Legion: Expressing Locality and Independence with Logical Regions

Michael Bauer, Sean Treichler, Alex Aiken
Stanford University

The Legion Programming System

Deep Memory Hierarchies
Memory hierarchies are becoming increasingly complex. Moving data through the memory hierarchy often requires understanding multiple APIs (MPI, GASNet, CUDA). Legion is designed for writing memory-hierarchy-agnostic programs.

Heterogeneous Architectures
Today's machines often have more than one type of processor (CPU,GPU) and future architectures will likely have specialized accelerators. Legion is designed to manage heterogeneity by allowing code to be retargeted to different kinds of processors.

Portability
Machine architectures are changing quickly. Porting code for every new architecture and API is unfeasible. Legion enables programs to be written once and then mapped to many different machine architectures via a novel mapping interface.

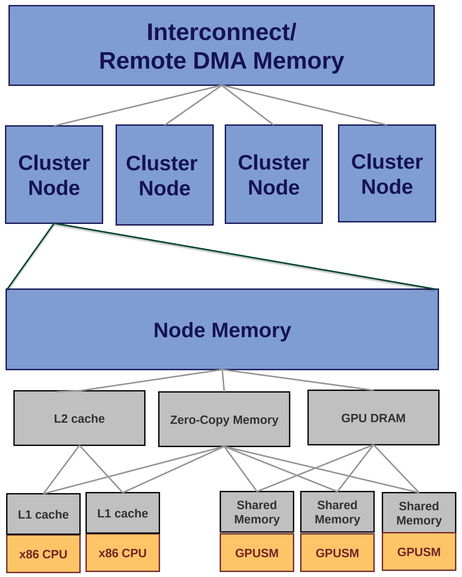

Figure 1. Example heterogeneous architecture with a deep memory hierarchy.

Programming with Logical Regions

Logical regions can be partitioned into logical sub-regions. Logical regions allow the programmer to express both locality in data structures as well as independence between tasks that use disjoint logical regions. Via the logical region abstraction, the Legion compiler and runtime can extract both parallelism and locality information to target many different architectures and memory hierarchies.

Programming with logical regions requires programmers to describe recursive decompositions of their data structures. To give an example, consider an arbitrary graph data structure. When performing graph computations on large clusters, programmers break up nodes into sets of nodes that are *owned* by a task and nodes that are *ghost* nodes (owned by another task, but must be accessed). Below we illustrate how we can describe this partitioning using logical regions. First the set of all nodes are partitioned into the nodes will be private to a task and the set of nodes that are shared between at least two tasks. Then nodes are partitioned into the sets of nodes that are private, shared, and ghost nodes for each individual task.

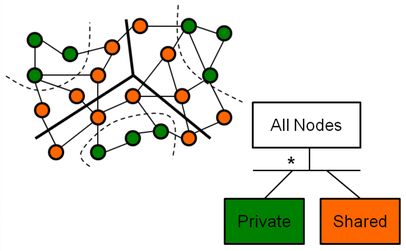

Figure 5. Decomposing an arbitrary graph into private and shared nodes

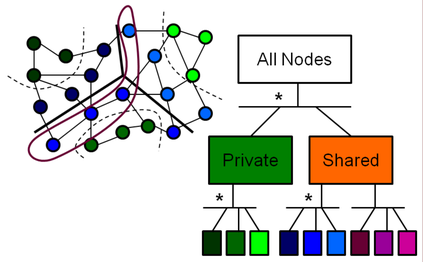

Figure 6. Further decomposing nodes into private, shared, and ghost for each task

Legion Runtime Systems

Legion programs are currently written in C++ and make calls into the Legion runtime library. The runtime system is broken down into a high-level and a low-level component.

High-Level Runtime
The Legion high-level runtime implements the Legion programming model. Applications call into the high-level runtime for creating, destroying, and partitioning logical regions as well as for launching tasks. The high-level runtime also makes queries of the mappers implemented by the programmer for determining where tasks should be placed and where data should be placed in the memory hierarchy.

Low-Level Runtime
To avoid having to re-implement the high-level runtime for each new architecture the high-level runtime sits on the low-level runtime. The low-level runtime abstracts the common low-level APIs (Pthreads, GASNet, MPI, CUDA) using an event-based model. The low-level runtime is designed to be easily extended as new APIs or architectures are released.

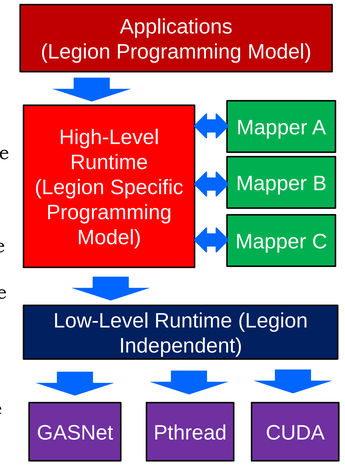

Figure 7. Legion Runtime Systems

The Legion Programming Model

The goal of the Legion programming model is to allow programmers to express the locality and independence properties of their programs in a machine agnostic manner. The primary abstraction for expressing these properties is a *logical region*. A logical region has the following properties:

- A collection of elements that will be referenced together (locality)
- No implied layout of data
- No implied location of in the memory hierarchy

Logical regions can be partitioned into sub-logical regions. This allows for recursive decomposition of data structures both statically and dynamically.

A Legion program is a tree of tasks where each task specifies the logical regions that the task will access. Additionally the region usages are annotated with the permissions and coherence properties for the task.

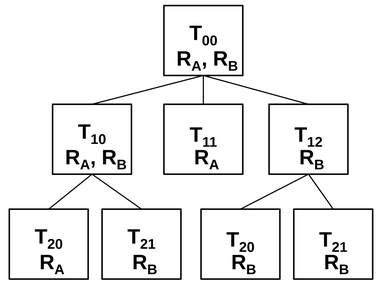

Figure 2. Legion tree of task with region usage annotations

Mutation Permissions	Allocation Permissions
Read-Only	Allocable
Read-Write	Freeable
Reduce <ReductionType>	

Figure 3. Region Permissions

Exclusive – Single task (program order)
Atomic – Single task (serializable order)
Simultaneous – Multiple tasks at the same time

Figure 4. Region Coherence Properties

Mapping Legion Programs to Deep Memory Hierarchies

To retarget Legion programs to many different machine architectures, we introduce a novel mapping interface. Mapping a Legion program onto a specific architecture requires answering two questions for each task:

1. On which (type of) processor will each task be run?
2. For each task, where in the memory hierarchy will the data for that task be placed?

In general, the answers to these questions are specific to each application and machine. Rather than requiring our compiler and runtime to be smart enough to figure out the answers, we instead provide a mapping API that a programmer implements for each application and architecture they wish to target. The runtime then queries the mapping interface every time a decision needs to be made.

Mapping API
The mapping API allows the programmer to specify on which processor each task will be run. This allows programmer to manage heterogeneity by explicitly running tasks on the processor best suited for them. Furthermore, for each logical region that a task requires, 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.

Portability
The mapping API is crucial to making Legion programs portable. To port an application to a new hardware, the programmer only has to re-implement the mapping API for that particular application on that hardware. This provides much better productivity than re-implementing an entire piece of software for a new architecture.

Performance Results

Our performance results are on a 4-Node cluster. Each node has 2 6-core CPUs and 2 Tesla C2070 GPUs.

Circuit Simulation
Our first experiment is a circuit simulation that models a graph of circuit elements (edges) which connect nodes of equal potential. To run this simulation on our cluster of GPUs, we partition the set of nodes and edges into logical regions corresponding to private, shared, and ghost components. We implement a custom mapper that places private nodes in GPU DRAM, while keeping shared and ghost nodes in the zero-copy memory making it easier to move data through GASNet RDMA operations.

Fluid Simulation
Our fluid simulation is a generalization of the fluidanimate benchmark from the PARSEC benchmark suite that is not constrained to only run on shared-memory machines. We partition the set of cells into owned and shared cells, with explicit logical regions for manual double buffering. A custom mapper keeps shared cell regions in GASNet backing memory for efficient RDMA operations.

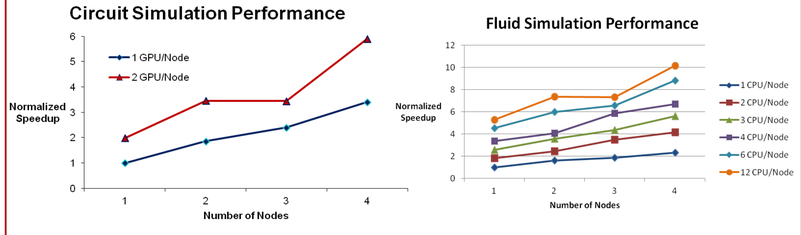

Figure 8. Legion Performance Results