Legion-based Scientific Data Analytics on Heterogeneous Processors

Lina Yu, Hongfeng Yu
University of Nebraska-Lincoln

***Abstract*—We present a study of scientific data analytics on heterogeneous architectures using the Legion runtime system. Legion is a new programming model and runtime system targeting distributed heterogeneous architectures. It introduces logical regions as a new abstraction for describing the structures and usages of program data. We describe how to leverage logical regions to express important properties of program data, such as locality and independence, for scientific data analytics that can consist of multiple operations with different data types. Our approach can help users simplify programming on the data partition, data organization, and data movement for distributed-memory heterogeneous architectures, thereby facilitating a simultaneous execution of multiple analytics operations on modern and future supercomputers. We demonstrate the scalability and the usability of our approach by a hybrid data partitioning and distribution scheme for different data types using both CPUs and GPUs on a heterogeneous system.**

***Keywords*-scientific data analytics; heterogeneous processors; Legion**

I. Introduction

The high computation expense of large-scale scientific applications has stimulated the advance of supercomputing systems over the last few decades. Through parallelism, higher performance and throughput have been achieved to enable scientists to simulate complex physical and chemical phenomena at an unprecedented scale. However, few frameworks have been established to allow effective analysis of large-scale scientific simulation data on modern parallel architectures that use both heterogeneous processors and deep, complex memory hierarchies. Besides the problem of division and assignment of computation in parallel programming, one of the most difficult issues may be the placement and movement of data, especially in heterogeneous, distributed machines with deep memory hierarchies:

First, communication costs are a critical issue for parallel system and software designers to consider. The selection of a parallel algorithm has a major impact on communication requirements between compute nodes. In practice, a scientific analytics workflow typically consists of multiple operations that intrinsically incur different communication or data movement requirements between compute nodes.

Second, the latest supercomputers are increasingly complex and often are composed of deep, distributed memory hierarchies and heterogeneous processing units [1]. With an increasingly demand on suitable analytics capabilities to explore large data at high interactivity and fidelity, we hope to make use of all processing units on supercomputers. However, having the data organized correctly within the machine is becoming even more difficult. In a scientific data analytics circumstance, we may need to partition data on a 2D image space for visualization or on a 3D object space for statistical analysis. An appropriate organization of the data on a complex heterogeneous memory hierarchy becomes a big challenge and has a significant impact on the performance and scalability of an analytics workflow.

Although many techniques have been proposed to tackle the heterogeneity of supercomputers, most efforts have focused on the improvement of the *scalability* of specific techniques, while the *usability* has not been fully investigated. When using these techniques, programmers may be still required to explicitly address complex data partitioning and distribution across heterogeneous processors. Therefore, it is often a non-trivial task to apply these techniques to build an analytics workflow in practice. Legion is a programming model and runtime system for describing hierarchical organizations of both data and computation at an abstract level [2]. Unlike other programming systems where these properties are managed by programmers, Legion provides abstractions for programmers to explicitly declare properties of program data including organization, partitioning, privileges, and coherence. Furthermore, Legion can implicitly extract parallelism and issue the necessary data movement operations in accordance with the application-specified data properties, thereby removing a significant burden from programmers [3]. A separate mapping interface allows programmers to control how data and computation are placed onto the actual memories and processors of a specific machine.

In this paper, we investigate the feasibility of using Legion to perform analytics for large-scale scientific data on heterogeneous processors. We implement a parallel scientific data analytics framework running on both GPUs and CPUs architectures using the Legion runtime system. We describe the mechanism of expressing data locality and independence provided by logical regions in the Legion programming model. Our solution makes it easy for users to implement a complex analytics workflow consisting of multiple operations with different data types, but ignores the data management details (e.g., data partitioning and distribution, data communication, etc.). We illustrate this framework using several representative analytics operations, and present the experimental results on a heterogeneous supercomputer.

978-1-4673-9005-7/16/$31.00 ©2016 IEEE

II. Related Work

A. Parallel Computer Memory Architectures

There are three memory architectures commonly used in parallel computing: shared memory, distributed memory and hybrid distributed-shared memory [4]. Shared memory parallel programming models are commonly based on threads that have both private and shared variables (e.g., OpenMP [5]). In a distributed memory system, each processor only accesses limited memory. Message Passing Interface (MPI) is often used to exchange data among processors through communications. Data transfer usually requires cooperative operations to be performed by each processor [6].

The largest and fastest computers in the world today employ both shared and distributed memory architectures [7]. Typically, memory is shared among multiple cores and/or graphics processing unit (GPU) within a node, while multiple nodes are inter-connected via networking. Current trends seem to indicate that this memory architecture will continue to prevail and increase at the high end of computing for the foreseeable future. Hybrid memory system combines the advantages of shared memory and distributed memory systems, but also significantly increases programming complexity.

B. Modern Parallel Languages

Most modern parallel languages have mainly focused on providing language-based approaches to specify concurrency and data distributions. The most seminal works are X10 [8], Chapel [9], Fortress [10] and Sequoia [11]. X10 is an Asynchronous Partitioned Global Address Space (APGAS) language featuring task parallelism and locality control by the usage of places [8], [12]. Chapel is an emerging programming language designed for productive parallel computing at scale [9]. Chapel's locale type enables users to specify and reason about the placement of data and tasks on a target architecture in order to tune for locality and affinity. Sequoia gives a programmer explicit control over data locality and communication for programming machines with multi-level memory hierarchies. It abstractly exposes hierarchical memory in the programming model and provides language mechanisms to describe communication vertically through the machine and to localize computation to particular memory locations within it [11].

Another programming system that makes use of an explicit mapping interface is the Halide language and compiler [13] for describing and optimizing image processing pipelines. Halide programs express operations that are performed on two-dimensional images and the Halide compiler optimizes the implementation of these pipelines for different architectures.

C. Scientific Data Analytics

Researchers have developed scalable solutions for individual analytics operations, such as rendering [14]–[16], querying [17]–[20], and so on. However, in practice, a scientific data analytics workflow can consist of multiple operations, which requires researchers to holistically address different data management requirements for different operations. Bennett et al. [21] presented a combined *in-situ* and *in-transit* framework to support multiple large-scale scientific analytics operations (e.g., visualization, statistics, and topological analysis). Sun et al. [22] used application knowledge to adaptively place data for staging-based coupled scientific workflows. Although the effectiveness of these approaches has been demonstrated with real-world large-scale applications, they were mainly developed for conventional distributed CPU-based architectures.

Researchers have also developed techniques for optimizing data processing on heterogeneous systems. For example, Wu et al. [23] presented pipeline frameworks to execute different operations on heterogeneous cloud computing environments. Pérez et al. [24] developed an OpenCL-based library to simply programming and load balancing of data parallel applications on heterogeneous systems. Breß et al. [25] used a hardware-oblivious data processing engine to optimize the operator placement in a heterogeneous hardware environment. However, these techniques targeted comparably small scale systems, and cannot be directly applied on large heterogeneous supercomputers.

III. The Legion Programming Model

Today's machines often have more than one type of processors (e.g., CPUs and GPUs) and future architectures will likely have specialized accelerators. *Legion* is a programming model and runtime system designed for decoupling the specification of parallel algorithms on distributed heterogeneous architectures [2].

Legion manages heterogeneity by allowing code to be retargeted to different types of processors. Because running on the target class of machines requires distributing both computation and data, Legion presents the abstraction of *logical region* for describing the structure of program data. Logical regions allow programmers to express both locality in data structures and independence between tasks that use disjoint logical regions. Given the knowledge of both the structures of tasks and data within the program, Legion can assist a programmer in solving the common programming burdens:

- Discovering/verifying the correctness of parallel execution: It is often difficult to determine when two tasks can run in parallel without a data race. Legion provides mechanisms to construct both implicit and explicit parallel task launches. For implicit constructs, Legion can automatically discover parallelism. For explicit constructs, Legion can notify the programmer if there are potential data races between tasks intended to run in parallel.
- Managing communication: When Legion determines that there are data dependencies between two tasks executed in different locations, Legion can automatically insert the

necessary data copies and apply the necessary constraints so the second task will not run until its data is available.

The Legion programming model is designed to abstract computations and makes them portable across many different potential architectures [2]. The challenge is to make it easy to map the abstracted computation of the program onto actual architectures. At a high level, mapping a Legion program needs making two kinds of decisions:

- For each task, select a processor on which to run the task.
- For each logical region, a task needs to select a memory in which to create and use a physical instance of the logical region.

To facilitate this process, Legion introduces a novel runtime mapping interface [1]. The mapping API can be used to specify on which processor each task will run. This allows the programmer to manage heterogeneity by explicitly running tasks on the best suitable processor. Furthermore, for each logical region required by a task, the mapping API allows the programmer to specify where the physical data for that logical region should be placed in the memory hierarchy. The runtime then handles all the copies necessary for moving data in the memory hierarchy. The mapping API is crucial to making Legion programs portable.

IV. Parallel Scientific Data Analytics Framework using Legion

When writing applications for distributed-memory parallel architectures, users must partition their data to enable parallel execution. As memory hierarchies become deeper, the latest supercomputers are now composed of heterogeneous processors and have multiple levels of memory, most of which are explicitly managed by software [26]. Thus, it is increasingly necessary for users to partition the data hierarchically. However, most current parallel programming languages perform this hierarchical partitioning statically, which excludes many important applications where the partitioning is data dependent and it must be computed dynamically. It is desired that users can be provided with a simplified data management, and do not need to handle the data duplication and partition problem. To this end, we present a parallel framework to illustrate how we can express computations with dynamically determined relationships between computations and data partitions based on the Legion programming language.

A. Mapper Interface

The goal of our parallel framework is to optimize computation performance by assigning operations to CPUs and GPUs heterogeneous processors and allowing them to work simultaneously. To achieve this goal, we design a custom mapper based on Legion's mapper interface to map operations onto target hardware and specify which memories are used to host the physical instances of the logical regions requested by such operations. Figure 1 shows the mapper

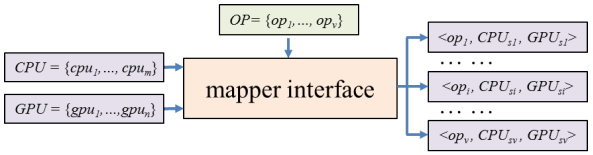


Figure 1. A custom mapper based on Legion's mapper interface in our framework to assign operations among CPUs and GPUs.

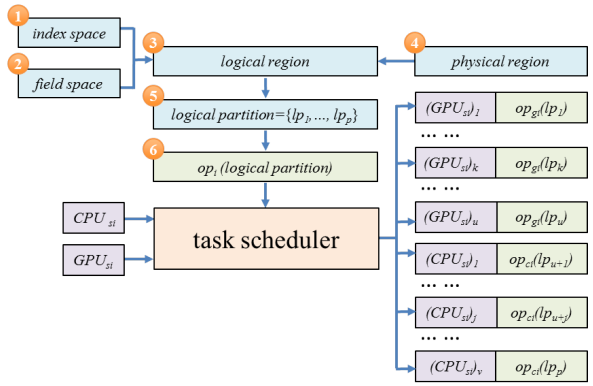


Figure 2. Data partition and subregion assignment among the processors.

interface in our framework where we assign a set of operations to different CPU and GPU subsets. We denote an operation set as $OP = \{op_1, ..., op_v\}$, a CPU set as $CPU = \{cpu_1, ..., cpu_m\}$, and a GPU set as $GPU = \{gpu_1, ..., gpu_n\}$, where $|OP| = v$, $|CPU| = m$ and $|GPU| = n$ denote the numbers of operations, CPUs, and GPUs respectively. We determine the assignment between processor types and operations using Legion's mapper interface. An operation op_i is assigned to CPU_{si} and GPU_{si} processed simultaneously, where $CPU_{si} \subseteq CPU$ and $GPU_{si} \subseteq GPU$ are the subsets of CPU and GPU respectively.

B. Region Construction and Task Scheduling

As we hope an operation op_i can be processed by CPUs and GPUs simultaneously, we divide it into two independent operations and assign them to a subset of CPUs and GPUs respectively: op_{ci} that is assigned to the CPU subset CPU_{si}, and op_{gi} that is assigned to the GPU subset GPU_{si}. After determining the assignment between an operation and a processor type, we need to partition the data associated with each operation in order to parallelize the operation on multiple instances of the assigned processor types.

For a general process of a parallel framework, we divide the input data into two portions: One portion is processed by op_{ci} on CPUs and another portion is processed by op_{gi} on GPUs. The partition ratio r between the data on CPUs and GPUs is assigned by users. We further divide each portion into a set of uniform blocks, and each processor is responsible for processing one block. We express one partition using one *logical region*. Conceptually, a logical

region can be expressed as a 2D table. The row entries in the logical region are defined as *index space*, which can be considered as the key of the logical region. The column entries in the logical region are defined as *field space*. Each field is defined by a pair of values: a unique name for the field (usually an integer) and the type of the field. Figure 2 shows the main steps of the process of our framework to make an operation op_i processed on heterogeneous processors:

1) We construct an index space of the logical region for the input data of each operation.
2) We construct a field space for the logical region. We allocate the field space for each portion of data.
3) We create a logical region using the index space and the field space defined in the previous two steps.
4) We create a corresponding physical region to hold the physical instances (i.e., the real values for the input data).
5) We use *coloring* to partition a logical region. We denote the partitions of a logical region as $logical\ partition = \{lp_1, ..., lp_p\}$, where $|logical\ partition| = p$ is the number of the logical partitions. Colorings are objects that describe an intended partition of an index space. Technically, a coloring is a map from colors to sets of points in an index space. For structured index spaces, colorings are maps from colors to Cartesian groups of points.
6) We execute operations on GPUs and CPUs according to the previous mapper interface we designed. We provide parallel CPU and GPU codes to make our program portable to different architectures. First, we need to define an enumeration for storing the IDs that we will direct the Legion runtime to associate with each operation. Second, we register operations with the Legion runtime.

Listing 1 shows the code to register tasks on CPUs and GPUs. In Line 6, the processor kind decides the operation run on latency-optimized cores (i.e., CPUs) or throughput-optimized cores (i.e., GPUs), the boolean single means that the operation can be performed as an individual operation, and the index allows the operation be preformed as an index space operation. We register op_{gi} on GPUs and register op_{ci} on CPUs (Lines 8 and 9). We import the `TaskHelper` namespace into the current program helping operation registration and dispatch using templates. The `register_cpu_variants` and `register_gpu_variants` are used to register either the CPU-only or GPU-only variants of operations respectively. The `dispatch_task` function is used to launch tasks. Legion also provides `FutureMap` types as a mechanism for managing the many return values that are returned from an index space task launch. `FutureMap` objects store a future value for every point in the index space task launch. Since op_{gi} and op_{ci} are independent with each other, we can dispatch them simultaneously, and obtain the two return values fm_{gi} and fm_{ci} (Lines 15 and 16). We use `fm_gi.wait_all_results()` and `fm_gi.wait_all_results()` to wait for all given operations to complete.

Listing 1
REGISTERING TASKS IN THE MAIN FUNCTION

```
1  int main(int argc,char* argv[]){
2    HighLevelRuntime::set_top_level_task_id
3                              (TOP_LEVEL_TASK_ID);
4    HighLevelRuntime::register_legion_task<top_level_task>
5          (TOP_LEVEL_TASK_ID,
6          Processor::LOC_PROC, true/*single*/, false/*index*/,
7          AUTO_GENERATE_ID, TaskConfigOptions(),
                 "top_level");
8    TaskHelper::register_gpu_variants<op_gi>();
9    TaskHelper::register_cpu_variants<op_ci>();
10 }
11 //TaskHelper namespace
12 namespace TaskHelper {
13   template<typename T>
14   FutureMap dispatch_task(T &launcher, Context ctx,
           HighLevelRuntime *runtime){
15     FutureMap fm = runtime->execute_index_space(ctx,
           launcher);
16     return fm;
17   }
18   //CPU implementation of the operation
19   template<typename T>
20   void base_cpu_wrapper(const Task *task,
21                         const std::vector<PhysicalRegion>
                                &regions,
22                         Context ctx, HighLevelRuntime
                                *runtime){
23     T::cpu_base_impl(task, task->local_args, regions, ctx,
           runtime);
24   }
25   //GPU implementation of the operation
26   template<typename T>
27   void base_gpu_wrapper(const Task *task,
28                         const std::vector<PhysicalRegion>
                                &regions,
29     T::gpu_base_impl(task, task->local_args, regions, ctx,
           runtime);
30   }
31   //register tasks on CPUs
32   template<typename T>
33   void register_cpu_variants(void){
34     HighLevelRuntime::register_legion_task<base_cpu_wrapper<T>
           >(T::TASK_ID, Processor::LOC_PROC,
           false/*single*/, true/*index*/, CPU_LEAF_VARIANT,
           TaskConfigOptions(T::CPU_BASE_LEAF),
           T::TASK_NAME);
35   }
36   //register tasks on GPUs
37   template<typename T>
38   void register_gpu_variants(void){
39     HighLevelRuntime::register_legion_task<base_gpu_wrapper<T>
           >(T::TASK_ID, Processor::TOC_PROC,
           false/*single*/, true/*index*/, GPU_LEAF_VARIANT,
           TaskConfigOptions(T::GPU_BASE_LEAF),
           T::TASK_NAME);
40   }
41 };
```

V. EXAMPLES

Our framework allows us to easily configure and execute multiple operations simultaneously on CPUs and GPUs. We show the detailed design by examples of a few but representative analytics operations, including entropy analysis and parallel volume rendering, on scientific volume data.

Entropy has been used to measure the information content of a variable [27]. Given a discrete random variable X and a probability mass function $p(x)$, $x \in X$, the entropy of X can be obtained as:

$$H(X) = -\sum_{x \in X} p(x) \log p(x), \tag{1}$$

where $p(x) \in [0,1]$, $-\sum_{x \in X} p(x) = 1.0$, and $-\log p(x)$ denotes the information associated with a single occurrence of x. For a scientific volume dataset, we partition it into a set of blocks, and compute entropy for each block to quantify the distribution of its variables. A data block with a higher value of $H(X)$ has more information.

Parallel volume rendering offers a viable solution to the large data visualization problem by distributing both data and rendering computations among multiple processing units. Parallel rendering algorithms, regardless of hardware architecture, consist of three categories first proposed by Molnar [28], named sort-first, sort-middle and sort-last, depending on how the volume data is sorted from object space to screen space.

In sort-first and sort-middle algorithms, each processor is assigned a sub-image space and is responsible for rendering partial volume data that lies in its assigned image space. During viewpoint changes, either some of the volume data must be transferred among processors or the data must be replicated on all processors. By comparison, in sort-last algorithms, each processor only needs to hold a fraction of the volume data that never needs to be transferred between processors, thereby avoiding communication during local rendering. However, an image compositing process is needed to combine all local partial images into a final image, which requires inter-processor communication. Stompel et al. [29] surveyed the methods for sort-last compositing and Cavin et al. [30] analyzed the relative theoretical performance of these methods. These overviews show that compositing algorithms usually fall into one of two categories: the direct-send based methods [31], [32] and the tree based methods [33].

With increasingly complex datasets, domain scientists often need to conduct detailed data analysis while perceiving volume rendering. In our examples, we add entropy analysis to show the process of assigning multiple operations using our framework.

A. Sort-last Parallel Volume Rendering with Entropy Analysis

In the sort-last parallel volume rendering algorithm, we first partition a volume data among compute nodes. Each compute node renders its local volume data using the ray casting method. Then, parallel image compositing is conducted to blend all partial images into a final image. We also perform entropy analysis using a block-based partitioning.

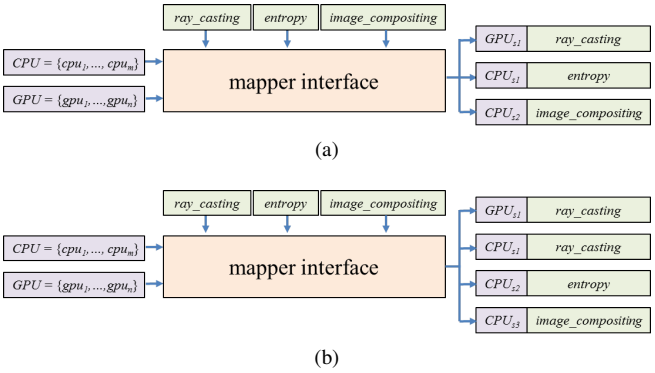


Figure 3. Two different assignments of ray casting, entropy analysis, and image compositing among CPUs and GPUs.

1) Mapper Interface: Local ray casting and parallel image compositing are the two main operations of the sort-last algorithm. Entropy computation is another operation for data analysis that can be independent as sort-last rendering. Given a set of CPUs and GPUs, we determine the assignment between processor types and operations using Legion's mapper interface. Figure 3 shows two examples in that we can assign the ray casting operation either to the GPUs or to the GPUs and a part of CPUs, and assign the entropy operation and the image compositing operation to the CPUs, given the relatively higher cost of ray casting.

The primary purpose of the Legion mapper is to decide how operations are assigned to processors and which memories are used to host the physical instances of the logical regions requested by the operations. We first define a mapper interface `RayMapper` derived from the default mapper of Legion. We override the `map_task` method of the mapper to create a ranking of memories in which we attempt to place physical instances for each region requirement of an operation, and provide the mapper the flexibility to specify which data should be placed close to the processors and which data can be left further away. To aid the mapper in decision making, the Legion runtime also provides the information for each region requirement about the available physical instances as well as which fields are already valid for these physical instances.

In our sort-last volume rendering, the ray casting operation is performed on GPUs and requires the memories for the input volume data and the output image. We select the GPU framebuffer memory for the volume data that is close to GPU, and select the *zero-copy memory* for the image data that will be accessed by both GPUs and CPUs. The zero-copy memory in Legion is the memory mapped to both GPU's address space and CPU's system memory (i.e., DRAM), and thereby is accessible by both GPUs and CPUs on the same node. The image compositing operation is performed on CPUs, and we select DRAM for the image data. We also select DRAM for the entropy operation.

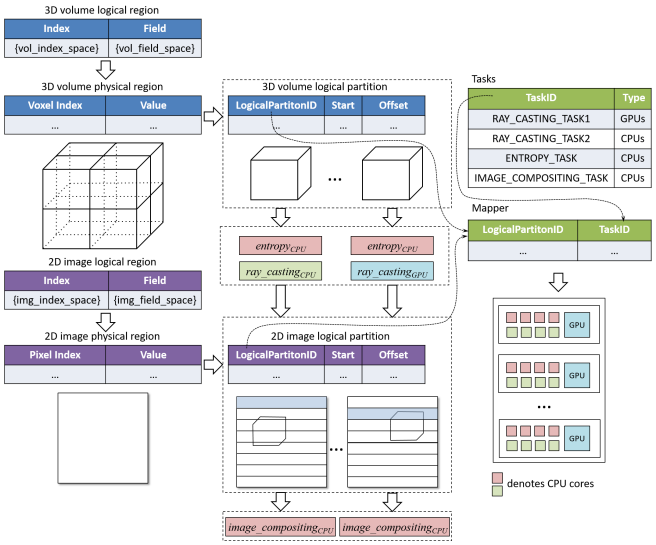


Figure 4. A sort-last parallel rendering process using legion.

2) Region Construction and Task Scheduling: After determining the assignment between an operation and a processor type, we need to partition the data associated with the operation in order to parallelize the operation on multiple instances of the assigned processor type.

We divide the 3D volume into a set of uniform 3D blocks, and each processor is responsible for rendering one block. In addition, we do not divide the 2D image. We express the volume partition using one logical region. We illustrate the whole process based on Legion in Figure 4.

In the first step, as shown in Figure 4, we construct the index space of the logical region for the volume and the 2D image. We create a 3D volume from (0, 0, 0) to (`vol_Width-1`, `vol_Height-1`, `vol_Depth-1`) for the volume and create a 2D rectangle from (0, 0) to (`img_Width-1`, `img_Height-1`) for the 2D image, where `vol_Width`, `vol_Height`, and `vol_Depth` denote the size of the 3D volume while `img_Width` and `img_Height` represent the size of a 2D image. Then, we create the index space `vol_index_space` for the 3D volume and `img_index_space` for the 2D image from the Legion runtime and context. We note that we use the Legion *structured index space* because it is more suitable for dense keys such as Cartesian grids. Legion also provides *unstructured index space* for more generic partitioning.

In the second step, we construct the field space of the logical region. In volume rendering, the data type of each image pixel is a 4-tuple of float data (i.e., an RGBA color), while the data type of the 3D volume data is float. Here, we allocate the field space `img_filed_space` for the partial images and one final image, as well as the field space `vol_field_space` for a volume data.

In the third step, we construct the logical region using the index space and the field space defined in the previous two steps. Each volume (image) logical partition consists of an index `LogicalPartitionID`, the start position `Start`, and the offset `Offset` within the whole volume (image).

In the fourth step, we create two physical regions to hold the physical instances (i.e., the real values in the volume data and 2D image). Specifically, we first create two requests for physical regions of the 2D image and the 3D volume with `READ_WRITE` privileges and `EXCLUSIVE` coherence, and then we add the fields created in the second step.

In the fifth step, we create the partitions in the logical regions using *coloring*. We divide a 3D volume into a set of uniform 3D blocks, and use a `DomainColoring` object to record our coloring. The return value is an `IndexPartition` object that is the handle to the partitioned volume index space. We also generate an array of subregions of the volume data. For each subregion, the `get_logical_subregion_by_color` function of Legion helps us associate a color space domain with each index sub-space we wish to make.

In the sixth step, we provide both parallel GPU and CPU rendering code to make our program portable to different architectures. In this example, we register our ray casting operation on GPUs or GPUs and CPUs, and register the image compositing and the entropy analysis only on CPUs, as shown in the Tasks table in Figure 4. Then, we execute the operations on the processing units according to the mapper interface we designed. As shown in Figure 4, the mapper assigns each logical partition to the corresponding tasks.

Our design pattern in Legion is to employ C++ classes to encapsulate the Legion operations. Instances of the class will describe launcher objects for launching operations, while static members functions will be used to give the many variant implementations of the operations.

3) Region Construction and Task Scheduling for Image Compositing: The final step in sort-last parallel rendering is to blend all partial images into a final image. In our design, we partition the 2D image index space into uniform 2D grids, and each CPU node is responsible for computing the blended color of an image partition separately and efficiently by direct-send parallel image compositing method, and writing the results to the final image's physical region. The underneath communication in image compositing traditionally is a challenging task to be tackled [29], which is handled by Legion in our solution.

B. Sort-first Parallel Volume Rendering

Recall from the sort-first algorithm, we divide the 2D image into uniform 2D grids, and each processor is responsible for the rendering of an image partition. In addition, we do not divide the 3D volume data (i.e., only one partition), and assume that it can be accessed by each processor via shared memory. Although the data partitioning and distribution requirements are significantly different between the sort-last

and sort-first algorithms, we can also easily implement the sort-first algorithm in our framework.

1) Mapper Interface: In sort-first parallel rendering, we do not need image compositing, and ray casting is the only one task. We assign the ray casting task to GPUs using Legion's mapper interface.

2) Region Construction and Task Scheduling: In the first step, we construct the same index space as in the first step of sort-last parallel rendering. However, in the second step, the constructions of the field spaces regions for the image and the volume are much easier than the sort-last algorithm. This is because we need only one field space for a 2D image and another field space for a 3D volume, partition this 2D image into even grids, and distribute each grid on a node to execute volume rendering in sort-first parallel rendering. In the third step, we use the same method in the third step of sort-last parallel rendering to construct the logical regions for a 2D image and a 3D volume. After we defined logical regions, we create physical regions for the 2D image and the 3D volume in the fourth step, which is similar to sort-last parallel rendering.

In the fifth step, we create the partitions in the 2D image logical region using coloring. We divide a 2D image into a set of uniform 2D regions. As the same method used in the sort-last algorithm, we use a `DomainColoring` object to record our coloring. The return value is two `IndexPartition` objects that are the handles to the image and volume index spaces. For each subregion `get_logical_subregion_by_color` function helps us associate a color space domain with each index subspace in the 2D image.

The last step is to execute ray casting sort-first parallel rendering on GPUs according to the setting in the mapper interface. We use almost the same main function of sort-last parallel rendering to register tasks, and register another image producing task.

C. Discussion

The sort-first and sort-last algorithms have considerable differences on data partitioning and distribution requirements [28], and their implementations are often diverse in existing work. Our framework can implement them in a similar manner. In addition, it is easy to add other operations (e.g., entropy analysis) with completely different data requirements in our framework. This shows that our solution provides a simple and feasible way to incorporate different operations in a unified framework using logical regions.

VI. Experiments and Results

We present our experimental results on Titan, a Cray XK7 supercomputer located at the Oak Ridge Leadership Computing Facility. Each node of Titan contains one 16-core AMD Opteron CPU and a NVIDIA Tesla K20 GPU.

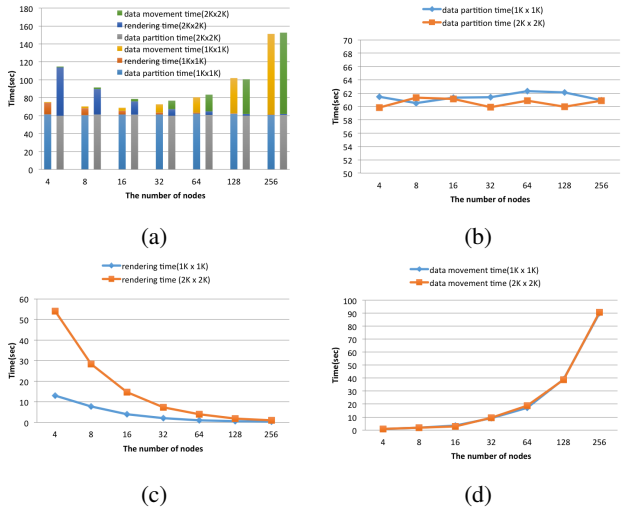


Figure 5. (a): the time breakdown of sort-first parallel volume rendering for a different number of nodes. (b): the data partition time. (c): the rendering time. (d): the data movement time. The output image resolutions are 1024^2 and 2048^2, respectively.

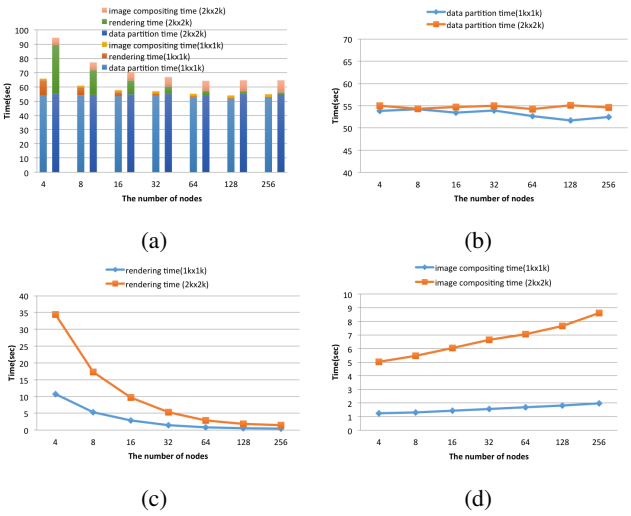


Figure 6. (a): the time breakdown of sort-last parallel volume rendering for a different number of nodes. (b): the data partition time. (c): the rendering time. (d): the image compositing time. The output image resolutions are 1024^2 and 2048^2, respectively.

We first tested our sort-first and sort-last parallel rendering implementations, and conducted scalability comparisons using a combustion dataset with the resolution of $1600\times1372\times430$. We tested between 1 to 256 processors with two output image resolutions of 1024^2 and 2048^2. We compared the algorithms in the worst case, where all pixel data are considered for rendering and compositing and where we did not implement any optimization techniques.

Figures 5 and 6 show the overview time breakdown, data partition time, rendering time, and data movement time on a different total number of nodes for sort-first rendering and

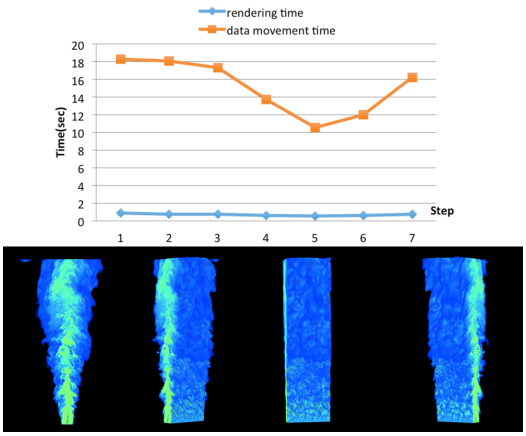
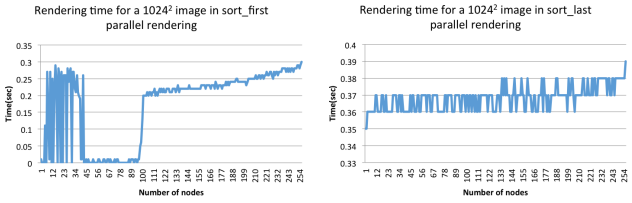


Figure 7. The rendering time and data movement time of sort-first rendering for 64 nodes from multiple view angles. The output image resolution is 1024^2.



(a) (b)

Figure 8. The time results of sort-first (a) and sort-last (b) parallel rendering on any number of nodes from 1 to 256. The output image resolution is 1024^2.

sort-last rendering, respectively. In sort-first rendering, the rendering times are reduced as the increase in the number of nodes. This is because the more number of nodes sharing the work, the smaller number of rays would be cast into the volume, which contributes to less workload on each node. However, data movement times are relatively high as the increase in the number of nodes. By comparison, in sort-last volume rendering, the rendering times are also dramatically decreased with the increase of the number of nodes, but we do not need to pay the cost of data movement. We used the parallel direct send method to realize image compositing task and obtained ideal image compositing time here. We achieved the scalable rendering performance for both algorithms.

From Figure 5 (b) and Figure 6 (b), we found that data partition in our experiment is relatively slow. There are two reasons leading to this cost. First, creating index space and field space and the data partition are processed in the top level task which only processed on one CPU. Second, Legion has not provided parallel file read functions yet, but treated them as just another kind of physical instance in a "disk" memory. Only "attach" and "detach" operations for files are provided for HDF5 files thus far. However, we note

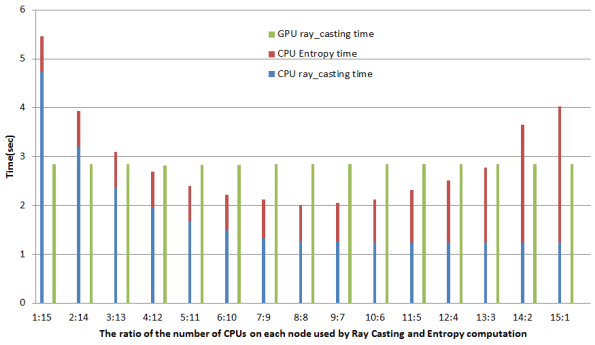


Figure 9. The time results of ray casting and entropy analysis with various ratios on allocation. The output image resolution is 1024^2.

that data partition is only a one-time cost for interactive rendering.

Figure 7 shows interactive rendering time and data movement time of sort-first parallel rendering for 64 nodes. The output image resolution is 1024^2. There are total seven interaction steps (i.e., seven view angles), and in Figure 7 we only attached four output images derived from the steps 1, 3, 5, and 7, respectively. As shown in Figure 7, the rendering time is relatively stable with the range from 0.56 to 0.9 seconds. The data movement time during the first five steps is decreased and increased in the last two steps. This is because we changed larger view angles during the final two steps than the previous ones.

Figure 8 shows the rendering time results of sort-first and sort-last parallel rendering on any number of nodes from 1 to 256. The output image resolution is 1024^2. As shown in the image (a), the rendering time is imbalanced, and a few of nodes are idle. The reason is that we did not use any optimized image partition method here, and only distribute one image brick to each node to process, so a few of nodes render the occluded area in the 2D image. By comparison, as shown in the image (b), the rendering times of sort-last on each node are nearly balanced, because we evenly partition the workload into small volume cubes and distribute each one to one node.

In general, sort-last rendering can achieve more scalable performance, and sort-first rendering inherently requires data movement [28]. Without conducting any special optimization and tackling physical data management, our framework can easily implement these two renderings and achieve the expected performance results based on logical regions.

As discussed in Section V-A and shown in Figure 3, we can assign different operations (e.g., ray casting, image compositing, and entropy analysis) among different CPUs and GPUs. In this way, we can maximize the usage of all computing units. For the tasks assigned to the same type of processor, Legion employs the work stealing strategy to

balance workload among the processors [2]. In our test with 16 computing nodes and sort-last rendering, we assigned entropy analysis to CPUs, and assigned 5% of ray casting workload to CPUs and 95% to GPUs. With Legion's default work stealing scheduling, the CPU ray casting time is 1.347 seconds, the CPU entropy time is 0.936 seconds, and the GPU ray casting time is 2.833 seconds. The principle is to partition the data into small blocks and the number of partition is much larger than the current processors, so that the idle processors can steal the jobs. Here, we partition the data for the CPU ray casting into 1280 blocks, and the data for entropy analysis into 1280 blocks. In this case, the image compositing time and the data access time will be increased, but the overhead is negligible.

Alternatively, we can manually assign the ratio of the number of CPU cores on each node for ray casting and entropy analysis. Given that each node has a 16-core CPU, we tested different ratios as shown in Figure 9. We can see that when both ray casting and entropy analysis use 8 cores, we gained the lowest time results that are close to the ones of the default work-stealing scheduling. This shows that we can obtain optimal performance for multiple operations on heterogeneous processors using the built-in scheduler without exhaust tuning, which is particularly useful for dynamic operations.

VII. Conclusions and Future Work

We present a study for conducting scientific data analytics on distributed heterogeneous architectures by leveraging the Legion programming model and runtime system. We consider both scalability and useability in our design, and show that our framework can facilitate the implementation and execution of complex analytics operations with completely different data partitioning and distribution requirements in a nearly unified manner. Furthermore, our framework can perform these operations across CPUs and GPUs and balance workload by automatic or manual scheduling strategies. With our framework, users can focus on scientific applications, rather than the cumbersome data management on massive heterogeneous processors. In the future, we will build pipelines among different operations. For example, we may use entropy analysis to capture data regions with more information and guide other analytics. We will also integrate our analytics framework with scientific simulations, and explore the feasibility to enhance end-to-end scientific discovery workflows.

Acknowledgment

This research has been sponsored in part by the Department of Energy through the ExaCT Center for Exascale Simulation of Combustion in Turbulence, and by the National Science Foundation through grant IIS-1423487. The allocation of supercomputing time on the Oak Ridge Leadership Computing Facility (OLCF) has been sponsored by the Department of Energy through the Innovative and Novel Computational Impact on Theory and Experiment (INCITE) program.

References

[1] M. E. Bauer, "Legion: Programming distributed heterogeneous architectures with logical regions," Ph.D. dissertation, Stanford University, 2014.

[2] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: expressing locality and independence with logical regions," in *Proceedings of the international conference on high performance computing, networking, storage and analysis*. IEEE Computer Society Press, 2012, p. 66.

[3] S. Treichler, M. Bauer, and A. Aiken, "Language support for dynamic, hierarchical data partitioning," *ACM SIGPLAN Notices*, vol. 48, no. 10, pp. 495–514, 2013.

[4] K. Hwang, A. Ramachandran, and R. Purushothaman, *Advanced computer architecture: parallelism, scalability, programmability*. McGraw-Hill New York, 1993, vol. 199.

[5] L. Dagum and R. Enon, "OpenMP: An industry standard API for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[6] R. Duncan, "A survey of parallel computer architectures," *Computer*, vol. 23, no. 2, pp. 5–16, 1990.

[7] B. Nitzberg and V. Lo, "Distributed shared memory: A survey of issues and algorithms," *Distributed Shared Memory-Concepts and Systems*, pp. 42–50, 1991.

[8] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar, "X10: An object-oriented approach to non-uniform cluster computing," *Acm Sigplan Notices*, vol. 40, no. 10, pp. 519–538, 2005.

[9] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel programmability and the chapel language," *International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.

[10] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele Jr, S. Tobin-Hochstadt, J. Dias, C. Eastlund *et al.*, "The fortress language specification," *Sun Microsystems*, vol. 139, p. 140, 2005.

[11] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally *et al.*, "Sequoia: Programming the memory hierarchy," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM, 2006, p. 83.

[12] S. Chandra, V. Saraswat, V. Sarkar, and R. Bodik, "Type inference for locality analysis of distributed data structures," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. ACM, 2008, pp. 11–22.

[13] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 519–530, 2013.

[14] K.-L. Ma, "Parallel volume ray-casting for unstructured-grid data on distributed-memory architectures," in *Proceedings of the IEEE Symposium on Parallel Rendering*, ser. PRS '95, 1995, pp. 23–30.

[15] S. Parker, P. Shirley, Y. Livnat, C. Hansen, and P.-P. Sloan, "Interactive ray tracing for isosurface rendering," in *Proceedings of the Conference on Visualization '98*, ser. VIS '98, 1998, pp. 233–238.

[16] H. Childs, M. Duchaineau, and K.-L. Ma, "A scalable, hybrid scheme for volume rendering massive data sets," in *Proceedings of the 6th Eurographics Conference on Parallel Graphics and Visualization*, ser. EGPGV '06, 2006, pp. 153–161.

[17] C.-Y. Chan and Y. E. Ioannidis, "An efficient bitmap encoding scheme for selection queries," *SIGMOD Rec.*, vol. 28, no. 2, pp. 215–226, Jun. 1999.

[18] K. Stockinger, E. W. Bethel, S. Campbell, E. Dart, and K. Wu, "Detecting distributed scans using high-performance query-driven visualization," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, ser. SC '06, 2006.

[19] L. Gosink, J. Anderson, W. Bethel, and K. Joy, "Variable interactions in query-driven visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 6, pp. 1400–1407, Nov. 2007.

[20] J. Chou, M. Howison, B. Austin, K. Wu, J. Qiang, E. W. Bethel, A. Shoshani, O. Rübel, Prabhat, and R. D. Ryne, "Parallel index and query for large scale data analysis," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11, 2011, pp. 30:1–30:11.

[21] J. C. Bennett, H. Abbasi, P.-T. Bremer, R. Grout, A. Gyulassy, T. Jin, S. Klasky, H. Kolla, M. Parashar, V. Pascucci, P. Pebay, D. Thompson, H. Yu, F. Zhang, and J. Chen, "Combining in-situ and in-transit processing to enable extreme-scale scientific analysis," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12, 2012, pp. 49:1–49:9.

[22] Q. Sun, T. Jin, M. Romanus, H. Bui, F. Zhang, H. Yu, H. Kolla, S. Klasky, J. Chen, and M. Parashar, "Adaptive data placement for staging-based coupled scientific workflows," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15, 2015, pp. 65:1–65:12.

[23] D. Wu, L. Zhu, X. Xu, S. Sakr, D. Sun, and Q. Lu, "Building pipelines for heterogeneous execution environments for big data processing," *IEEE Software*, vol. 33, no. 2, pp. 60–67, Mar 2016.

[24] B. Pérez, J. L. Bosque, and R. Beivide, "Simplifying programming and load balancing of data parallel applications on heterogeneous systems," in *Proceedings of the 9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit*, ser. GPGPU '16, 2016, pp. 42–51.

[25] S. Breß, B. Köcher, M. Heimel, V. Markl, M. Saecker, and G. Saake, "Ocelot/HyPE: Optimized data processing on heterogeneous hardware," *Proc. VLDB Endow.*, vol. 7, no. 13, pp. 1609–1612, Aug. 2014.

[26] M. Bauer, J. Clark, E. Schkufza, and A. Aiken, "Programming the memory hierarchy revisited: Supporting irregular parallelism in sequoia," *ACM SIGPLAN Notices*, vol. 46, no. 8, pp. 13–24, 2011.

[27] C. Wang and H.-W. Shen, "Information theory in scientific visualization," *Entropy*, vol. 13, no. 1, pp. 254–273, 2011.

[28] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs, "A sorting classification of parallel rendering," *Computer Graphics and Applications, IEEE*, vol. 14, no. 4, pp. 23–32, 1994.

[29] A. Stompel, K.-L. Ma, E. B. Lum, J. Ahrens, and J. Patchett, "SLIC: Scheduled linear image compositing for parallel volume rendering," in *Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*. IEEE Computer Society, 2003, p. 6.

[30] X. Cavin, C. Mion, and A. Filbois, "Cots cluster-based sort-last rendering: Performance evaluation and pipelined implementation," in *Visualization, 2005. VIS 05. IEEE*. IEEE, 2005, pp. 111–118.

[31] W. M. Hsu, "Segmented ray casting for data parallel volume rendering," in *Proceedings of the 1993 symposium on Parallel rendering*. ACM, 1993, pp. 7–14.

[32] U. Neumann, "Communication costs for parallel volume-rendering algorithms," *Computer Graphics and Applications, IEEE*, vol. 14, no. 4, pp. 49–58, 1994.

[33] K.-L. Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh, "Parallel volume rendering using binary-swap compositing," *Computer Graphics and Applications, IEEE*, vol. 14, no. 4, pp. 59–68, 1994.