CS-5802: Project

Scalable Computation of Molecular Trajectory Analysis Using Apache Spark

Submitted by:
Rasman Mubtasim Swargo

Wazih Tausif

[Project Repository]

Submitted to: Dr. Satish Puri
10th December 2024

Contents

Introduction

The objective of this project is to implement the knowledge learned from the course Introduction to Parallel Computing to solve real life problems. Parallel computing is used extensively for molecular dynamics simulations and post processing of the results. In mechanical engineering, the molecular behavior of vapor at liquid surface is of great importance and interest, which is very difficult to measure experimentally. Hence, molecular dynamics simulations are used to study these problems. One of the challenges in this study is post processing simulation results. A typical simulation may contain from hundreds of thousands to millions of molecules. This increases the output data file and complicates post processing processes. We have identified such a case, where post processing of hundreds of thousands of molecules is necessary to analyze the simulation results. Typically, the sequential code written by one of the project members takes more than three hours to do the processing. Our objective for this project is to write a parallel code using Apache PySpark, then benchmark its performance using the knowledge learned in this course.

Background

In engineering research, many of the problems lie in microscopic behavior of the systems. With sophisticated equipment, such as electron microscopes and x-ray diffraction techniques, we can probe into the microscopic world. However, there are many properties and phenomena which cannot be observed directly by experiments. Sometimes, the analysis can be performed, but at a prohibitively large cost or with less than desirable accuracy. In these cases, computer simulations can be performed to do the analysis.

One of the most important computer simulation techniques to perform investigations at microscopic levels is molecular dynamics (MD) simulations. The key idea of MD simulations is very simple. It treats each molecule as a classical particle which has interaction (by force) with other particles. Then, numerical integration is performed to integrate Newton's second law of motion (F=ma) to generate trajectories of the particle. These trajectories provide us with a clear view of how the particles are moving with time, just like watching a real-life movement. From the trajectory of each molecule, we can then calculate macroscopic properties (pressure, temperature etc.) using the help of statistical mechanics. MD simulations are used by almost every engineering and scientific discipline. From physicists who study quantum mechanics to mechanical engineers who study boiling and condensation, it is also used by biologists to study the movement of protein in cells.

Early molecular dynamics simulations were performed with hundreds or thousands of very simple particles, such as monoatomic gases. With time, complicated models were developed to simulate complex molecules, such as water and organic matter. With time, the scale of the simulations increased to hundreds of thousands of molecules in each simulation. However, this also brought a unique challenge- that is computing power. Sophisticated simulations require a lot of computing power and soon, researchers were using multiple processors to run simulations, thus introducing

parallel computing to MD simulations. But writing efficient and effective parallel computing programs require unique set of skills that are only available to skilled computer scientists. Therefore, soon MD simulations became a problem for computer scientists, as equal it is to material or mechanical engineers. With the help of computer scientists, who can develop parallel computation techniques, and with engineers from other disciplines who can develop physics for simulations- current MD simulations can be highly parallelized, enabling researchers to run simulations on supercomputers.

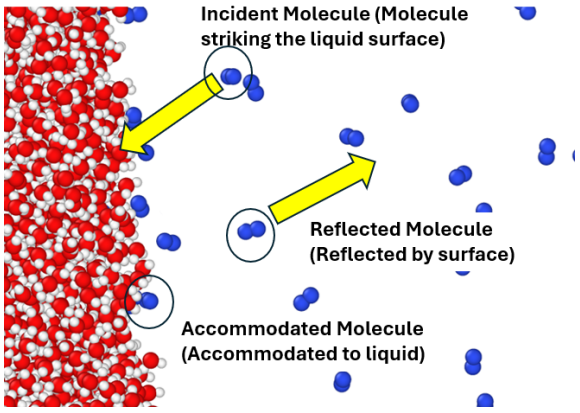
One of the most used MD simulation programs is called LAMMPS[1] (Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS). Developed at the Sandia National Laboratory, LAMMPS uses message passing interface (MPI) to parallelize the simulations. The simulation domain is spatially decomposed into smaller chunks, which are then assigned to individual processors. Using MPI parallelization, the processors can effectively run each chunk of the simulation domain and exchange information with one another, essentially running the simulation at once. Current simulations can run with millions of molecules using thousands of processors to perform sophisticated investigations.

MD simulations simply create trajectories of individual molecules. The simulations themselves can run very efficiently using parallel computation, thanks to the computer scientists. However, the simulations typically produce very rudimentary results, which requires significant amount of post processing. Post processing is as important as running the MD simulations itself. For example, some MD simulations can produce tens of gigabytes of data, which need to be post processed. Some of the post processing is very simple, involving simple sorting. Other post processing techniques simply use trajectories to generate images and videos of the molecules. On the other hand, some post processing is very complicated, and it takes a significant amount of time to process the data. Therefore, again, parallel computing can be implemented to post process the data.

Unlike MD simulation, which is universally designed and can be used by researchers in any field, post processing is highly specific to one's research, even varying within the same laboratory. Therefore, there is no universal post processing software, and it is up to the general researcher to prepare his own post processing code, who in most cases are not computer scientists, thereby not possessing the skillset to write efficient parallel programs. Our problem for the project involves a similar problem- to process a huge amount of data generated by molecular dynamics simulations to obtain microscopic properties of the system. Using the lessons from Introduction to Parallel Computing course, we intend to write parallel code to process data from MD simulation. A new parallel program with new algorithm will be developed for this project, and then compared with a sequential code that's already developed by one of the group members to realize performance improvements.

Problem Description

In a system with liquid and vapor, for example, water in a water bottle, or hot water in a cup, there is a liquid-vapor interface, or simply the liquid surface. At the liquid surface, some of the vapor molecules strike the liquid surface, which are called incident molecules. Some of the incident molecules get accommodated (in simple terms, they get absorbed by the liquid and become part of the liquid) to the liquid. The other incident molecules are reflected by the liquid surface. In summary, vapor molecules incident on a liquid surface may either get reflected by the surface or get accommodated to the liquid. The possibility of phase change (boiling or evaporation) directly depends on this phenomenon. To calculate what's the probability of an incident molecule getting accommodated, we can take the ratio of accommodated molecules over the total number of incident molecules. This quantity is known as the mass accommodation coefficient (MAC)[2].



This property is very difficult to analyze experimentally, because it requires the count of how many molecules strike the liquid surface and how many of them are accommodated to the liquid. This is where MD simulation becomes very useful. MD simulation keeps track of individual molecules. By processing the data generated by MD simulation, we can find out which molecules are striking the surface, and how many of them are getting accommodated.

$$MAC = \frac{Number\ of\ accommodated\ molecules}{Number\ of\ incident\ molecules}$$

The reasons to implement parallel computing to do the processing are:

1. Data files are large: For a two-nanosecond runtime, data files containing molecule trajectories can be 2-3 gigabytes. For good statistics, sometimes 10/15 nanosecond runtime is necessary. Due to the nature of processing, the runtime does not increase linearly with data file size, rather it becomes exponentially larger and takes longer and longer time to process.
2. The data files contain information (position with time) about all the vapor molecules. The processing code must identify which vapor molecules are incident (striking the liquid

surface) and how many of the incident molecules are getting accommodated. In a simulation with 100,000 molecules, it is possible that the incident count is less than 500. But this is not known in advance and all molecules must be processed. Hence, parallel programs are suitable for this type of processing.

Data File Structure

A data file contains the trajectory of each molecule at discrete timesteps. There are two columns in a data file, first one contains the unique molecule ID and the second one contains it's position at that time. A sample data file is attached for reference. This file is output from MD simulation. The rows containing NaN values represent the beginning and end of a timestep. By identifying the NaN values, we can determine when a timestep begins or ends.

For example, in the picture attached, in the first timestep- there are position data for 18 molecules, each with their unique ID. The actual data files used for calculations contain around 100,000 molecules.

```
1   ID x
2   NaN NaN
3   85 -5.87273
4   136 -5.86439
5   199 70.131
6   226 -5.34004
7   277 72.4391
8   301 69.975
9   388 74.624
10  412 115.264
11  472 -5.10905
12  490 -3.50853
13  538 74.133
14  592 77.743
15  613 74.0822
16  649 70.8455
17  694 73.582
18  724 -4.47199999999998
19  802 72.1899
20  838 -3.06099999999998
21  NaN NaN
```

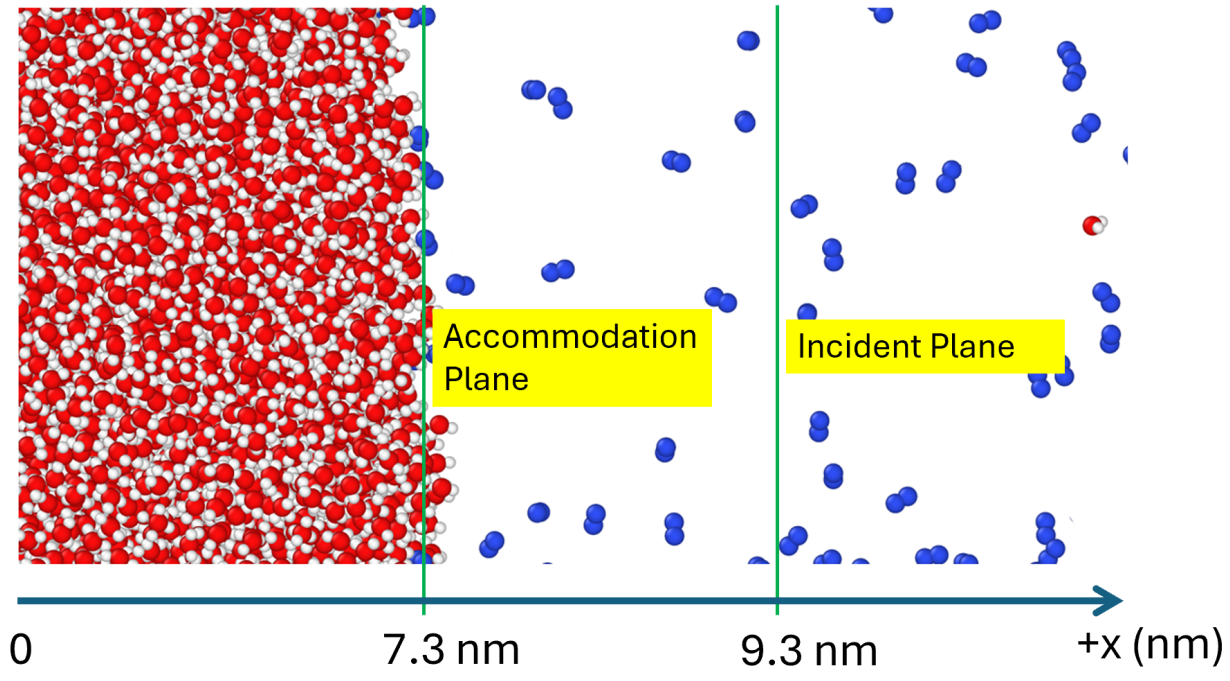
MAC Calculation Method

Calculation of MAC accurately is difficult. One of the ways to accurately calculate MAC is the two plane method[3]. Two planes are defined to determine which molecules are incident, and which molecules are accommodated. Two values are counted, number of incident molecules and number of accommodated molecules.

The logic is discussed below for right plane (A figure is attached for reference):

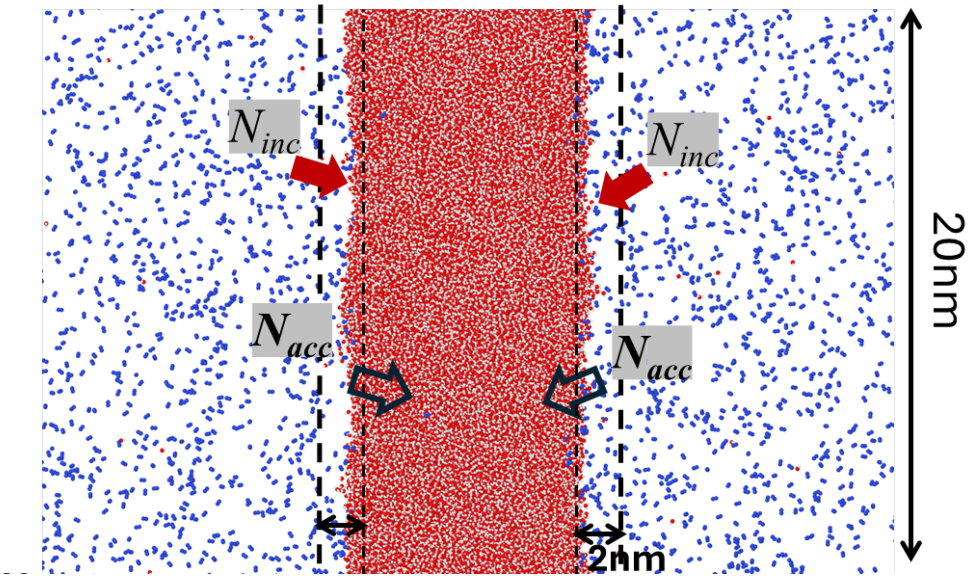
1. If a molecule crosses the incident plane from right to left (If initial position is 10nm and next position is 8nm), the incident count is increased by 1. Molecules exiting the plane will not be counted.
2. If a incident molecule crosses the accommodation plane from right to left, accommodation count is increased by 1. For a molecule to accommodate, it must be incident first. Molecules exiting the accommodation plane will not be counted.
3. If a molecule exits an accommodation plane but not the incident plane, then reenters accommodation plane, it will not be counted as a new accommodated molecule.
4. If a molecule exits the incident plane, then reenters the incident plane, incident count is increased by 1.

Once we have the number of incident and accommodated molecules, MAC is simply the ratio of accommodated and incident molecules.



Additional Constraints:

1. The actual simulation is symmetric, so there are two surfaces of the liquid. Similar analysis is performed for both surfaces. The figure below shows an actual simulation box.



2. Due to the nature of the simulation setup, there is "periodic boundary condition" in all three directions. In PBC, if a molecule exits the simulation box from left, it will reappear from the right and vice versa. The code must take account of this effect may give false results if it is not taken into account properly.
3. Cutoff time: A data file contains trajectory for 2ns. The data are dumped every 40fs (femtoseconds), so- there are data for 50,000 timesteps in the file. The code takes a molecule, and starting from zero, it goes through all 50,000 timesteps to check when the molecule crossed the incident and accommodation plane. However, if a molecule is incident at the last timestep, it may happen that the molecule could actually be accommodated if given enough time. This is a limitation imposed by finite data file size and time. To overcome this, we need to let the incident molecules either get accommodated, or reflected. This is why a cutoff time is necessary. To do this, we analyzed the velocity distribution function (which follows a normal distribution) and calculated that 0.2ns should be optimum. So, incident molecule count is stopped after 1.8ns (45,000 timesteps) and the rest of 0.2ns (5000 timesteps), we allow the incident molecules to be either accommodated or reflected.

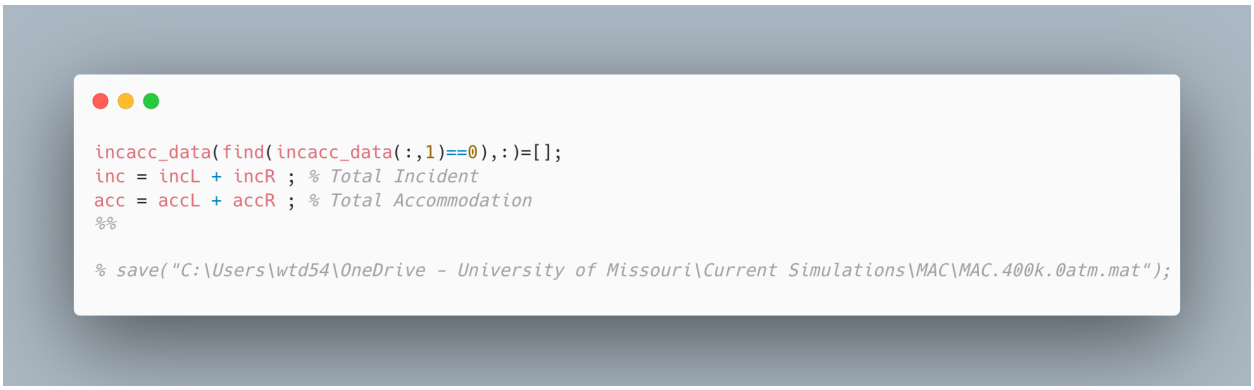
Sequential Code

A sequential code was written by one of the members in MATLAB.

```
clear all
clc
format shortG

%% File Reading Details.
run_time = 2; %total run time in nanosecond
cutoff_time = 0.2; %cutoff time in nanosecond
incident_time = run_time - cutoff_time; %incident run time in nanosecond
timestep_size = 2; %fs
dump_interval = 20; %dump every this interval
run_timestep = incident_time/(timestep_size*10^-6);
data_cutoff = round(run_timestep/dump_interval)+3;

liqslab= 75;
total_length = 300;
box_length = liqslab+(total_length-liqslab)/2;
box_width = 200;

%% Data file read

dir = 'D:\Courses\24-Fall\Intro To Parallel\Project\';
myfilename = 'myData.txt';
path = strcat(dir,myfilename);
opts = detectImportOptions(path,'FileType','text','Range',"A:B");
mydata = readtable(path,opts);
mydatanew(:,1)=mydata(:,1);
mydatanew(:,2)=mydata(:,2);
mymat = table2array(mydatanew);

clearvars mydata mydatanew
testpath = mymat(find(mymat(:,1)==117427),2);
%%
cutoff_index=find(isnan(mymat(:,2)));
cut_line=cutoff_index(data_cutoff,1);
incidenttmat = mymat(1:cut_line,:);
% writematrix(incidenttmat,'chopped.txt','Delimiter',' ');
```

```
%%
mymat(find(isnan(mymat(:,2))),:) = [];
incidenttmat(find(isnan(incidenttmat(:,2))),:) = [];
totalID=mymat(:,1);
ID = incidenttmat(:,1);
x_inc = incidenttmat(:,2);
x_acc = mymat(:,2);

InCdex=find(isnan(x_acc(:,1)));
Accdex=find(isnan(x_acc(:,1)));
% clearvars mymat

%% Incident and Accommodation Plane Locations

xlgRight = 73.2186 ; %Right Accommodation Plane location
xplaneRight = xlgRight + 20 ; %Right Incident Plane location
xlgLeft = 2.709 ; %Left Accommodation Plane location
xplaneLeft = xlgLeft - 20 ; %Left Incident Plane location
IDvapor = unique(ID,'rows'); %remove ID duplicates
accL = 0;
accR = 0;
incL = 0;
incR = 0;
incacc_data=zeros(length(IDvapor),5);
%%
testerid=find(IDvapor==117427);
```

```
%% Determination of Incident and Accommodated Molecules

for i = 1:length(IDvapor)
    xpath_inc = x_inc(find(ID==IDvapor(i)));
    % xpath_acc=mymat(find(mymat(:,1)==117427),2);
    xpath_acc = x_acc(find(totalID==IDvapor(i)));

    inc_numberL=0; inc_numberR=0;
    acc_numberL=0; acc_numberR=0;
    xpathlength_inc=length(xpath_inc);
    xpathlength_acc=length(xpath_acc);
    xcrossL=zeros(xpathlength_acc,1);
    xcrossR=xcrossL;

    % Right Incident
    for j=1:xpathlength_inc-1
        if xpath_inc(j)>xplaneRight && xpath_inc(j+1)<xplaneRight
           if abs(xpath_inc(j)-xpath_inc(j+1))<290
                incR = incR + 1
                inc_numberR = inc_numberR+1;
                xcrossR(j)=1;
           end
        end

        % Left Incidence
        if xpath_inc(j)<xplaneLeft && xpath_inc(j+1)>xplaneLeft
           if abs(xpath_inc(j+1)-xpath_inc(j))<290
                incL = incL + 1
                inc_numberL=inc_numberL+1;
                xcrossL(j)=1;
           end
        end
    end
    for j=1:xpathlength_acc-1
    % Right Accommodation Path
        if xpath_acc(j)>xlgRight && xpath_acc(j+1)<xlgRight
           if abs(xpath_acc(j)-xpath_acc(j+1))<290
                xcrossR(j)=2;
           end
        end

    % Left Accommodation Path
        if xpath_acc(j)<xlgLeft && xpath_acc(j+1)>xlgLeft
           if abs(xpath_acc(j+1)-xpath_acc(j))<290
                xcrossL(j+1)=2;
           end
        end
    end
        xcrossR(find(xcrossR==0))=[];
        xcrossL(find(xcrossL==0))=[];
    % Right Accommodation
        for k=1:length(xcrossR)-1
            if xcrossR(k)==1 && xcrossR(k+1)==2
                accR = accR + 1
                acc_numberR = acc_numberR + 1;
            end
        end
     % Left Accommodation
        for k=1:length(xcrossL)-1
            if xcrossL(k)==1 && xcrossL(k+1)==2
                    accL = accL +1
                    acc_numberL = acc_numberL + 1;
            end
        end


    incacc_data(i,1)=IDvapor(i,1);
    incacc_data(i,2)=inc_numberL;
    incacc_data(i,3)=inc_numberR;
    incacc_data(i,4)=acc_numberL;
    incacc_data(i,5)=acc_numberR;

end
```



```
incacc_data(find(incacc_data(:,1)==0),:)=[];
inc = incL + incR ; % Total Incident
acc = accL + accR ; % Total Accommodation
%%

% save("C:\Users\wtd54\OneDrive - University of Missouri\Current Simulations\MAC\MAC.400k.0atm.mat");
```

Parallel Code using PySpark

```python
from pyspark import SparkContext, SparkConf
import os
import time

def validate_and_map(row):
    if len(row) >= 3:
        return (row[1], (row[0], row[2]))
    else:
        return None

def inside_accomodation(x_position, left, right):
    return float(x_position) > left and float(x_position) < right

def inside_incident(x_position, left, right):
    return float(x_position) > left and float(x_position) < right

def find_accomodation_or_incident(id, values, cutoff_index, accom_left, accom_right, inc_left,
inc_right):
    if not values:
        return 0, 0

    sorted_values = sorted(values, key=lambda x: int(x[0]))
    accomodation_count = 0
    incident_count = 0
    current_state = -1

    for line, x_position in sorted_values:
        if inside_accomodation(x_position, accom_left, accom_right):
            if current_state == 1:
                accomodation_count += 1
                current_state = 2
        elif inside_incident(x_position, inc_left, inc_right):
            if current_state == 0 and int(line) <= cutoff_index:
                incident_count += 1
                current_state = 1
        else:
            current_state = 0

    return incident_count, accomodation_count
```

```python
if __name__ == "__main__":
    start_time = time.time()
    printable_string = ''

    # File paths
    input_file_path = "myData (1).txt"
    input_with_line_numbers_file_path = "input_with_line_numbers.txt"
    output_file_path = "sbatch_output_" + input_file_path

    set_count_to_deduct_for_incident = 500

    # Add line numbers to input file
    with open(input_file_path, 'r') as infile, open(input_with_line_numbers_file_path, 'w') as outfile:
        for i, line in enumerate(infile):
            outfile.write(f"{i + 1} {line}")

    # Read lines and find NaN indices
    with open(input_with_line_numbers_file_path, 'r') as infile:
        lines = infile.readlines()

    nan_indices = [i for i, line in enumerate(lines) if 'NaN' in line]
    cutoff_index = nan_indices[0] if nan_indices else len(lines)

    if set_count_to_deduct_for_incident > 0 and len(nan_indices) >= set_count_to_deduct_for_incident:
        cutoff_index = nan_indices[-set_count_to_deduct_for_incident]

    # Configure Spark
    conf = SparkConf().setAppName("MolecularDynamics").setMaster("local")
    sc = SparkContext(conf=conf)

    # Process data with Spark
    accomodation_rdd = sc.textFile(input_with_line_numbers_file_path, minPartitions=64)

    header = accomodation_rdd.first()
    accomodation_data_rdd = accomodation_rdd.filter(lambda line: line != header).map(lambda line:
line.split())
    accomodation_cleaned_rdd = accomodation_data_rdd.filter(lambda row: not any(value == "NaN" for value
in row))

    accomodation_key_value_rdd = accomodation_cleaned_rdd.map(validate_and_map).filter(lambda x: x is not
None)
    accomodation_grouped_rdd = accomodation_key_value_rdd.groupByKey()

    # Define planes
    accomodation_plane_Right = 73.2186
    accomodation_plane_Left = 2.709
    incident_plane_Right = accomodation_plane_Right + 20
    incident_plane_Left = accomodation_plane_Left - 20

    # Compute incidents and accommodations
    accomodation_rdd = accomodation_grouped_rdd.map(
        lambda x: [x[0], find_accomodation_or_incident(
            x[0], list(x[1]), cutoff_index,
            accomodation_plane_Left, accomodation_plane_Right,
            incident_plane_Left, incident_plane_Right)
        ]
    )

    total_counts = accomodation_rdd.map(lambda x: x[1]).reduce(lambda a, b: (a[0] + b[0], a[1] + b[1]))
    total_incidents, total_accommodations = total_counts
    overall_ratio = total_accommodations / total_incidents if total_incidents != 0 else "Infinity"

    # Display results
    printable_string += f"Total Accommodations: {total_accommodations}\n"
    printable_string += f"Total Incidents: {total_incidents}\n"
    printable_string += f"Overall Accommodation-to-Incident Ratio: {overall_ratio}\n"

    end_time = time.time()
    elapsed_time = end_time - start_time
    printable_string += f"Elapsed time: {elapsed_time:.6f} seconds\n"

    # Save the result to a file
    with open(output_file_path, 'w') as f:
        f.write(printable_string)

    # Stop Spark
    sc.stop()
```

Evaluation

We ran the parallel program on two files. Here are the corresponding results. All the times are denoted in seconds.

File 1

File Size: 149.1 MB

Sequential Code Run Time: 240.839468

SLURM Script Runtime

RAM: 32GB

CPU Count	Runtime 1	Runtime 2	Runtime 3
32	63.743285	64.923132	64.010050
16	63.169284	63.172451	63.131335
8	64.955925	62.623662	62.632954

Jupyter Notebook Runtime

RAM: 32GB

CPU Count	Runtime 1	Runtime 2	Runtime 3
8	52.981259	52.839927	52.709686
4	53.759185	52.776652	53.605567

Databricks Notebook

RAM: 15GB

Node: 1

Runtime 1: 260.872363

Runtime 2: 255.398457

Runtime 3: 253.348563

File 2

File Size: 3056.7 MB

Sequential Code Run Time: 10796.375684

SLURM Script Runtime

RAM: 32GB

CPU Count	Runtime 1	Runtime 2	Runtime 3
32	2842.952498	2817.250066	2897.034255
16	2895.571187	2817.491315	2793.015825
8	2854.846730	2815.657541	2893.429747

Jupyter Notebook Runtime

RAM: 32GB

CPU Count	Runtime 1	Runtime 2	Runtime 3
8	2362.964152	2350.852000	2353.838679
4	2356.660744	2397.659651	2390.808288

Databricks Notebook

RAM: 15GB

Node: 1

Runtime 1: 11299.345910

Runtime 2: 11390.771182

Runtime 3: 11634.907390

Speedup and Efficiency Calculations:

Speedup and efficiency are calculated using the following formulas:

1. **Speedup** = $\frac{T_{\text{sequential}}}{T_{\text{parallel}}}$
   - $T_{\text{sequential}}$: Sequential code runtime.
   - T_{parallel}: Parallel code runtime.
2. **Efficiency** = $\frac{\text{Speedup}}{\text{CPU Count}}$
   - CPU Count: Number of processors used.

For File 1 (149.1 MB):

Sequential Runtime = 240.839468 seconds

SLURM Script Runtime:

CPU Count	Average Runtime	Speedup	Efficiency
32	64.225489	3.75	11.72%
16	63.157690	3.81	23.83%
8	63.404180	3.80	47.50%

Jupyter Notebook Runtime:

CPU Count	Average Runtime	Speedup	Efficiency
8	52.843624	4.56	57.04%
4	53.380468	4.51	112.75%

Databricks Notebook:

- Single Node Runtime (Average): 256.539 seconds
- Speedup = $\frac{240.839}{256.539} = 0.94$
- Efficiency = $\frac{0.94}{1} = 94\%$

For File 2 (3056.7 MB):

Sequential Runtime = 10796.375684 seconds

SLURM Script Runtime:

CPU Count	Average Runtime	Speedup	Efficiency
32	2852.412273	3.79	11.84%
16	2835.359442	3.81	23.81%
8	2854.644673	3.78	47.25%

Jupyter Notebook Runtime:

CPU Count	Average Runtime	Speedup	Efficiency
8	2355.884277	4.58	57.25%
4	2381.709561	4.53	113.25%

Databricks Notebook:

- Single Node Runtime (Average): 11408.341 seconds
- Speedup = $\frac{10796.375}{11408.341} = 0.95$
- Efficiency = $\frac{0.95}{1} = 95\%$

Discussion

One of the primary challenges we encountered was the inability to fully leverage Apache Spark's capabilities due to the lack of an HDFS cluster. We attempted various approaches to address this issue. Initially, multiple nodes were acquired using Mill; however, as the file system lacked Hadoop support, all nodes, except the master, remained idle. We also explored cloud solutions like AWS EMR (Elastic MapReduce) and Azure HDInsight, but their paid nature made them impractical within our project constraints. Ultimately, we ran the program on Databricks Community Edition, which provided only a single node with 15 GB of RAM. Consequently, the runtime for this setup was even longer than the sequential code.

Despite the resource limitations, code optimizations significantly improved performance. Specifically, these optimizations reduced the runtime by a factor of four compared to the unoptimized version of the code. Importantly, the correctness of the optimized code was verified by producing outputs identical to those of the sequential implementation. As the program is not computation-heavy increasing CPU count did not increase the speedup.

This project successfully demonstrates the potential of Apache Spark’s distributed memory mechanism for molecular dynamics simulation data processing. Although resource constraints limited our ability to achieve the desired speedup, the findings indicate that substantial time savings can be realized in practice when appropriate infrastructure is available.

References

1. LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales, A. P. Thompson, H. M. Aktulga, R. Berger, D. S. Bolintineanu, W. M. Brown, P. S. Crozier, P. J. in 't Veld, A. Kohlmeyer, S. G. Moore, T. D. Nguyen, R. Shan, M. J. Stevens, J. Tranchida, C. Trott, S. J. Plimpton, Comp Phys Comm, 271 (2022) 10817.
2. J. Phys. Chem. A 2012, 116, 44, 10810–10825
3. Z. Liang , P. Keblinski , Molecular simulation of steady-state evaporation and condensation in the presence of a non-condensable gas, J. Chem. Phys. 148 (2018) 064708 .