A Scalable Parallel Framework for Analyzing Terascale Molecular Dynamics Simulation Trajectories

Tiankai Tu, Charles A. Rendleman, David W. Borhani, Ron O. Dror, Justin Gullingsrud, Morten Ø. Jensen, John L. Klepeis, Paul Maragakis, Patrick Miller, Kate A. Stafford, David E. Shaw*

D. E. Shaw Research, New York, NY 10036 USA

Abstract—**As parallel algorithms and architectures drive the longest molecular dynamics (MD) simulations towards the millisecond scale, traditional sequential post-simulation data analysis methods are becoming increasingly untenable. Inspired by the programming interface of Google's MapReduce, we have built a new parallel analysis framework called *HiMach*, which allows users to write trajectory analysis programs sequentially, and carries out the parallel execution of the programs automatically. We introduce (1) a new MD trajectory data analysis model that is amenable to parallel processing, (2) a new interface for defining trajectories to be analyzed, (3) a novel method to make use of an existing sequential analysis tool called VMD, and (4) an extension to the original MapReduce model to support multiple rounds of analysis. Performance evaluations on up to 512 cores demonstrate the efficiency and scalability of the HiMach framework on a Linux cluster.**

I. Introduction

One of the challenging goals of high-performance molecular dynamics (MD) simulations is to model important biological processes that occur on the millisecond time scale—about two orders of magnitude beyond the duration of the longest current MD simulations. While great effort has gone into the design, implementation, and performance optimization of scalable parallel MD simulations using both software [1–7] and hardware [8–11] techniques, the analysis of the MD trajectories (simulation output data sets) has taken a back seat when it comes to scalability and performance, and is usually relegated to sequential processing.

Efficient and effective though they are for manipulating relatively short trajectories, sequential analysis tools lack the necessary scalability and performance to efficiently handle very long MD trajectories with millions of frames. Today's MD codes are capable of simulating molecular systems with tens of thousands of atoms at a speed of roughly a hundred nanoseconds per day, producing MD trajectories on the order of tens of gigabytes—a scale that already stresses the computational, memory, and I/O capabilities of existing sequential analysis tools. As petascale computers [12] and new special-purpose MD machines [11] become available, trajectories of unprecedented length will be generated, and the pressure on post-simulation data analysis tools will continue to mount.

The widening gap between highly scalable parallel MD simulations and unscalable sequential data analysis methods poses a serious analytics challenge. Left unaddressed, it would hamper scientists' ability to fully understand and interpret simulation results, thus defeating the purpose of developing faster and more scalable MD simulations.

Our research focuses on how to bridge this gap and provide a new analytical tool to deal with massive MD trajectories. At first glance, it might appear that implementing a predefined set of analysis functions within an efficient parallel program could solve the problem. But because the analysis needs of end users are highly varied, it is impossible to foresee all the required functionality and develop a one-size-fits-all parallel analysis program. On the other hand, passing along all responsibility to the end users is not a feasible solution either. Researchers who study MD trajectories are typically trained in biology, chemistry, physics, or medicine, and may not be experts in managing large data sets or writing parallel analysis software.

Inspired by Google's MapReduce framework [13, 14], we propose a new approach to this challenge. Our main idea is to provide a simple, MapReduce-style programming interface for users to write sequential MD trajectory analysis codes, which are then executed in parallel without user involvement. We have implemented our methodology within a new parallel analysis framework called *HiMach.* User programs interact with HiMach through an application programming interface, the HiMach API, to (1) define the MD trajectories to be analyzed, (2) specify the procedure of data acquisition, (3) implement analysis functions on the retrieved data, and (4) aggregate intermediate results when necessary. Computational chemists within our group have already used the HiMach API to develop a variety of different analyses, including the construction of electron density maps, the tracking of ions that permeate through a channel, and the calculation of self-diffusion coefficients.

The machinery of automatic parallel execution of user programs is implemented within the HiMach runtime, which is responsible for assigning tasks to processors, issuing I/O requests, interacting with VMD [15] (a sequential MD trajectory analysis tool developed by the Theoretical and Computational Biophysics group at the University of Illinois at Urbana-Champaign), storing and managing intermediate results, and communicating and exchanging data among processors. To make the programming model of MapReduce amenable to MD trajectory analysis, we introduce (1) a new MD trajectory data analysis model that is suitable for parallel processing, (2) a new interface for defining trajectories to be

*David E. Shaw is also with the Center for Computational Biology and Bioinformatics, Columbia University, New York, NY 10032. E-mail correspondence: David.Shaw@DEShawResearch.com.

analyzed, (3) a novel method to make use of VMD, and (4) an extension to MapReduce to support multiple rounds of analysis (i.e., chained reduce operations).

We assessed the efficiency of our framework on a commodity Linux cluster using two HiMach-based trajectory analysis programs. Both programs achieved nearly two orders of magnitude speedup when going from 1 core to 512 cores. Furthermore, we were able to perform a complex analysis on a one-terabyte trajectory in 15 minutes on 512 cores.

To the best of our knowledge, no existing MD trajectory analysis tools provide parallel execution capabilities, with the exception of VMD, which executes multi-threaded codes for a limited number of computationally intensive analysis routines such as finding neighboring pairs of atoms. We believe HiMach to be the first framework to support general-purpose parallel analysis of very long MD trajectories.

II. Background

An MD simulation models the motion of atoms within a molecular system. Given a set of initial conditions, an MD simulation computes a molecular system's *state*—the positions and velocities of the constituent atoms—over a sequence of discretized time steps. Typically, the duration of a time step is limited to no more than a few femtoseconds (1 femtosecond = 10^{-15} seconds) in order to resolve the highest frequency modes in the molecular system. At each time step, the force exerted on each atom is computed and a new state is calculated by numerically integrating Newton's laws of motion. At certain user-prescribed intervals, for example, every 10^4 femtoseconds (i.e., 10 picoseconds), snapshots of the state, called *frames*, are stored to disk. The set of all output frames constitutes the *trajectory* of an MD simulation. A trajectory frame consists of a collection of records (of positions and velocities), each corresponding to an atom. The ordering of the records in a trajectory frame corresponds to the ordering of atoms in a molecular structure file (e.g., a Protein Data Bank (PDB) file [16]), which specifies the types and initial positions of individual atoms as well as their bond connectivity.

An MD trajectory analysis program usually takes a molecular structure file and a sequence of trajectory frames as input, conducts an analysis calculation, and outputs the quantities of interest. An analysis program can be either generic or special purpose. Generic analyses—for example, the computation of properties such as the kinetic energy or center-of-mass velocity of a set of atoms, or the bond lengths between pairs of atoms—are used mostly for exploratory purposes. Special-purpose analysis programs, often devised and implemented by domain experts, aim to further quantify results or stimulate new insights.

A. *Visual Molecular Dynamics (VMD)*

Visual Molecular Dynamics (VMD) [15] is a widely used MD trajectory analysis tool. A user loads an MD trajectory into VMD and issues commands through either the commandline or the graphical user interface to manipulate the atoms, which are rendered accordingly on a computer display. In addition, a user can also write analysis scripts using Tcl or Python via VMD's scripting language interface. VMD executes these scripts in text mode without user intervention.

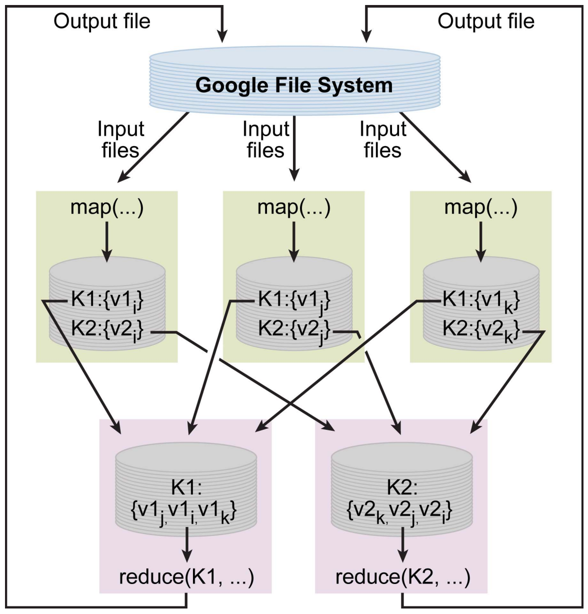


Figure 1. Data flow of a MapReduce computation.

For very long trajectories that do not fit into the main memory, a user may write analysis scripts to implement external memory algorithms [17] that explicitly store and retrieve temporary results to and from disk.

B. *MapReduce*

Google's MapReduce [13, 14] and its open-source implementation Hadoop [18] advocate a new way of developing and executing data-intensive parallel codes. The core idea of MapReduce is to provide a simple programming model to support a large class of computational problems commonly encountered in Web search engine applications. A MapReduce program requires a user to implement two functions:

```
map:    (k1, v1)         → list(k2, v2)
reduce: (k2, list(v2)) → v2
```

The `map()` function takes an input key-value pair and outputs a list of intermediate key-value pairs. The `reduce()` function accepts an intermediate key and a list of values associated with the key, and merges the values to produce a possibly smaller list of values. A reduce operation typically produces either one or zero output values. For example, to count the occurrences of each word in a large collection of documents, a user implements a `map()` function that takes as input a key-value pair `(k1,v1)`, where `k1` is the name of a document and `v1` is a list of words in the document, and for each word within the document, produces a new key-value pair `(k2,v2)`, where `k2` is the word itself and `v2` is 1 (indicating a single encounter of that word). The corresponding `reduce()` function takes as input `(k2,list(v2))`, where `k2` is a particular word and `list(v2)` is the list of values (all 1's) associated with `k2`, sums up the total number of occurrences of the word, and stores the count to disk.

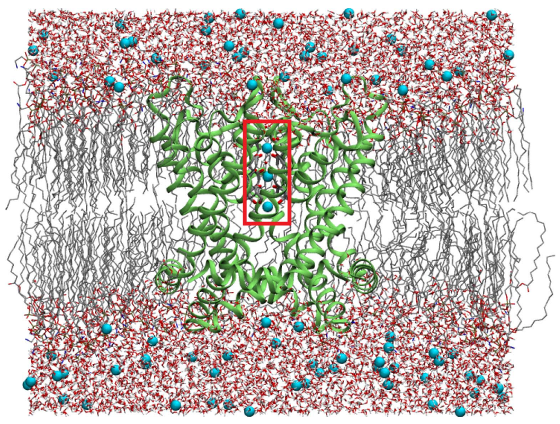


Figure 2. Ion permeation through a channel.

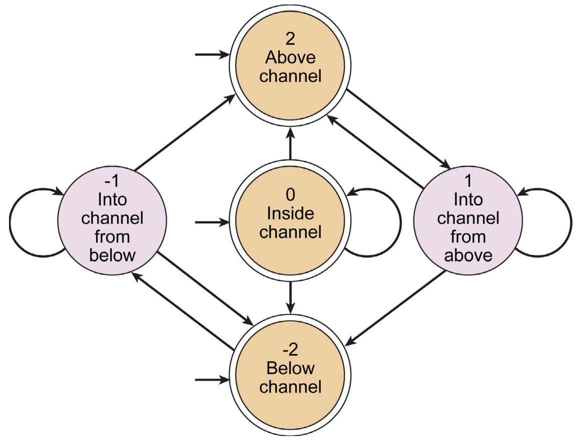


Figure 3. State transition of a potassium ion.

Fig. 1 illustrates the data flow of a typical MapReduce computation. The input files to a MapReduce program are fetched from the Google File System (GFS) [19], a distributed file system developed and used internally by Google. Compute nodes executing instances of the `map()` function produce intermediate key-value pairs that are stored on local disks. Compute nodes executing instances of the `reduce()` function use remote procedure calls (RPCs) to copy data to their local disks, and store output back into the GFS. A GFS server, a map function, and a reduce function may execute on the same compute node, although Fig. 1 shows them executing on different nodes.

The main advantage of MapReduce is that the details of parallel execution, such as data distribution and load balancing, are handled by the MapReduce library without user involvement. For example, the counts of a particular word—distributed across the compute nodes—are grouped together automatically on a single node and passed as an input parameter to the `reduce()` function, which then sums up the total count. A user need only focus on the operations that are to be applied to the collected values, rather than on the operations of collecting the values.

The programming model of MapReduce, though seemingly restrictive, has proven to be sufficiently flexible to support a large number of specific needs of Google's daily operation, including the construction of the indexing system that produces the data structures used for the Google Web search service. An average of 100,000 MapReduce jobs are executed on Google's clusters every day, processing a total of more than 20 petabytes of data per day [14].

Attractive as it is, applying the programming model of MapReduce to the analysis of an MD trajectory poses a number of technical challenges: How can we parallelize MD trajectory analysis in the first place? How can we write parallel trajectory analysis code using the `map()` and `reduce()` functions? How can we specify trajectories of interest to MapReduce? How can we select and manipulate atoms, bonds, and other aspects of the molecular systems? And how can we implement complex analysis codes that require multiple iterations of data reduction and synthesis (i.e., chained reduce operations) when MapReduce supports only one? We describe how to parallelize trajectory analysis in the Section III and address the other problems in Section IV.

III. Parallel MD Trajectory Analysis

We first use an example to demonstrate the challenges of parallelizing traditional sequential MD analysis codes and then present an alternative data analysis model that is amenable to parallel processing and that is able to take advantage of the simple programming interface of MapReduce.

A. A Sample Application

Fig. 2 shows a membrane protein that forms a channel (marked by a red rectangle) connecting two compartments (top and bottom). The channel allows only potassium ions, depicted as light blue balls in Fig. 2, to permeate. The objective of the analysis is to (1) count the total number of potassium ions that permeate through the channel from the bottom to the top or from the top to the bottom, (2) record the positions of each ion from the moment it enters the channel from the bottom (or top) to the moment it exits the channel at the top (or bottom), and (3) compute the time each ion takes to pass through the channel.

Fig. 3 shows the finite state diagram of a potassium ion as it interacts with the channel. Double circles represent possible initial states of an ion. A permeation event takes place if and only if the ion transitions from state –1 to state 2 or from state 1 to state –2.

A reasonably efficient sequential analysis might be implemented as follows. We access the frames of the trajectory one by one in ascending simulated physical time order (which is same as the frame number order). We keep in memory the positions and states of all potassium ions of the previous frame. For each new frame, we retrieve the coordinates of the potassium ions, update their states according to the finite state diagram of Fig. 3, and record the current simulated time and the coordinates of those ions that transition to or remain at states –1 or 1. After all the frames are processed, we examine the memory-resident data structures to compute the required quantities and store the results.

A large number of other analysis applications fall in the category of time-series analysis. After all, an MD trajectory is, in essence, a gigantic time series that records the coordinates of all atoms over time. Because the number of quantities to be analyzed (associated with atoms, bonds, or ions) is usually significantly smaller than the total number of atoms in a molecular system (e.g., the potassium ions account for fewer than 1% of all the atoms in our example), it is a natural choice to implement a sequential algorithm using memory-resident data structures to keep track of the updated states associated with the time series (because all the states fit in memory).

Maintaining a correctly updated data structure, however, requires that the frames be accessed in a strictly ascending order. Out-of-order processing of frames could result in incorrect results—for example, by disrupting the state transition flow dictated by Fig. 3. An undesirable side effect of such a traditional sequential analysis method is that it is difficult, if not impossible, to parallelize the analysis codes due to the strong data dependence imposed by the order in which frames must be accessed.

B. *A New Data Analysis Model*

To take advantage of parallel processing, we introduce a new data analysis model that organizes an analysis task into three distinct steps, as shown in Fig. 4:

1) Trajectory definition. Instead of specifying the order of accessing trajectory frames, we declare the frames to be analyzed as a set—for example, by providing the indices of the first and last frame to be analyzed and a stride value that specifies how many frames should be skipped between frames.

2) Per-frame data acquisition and analysis. After acquiring the atom coordinates and velocities from a particular frame, we extract quantities of interest or compute analysis results for that frame independently from operations performed on other frames or any prior results.

3) Cross-frame data analysis. After all data of interest have been retrieved or computed from individual frames, we conduct cross-frame analysis for each time series independently, which may involve a number of iterations as indicated by the loop-back arrow in Fig. 4.

This model allows for parallel processing of an MD trajectory analysis in two stages. In the first stage, the trajectory definition step and the per-frame data acquisition analysis step jointly create parallel I/O and computational tasks that can be distributed among processors. Knowing what data (i.e., the frames) a user program needs rather than how the program wants to access the data (i.e., the order) provides an opportunity for out-of-order parallel I/O. The only requirement is that all frames must be accessed at some point by some processor; the particular order of data access is no longer relevant. Once a distributed consensus is reached regarding which processor is responsible for which frames, per-frame data acquisition and analysis can proceed in parallel without further communication. In the second stage, cross-frame data analysis tasks associated with different time series are carried out by parallel processors simultaneously.

The proposed model may appear to be inapplicable to those analysis scenarios where an algorithm depends on the results or states computed from previous frames to decide what to do with the current frame. Our solution to this problem is to redesign an analysis to eliminate the strong data dependence between adjacent frames. We first retrieve all potentially useful data in the per-frame data acquisition and analysis step, and later identify and ignore unused data in the cross-frame analysis step. In our sample application, we can record the

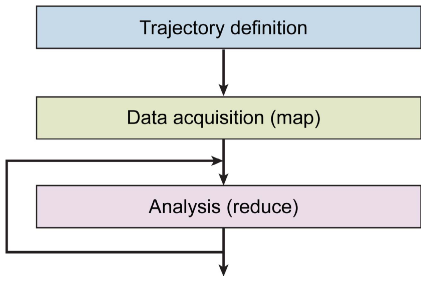


Figure 4. Casting an MD analysis into three steps that are amenable to parallel processing.

coordinates of all ions. As we execute the state transition machine for each time series in the cross-frame data analysis step, we can identify the ions that have never permeated the channel and simply ignore them. Because the quantity of data associated with useless coordinates is bounded by the number of ions (recall that they account for fewer than 1% of all atoms), the overhead of retrieving and storing the extra data is negligible. In general, whether our model is applicable depends on the data access pattern and algorithmic design of an application.

The real challenge is how to connect the two stages of analysis—that is, how to tie together the pieces of a time series extracted in the per-frame data acquisition and analysis step, and deliver the combined result to a single processor that is responsible for carrying out the cross-frame data analysis task for that particular time series. To solve this problem, we make use of the programming interface of Google's MapReduce.

C. *MapReduce Revisited*

The conceptual jump to connect the data analysis model just described to MapReduce is the re-thinking of the usage of the key-value pairs. A key, instead of being thought of as a numerical value or a word, should be thought of as a *categorical identifier* or *name* for a group of related values. For instance, we can use the unique integer identifier of a potassium ion as a key to identify all its time-varying coordinates extracted from a trajectory. Additionally, a value associated with a key should carry a *timestamp* field (e.g., frame number or physical time) in addition to the quantities of interests (e.g., the coordinates of ions). Because the order of access to trajectory frames is not specified in our model, the timestamp field is necessary for reconstructing a valid time series (in ascending time order). Therefore, a key-value pair within an MD analysis takes the following form:

(*categorical name,* (*timestamp, quantities of interest*))

Cast into the programming model of MapReduce, the per-frame data analysis step corresponds to the `map()` function. It retrieves or computes quantities of interest and produces key-value pairs that contain categorical names and timestamps. The cross-frame data analysis step corresponds to the `reduce()` function of MapReduce. It processes and analyzes the value lists (i.e., time series) associated with individual keys (e.g., a particular ion). It is the understanding and realization of such a connection that has prompted us to adopt the programming interface of MapReduce.

TABLE I. ATTRIBUTES OF A `TrajectoryDescriptor` OBJECT

Attribute	Usage
`molfmt`	Format of the molecular structure file
`molfname`	Pathname to the molecular structure file
`traj_dir`	Directory where frames are stored
`usevmd`	Whether VMD is used
`traj_am`	User-defined frame access method
`seltype`	Frame access pattern
`begin`	First frame to be accessed for a strided access
`end`	Last frame to be accessed for a strided access
`stride`	Number of frames to skip for strided access
`fixed_frameno`	Frame number of a fixed access

IV. THE HIMACH FRAMEWORK

To support the parallel data analysis model described in Section III, we have developed a parallel analysis framework called HiMach, which consists of an API that allows a user to write trajectory analysis programs sequentially, and a runtime that carries out the parallel execution of user programs automatically. We have implemented HiMach using Python and pyMPI [20]. In our implementation, we treat multiple cores on a single chip as separate processing units as if they were independent processors. We use the term processor generically in this paper to refer to either a single-core processor or a core on a multi-core chip.

A. The HiMach API

The HiMach API allows users to (1) define a trajectory of interest, (2) carry out per-frame analysis, (3) implement cross-frame analysis, (4) aggregate intermediate results when necessary, and (5) launch an analysis job and transfer control to the HiMach runtime.

1) Trajectory Definition: The HiMach API provides a Python class called `TrajectoryDescriptor` for the user to define a trajectory of interest. Except for the initialization method that performs basic sanity checks, this class has no other methods.

A `TrajectoryDescriptor` object records the characteristics of the trajectory of interest, using attributes shown in TABLE I. Most of the attributes are self-explanatory except for `usevmd`, `traj_am`, and `seltype`.

The `usevmd` attribute allows a user to specify whether VMD is used within their analysis programs. We will explain how VMD is automatically loaded and how HiMach and user programs interact with VMD in Section IV-B3.

The `traj_am` attribute allows a user to define a function that takes a frame number as input and outputs the name of the file to be accessed, the offset in the file, and the number of bytes to be read, thus overriding HiMach's default file name of `traj_dir/frameX`, where `X` stands for the number of the frame being accessed.

The `seltype` attribute specifies how a trajectory is accessed for the purpose of an analysis. The default value of the `seltype` attribute is `strided` and the only other valid value is `fixed`. Although most applications analyze only one trajectory at a time, a user may need to pair up two trajectories, or even group three or more trajectories for cross-trajectory

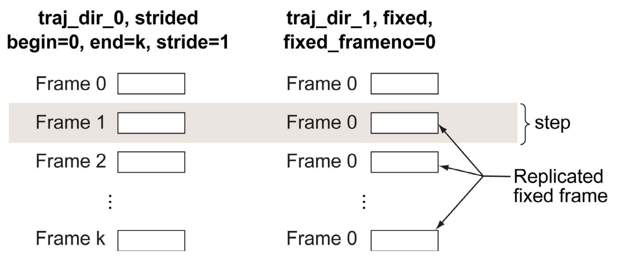


Figure 5. A contrail consisting of two trajectories.

analysis. The HiMach API supports such analysis scenarios by allowing a user to create a *contrail*, a Python list that contains a number of `TrajectoryDescriptor` objects. A contrail is an extension to the simple concept of a trajectory and may take one of the following forms:

- `Strided` access mode: a single strided trajectory, defined by the `begin`, `end`, and `stride` attributes
- `Fixed` access mode: a particular frame of a trajectory, defined by the `fixed_frameno` attribute
- Hybrid access mode: one or more `strided` trajectories and one or more `fixed` trajectories

For the hybrid access mode, we define the length of the contrail as the length of the longest `strided` trajectory. The `fixed_frameno` frames from the `fixed` trajectories are logically replicated as many times as necessary to match the length of the contrail as shown in Fig. 5.

2) Per-Step Data Acquisition: The HiMach API provides a class named `Mapper` as a template for a user to define the logic of per-step data acquisition. A step comprises one frame from each of the trajectories that a user defines in a contrail, as shown in Fig. 5. Note that we have dropped the use of term "per-frame data acquisition" defined in Section III-B in favor of "per-step data acquisition" to refer to the more general case of contrails.

The `Mapper` class consists of two methods, `init()` and

```
class Mapper(object):
    def init(self, step):
        return
    def map(self, step):
        raise NotImplementedError
```

`map()`, both receiving the same input parameter, `step`, which is constructed and passed in by the HiMach runtime. The parameter `step` is a Python list whose entries, referred to as *cursors*, record the properties of the frames belonging to a step on a contrail. The order of cursors in the `step` list is the same as the order of the `TrajectoryDescriptor` objects a user specifies in the corresponding contrail list.

The `init()` method allows a user to implement deferred runtime initialization of an analysis program. It is called by the HiMach runtime *once* on each processor, before any frame is processed on that processor. For example, a user may define a set of VMD atom selection objects within an `init()` method. Such statements are executable only if VMD has been initialized and the molecular structure file has been loaded, neither of which has taken place when a `Mapper` object is created. Hence, it is impossible to combine such initialization with the default Python initialization routine `__init__()`; we must defer the initialization until HiMach has loaded VMD and the molecule structure file. If a user does not override this method, it has no effect at all.

The `map()` method, also implemented by a user, typically uses the VMD atom selection language to conduct RMSD alignment or select coordinates of the atoms, and produces key-value pairs (in the form as specified in Section III-C) using the Python `yield` statement.[1] The key-value pairs are saved on local disk transparently to the application by the HiMach runtime. Besides simple data extraction, a `map()` method can also perform more compute-intensive processing of a frame and produce derived key-value pairs.

3) Cross-Step Data Analysis: A class called `Reducer` provides an interface for a user to implement cross-step data analysis. The only method of this class is `reduce()`, with input parameters `key` and `valuelist` representing a categorical name and the associated list of values, respectively. Both parameters are constructed and passed in by the HiMach runtime.

```
class Reducer(object):
    def reduce(self, key, valuelist):
        return
```

If a user program needs to process the value lists as a time series, the first step in the `reduce()` method is to sort the list in ascending timestamp order. Existing sequential algorithms can then be applied to the reconstructed time series.

In the process of developing and deploying the HiMach framework, we noticed that a single-round reduce, as implemented by the original MapReduce model, is insufficient for many MD trajectory analyses, which instead require multiple rounds of data analysis and synthesis—for instance, summarization or reduction of previously computed analysis results.

Based on this observation, we implemented a new feature within HiMach to allow a user program to conduct multiple rounds of reduce operations. If a `reduce()` method yields key-value pairs instead of returning directly, it is an indication that another round of reduce operation is needed by the user code. The newly yielded key-value pairs are saved by the HiMach runtime in the same manner as are the key-value pairs yielded by the map step. To avoid mixing the two types of key-value pairs, the HiMach runtime keeps the new list of key-value pairs separate from those generated by the map step or by previous rounds of reduce operations.

4) Aggregation of Partial Results: When a `map()` or `reduce()` method yields a value for a particular key, the default action taken by the HiMach runtime is to append the value to a list associated with that key. In certain cases, however, it is more efficient to aggregate the partial results in an application-specific way rather than simply keeping the values in a list. To support this kind of processing, the HiMach API provides an additional class called `MapReducer`, which exports an `assemble()` method.

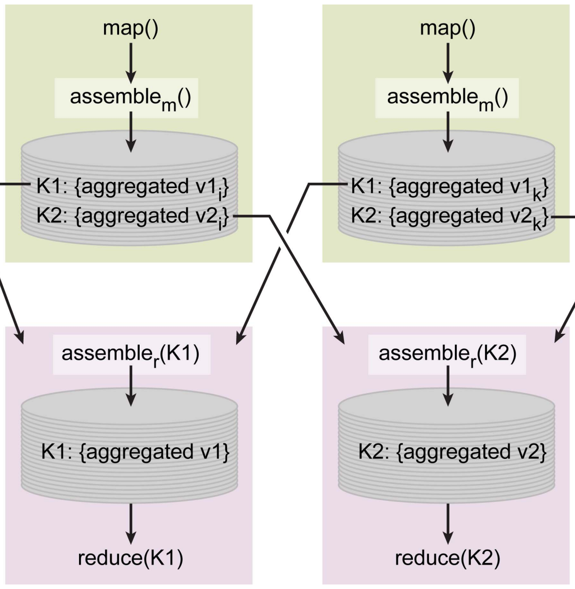


Figure 6. User-defined assemble functions.

As illustrated in Fig. 6, a user can optionally override the default `assemble()` method to instruct the HiMach runtime on how to aggregate (shown as $\texttt{assemble}_m$ in Fig 6) the key-value pairs as they are produced by the `map()` method locally on the same processor, and how to aggregate (shown as $\texttt{assemble}_r$ in Fig 6) the value lists associated with a particular key sent by another processor. Whether the two types of aggregation are the same is application-dependent. By aggregating the intermediate results, a user can reduce both the memory usage by each processor and the traffic on the communication network.

HiMach's support for dynamic data aggregation is an extension of the *combiner* function originally proposed by Google's MapReduce [13]. The difference between the two is that the combiner function is called only after all the key-value pairs have been produced on a processor, while the `assemble()` method is called as lists become available. In addition, no explicit data aggregation is supported on the receiving processors in MapReduce.

5) Control Transfer to the HiMach Runtime: After the user defines the data (a `contrail` object) and the computation (a `MapReducer` object), the user then calls a HiMach API function named `launch()` to transfer the control to the HiMach runtime.

B. The HiMach Runtime

The HiMach runtime, as shown in Fig. 7, executes in parallel on all processors. It implements all the machinery of executing user programs in parallel, which include (1) assigning tasks to processors, (2) issuing parallel I/O requests, (3) interacting with VMD, (4) storing and managing temporary results (key-value pairs), and (5) communicating and exchanging data among processors.

[1] When a `yield` statement within a Python function is executed, the data object being yielded is returned to the caller of the function and control is transferred back to the caller. When the same function is called again, execution resumes from the statement immediately following `yield`.

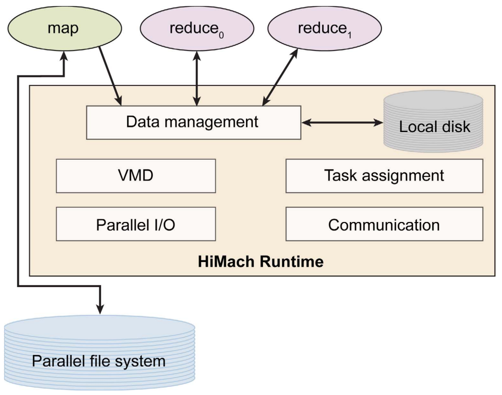


Figure 7. Overview of the HiMach runtime on a single processor. The ovals represent a map task and two rounds of reduce tasks executing on the same processor.

1) Task Assignment: Two separate types of tasks are associated with a HiMach program: per-step data acquisition and analysis tasks (*map tasks*) and cross-step analysis tasks (*reduce tasks*). The assignment of map tasks is independent from that of reduce tasks, and the assignment of reduce tasks for a particular round is independent from that of other rounds. Fig. 7 shows one map task and two rounds of reduce tasks assigned to a processor.

We assume that the amount of work associated with each map task—for example, the time spent retrieving and extracting data from a single frame or computing a quantity associated with a frame—is more or less the same. We also assume that the cross-step analysis workload associated with each reduce task of the same round is roughly the same.

In our current implementation, we use a simple block data decomposition algorithm to assign tasks to processors. For map tasks, we divide a list of trajectory frames evenly among all processors, each responsible for executing the map tasks associated with a contiguous set of frames. No interprocessor communication is needed for assigning map tasks. To assign reduce tasks, we conduct one round of a global collective operation (i.e., with `MPI_Allgather`) to obtain the complete list of intermediate keys to reduce. The keys are treated as if they were elements of a contiguous array. We apply the same block data decomposition algorithm to partition these keys among processors. Note that to avoid a potential performance bottleneck, we do not use a single master node to assign map or reduce tasks.

2) Parallel I/O: Parallel I/O takes place after the map tasks have been assigned to processors. Since the frames to be accessed by each processor have already been determined, no communication is required to coordinate I/O. We assume that trajectory frames are organized in such a way—for example, as a sequence of megabyte-sized files—that they are amenable to efficient parallel I/O if a sufficiently large number of I/O requests are outstanding. At runtime, HiMach issues file read requests on behalf of a user program and passes the data of each frame to the `map()` method as an input parameter. The write operation, which outputs the final results of an analysis, is handled by the user program itself. Since a user program cannot select on which processor a `map()` or `reduce()` function is executed, it must create or open files on a global file system such as a network file system (NFS) or a parallel file system to write out the results.

3) Interaction with VMD: We use VMD to allow user programs to manipulate and conduct computation on atoms, bonds, and other molecular structures. We have not parallelized VMD; instead, we support parallelism by running and interacting with an instance of VMD on each processor using the following method.

When a user program indicates that it needs to use VMD, the HiMach runtime first automatically starts a local instance of VMD on each processor and loads the appropriate molecular structure file. Next, it calls a VMD built-in function named `molecule.dupframe()` to create a placeholder frame buffer for the molecule structure. Then, the HiMach runtime calls the VMD `vmdnumpy.positions()` function to obtain a handle to the newly created internal frame buffer. This buffer handle becomes the interface between HiMach and VMD.

At this point, HiMach calls the user-defined `init()` method to perform deferred initialization and then starts reading trajectory frames on behalf of the user program. Once a frame is retrieved, the HiMach runtime copies the coordinates of the atoms to the buffer handle, overwriting the coordinates of the previously processed frame.

The frame number, the VMD frame buffer handle, and the molecule ID (generated automatically by VMD) are passed to the user-defined `map()` method as input parameters. Within the `map()` method, all user-issued VMD commands operate on the coordinates of the new frame. Thus, the loading of new atom coordinates and the updating of the input parameter list are handled automatically by the HiMach runtime.

4) Key-value data management: We have observed that the Python interpreter uses a large amount of memory to keep track of long value lists, resulting in memory thrashing of both user programs and the HiMach runtime. As a remedy to this problem and also as a measure to deal with long MD trajectories (which will exhaust the main memory on each processor regardless of the behavior of a Python interpreter), as well as to support applications that retrieve a large percentage of input frame data (instead of the 1% that our sample application uses), we use an out-of-core data management scheme to keep track of key-value pairs.

We set an internal counter to record the number of key-value pairs that have been yielded. Once the counter reaches a preset, tunable threshold, the values associated with the keys are serialized and output to temporary files on local disk. Memory space previously used by the value objects is released and the internal counter is reset to 0. The management of the temporary files is transparent to the user program. They are given unique names by the HiMach runtime and are associated with the memory-resident partial value lists. When inter-processor communication takes place, the temporary serialization files are exchanged among processors directly. The HiMach runtimes on the receiving processors deserialize the values from the files and reconstitute the lists for the respective keys without user involvement.

5) Communication: Unlike Google's MapReduce, which uses remote procedure calls (RPCs) to copy data from other processors, the HiMach runtime uses MPI to exchange data

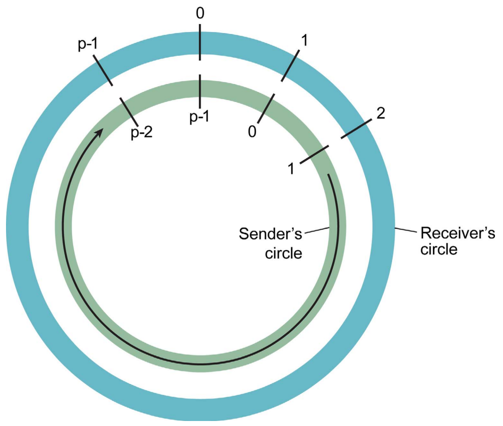


Figure 8. Data exchange between MPI processes. The inner circle represents the senders and the outer circle the receivers. Each (tick-mark) alignment configuration defines a round of communication.

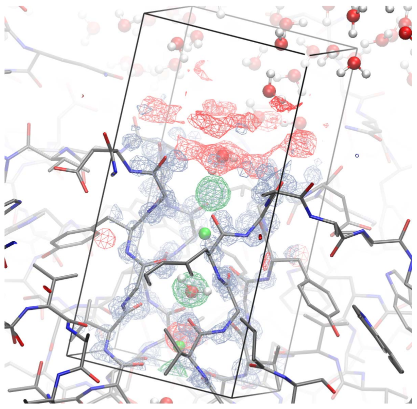


Figure 9. An electron density map. The green contours are associated with potassium ions, red contours with water molecules, and grey-blue contours with the atoms of proteins.

among processors. The advantage is that HiMach avoids the potential performance bottleneck associated with a single master node, which has to keep track of the progress of each process and inform processes when to start RPC copies. The disadvantage is that HiMach does not provide the fault tolerance offered by MapReduce: if one compute node goes down during an analysis execution, the HiMach user program must be re-executed from the beginning.

Because the amount of data (value lists) associated with the keys from a very long trajectory may approach or even exceed the main memory size on each processor, it is inefficient, and sometimes impossible, to post all the MPI receives simultaneously. We have therefore devised a more regulated communication protocol to exchange data among MPI processes, which takes place in $p - 1$ rounds where p is the number of MPI processes. At round k, process i sends data (if any) to process $(i + k + 1) \bmod p$ and receives data (if any) from process $(i - k - 1) \bmod p$.

Fig. 8 shows the conceptual model of the communication protocol. The tick-marks on the circles denote MPI ranks. The inner circle represents the senders and the outer circle the receivers. Now fix the outer circle and dial the inner circle clockwise one tick-mark at a time. We obtain $p - 1$ different alignment configurations (ignoring the case of a matching alignment that represents self-communication). Each of the inner-outer circle alignments defines a unique round of communication in which each MPI process on the inner circle sends to the corresponding MPI process on the outer circle the values associated with the keys that the outer process is responsible for reducing. For example, in the configuration shown in Fig. 8, process 0 sends data to process 1 and receives data from process $p - 1$. Each MPI process proceeds to the next round of communication only after it finishes the current round. In the end, every process communicates to every other process in the system. The multiple rounds of pair-wise communication are designed to control the amount of data pushed to the communication network and to bound the memory usage of both senders and receivers.

In summary, the HiMach framework allows users to implement an MD trajectory analysis using a MapReduce-style interface, make use of the analytical capability of VMD, perform data aggregation when necessary, and conduct multiple rounds of data analysis and synthesis.

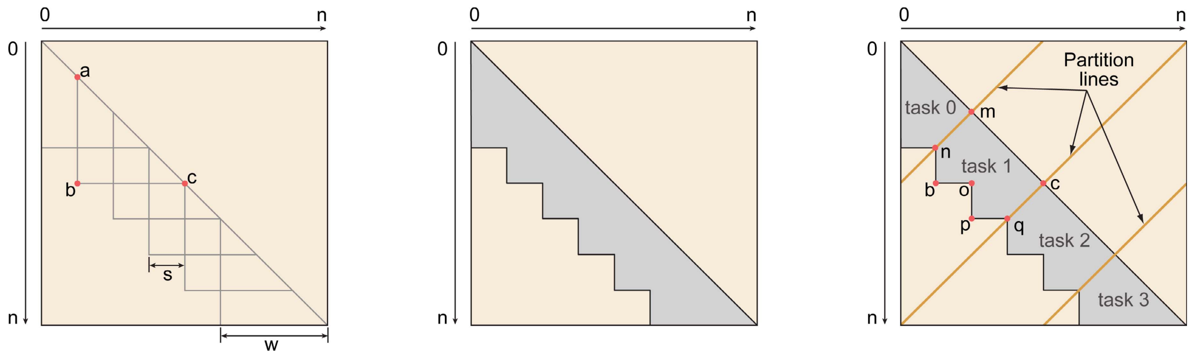
V. Applications

We have so far presented the HiMach framework in an abstract setting, referring only briefly to the needs of applications. This section provides three real-world examples to illustrate the programmability and flexibility of the HiMach framework. For the purpose of this paper, we provide only an overview of the example applications and an outline of the key algorithmic techniques.

A. *Electron Density Map Construction*

An electron density map defines a frame's 3D spatial (probability) distribution of electrons, thus illustrating the atomic structure of proteins and other macromolecules. While electron density maps are generally obtained through physical experiments such as X-ray diffraction measurements or electron microscopy reconstructions, MD simulations have introduced a new way of constructing electron density maps, as shown in Fig. 9. The iso-electron-density contours, shown as chicken-wire meshes, approximately represent the locations of potassium ions and water molecules within a conduction channel outlined by the rectangular box. The electron density map of an MD trajectory is defined as the average of the electron density maps associated with individual frames.

We design a HiMach analysis algorithm as follows: (1) write a stand-alone function that computes the electron density map for a single frame (the specifics are beyond the scope of this paper); (2) define a `map()` method that calls that function and yields `('electron map', (emap, fcount))` as the output key-value pair, where `emap` (a Python Numpy array) records the electron map of a single frame, and `fcount`, which is initialized to 1, records how many frames have contributed to the resulting `emap`; (3) override the `assemble()` method of the `MapReducer` object to sum up partial results of `emap` and `fcount` (to minimize memory usage on each processor and enable parallel summation of partial results); and (4) define a `reduce()`



(a) A symmetric RMSD matrix. (b) A "serrated" band. (c) Workload partitioning.

Figure 10. All-to-all RMSD within a sliding window.

method that divides the final accumulated electron density map (`emap`) by the number of frames processed (`fcount`).

Algorithmically, this program falls into the so-called "embarrassingly parallel" category. It does not even need to perform cross-step data analysis on time series. Nevertheless, it represents an effective and commonly-used method for MD trajectory analysis.

B. *Ion Permeation Reimplemented*

The reimplementation of the analysis of ion permeation through a channel is more complicated than the example in the previous section. The approach to achieving parallelism, as outlined in Section III-A, is to retrieve all coordinates of ions, and to identify and ignore the useless data later.

We implement a `map()` method to extract all the coordinates of potassium ions and yield `(ion_id, (t,x,y,z))`, where `ion_id` is an ion identifier, `t` is the physical time of trajectory frame, and `x`, `y`, and `z` are the coordinates of the ion within a trajectory frame. The `reduce()` method, responsible for processing a time series associated with the coordinates of each potassium ion, first reorders the value list using the physical time `t` as the sorting key, and then implements the state transition diagram shown in Fig. 3.

Such an algorithmic design may seem counterintuitive, since we have to do more—and potentially useless—work in order to carry out the analysis more efficiently. But as noted above, the potassium ions account for only a small percentage of the atoms of a molecular system; in exchange for a small amount of additional data retrieval and processing, we are able to parallelize and speed up the entire computation.

C. *All-to-All RMSD within a Sliding Window*

The root mean squared deviation (RMSD) is a measure that quantifies the structural "proximity" between two frames with respect to a set of atoms. It is defined as $RMSD_{i,j} = \sqrt{\left(\sum_{a=1}^{k} r_a^2\right)/k}$, where i, j are frame numbers, k is the number of user-specified atoms, and r_a is the distance between the positions of the atom a in frame i and j.

The goal of the analysis is to compute the mean value of all pair-wise RMSD values within a fixed-size sliding window of w consecutive frames, which slides forward in time by a progression size of s frames at a time. The user-specified atoms involved in the RMSD calculation are the alpha carbons on the main chain of a protein, which account for fewer than 1% of the atoms in the system.

Fig. 10(a) shows the symmetric RMSD matrix for an MD trajectory with n frames. Each triangle, for example the one labeled `abc`, represents a sliding window. All matrix entries (RMSD values) within a triangle are needed by the corresponding sliding window to compute the statistics correctly. Naive parallelization of this calculation by assigning each window to a processor (whether using HiMach or MPI directly) would result in inefficient and redundant data accesses and computations due to the overlaps between the sliding windows.

The HiMach algorithm we developed reads each frame once and computes the RMSD of each distinct (i, j) pair once. The key idea is to convert the computation associated with a dynamically sliding window to that of a fixed set: all entries in the "serrated" band of the RMSD matrix, as shown in Fig. 10(b), are computed first regardless of how many sliding windows overlap on the entries. The RMSD values are then distributed to the associated sliding windows for statistical calculations.

We partition the workload of computing the "serrated" band among processors (more or less) evenly as shown by the 45-degree partition lines in Fig. 10(c). Each computational task—identified in Fig. 10(c) as a patch bounded by the diagonal of the matrix, two adjacent partition lines, and the "serrated" edge—is assigned a unique task ID, starting from 0 at the upper left corner and increasing monotonically towards the lower right corner of the RMSD matrix. For example, task 1 corresponds to a polygonal patch labeled by `mnbopqc`.

The task IDs provide the bridge that connects the per-step data analysis step with the cross-step analysis step. When a `map()` method receives an input frame i from the HiMach runtime, it extracts the atomic positions of interest, computes the IDs of the tasks whose patches intersect row i or column i, and yields a sequence of key-value pairs, using the computed task ids as the keys, and the frame number i and the atomic positions as the value. That is, those task IDs need the set of atomic positions from frame i to compute the matrix entries within their respective patches.

TABLE II. STRONG SCALING OF THE PERMEATION PROGRAM

Nodes	1	1	1	2	4	8	16	32	64	128
Cores	1	2	4	8	16	32	64	128	256	512
Frames/core	83750	41875	20938	10469	5234	2617	1309	654	327	164
Total time(s)	17792	11740	9458	4828	2495	1308	691	389	233	183
Map time (s)	17767	11716	9435	4804	2471	1284	665	362	202	134
Reduce time(s)	25	24	23	24	24	24	26	27	31	49
Map throughput (MB/s)	8.9	13.6	16.6	32.5	63.3	119.8	225.7	414.0	695.1	1061.6
Parallel efficiency (node)	n/a	n/a	100%	98%	95%	90%	86%	76%	64%	40%

We conduct two rounds of reductions. In the first round, each reduce task computes the RMSD values within its patch, identifies the sliding windows to which each of its matrix entries contributes, and yields key-value pairs, using the sliding window IDs as the keys and the RMSD values as the values. In the second round, each reduce task computes the mean of the input RMSD values associated with each sliding window.

Although data replication occurs in both rounds of reduce operations, neither represent a scalability bottleneck. In the first round, some atomic positions are replicated (distributed) among multiple reduce tasks that compute matrix band entries. The overhead is negligible since (1) it only occurs when frames near the partition lines are processed; (2) the quantity of data involved is small (fewer than 1% of all atoms); and (3) the replication ratio drops when very long trajectories are involved. In the second round, the RMSD values are replicated (distributed) among multiple reduce tasks that calculate statistics associated with the sliding windows. The cost, bounded by w/s, where w is the size of the sliding window and s is the progression size, is not affected by the number of processors or the length of a trajectory.

The three examples, though constituting only a small selection of real-world applications, capture the complexities of a large variety of analysis scenarios, ranging from the "embarrassingly parallel" to the moderately complex to the more sophisticated.

VI. PERFORMANCE EVALUATION

We use two HiMach analysis programs presented in Section V—the tracking of ion permeation through a channel and the computation of a sliding-window RMSD—to drive the performance evaluation of our framework. We refer to the two analyses as *the permeation program* and *the RMSD program,* respectively. Note that our benchmarking implementation of the ion permeation analysis is slightly different from what we described in Section V-B. Instead of generating one time series for each individual ion, the permeation program generates a time series of vectors, each of which records the positions of all ions within a frame. In other words, one reduce task processes all ions.

Our performance data were obtained from running the permeation and RMSD programs on a commodity Linux cluster with 128 compute nodes. Each node has two Intel Xeon 2.66 GHz quad-core processors, 16 GB of memory that is shared by all 8 processor cores, and two 250 GB SATA disks that are organized in a RAID 1 (mirrored) configuration. The operating system running on the nodes is CentOS 4.6 with a Linux kernel version of 2.6.22. The nodes are interconnected via a Gigabit Ethernet connection. All nodes have access to a NFS directory exported by a Sun x4500 storage server, which has a capacity of 40 TB. Although the NFS directory has the capacity to store terabyte MD trajectories, our experiments show that the peak read bandwidth from the NFS server flattens out at around 300 MB/s for our applications.

To achieve better I/O performance, we installed a PVFS2 [21, 22] parallel file system on 64 of the 128 compute nodes and used PVFS2 to store and retrieve trajectory frames. To alleviate resource contention, we used only 4 cores per node for the analysis applications in our experiments, leaving the remaining 4 cores to run the PVFS2 servers. Although the 64-node PVFS2 file system does not match the performance of NFS when a single core is used to run applications (8.9 MB/s on PVFS2 vs. 38 MB/s on NFS), PVFS2 outperforms NFS by up to a factor of 4 when a large number of cores issue parallel I/O requests simultaneously.

A. *Strong Scalability*

In this set of experiments, we fix the MD trajectory to be analyzed and increase the number of cores that execute a HiMach program. The objective is to assess how much faster a HiMach program can run while more resources (compute nodes) are added to solve a fixed-size problem.

The MD trajectory analyzed by the permeation program consists of 83,750 frames, each 1.97 MB in size. The total size of the trajectory is 154 GB. The frames are organized in files of size 67.2 MB, each of which contains 34 consecutive frames.

TABLE II shows the performance of the permeation program. As we increase the number of cores from 1 to 512, the running time decreases from about 5 hours to 3.5 minutes—an improvement of nearly 2 orders of magnitude. The permeation program, like many other trajectory analysis codes, is I/O bound. The time spent in the `map()` function, which is responsible for retrieving and extracting data from a frame, dominates the time spent in executing the `reduce()` function, especially on a small number of cores. Thus, the main performance gain comes from the improvement of map throughput, which increases from 8.9 MB/s on 1 core to 1061.6 MB/s on 512 cores. The increase of map throughput, however, is non-linear. At the low end, when we increase the number of

TABLE III. STRONG SCALING OF THE RMSD PROGRAM

Nodes	1	1	1	2	4	8	16	32	64	128
Cores	1	2	4	8	16	32	64	128	256	512
Frames/core	110740	55370	27685	13843	6921	3461	1730	865	433	216
Total time(s)	12332	8564	6443	3383	1717	897	452	235	129	104
Map time (s)	8166	6398	5113	2575	1307	691	350	183	98	64
Reduce time(s)	4166	2165	1330	808	410	205	102	52	31	40
Map throughput (MB/s)	10.6	13.6	17.0	33.8	66.5	125.8	248.5	474.6	884.6	1351.3
Parallel efficiency (node)	n/a	n/a	100%	99%	98%	92%	91%	87%	78%	45%

cores from 1 to 2 to 4, the sustained map throughput rate does not improve linearly (the suboptimal performance may be caused by resource contention on a single compute node). As we increase the number of cores from 4 to 128, using more and more compute nodes, the performance improves almost linearly. The parallel efficiency with respect to the performance of 4 cores on a single node is above 75% until we use 64 nodes. At this point, the performance improvement slows down, as we approach the upper limit of sustained parallel read bandwidth of the PVFS2 file system.

The MD trajectory analyzed by the RMSD program consists of 110,740 frames, each with a size of 823 KB. The total size of the trajectory is 85 GB. Every 82 consecutive frames are organized in a file of size 67.5 MB.

The performance characteristics of the RMSD program, shown in TABLE III, are similar to those of the permeation program. As in the case of the permeation program, the running time of the RMSD program improves from about 3.5 hours to 2 minutes as we increase the number of cores used from 1 to 512. The `reduce()` function of the RMSD program, unlike that of the permeation program, is executed in parallel on all the cores, as explained in Section V-C. The reported time for the `reduce()` function also includes the overhead of inter-processor communication. TABLE III also shows that the time spent in the `reduce()` function decreases proportionally to the number of cores used until 512 cores, at which point the cost of communication overtakes the execution time of the `reduce()` function and becomes the dominant factor. In fact, the reduce time for 512 cores is longer than that for 256 cores. The degradation in performance suggests that the communication-to-computation ratio (i.e., the ratio between the width of an inter-task partition edge and the area of a patch) for this problem on 512 cores has become too large to be efficient.

B. *Weak Scalability*

In this set of experiments, we fix the amount of work each core carries out and increase both the length of a trajectory and the number of cores for executing a HiMach program. The goal is to assess how the HiMach framework will perform when we add more resources to solve a proportionally larger problem. We created a synthetic data set by replicating the RMSD trajectory 12 times to generate a one-terabyte data set.

Our experiment started from 4 cores analyzing an 87 GB trajectory, and scaled to 512 cores analyzing the full 1 TB synthetic trajectory. In the ideal case, the running time of each experiment would be the same. TABLE IV shows the weak scaling performance of the RMSD program. Note that in all experiments each core processes 2,400 frames. The per-node parallel efficiency exceeds 85% until 512 cores and then drops to 66% due to sub-linear improvement in map throughput and extra overhead in the reduce function. Nevertheless, the running time of carrying out the analysis of a one-terabyte trajectory is only 15 minutes on 512 cores.

Assuming that an MD simulation is capable of simulating some biochemical system at a rate of 1 millisecond per day—well beyond the capability of present day computer systems—and also assuming that the simulation outputs frames every 10 picoseconds (1 picosecond = 10^{-12} second), then a total of 100 million frames will be generated a day, or roughly 1 million frames every 15 minutes. Given that the analysis throughput rate of the RMSD program on 512 cores is in the same range as the hypothetical trajectory frame generation rate, we believe that HiMach has the scalability and performance to support data analysis of millisecond-scale MD simulations.

VII. SUMMARY

Using the end-to-end time to solution—be it discovery, understanding, or intuition—as the metric of success of computer-based molecular modeling, we must take into account the time of post-simulation analysis that converts data to human-interpretable results in addition to the time spent in an MD simulation. Because sequential analysis methods are unable to deliver the necessary computational power, memory capacity, and I/O bandwidth to keep pace with the rapid development of parallel MD simulations, we must seek a more scalable solution.

In this paper, we have demonstrated—for the first time, we believe—how to build a scalable and flexible parallel framework to deal with massive MD trajectories, by combining and extending the strengths of an emerging enterprise computing technology (Google's MapReduce) and an existing sequential analysis tool (UIUC's VMD).

Even though HiMach, our prototype system, is developed specifically for MD trajectory analysis, its design principles and implementation techniques are potentially applicable to data analysis for other types of simulations, including gravitational simulations in astrophysics, particle simulations in plasma physics, smooth particle hydrodynamics simulations in fluid dynamics, and unstructured mesh-based PDE simulations.

TABLE IV. WEAK SCALING OF THE RMSD PROGRAM

Nodes	1	2	4	8	16	32	64	128
Cores	4	8	16	32	64	128	256	512
Frames	9600	19200	38400	76800	153600	307200	614400	1228800
Frames/core	2400	2400	2400	2400	2400	2400	2400	2400
Total time(s)	603	620	614	605	612	610	692	858
Map time (s)	465	480	476	466	469	505	522	634
Reduce time(s)	138	140	138	139	143	155	170	224
Map throughput (MB/s)	16.2	31.4	63.3	129.4	257.4	477.9	925.1	1521.2
Parallel efficiency (node)	100%	99%	98%	92%	91%	87%	78%	45%

ACKNOWLEDGMENT

We are grateful to Rebecca Kastleman for her assistance with this paper. We thank Phil Carns, Sam Lang, and Rob Ross from Argonne National Laboratory for answering technical questions specific to the PVFS2 file system that they have been developing. We also thank the anonymous reviewers for their comments that helped improve this paper.

REFERENCE

[1] Y.-S. Hwang, R. Das, J. H. Saltz, M. Hodoscek, B. R. Brooks, "Parallelizing Molecular Dynamics Programs for Distributed-Memory Machines," IEEE Computational Science and Engineering, vol. 02, no. 2, pp. 18-29, 1995.

[2] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kalé, and K. Schulten, "Scalable molecular dynamics with NAMD," J. Comp. Chem., vol. 26, no. 16, pp. 1781–1802, 2005.

[3] D. A. Case, T. E. Cheatham III, T. Darden, H. Gohlke, R. Luo, K. M. Merz Jr., A. Onufriev, C. Simmerling, B. Wang, and R. J. Woods, "The Amber biomolecular simulation programs," J. Comp. Chem., vol. 26, no. 16, pp. 1668–1688, 2005.

[4] M. Christen, P. H. Hünenberger, D. Bakowies, R. Baron, R. Bürgi, D. P. Geerke, T. N. Heinz, M. A. Kastenholz, V. Kräutler, C. Oostenbrink, C. Peter, D. Trzesniak, and W. F. van Gunsteren, "The GROMOS software for biomolecular simulation: GROMOS05," J. Comp. Chem., vol. 26, no. 16, pp. 1719–1751, 2005.

[5] K. J. Bowers, E. Chow, H. Xu, R. O. Dror, M. P. Eastwood, B. A. Gregersen, J. L. Klepeis, I. Kolossváry, M. A. Moraes, F. D. Sacerdoti, J. K. Salmon, Y. Shan, and D. E. Shaw, "Scalable algorithms for molecular dynamics simulations on commodity clusters," in Proc. 2006 ACM/IEEE Conf. on Supercomputing (SC06), Tampa, FL, November 2006.

[6] B. Hess, C. Kutzner, D. van der Spoel, and E. Lindahl, "GROMACS 4: Algorithms for highly efficient, load-balanced, and scalable molecular simulation," J. Chem. Theory and Comp., vol. 4, no. 3, pp. 435–447, 2008.

[7] B. G. Fitch, A. Rayshubskiy, M. Eleftheriou, T. J. C. Ward, M. E. Giampapa, M. C. Pitman, J. W. Pitera, W. C. Swope, and R. S. Germain, "Blue Matter: Scaling of *N*-body simulations to one atom per node," IBM J. Research and Develop., vol. 52, no. 1/2, pp. 145–158, 2008.

[8] R. Fine, G. Dimmler, and C. Levinthal, "FASTRUN: A special purpose, hardwired computer for molecular simulation," Proteins, vol. 11, no. 4, pp. 242–253, 1991, (erratum: 14(3): 421–422, 1992).

[9] S. Toyoda, H. Miyagawa, K. Kitamura, T. Amisaki, E. Hashimoto, H. Ikeda, A. Kusumi, and N. Miyakawa, "Development of MD Engine: High-speed accelerator with parallel processor design for molecular dynamics simulations," J. Comp. Chem., vol. 20, no. 2, pp. 185–199, 1999.

[10] M. Taiji, T. Narumi, Y. Ohno, N. Futatsugi, A. Suenaga, N. Takada, and A. Konagaya, "Protein Explorer: A petaflops special-purpose computer system for molecular dynamics simulations," in Proc. 2003 ACM/IEEE Conf. on Supercomputing (SC03), Phoenix, AZ, November 2003.

[11] D. E. Shaw, M. M. Deneroff, R. O. Dror, J. S. Kuskin, R. H. Larson, J. K. Salmon, C. Young, B. Batson, K. J. Bowers, J. C. Chao, M. P. Eastwood, J. Gagliardo, J. P. Grossman, R. C. Ho, D. J. Ierardi, I. Kolossváry, J. L. Klepeis, T. Layman, C. McLeavey, M. A. Moraes, R. Mueller, E. C. Priest, Y. Shan, J. Spengler, M. Theobald, B. Towles, and S. C. Wang, "Anton, a special-purpose machine for molecular dynamics simulation," in Proc. 34th Ann. Intl. Symp. on Computer Architecture, San Diego, June 2007, pp. 1–12.

[12] K. Barker, K. Davis, A. Hoisie, D. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho, "Entering the petaflop era: The architecture and performance of roadrunner," in Proc. 2008 ACM/IEEE Conf. on Supercomputing (SC08), Austin, TX, November 2008.

[13] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc.* 6th Symp. on Operating System Design and Implementation (OSDI04), San Francisco, CA, December 2004.

[14] ——, "MapReduce: Simplified data processing on large clusters," Comm. ACM, vol. 51, no. 1, pp. 107–113, 2008.

[15] W. Humphrey, A. Dalke, and K. Schulten, "VMD—Visual Molecular Dynamics," J. Mol. Graphics, vol. 14, no. 1, pp. 33–38, 1996.

[16] PDB File Format, http://www.rcsb.org/pdb/home/home.do.

[17] J. S. Vitter, "External memory algorithms and data structures: Dealing with massive data," ACM Computing Surveys, vol. 33, no. 2, pp. 209–271, 2001.

[18] Hadoop, http://hadoop.apache.org/core/.

[19] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in Proc. 19th ACM Symp. Operating Systems Principles, Bolton Landing, NY, October 2003.

[20] pyMPI, http://pympi.sourceforge.net/.

[21] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur, "PVFS: A parallel file system for Linux clusters," in *Proc. 4th Ann. Linux Showcase and Conf.*, Atlanta, GA, October 2000.

[22] PVFS, http://www.pvfs.org/.