Advection-Based Sparse Data Management for Visualizing Unsteady Flow

Hanqi Guo, *Student Member, IEEE*, Jiang Zhang, Richen Liu, Lu Liu, Xiaoru Yuan, *Member, IEEE*, Jian Huang, *Member, IEEE*, Xiangfei Meng, and Jingshan Pan

Abstract—When computing integral curves and integral surfaces for large-scale unsteady flow fields, a major bottleneck is the widening gap between data access demands and the available bandwidth (both I/O and in-memory). In this work, we explore a novel advection-based scheme to manage flow field data for both efficiency and scalability. The key is to first partition flow field into blocklets (e.g. cells or very fine-grained blocks of cells), and then (pre)fetch and manage blocklets on-demand using a parallel key-value store. The benefits are (1) greatly increasing the scale of local-range analysis (e.g. source-destination queries, streak surface generation) that can fit within any given limit of hardware resources; (2) improving memory and I/O bandwidth-efficiencies as well as the scalability of naive task-parallel particle advection. We demonstrate our method using a prototype system that works on workstation and also in supercomputing environments. Results show significantly reduced I/O overhead compared to accessing raw flow data, and also high scalability on a supercomputer for a variety of applications.

Index Terms—Flow visualization, Data management, High performance visualization, Key-value store

1 Introduction

Simulation of unsteady flow is key to many domains of computational research. As computing power at large scale has become very affordable, large and more complex unsteady flow data are nowadays prevalent. From the most practical point of view, however, the computing resources available for scientific visualization are typically much smaller than those used for the original simulations [3], thereby creating a challenge. That is, how to effectively carry out state-of-the-art flow visualization, for example to visualize a tera-byte level unsteady flow, without requiring a full-scale supercomputer.

In this work we focus on particle advection, which is a fundamental part of many flow visualization and analysis methods. Typical examples include texture-based methods [23], geometry-based methods [25], Finite-Time Lyapunov Exponent (FTLE) analysis [18] and source-destination queries [22]. Among these methods, data access is the known bottleneck of performance, scalability and space-efficiency. Many existing implementations of advection-based flow visualization store data based on the same grid as used by the original simulation. We address the multi-faceted bottleneck by changing the grid-centered data model to a key-value store based sparse data model.

We aim at developing a general mechanism by which fast, scalable visualization of large unsteady flow can be carried out on a relatively commonly available type of computing resource. For instance, as demonstrated in Section 5, we performed streak surface computation on a 860GB turbulent flow in 92 seconds (including I/O time) using 64 processors on a system that provided only $\sim$100MB/sec sustained I/O bandwidth. During the entire process, only 1% of the memory is used on each processing core. We also report similar results on several different kinds of platforms as well as for other use cases such as FTLE computation. This unprecedented result demonstrates that large scale unsteady flow analysis can be effectively and generally applied, requiring only a very frugal budget of computing resource. Potential use cases include extreme scale in-situ visualization and when the flow analysis application has to be carried out on-demand on a heavily loaded computing system.

The rationale of our research is based on a few observations. First, although the entire flow dataset is large, the working set is very small during particle advection. This is true even in large scale parallel settings. Second, while data access patterns appear random in unsteady flow visualization, it is practical and reliable to discover the working set on-demand by predicting data access needs based on current directions of advection. Third, with data partition as a standard practice in parallel flow visualization, changing from coarse-grained partition to fine-grained partition is feasible, especially when modern particle tracers increasingly employ local task queues to manage the start, stop, and communication of advection tasks. To this end, key-value store based sparse data management can greatly reduce the memory footprint of a parallel particle advection run, adapt well to the on-demand data access needs, and can be efficiently managed without impeding performance or scalability.

Data reorganization has been proven successful in mesh data and unstructured data management [36, 1]. In our method, the working set discovered at runtime is managed on the granularity of *blocklets*, which contain single cells or fine-grained blocks of cells in unsteady flow. The functional architecture of our methodology has two components: (1) a parallel key-value store that collectively and predictively manages blocklet I/O to maximally hide the I/O latency from advection computation, and (2) a group of completely independent task-parallel tracers. Every advection computing task is assigned to one and only one tracer, while each tracer owns and manages a large number of advection tasks in a task queue. Tracers request data by

- *Hanqi Guo is with Key Laboratory of Machine Perception (Ministry of Education), School of EECS, and Center for Computational Science and Engineering, Peking University. E-mail: hanqi.guo@pku.edu.cn.*
- *Jiang Zhang is with Key Laboratory of Machine Perception (Ministry of Education), School of EECS, and Center for Computational Science and Engineering, Peking University. E-mail: jiang.zhang@pku.edu.cn.*
- *Richen Liu is with Key Laboratory of Machine Perception (Ministry of Education), School of EECS, Peking University. E-mail: richen.liu@pku.edu.cn.*
- *Lu Liu is with Key Laboratory of Machine Perception (Ministry of Education), School of EECS, Peking University. E-mail: lu.liu@pku.edu.cn.*
- *Xiaoru Yuan is with Key Laboratory of Machine Perception (Ministry of Education), School of EECS, and Center for Computational Science and Engineering, Peking University. E-mail: xiaoru.yuan@pku.edu.cn.*
- *Jian Huang is with Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville. E-mail: huangj@eecs.utk.edu.*
- *Xiangfei Meng is with National Supercomputer Center in Tianjin, Binhai, Tianjin, China. E-mail: mengxf@nscc-tj.gov.cn.*
- *Jingshan Pan is with National Supercomputer Center in Jinan, Shandong, China. E-mail: panjsh@sdas.org.*

Manuscript received 31 Mar. 2014; accepted 1 Aug. 2014; date of publication xx xxx 2014; date of current version xx xxx 2014.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org.

querying the parallel key-value store, and keep the blocklets received in a local LRU cache.

In the past, the field has made great progress on improving the speed and scalability of unsteady flow visualization. For example, with I/O potentially takes up 90% time in a typical visualization task [21], researchers improved efficiency by using irregular data partition [12] and by optimizing file layout according to flow features [10, 11]. Load-balancing is difficult because optimal task distribution is hard to predict in unsteady flow visualization. Researchers improved scalability by using a multi-tier task distribution scheme that minimized global barriers and optimized the efficiency of task redistribution [22].

Recent implementations of large-scale unsteady flow analyses commonly employ static data distribution, and intermix data parallelism with task parallelism. In part, the key consideration is that moving large amounts of data on the fly is overwhelmingly expensive. This trend is true both on large supercomputer tier platforms and also on small to medium sized institutional HPC platforms. Regardless of scenario, great efforts are needed to fine tune both the algorithms and the configuration of the runs for best performance.

In contrast, our method uses static task distribution and employs task parallelism, by accurately and efficiently discovering the working set and passing data from parallel key-value store to tracers on-demand. Our implementation of parallel key-value store effectively manages in-core data and reduces the latency of data access in general. On top of that, our method constructs a graph based predictive hint model to accurately prefetch data from high a performance system (i.e. parallel file system or SSD enhanced file system) with built-in flow control, thereby effectively hides latency due to on-demand disk I/O.

In the remainder of the paper, we describe the background in Section 2. The sparse data management and the implementation details are described in Sections 3 and 4, respectively. We demonstrate the application cases in Section 5, and then conclude in Section 6.

2 Background

2.1 Advection-based Flow Visualization

Particle advection traces integral curves from user specified seed points. It is widely used in flow visualization methods, such as texture-based methods [23], geometry-based [25] methods, and flow feature extraction and tracking [32]. We divide these advection-based methods into two categorizes, namely local-range and full-range analysis.

Local-range analysis seeds locally or sparsely in a given spatiotemporal domain. For example, source-destination analysis is a local-range analysis useful for visualizing contaminant transport between two regions. For another example, flow surfaces [14] are ideal to illustrate unsteady flows as they directly show the behaviors of coherent moving particles.

Full-range analysis densely seeds over the entire simulation domain. In texture-based flow visualization, such as Line Integral Convolution (LIC) [7] and Unsteady Flow LIC (UFLIC) [34], integral curves are computed over all spatiotemporal samples in the domain. Densely seeded integral curves are also used for extracting Lagrangian Coherent Structures (LCS) with Finite-Time Lyapnov Exponents (FTLE) [15], for identifying differences between ensemble flow simulation results [17], and for selecting flow features according to pathline attributes [16]. Full-range analysis is computationally expensive, even with acceleration methods such as seeding with adaptive refinement [2] and reusing the already calculated parts [19].

This work improves the performance, scalability as well as space efficiency for both the local-range and the full-range analysis using blocklet-based sparse data management.

2.2 Parallel Particle Tracing

In general, parallel particle tracing methods can be categorized as task-parallelism, data-parallelism and hybrid methods that combine task- and data-parallelism.

Data-parallel particle tracing relies on data partitioning and distribution for load-balancing. For example, blocks can be statically assigned by round-robin [31], hierarchical clustered [37], partitioned according to boundaries of flow features [12], or partitioned over time steps to reduce memory cost in unsteady flow FTLE computation [28]. Task-parallel particle tracing revolves load-balancing by scheduling, such as workload estimation [29], dynamic load balancing [33, 27], etc. On-demand strategies are also used to reduce I/O costs and communication [8]. More recently, hybrid methods are considered to be more scalable. DStep [22] minimizes global barriers with a multi-tier task distribution and static data distribution. More recently in [17], Guo et al. demonstrated unprecedented ensemble flow analysis in Lagrangian scheme by extending DStep [22].

Common among parallel particle tracing methods, I/O is the typical bottleneck for scalability and performance. A few approaches have been proposed to improve data partitioning, including the block-based methods [31, 29], fine-grained partitioning [37, 12]. Optimized flow file layout can also be used to hide the latency of data retrieval [10, 11].

In this work, by transforming the in-core data management of particle advection to one using large-scale parallel key-value store, we have been able to further reduce the granularity of data retrieval to *blocklets*, and achieve better utilization of I/O and communication bandwidth.

2.3 Key-Value Store and Data Prefetching

Key-value store is a simple, general and powerful data model for data retrieval. It is widely used in NoSQL databases for big data applications in industry settings. Key-value store techniques can be roughly categorized into RAM cache vs. persistent store.

RAM cache key-value store could be as simple as a hash table, and there are also a series of production systems such as Memcached[1] and Redis[2]. For real-time and high-throughput applications, distributed key-value store is routinely used. Distributed hash table implementation in high-end HPC environments has been reported in the literature [24]. Persistent key-value store is used when data cannot fit into the memory, or it is necessary to persistently store the data. Popular solutions include BigTable[9], LevelDB[3], and Cassandra[4], etc.

A key efficiency technique among key-value store systems is Sorted String Tables (SST), which is based on log-structured storage [30]. In our work, we have implemented our own SST and extended it to work with global and parallel file systems. In addition to the traditional key-value store data models, we further developed a graph-based method to capture advection-based predictions of on-demand data access needs.

Our work is related to data caching and prefetching techniques. In parallel applications, I/O signatures are automatically traced and used to guide prefetching in MPI-IO [6]. Pre-execution techniques are also leveraged to hide parallel I/O latencies without perceivable patterns [13]. In our work, the data access patterns which are based on flow advection, are extracted and reused during the runtime. Although prefetching is usually beneficial, over-prefetch that saturations the system could impact performance. As demonstrated for remote visualization in [35], our system also needs to adaptively optimize prefetching parameters at runtime.

3 Advection-based Sparse Data Management for Particle Advection

Figure 1 illustrates the pipeline of our approach. The central part is sparse data management, which provides high performance data retrieval for the particle tracers in visualization and analysis applications. Our pipeline is generally applicable to post-processing (e.g. the input is files containing raw flow field data), and in-situ visualization as well as co-processing, where the input to our pipeline is unsteady flow data from live simulations. The data conversion process partitions the raw data into blocklets, which are indexed by their spatiotemporal locations. Tracer processes focus on advection computation, request blocklets by spatiotemporal indices, but are otherwise not involved in concurrently managing I/O, transfer and in-core storage aspects of the unsteady flow data.

[1] http://memcached.org
[2] http://redis.io
[3] http://code.google.com/p/leveldb
[4] http://cassandra.apache.org

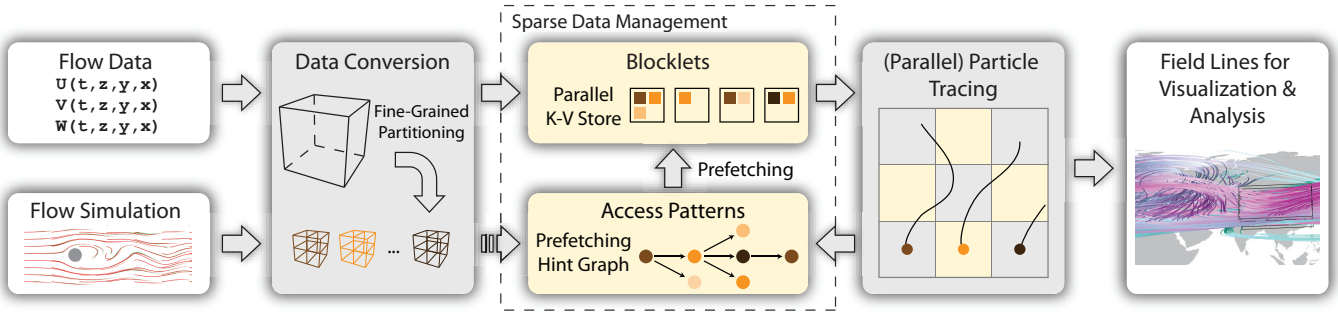

Fig. 1. The pipeline of the advection-based sparse data management. Raw data from storage or simulation is converted by fine-grained data partitioning, and then handled by the sparse data management system. The system provides high performance data access with prefetching, which brings both efficiency and scalability for field line tracing in various visualization tasks. The access patterns can be updated in runtime and effectively reused in later runs.

There are two mechanisms to manage data flow in our system, namely the parallel key-value store and the prefetching. The parallel key-value store is efficient and scalable. Better performance could be achieved by adding more computing resources, e.g. cores, memory, and I/O bandwidth, etc. The prefetching in our system uses a graph based model to capture and represent access patterns. We refer to this graph as *prefetching hints*. The prefetching hints are reusable and constructed on-the-fly. When prefetching hints have been gradually added as advection computation progresses, the reusability of prefetching hints considerably expedites data access.

3.1 Data Flow and Process Models

As shown in Figure 2, data primarily flows from the parallel key-value store to tracers. The required data entries (e.g. cells, blocklets) are fetched from the parallel key-value store, which performs actual data I/O from the file system. Tracers update the parallel key-value store by defining new prefetching hints, so that the prefetch hints graph are continuously constructed and expanded. Each tracer also employs a simplistic LRU-cache (implemented as a simple RAM key-value store) for faster data access. In this regard, data access by a tracer is quite similar to data access on a CPU with a multi-layer cache, where if the required entries are not cached by the tracer, then it takes a longer time to fetch from the parallel key-value store. If the parallel key-value store does not have the data either, then the data will be fetched from the file system and take an even longer period of time. Of course the parallel key-value store leverages the hints graph to aggressively prefetch to avoid cache-miss, while also implementing flow-control to avoid congesting the I/O system.

Parallelism is achieved with distributed processes and message passing. There are two types of processes: tracer processes and key-value store processes (k-v processes). Tracer processes and k-v processes use the same runtime API (Section 3.3) to request/retrieve and send data entries.

For load-balance, k-v processes partition the key space by hashing and round-robin assignment. A key k is assigned to a k-v process according to: $i = hash(k) \mod n$, where n is the number of k-v processes. Data requests from tracers are directly sent to the corresponding k-v processes, via point-to-point communication.

In addition to data requests, a tracer can also issue a prefetch hint as a chain of predicted data requests, where each request is identified by a hash key in the same way as in real data requests. Every prefetch hint is first sent to the k-v process in charge of the first key in the chain. That k-v process processes the first prefetch request, and then recursively forwards the remaining part of the chain to the k-v processes in charge of the first remaining prefetch request. This forwarding process iterates until the chain has been exhausted. After a blocklet has been prefetched, the k-v process keeps that data and also forwards that blocklet to the original tracer process that has requested that entry.

This process model is flexible for different environments. There are three scenarios. The first is to run tracer and k-v processes in the same MPI communicator (`MPI_COMM_WORLD`). The second way is to run the k-v processes as a service, where at the start of the analysis, the parallel key-value store has already been populated and can lead to even better analysis performance. This can be effectively implemented through dynamic process management as provided by MPI-2 (`MPI_Publish_name`), provided that the job scheduler of the computing facility supports that feature of MPI-2. The third is where an analysis process manages tracer and k-v threads, as opposed to tracer and k-v processes. To conservatively measure performance, our tests in this paper are run in the first way.

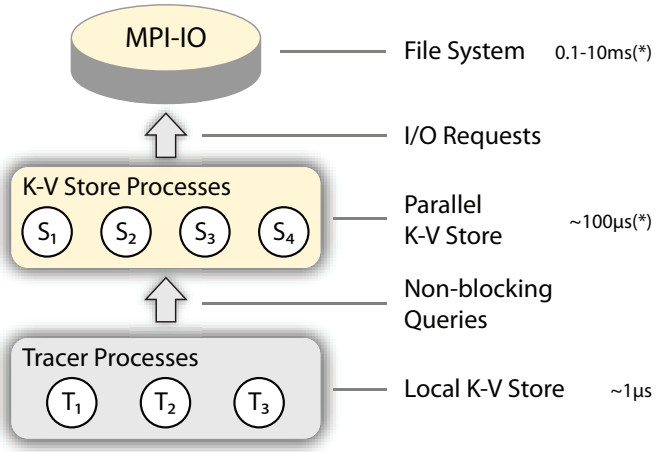

Fig. 2. The software architecture of the system. There are two kinds of roles in the process model, namely the tracer processes and key-value store processes (i.e. k-v processes). Logically, data retrieval from tracer processes follows the hierarchy from the local and parallel key-value stores, and finally the file system.

3.2 Prefetching and Access Pattern Reusability

Prefetching is the key to hide the latency of data access. The parallel key-value store not only loads and returns the requested data entries to the tracers, but also prefetches the highly possible entries to retrieve according to the access patterns.

Figure 3(a) illustrates the prefetching hint graph for a small dataset. The data domain is uniformly partitioned into 8×8 blocklets, and each of them is indexed by their X- and Y-coordinates marked on the boundaries. In the key-value store, the indices (X,Y) are used as the keys, and the blocklets data are stored as values. At the very beginning, there is no edges (prefetching hints) in this graph. During the particle tracing, the tracer issues a prefetching hint, if a particle passes from one blocklet (X,Y) to another (X',Y'). Thus, a prefetching hint is de-

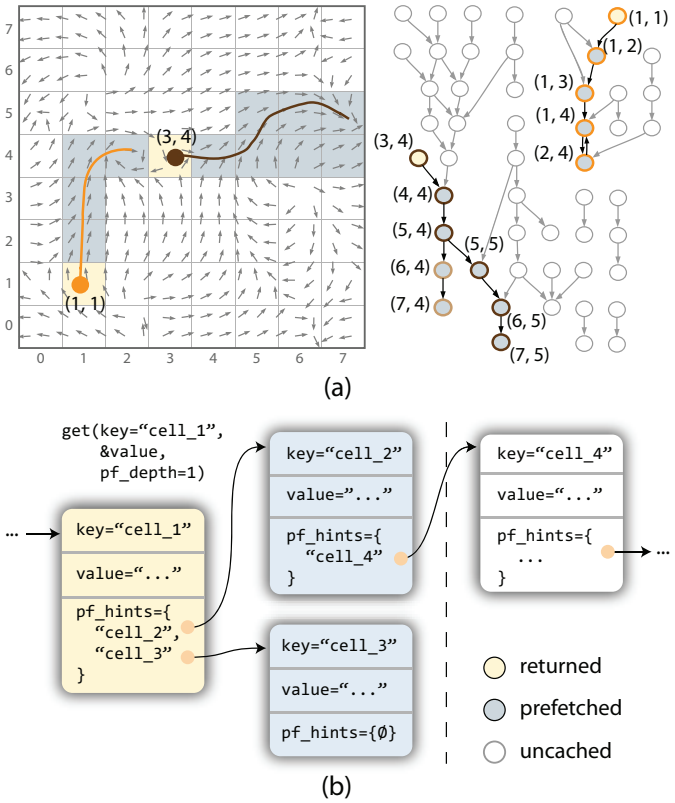

Fig. 3. The data structures in the sparse data management: (a) the prefetching hint graph of a small dataset; (b) key-value store with prefetching hints. The prefeching hints are issued by the tracers, and reused by the parallel key-value store.

fined as $(X,Y) \rightarrow (X',Y')$. As tracers run several times with some of different problem sets, more and more prefetching hints will join the graph. In a subsequent run, the prefetching hints will be reused. In Figure 3(a) left, the tracers compute two pathlines, which are seeded from the blocklets $(1,1)$ and $(3,4)$. The two blocks are requested by the tracer processes and fetched by the k-v processes. The k-v processes not only return the requested blocklets, but also prefetch the corresponding blocklets according to the prefetching hints. For example, when $(1,1)$ is requested, $(1,2)$, $(1,3)\ldots$ are also prefetched and sent to the tracers.

As shown in Figure 3(b), the access patterns are stored as prefetching hints in the data structure. Essentially, we extended the traditional key-value store model by adding a pre-feching hints field. Each data entry is stored as:

```
<key, value, pf_hints[]>,
```

where `pf_hints[]` are the keys of the data to prefetch. Essentially, a directed graph was constructed by the data entries and prefetching hints. Figure 3 illustrates an example of the prefetching hint graph and the data structures.

It is noteworthy that the sparse data management itself does not generate prefetching hints. Instead, the prefetching hints are issued by the tracer processes, and reused by the k-v processes as exemplified by the above example. By iteratively running tracers, more and more prefetching hints will be added into the store, and then accelerate the data access in further runs.

3.3 Runtime APIs

The runtime API of the sparse data management promotes a similar design to state-of-the-art key-value store system for data retrieval. In the tracers, the key is the spatiotemporal index of the cell/blocklet, and the value stores the actual data. Users can get or modify the values (actual data) by `get()` or `put()` functions. In addition several additional APIs are designed for tracers to define prefetching hints, as the runtime itself does not analyze the access patterns of the flow data. The tracers need to explicitly define the prefetching hints to accelerate further analysis. During the data retrieval, the prefetching depths could also be optionally assigned. The runtime will recursively load the data in the memory for fast access. The most important runtime APIs provided to the tracers are as follows:

`get(key, &value, pf_depth)` If the entry `key` locally exists, return the corresponding data as `value`. `pf_depth` is an integer which limits the maximum depth of prefetching. The runtime will prefetch the values corresponding to the keys in `pf_hints` by Breath-First Search (BFS) in a recursive manner. Prefetching is not performed if the depth equals to zero.

`put(key, value)` This function is equivalent to *put* in other key-value store systems. It sets the entry `key` to `value`. The entry is removed if `value` is empty.

`add_hint(key, key')` Add a prefetching hint for the entry associated with `key`. Essentially, it adds an edge `key->key'` to the prefetching hint graph.

`reset_hints(key)` Remove all prefetching hints of the entry that corresponds to `key`.

3.4 Task-Parallel Particle Advection with Sparse Data Management

Task-parallel particle advection is considered to be the most straightforward approach in visualizing large-scale unsteady flow, despite of complex I/O and data management. However, it is very difficult to scale due to the data access problems. Our method improves both I/O bandwidth-efficiency and scalability by managing the complex data store and access. It also simplifies the data management on the tracer side, thus users do not need to manually manage the memory use and synchronization. The only additional effort for users is to define the prefetching hints when the particle travels from one cell/blocklet to another, because the runtime cannot differentiate the traversal sequence when the runtime is concurrently accessed.

Algorithm 1 Tracing the pathline from a given seed with the sparse data management runtime.

```
function TRACE(seed, pathline)
    pathline.add(seed)                          ▷ Initialize the pathline
    while pathline.trace_size < max_trace_size do
        point = rk4(pathline.last_point)
        pathline.add(point)
        if not inside_domain(point) then        ▷ Finish the pathline
            break
        else if not inside_blocklet(current_blocklet, point) then
            next_blocklet = point_to_blocklet(point)
            if not get(next_blocklet) then
                Suspend and wait for next_blocklet
            end if
            add_hint(current_blocklet, next_blocklet)   ▷ Add a
prefetching hint
        end if
    end while
end function
```

The pseudo code of tracing the pathline from a given seed is shown in Algorithm 1. Massive particles can be traced with more processors in "share-nothing" and naive task-parallelism way. In our applications, especially for local-range analysis, the sparse data management only loads the data blocklets that are requested or potentially used for the analysis, without accessing the whole data. The sparse data management scheme not only simplifies the data access, but also enhances the memory and I/O bandwidth efficiency of task-parallel particle tracing.

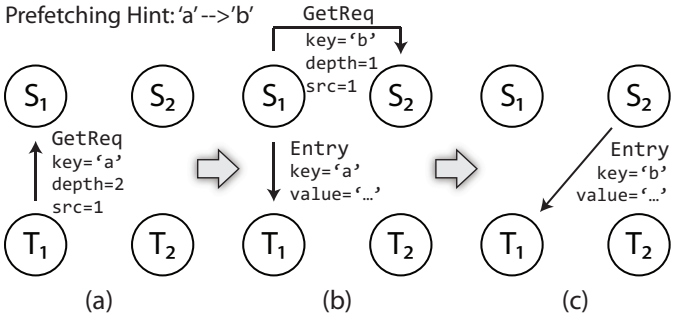

Fig. 4. An example of inter-process communication: (a) data entry 'a' is requested by an tracer process; (b) the corresponding k-v process returns entry 'a', and forward a request to prefetch 'b'; (c) the prefetched entry 'b' is also returned to the tracer process.

4 Implementation and Performance Evaluation

4.1 Inter-Process Communication

We have developed our prototype system using C++ with hybrid MPI/thread parallelism. All data communication are through message passing. In typical scenarios, the communication pattern follows the data access hierarchies shown in Figure 2. Based on the software architecture, there are three types of inter-process communications in our framework, namely tracer-to-store, store-to-tracer, and store-to-store communications. Notice that there is no communication between tracer processes. In general, the tracer processes send requests to the corresponding k-v processes, and the data entries are then sent back to the tracers. Data prefetching is done via store-to-store communications.

Through the runtime API, different types of messages are sent and processed, including `GetRequests` (requesting an entry), `SetRequests` (updating or inserting an entry), `HintRequests` (modifying the prefetching hint of an entry), `FlushRequests` (dumping all updated entries into the file system) and `ExitRequests` (exiting after all requests are processed), etc. There are other messages for delivering data entries, process states and statistics. `GetRequests` are the most important requests for internal data exchange among processes. Taking (pre)fetching for example (Figure 4), `GetRequests` can be used for both data requesting (tracer-to-store) and for forwarding prefetching requests (store-to-store).

Asynchronous communication is leveraged to reduce the latencies. All messages are queued, and then serialized and sent to the destination processes. After receiving, the messages are deserialized and then processed. We use Google protobuf library [5] to handle message serialization.

4.2 Local and Parallel Key-Value Store

Regardless of the roles of processes, each runtime instance keeps a local key-value store, The local key-value store, which acts like cache (e.g. Memcached), keeps the recently used data entries and greatly improves the data locality. For tracer processes, during the particle advection, the same entries are often accessed several times from the local key-value store, without fetching from the k-v processes.

In the prototype system, in order to limit the memory footprint, Least-Recently Used (LRU) policy is employed in the local key-value store. The implementation is based on a linked list and hash table, which keep the actual key-value pairs and the pointers to the linked list nodes, respectively. During the data access, the addresses of the entries are looked up in the hash table, and the orders of linked list are updated. When the cache is reaching the capacity, the least recently used entries in the tail of the linked list are removed. Thread-safety is provided for concurrent data access. The performance of the key-value store is comparable to decent production libraries. In our experiment, the set and get throughput of the local key-value store can reach up to 1.4M and 2.6M entries per second, respectively.

The parallel key-value store is built upon the local store. Data is partitioned and distributed by keys to achieve data-parallelism. During the lookup, the runtime first checks if the data entry is locally available, other wise sends a `GetRequest` to the destination process according to the data partitioning. Prefetching can further enhance the performance of the parallel key-value store, if applicable. Depending on the interconnection latency and runtime payload, the latency of getting an entry from the parallel key-value store is 10 to 100 microseconds in our experiments. The overall throughput is scalable if accessed concurrently. The parallel data store also provides larger space for data caching. For example, in a typical configuration with 16 k-v processes, if the size limit of each local key-value store is 1GB, then the overall size of the parallel store space will be 16GB.

4.3 Persistent Key-Value Store with Global SST

Our persistent key-value store is based on Sorted String Tables (SST), which is a simple log-structured storage solution [30] widely used in many popular production systems like BigTable [9]. In our prototype system, we extend SST to Global SST, where data is stored in a high performance file system, such as a parallel file system or one enhanced by SSD. We omitted data replication features for better performance.

The flow chart of reading and writing entries to SSTs is illustrated in Figure 5. Generally speaking, an SST is as simple as a data file with an index file (SST Index). The data file stores the actual values, while the SST Index records the `key:offset` pairs for fast access, where `offset` is the position of the value in the data file. The system may generate a series of SSTs during execution, and an SST is immutable after it is generated. When a key exists in multiple SSTs, the one in the latest SST takes precedence.

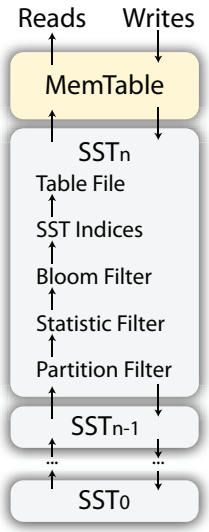

Fig. 5. Reading and writing data entries with Global SSTs.

To make SSTs into Global SSTs, we introduce the partition filter for membership test. In our method, it is imperative to be able to change N, the number of k-v processes that access the persistent key-value store. Traditionally, SSTs may be written with different data partitioning scheme defined by number of k-v processes N and the process id i. However, with such partitioning methods, given K entries and N slots, it is very hard to change N due to the expensive data redistribution. Consistent hashing [20] is proposed to solve this problem, yet data moving is still required. As we need all SSTs to be globally accessible to all k-v processes, we address this problem by adding a partition filter for SSTs. It checks whether a queried key belongs to the key space partition of an SST. The query could be accelerated by the membership test with partitioning schemes defined by N and i. For example, an SST was written by the i-th k-v process (N processes in total), then partition filter checks if $hash(key) \mod N$ equals to i.

Our global SST also uses other standard SST filters to accelerate membership test. Statistic filter tests whether the queried key is within the range of the key (the maximum and minimum keys in an SST). Bloom filter [4] is used to check whether a key belongs to an SST in constant time. Keys that have passed all of the filters will be actually queried in the SST indices using binary search.

During initialization, the stored Global SST meta data are loaded into memory across all k-v processes. SST indices are cached in memory, not all contents in an SST index need to be loaded.

Key-value pair query is straightforward. A k-v process first checks whether a queried key is available in local cache. If not, it then check whether the key passes the filter of SST_k, SST_{k-1}, ..., SST_1, where

[5] http://code.google.com/p/protobuf

SST_k and SST_1 are the latest and earliest ones, respectively. If the key does not pass the SST filters, then the k-v process returns the query with a "not found". If the key passes the SST filters, the k-v process then checks whether the key is in the indices of the SSTs. Finally, the k-v process reads the actual data (i.e. value) from file and return the value. The actual read operation in prefetching works similarly.

For insertion into key-value store, the data record goes directly to a standard MemTable (i.e. an in-core lookup table). We implemented our MemTable as a red-black tree. When the size of MemTable exceeds the limit or receives a flush request, it will be dumped into an SST and written to the file system asynchronously. We omitted fault tolerance features (e.g. commit logs) that are important to other applications for simplicity and for reducing system overheads. For example, when the tracer abnormally exits, losing an newly inserted prefetching hint does not affect the completeness of data for future uses.

In our experiments, the peak query performance of our global SST implementation can reach up to 900 key-value pair reads per second on a 7200rpm hard drive, and about 9000 pair reads per second on a Solid-State Drive (SSD) with SATA 3.0 connection.

5 Application Cases and Performance Evaluation

We demonstrate three driven application cases in flow visualization and analysis with our methods, including streak surface computation, source-destination queries, and FTLE computation. The first two are typical local-range analysis, as only a small portion of data needs to be actually accessed. The third is usually full-range analysis, yet sometimes be local if regions of interest and limited time scopes are focused. We also have different environments for the performance benchmark, ranging from single workstation to supercomputing environments. The datasets and test platforms are enumerated in Table 1.

Case	Application	Dataset	Size
I	Streak Surface Computation	Turbulence	0.82TB
II	Source-Destination Analysis	GEOS-5	1.34GB
III	FTLE Computation	Isabel	13.4GB

Table 1. The application cases and datasets.

The Beowulf cluster consists of eight computing nodes and one I/O node. Each computing node is equipped with two quad-core Intel Xeon E5520 CPUs, which work at 2.26GHz and with 48GB RAM. The I/O node shares a Lustre parallel file system to computing nodes, which is composed by only one OST (Object Storage Target). The storage device is a RAID6 disk array which contains 16 2TB HDDs. The interconnection between all nodes is InfiniBand QDR with 40Gbps theoretical bandwidth.

The single workstation platform has a quad-core Intel i5-4670 CPU, which operates at 3.40GHz. The main memory in this workstation is 16GB. The workstation is also equipped with two different consumable storage devices, including a Seagate 1TB HDD and an Intel 320 Series 120GB SSD. The two hard drives are connected to the motherboard with SATA 3 interface (6Gbps theoretical bandwidth). Random I/O performances on the both devices are reported to be 75$\sim$100 IOPs and $\sim$20K IOPs, respectively.

We also use the x86-based supercomputer in National Supercomputing Center in Jinan for the tests. It consists of 700 computing nodes, and each of them has two Intel Xeon E5675 processors (hexa-core, 3.06GHz) and 36GB main memory. Our allocation can use about 10% of the resources. The interconnection is InfiniBand QDR with a theoretical bandwidth of 40Gbps. SunWay Global File System (SWGFS) is provided for high performance parallel I/O.

5.1 Case I: Streak Surface Computation

Streak surfaces [14], which visualize unsteady flow by advecting continuously released particles from given seed curves, are capable of depecting the flow field over the entire lifetime. A streak surface is a mesh which consists of the locus of a set of particles that are advected by a time-varying flow field. As the integration goes on, new particles seeded at different time steps are added into the surface. The meshes are usually constructed and then refined to obtain better visual effects [5, 26]. The main bottleneck in streak surface computation is the massive particle tracing, whihc requires extensive access of the whole dataset.

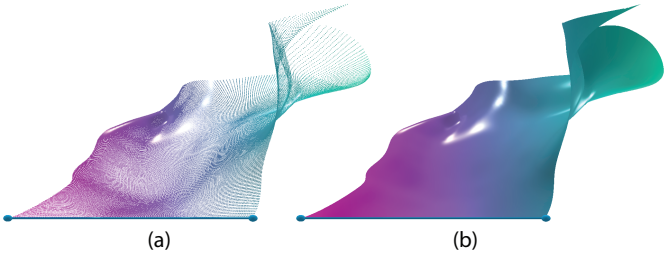

Fig. 7. Streak surface visualization of a terabyte-scale turbulence simulation: (a) the traced particles for surface generation; (b) the streak surface.

In the experiment, highly densely-seeded particles are generated from the seed line. In current implementation, the surface generation is done during the post-processing by triangulation, instead of using commonly-used quadrangular techniques [26], as the particles are dense enough to generate fine visualization results in this case. Further implementation could be added to adptively refine the surface on-the-fly. In the particle tracing stage, seeds are partitioned by round-robin and traced in different tracer processes in parallel. With the sparse data management scheme, the naive task-parallelism implementation can still keep scalability. As streak surface computation only needs to access a small portion of data, the on-demand and high performance data access greatly reduces the I/O cost for the computation.

Figure 7 shows the visualization results of a large scale turbulence simulation data with our method. A seed line is selected to show the features in the unsteady flow. The dataset is defined on a curvilinear grid, with the spatial resolution of $1024 \times 1024 \times 720$. In our experiments, we use 100 time steps for streak surface generation. As 3 velocity components are used, the effective data size is 0.82TB. The dataset is further partitioned and stored in the sparse data management system, and each blocklet contains $8 \times 8 \times 8 \times 1$ cells ($9 \times 9 \times 9 \times 2$ grid size). The overall key-value store size is about 2.2TB.

The benchmark results with different memory limits and number of processes are shown in Figure. 6. The most noteworthy point in this result is the data handling of TB-scale data with limited hardware resources. First, the sparse data management is extremely memory-efficient. For 64 processes, the total amount of the distributed memory cannot fit a terabyte, but the system is even workable with very small memory limit. In this memory-bounded case, it shows super-linear acceleration as the number of processes increases. Second, the system is I/O bandwidth efficient. On average, only 10GB out of 2.2TB data is accessed to compute a streak surface. With traditional methods which require in-core flow data management, given a decent fusion parallel I/O bandwidth (say 10GB/s), merely I/O time will take about 100 seconds ideally. Yet it is not possible to keep the TB-scale data in memory on relatively a small number of computing nodes. In our results, it only takes 92 seconds to finish the computation on 8 computing nodes (including the data access time), while keeping very low memory footprint.

Although the data scale of the key-value store is usually larger than traditional data formats, our method enables large-scale visualization and analysis with resource-bounded computing facilities. The main trade-off is between the storage space and the resources (memory footprint, I/O bandwidth, etc.), i.e. larger storage space for lower memory footprint, larger storage space for lower I/O access.

5.2 Case II: Source-Destination Queries

Source-destination queries are important to investigate teleconnection relationships of arbitrary two regions in the flow field. In real

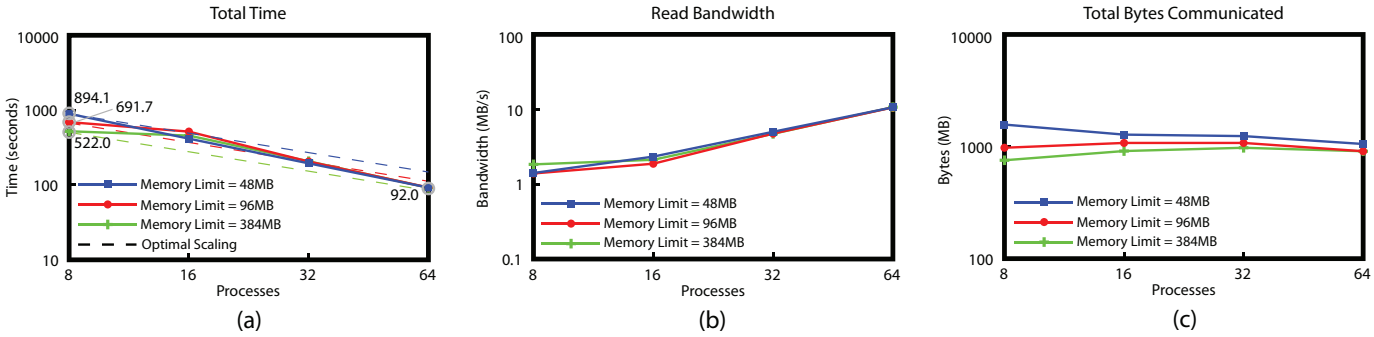

Fig. 6. The performance of the streak surface computation of the terabyte-scale turbulence simulation with different number of processes and configurations. The number of tracer processes and k-v processes are 1:1. Read bandwidth is the amount of bytes read divided by total running time, and total bytes communicated is the summation of sent messages sizes from all processes. Super-linear acceleration is observed when the memory limit is small. As the number of processes increases, the timings, under different memory limit, converge because the total amount of distributed memory is enough to fit the requested data.

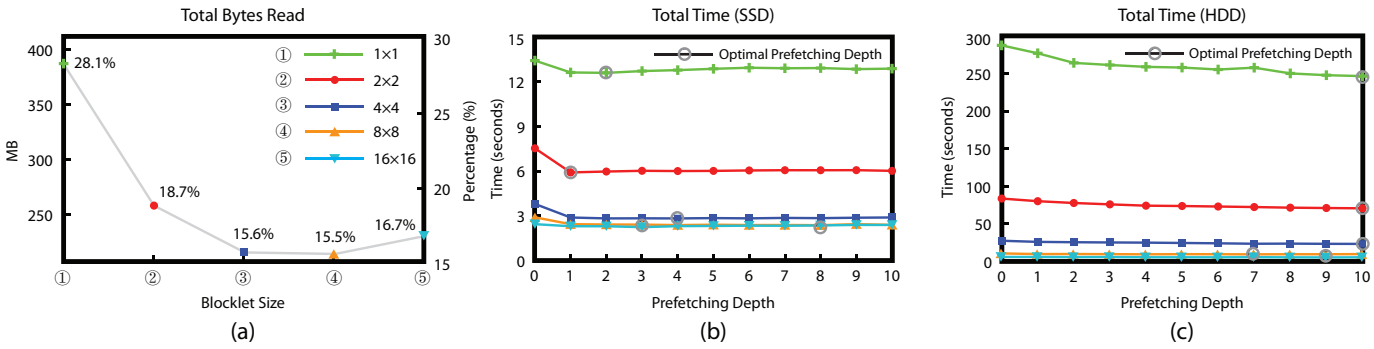

Fig. 9. Data access amount and performance on single workstation with different partitioning granularities (GEOS-5 dataset): (a) the data access amount with different blocklet size; (b) the performance on SSD hard disk; (c) the performance on HDD hard disk. Optimal prefetching depths are highlighted in (b) and (c).

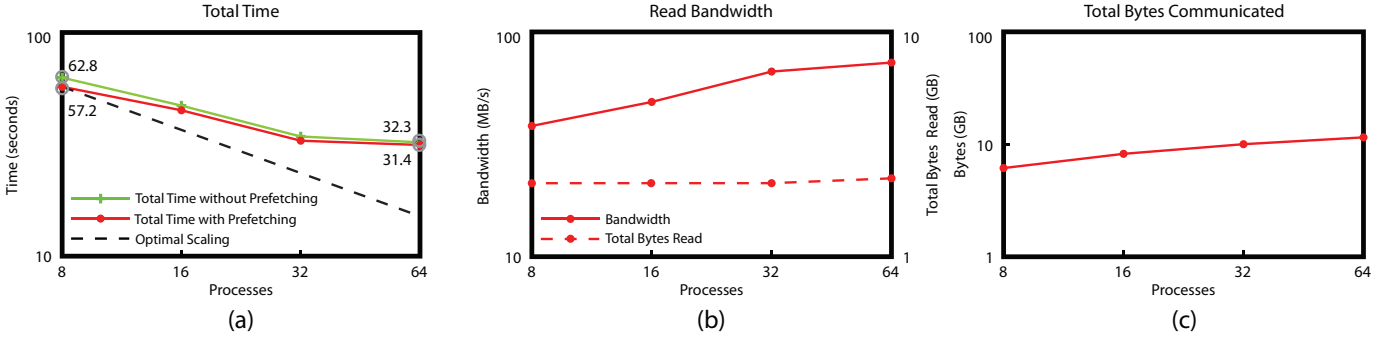

Fig. 10. The performance of the source-destination queries of the GEOS-5 dataset with different number of processes and configurations. The number of tracer processes and k-v processes are 1:1. Read bandwidth is the amount of bytes read divided by total running time, and total bytes communicated is the summation of sent messages sizes from all processes.

applications, end users can make hypotheses on source-destinations by defining regions, and validate them by visualizing the field lines between the two regions. For example, the CO_2 exchange from Northern and Southern Hemispheres, pollution dispersion between the source to sensitive regions, etc. The queries are completed by tracing dense seeded pathlines from the source region. In our experiment, both memory- and I/O-bandwidth efficiency can be achieved with on-demand data access and fine-grained data partitioning.

In this case, we use simulation result from GEOS-5 global climate model which is developed by NASA Goddard Space Flight Center. The spatial resolution of this model is $1^\circ \times 1.25^\circ$, with 72 vertical pressure levels. The dataset consists 24 monthly averaged results of the simulation. The output of this model is stored in hybrid-sigma grid, which requires a customized interpolation scheme for particle tracing with Runge-Kutta 4th order numerical integral method. 4 out of 32 variables are used for the analysis, which are necessary for calculating the three components of the wind speed vector.

In the conversion stage, the raw data is partitioned into small blocklets of cells. In the optimal experiment, each blocklet contains $8 \times 8 \times 71 \times 1$ cells ($9 \times 9 \times 72 \times 1$ grids). Instead of using individual cells, we have to keep a whole vertical column of data, in order to precisely interpolate the attribute values in the hybrid-sigma grid.

We select two regions to see how massless particles trace with the wind field from North America to East Asia within a month. 200 pathlines are uniformly seeded every month (wall clock) from the source region, and then advected in the entire domain. In Figure 8, there

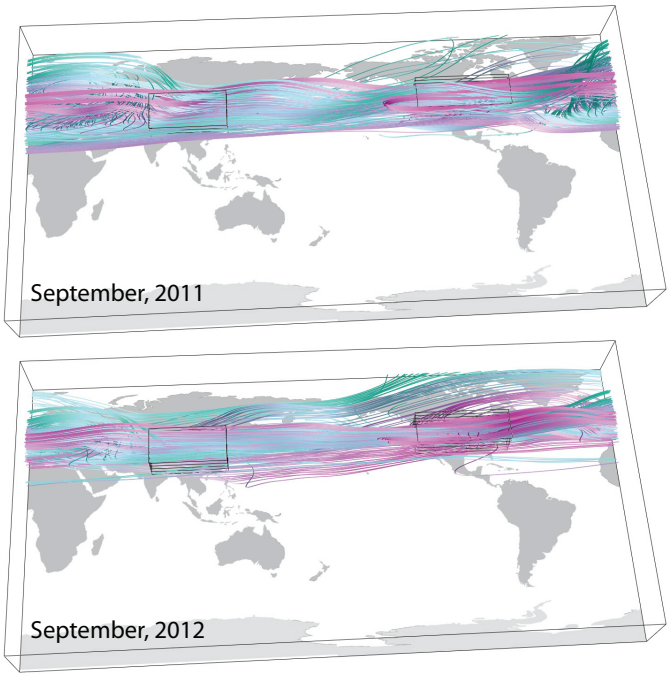

Fig. 8. Source-destination query on GEOS-5 dataset. Every month there are 200 pathlines initially seeded in red and gradually changed to be green as they advect in the domain. Results that meet the query from North America to East Asia within two months are visualized, respectively.

are 309 pathlines in total that meet the source-destination query in September 2011 and September 2012.

We test the performance on GEOS-5 dataset on a single workstation to see the I/O bandwidth efficiency of our method (Figure 9). Three tracer processes and one k-v process run on the quad-core CPU. For each test run, we dump Linux page caches to obtain fair timings. The sparse data management greatly decreases the amount of the data access. With the optimal configuration, only 2.77% blocklets are actually accessed for this specific query. Although the size of key-value store is usually larger than the raw data, our method can still save 84.5% amount of bytes to read from the file system. In the results, SSD usually outperforms HDD, because the key-value store implementation heavily relies on random I/O operations. Prefetching also brings benefits. We ran different queries multiple times, thus the access patterns are written and effectively reused in each run. With prefetching, the overall computation time is decreased by 18.75%. In addition, we have also tested the performance on the Beowulf cluster (Figure 10). Prefetching also improves the scalability in parallel environments.

5.3 Case III: FTLE Computation

FTLE is widely used to measure the separation and to extract Lagrangian Coherent Structures (LCS) in flow field. The FTLE value at a certain point indicates the possibility to diverge from particles around this point within a finite time scope T. However, FTLE computation is expensive due to the massive particle tracing from dense sample points.

In this case, we demonstrate scalable FTLE computation with task-parallel particle tracing based on the sparse data management. Previously, both sample reduction [2] and massively data-parallelism [28] techniques are presented to accelerate the FTLE computation. Although task-parallel particle tracing (partitioning over seeds) is the most straightforward way to achieve parallelism in a "share-nothing" manner, the efficiency and scalability are limited. By improving the memory and I/O bandwidth-efficiency with the sparse data management, we are capable of observing good scalability of this method.

We trace the uniformly-seeded pathlines and compute the FTLE

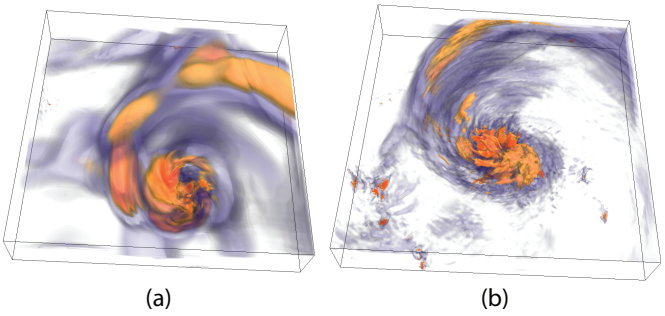

Fig. 11. FTLE field of Isabel dataset at (a) time step 0 and (b) time step 20.

field of the Hurricane Isabel dataset, which is from an atmospheric simulation. The spatial resolution of the data grid is $500 \times 500 \times 100$. 48 hourly averaged data is stored in separated files. Figure 11 are the visualization results of the FTLE field. We choose the optimal blocklet size $4 \times 4 \times 4 \times 1$ for data conversion. Pathline seeds are uniformly placed in the data domain.

The scalability is studied on the supercomputing environment. The performance with different number of processes are shown in Figure 12. Without complicated task distribution strategies, our method still shows good scalability as number of processes increases. Prefetching greatly improves both performance and scalability in this case. With all number of processes, the run with prefetching outperforms the run with no prefetching. On 256 processes, the run with no fetching does not accelerate anymore, but the timings with prefetching are keeping descending as the number of processes increases.

FTLE is usually regarded as full-range analysis, as users often tend to visualize the global distributions. Similar to FTLE, the dense-seeded pathline computation can also be extended to other analysis, including ensemble flow simulation [17] and pathline attributes [16]. Yet, such tools are also used locally when a region of interest is focused, or a limited time scope is chosen. Benefits in local-range analysis with our method are still effective when the particles are traced in a local region.

5.4 Observations and Discussion

All cases demonstrated in this section are from typical applications from flow visualization and analysis. The advection-based sparse data management greatly benefits the data access in various platforms, ranging from single workstation to supercomputing environments. In summary, the key observations in the three cases are as follows:

- **Case I:** Our method is memory-efficient, and it enables scalable visualization of large-scale unsteady flow (e.g. terabyte scale turbulence flow) while requiring a very limited amount of hardware resources.
- **Case II:** Our method is I/O bandwidth efficient. It greatly reduces the amount of I/O with fine-grained and on-demand data access.
- **Case III:** Our method greatly improves both performance and scalability of the naive task-parallel particle tracing.

There are primarily two parameters for performance tuning, including the prefetching depth and the blocklet size. Both of them are usually empirically determined. As studied by Sisneros et al. [35], in multi-leveled cache architecture, there is no theoretical ways to choose optimal prefetching parameters. For example, in processor caching, the prefetching policies and word sizes are often determined by experiments. In flow visualization and analysis, as the data features access patterns are even more complex, it requires even more efforts to achieve optimal performance. The prefetching depth cannot be too large, because over-prefetching may saturate the bandwidth[35]. In

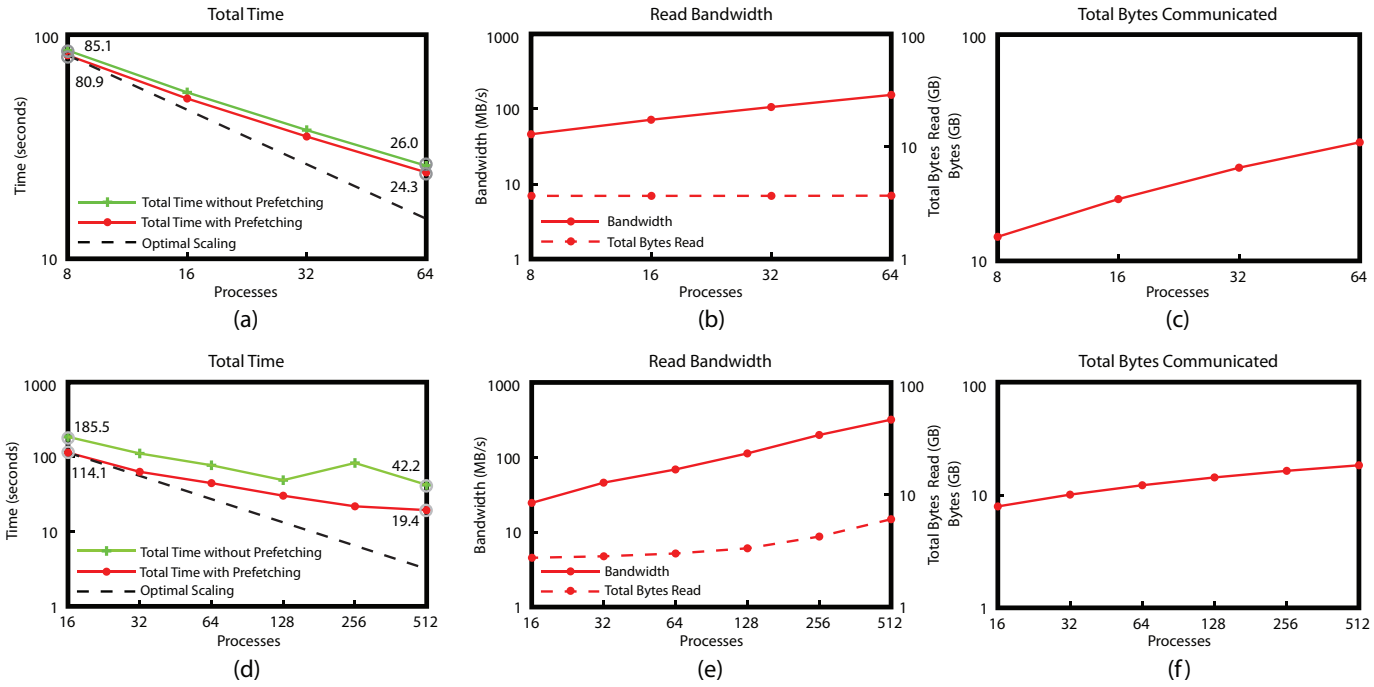

Fig. 12. The performance of the FTLE computation of the Isabel dataset with different number of processes and configurations. (a), (b) and (c) are tested on the Beowulf cluster, while (d), (e) and (f) are tested on the supercomputer. The number of tracer processes and k-v processes are 1:1. Performance with and without prefetching is shown in (a) and (d). (b) and (e), (c) and (f) display the read bandwidth (with total bytes read) and total bytes communicated with prefetching, respectively. Both performance and scalability are improved in the task-parallel particle tracing using prefetching.

general, it is usually more memory-efficient to use smaller blocklet size. However, due to the additional ghost grids storage in blocklets, it may require additional data access if the blocklets are too small. Additional space and time costs are also incurred by the key-value store.

6 Conclusions

In this work, we have explored a novel advection-based sparse data management scheme for visualizing large unsteady flow data. We have made extensive use of key-value store to manage blocklets in memory, and used hint graphs for predictively guide data I/O. When visualizing terabyte-scale unsteady flow data, our approach achieved significantly better space efficiency in memory, as well as high performance and scalability. Our results demonstrate that such sparse data management greatly increases the scale of local-range analyses that are feasible for resource-constrained systems, while it also improves the scalability of full-range task-parallel particle tracing.

For future work, we want to extend our method for use by in-situ visualization. Our data management can be extended to handle irregular and unstructured grid data. We would like to also improve the fine-grained data partitioning by considering adaptive mesh refinement.

Acknowledgments

The authors would like to thank Prof. Qingdong Cai for providing the TB-scale turbulence simulation data. This work is supported by NSFC No. 61170204. This work is also partially supported by NSFC Key Project No. 61232012 and the "Strategic Priority Research Program - Climate Change: Carbon Budget and Relevant Issues" of the Chinese Academy of Sciences Grant No. XDA05040205.

References

[1] O. O. Akande and P. J. Rhodes. Iteration aware prefetching for unstructured grids. In *Proceedings of the 2013 IEEE International Conference on Big Data*, pages 219–227, 2013.

[2] S. S. Barakat and X. Tricoche. Adaptive refinement of the flow map using sparse samples. *IEEE Trans. Vis. Comput. Graph.*, 19(12):2753–2762, 2013.

[3] E. Bethel, J. van Rosendale, D. Southard, K. Gaither, H. Childs, E. Brugger, and S. Ahern. Visualization at supercomputing centers: The tale of little big iron and the three skinny guys. *IEEE Comput. Graph. Appl.*, 31(1):90–95, 2011.

[4] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[5] K. Bürger, F. Ferstl, H. Theisel, and R. Westermann. Interactive streak surface visualization on the GPU. *IEEE Trans. Vis. Comput. Graph.*, 15(6):1259–1266, 2009.

[6] S. Byna, Y. Chen, X.-H. Sun, R. Thakur, and W. Gropp. Parallel I/O prefetching using MPI file caching and I/O signatures. In *SC08: Proceedings of the ACM/IEEE Conference on Supercomputing*, pages 44:1–44:12, 2008.

[7] B. Cabral and L. C. Leedom. Imaging vector fields using line integral convolution. In *Proceedings of SIGGRAPH 1993*, pages 263–270, 1993.

[8] D. Camp, C. Garth, H. Childs, D. Pugmire, and K. I. Joy. Streamline integration using MPI-hybrid parallelism on a large multicore architecture. *IEEE Trans. Vis. Comput. Graph.*, 17(11):1702–1713, 2011.

[9] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. BigTable: A distributed storage system for structured data. In *OSDI'06: Proceedings of Symposium on Operating Systems Design and Implementation*, pages 205–218, 2006.

[10] C.-M. Chen, B. Nouanesengsy, T.-Y. Lee, and H.-W. Shen. Flow-guided file layout for out-of-core pathline computation. In *Proceedings IEEE Symposium on Large Data Analysis and Visualization 2012*, pages 109–112, 2012.

[11] C.-M. Chen, L. Xu, T.-Y. Lee, and H.-W. Shen. A flow-guided file layout for out-of-core streamline computation. In *Proceedings of IEEE Pacific Visualization Symposium 2012*, pages 145–152, 2012.

[12] L. Chen and I. Fujishiro. Optimizing parallel performance of streamline visualization for large distributed flow datasets. In *Proceedings of Pacific Visualization Symposium 2008*, pages 87–94, 2008.

[13] Y. Chen, S. Byna, X.-H. Sun, R. Thakur, and W. Gropp. Hiding I/O latency with pre-execution prefetching for parallel applications. In *SC08: Proceedings of the ACM/IEEE Conference on Supercomputing*, pages 40:1–40:10, 2008.

[14] M. Edmunds, R. S. Laramee, G. Chen, N. Max, E. Zhang, and C. Ware.

Surface-based flow visualization. *Computers & Graphics*, 36(8):974–990, 2012.

[15] C. Garth, F. Gerhardt, X. Tricoche, and H. Hagen. Efficient computation and visualization of coherent structures in fluid flow applications. *IEEE Comput. Graph. Appl.*, 13(6):1464–1471, 2007.

[16] H. Guo, F. Hong, Q. Shu, J. Zhang, J. Huang, and X. Yuan. Scalable Lagrangian-based attribute space projection for multivariate unsteady flow data. In *Proceedings of IEEE Pacific Visualization 2014*, pages 33–40, 2014.

[17] H. Guo, X. Yuan, J. Huang, and X. Zhu. Coupled ensemble flow line advection and analysis. *IEEE Trans. Vis. Comput. Graph.*, 19(12):2733–2742, 2013.

[18] G. Haller. Distinguished material surfaces and coherent structures in three-dimensional fluid flows. *Physica D*, 149(4):248–277, 2001.

[19] M. Hlawatsch, F. Sadlo, and D. Weiskopf. Hierarchical line integration. *IEEE Trans. Vis. Comput. Graph.*, 17(8):1148–1163, 2011.

[20] D. R. Karger, E. Lehman, F. T. Leighton, R. Panigrahy, M. S. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *STOC'97: Proceedings of the ACM Symposium on the Theory of Computing*, pages 654–663, 1997.

[21] W. Kendall, J. Huang, T. Peterka, R. Latham, and R. B. Ross. Toward a general I/O layer for parallel-visualization applications. *IEEE Comput. Graph. Appl.*, 31(6):6–10, 2011.

[22] W. Kendall, J. Wang, M. Allen, T. Peterka, J. Huang, and D. Erickson. Simplified parallel domain traversal. In *SC11: Proceedings of the ACM/IEEE Conference on Supercomputing*, pages 10:1–10:11, 2011.

[23] R. Laramee, H. Hauser, H. Doleisch, B. Vrolijk, F. Post, and D. Weiskopf. The state of the art in flow visualization: dense and texture-based techniques. *Comput. Graph. Forum*, 23(2):203–222, 2004.

[24] T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, and I. Raicu. ZHT: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table. In *IPDPS'13: Proceedings of IEEE International Symposium on Parallel and Distributed Processing*, pages 775–787, 2013.

[25] T. McLoughlin, R. Laramee, R. Peikert, F. Post, and M. Chen. Over two decades of integration-based, geometric flow visualization. *Comput. Graph. Forum*, 29(6):1807–1829, 2010.

[26] T. McLoughlin, R. S. Laramee, and E. Zhang. Constructing streak surfaces for 3D unsteady vector fields. In *SCSG'10: Proceedings on Spring Conference on Computer Graphis*, pages 17–26, 2010.

[27] C. Mueller, D. Camp, B. Hentschel, and C. Garth. Distributed parallel particle advection using work requesting. In *Proceedings IEEE Symposium on Large Data Analysis and Visualization 2013*, pages 109–112, 2013.

[28] B. Nouanesengsy, T.-Y. Lee, K. Lu, H.-W. Shen, and T. Peterka. Parallel particle advection and FTLE computation for time-varying flow fields. In *SC12: Proceedings of the ACM/IEEE Conference on Supercomputing*, pages 61:1–61:11, 2012.

[29] B. Nouanesengsy, T.-Y. Lee, and H.-W. Shen. Load-balanced parallel streamline generation on large scale vector fields. *IEEE Trans. Vis. Comput. Graph.*, 17(12):1785–1794, 2011.

[30] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Inf.*, 33(4):351–385, 1996.

[31] T. Peterka, R. B. Ross, B. Nouanesengsy, T.-Y. Lee, H.-W. Shen, W. Kendall, and J. Huang. A study of parallel particle tracing for steady-state and time-varying flow fields. In *IPDPS11: Proceedings of IEEE International Symposium on Parallel and Distributed Processing*, pages 580–591, 2011.

[32] F. Post, B. Vrolijk, H. Hauser, R. Laramee, and H. Doleisch. The state of the art in flow visualisation: Feature extraction and tracking. *Comput. Graph. Forum*, 22(4):1–17, 2003.

[33] D. Pugmire, H. Childs, C. Garth, S. Ahern, and G. H. Weber. Scalable computation of streamlines on very large datasets. In *SC09: Proceedings of the ACM/IEEE Conference on Supercomputing*, pages 16:1–16:12, 2009.

[34] H.-W. Shen and D. L. Kao. UFLIC: a line integral convolution algorithm for visualizing unsteady flows. In *Proceedings of IEEE Visualization 1997*, pages 317–322, 1997.

[35] R. Sisneros, C. Jones, J. Huang, J. Gao, B.-H. Park, and N. F. Samatova. A multi-level cache model for run-time optimization of remote visualization. *IEEE Trans. Vis. Comput. Graph.*, 13(5):991–1003, 2007.

[36] S.-E. Yoon, P. Lindstrom, V. Pascucci, and D. Manocha. Cache-oblivious mesh layouts. *ACM Trans. Graph.*, 24(3):886–893, 2005.

[37] H. Yu, C. Wang, and K.-L. Ma. Parallel hierarchical visualization of large time-varying 3D vector fields. In *SC07: Proceedings of the ACM/IEEE Conference on Supercomputing*, pages 24:1–24:12, 2007.