Fault-Tolerant Parallel Analysis of Millisecond-scale Molecular Dynamics Trajectories

Tiankai Tu

D. E. Shaw Research
Anton: A Special-Purpose Parallel Machine for MD Simulations
“Routine” Data Analysis Tools

- Data validation: detect corrupted simulation output data via checksums
- Summary statistics: rmsd, bond length, dihedral angles, electron density map
- Event detection: ion permeation, conformational change, binding events
- Data clustering: k-mean, EM-based Procrustes analysis
## Millisecond-Scale MD Trajectories

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulation length:</td>
<td>$1 \times 10^{-3}$ s</td>
</tr>
<tr>
<td>Output interval:</td>
<td>$100 \times 10^{-12}$ s</td>
</tr>
<tr>
<td>Total output frames:</td>
<td>10 M frames</td>
</tr>
<tr>
<td>A biomolecular system:</td>
<td>25 K atoms</td>
</tr>
<tr>
<td>Position and velocity:</td>
<td>24 bytes/atom</td>
</tr>
<tr>
<td>Frame size:</td>
<td>0.6 MB/frame</td>
</tr>
</tbody>
</table>

A Single Trajectory size is about 6 TB
Performance Bottleneck in Analysis

Millissecond-scale trajectory size: 6 TB
Local disk read bandwidth: 100 MB/s
Data access time: 16 hours
Analysis time: O(n)
Total time: day(s)

Sequential analysis lack the computational, memory, and I/O capabilities!
Support for “Routine” Data Analysis

<table>
<thead>
<tr>
<th>Data validation</th>
<th>Summary statistics</th>
<th>Event detection</th>
<th>Data Clustering</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel programming model</td>
<td>Fault tolerance</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parallel I/O (read in particular)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Part I: How to Analyze MD Trajectories in Parallel?
Infrastructure for Data Analysis

- Data validation
- Summary statistics
- Event detection
- Data Clustering

- Parallel programming model
- Parallel I/O (read in particular)
- Fault tolerance
An Event Detection Example: Ion Permeation
A Hypothetic Trajectory

20,000 atoms in total; two ions of interest

Ion A

Ion B
Ion State Transition

- Above channel
- Inside channel
- Below channel
- Into channel from above
- Into channel from below

S

11
Sequential Analysis

- Maintain a main-memory resident data structure to record ion states and positions
- Process frames in ascending simulated physical time order

**Strong inter-frame data dependence:**
Analysis (state transition) tightly coupled with data acquisition (positions of ions)
A Different View of Analysis

- Specify which frames to be accessed
- Decouple data acquisition from data analysis

Trajectory definition

Stage 1: Per-frame data acquisition

Stage 2: Cross-frame data analysis
Trajectory Definition

Every other frame in the trajectory

- Ion A
- Ion B
Per-frame Data Acquisition (stage 1)
Cross-frame Data Analysis (stage 2)

Analyze ion A on P0 and ion B on P1

![Graph showing analysis of ions A and B on different stages](image-url)
A Parallel Programmer’s Perspective

Trajectory definition

Stage 1: Per-frame data acquisition

Data distribution among processors

Stage 2: Cross-frame data analysis

How to implement the model without writing explicit parallel code?
Inspiration: Google’s MapReduce

Google File System

Input files

map(...) → map(...) → map(...)

K1: {v1, v1_i, v1_k}
K2: {v2, v2_j, v2}
K1: {v1}
K2: {v2}
K1: {v1_i}
K2: {v2_i}
K1: {v1_k}
K2: {v2_k}

output file

reduce(K1, ...) → reduce(K1, ...) → reduce(K2, ...)
Trajectory Analysis Cast Into MapReduce

- Per-frame data acquisition (stage 1): map()
- Cross-frame data analysis (stage 2): reduce()
- **Key-value pairs:** connecting stage 1 and stage 2
  - **Keys:** categorical identifiers or names
  - **Values:** including timestamps
  - **Examples:** \((\text{ion-id}_j, (t_k, x_{ik}, y_{jk}, z_{jk}))\)
Implementation: HiMach

- A MapReduce-style library that allows users to write Python programs to analyze MD trajectory
- An MPI-based parallel runtime that executes HiMach programs in parallel on a Linux cluster
- Performance: two orders of magnitude faster on 512 cores than on a single core
Part II: How to Overcome the I/O Bottleneck in Data Analysis?
Infrastructure for Data Analysis

Data validation
Summary statistics
Event detection
Data Clustering

Parallel programming model
Parallel I/O (read in particular)
Fault tolerance
Traditional I/O Infrastructure

Linux cluster

HiMach analysis program

Local disks

File servers

FE nodes

Anton supercomputers
Characteristics of MD Trajectories

- A large number of small frames
- Write once, read many
- Distinguishable by unique integer sequence numbers
- Amenable to parallel processing in the map phase
The Main Ideas

- Convert local hard drives on analysis nodes into a confederated disk cache
- Do not use a metadata server (single point of failure, performance bottleneck)
- Use MPI collective operations and bitmaps to arbitrate which nodes need to process which frames (a distributed consensus problem)
Implementation: Zazen

Linux cluster

Analysis node

HiMach analysis program

MPI-based Zazen protocol

Bodhi server

Local disks

File servers

FE nodes

Anton supercomputers
Efficiency: Outperforming NFS/PFS

I/O bandwidth of reading files of different sizes

Time to read 1 terabyte files of different sizes

File size for read

GB/s

File size for read

Time (s)
End-to-End Performance

- A HiMach parallel analysis program call *water-residence*
- 2.5 million small files/frames (430 KB each)

![Graph showing performance with different processes per node for NFS, Zazen, and Memory]
Part III: How to make analysis computation fault tolerant?
Infrastructure for Data Analysis

- Data validation
- Summary statistics
- Event detection
- Data Clustering

- Parallel programming model
- Parallel I/O (read in particular)
- Fault tolerance
Common Types of Failures

- Network File System (NFS), local hard drives, network switches
- Interplay between MPI and NFS may cause stalling of a HiMach analysis job
The Main Ideas

- Decouple MPI from disk read operation
- Retain MPI for the compute/communication intensive operations (reduce phase)
- Use a client-server model for the map phase
- Build a distributed parallel execution engine in conjunction with a disk cache manager
- Treat servers as a peer-to-peer network
- Use consistent hashing for frame placement
- Allow application-level data re-organization
Implementation: Pitstop

Analysis cluster
- MPI Process 0
- MPI Process 1
- MPI Process k
  - Pitstop client
- MPI Process n

Pitstop cluster
- Analysis node
  - Pitstop server
  - Bodhi server
  - Local disks
- Analysis node
  - Pitstop server
  - Bodhi server
  - Local disks

File servers

FE nodes

Anton supercomputers
Pitstop as a Fail-Fast System

- Supports molecular visualization applications
- Enables interactive data retrieval
- Executes HiMach batch analysis jobs
- Survives various node failures and NFS mount problems
Summary

- Interpretation of massive MD trajectories calls for (1) strong scientific intuition, (2) pertinent and efficient analysis algorithms, and (3) fast and scalable implementation.

- The progression from HiMach to Zazen to Pitstop has improved our chemists’ ability to analyze millisecond-scale MD trajectories more efficiently and effectively.