Programming with Legion

Alex Aiken Michael Bauer

June 21, 2019

Preface

The first paper describing the Legion programming model was published in 2012 [BTSA12]. Since then, there has been enormous progress 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

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/Installation` 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 have *variants* specific to a particular kind of *processor* (most commonly CPUs or GPUs) and memory layout of the task's arguments. A task may have multiple variants.
- Once a task begins execution on a processor, that task will execute in its entirety on that processor—tasks do not migrate mid-computation.

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.

```
1  include <cstdio>include "legion.h"
2
3  using namespace Legion;
4
5  // All tasks must have a unique task id (a small integer).
6  // A global enum is a convenient way to assign task ids.
7  enum TaskID {
8    SUM_ID,
9  };
10
11 void sum_task(const Task *task,
12                const std::vector<PhysicalRegion> &regions,
13                Context ctx,
14                Runtime *runtime)
15 {
16   int sum = 0;
17   for (int i = 0; i <= 1000; i++) {
18     sum += i;
19   }
20   printf("The sum of 0..1000 is %d\n", sum);
21 }
22
23 int main(int argc, char **argv)
24 {
25   Runtime::set_top_level_task_id(SUM_ID);
26   Runtime::register_legion_task<sum_task>(
27         SUM_ID,
28         Processor::LOC_PROC,
29         true/*single launch*/,
30         false/*no multiple launch*/);
31   return Runtime::start(argc, argv);
32 }
```

Figure 2.1: Tasks/sum/sum.cc

- Every task and its 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 6). 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 7).

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.
- `Runtime *runtime`: A pointer to the 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).

```
1  void top_level_task(const Task *task,
2                      const std::vector<PhysicalRegion> &regions,
3                      Context ctx,
4                      Runtime *runtime)
5  {
6    printf("Top_level_task_start.\n");
7    for(int i = 1; i <= 100; i++) {
8      TaskLauncher launcher(SUBTASK_ID, TaskArgument(&i,sizeof(int)));
9      runtime->execute_task(ctx,launcher);
10   }
11   printf("Top_level_task_ddone_launching_subtasks.\n");
12 }
13
14 void subtask(const Task *task,
15              const std::vector<PhysicalRegion> &regions,
16              Context ctx,
17              Runtime *runtime)
18 {
19   int subtask_number = *((int *) task->args);
20   printf("\tSubtask_%d\n", subtask_number);
21 }
```

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. 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

```
1  void top_level_task(const Task *task,
2                      const std::vector<PhysicalRegion> &regions,
3                      Context ctx,
4                      Runtime *runtime)
5  {
6    printf("Top level task start.\n");
7    for(int i = 1; i <= 100; i += 2) {
8      TaskLauncher producer_launcher(SUBTASK_PRODUCER_ID,
9                                     TaskArgument(&i,sizeof(int)));
10     Future doubled_task_number =
11           runtime->execute_task(ctx,producer_launcher);
12     TaskLauncher consumer_launcher(SUBTASK_CONSUMER_ID,
13                                    TaskArgument(NULL,0));
14     consumer_launcher.add_future(doubled_task_number);
15     runtime->execute_task(ctx,consumer_launcher);
16   }
17   printf("Top level task done launching subtasks.\n");
18 }
19
20 int subtask_producer(const Task *task,
21                      const std::vector<PhysicalRegion> &regions,
22                      Context ctx,
23                      Runtime *runtime)
24 {
25   int subtask_number = *((int *) task->args);
26   printf("\tProducer subtask %d\n", subtask_number);
27   return subtask_number + 1;
28 }
29
30 void subtask_consumer(const Task *task,
31                       const std::vector<PhysicalRegion> &regions,
32                       Context ctx,
33                       Runtime *runtime)
34 {
35   Future f = task->futures[0];
36   int subtask_number = f.get_result<int>();
37   printf("\tConsumer subtask %d\n", subtask_number);
38 }
```

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

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<2> zeroes(0,0);   // The 2 dimensional point <0,0>
Point<2> twos(2,2);     // The 2 dimensional point <2,2>
Point<2> threes(3,3);   // The 2 dimensional point <3,3>
Point<3> fours(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
```

The following are true:

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

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.intersection(big) == small
```

Note that the intersection of two rectangles is always a rectangle. A *domain* is alternative type for rectangles. A `Rect` can be converted to a `Domain`:

```
Domain bigdomain = 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 = 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 launching 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, as index launch should be used if possible. Not all sets of tasks can be initiated using an index launch; index launches are for executing multiple instances of the same task where all of the task instances can run in parallel.

Figure 2.4 implements the same computation as the example in Figure 2.3, but 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 launch_domain` 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 an index task launch.

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. Following standard Legion practice, we first create a class derived from `IndexLauncher` for each kind of task we will use in an index launch. These classes, `ProducerTasks` and `ConsumerTasks` in this example, encapsulate all of the information about the index task launch that is the same across all calls (e.g., the task id to be launched). The `ProducerTasks` index launcher takes the launch domain and an argument map. Executing

```
1  void top_level_task(const Task *task,
2                      const std::vector<PhysicalRegion> &regions,
3                      Context ctx,
4                      Runtime *runtime)
5  {
6    // Launch 50 tasks.
7    int points = 50;
8    const Rect<1> launch_domain(1,points);
9    ArgumentMap producer_arg_map;
10   for (int i = 0; i < points; i += 1)
11   {
12     int subtask_id = 2*i;
13     producer_arg_map.set_point(DomainPoint::from_point<1>(i+1), TaskArgument(&subtask_id,sizeof(int)));
14   }
15   ProducerTasks  producer_launcher(launch_domain, producer_arg_map);
16   FutureMap fm = runtime->execute_index_space(ctx, producer_launcher);
17   ArgumentMap consumer_arg_map(fm);
18   ConsumerTasks consumer_launcher(launch_domain, consumer_arg_map);
19   runtime->execute_index_space(ctx, consumer_launcher);
20 }
21
22 int subtask_producer(const Task *task,
23                      const std::vector<PhysicalRegion> &regions,
24                      Context ctx,
25                      Runtime *runtime)
26 {
27   int subtask_number = *((const int *)task->local_args);
28   printf("\tProducer subtask %d\n", subtask_number);
29   return subtask_number + 1;
30 }
31
32 void subtask_consumer(const Task *task,
33                       const std::vector<PhysicalRegion> &regions,
34                       Context ctx,
35                       Runtime *runtime)
36 {
37   int subtask_number = *((const int*)task->local_args);
38   printf("\tConsumer subtask %d\n", subtask_number);
39 }
```

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

the `runtime->execute_index_space` method invokes all of the tasks in the launch domain.

The `execute_task_space` for the producer tasks returns not a single `Future`, but a `FutureMap`, which 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 passed to the index launch for the consumer tasks. Note that the launch of the consumer subtasks does not block waiting for all of the futures to be resolved; instead, each consumer subtask runs only afer the future it depends on is resolved.

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

Chapter 3

Regions

Regions are the primary abstraction for managing data in Legion. Futures, which the examples in Chapter 2 emphasize, are for passing small amounts of data between tasks. Regions are for holding and processing bulk data.

Because data placement and movement are 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 illustrates a number of points:

- An `IndexSpace` defines a set of indices for a region. The `create_index_space` call in this program creates a index space with 100 elements. Multidimensional index spaces can be created from multidimensional `Rect`s.
- 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`. Note that each field has an associated type, the size of which is the first argument to `allocate_field`.
- 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 a separate logical region `lr2`. It is very common to build multiple

```
1  // create an index space
2  Rect⟨1⟩ rec(Point⟨1⟩(0),Point⟨1⟩(99));
3  IndexSpace is = runtime->create_index_space(ctx,rec);
4
5  // create a field space
6  FieldSpace fs = runtime->create_field_space(ctx);
7  FieldAllocator field_allocator = runtime->create_field_allocator(ctx,fs);
8  FieldID fida = field_allocator.allocate_field(sizeof(float), FIELD_A);
9  assert(fida == FIELD_A);
10
11 // create two distinct logical regions
12 LogicalRegion lr1 = runtime->create_logical_region(ctx,is,fs);
13 LogicalRegion lr2 = runtime->create_logical_region(ctx,is,fs);
14
15 // Clean up.  IndexAllocators and FieldAllocators automatically have their resources reclaimed
16 // when they go out of scope.
17 runtime->destroy_logical_region(ctx,lr1);
18 runtime->destroy_logical_region(ctx,lr2);
19 runtime->destroy_field_space(ctx,fs);
20 runtime->destroy_index_space(ctx,is);
```

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

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 evolves.

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.

```
1  TaskLauncher init_launcher(INIT_TASK_ID, TaskArgument(NULL,0));
2  init_launcher.add_region_requirement(RegionRequirement(lr, WRITE_DISCARD, EXCLUSIVE, lr));
3  init_launcher.add_field(0, FIELD_A);
4  rt->execute_task(ctx, init_launcher);
5
6  TaskLauncher sum_launcher(SUM_TASK_ID, TaskArgument(NULL,0));
7  sum_launcher.add_region_requirement(RegionRequirement(lr, READ_ONLY, EXCLUSIVE, lr));
8  sum_launcher.add_field(0, FIELD_A);
9  rt->execute_task(ctx, sum_launcher);
```

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

3.1 Physical Instances, Region Requirements, Permissions and Accessors

Actually doing something with a logical region requires 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 **??**.

Figure 3.2 shows an excerpt from the top level task in `LegionManual/Examples/Regions/physicalregions/physicalregions.cc`. 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 operate 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 permissions: 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_ONLY`. 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` (see Chapter 5).

- 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.

```
1  void sum_task(const Task *task,
2                const std::vector<PhysicalRegion> &rgns,
3                Context ctx, Runtime *rt)
4  {
5    const FieldAccessor<READ_ONLY,int,1> fa_a(rgns[0], FIELD_A);
6    Rect<1> d = rt->get_index_space_domain(ctx,task->regions[0].region.get_index_space());
7    int sum = 0;
8    for (PointInRectIterator<1> itr(d); itr(); itr++)
9      {
10       sum += fa_a[*itr];
11     }
12   printf("The_sum_of_the_elements_of_the_region_is_%d\n",sum);
13 }
```

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

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 `FieldAccessor`. 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 has extensive debugging but it is also very slow and should never be used in production code. The `FieldAccessor` used in Figure 3.3 does no checking and is much more performant.

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 other requirement 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 dense 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`.

```
1  LogicalRegion lr = rt->create_logical_region(ctx,is,fs);
2
3  int init = 1;
4  rt->fill_field(ctx,lr,lr,fida,&init,sizeof(init));
```

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

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 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 Inline Launchers

Chapter 4

Partitioning

4.1 Independent Partitions

equal, partition by field, partition by restriction?

4.2 Set-Based Partitions

4.3 Image and Preimage Partitions

Chapter 5

Coherence

5.1 Atomic

5.2 Simultaneous

Chapter 6

Mapping

The Legion mapper interface is one of the most important parts of the Legion programming system. Through it, all (and we really do mean *ALL*) possible decisions that can impact performance are exposed. The Legion runtime has absolutely *no* internal heuristics that will aid users in achieving good or even reasonable performance. It is solely the responsibility of the user to make good mapping decisions to ensure high performance.

It is important to note that this is fundamental tenant of the system: Legion is designed for expert users such as library authors and domain specific language developers that understand precisely how a program should execute on real hardware and do not want any interference from the system in getting what they want[1]. This level of control can be overwheling at first to users who are not used to considering all the possible dimensions that influence performance in large distributed and heterogeneous systems.

In order to avoid users having to deal with this complexity initially, we provide a default implementation of this interface that we refer to as the *default mapper*. We will use several examples from the default mapper when discussing how mappers are constructed. We will also describe where possible the heuristics that the default mapper employs in order to get reasonable performance. We caution however that the default mapper is unlikely to get true speed-of-light performance for any real application as its heurisitcs will

[1]In our own experiences with large programming systems, we have often encoutered scenarios where either a static or a dynamic scheduler did not behave as we wanted, but we had no means of recourse to address the problems. The heuristics in the Linux thread scheduler and the PTX assembler in the CUDA compiler are two particularly egregious examples that have confounded us in many ways in the past. We want all expert users of Legion to have absoulte and total control over how a program executes so they never have to suffer the same fate.

likely make poor decisions during some phase of each application. Performance benchmarking using only the default mapper is strongly discouraged, while using custom mappers that extend the default mapper are completely reasonable. We promise that it will not be long into your use of Legion that there will come a moment when you will be dissatisfied with the heuristics in the default mapper. At that time, you will be grateful that the heuristic is not baked into the internals of the runtime, and that you have a means to remedy the situation yourself by writing a custom implementation of a mapper function without needing to dig into the internals of the runtime.

6.1 Mapper Organization

The Legion mapper interface is an abstract C++ class that defines a set of pure virtual functions that the Legion runtime can invoke as callbacks for making performance related decisions. A Legion mapper is therefore simply a class that inherits from the base abstract class and provides implementations of the associated pure virtual methods.

6.1.1 Mapper Registration

After the Legion runtime is created, but before the application itself begins, the user is given the opportunity to register mapper objects with the runtime. Figure 6.1 gives a small example demonstrating how to register a custom mapper with the runtime.

In order to register the `CustomMapper` objects with the runtime, the application first adds the mapper callback function by invoking the `Runtime::add_registration_callb` method which takes as an argument a function pointer to be invoked. The function pointer must have a specific type taking as arguments a `Machine` object, a `Runtime` pointer, and a reference to an STL set of `Processor` objects. The call can be invoked multiple times to record multiple callback functions (e.g., to register multiple custom mappers). All callback functions must be added prior to the invocation of the `Runtime::start` method. We recommend that users make the registration method a static method on the mapper class as is done in this example so that it is closely coupled with the actual mapper itself.

Before invoking any of the callback functions, the runtime will first create an instance of the default mapper for each processor throughout the system. The runtime then invokes the callback functions in the order that they were added. Each callback function is invoked once on each instance of the Legion runtime. For multi-process jobs, there will be one copy of the Legion runtime

```
1  void top_level_task(const Task *task,
2                      const std::vector<PhysicalRegion> &regions,
3                      Context ctx,
4                      Runtime *runtime)
5  {
6    printf("Running top level task...\n");
7  }
8
9  class CustomMapperA : public DefaultMapper {
10 public:
11   CustomMapperA(MapperRuntime *rt, Machine m, Processor p)
12     : DefaultMapper(rt, m, p) { }
13 public:
14   static void register_custom_mappers(Machine machine, Runtime *rt,
15                                       const std::set<Processor> &local_procs);
16 };
17
18 /*static*/
19 void CustomMapperA::register_custom_mappers(Machine machine, Runtime *rt,
20                                             const std::set<Processor> &local_procs)
21 {
22   printf("Replacing default mappers with custom mapper A on all processors...\n");
23   MapperRuntime *const map_rt = rt->get_mapper_runtime();
24   for (std::set<Processor>::const_iterator it = local_procs.begin();
25        it != local_procs.end(); it++)
26     {
27       rt->replace_default_mapper(new CustomMapperA(map_rt, machine, *it), *it);
28     }
29 }
30
31 class CustomMapperB : public DefaultMapper {
32 public:
33   CustomMapperB(MapperRuntime *rt, Machine m, Processor p)
34     : DefaultMapper(rt, m, p) { }
35 public:
36   static void register_custom_mappers(Machine machine, Runtime *rt,
37                                       const std::set<Processor> &local_procs);
38 };
39
40 /*static*/
41 void CustomMapperB::register_custom_mappers(Machine machine, Runtime *rt,
42                                             const std::set<Processor> &local_procs)
43 {
44   printf("Adding custom mapper B for all processors...\n");
45   MapperRuntime *const map_rt = rt->get_mapper_runtime();
46   for (std::set<Processor>::const_iterator it = local_procs.begin();
47        it != local_procs.end(); it++)
48     {
49       rt->add_mapper(1/*MapperID*/, new CustomMapperA(map_rt, machine, *it), *it);
50     }
51 }
52
53 int main(int argc, char **argv)
54 {
55   Runtime::set_top_level_task_id(TOP_LEVEL_TASK_ID);
56   {
57     TaskVariantRegistrar registrar(TOP_LEVEL_TASK_ID, "top_level_task");
58     registrar.add_constraint(ProcessorConstraint(Processor::LOC_PROC));
59     Runtime::preregister_task_variant<top_level_task>(registrar);
60   }
61   Runtime::add_registration_callback(CustomMapperA::register_custom_mappers);
62   Runtime::add_registration_callback(CustomMapperB::register_custom_mappers);
63
64   return Runtime::start(argc, argv);
65 }
```

Figure 6.1: `LegionManual/Examples/Mapping/registration/registration.cc`

per process and therefore one invocation of each callback per process. The set or processors passed into each registration callback function will be the set of application processors that are local to the process[2], thereby providing the registration callback functions with the necessary context to know which processors it will be responsible for in terms of creating new custom mappers. Note that Note that if no callback functions are registered then the only mappers that will be available are instances of the default mapper associated with each application processor.

Upon invocation, the registration callbacks should create instances of custom mappers and associate them with application processors. This can be done through one of two runtime mapper calls. The mapper can replace the default mappers (always registered with `MapperID` 0) by calling `Runtime::replace_default_mapper`. This is the only way to replace the default mappers. Alternatively, the registration callback can use `Runtime::add_mapper` to register a mapper with a new `MapperID`. Both the `Runtime::replace_default_mapper` and the `Runtime::add_mapper` methods support an optional processor argument, which tells the runtime to associate the mapper with a specific processor. If no processor is specified, the mapper will be associated with all processors on the local node. This is a mapper specific choice: whether one mapper object should handle a single application processor's mapping decisions, or whether it should handle the mapping decisions for all application processors on a node. Legion supports both use cases and it is up to custom mappers to make the best choice. From a performance perspective, the best choice is likely to rely upon the mapper synchronization model that is chosen as we will discuss in Section 6.1.2.

When creating custom mappers, the registration callback should get a pointer to the `MapperRuntime` and pass it as an argument to all mapper objects. The mapper runtime will provide the interface for mapper calls to call back into the runtime to acquire access to different physical resources. We will see instances of the use of the mapper runtime throughout the rest of the examples in this chapter.

6.1.2 Synchronization Model

Inside of the Legion runtime, there are often several different threads performing the program analysis necessary to advance the execution of a application. If some threads are performing work for operations owned by the same mapper, it is possible that they will attempt to invoke mapper calls for

[2]Mappers cannot be associated with utility processors, and therefore utility processors are not included in the set.

the same mapper object concurrently. For both productivity and correctness reasons, we do not want users having to be responsible for making their mappers thread-safe. Therefore we allow mappers to specify a *synchronization model* which the runtime will follow when concurrent mapper calls need to be made.

Each mapper object can specify its own synchronization model via the `get_mapper_sync_model` mapper call. The runtime will invoke this method exactly once per mapper object immediately after the mapper is registered with the runtime. Once the synchronization model has been set for a mapper object it cannot be changed. There are currently three different synchronization models supported by the runtime.

- Serialized Non-Reentrant - Ensure that all mapper calls to the mapper object are serialized and execute atomically. If the mapper calls out to the runtime and the mapper call is preempted, then no other mapper calls will be allowed to be invoked by the runtime. This synchronization model conforms with the original version of the Legion mapper interface.

- Serialized Reentrant - Ensure that at most one mapper call is executing at a time. However, if a mapper call invokes a runtime method that preempts the mapper call, then the runtime can either start executing another mapper call or resume a previously blocked one. It is up to the user to handle any changes in internal mapper state that might occur while a mapper call is preempted (e.g., the invalidation of STL iterators to internal mapper data structures).

- Concurrent - Permit all mapper calls to the same mapper object to proceed concurrently. Users can invoke the `lock_mapper` and `unlock_mapper` calls to perform their own synchronization of the mapper. This synchronization model is also very useful for mappers that encode mainly static mapping decisions and simply need to report their results without changing internal mapper state.

The mapper synchronization models affords mappers the choice of much complexity they are willing to accept when dealing with concurrency of mapper calls. For reference, the current default mapper uses the serialized reentrant synchronization model as it offers a good trade-off between programmability and performance.

6.1.3 Machine Interface

In order to make decisions regarding policy, it is crucial that the mapper interface have a way to introspect the underlying machine architecture. All mappers are therefore given a `Machine` object as an interface for introspecting the hardware on which the program is executing. The `Machine` object is actually a Realm level object which we directly expose to the mapper. The declaration of the interface for the `Machine` object can therefore be found in the realm/machine.h file.

There are effectively two different interfaces for querying the machine object. The old interface contains methods such as `get_all_processors` and `get_all_memories`. These methods populate STL data structures with the appropriate names of processors and memories directly. We **strongly** discourage users from using these methods as they are not scalable on large architectures where there may literally be tens to hundreds of thousands of processors or memories.

We also provide a much more efficient and scalable internface as an alternative to the old machine interface. This interface is based on the concept of a *query*. There are two types of queries: `ProcessorQuery` and `MemoryQuery`. Each query is initially given a reference to the machine object. After initialization the query will lazily represent the entire set of either processors or memories in the machine depending on the query type. The mapper can then apply various *filters* to the query to reduce the set of processor or memories of interest. These filters can include specializing the query on the kind of processors with the `only_kind` method or by requesting that the processor or memory have a specific affinity to another processor or memory with the `has_affinity_to`. Affinity can either be specified as a minimum bandwidth or a maximum latency. Figure 6.2 shows how to create a custom mapper that uses queries to find local set of processors with the same kind and the memories with affinities to the local mapper processor. In some cases, these queries can still be expensive, so we encourage the creation of mappers that memoize the results of their most commonly invoked queries to avoid duplicated work.

```
1  void top_level_task(const Task *task,
2                      const std::vector<PhysicalRegion> &regions,
3                      Context ctx,
4                      Runtime *runtime)
5  {
6    printf("Running top level task...\n");
7  }
8
9  class MachineMapper : public DefaultMapper {
10 public:
11   MachineMapper(MapperRuntime *rt, Machine m, Processor p);
12 public:
13   static void register_machine_mappers(Machine machine, Runtime *rt,
14                                        const std::set<Processor> &local_procs);
15 };
16
17 MachineMapper::MachineMapper(MapperRuntime *rt, Machine m, Processor p)
18   : DefaultMapper(rt, m, p)
19 {
20   // Find all processors of the same kind on the local node
21   Machine::ProcessorQuery proc_query(m);
22   // First restrict to the same node
23   proc_query.local_address_space();
24   // Make it the same processor kind as our processor
25   proc_query.only_kind(p.kind());
26   for (Machine::ProcessorQuery::iterator it = proc_query.begin();
27         it != proc_query.end(); it++)
28   {
29     // skip ourselves
30     if ((*it) == p)
31       continue;
32     printf("Mapper %s: shares " IDFMT "\n", get_mapper_name(), it->id);
33   }
34   // Find all the memories that are visible from this processor
35   Machine::MemoryQuery mem_query(m);
36   // Find affinity to our local processor
37   mem_query.has_affinity_to(p);
38   for (Machine::MemoryQuery::iterator it = mem_query.begin();
39         it != mem_query.end(); it++)
40     printf("Mapper %s: has affinity to memory " IDFMT "\n", get_mapper_name(), it->id);
41 }
42
43 /*static*/
44 void MachineMapper::register_machine_mappers(Machine machine, Runtime *rt,
45                                              const std::set<Processor> &local_procs)
46 {
47   printf("Replacing default mappers with custom mapper A on all processors...\n");
48   MapperRuntime *const map_rt = rt->get_mapper_runtime();
49   for (std::set<Processor>::const_iterator it = local_procs.begin();
50        it != local_procs.end(); it++)
51     {
52       rt->replace_default_mapper(new MachineMapper(map_rt, machine, *it), *it);
53     }
54 }
55
56 int main(int argc, char **argv)
57 {
58   Runtime::set_top_level_task_id(TOP_LEVEL_TASK_ID);
59   {
60     TaskVariantRegistrar registrar(TOP_LEVEL_TASK_ID, "top_level_task");
61     registrar.add_constraint(ProcessorConstraint(Processor::LOC_PROC));
62     Runtime::preregister_task_variant<top_level_task>(registrar);
63   }
64   Runtime::add_registration_callback(MachineMapper::register_machine_mappers);
65
66   return Runtime::start(argc, argv);
67 }
```

Figure 6.2: LegionManual/Examples/Mapping/machine/machine.cc

6.2 Mapping Tasks

6.2.1 Task Placement

6.2.2 Selecting Task Variants

6.2.3 Creating Physical Instances

6.2.4 Using Virtual Mappings

6.2.5 Profiling Requests

6.2.6 Resilience Support

6.3 Mapping Other Operations

6.3.1 Mapping Copies

6.3.2 Mapping Acquires and Releases

6.3.3 Mapping Must Epoch Launches

6.4 Managing Execution

6.4.1 Context Management

6.4.2 Mapper Communication

6.4.3 Controlling Stealing

Chapter 7

Interoperation

Part II

Reference

7.1 High Level Runtime

7.2 Tasks

7.3 Task Launchers

7.4 Futures

7.5 Regions

7.6 Partitions

Bibliography

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