''

#For each temperature, write the state
for T in range(300,800,10):
    towrite=towrite+str(T)+' NEWFIELD '+str(oldtemp)+' NEWFIELD '+startpdb+' NEWFIELD '+startvel+'\n'

#Write the output
f=open('torun','w+')
f.write(towrite)
f.close()
```

5.4 NAMD Configuration File (alanin.namd)

This is the required control file for NAMD, which loads the atomic coordinates and velocities from files written by the map/reduce call and runs the simulation. The setpoint temperature is read from the set_temp.tcl control file also written by the map/reduce call.

```
# NAMD CONFIGURATION FILE FOR DECALANIN

#Get the simulation conditions
source set_temp.tcl

# Set the atomic details
coordinates $ncoor
velocities  $nvel
#seed 12345

outputEnergies          50000

# output params
outputname $oname
binaryoutput no
#DCDfile $oname.dcd
#DCDfreq 100

# integrator params
timestep 1.0

# force field params
structure alanin.psf
parameters alanin.params
exclude scaled1-4
1-4scaling 1.0
switching on
switchdist 8.0
cutoff 12.0
pairlistdist 13.5
margin 0.0
stepspercycle 20

langevin on
langevinTemp $temp

rescalevels $rsv

#Run the simulation
run 500000
```

18.337J / 6.338J Parallel Computing

Fall 2011