Programming with Legion

Alex Aiken

June 21, 2015

Preface

The first paper describing the Legion programming model was published in 2012 [BTSA12]. Since then, there has been enormous progress on many fronts and many people have contributed to the project. Throughout this period new application developers have learned Legion through a combination of examples, lore from other members of the project, research papers and reading the source code of the Legion implementation. The intention here is to put down in a systematic fashion what a programmer who wants to use Legion to develop high performance applications needs to know.

This book is intended to be a combination tutorial, rationale and manual. The first part is the tutorial and rationale, laying out in some detail what Legion is and why it is that way. The second part is the manual, which describes each of the API calls for the Legion C++ runtime.

The example programs and configuration files referred to in this book can be found in the directory `LegionManual/Examples/` included in the Legion distribution.

This book is incomplete and will remain incomplete for some time to come. But on the theory that partial documentation is better than no documentation, the manual is being made available while it is still in progress in the hope that it will be useful to new Legion programmers. Please report any errors or other issues to `aiken@cs.stanford.edu`.

Alex Aiken
Stanford, CA
June 2015

Contents

Preface 2

I Legion Runtime Tutorial **5**

1 Installation **7**
1.1 Regent 7

2 Tasks **9**
2.1 Subtasks 12
2.2 Futures 14
2.3 Points, Rectangles and Domains 16
2.4 Index Launches 19

3 Regions **23**
3.1 Physical Instances and Permissions 25
3.2 Fill Fields 28
3.3 Coherence 29
3.4 Inline Launchers 29
3.5 Accessors 29

4 Partitioning **31**

5 Mapping **33**

6 Interoperation **35**

II Reference **37**
6.1 High Level Runtime 39
6.2 Tasks 39

6.3 Task Launchers 39
6.4 Futures 39
6.5 Regions 39
6.6 Partitions 39

Part I

Legion Runtime Tutorial

Chapter 1

Installation

The Legion homepage is `legion.stanford.edu`. Here you will find links to everything associated with the project, including a set of tutorials that are distinct from this manual. The Legion distribution is at `https://github.com/StanfordLegion/legion`. The distribution has been tested on Linux machines and Mac OS X. To install, in a shell type

```
> cd DIR
> git clone https://github.com/StanfordLegion/legion
```

where `DIR` is a directory of your choice. This command creates the directory `DIR/legion`. To complete the installation, set the environment variable `LG_RT_DIR` to `DIR/legion/runtime`. For `bash` users, an example `.bashrc` is included in `LegionManual/Examples/Installation`.

1.1 Regent

If only the Legion C++ runtime is desired, there is no need to install Regent, the companion Legion programming language, and this section can be ignored. To install Regent, download LLVM from the following URL `http://llvm.org/releases/3.4.2/clang+llvm-3.4.2-x86_64-apple-darwin10.9.xz` and extract it in a directory of your choice:

```
> cd DIR
> tar xf http://llvm.org/releases/3.4.2/clang+llvm-3.4.2-x86_64-apple-darwin10.9.xz
```

You must then put the LLVM `bin` directory in your search `PATH` and the LLVM `lib` in yor `DYD_LIBRARY_PATH`. The example `.bashrc` file in `LegionManual/Examples/Installatio` contains the necessary commands for `bash` users.

Chapter 2

Tasks

The Legion runtime is a C++ library, and Legion programs are just C++ programs that use the Legion runtime API. One important consequence of this design is that almost all Legion decisions (such as what data layout to use, in which memories to place data and on which processors to run computations) are made dynamically, during the execution of a Legion application. Dynamic decision making provides maximum flexibility, allowing the runtime's decisions to be reactive to the current state of the computation. Implementing Legion as a C++ library also allows high performance C++ code (e.g., vectorized kernels) to be used seamlessly in Legion applications.

In Legion, *tasks* are distinguished functions with a specific signature. Legion tasks have several important properties:

- Tasks are the unit of parallelism in Legion; all parallelism occurs because tasks are executed in parallel.
- Tasks are specific to a particular kind of *processor* (most commonly CPUs or GPUs), but a task can always be run on any processor of the correct kind.
- Once a processor is picked for a task, that task will execute in its entirety on that processor—tasks do not migrate.

Figure 2.1 shows a very simple, but complete, Legion program for summing the first 1000 positive integers (also available as `sum.cc` in `LegionManual/Examples/Tasks`). At a high level, every Legion program has three components:

- The id of the top-level task must be set with Legion's *high level runtime*. The top-level task is the initial task that is called when the Legion runtime starts.

```
#include <cstdio>
#include "legion.h"

using namespace LegionRuntime::HighLevel;

// All tasks must have a unique task id (a small integer).
// A global enum is a convenient way to assign task ids.
enum TaskID {
  SUM_ID,
};

void sum_task(const Task *task,
              const std::vector<PhysicalRegion> &regions,
              Context ctx,
              HighLevelRuntime *runtime)
{
  int sum = 0;
  for (int i = 0; i <= 1000; i++) {
    sum += i;
  }
  printf("The sum of 0..1000 is %d\n", sum);
}

int main(int argc, char **argv)
{
  HighLevelRuntime::set_top_level_task_id(SUM_ID);
  HighLevelRuntime::register_legion_task<sum_task>(
        SUM_ID,
        Processor::LOC_PROC,
        true/*single launch*/,
        false/*no multiple launch*/);
  return HighLevelRuntime::start(argc, argv);
}
```

Figure 2.1: Tasks/sum/sum.cc

- Every task and it's task id must be registered with the high level runtime. Currently all tasks must be registered before the runtime starts.
- The start method of the high level runtime is invoked, which in turn calls the top-level task. Note that by default this call does not return—the program is terminated when the start method terminates.

In Figure 2.1, these three steps are the three statements of `main`. The only task in this program is `sum_task`, which is also the top-level task invoked when the Legion runtime starts up. Note that the program does not say where the task is executed; that decision is made at runtime by the *mapper* (see Chapter 5). Note also that tasks can perform almost arbitrary C++ computations. In the case of `sum_task`, the computation performed is very simple, but in general tasks can call ordinary C++ functions, including allocating and deallocating memory. Tasks must not, however, call directly into other packages that provide parallelism or concurrency. Interoperation with MPI is possible but must be done in a standardized way (see Chapter 6).

As mentioned above, every task must be registered with the Legion runtime before the runtime's `start` method is called. Registration passes several arguments about a task to the runtime:

- The name of the subtask is a template argument to the `register_legion_task` method.
- The task ID is the first (regular) argument.
- The kind of processor the task can run on is the second argument. Current options are *latency optimized cores* or CPUs (constant `LOC`) and *throughput optimized cores* or GPUs (constant `TOC`).
- Two boolean flags, the first of which indicates whether the task can be used in a single task launch and the second of which indicates whether the task can be used in a multiple (or *index*) task launch.

We will see shortly that tasks can call other tasks and pass those tasks arguments and return results. Because the called task may be executed in a different address space than the caller, arguments passed between tasks must not contain C++ pointers, as these will not make sense outside of the address space in which they were created. Neither should tasks refer to global variables. A common programming error for beginning Legion programmers is to pass C++ pointers or references between tasks, or to refer to global variables from within tasks. As long as all the tasks are mapped to a single

node (i.e., the same address space) the program is likely to work, but when efforts are made to scale up the application by running on multiple nodes, C++ crashes result from the wild pointers or references to distinct instances of global variables of the same name in different address spaces. It is possible to pass data structures between tasks, but not by using C++ pointers (see Chapter 3).

All tasks have the same input signature as `sum_task`:

- `const Task *task`: An object representing the task itself.
- `const std::vector<PhysicalRegion> &regions`: A vector of *physical region instances*. This argument is the primary way to pass data between tasks (see Chapter 3).
- `Context ctx`: Every task is called in a context, which contains metadata for the task. Application programs should not directly manipulate the context.
- `HighLevelRuntime *runtime`: A pointer to the high level runtime, which gives the task access to the Legion runtime's methods.

2.1 Subtasks

Task can call other tasks, known as *subtasks*. We also refer to the calling task as the *parent task* and the called task as the *child task*. Two or more child tasks of the same parent task are *sibling tasks*. Figure 2.2 shows the definition of the parent task and the child task from the example `LegionManual/Examples/Tasks/subtask/subtask.cc`.

Consider the parent task `top_level_task`. There are two steps to executing a subtask. First, a `TaskLauncher` object is created. The `TaskLauncher` constructor takes two arguments, the ID of the task to be called and a `TaskArgument` object that holds a pointer to a buffer containing data for the subtask together with the size of the buffer. The semantics of the task arguments are particularly important. Recall that a task may be run on any processor in the system (of a kind that can execute the task). Thus, the parent task and the child task may run in different address spaces, and so the arguments are passed *by value*, meaning that the buffer pointed to by the `TaskArgument` is copied to where the subtask runs. Even if the subtask happens to run in the same address space as the parent task, the buffer referenced by the `TaskArgument` is passed by value (i.e., copied).

```
void top_level_task(const Task *task,
                    const std::vector<PhysicalRegion> &regions,
                    Context ctx,
                    HighLevelRuntime *runtime)
{
  printf("Top level task start.\n");
  for(int i = 1; i <= 100; i++) {
    TaskLauncher launcher(SUBTASK_ID, TaskArgument(&i,sizeof(int)));
    runtime->execute_task(ctx,launcher);
  }
  printf("Top level task ddone launching subtasks.\n");
}

void subtask(const Task *task,
             const std::vector<PhysicalRegion> &regions,
             Context ctx,
             HighLevelRuntime *runtime)
{
  int subtask_number = *((int *) task->args);
  printf("\tSubtask %d\n", subtask_number);
}
```

Figure 2.2: `LegionManual/Examples/Tasks/subtask/subtask.cc`

`TaskArgument` objects should be used to pass small amounts of data, such as an integer, float, struct or a (very) small array. To pass large amounts of data, use regions (see Chapter 3). As discussed earlier in this chapter, task arguments may not contain C++ pointers or references. In addition, task arguments may not contain futures (see Section 2.2).

A subtask is actually launched by the `runtime->execute_task` method, which requires both the parent task's context and the `TaskLauncher` object for the subtask as arguments. Note that the the argument buffer pointed to by the `TaskArgument` is copied only when `execute_task` is called. On the callee's side, note that the task arguments are available as a field of the `task` object. Since C++ doesn't know the type of the buffer, it is necessary to first cast the pointer to the buffer to the correct type before it can be used.

Finally, there are two other important properties of subtasks. First, the `execute_task` method is *non-blocking*, meaning it returns immediately and the subtask is executed asynchronously from the parent task, allowing the parent task to continue executing while the subtask is running (potentially) in parallel. In `subtask.cc`, the parent task launches all of the subtasks in a loop, sending each subtask a unique integer argument that the subtask simply prints out. Compile and run `subtask.cc` and observe that the parent task reports that it is done launching all of the subtasks before all of the subtasks execute. Second, parent tasks do not terminate until all of the child tasks have terminated. Thus, even though `top_level_task` reaches the end of its function body before all of its child tasks have completed, at that point the parent task waits until all the child tasks terminate, at which point `top_level_task` itself terminates.

2.2 Futures

In addition to taking arguments, subtasks may also return results. However, because a subtask executes asynchronously from its parent task, there is no guarantee that the result of the subtask will be available when the parent task or another task attempts to use it. A standard solution to this problem is to provide *futures*. A future is a value that, if read, causes the task that is performing the read to block if necessary until the value is available.

Figure 2.3 shows an excerpt from `futures.cc`, which is an extension of `substask.cc` from Section 2.1. In this example, there are two subtasks, a producer and a consumer. The top level task repeatedly calls producer/consumer pairs in a loop. The top level task first calls the producer task, passing it a unique odd integer, which the producer prints out.

```
void top_level_task(const Task *task,
                    const std::vector<PhysicalRegion> &regions,
                    Context ctx,
                    HighLevelRuntime *runtime)
{
  printf("Top level task start.\n");
  for(int i = 1; i <= 100; i += 2) {
    TaskLauncher producer_launcher(SUBTASK_PRODUCER_ID,
                                   TaskArgument(&i, sizeof(int)));
    Future doubled_task_number =
            runtime->execute_task(ctx, producer_launcher);
    TaskLauncher consumer_launcher(SUBTASK_CONSUMER_ID,
                                   TaskArgument(NULL, 0));
    consumer_launcher.add_future(doubled_task_number);
    runtime->execute_task(ctx, consumer_launcher);
  }
  printf("Top level task done launching subtasks.\n");
}

int subtask_producer(const Task *task,
                     const std::vector<PhysicalRegion> &regions,
                     Context ctx,
                     HighLevelRuntime *runtime)
{
  int subtask_number = *((int *) task->args);
  printf("\tProducer subtask %d\n", subtask_number);
  return subtask_number + 1;
}

void subtask_consumer(const Task *task,
                      const std::vector<PhysicalRegion> &regions,
                      Context ctx,
                      HighLevelRuntime *runtime)
{
  Future f = task->futures[0];
  int subtask_number = f.get_result<int>();
  printf("\tConsumer subtask %d\n", subtask_number);
}
```

Figure 2.3: `LegionManual/Examples/Tasks/futures/futures.cc`

The producer returns a unique even integer as a future. The top level task then passes this future to a consumer task that reads and prints the number.

The launch of the producer task is exactly as before in Figure 2.2. Unlike in that example, however, the producer subtask has a non-void return value, and so the `runtime->execute_task` invocation returns a useful result of type `Future`. Note that the future is passed to the consumer task using the `add_future` method of the `TaskLauncher` class, not through the `TaskArgument` object used to construct the `TaskLauncher`; futures must always be passed as arguments using `add_future` and must not be included in `TaskArgument`s. Having a distinguished method for tracking arguments to tasks that are futures allows the Legion runtime to track *dependencies* between tasks. In this case, the Legion runtime will know that the consumer task depends on the result of the corresponding producer task.

Legion gives access to the value of a future through the `get_result` method of the `Future` class, as shown in the code for `subtask_consumer` in Figure 2.3. (Note that `get_result` is templated on the type of value the future holds.) There are two interesting cases of tasks reading from futures:

- If a parent task attempts to access a future returned by one of its child tasks that has not yet completed, the parent task will block until the value of the future is available. This behavior is the standard semantics for futures, as described above. In Legion, however, this style of programming is discouraged, as blocking operations are generally detrimental to achieving the highest possible performance.

- Figure 2.3 illustrates idiomatic use of futures in Legion: a future returned by one subtask is passed as an argument to another subtask. Because Legion knows the consumer task depends on the producer task, the consumer task will not be run by the Legion runtime until the producer task has terminated. Thus, all references to the future in the consumer task are guaranteed to return immediately, without blocking.

2.3 Points, Rectangles and Domains

Up to this point we have discussed individual tasks. Legion also provides mechanisms for naming and launching sets of tasks. The ability to name and manipulate sets of things, and in particular sets of points, is useful for more than dealing with sets of tasks, and so we first present the general mechanism in Legion for defining *points*, *rectangles* and *domains*.

A *point* is an n-tuple of integers. The `Point` constructor, which is templated on the dimension n, is used to create points:

```
Point<1> one(1);                      // The 1 dimensional point <1>
Point<1> two(2);                      // The 1 dimensional point <2>
Point<1> anotherone = make_point(1); // The 1 dimensional point <1>
Point<2> zeroes = make_point(0,0);   // The 2 dimensional point <0,0>
Point<2> twos = make_point(2,2);     // The 2 dimensional point <2,2>
Point<2> threes = make_point(3,3);   // The 2 dimensional point <3,3>
Point<3> fours = make_point(4,4,4);  // The 3 dimensional point <4,4,4>
```

There are many operations defined on points. For example, points can be summed:

```
twos + threes  // the point <5,5>
```

and one can take the dot product of two points:

```
twos.dot(threes)  // the integer 12
```

All of the usual boolean operations are defined on points. The following are all true:

```
twos == twos
twos != threes
twos <= threes
```

For `<=`, the ordering is lexicographic: a point a is less than or equal to a point b if they either agree in all coordinates, or the first coordinate in which they disagree reading from the left is smaller in a than in b.

A pair of points a and b defines a *rectangle* that includes all the points that are greater than or equal to a and less than or equal to b. For example:

```
// the points  <0,0> <0,1> <0,2> <0,3>
//             <1,0> <1,1> <1,2> <1,3>
//             <2,0> <2,1> <2,2> <2,3>
//             <3,0> <3,1> <3,2> <3,3>
Rect<2> big(zeroes,threes);

// the points  <2,2> <2,3>
//             <3,2> <3,3>
Rect<2> small(twos,threes);
```

There are also many operations defined on rectangles. A few examples, all of which evaluate to true:

```
big != small
big.contains(small)
small.overlaps(big)
small.convex_hull(big) == big
small.intersection(big) == small
```

Note that the intersection of two rectangles is always a rectangle. The union of two or more rectangles is not necessarily a rectangle, however. The *convex hull* of a set of rectangles S is the smallest rectangle that contains all of the rectangles in S.

A *domain* is alternative type for rectangles. A `Rect` can be converted to a `Domain`:

```
Domain bigdomain = Domain::from_rect<2>(big);
```

The difference between the two types is that `Rect`s are templated on the dimension of the rectangle, while `Domains` are not. Legion runtime methods generally take `Domain` arguments and use `Domain`s internally, but for application code the extra type checking provided by the `Rect` type (which ensures that the operations are applied to `Rect` arguments with compatible dimensions) is useful. The recommended programming style is to create `Rect`s and convert them to `Domain`s at the point of a Legion runtime call. It is also possible to work directly with the `Domain` type, which has many of the same methods as `Rect` (see `lowlevel.h` in the `runtime/` directory).

Analagous to `Rect` and `Domain`, there is a less-typed version of the type `Point` called `DomainPoint`. Again, the difference between the two types is that the `Point` class is templated on the number of dimensions while `DomainPoint` is not. For Legion methods that require a `DomainPoint`, there is a function to convert a `Point`:

```
DomainPoint dtwos = DomainPoint::from_point<2>(twos);
```

As before, most Legion runtime calls take `DomainPoints`, but programmers should probably prefer using the `Point` type for the extra type checking provided.

The example program `LegionManual/Examples/Tasks/domains/domains.cc` includes all of the examples in this section and more.

2.4 Index Launches

We now return to the Legion mechanisms for launch multiple tasks in a single operation. The main reason for using such *index launches* is efficiency, as the overhead of starting n tasks with a single call is much less than launching n separate tasks, and the difference in performance only grows with n. Thus, when launching even tens of tasks index launches should be used, provided of course that the structure of the task launch is sufficiently regular that an index launch is natural.

Figure 2.4 gives an example that uses index launches. This program is an extension of the example in Figure 2.3, but this time instead of launching a single producer and consumer pair at a time, in Figure 2.4 all of the producers are launched in a single Legion runtime call, followed by another single call to launch all of the consumers.

We now work through this example in detail, as it introduces several new Legion runtime calls. First a one dimensional `Rect` called `launch_bounds` is created with the points `1..points`, where `points` is set to 50. The `Rect` is then converted to a `Domain` in preparation for passing it to a Legion call; one task will be launched for each point in the `Domain`. Currently, only `Rects` can be used to define the index space of a task launch—i.e., the index space must be a dense 1, 2 or 3 dimensional space.

When launching mutliple tasks simultaneously, we need some way to describe for each task what argument it should receive. There are two kinds of arguments that Legion supports: arguments that are common to all tasks (i.e., the same value is passed to all the tasks) and arguments that are specific to a particular task. Figure 2.4 illustrates how to pass a (potentially) different argument to each subtask. An `ArgumentMap` maps a point (specifically, a `DomainPoint`) p in the task index space to an argument for task p. In the figure, the `ArgumentMap` maps p to $2p$. Note that an `ArgumentMap` does not need to name an argument for every point in the index space.

The procedure for launching a set of tasks is analogous to launching a single task. First an `IndexLauncher` is created, which takes four arguments: the ID of the task to be launched, the index domain (which determines the number and identity of the tasks launched), a `TaskArgument` which is passed to all of the tasks, and an `ArgumentMap` that gives a mapping from task ID's to the per task arguments. Once the `IndexLauncher` is created it can be executed with a call to `runtime->execute_index_space`, which takes the parent task's context and the `IndexLauncher` as arguments.

Note that `execute_task_space` for the producer tasks returns not a single `Future`, but a `FutureMap`, which is analagous to the `ArgumentMap`

```
void top_level_task(const Task *task,
                    const std::vector<PhysicalRegion> &regions,
                    Context ctx,
                    HighLevelRuntime *runtime)
{
  int points = 50;
  Rect<1> launch_bounds(Point<1>(1),Point<1>(points));
  Domain launch_domain = Domain::from_rect<1>(launch_bounds);
  ArgumentMap producer_arg_map;
  for (int i = 0; i < points; i += 1)
  {
    int subtask_id = 2*i;
    producer_arg_map.set_point(DomainPoint::from_point<1>(Point<1>(i)),
                               TaskArgument(&subtask_id,sizeof(int)));
  }
  IndexLauncher producer_launcher(INDEX_PRODUCER_ID,
                                  launch_domain,
                                  TaskArgument(NULL, 0),
                                  producer_arg_map);
  FutureMap fm = runtime->execute_index_space(ctx, producer_launcher);
  ArgumentMap consumer_arg_map = fm->convert_to_argument_map();
  IndexLauncher consumer_launcher(INDEX_CONSUMER_ID,
                                  launch_domain,
                                  TaskArgument(NULL, 0),
                                  consumer_arg_map);
  runtime->execute_index_space(ctx, consumer_launcher);
}

int subtask_producer(const Task *task,
                     const std::vector<PhysicalRegion> &regions,
                     Context ctx,
                     HighLevelRuntime *runtime)
{
  int subtask_number = *((const int *)task->local_args);
  printf(``\tProducer subtask %d\n'', subtask_number);
  return subtask_number + 1;
}

void subtask_consumer(const Task *task,
                      const std::vector<PhysicalRegion> &regions,
                      Context ctx,
                      HighLevelRuntime *runtime)
{
  int subtask_number = *((const int *)task->local_args);
  printf(``\tConsumer subtask %d\n'', subtask_number);
}
```

Figure 2.4: `LegionManual/Examples/Tasks/indexlaunch/indexlaunch.cc`

in that it maps each point in the index space to a `Future`. Figure 2.4 shows one way to use the `FutureMap` by converting it to an `ArgumentMap` that is fed into the index launch for the consumer tasks. Note that the `convert_to_argument_map` call does not block on the `Future` values, but simply notifies Legion of the dependency of each point in the index space on a specific `Future`, which is used in the consumer index launch to provide the same execution semantics as the single task launches in Figure 2.3: each consumer subtask runs only after its `FutureMap` argument is available.

The subtask definitions are straightforward. Note that the argument specific to the subtask is in the field `task->local_args`. Also note that in the consumer task that the argument is not a future, but a fully evaluated `int`.

Finally, while we do not show the code in Figure 2.4, the boolean flags passed in registering the tasks with the Legion runtime are not the same as in previous examples. For instance, the registration call

```
HighLevelRuntime::register_legion_task<subtask_producer>(
  INDEX_PRODUCER_ID, Processor::LOC_PROC,
  false/* no single launch*/, true/* multiple launch*/);
```

says that the task `subtask_producer` with ID `INDEX_PRODUCER_ID` is ineligible for single task launches and must only be used for index task launches.

Chapter 3

Regions

Regions are the primary abstraction for managing data in Legion. While the examples in Chapter 2 emphasized the use of futures, this was for simplicity. As mentioned previously, futures are for passing small amounts of data between tasks; regions are for holding and processing bulk data.

Because data placement and movement is crucial to performance in modern machines, Legion provides extensive facilities for managing regions. These features are a distinctive aspect of Legion and also probably the most novel and unfamiliar to new Legion programmers. Most programming systems attempt to hide the placement, movement and organization of data; in Legion, these operations are exposed to the application.

Figure 3.1 shows a very simple program that creates a *logical region*. A logical region is a table (or, equivalently, a relation), with an *index space* defining the rows and a *field space* defining the columns. The example in Figure 3.1 has a number of details worth discussion:

- An `IndexSpace` can be structured, with rectilinear coordinates like a standard array, or unstructured, where the indices are opaque, like references. The `create_index_space` call in this program creates a structured index space with 100 elements.
- Field spaces are created in a manner analogous to index spaces. Unlike indices, whose size must be declared, there is a global upper bound on the number of fields in a field space (and exceeding this bound will cause the Legion runtime to report an error). This particular field space has only a single field `FIELD_A`.
- Once the index space and field space are created, they are used to create a logical region `lr1`. A second call to `create_logical_region` creates

```
// create a structured index space
Rect<1> rec(Point<1>(0),Point<1>(99));

// create a field space  FieldSpace fs = runtime->create_field_space(ctx
FieldAllocator field_allocator = runtime->create_field_allocator(ctx,fs)
FieldID fida = field_allocator.allocate_field(sizeof(float), FIELD_A);
assert(fida == FIELD_A);

// create two distinct logical regions
LogicalRegion lr1 = runtime->create_logical_region(ctx,sis,fs);
LogicalRegion lr2 = runtime->create_logical_region(ctx,sis,fs);

// Clean up.  IndexAllocators and FieldAllocators automatically
// have their resources reclaimed when they go out of scope.
runtime->destroy_logical_region(ctx,lr1);
runtime->destroy_logical_region(ctx,lr2);
runtime->destroy_field_space(ctx,fs);
runtime->destroy_index_space(ctx,is);
```

Figure 3.1: `LegionManual/Examples/Regions/logicalregions/logicalregions.cc`

a separate logical region `lr2`. It is very common to build multiple logical regions with either the same index space, field space or both. By providing separate steps for creating the field and index spaces prior to creating a logical region, application programmers can reuse them in the creation of multiple regions, thereby making it easier to keep all the regions in synch as the program the evolves.

Note that the logical regions in this example never hold any data. In fact, the logical regions consume no space except for their metadata (number of entries, names of the fields, etc.). A *physical instance* of a logical region holds a copy of the actual data for that region. The reason for having both concepts, logical region and physical instance, is that there is not a one-to-one relationship between logical regions and instances. It is common, for example, to have multiple physical instances of the same logical region (i.e., multiple copies) distributed around the system in some fashion to improve read performance. Because this program does not create any physical instances, no real computation takes place, either; the example simply shows how to create, and then destroy, a logical region.

```
TaskLauncher init_launcher(INIT_TASK_ID, TaskArgument(NULL,0));
init_launcher.add_region_requirement(
                RegionRequirement(lr, WRITE_DISCARD, EXCLUSIVE, lr));
init_launcher.add_field(0, FIELD_A);
runtime->execute_task(ctx, init_launcher);

TaskLauncher sum_launcher(SUM_TASK_ID, TaskArgument(NULL,0));
sum_launcher.add_region_requirement(
                RegionRequirement(lr, READ_ONLY, EXCLUSIVE, lr));
sum_launcher.add_field(0, FIELD_A);
runtime->execute_task(ctx, sum_launcher);
```

Figure 3.2: Task launches from `LegionManual/Examples/Regions/physicalregions/`

3.1 Physical Instances and Permissions

As discussed in the previous section, to actually do something with a logical region one must create a *physical instance*. The simplest way to create a physical instance is to pass a logical region to a subtask, as Legion automatically provides a physical instance to the subtask. This instance is guaranteed to be up-to-date, meaning it reflects any changes made to the region by previous tasks that the subtask depends on. In the common case, this means that the results of all previously launched tasks that updated the region will be reflected in the instance, but the programmer can specify other semantics if desired; see Section 3.3.

Figure 3.2 shows an excerpt from the top level task in `LegionManual/Examples/Regions/physicalreg` This program is an extension of the program in Figure 3.1—the creation of the (single) logical region is exactly the same as in the previous example. Here we call two tasks that both operation on the logical region `lr`. The first task intializes the elements of the region and the second sums the elements and prints out the results. As in previous examples, a `TaskLauncher` object describes the task to be called and its non-region arguments, of which there are none. When tasks also have region arguments, additional information must be added to the `TaskLauncher`. For each region the task will access, a *region requirement* must be added to the launcher using the method `add_region_requirement`. A `RegionRequirement` has four components:

- The logical region that will be accessed.
- A *permission*, which indicates how the subtask is going to use the logical region. In this program, the two tasks have different per-

missions: the initialization task accesses the region with permission `WRITE_DISCARD` (which means it will overwrite everything that was previously in the region) and the sum task accesses the region with permission `READ`. Permissions are used by the Legion runtime to determine which tasks can run in parallel. For example, if two tasks only read from a region, they can execute simultaneously. Other interesting permissions that we will see in future examples are `READ_WRITE` (the task both reads and writes the region), `WRITE` (the task only writes the region, but may not update every element as in `WRITE_DISCARD`), and `REDUCE` (the task performs reductions to the region). It is an error to attempt to access a region in a manner inconsistent with the permissions, and most such errors can be checked by the Legion runtime with appropriate debugging settings. The runtime cannot check that every element is updated when using permission `WRITE_DISCARD` and failure to do so may result in incorrect behavior.

- A *coherence mode*, which indicates what the subtask expects to see from *other* tasks that may access the region simultaneously. The mode `EXCLUSIVE` means that this subtask must appear to have exclusive access to the region—if any other tasks do access the region, any changes they make cannot be visible to this subtask. Furthermore, the subtask must see all updates from previously launched tasks. Other coherence modes that we will discuss are `ATOMIC` and `SIMULTANEOUS`.

- Finally, the region requirement names its *parent region*. We have not yet discussed subregions (see Chapter 4), so we defer a full explanation of this argument. Suffice it to say that it should either be the parent region or, if the region in question has no parent, the region itself, as in this example.

Finally, each region requirement applies to one or more fields of the region, and the method `add_field` is used to record which field(s) each region requirement applies to. In this example, there is only one region requirement with index 0 (region requirements are numbered from 0 in the order they are added to the launcher) and a single field `FIELD_A` that will be accessed by the subtask.

We now turn our attention to the two subtasks. The initialization task and the sum task have very similar structures, differing only in that the intialization task writes a "1" in `FIELD_A` of every element of the region and the sum task adds these numbers up and reports the sum. The sum task is shown in Figure 3.3.

```
using namespace LegionRuntime::Accessor;

...

void sum_task(const Task *task,
                    const std::vector<PhysicalRegion> &regions,
                    Context ctx, HighLevelRuntime *runtime)
{
  FieldID fid = FIELD_A;
  RegionAccessor<AccessorType::Generic, int> acc =
    regions[0].get_field_accessor(fid).typeify<int>();

  Domain dom = runtime->get_index_space_domain(ctx,
                    task->regions[0].region.get_index_space());
  Rect<1> rect = dom.get_rect<1>();
  int sum = 0;
  for (GenericPointInRectIterator<1> pir(rect); pir; pir++)
    {
      sum += acc.read(DomainPoint::from_point<1>(pir.p));
    }
  printf("The sum of the elements of the region is %d\n",sum);
}
```

Figure 3.3: Region accessors from `LegionManual/Examples/Regions/physicalregions/`

When `sum_task` is called, the Legion runtime guarantees that it will have access to an up-to-date physical instance of the region `lr` reflecting all the changes made by previously launched tasks that modify the `FIELD_A` of the region (which in this case is just the initialization task `init_task`). The only new feature that we need to discuss, then, is how the task accesses the data in `FIELD_A`.

Access to the fields of a region is done through a `RegionAccessor`. Accessors in Legion provide a level of indirection that shields application code from the details of how physical instances are represented in memory. Under the hood, the Legion runtime chooses among many different representations depending on the circumstances, so this extra level of abstraction avoids having those details exposed and fixed in application code. There are several different types of region accessors provided by Legion. The `Generic` accessor type used in Figure 3.3 has the virtue of having extensive debugging built in, but it is also very slow and should never be used in production code. We discuss higher performane accessors in Section 3.5. In Figure 3.3, the field `FIELD_A` is named in the creation of a `RegionAccessor` for the first (and only) physical region argument. Note that the type of the field is also included as part of the construction of the accessor.

The only other thing that is required to access the region is knowledge of the region's index space. Figure 3.3 illustrates how to recover a region's index space from a physical instance of the region using the `get_index_space` method. Since this region has a structured index space, we convert the domain to a rectangle (using the `get_rect` method). All that is left, then, is to iterate over all the points of the index space (the rectangle `rect`) and read the field `FIELD_A` for each such point in the region using the field accessor `acc`.

As a final comment, note that the accessor methods are in the namespace `LegionRuntime::Accessor` (see the first line of Figure 3.3).

3.2 Fill Fields

It is common to initialize all instances of a particular field in a region to the same value, and so Legion provides direct support for this idiom. Figure 3.4 gives an excerpt from an example identical to the one in Figure 3.3, except that the initialization task has been replaced by a call to the runtime that fills every occurrence of `FIELD_A` with a default value.

The code in Figure 3.4 uses the Legion runtime method `fill_field` to initialize every occurrence of `FIELD_A` to 1. The `fill_field` method takes

```
LogicalRegion lr = runtime->create_logical_region(ctx, sis, fs);

int init = 1;
runtime->fill_field(ctx, lr, lr, fida, &init, sizeof(init));
```

Figure 3.4: `LegionManual/Examples/Regions/fillfields/`

six arguments:

- Like almost all runtime calls, the first argument is the current task's context.
- The second argument is the region to be initialized.
- The third argument is the parent region, or the region itself if it has no parent. The parent region is needed to ensure that there are sufficient privileges to perform the initialization (`READ_WRITE` permission is required).
- The fourth argument is the ID of the field to be initialized.
- The fifth argument is a buffer holding the initial value.
- The sixth argument is the size of the buffer. The `fill_field` call makes a copy of the buffer.

The advantage of using `fill_field` is that the Legion runtime performs the initializaion lazily the next time that the field is used, which makes the operation less expensive than a normal task call. Thus, `fill_field` is preferred whenever all instances of a field are initialized to the same value.

3.3 Coherence

3.4 Inline Launchers

3.5 Accessors

Chapter 4

Partitioning

Chapter 5

Mapping

Chapter 6

Interoperation

Part II

Reference

6.1 High Level Runtime

6.2 Tasks

6.3 Task Launchers

6.4 Futures

6.5 Regions

6.6 Partitions

Bibliography

[BTSA12] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing locality and independence with logical regions. In *Supercomputing (SC)*, 2012.