Legion-based Scientific Data Analytics on Heterogeneous Processors

Lina Yu, Hongfeng Yu

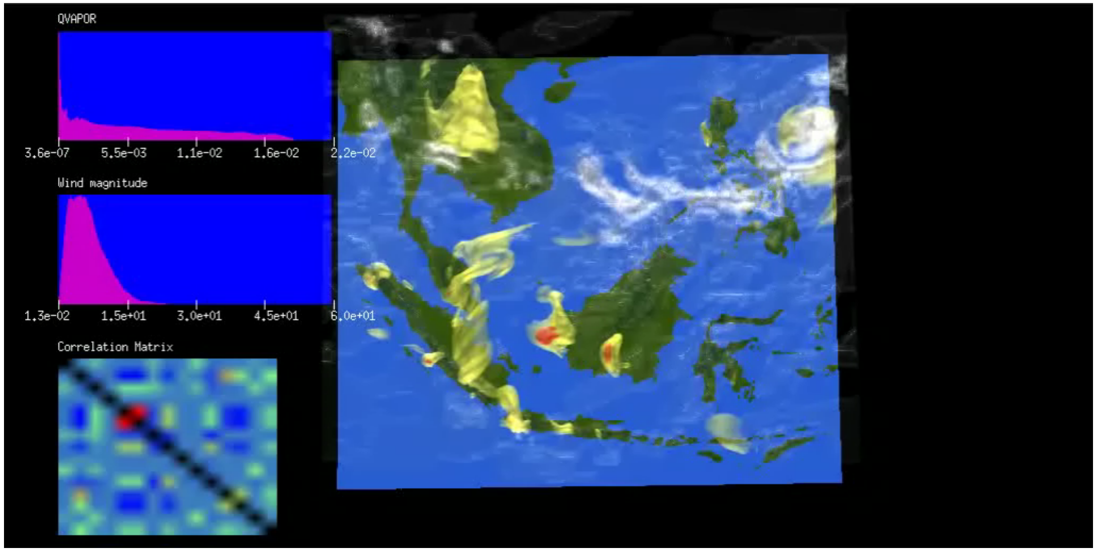
Department of Computer Science & Engineering

University of Nebraska Lincoln, Lincoln, Nebraska

Outline

- Motivation
- Contributions
- Framework
- Examples
- Experiments and Results
- Conclusion

Motivation



– A scientific analytics workflow consists of **multiple operations** that intrinsically incur different **communication** or **data movement** requirements between compute nodes

Motivation

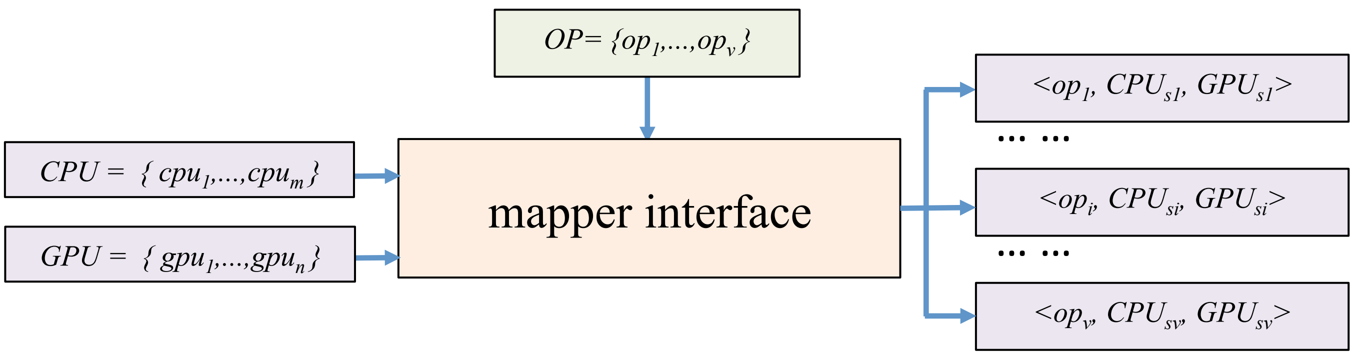
- Legion : programming model + runtime system
  - Describe hierarchical organizations of both data and computation at an abstract level
- Legion assists a programmer in solving the common programming burdens
  - Discover/verify the correctness of parallel execution
  - Manage communication
- At a high level, mapping a Legion program needs making two kinds of decisions
  - For each task, select a processor on which to run the task by the **mapping interface**
  - For each **logical region**, a task needs to select a memory in which to create and use a physical instance of the logical region

Our Contribution

- Investigate the feasibility of using **Legion** to perform analytics for **large-scale** scientific data on **heterogeneous** processors
- Help users **simplify** programming on the **data partition**, **data organization**, and **data movement** for distributed-memory heterogeneous architectures
- Facilitate a simultaneous execution of **multiple analytics operations** on modern and future supercomputers
- Demonstrate the **scalability** and the **usability** of our approach using several representative analytics operations on a heterogeneous supercomputer

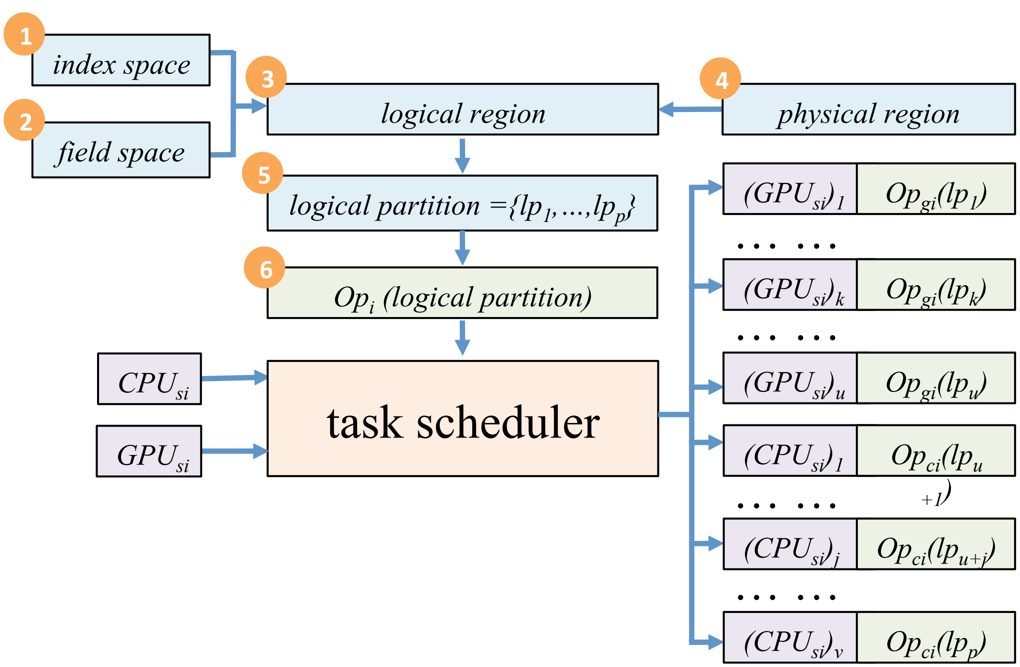
Mapper Interface

- We design a custom mapper based on Legion’s mapper interface
  - Map operations onto target processors
  - Specify which memories are used to host the physical instances of the logical regions requested by such operations



Region Construction and Task Scheduling

- Main steps of the process of our approach
  - Make an operation *opi* processed on heterogeneous processors



1. Construct an index space of the logical region for the input data of each operation.
2. Construct a field space of the logical region, and allocate the field space for each portion of data.
3. Create a logical region using the index space and the field space defined in the previous two steps.
4. Create a corresponding physical region to hold the physical instances (i.e., the real values for the input data).
5. Use coloring to partition a logical region (colorings are objects that describe an intended partition of an index space).
6. Execute operations on GPUs and CPUs according to the previous mapper interface we designed.

Listing 1
REGISTERING TASKS IN THE MAIN FUNCTION

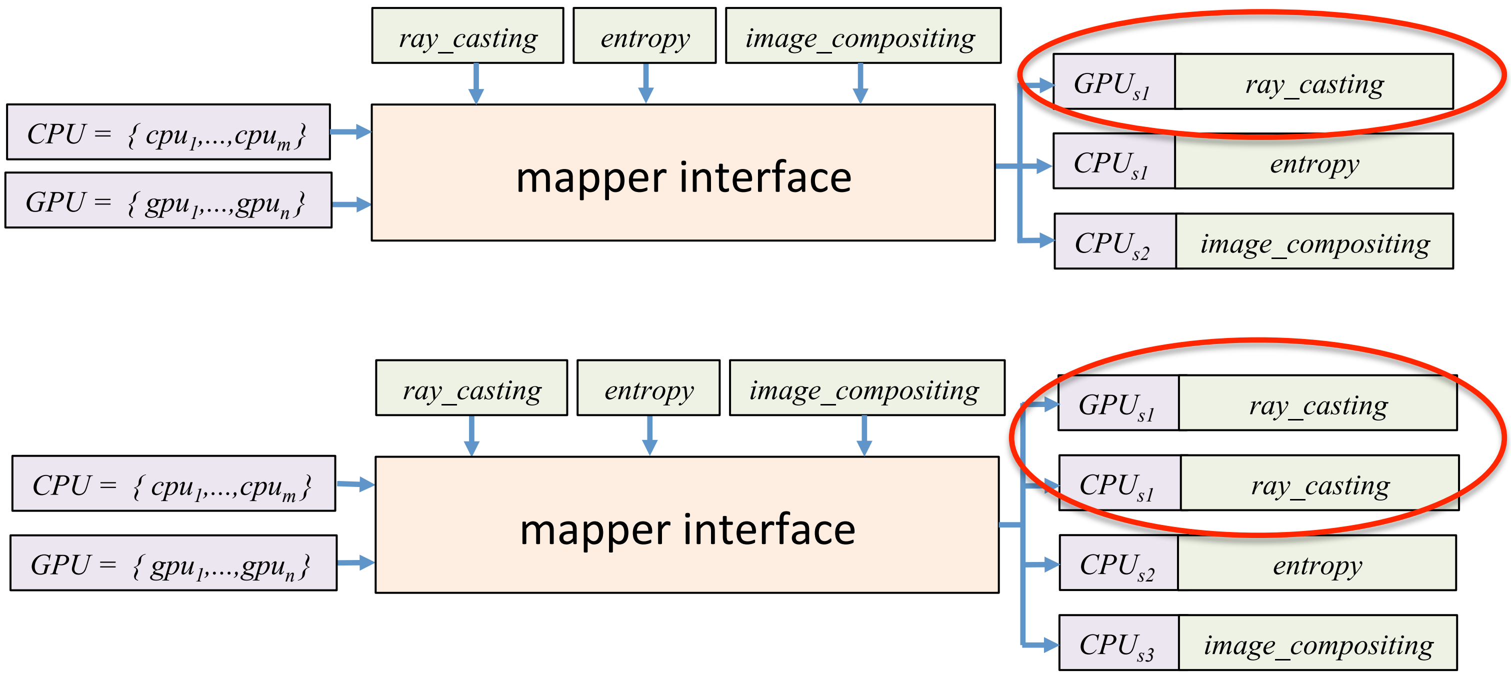
```
int main(int argc,char* argv[]){
  HighLevelRuntime::set_top_level_task_id
                        (TOP_LEVEL_TASK_ID);
  HighLevelRuntime::register_legion_task<top_level_task>
        (TOP_LEVEL_TASK_ID,
        Processor::LOC_PROC, true/*single*/, false/*index*/,
        AUTO_GENERATE_ID, TaskConfigOptions(),
              "top_level");
  TaskHelper::register_gpu_variants<op_gi>();
  TaskHelper::register_cpu_variants<op_ci>();
}
//TaskHelper namespace
namespace TaskHelper {
  template<typename T>
  FutureMap dispatch_task(T &launcher, Context ctx,
        HighLevelRuntime *runtime){
    FutureMap fm = runtime->execute_index_space(ctx,
          launcher);
    return fm;
  }
  //CPU implementation of the operation
  template<typename T>
  void base_cpu_wrapper(const Task *task,
                         const std::vector<PhysicalRegion>
                               &regions,
                         Context ctx, HighLevelRuntime
                               *runtime){
    T::cpu_base_impl(task, task->local_args, regions, ctx,
          runtime);
  }
```

Continued List 1

```
//GPU implementation of the operation
template<typename T>
void base_gpu_wrapper(const Task *task,
                       const std::vector<PhysicalRegion>
                              &regions,
  T::gpu_base_impl(task, task->local_args, regions, ctx,
         runtime);
}
//register tasks on CPUs
template<typename T>
void register_cpu_variants(void){
  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);
}
//register tasks on GPUs
template<typename T>
void register_gpu_variants(void){
  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);
}
};
```

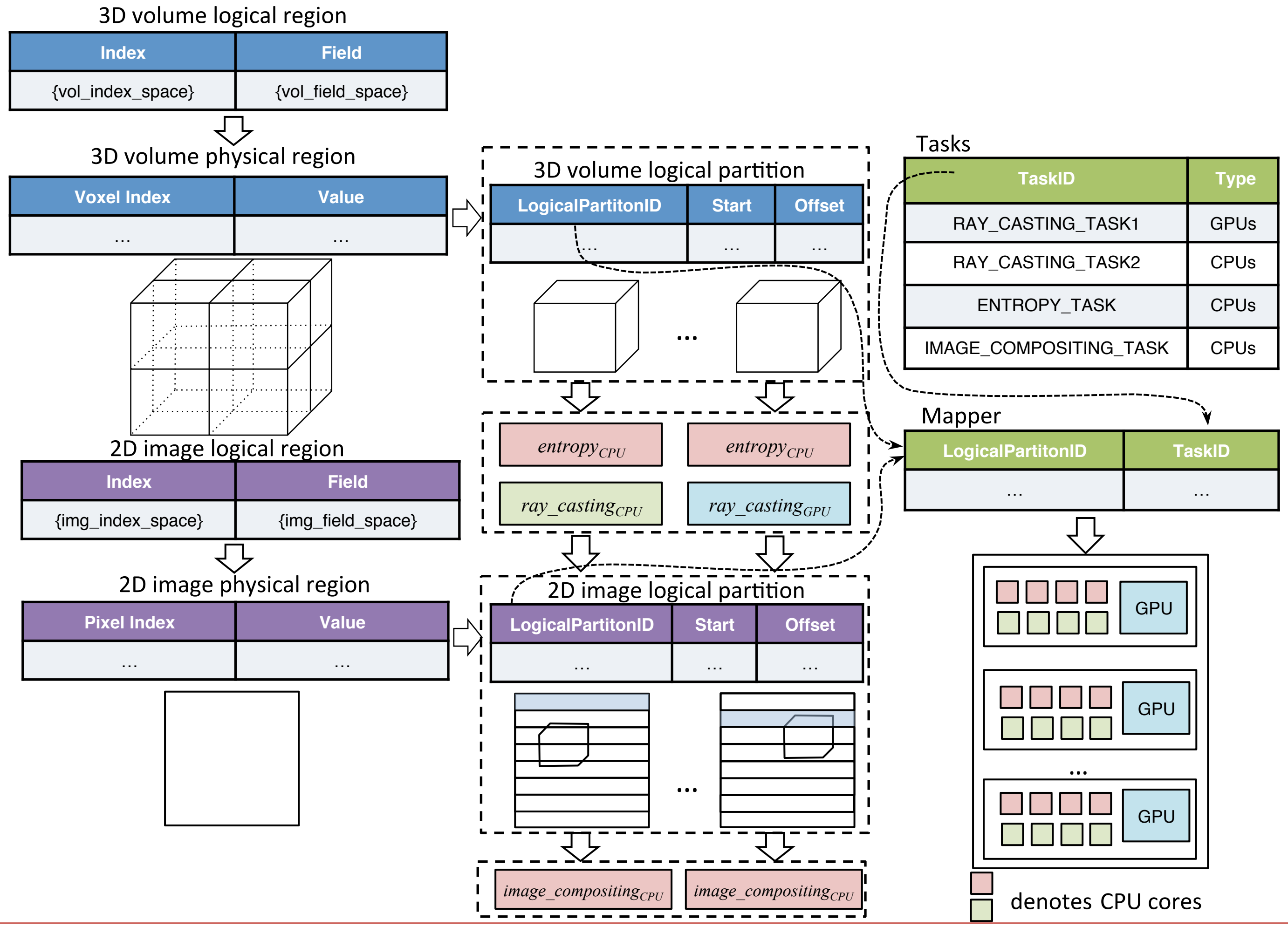
Examples

- **Sort-last** parallel volume rendering with entropy analysis
  - Mapper interface



Examples

- **Sort-last** parallel volume rendering with entropy analysis
  - Region construction and task scheduling



Examples

- **Sort-first** parallel volume rendering with entropy analysis
  - Mapper interface
    - Ray casting task(GPUs)
    - Entropy task(CPUs)
  - Region construction and task scheduling
    - **Divide the 2D image** into uniform 2D grids
    - Each processor is responsible for the rendering of an image portion
    - **No need to divide the 3D volume data**
    - **No need image compositing**

- The sort-first and sort-last algorithms have **differences on data partitioning and distribution requirements**, but our solution provides a **simple** and **feasible** way to incorporate different operations in a unified framework using logical regions

Experiments and Results

- Conduct experiments 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**
- Test sort-first and sort-last parallel rendering
- Conduct scalability comparisons using a combustion dataset with the resolution of 1600x1375x430
- Test between 1 to 256 processors with two output image resolutions of 1024^2 and 2048^2

Experiments and Results

- 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 sort-last rendering

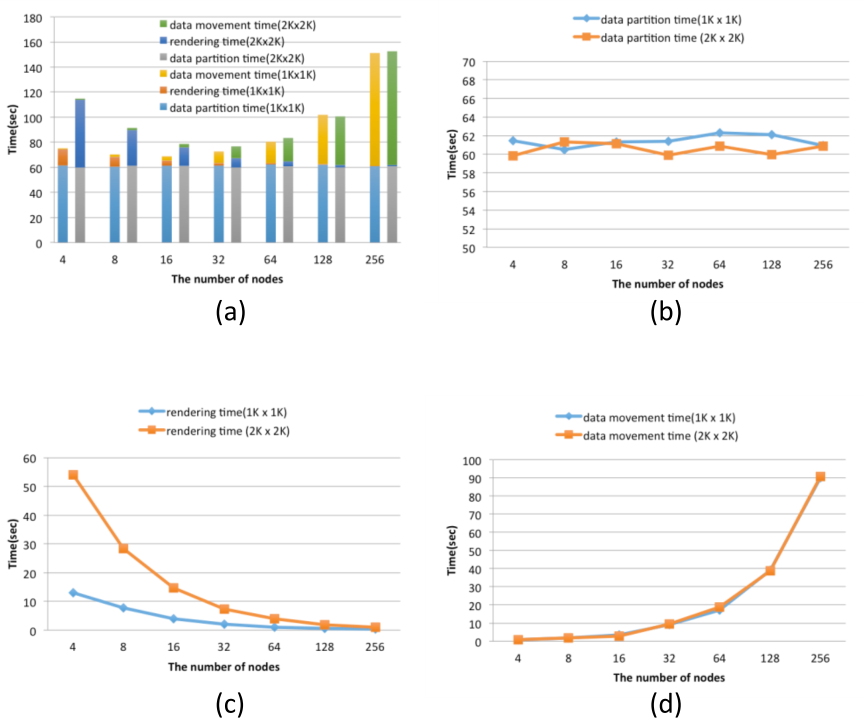


Fig. 1: (a): the time breakdown of sort-first parallel volume rendering for different number of nodes. (b): the data partition time. (c): the rendering time. (d): the data movement time. Two output image resolutions, 1024^2 and 2048^2, are used.

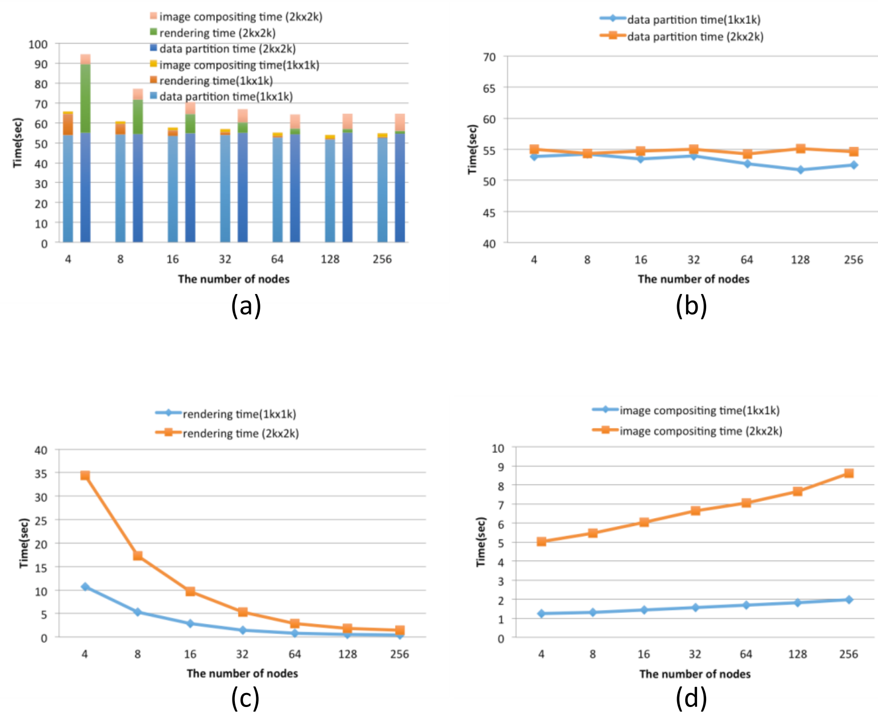


Fig. 2: (a): the time breakdown of sort-last parallel volume rendering for different number of nodes. (b): the data partition time. (c): the rendering time. (d): the image compositing time. Two output image resolutions, 1024^2 and 2048^2, are used.

Experiments and Results

- Interactive rendering time and data movement time of sort-first parallel rendering for 64 nodes with image resolution of 1024^2

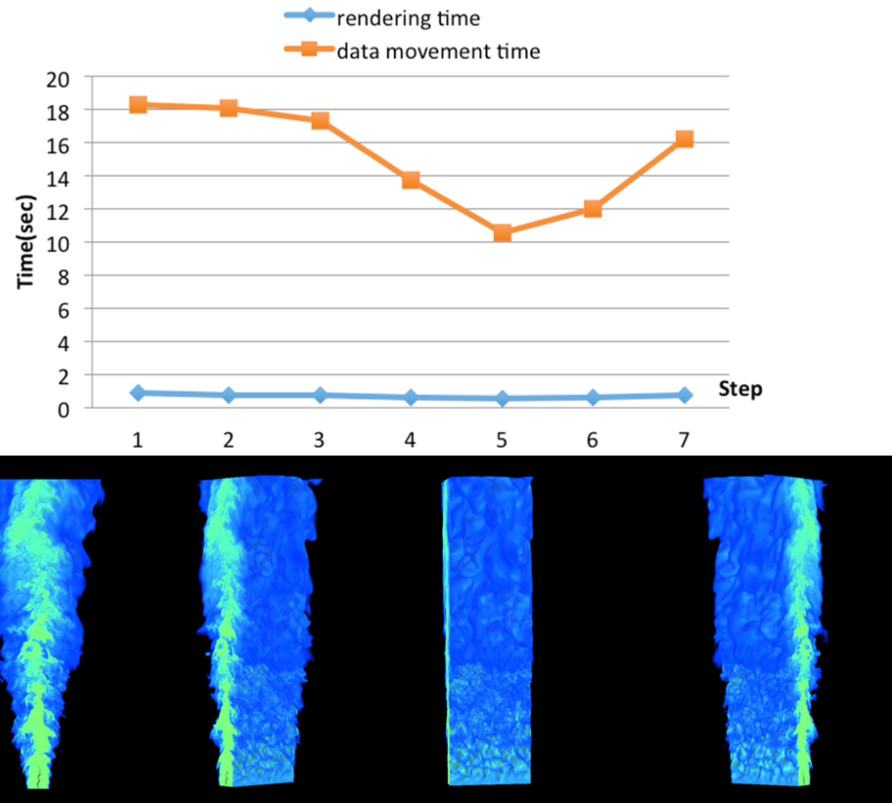


Fig. 3: The rendering time and data movement time of sort-first rendering for 64 nodes from multiple view angles. The output image resolution is 1024^2.

Experiments and Results

- The rendering time results of sort-first and sort-last parallel rendering on any number of nodes from 1 to 256 with image resolution of 1024^2

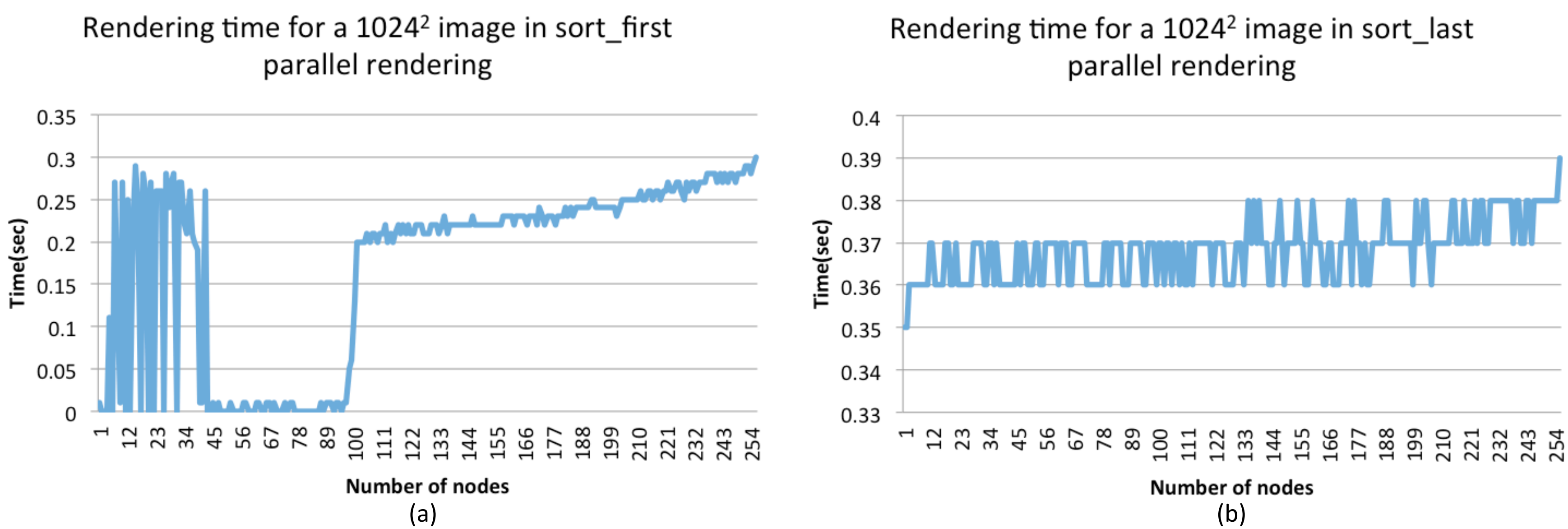


Fig. 4: The time results of sort-first (a) and sort-last (b) parallel rendering on any number of nodes from 1 to 256. The output image resolution is 1024^2.

Experiments and Results

- Legion job stealing scheduling performance
  - CPU ray casting time is 1.347 seconds(5%)
  - CPU entropy time is 0.936 second
  - GPU ray casting time is 2.833 seconds (95%)
- Given that each node has a 16-core CPU, we tested different ratios between ray casting and entropy operations

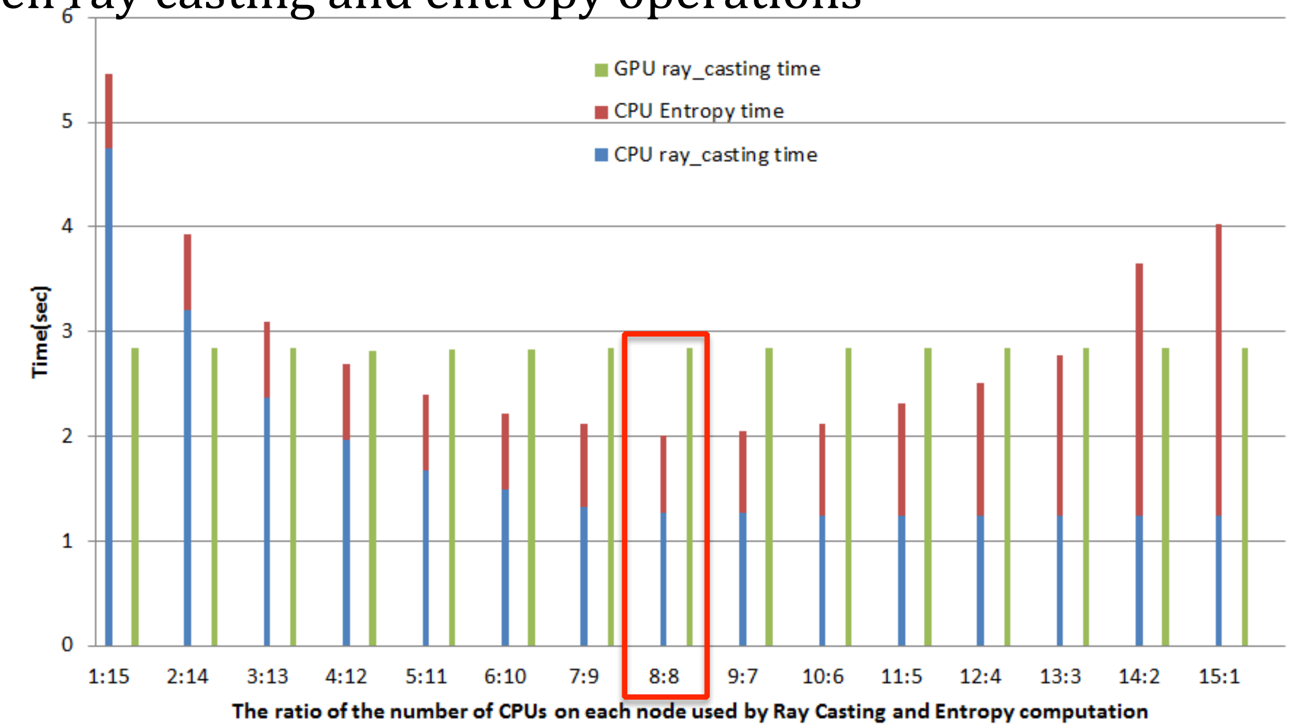


Fig. 5: The time results of ray casting and entropy analysis with various ratios on allocation. The output image resolution is 1024^2.

Conclusion

- A study for conducting scientific data analytics on **distributed heterogeneous** architectures by leveraging the **Legion** programming model and runtime system
- Consider both **scalability** and **usability** in our design
- Facilitate complex analytics operations with **completely different data partitioning** and **distribution** requirements in a nearly **unified** manner
- Perform operations across **CPUs** and **GPUs** and **balance** workload by automatic or manual scheduling strategies

Acknowledgement

- This research has been sponsored in part by the Department of Energy through the ExaCT Center for Exascale Simulation of Combustion in Turbulence 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

University of Nebraska Lincoln

Thank You!