Programming with Legion

Alex Aiken Michael Bauer

September 27, 2022 (legion-22.09.0)

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 `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
September 2022

Contents

Part I

Legion Runtime Tutorial

Chapter 1

Installation

The Legion homepage is `https://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 and macOS. 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 `Examples/Installation`.

1.1 Regent

Regent is the companion programming language for Legion. Regent provides the same programming model as the Legion C++ API, but with a nicer syntax, static checking of various requirements of Legion programs, and compile-time optimizations. Instructions for installing Regent are maintained at `https://regent-lang.org/install`.

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, but there is also experimental support for FPGAs) 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 `Examples/Tasks`). This example, like every other example in this manual, can be run by first setting an environment variable to point to the Legion runtime

```
export LG_RT_DIR="...path to legion directory.../legion/runtime"
```

```
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, Runtime *runtime)
14 {
15   int sum = 0;
16   for (int i = 0; i <= 1000; i++) {
17     sum += i;
18   }
19   printf("The sum of 0..1000 is %d\n", sum);
20 }
21
22 int main(int argc, char **argv)
23 {
24   Runtime::set_top_level_task_id(SUM_ID);
25   {
26     TaskVariantRegistrar registrar(SUM_ID, "sum");
27     registrar.add_constraint(ProcessorConstraint(Processor::LOC_PROC));
28     Runtime::preregister_task_variant<sum_task>(registrar);
29   }
30   return Runtime::start(argc, argv);
31 }
```

Figure 2.1: `Examples/Tasks/sum/sum.cc`

and then typing `make` in the directory containing the example.

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.
- 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 7). 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 OpenMP and MPI is possible but must be done in a standardized way (see Chapter 8).

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 task 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. The most important options are *latency optimized cores* or CPUs (constant `LOC`) and *throughput optimized cores* or GPUs (constant `TOC`). Declaring the processor kind for a task is an example of a *constraint*. Legion has an extensive system of constraints that can be used to direct the Legion runtime in running a Legion program. There are other kinds of constraints that can be specified for tasks, but the processor kind is the most commonly used.

- 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. Legion provides its own abstractions for passing data structures between tasks (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 `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`

```
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    TOP_LEVEL_TASK_ID,
9    SUBTASK_ID
10 };
11
12 void top_level_task(const Task *task,
13                     const std::vector<PhysicalRegion> &regions,
14                     Context ctx,
15                     Runtime *runtime)
16 {
17   printf("Top level task start.\n");
18   for(int i = 1; i <= 100; i++) {
19     TaskLauncher launcher(SUBTASK_ID, TaskArgument(&i,sizeof(int)));
20     runtime->execute_task(ctx,launcher);
21   }
22   printf("Top level task ddone launching subtasks.\n");
23 }
24
25 void subtask(const Task *task,
26              const std::vector<PhysicalRegion> &regions,
27              Context ctx,
28              Runtime *runtime)
29 {
30   int subtask_number = *((int *) task->args);
31   printf("\tSubtask %d\n", subtask_number);
32 }
33
34 int main(int argc, char **argv)
35 {
36   Runtime::set_top_level_task_id(TOP_LEVEL_TASK_ID);
37   {
38     TaskVariantRegistrar registrar(TOP_LEVEL_TASK_ID, "top_level_task");
39     registrar.add_constraint(ProcessorConstraint(Processor::LOC_PROC));
40     Runtime::preregister_task_variant<top_level_task>(registrar);
41   }
42   {
43     TaskVariantRegistrar registrar(SUBTASK_ID, "subtask");
44     registrar.add_constraint(ProcessorConstraint(Processor::LOC_PROC));
45     Runtime::preregister_task_variant<subtask>(registrar);
46   }
47   return Runtime::start(argc, argv);
48 }
```

Figure 2.2: `Examples/Tasks/subtasks/subtasks.cc`

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).

`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 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, a parent task does not terminate until all of its 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.

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

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

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 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 an 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. Most of these type conversions will be handled implicitly—the programmer usually does not need to explicitly cast a `Rect` to a `Domain`. 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 `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, an 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. Note that while the application code uses `Rects` and `Points` that the signatures of the runtime interfaces that are called use `Domains` and `DomainPoints` and Legion takes care of the conversions.

When launching multiple 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 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

```
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(Point<1>(i+1), TaskArgument(&subtask_id,sizeof(int)));
14   }
15   ProducerTasks  producer_launcher(launch_domain, producer_arg_map);
16   //
17   //  Since each producer task returns an integer, the index launch will return a FutureMap, a map from
18   //  the launch domain to a future for each point.  The FutureMap can be used as an ArgumentMap to a subsequent
19   //  index launch.
20   //
21   FutureMap fm = runtime->execute_index_space(ctx, producer_launcher);
22   ArgumentMap consumer_arg_map(fm);
23   ConsumerTasks consumer_launcher(launch_domain, consumer_arg_map);
24   runtime->execute_index_space(ctx, consumer_launcher);
25 }
26
27 int subtask_producer(const Task *task,
28                      const std::vector<PhysicalRegion> &regions,
29                      Context ctx,
30                      Runtime *runtime)
31 {
32   int subtask_number = *((const int *)task->local_args);
33   printf("\tProducer subtask %d\n", subtask_number);
34   return subtask_number + 1;
35 }
36
37 void subtask_consumer(const Task *task,
38                       const std::vector<PhysicalRegion> &regions,
39                       Context ctx,
40                       Runtime *runtime)
41 {
42   int subtask_number = *((const int*)task->local_args);
43   printf("\tConsumer subtask %d\n", subtask_number);
44 }
```

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

`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 after 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 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: `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 sync as the program evolves.

Logical regions never hold any data. In fact, 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 `Examples/Regions/physicalregions/physicalregions.cc`.

3.1 Physical Instances, Region Requirements, Privileges 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 Chapter 6.

Figure 3.2 shows an excerpt from the top level task in `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 *privilege*, which indicates how the subtask will use the logical region. In this program, the two tasks have different privileges: the initialization task accesses the region with privilege `WRITE_DISCARD` (which means it will overwrite everything that was previously in the region) and

the sum task accesses the region with privilege `READ_ONLY`. Privileges 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 privileges 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 privileges, and most such errors can be checked by the Legion runtime with appropriate debugging settings. The runtime cannot check that every region element is updated when using privilege `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 6).

- 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 `Examples/Regions/physicalregions/physicalregions.cc`.

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.

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 `fa_a`.

The example in Figure 3.3 uses an iterator, which is convenient when the index space is a dense rectangle and one wants to operate on all of the points in a region. Accessors can also take a `Point` argument of the correct dimension for their region to directly access a single point in the index space.

```
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: `Examples/Regions/fillfields/fillfields.cc`

There are many different types of region accessors provided by Legion. We mention a few of the more common ones here; the comments in `legion/runtime/legion.h` provides a good overview of the complete set of accessors.

- There are many accessor constructors pre-defined for different combinations of privileges and field types. For example, a `AccessorROfloat` is the type of an accessor with read-only privileges on a field of type `float`. The accessor in Figure 3.3 could have been constructed using `AccessorROint(regns[0],FIELD_A)` instead of directly invoking the `FieldAccessor` template.
- There is a different template, `ReductionAccessor`, to use with reduction privileges. For instances with reduction-only privileges, only `ReductionAccessor`s should be used.
- The `Generic` accessor has extensive debugging support and will, for example, detect out of bounds accesses, which is a common programming error. The `Generic` accessor 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.

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` privilege 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

The most common way to gain access to a region r is by launching a task t with a region requirement on r, in which case the Legion runtime will automatically map a physical instance of r that will be accessible in t. There are situations where a task may need to map a physical instance of a region explicitly, such as when a task needs to access a newly created region or a new region is returned from a child task. Figure 3.5 shows the use of an *inline mapping* to explicitly map a physical region. After creating a new logical region an `InlineLauncher` object (so named because it has similar functionality to a `TaskLauncher` object) is created with a region requirement and any associated fields. The runtime methods `map_region` and `unmap_region` assign and unassign a physical instance to `pr`. Note that `map_region` is an asynchronous call and it is necessary to wait for the physical instance to become valid before it can be used.

A important invariant maintained by the Legion runtime system is that tasks have exclusive access to regions to which they have write access (we will see how to relax this requirement in Chapter 6). This invariant implies that when a parent task calls a child task, any regions passed to the child that may be written must be unmapped before the call and remapped after the call. (To see that unmapping a region before passing it to a child task is necessary, keep in mind that task calls are asynchronous, and so the parent

```
1  LogicalRegion lr = rt->create_logical_region(ctx,is,fs);
2  InlineLauncher launcher(RegionRequirement(lr, WRITE_DISCARD, EXCLUSIVE, lr).add_field(0,FIELD_A));
3  PhysicalRegion pr = rt->map_region(ctx, launcher);
4  pr.wait_until_valid();
```

... lines omitted ...

```
1  // cleanup
2  rt->unmap_region(ctx, pr);
```

Figure 3.5: `Examples/Regions/inlinemapping/inlinemapping.cc`

and child tasks may execute in parallel. If a parent task does not unmap a region required by a child task with write privileges, the application will most likely deadlock.) For regions passed as region requirements to task launches and where the programmer has not explicitly mapped (or unmapped) the region, the runtime system automatically wraps the task call in the necessary calls to `unmap_region` and `map_region`. In cases where the parent task does not touch the region across many child task calls, performance can be improved if the application explicitly unmaps the region at the earliest possible point and maps the region again at the latest possible point, thereby avoiding any mapping/unmapping of the region during intermediate task calls. Whenever a program explicitly maps or unmaps a region r within a task, the Legion runtime will no longer silently wrap child task invocations with calls to unmap/map r.

3.4 Layout Constraints

In Chapter 2 we introduced the idea of a *constraint*, a restriction specified by the program on how the Legion runtime may use certain application objects, such as specifying what kind of processor a task can execute on. The most commonly used constraints on regions are *layout constraints*.

Figure 3.6 shows the `main` function from one of the examples in the Legion repository. The function first defines two layout constraints, named `column_major` (lines 5-14) and `row_major` (liines 16-25). The constants `DIM_X`, `DIM_Y` and `DIM_Z` are distinguished names given to the first three dimensions of an index space; `DIM_F` stands for all the fields of a region. An `OrderingConstraint` specifies which dimension varies the fastest in a region's layout: In `column_major` it is the `x` dimension, in `row_major` it is the `y` dimension. The `z` and higher dimensions are ignored here because the

```
1  int main(int argc, char **argv)
2  {
3    Runtime::set_top_level_task_id(TOP_LEVEL_TASK_ID);
4
5    LayoutConstraintID column_major;
6    {
7      OrderingConstraint order(true/*contiguous*/);
8      order.ordering.push_back(DIM_X);
9      order.ordering.push_back(DIM_Y);
10     order.ordering.push_back(DIM_F);
11     LayoutConstraintRegistrar registrar;
12     registrar.add_constraint(order);
13     column_major = Runtime::preregister_layout(registrar);
14   }
15
16   LayoutConstraintID row_major;
17   {
18     OrderingConstraint order(true/*contiguous*/);
19     order.ordering.push_back(DIM_Y);
20     order.ordering.push_back(DIM_X);
21     order.ordering.push_back(DIM_F);
22     LayoutConstraintRegistrar registrar;
23     registrar.add_constraint(order);
24     row_major = Runtime::preregister_layout(registrar);
25   }
26
27   {
28     TaskVariantRegistrar registrar(TOP_LEVEL_TASK_ID, "top_level");
29     registrar.add_constraint(ProcessorConstraint(Processor::LOC_PROC));
30     Runtime::preregister_task_variant<top_level_task>(registrar, "top_level");
31   }
32
33   {
34     TaskVariantRegistrar registrar(INIT_MATRIX_TASK_ID, "init_matrix");
35     registrar.add_constraint(ProcessorConstraint(Processor::LOC_PROC));
36     registrar.add_layout_constraint_set(0, column_major);
37     registrar.set_leaf();
38     Runtime::preregister_task_variant<init_matrix_task>(registrar, "init_matrix");
39   }
40
41   {
42     TaskVariantRegistrar registrar(TRANSPOSE_MATRIX_TASK_ID, "transpose");
43     registrar.add_constraint(ProcessorConstraint(Processor::LOC_PROC));
44     registrar.add_layout_constraint_set(0, column_major);
45     registrar.add_layout_constraint_set(1, row_major);
46     registrar.set_leaf();
47     Runtime::preregister_task_variant<transpose_matrix_task>(registrar, "transpose");
48   }
49
50   {
51     TaskVariantRegistrar registrar(CHECK_MATRIX_TASK_ID, "check_matrix");
52     registrar.add_constraint(ProcessorConstraint(Processor::LOC_PROC));
53     registrar.add_layout_constraint_set(0, column_major);
54     registrar.set_leaf();
55     Runtime::preregister_task_variant<check_matrix_task>(registrar, "check_matrix");
56   }
57
58   return Runtime::start(argc, argv);
59 }
```

Figure 3.6: The `main` function from `legion/examples/layout_constraints/transpose.cc`.

regions involved for this application are two dimensional. Note that these are struct-of-array layouts; by putting the field dimension first we could specify array-of-struct layouts as well.

To use a layout constraint we associate it with a region argument of a task. When the `transpose_matrix_task` is registered (lines 42-47), the first region argument (region 0) is constrained to be in `column_major` layout while the second region argument (region 1) is constrained to be in `row_major` layout. When the task is run the runtime system ensures that the physical instances used by the task adhere to any layout constraints.

It is also possible to specify blocked layouts. See the declarations and comments in `legion/runtime/legion/legion_constraint.h`.

Layout constrains are most commonly used in two situations. First, the code for a task may require a certain layout. For example, vectorized code will necessarily require that the vectorized dimension be the fastest varying. Second, when interoperating with external systems, by using layout constraints the Legion system can describe what the layout of the pre-existing data is, allowing the runtime to correctly interpret it and work with it without making unnecesssary copies.

Chapter 4

Partitioning

The ability to partition regions into *subregions* is a core feature of Legion. Many parallel programming systems have some notion of a distributed collection—a collection of data that is broken up into pieces and put in different places across a distributed machine. In Legion, the facilities for partitioning data are more expressive than most ther programming systems in several important ways. First, partitioning can be done recursively to arbitrary levels: regions can be partitioned into subregions, which can themselves be partitioned further into additional subregions, and so on. The partitioning hierarchy defines a tree, called the *region tree*, that is a useful abstraction of how data is organized in a Legion application. An important step in designing a Legion application is deciding how data will be partitioned—i.e., deciding what the region tree will look like.

A second distinguishing characteristic is that partitioning is done dynamically: The application can create and destroy region partitions at runtime. Thus, Legion can naturally express methods where the organization of data needs to change during the computation, such as adaptive mesh refinement algorithms. It is worth remembering, however, that partitioning can be an expensive operation, so it is important that it be used judiciously. As long as the cost of the partitioning is amortized over lots of computation on the partition's subregions, no performance problems from partitioning will arise.

A third distinguishing characteristic is that partitions are themselves first-class objects in Legion. A *partition* is a collection of subregions, and Legion has many different built-in operations for creating useful partitions; presenting the most common of these methods for partitioning data is the heart of this chapter.

Partitions do not need to be mathematical partitions, in fact in Legion a

partition can be *any* set of subsets of a region. It is important to keep in mind that partitions only name subsets of data; a partition does not allocate storage for the subregions or make copies of data. This feature of partitions leads to some useful programming idioms. For example, a program can create a very large region, perhaps with billions of elements, and then partition it and only materialize needed subregions, which is useful in cases where there is a very large potential domain but only a relatively small subset will actually be used. It is also common to create a region too large to fit in any physical memory, partition it into subregions that will fit on each node, and then create physical instances only of the subregions. The global name space is still useful (e.g., in neighbor computations for stencils) even if the parent region is never physically allocated.

The subregions in a partition can overlap (share elements), in which case we say the partition is *aliased*; otherwise the partition is *disjoint*. For example, aliased subregions can be useful in describing stencil patterns, where the subblocks overlap by the width of the stencil perations. A *complete* partition is one in which every element of the partitioned region is in at least one subregion of the partition. Partitions in Legion do not need to be complete; as an example, it is often useful to name only the overlapping boundaries of blocks in stencils (the so-called *ghost regions*, which are a strict subset of the entire computational domain).

These idioms for using partitions are illustrated in the examples in the Legion repository. In this tutorial we focus on illustrating the basic application of the most commonly used Legion partitioning operators.

4.1 Equal Partitions

The simplest and most common case of partitioning is an *equal* partition, where a region is automatically partitioned into subregions of approximately equal size. Figure 4.1 gives an example of partitioning a region into four equal-size subregions. The number of subregions in the partition is given by an index space, with one subregion created for each of the index space's points. On line 17 of Figure 4.1 an index space with four points is created from a 1D `Rect` defined on line 16. This set of *colors* (the term for the points in an index space used for naming the subregions of a partition) is passed as an argument to `create_equal_partition` on line 18. Note that the partitioning operation is applied to the index space, not directly to a logical region. By partitioning the index space, the same partitioning can be reused for multiple logical regions that share an index space. The

`get_logical_partition` call on line 19 returns the logical partition of the region defined by the index partition.

Lines 25-28 illustrate an important Legion idiom where an index launch is performed over the subregions of a partition. Here the `sum_task` is applied to each subregion of the `lp` partition of the region `lr`. Note that `IndexLauncher` created on line 25 uses the color space of the partition as its launch domain. A region requirement for `sum_task` is added to the launcher on line 26. We saw region requirements in Chapter 3 for launches of individual tasks. Region requirements for index launches have slightly different arguments:

- For an index launch the first argument is a logical partition. (For an individual task the first argument is a logical region.)
- If the first argument is a logical partition, then the second argument is the identifier of a *projection functor* f. For index point i in the launch space and logical partition lp,, the task is passed $f(lp, i)$ as its argument. Projection function 0 is predefined to return the ith subregion for the ith point in the launch domain—i.e., the task will be applied to every subregion of lp, which is the most common case in practice. Users can write and register their own projection functors with the runtime system, much like task registration, for more complex patterns of selecting the arguemnts to index tasks from a region tree.
- The rest of the fields are the same for both kinds of region requirements: the privilege for the region (`READ_ONLY` in this example), the coherence mode (`EXCLUSIVE`), and the parent region (`lr`) from which privileges are derived.

On line 27 the field `FIELD_A` is added to the region requirement. The naming of the fields in a region requirment is separated from the region requirement's construction because any number of fields can be part of a region requirement; these are all the fields that the task will touch with the given permissions and coherence mode.

The execution of the launcher on line 28 runs `sum_task` on all the subregion sof `lp`. Instead of summing the entire region, the same `sum_task` is now used to sum all four subregions separately.

An equal partition is an example of a mathematical partition: an equal partition is always both disjoint (none of the subregions overlap) and complete (every element of the region is included in some subregion).

The runtime does not guarantee anything about equal partitions other than that the subreqions will be of approximately the same size. At the

```
1  void top_level_task(const Task *task,
2                      const std::vector<PhysicalRegion> &rgns,
3                      Context ctx,
4                      Runtime *rt)
5  {
6    Rect<1> rec(Point<1>(0),Point<1>(99));
7    IndexSpace is = rt->create_index_space(ctx,rec);
8    FieldSpace fs = rt->create_field_space(ctx);
9    FieldAllocator field_allocator = rt->create_field_allocator(ctx,fs);
10   FieldID fida = field_allocator.allocate_field(sizeof(int), FIELD_A);
11   assert(fida == FIELD_A);
12
13
14   LogicalRegion lr = rt->create_logical_region(ctx,is,fs);
15   int num_subregions = 4;
16   Rect<1> colors(0,num_subregions - 1);
17   IndexSpace color_is = rt->create_index_space(ctx, colors);
18   IndexPartition ip = rt->create_equal_partition(ctx, is, color_is);
19   LogicalPartition lp = rt->get_logical_partition(ctx, lr, ip);
20
21   int init = 1;
22   rt->fill_field(ctx,lr,lr,fida,&init,sizeof(init));
23
24   ArgumentMap arg_map;
25   IndexLauncher sum_launcher(SUM_TASK_ID, colors, TaskArgument(NULL,0), arg_map);
26   sum_launcher.add_region_requirement(RegionRequirement(lp, 0, READ_ONLY, EXCLUSIVE, lr));
27   sum_launcher.region_requirements[0].add_field(FIELD_A);
28   rt->execute_index_space(ctx, sum_launcher);
29
30   // Clean up.  IndexAllocators and FieldAllocators automatically have their resources reclaimed
31   // when they go out of scope.
32   rt->destroy_logical_region(ctx,lr);
33   rt->destroy_field_space(ctx,fs);
34   rt->destroy_index_space(ctx,is);
35 }
```

Figure 4.1: `Examples/Partitions/equal/equal.cc`

time of this writing, for example, an equal partition of a multidimensional region will partition the region in just the first dimension. If a specific kind of equal partition is desired other partitioning operators can be used. For example, a blocked partition can be created with partition by restriction (see Section 4.3).

4.2 Partition by Field

Equal partitioning leaves the choice of how to partition the elements of a region up to the runtime system, requiring only that the sizes of subregions are equal or nearly equal. At the other extreme is *partition by field*, where the

application prescribes for each individual element of a region which subregion it should be assigned to.

The program in Figure 4.2 illustrates partitioning by field. This example is similar to the previous one, but there is now an additional field `FIELD_PARTITION` that holds a coloring of the elements of the region. The `color_launcher` on lines 23-26 invokes the `color_task` which assigns a color (a `Point<1>` in this example; colors can also be multidimensional points) to each element of the region. In this case the assignment is a simple blocking, with each contiguous quarter of the elements assigned the same color (lines 46-56; the function uses the number of colors as the divisor, which is 4 in this example), but the coloring could be any assignment of the color space to the elements of the field. Note also that the coloring is dynamically computed; programs can compute a coloring for a field and then partition the region accordingly.

The actual partitioning of the index space occurs on line 28. The runtime call `create_partition_by_field` takes the current context, the region, the parent region (or the same region if it has no parent, as in this case), and the field with the coloring. On line 29 the partition of the index space is used to retrieve the logical partition.

The logic for launching the `sum_task` over the paritition on lines 31-35 is the same as in Figure 4.1.

4.3 Partition by Restriction

Another common partitioning idiom is to divide a region into blocks of the same size, a *blocked partitioning*. For applications involving stencils, it can also be useful for the blocks to include "ghost cells" adjacent to the block, essentially expanding the block in one or more dimensions. Figure 4.3 shows a 1D region partitioned into four subblocks, where each block includes one ghost element on each side. For a 1D region of 100 elements as shown in the figure, the result is four subblocks of size 26, 27, 27, and 26: the two interior blocks have a ghost element on each side, while the first and last blocks have one ghost element each as the other element would be out of bounds of the region.

Figure 4.4 gives code for computing the partitioning shown in Figure 4.3. The essential differences from previous examples are on lines 21-27. Starting with the `create_partition_by_restriction` call itself on line 27, we see that in addition to the usual context, index space, and color space this operation also takes a *transform* and an *extent*. The extent E is a "generic"

```
1  void top_level_task(const Task *task,
2                      const std::vector<PhysicalRegion> &rgns,
3                      Context ctx,
4                      Runtime *rt)
5  {
6    Rect<1> rec(Point<1>(0),Point<1>(99));
7    IndexSpace is = rt->create_index_space(ctx,rec);
8    FieldSpace fs = rt->create_field_space(ctx);
9    FieldAllocator field_allocator = rt->create_field_allocator(ctx,fs);
10   FieldID fida = field_allocator.allocate_field(sizeof(int), FIELD_A);
11   FieldID fidp = field_allocator.allocate_field(sizeof(Point<1>), FIELD_PARTITION);
12   assert(fida == FIELD_A);
13   assert(fidp == FIELD_PARTITION);
14
15   LogicalRegion lr = rt->create_logical_region(ctx,is,fs);
16
17   int init = 1;
18   rt->fill_field(ctx,lr,lr,fida,&init,sizeof(init));
19
20   int num_subregions = 4;
21   Rect<1> colors(0,num_subregions-1);
22   IndexSpace cis = rt->create_index_space(ctx,colors);
23   TaskLauncher color_launcher(COLOR_TASK_ID, TaskArgument(&colors,sizeof(Rect<1>)));
24   color_launcher.add_region_requirement(RegionRequirement(lr, WRITE_DISCARD, EXCLUSIVE, lr));
25   color_launcher.add_field(0,FIELD_PARTITION);
26   rt->execute_task(ctx, color_launcher);
27
28   IndexPartition ip = rt->create_partition_by_field(ctx, lr, lr, FIELD_PARTITION,cis);
29   LogicalPartition lp = rt->get_logical_partition(ctx, lr, ip);
30
31   ArgumentMap arg_map;
32   IndexLauncher sum_launcher(SUM_TASK_ID, colors, TaskArgument(NULL,0), arg_map);
33   sum_launcher.add_region_requirement(RegionRequirement(lp, 0, READ_ONLY, EXCLUSIVE, lr));
34   sum_launcher.region_requirements[0].add_field(FIELD_A);
35   rt->execute_index_space(ctx, sum_launcher);
36 }
37
38 void color_task(const Task *task,
39                 const std::vector<PhysicalRegion> &rgns,
40                 Context ctx, Runtime *rt)
41 {
42   //
43   //  FieldAccessor is templated on privilege, field type, and number of index space dimensions
44   //
45
46   Rect<1> colors = *((Rect<1> *) task->args);
47   int divisor = (colors.hi - colors.lo) + 1;
48   const FieldAccessor<WRITE_DISCARD,Point<1>,1> fa_p(rgns[0], FIELD_PARTITION);
49   Rect<1> d = rt->get_index_space_domain(ctx, task->regions[0].region.get_index_space());
50   int quarter = ((int) (d.hi-d.lo) + 1) / divisor; // assume this number is an integer
51   int x = 0;
52
53   for (PointInRectIterator<1> itr(d); itr(); itr++)
54     {
55       fa_p[*itr] =  Point<1>((int) (x / quarter));
56       x = x + 1;
57     }
58 }
```

Figure 4.2: `Examples/Partitions/partition_by_field/pbf.cc`

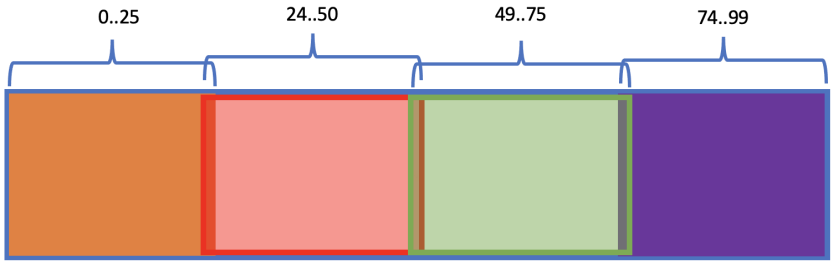

Figure 4.3: A blocked partition of a 1D region with ghost elements.

rectangle of the desired size—all subregions in the partition will have the shape of E. The `transform` T is an $n \times m$ matrix, where n is the number of dimensions of the color space and m is the number of dimensions of the region. For a point p in the color space, the points in the corresponding subregion are defined by the rectangle $Tp + E$. That is Tp defines an offset that is added to E to name the points in the subregion associated with p.

In this example, since the region and color space are both 1D, the transform is a 1x1 matrix, a single integer (lines 21-23); this transform says that subregions will start 25 elements apart (the `blocksize`). The extent defined on line 26 says that a subregion will extend one element to the left and 25 elements to the right of the 0-point of the subregion, so in general each subregion will have 27 elements. Legion automatically clips any subregions that extend beyond the bounds of the region being partitioned, so for a color space of $0, 1, 2, 3$ the corresponding subregions will have elements $0..25, 24..50, 49..75, 74..99$. Note that unlike the other partitioning operations we have seen so far, the values of the color space are significant and affect the position of the subregion—for a region with an index space 0..99, it is necessary that the color space be 0..3 and not some other set of four points.

4.4 Set-Based Partitions

Equal partitions, partitions by field and partitions by restriction are all examples of *independent* partitions, which are partitions that do not depend on other partitions. Legion also provides a number of *dependent* partitioning operators that take partitions as input and produce partitions as output. Dependent partitioning is used heavily in most Legion programs; it is not uncommon to see long chains of partitioning operators to name complex

```
1  void top_level_task(const Task *task,
2                      const std::vector<PhysicalRegion> &rgns,
3                      Context ctx,
4                      Runtime *rt)
5  {
6    Rect<1> rec(Point<1>(0),Point<1>(99));
7    IndexSpace is = rt->create_index_space(ctx,rec);
8    FieldSpace fs = rt->create_field_space(ctx);
9    FieldAllocator field_allocator = rt->create_field_allocator(ctx,fs);
10   FieldID fida = field_allocator.allocate_field(sizeof(int), FIELD_A);
11   assert(fida == FIELD_A);
12
13
14   LogicalRegion lr = rt->create_logical_region(ctx,is,fs);
15   int init = 1;
16   rt->fill_field(ctx,lr,lr,fida,&init,sizeof(init));
17
18   int num_subregions = 4;
19   Rect<1> colors(0,num_subregions - 1);
20   IndexSpace color_is = rt->create_index_space(ctx, colors);
21   int block_size = 25;
22   Transform<1,1> transform;
23   transform[0][0] = block_size;
24   // Each subregion will have one "ghost" element on each side.
25   // Ghost elements out of bounds of the parent region are clipped.
26   Rect<1> extent(-1, block_size);
27   IndexPartition ip =  rt->create_partition_by_restriction(ctx, is, color_is, transform, extent);
28   LogicalPartition lp = rt->get_logical_partition(ctx, lr, ip);
29
30   ArgumentMap arg_map;
31   IndexLauncher sum_launcher(SUM_TASK_ID, colors, TaskArgument(NULL,0), arg_map);
32   sum_launcher.add_region_requirement(RegionRequirement(lp, 0, READ_ONLY, EXCLUSIVE, lr));
33   sum_launcher.region_requirements[0].add_field(FIELD_A);
34   rt->execute_index_space(ctx, sum_launcher);
35 }
```

Figure 4.4: `Examples/Partitions/partition_by_restriction/pbr.cc`

subsets of the data that the program needs to manipulate.

An *difference* partition takes two index space partitions and computes their set difference by color: A subspace of the difference partition is the set difference of two subspaces, one from each of the two argument partitions, with the same color. Thus, there is a subspace in the difference partition for every color that the two argument partitions have in common.

Figure 4.5 gives an example that creates two partitions by restriction: a "big" partition that includes blocks of 26 elements spaced 25 elements apart (lines 22-28, similar to the partition in Figure 4.2 but with only one ghost element per subspace) and a "small" partition that is a disjoint partition of blocks of 25 elements spaced 25 elements apart (lines 22-24, 26, and 29). The region has 101 elements (line 6) to ensure that every subspace of the big partition actually has 26 elements (i.e., no elements are clipped for being out of bounds). The difference partition subtracts the small partition from the big partition, so each subspace in the index partition names exactly the ghost element of the corresponding big subspace. The result of the `sum_task` shows that there is exactly one element in each subregion of the difference partition.

Legion also provides `create_partition_by_union`, which computes the set union of two partitions, and `create_partition_by_intersection`, which computes the set intersection of two partitions. These dependent partitioning functions have the same signature as `create_partition_by_difference`.

4.5 Image Partitions

Often we need to partition a region in a way compatible with an already computed partition of another region. For example, consider a graph represented by a region of nodes and a region of edges. Assume we have chosen a partition of the nodes $\texttt{P}_\texttt{N}[0], \ldots, \texttt{P}_\texttt{N}[k]$. We will often want to partition the edges into subregions $\texttt{P}_\texttt{E}[0], \ldots, \texttt{P}_\texttt{E}[k]$ such that the edges in $\texttt{P}_\texttt{E}[i]$ all have source (or alternatively destination) nodes in $\texttt{P}_\texttt{N}[i]$. The *image* and *preimage* partitioning operators discussed in this section and the next provide mechanisms for using a pointer relationship between two regions to induce a partitioning of one region given a partitioning of the other.

The example in Figure 4.7 creates two regions. The region `lr_src` (line 19) has a single field `FIELD_PTR` of type `Point<1>` (line 10). Pointers between regions are represented by points in the index space of the pointed-to region, which in this case is `lr_dst` (line 20). The `ptr_task` (defined on lines 47-60 and called on lines 32-36) assigns pointers so that the ith element of `lr_src`

```
1  void top_level_task(const Task *task,
2                      const std::vector<PhysicalRegion> &rgns,
3                      Context ctx,
4                      Runtime *rt)
5  {
6    Rect<1> rec(Point<1>(0),Point<1>(100));
7    IndexSpace is = rt->create_index_space(ctx,rec);
8    FieldSpace fs = rt->create_field_space(ctx);
9    FieldAllocator field_allocator = rt->create_field_allocator(ctx,fs);
10   FieldID fida = field_allocator.allocate_field(sizeof(int), FIELD_A);
11   assert(fida == FIELD_A);
12
13   LogicalRegion lr = rt->create_logical_region(ctx,is,fs);
14
15   int init = 1;
16   rt->fill_field(ctx,lr,lr,fida,&init,sizeof(init));
17
18   int num_subregions = 4;
19   Rect<1> colors(0,num_subregions-1);
20   IndexSpace cis = rt->create_index_space(ctx,colors);
21
22   int block_size = 25;
23   Transform<1,1> transform;
24   transform[0][0] = block_size;
25   Rect<1> extentbig(0, block_size);
26   Rect<1> extentsmall(0,block_size-1);
27
28   IndexPartition ipbig =  rt->create_partition_by_restriction(ctx, is, cis, transform, extentbig);
29   IndexPartition ipsmall =  rt->create_partition_by_restriction(ctx, is, cis, transform, extentsmall);
30   IndexPartition ipdiff = rt->create_partition_by_difference(ctx,is,ipbig,ipsmall,cis);
31
32   LogicalPartition lpdiff = rt->get_logical_partition(ctx, lr, ipdiff);
33   ArgumentMap arg_map;
34   IndexLauncher sum_launcher(SUM_TASK_ID, colors, TaskArgument(NULL,0), arg_map);
35   sum_launcher.add_region_requirement(RegionRequirement(lpdiff, 0, READ_ONLY, EXCLUSIVE, lr));
36   sum_launcher.region_requirements[0].add_field(FIELD_A);
37   rt->execute_index_space(ctx, sum_launcher);
38 }
```

Figure 4.5: `Examples/Partitions/sets/sets.cc`

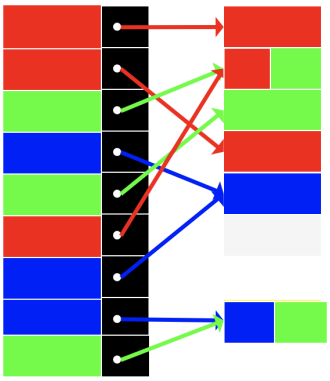

Figure 4.6: An image coloring.

points to the ith element of `lr_dst`.

The example creates an equal partition of `lr_src` on lines 29-30. The `create_partition_by_image` "transfers" the partition of `lr_src` to the index space `is`: if we think of the pointer field as a function from the source region to the destination index space and visualize the source partition as a coloring of the elements, then each pointer copies the color of its source element to the element of the destination. An example (unrelated to Figure 4.7) of taking the image of a pointer field under a partitioning of the source region is depicted in Figure 4.6. In this abstract example, the coloring of the elements on the region on the left is copied via the pointer field to the elements of the index space or region on the right. Because an element in the destination may have multiple pointers to it from the source, elements of the destination may have multiple colors, illustratedby the elements with two colors in Figure 4.6. Thus, in general, the partition computed by `create_partition_by_image` may be aliased. It may also be incomplete, as some elements of the destination region may have no pointers to them at all. Because the pointer relationship is 1-1 between the source and destination regions in the program in Figure 4.7, the partition of the destination in this case is both disjoint and complete.

The `create_partition_by_image` call on line 38 takes the current context, the index space to be partitioned, the source region partition, the source region, the identity of the pointer field in the source region, and the color space of the partition to be computed. The result is an `IndexPartition` of the destination index space.

The rest of the program (lines 41-45) sums the value field of the destination partition's subregions (which as in other examples has the same value 1 for

every element). Since the 1-1 pointer relationship copies the coloring exactly from source to destination and the source was an equal partition, the sums printed for each subregion are the same.

4.6 Pre-Image Partitions

Where an image partition transfers a partitioning from a pointer's source region to its destination, a pre-image transfers a partitioning of the destination to the source. Viewing the pointer field as a function from source to destination, this operation is a preimage computation of that function.

The program in Figure 4.9 gives an example. As before there is a source region and a destination index space. (Note that the source must be a region because we need a pointer field, and only regions have fields.) The source region has both the pointer field and the value field (lines 8-11), because we will be summing the subregions computed by the preimage operation, which are subregions of the source region. We do not need any fields in the destination region in this simple example, so its field space is empty (line 13).

Line 32 creates an equal partition `ip_dst` of the index space `is` of the destination region. The call to `create_partition_by_preimage` on line 35 transfers this partitioning backwards across the pointer field `FIELD_PTR` to the index space of the source region. The call takes the current context, the partition of the destination index space, the source region, the source region's parent (or the source region itself if it has no parent, as in this example), the name of the pointer field in the source region, and the color space for the computed partition.

Figure 4.8 gives an abstract example of a preimage computation. Here the pre-existing partition is on the right, and the color of each element is copied backwards through the pointer field to derive a coloring (a partitioning) of the source index space. Note that because a pointer has a single source, a preimage is always guaranteed to be a disjoint partitioning of the source (but the partition may be incomplete—not every element of the source necessarily has a pointer into the destination).

Returning to the program in Figure 4.9, lines 38-43 sum the elements of the value field of each of the subregions of the source. Again in this example the ith element of the source points to the ith element of the destination (lines 24-27 and 45-58), so the preimage operation replicates the equal partition of the destination in the source index space. Since the value field is initialized to 1 (lines 21-22), the sums count the number of elements of each source

```
1  void top_level_task(const Task *task,
2                      const std::vector<PhysicalRegion> &rgns,
3                      Context ctx,
4                      Runtime *rt)
5  {
6    Rect<1> rec(Point<1>(0),Point<1>(99));
7    IndexSpace is = rt->create_index_space(ctx,rec);
8    FieldSpace fs1 = rt->create_field_space(ctx);
9    FieldAllocator field_allocator1 = rt->create_field_allocator(ctx,fs1);
10   FieldID fidptr = field_allocator1.allocate_field(sizeof(Point<1>), FIELD_PTR);
11
12   FieldSpace fs2 = rt->create_field_space(ctx);
13   FieldAllocator field_allocator2 = rt->create_field_allocator(ctx,fs2);
14   FieldID fidv = field_allocator2.allocate_field(sizeof(int), FIELD_VAL);
15
16   assert(fidptr == FIELD_PTR);
17   assert(fidv == FIELD_VAL);
18
19   LogicalRegion lr_src = rt->create_logical_region(ctx,is,fs1);
20   LogicalRegion lr_dst = rt->create_logical_region(ctx,is,fs2);
21
22   int init = 1;
23   rt->fill_field(ctx,lr_dst,lr_dst,FIELD_VAL,&init,sizeof(init));
24
25   int num_subregions = 4;
26   Rect<1> colors(0,num_subregions-1);
27   IndexSpace cis = rt->create_index_space(ctx,colors);
28
29   IndexPartition ip_src = rt->create_equal_partition(ctx, is ,cis);
30   LogicalPartition lp_src = rt->get_logical_partition(ctx, lr_src, ip_src);
31
32   ArgumentMap arg_map;
33   IndexLauncher ptr_launcher(PTR_TASK_ID, colors, TaskArgument(NULL,0), arg_map);
34   ptr_launcher.add_region_requirement(RegionRequirement(lp_src, 0, WRITE_DISCARD, EXCLUSIVE, lr_src));
35   ptr_launcher.region_requirements[0].add_field(FIELD_PTR);
36   rt->execute_index_space(ctx, ptr_launcher);
37
38   IndexPartition ip_dst = rt->create_partition_by_image(ctx, is, lp_src, lr_src, FIELD_PTR, cis);
39   LogicalPartition lp_dst = rt->get_logical_partition(ctx, lr_dst, ip_dst);
40
41   IndexLauncher sum_launcher(SUM_TASK_ID, colors, TaskArgument(NULL,0), arg_map);
42   sum_launcher.add_region_requirement(RegionRequirement(lp_dst, 0, READ_ONLY, EXCLUSIVE, lr_dst));
43   sum_launcher.region_requirements[0].add_field(FIELD_VAL);
44   rt->execute_index_space(ctx, sum_launcher);
45 }
46
47 void ptr_task(const Task *task,
48               const std::vector<PhysicalRegion> &rgns,
49               Context ctx, Runtime *rt)
50 {
51   const FieldAccessor<WRITE_DISCARD,Point<1>,1> fa_ptr(rgns[0], FIELD_PTR);
52   Rect<1> d = rt->get_index_space_domain(ctx, task->regions[0].region.get_index_space());
53
54   int x = 0;
55   for (PointInRectIterator<1> itr(d); itr(); itr++)
56     {
57       fa_ptr[*itr] =  (Point<1>) x;
58       x = x + 1;
59     }
60 }
```

Figure 4.7: `Examples/Partitions/image/image.cc`

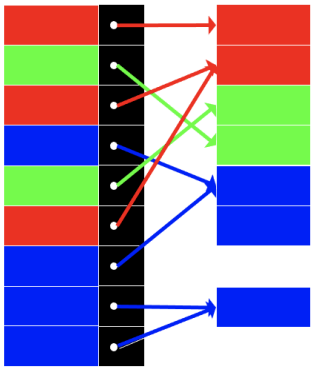

Figure 4.8: An preimage coloring.

subregion, which are all equal.

```
1  void top_level_task(const Task *task,
2                      const std::vector<PhysicalRegion> &rgns,
3                      Context ctx,
4                      Runtime *rt)
5  {
6    Rect<1> rec(Point<1>(0),Point<1>(99));
7    IndexSpace is = rt->create_index_space(ctx,rec);
8    FieldSpace fs1 = rt->create_field_space(ctx);
9    FieldAllocator field_allocator1 = rt->create_field_allocator(ctx,fs1);
10   FieldID fidptr = field_allocator1.allocate_field(sizeof(Point<1>), FIELD_PTR);
11   FieldID fidv = field_allocator1.allocate_field(sizeof(int), FIELD_VAL);
12
13   FieldSpace fs2 = rt->create_field_space(ctx);
14
15   assert(fidptr == FIELD_PTR);
16   assert(fidv == FIELD_VAL);
17
18   LogicalRegion lr_src = rt->create_logical_region(ctx,is,fs1);
19   LogicalRegion lr_dst = rt->create_logical_region(ctx,is,fs2);
20
21   int init = 1;
22   rt->fill_field(ctx,lr_src,lr_src,FIELD_VAL,&init,sizeof(init));
23
24   TaskLauncher ptr_launcher(PTR_TASK_ID, TaskArgument(NULL,0));
25   ptr_launcher.add_region_requirement(RegionRequirement(lr_src, WRITE_DISCARD, EXCLUSIVE, lr_src));
26   ptr_launcher.add_field(0,FIELD_PTR);
27   rt->execute_task(ctx, ptr_launcher);
28
29   int num_subregions = 4;
30   Rect<1> colors(0,num_subregions-1);
31   IndexSpace cis = rt->create_index_space(ctx,colors);
32   IndexPartition ip_dst = rt->create_equal_partition(ctx, is, cis);
33   LogicalPartition lp_dst = rt->get_logical_partition(ctx, lr_dst, ip_dst);
34
35   IndexPartition ip_src = rt->create_partition_by_preimage(ctx, ip_dst, lr_src, lr_src, FIELD_PTR, cis);
36   LogicalPartition lp_src = rt->get_logical_partition(ctx, lr_src, ip_src);
37
38   ArgumentMap arg_map;
39   IndexLauncher sum_launcher(SUM_TASK_ID, colors, TaskArgument(NULL,0), arg_map);
40   sum_launcher.add_region_requirement(RegionRequirement(lp_src, 0, READ_ONLY, EXCLUSIVE, lr_src));
41   sum_launcher.region_requirements[0].add_field(FIELD_VAL);
42   rt->execute_index_space(ctx, sum_launcher);
43 }
44
45 void ptr_task(const Task *task,
46               const std::vector<PhysicalRegion> &rgns,
47               Context ctx, Runtime *rt)
48 {
49   const FieldAccessor<WRITE_DISCARD,Point<1>,1> fa_ptr(rgns[0], FIELD_PTR);
50   Rect<1> d = rt->get_index_space_domain(ctx, task->regions[0].region.get_index_space());
51
52   int x = 0;
53   for (PointInRectIterator<1> itr(d); itr(); itr++)
54     {
55       fa_ptr[*itr] =  (Point<1>) x;
56       x = x + 1;
57     }
58 }
```

Figure 4.9: `Examples/Partitions/pre_image/preimage.cc`

Chapter 5

Control Replication

In Legion and most other tasking models, a root task is responsible for launching subtasks that will fill a parallel machine. We have already seen that index launches (recall Section 2.4) can be used to compactly express the launch of a set of n tasks, where n is usually scaled with the size of the machine being used.

A problem arises when the number of child tasks to be launched by a parent task is large: The amount of work the parent task needs to do to launch all of the child tasks can itself become a serial bottleneck in the program. In practice, it turns out that this effect does not require especially large numbers of tasks to become noticeable. For most applications, a parent task repeatedly launching more than 16 or 32 tasks at a time has a measurable impact on scalability.

Control replication is Legion's solution to this problem and a key feature of the programming model. Almost any application with tasks that launch a large number of subtasks will perform significantly better with control replication.

The idea behind control replication is simple: Instead of having one copy of the parent task launching all of the child tasks, multiple copies of the parent task are executed in parallel, each of which launches a subset of the child tasks. For example, if a parent tasks launches subtasks using index launches, then control-replicating the parent tasks n times will result in all copies of the parent task launching $1/n$th of the tasks in each index launch (using the default mapper, see below). A common pattern is to replicate a task once per Legion process in the computation, with each replicated instance launching the subtasks destined to execute locally on the resources managed by that Legion runtime.

```
1  void top_level_task(const Task *task,
2                      const std::vector<PhysicalRegion> &rgns,
3                      Context ctx,
4                      Runtime *rt)
5  {
6    Rect<1> rec(Point<1>(0),Point<1>(99));
7    IndexSpace is = rt->create_index_space(ctx,rec);
8    FieldSpace fs = rt->create_field_space(ctx);
9    FieldAllocator field_allocator = rt->create_field_allocator(ctx,fs);
10   FieldID fida = field_allocator.allocate_field(sizeof(int), FIELD_A);
11   assert(fida == FIELD_A);
12
13
14   LogicalRegion lr = rt->create_logical_region(ctx,is,fs);
15   int num_subregions = 4;
16   Rect<1> colors(0,num_subregions - 1);
17   IndexSpace color_is = rt->create_index_space(ctx, colors);
18   IndexPartition ip = rt->create_equal_partition(ctx, is, color_is);
19   LogicalPartition lp = rt->get_logical_partition(ctx, lr, ip);
20
21   int init = 1;
22   rt->fill_field(ctx,lr,lr,fida,&init,sizeof(init));
23
24   ArgumentMap arg_map;
25   IndexLauncher sum_launcher(SUM_TASK_ID, colors, TaskArgument(NULL,0), arg_map);
26   sum_launcher.add_region_requirement(RegionRequirement(lp, 0, READ_ONLY, EXCLUSIVE, lr));
27   sum_launcher.region_requirements[0].add_field(FIELD_A);
28   rt->execute_index_space(ctx, sum_launcher);
29 }
30 int main(int argc, char **argv)
31 {
32   Runtime::set_top_level_task_id(TOP_LEVEL_TASK_ID);
33   {
34     TaskVariantRegistrar registrar(TOP_LEVEL_TASK_ID, "top_level_task");
35     registrar.add_constraint(ProcessorConstraint(Processor::LOC_PROC));
36     registrar.set_replicable();   // The only change from Examples/Partitions/equal/equal.cc
37     Runtime::preregister_task_variant<top_level_task>(registrar);
38   }
39   {
40     TaskVariantRegistrar registrar(SUM_TASK_ID, "sum_task");
41     registrar.add_constraint(ProcessorConstraint(Processor::LOC_PROC));
42     Runtime::preregister_task_variant<sum_task>(registrar);
43   }
44   return Runtime::start(argc, argv);
45 }
```

Figure 5.1: `Examples/ControlReplication/sum/cp.cc`

By far the most common case is that the top-level task is control-replicated and all other tasks are not, but sometimes overall performance can be improved by control replicating other tasks in the task hiearchy. It is also legal to nest control-replicated tasks: control-replicated tasks can be launched from within other control-replicated tasks.

At the program level, the use of control replication is straightforward. Typically, the only thing that needs to be done is to notify the system that a task is *replicable*, as shown in Figure 5.1, line 36. In this example, the top-level task is marked as replicable, while the `sum` task (not shown) is not. As we discuss below, not every task is replicable. Even if a task t is potentially replicable, if it does not launch enough subtasks to make control replication worthwhile then it will be better overall not to replicate t.

If a task is marked as replicable, then the decisions of whether to replicate the task or not and, if the task is replicated, how to *shard* the work of analyzing the subtasks across all the instances of the replicated task are made by the mapper. The default mapper handles the case of a top-level task that is control-replicated and subtasks are sharded evenly and to the instance of the Legion runtime where they will execute. Anything other than this simple, but very common, case will likely require writing some custom mapping logic. Because the dynamic safety checks for control replication do not currently cover every way that a task might fail to be replicable, it is possible, but unlikely, for a task that is incorrectly marked as replicable to pass the safety checks.

Chapter 6

Coherence

Every task has associated *privileges* and *coherence modes* for each region argument. Privileges, which declare what a task may do with its region argument (such as reading it, writing it, or performing reductions to it), are discussed in Section 3.1. A coherence mode declares what other tasks may do concurrently with a region. So far we have focused on `Exclusive` coherence, which is the default if no other coherence mode is specified. Exclusive coherence means that it must appear to a task that it has exclusive access to a region argument—all updates from tasks that precede the task in sequential execution order must be included in the region when the task starts executing, and no updates from tasks that come after the task in sequential execution order can be visible while the task is running.

More precisely, the coherence mode of region argument r for a task t is a declaration of what updates to r by t's sibling tasks can be visible to t. The scope of a coherence declaration for task t is always the sibling tasks of t. Each region argument may have its own coherence declaration—not all regions need have the same coherence mode.

Besides `Exclusive` coherence, there are three other coherence modes: `Atomic`, `Simultaneous`, and `Relaxed`.

6.1 Atomic

An example using `Atomic` coherence is given in Figure 6.1. The loop on lines 19-24 launches a number of individual `inc` tasks, each of which increments all the elements of its region argument by one. On line 21, we see the task launcher declares the (single) region argument to the `inc` task has `Atomic` coherence. Atomic coherence means that the `inc` task only requires that

```
1  void top_level_task(const Task *task,
2                      const std::vector<PhysicalRegion> &rgns,
3                      Context ctx,
4                      Runtime *rt)
5  {
6    Rect<1> rec(Point<1>(0),Point<1>(99));
7    IndexSpace is = rt->create_index_space(ctx,rec);
8    FieldSpace fs = rt->create_field_space(ctx);
9    FieldAllocator field_allocator = rt->create_field_allocator(ctx,fs);
10   FieldID fida = field_allocator.allocate_field(sizeof(int), FIELD_A);
11   assert(fida == FIELD_A);
12
13
14   LogicalRegion lr = rt->create_logical_region(ctx,is,fs);
15
16   int init = 1;
17   rt->fill_field(ctx,lr,lr,fida,&init,sizeof(init));
18
19   for (int i = 0; i < 10; i++) {
20     TaskLauncher inc_launcher(INC_TASK_ID, TaskArgument(&i,sizeof(int)));
21     inc_launcher.add_region_requirement(RegionRequirement(lr, READ_WRITE, ATOMIC, lr));
22     inc_launcher.add_field(0,FIELD_A);
23     rt->execute_task(ctx, inc_launcher);
24   }
25
26   TaskLauncher sum_launcher(SUM_TASK_ID, TaskArgument(NULL,0));
27   sum_launcher.add_region_requirement(RegionRequirement(lr, READ_ONLY, EXCLUSIVE, lr));
28   sum_launcher.add_field(0,FIELD_A);
29   rt->execute_task(ctx, sum_launcher);
30 }
31
32 void inc_task(const Task *task,
33                      const std::vector<PhysicalRegion> &rgns,
34                      Context ctx, Runtime *rt)
35 {
36   int id = *((int *) task->args);
37   const FieldAccessor<READ_WRITE,int,1> fa_a(rgns[0], FIELD_A);
38   Rect<1> d = rt->get_index_space_domain(ctx,task->regions[0].region.get_index_space());
39   printf("Task %d\n",id);
40   for (PointInRectIterator<1> itr(d); itr(); itr++)
41     {
42       fa_a[*itr] = fa_a[*itr] + 1;
43     }
```

Figure 6.1: `Examples/Coherence/atomic/atomic.cc`

sibling tasks execute atomically with respect to the region `lr`—as far as one `inc` task is concerned, it is fine for other tasks t that modify `lr` to appear to execute either before or after the `inc` task, provided that *all* of t's updates to `lr` come either before or after the `inc` task executes. Since the loop launches 10 `inc` tasks all with atomic coherence on region `lr`, these tasks are free to run in any sequential order, but not in parallel (since they all write `lr` and must execute atomically). The `sum` task (lines 26-29) is also a sibling task of the `inc` tasks, but the `sum` tasks requires exclusive coherence for region `lr`. Thus, `sum` must run after all of the `inc` tasks have completed and all of their updates have been performed.

6.2 Simultaneous

Simultaneous coherence provides the equivalent of shared memory semantics for a region: A task t that requests simultaneous coherence on a region r is permitting other tasks to update r and have those updates be visible while t is executing. Note that simultaneous coherence does not require that multiple tasks with simultaneous coherence run at the same time and are able to see each others updates, but that behavior is certainly allowed.

By definition simultaneous coherence permits race conditions—the program is explicitly requesting that race conditions be permitted on the region. Thus, another way to understand simultaneous coherence is that it notifies the runtime system that the application itself will take care of whatever synchronization is needed to guarantee that the tasks accessing the region produce correct results, as the runtime will not necessarily enforce any ordering on the accesses of two or more tasks to the region.

Like all explicit parallel programming, a program that uses simultaneous coherence is more difficult to reason about than a program that does not. There are legitimate reasons to use simultaneous coherence, but they are rare. We will cover two in this manual. First, we will look at an example where what we truly want is to exploit shared memory. While this may in some circumstances improve performance, requiring shared memory is also less portable. This example is most likely to be useful when tasks are extremely fine-grain and there needs to be concurrency between tasks to fully exploit the hardware. We have not found many such use cases in practice.

The second use case is interoperating with external programs or other resources, where simultaneous coherence is the only safe model of data shared with an external process. Because this case is common and important, Legion encapsulates the most useful interoperation patterns in higher level constructs

```
1  DistributeChargeTask::DistributeChargeTask(LogicalPartition lp_pvt_wires,
2                                             LogicalPartition lp_pvt_nodes,
3                                             LogicalPartition lp_shr_nodes,
4                                             LogicalPartition lp_ghost_nodes,
5                                             LogicalRegion lr_all_wires,
6                                             LogicalRegion lr_all_nodes,
7                                             const Domain &launch_domain,
8                                             const ArgumentMap &arg_map)
9   : IndexLauncher(DistributeChargeTask::TASK_ID, launch_domain, TaskArgument(), arg_map,
10                  Predicate::TRUE_PRED, false/*must*/, DistributeChargeTask::MAPPER_ID)
11 {
12   RegionRequirement rr_wires(lp_pvt_wires, 0/*identity*/,
13                              READ_ONLY, EXCLUSIVE, lr_all_wires);
14   rr_wires.add_field(FID_IN_PTR);
15   rr_wires.add_field(FID_OUT_PTR);
16   rr_wires.add_field(FID_IN_LOC);
17   rr_wires.add_field(FID_OUT_LOC);
18   rr_wires.add_field(FID_CURRENT);
19   rr_wires.add_field(FID_CURRENT+WIRE_SEGMENTS-1);
20   add_region_requirement(rr_wires);
21
22   RegionRequirement rr_private(lp_pvt_nodes, 0/*identity*/,
23                                READ_WRITE, EXCLUSIVE, lr_all_nodes);
24   rr_private.add_field(FID_CHARGE);
25   add_region_requirement(rr_private);
26
27   RegionRequirement rr_shared(lp_shr_nodes, 0/*identity*/,
28                               REDUCE_ID, SIMULTANEOUS, lr_all_nodes);
29   rr_shared.add_field(FID_CHARGE);
30   add_region_requirement(rr_shared);
31
32   RegionRequirement rr_ghost(lp_ghost_nodes, 0/*identity*/,
33                              REDUCE_ID, SIMULTANEOUS, lr_all_nodes);
34   rr_ghost.add_field(FID_CHARGE);
35   add_region_requirement(rr_ghost);
36 }
```

Figure 6.2: From Legion/examples/circuit/circuit_cpu.cc

described in Chapter 8 and we strongly recommend using those abstractions if possible. These higher level abstractions are built on simultaneous coherence and the other concepts introduced in this section.

To provide shared-memory semantics, a region for which simultaneous coherence is requested by a task can usually have only one physical instance, which is called the *copy restriction*. That is, there can be only one instance of the data—no copies can be made—and all tasks using the region share it. As we discuss below, Legion provides a mechanism for explicitly relaxing the copy restriction and allowing copies of a region to be made, but the default behavior is a single physical instance.

Figure 6.2 gives an example of the use of simultaneous coherence from the Legion repository that is intended specifically to exploit shared memory. This excerpt comes from a much larger program that simulates the behavior of an arbitrary electrical circuit, modeled as a graph of wires and nodes where where wires connect. Here we see that the `DistributeChargeTask` uses simultaneous coherence on two regions `rr_shared` and `rr_ghost`. The electrical circuit is divided up into pieces and the simulation is carried out in parallel for each piece of the circuit. The regions `rr_shared` and `rr_ghost` represent regions that may alias pieces of the graph that overlap with other pieces. The `DistributeCharge` task is performing reductions to these two regions (the `REDUCE_ID` privilege is the identity of a reduction operator registered with the runtime system); all the tasks from different pieces may be performing reductions to these aliased regions in parallel. Thus, the implementation of the task body of `DistributeCharge` uses atomic updates to guarantee that no reductions are lost (not shown).

In contrast, the region `rr_private` is a set of nodes private to a particular piece of the graph (not shared with any other piece); the task uses exclusive access for this region since no other task will access it.

Because of the copy restriction, this implementation strategy, using simultaneous coherence for multiple tasks that may reduce to the same elements of some regions, can only be used for shared-memory CPU-based systems. Trying to use this code on a distributed machine, or on a machine with GPUs with their own framebuffer memories, will result in errors from the runtime system when the program tries to copy restricted regions to other nodes or GPU memory.

Another use of simultaneous coherence is shown in Figure 6.3. In this example there are two tasks, a producer and a consumer, that alternate access to a region that has simultaneous coherence. A producer task fills a region with some values (for illustration the ith time the producer is called it writes i to every element of the region), and a consumer task reads the values written by the previous producer and resets the values to 0. Because the region has simultaneous coherence, the producer and consumer tasks synchronize: a consumer task waits until the region is filled by a producer task, and a producer task waits until the previous consumer task has emptied the region.

Phase barriers are a lightweight synchronization abstraction designed for deferred execution. The name "barrier" is not meant to evoke MPI barriers, and attempting to understand phase barriers in terms of MPI barriers will lead to confusion. A phase barrier has four important characteristics:

- An operation can *wait* on a phase barrier; the waiting operation will not begin execution until the phase barrier is triggered.
- An operation can *arrive* at a phase barrier. Every phase barrier has an *arrival count*, which is the number of arrivers required to trigger the barrier. Once triggered, all of the waiters (and any future waiters on the same barrier) are notified. By default, the arrival count of a phase barrier is 1.
- A phase barrier has a *generation*. When an operation waits on or arrives at a barrier, it waits on or arrives at a specific generation. A phase barrier can be *advanced* to a new generation, and different operations can wait on or arrive at that generation. Generations support deferred execution, as illustrated below. A phase barrier has 2^{32} generations—so a very large, but not unlimited, number.
- When a phase barrier triggers, it signals the waiters on the next generation. For a phase barrier with arrival count 1, an arrival at generation n causes waiters on generation $n+1$ to be notified. (Realm, the abstraction level below Legion, has its own more primitive phase barrier type with slightly different semantics. Here we are describing the Legion-level `PhaseBarrier` type.)

The final concepts we need to explain idiomatic use of simultaneous coherence in Legion are *acquire* and *release* of regions. Acquiring a region with simultaneous coherence tells the runtime system that it is safe to make copies of the region's sole physical instance: an acquire means that the task is promising it will be the only user of the data until the region is released. It is up to the application to use acquire and release correctly; there is no checking done by the runtime system. An acquire removes the copy restriction on a region, which allows an instance of the region, and therefore the task itself, to be mapped anywhere in the system—for example on a different node or onto an accelerator such as a GPU. A release copies all updates to the region back to the original physical instance (i.e., if flushes all the updates) and restores the copy restriction. In a typical use of simultaneous coherence, phase barriers are used to ensure that the acquires and releases of the region are properly synchronized.

In Figure 6.3, the structure of the loop from lines 18-62 is alternating producer and consumer tasks. There are two phase barriers, called `even` and `odd` (lines 15-16). The odd barrier is used by a producer task to wait for the preceding consumer task to finish. The even barrier is used by a consumer task to wait for the preceding producer to finish.

```
1  void top_level_task(const Task *task,
2                      const std::vector<PhysicalRegion> &rgns,
3                      Context ctx,
4                      Runtime *rt)
5  {
6    Rect<1> rec(Point<1>(0),Point<1>(99));
7    IndexSpace is = rt->create_index_space(ctx,rec);
8    FieldSpace fs = rt->create_field_space(ctx);
9    FieldAllocator field_allocator = rt->create_field_allocator(ctx,fs);
10   FieldID fida = field_allocator.allocate_field(sizeof(int), FIELD_A);
11   assert(fida == FIELD_A);
12
13   LogicalRegion lr = rt->create_logical_region(ctx,is,fs);
14
15   PhaseBarrier odd = rt->create_phase_barrier(ctx,1);
16   PhaseBarrier even = rt->create_phase_barrier(ctx,1);
17
18   for (int i = 0; i < 10; i++) {
19     PhaseBarrier odd_next = rt->advance_phase_barrier(ctx,odd);
20     PhaseBarrier even_next = rt->advance_phase_barrier(ctx,even);
21
22     /* Producer task */
23     AcquireLauncher al_producer(lr,lr);
24     al_producer.add_field(FIELD_A);
25     if (i > 0)
26       al_producer.add_wait_barrier(odd_next);
27     rt->issue_acquire(ctx,al_producer);
28
29     TaskLauncher producer_launcher(PRODUCER_TASK_ID, TaskArgument(&i,sizeof(int)));
30     producer_launcher.add_region_requirement(RegionRequirement(lr, WRITE_DISCARD, SIMULTANEOUS, lr));
31     producer_launcher.add_field(0,FIELD_A);
32     rt->execute_task(ctx, producer_launcher);
33
34     ReleaseLauncher rl_producer(lr,lr);
35     rl_producer.add_field(FIELD_A);
36     rl_producer.add_arrival_barrier(even);
37     rt->issue_release(ctx,rl_producer);
38
39     /* Consumer task */
40     AcquireLauncher al_consumer(lr,lr);
41     al_consumer.add_field(FIELD_A);
42     al_consumer.add_wait_barrier(even_next);
43     rt->issue_acquire(ctx,al_consumer);
44
45     TaskLauncher consumer_launcher(CONSUMER_TASK_ID, TaskArgument(NULL,0));
46     consumer_launcher.add_region_requirement(RegionRequirement(lr, READ_WRITE, SIMULTANEOUS, lr));
47     consumer_launcher.add_field(0,FIELD_A);
48     rt->execute_task(ctx, consumer_launcher);
49
50     ReleaseLauncher rl_consumer(lr,lr);
51     rl_consumer.add_field(FIELD_A);
52     rl_consumer.add_arrival_barrier(odd);
53     rt->issue_release(ctx,rl_consumer);
54
55     odd = odd_next;
56     even = even_next;
57     printf("Iteration %d of top level task.\n",i);
58   }
59   printf("Deallocating phase barriers.\n");
60   rt->destroy_phase_barrier(ctx,odd);
61   rt->destroy_phase_barrier(ctx,even);
62 }
```

Figure 6.3: `Examples/Coherene/simultaneous/sim.cc`

At the top of the loop the phase barriers are advanced to the next generation (lines 19-20). Because triggering a phase barrier in a generation signals waiters on the next generation, both the current and next generations are used by the tasks in the current iteration of the loop. Executing the producer task consists of three phases: first an *acquire launcher* is used and then executed to acquire coherence on the region `lr`. An acquire launcher acquires a set of fields of a region and can optionally wait on or arrive at phase barriers; in this case the launcher waits on the next generation of the `odd` phase barrier except in the first iteration of the loop (lines 23-27). We then construct a task launcher and execute the producer task (lines 29-32). Finally, a release launcher is constructed that releases coherence on `lr` and arrives at the current generation of the `even` phase barrier (lines 34-37).

The three phases for the consumer task (acquiring coherence on `lr`, executing the consumer task, and releasing coherence on `lr`) are similar, with the roles of the `odd` and `even` phase barriers reversed (lines 40-53).

When this program is executed, note that the ten iterations of the main loop likely complete before any of the operations in the loop body execute (examine the order of `printf`'s from the top-level loop and the producer and consumer tasks). Thus, the entire chain of dependencies between the different producers, consumers, acquires, and releases may be constructed before any of that work is done. Allowing the runtime to defer large amounts of work depends on having unique names for the synchronization operations used in different iterations of the loop with different tasks, which is the purpose of the generation property of phase barriers.

Finally, note that if we simply replaced simultaneous coherence by exclusive coherence the example could be dramatically simplified to just the two task launches in the loop body, removing all operations to acquire and release and operate on phase barriers. In a self-contained Legion program there is usually little reason to add the extra complexity of simultaneous coherence, except in the case of data shared between Legion and an external process where such semantics are really required.

6.2.1 Simple Cases of Simultaneous Coherence

The example in Figure 6.3 is actually a bit too simple to require the use of phase barriers and acquire/release. The example in `Examples/Coherence/simultaneous/simultaneous_simple` gives another version of the same program with the same behavior using simultaneous coherence with the phase barriers and acquire/release operations stripped out. The reason this example works is that when there are only sibling tasks

that use simultaneous coherence, the Legion runtime is still able to deliver correct semantics without explicit synchronization: If the sibling tasks use the same instance of the data and run in parallel, the desired semantics is achieved, but if they use different instances of the region then the runtime serializes the tasks and ensures the results of the first task are visible to the second task by copying the final contents of the instance used by the first task to the instance used by the second task. In this simple situation, the Legion runtime detects automatically that the copy restriction is not needed because there is always a single instance in use. The need for application synchronization arises when tasks have no well-defined default execution order when using simultaneous coherence, such as both a parent task and its subtasks using simultaneous coherence on the same region or a task sharing a region with an external process—in these cases the runtime enforces the copy restriction.

Thus, while there are simple cases where synchronization is unnecessary even in the presence of simultaneous coherence, in general simultaneous coherence does require explicit application synchronization, and the use of phase barriers and acquire/release is the recommended approach to providing that synchronization.

6.3 Relaxed

The design of Legion includes one other coherence mode, `Relaxed`. Relaxed coherence tells the runtime system that the application will handle all aspects of the correct use of data—there is no checking of any kind and all runtime support is disabled, allowing the application to do whatever it wants with the data, at the cost of the application being entirely responsible for the coherence of the data. We discourage the use of relaxed coherence in application code.

Chapter 7

Mapping

The Legion mapper interface is a key part of the Legion programming system. Through the mapping interface applications can control most decisions that impact application performance. The philosophy is that these choices are better left to applications rather than using hard-wired heuristics in Legion that attempt to "do the right thing" in every situation. The few performance heuristics that are included in Legion are associated with low levels of the system where there is no good way to expose those choices to the application. For everything else applications can set the policies.

This design resulted from our own past experience with systems where built-in performance heuristics did not behave as we desired and there was no recourse to override those decisions. While Legion does allows experts to squeeze every last bit of performance from a system, it is important to realize that doing so potentially requires understanding and setting a wide variety of parameters exposed in the mapping interface. 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 complex, distributed and heterogeneous systems.

To help users write initial versions of their applications without needing to concern themselves with tuning the performance knobs exposed by the mapper interface, Legion provides a *default mapper*. The default mapper implements the Legion mapper API (like any other mapper) and provides a number of heuristics that can provide reasonably performant, or at least correct, initial settings. A good way to think about the default mapper is that it is the version of Legion with built-in heuristics that allows casual users to write Legion applications and allows experts to start quickly on a new application. It is, however, unreasonable to expect the default mapper

to provide excellent performance, and in particular assuming that the performance of an application using the default mapper is even an approximation of the performance that could be achieved with a custom mapper is a mistake.

We will use several examples from the default mapper to illustrate how mappers are constructed. We will also describe where possible the heuristics that the default mapper employs to achieve reasonable performance. Because the default mapper uses generic heuristics with no specific knowledge of the apllication, it is almost certain to make poor decisions at least some of the time. Performance benchmarking using only the default mapper is strongly discouraged, while using custom application-specific mappers is encouraged.

It is likely that the moment when you are dissatisfied with the heuristics in the default mapper will come sooner rather than later. At that point the information in this chapter will be necessary for you to write your own custom mapper. In practice, our experience has been that in many cases all that is necessary is to replace a small number of policies in the default mapper that are a poor fit for the application.

7.1 Mapper Organization

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

A callback is just a function pointer—when the runtime system calls a mapper function, it is said to have "invoked the callback". Callbacks are a commomly-used mechanism in software systems for parameterizing some specific functionality; in our case mappers parameterize the performance heuristics of the Legion runtime system. There are a few general things to keep in mind about mappers and callbacks:

- The runtime may invoke callbacks in an unpredictable order. While multiple callbacks associated with a single instance of a Legion object, such as a task, will happen in a specific order for that task, other callbacks for other operations may be interleaved.
- Depending on the synchronization model selected (see Section 7.1.2), mappers may have a degree of concurrency between mapper callbacks.
- Since mappers are C++ objects, they can have arbitrary internal state. For example, it may be useful to maintain performance or load-

balancing statistics that inform mapping decisions. However, state updates done by a mapper must take into account the unpredictable order in which callbacks are invoked as well any issues of concurrent access to mapper data structures.

7.1.1 Mapper Registration

After the Legion runtime is created, but before the application begins, mapper objects can be registered with the runtime. Figure 7.1 gives a small example registering a custom mapper.

To register `CustomMapper` objects, the application adds the mapper callback function by invoking the `Runtime::add_registration_callback` 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, perhaps for different libraries). All callback functions must be added prior to the invocation of the `Runtime::start` method. We recommend that applications include the registration method as a static method on the mapper class (as in Figure 7.1) so that it is closely coupled to the custom mapper itself.

Before invoking any of the registration callback functions, the runtime creates an instance of the default mapper for each processor of the system. The runtime then invokes the callback functions in the order 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 per process and therefore one invocation of each callback per process. The set of processors passed into each registration callback function will be the set of application processors that are local to the process[1], thereby providing a registration callback function with the necessary context to know which processors it will create new custom mappers for. 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 step can be done through one of two runtime mapper calls. The mapper can replace the default mappers (always registered with `MapperID` 0) by

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

```
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 7.1: `Examples/Mapping/registration/registration.cc`

calling `Runtime::replace_default_mapper`, which 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 is associated with all processors on the local node. The choice between whether one mapper object should handle a single application processor's mapping decisions or one mapper object handling mapping decisions for all application processors on a node is mapper-specific. 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 depend on the mapper synchronization model (see Section 7.1.2).

Note that the mapper calls require a pointer to the `MapperRuntime`, such as on lines 27 and 49 of Figure 7.1. The mapper runtime provides the interface for mapper calls to call back into the runtime to acquire access to different physical resources. We will see examples of the use of the mapper runtime throughout this chapter.

7.1.2 Synchronization Model

Within an instance of the Legion runtime there are often several threads performing the analysis necessary to advance the execution of an 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 the same mapper object concurrently. For both productivity and correctness reasons, we do not want users to be responsible for making their mappers thread-safe. Therefore we allow mappers to specify a *synchronization model* that the runtime follows when concurrent mapper calls are made.

Each mapper object can specify its synchronization model via the `get_mapper_sync_model` mapper call. The runtime invokes 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. Currently three synchronization models are supported:

- *Serialized Non-Reentrant.* Calls to the mapper object are serialized and execute atomically. If the mapper calls out to the runtime and the mapper call is preempted, no other mapper calls can be invoked by the runtime. This synchronization model conforms to the original version of the Legion mapper interface.

- *Serialized Reentrant.* At most one mapper call executes at a time. However, if a mapper call invokes a runtime method that preempts the mapper call, the runtime may execute another mapper call or resume a previously blocked mapper call. 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.* Mapper calls to the same mapper object can proceed concurrently. Users can invoke the `lock_mapper` and `unlock_mapper` calls to perform their own synchronization of the mapper. This synchronization model is particularly useful for mappers that simply return static mapping decisions without changing internal mapper state.

The mapper synchronization offers mappers tradeoffs between mapper complexity and performance. The default mapper uses the serialized reentrant synchronization model as it offers a good trade-off between programmability and performance.

7.1.3 Machine Interface

All mappers are given a `Machine` object to enable introspection of the hardware on which the application is executing. The `Machine` object is defined by Realm, Legion's low-level portability layer (see `realm/machine.h`).

There are two 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. We strongly discourage using these methods as they are not scalable on large architectures with tens to hundreds of thousands of processors or memories.

The recommended, and more efficient and scalable, interface is based on *queries*, which come in two types: `ProcessorQuery` and `MemoryQuery`. Each query is initially given a reference to the machine object. After initialization the query lazily materializes the (entire) set of either processors or memories of the machine. The mapper applies *filters* to the query to reduce the set to processors or memories of interest. These filters can include specializing the query to the local node using `local_address_space`, to one 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` method. Affinity can either be specified as a maximum bandwidth or a minimum latency. Figure 7.2 shows how to create a custom

mapper that uses queries to find the local set of processors with the same processor kind as and the memories with affinities to the local mapper processor. In some cases, these queries are still expensive, so we encourage the creation of mappers that memoize the results of their most commonly invoked queries to avoid duplicated work.

7.2 Mapping Tasks

There are a number of different kinds of operations with mapping callbacks, but the core of the mapping interface, and the parts of mappers that users will most commonly customize, are the callbacks for mapping tasks. When a task is launched it proceeds through a pipeline of mapping callbacks. The most important pipeline stages are:

1. `select_task_options`
2. `select_sharding_functor` (for control-replicated tasks)
3. `slice_task` (for index launches)
4. `select_tasks_to_map` (tasks remain in this stage until selected for mapping)
5. `map_task`

Stages 2 and 3 do not apply to every task, and tasks may repeat stage 4 any number of times depending on the implementation of `select_tasks_to_map`.

After discussing these five components of the task mapping pipeline, we discuss a few other topics relevant to task mapping: allocating new physical instances, postmapping of tasks, virtual mappings, and profiling requests.

7.2.1 Controlling Task Mapping

`select_task_options` is the first callback for mapping tasks. It is invoked for every task t exactly once in the Legion process where t is launched. The signature of the function is:

```
1  virtual void select_task_options(const MapperContext    ctx,
2                                   const Task&             task,
3                                   TaskOptions&            output) = 0;
```

The purpose of the callback is to set fields of the `output` object. All of the fields have defaults, so none are required to be set by the callback implementation. This callback comes first because the fields of `TaskOptions` control the rest of the mapping process for the task.

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

Figure 7.2: `Examples/Mapping/machine/machine.cc`

- For a single task t (not an index launch), `output.initial_proc` is the processor that will execute t; the default is the current processor. The processor does not need to be local—the mapper can select any processor in the machine model for which a variant of t exists. As we will see, t's target processor can be changed by subsequent stages. The reason for choosing a target processor here is that by default t is sent to the Legion process that manages the target processor to be mapped.

- If `output.inline_task` is true (the default is false) the task will be inlined into the parent task and use the parent task's regions. Any needed regions that are unmapped will be remapped. Inline tasks do not go through the rest of the task pipeline, except for the selection of a task variant.

- If `output.stealable` is true then the task can be stolen for load balancing; the default is false. A stealable task t can be stolen by another mapper until t is chosen by `select_tasks_to_map`.

- As mentioned above, by default the `map_task` stage of the mapping pipeline is done by the Legion process that manages the processor where the task will execute. If `output.map_locally` is true (the default is false) then `map_task` will be run by the current mapper. Just to emphasize: `map_locally` controls where a mapping callback for the task is run, not where the task executes. This option is mostly useful for leaf tasks that will be sent to remote processors. In this case, making the mapping decisions locally saves transmitting task metadata to the remote Legion runtime.

- If `valid_instances` is set to false, then the task will not recieve a list of the currently valid instances of regions in subsequent calls to `request_valid_instances`, which saves some runtime overhead. This setting is useful if the task will never use a currently valid region instance, such as when all the regions of an inner task will be virtually mapped.

- Setting `replicate_default` to true turns on replication of single tasks in a control-replication context, which means that the task will be executed separately in every Legion process participating in the replication of the parent task. The default setting is false; in this case only one instance of a single task with a control-replicated parent is executed on one processor and then the results are broadcast to the other Legion processes. Replicating single tasks avoids the broadcast

communication. There are some restrictions on replicated single tasks to ensure the replicated versions all have identical behavior: the tasks cannot have reduction-only privileges on any field, and any fields with write privileges must use a separate instance for each replicated task.

- A task can set the priority of the parent task by modifying `output.parent_priority`, if that is permitted by the mapper. The default is the parent's current priority. When tasks are ready to execute, tasks with higher priority are moved to the front of the ready queue.

7.2.2 Sharding

As the name suggests, `select_sharding_functor` is used to select the functor for *sharding* index task launches in control-replicated contexts. Sharding divides the index space of the task launch into subspaces and associates each shard with a mapper (a processor) where those tasks will be mapped. This callback is invoked once per replicated task index launch in each replicated context:

```
1  virtual void select_sharding_functor(
2      const MapperContext ctx,
3      const Task& task,
4      const SelectShardingFunctorInput& input,
5      SelectShardingFunctorOutput& output) = 0;
6
7  struct SelectShardingFunctorInput {
8    std::vector<Processor> shard_mapping;
9  };
10
11 struct SelectShardingFunctorOutput {
12   ShardingID chosen_functor;
13   bool slice_recurse;
14 };
```

The `shard_mapping` of the input structure provides a vector of the processors where the replicated task is running. The callback must fill in the `chosen_functor` field of the output structure with the id of a sharding function registered with the mapper at startup. The callback can set `slice_recurse` to indicate whether or not the index subspaces chosen by the sharding functor should be recursively sharded on the destination processor. The same sharding functor must be selected in every control-replicated context, which will be checked by the runtime when in debug mode.

7.2.3 Slicing

`slice_task` is called for every index launch. To make index launches efficient, the index space of tasks is first sliced into smaller sets of tasks and each set is sent to a destination mapper as a single object rather than sending multiple individual tasks. The signature of `slice_task` is:

```
virtual void slice_task(const MapperContext ctx,
                        const Task& task,
                        const SliceTaskInput& input,
                        SliceTaskOutput& output) = 0;
```

The `SliceTaskInput` includes the index space of the task launch (field `domain_is)`. The index space of the shard is also included for control-replicated tasks.

```
struct SliceTaskInput {
  IndexSpace domain_is;
  Domain domain;
  IndexSpace sharding_is;
};
```

The implementation of `slice_task` should set the fields of `SliceTaskOutput`:

```
struct SliceTaskOutput {
   std::vector⟨TaskSlice⟩ slices;
   bool verify_correctness; // = false

struct TaskSlice {
   public:
      TaskSlice(void) : domain_is(IndexSpace::NO_SPACE),
        domain(Domain::NO_DOMAIN), proc(Processor::NO_PROC),
        recurse(false), stealable(false) { }
      TaskSlice(const Domain &d, Processor p, bool r, bool s)
        : domain_is(IndexSpace::NO_SPACE), domain(d),
          proc(p), recurse(r), stealable(s) { }
      TaskSlice(IndexSpace is, Processor p, bool r, bool s)
        : domain_is(is), domain(Domain::NO_DOMAIN),
          proc(p), recurse(r), stealable(s) { }
   public:
      IndexSpace domain_is;
      Domain domain;
      Processor proc;
      bool recurse;
      bool stealable;
};
```

The `slices` field is a vector of `TaskSlice`, each of which names a subspace of the index space in `domain_is` and a destination processor `proc` for the slice of tasks. The tasks of the slice can be marked as stealable, and setting the `recurse` field means that `slice_task` will be called again by the mapper associated with the destination processor to allow the slice to be further subdivided before processing individual tasks.

7.2.4 Selecting Tasks to Map

`select_tasks_to_map` gives the mapper control over which tasks should be mapped and which should be sent to other processors—the initial processor assignment set in `select_task_options` can be changed if desired. At this point in the task mapping pipeline all index tasks have been expanded into single tasks, and `select_tasks_to_map` is called by the mapper associated with the destination process, unless `map_locally` was chosen in `select_task_options`. The signature of the callback is:

```
1   virtual void select_tasks_to_map(const MapperContext ctx,
2                                    const SelectMappingInput& input,
3                                    SelectMappingOutput& output) = 0;
4   struct SelectMappingInput {
5     std::list<const Task*> ready_tasks;
6   };
7   struct SelectMappingOutput {
8     std::set<const Task*> map_tasks;
9     std::map<const Task*,Processor> relocate_tasks;
10    MapperEvent deferral_event;
11  };
```

For each task in `ready_tasks` of the `SelectMappingInput` structure, the callback implementation can do one of three things:

- Add the task to `map_tasks`, in which case the task will proceed with mapping on the assigned local processor.
- Add the task to `relocate_tasks` along with a new destination processor to which the task will be transferred.
- Nothing, in which case the task will remain in the `ready_tasks` list for the next call to `select_tasks_to_map`.

If the call does not select at least one task to map or transfer, then it must provide a `MapperEvent` in the field `deferral_event`—another call to `select_tasks_to_map` will not be made until that event is triggered. Of course, it is up to the mapper to guarantee that the event is eventually triggered.

7.2.5 Map_Task

`map_task` is normally the final stage of the task mapping pipeline. This callback selects a processor or processors for the task, maps the task's region arguments, and selects the task variant to use, after which the task will run on one of the selected processors.

```
1  virtual void map_task(
2      const MapperContext ctx,
3      const Task& task,
4      const MapTaskInput& input,
5      MapTaskOutput& output) = 0;
6
7  struct MapTaskInput {
8      std::vector<std::vector<PhysicalInstance> > valid_instances;
9      std::vector<unsigned>                       premapped_regions;
10 };
11
12 struct MapTaskOutput {
13     std::vector<std::vector<PhysicalInstance> > chosen_instances;
14     std::vector<std::vector<PhysicalInstance> > source_instances;
15     std::vector<Memory> output_targets;
16     std::vector<LayoutConstraintSet> output_constraints;
17     std::set<unsigned> untracked_valid_regions;
18     std::vector<Memory>future_locations;
19     std::vector<Processor> target_procs;
20     VariantID chosen_variant; // = 0
21     TaskPriority task_priority;  // = 0
22     TaskPriority profiling_priority;
23     ProfilingRequest task_prof_requests;
24     ProfilingRequest copy_prof_requests;
25     bool postmap_task; // = false
26 };
```

The input structure contains a vector of vector of valid instances: each element of the vector is a vector of instances that hold valid data for the corresponding region requirement. The `premapped_regions` is a vector of indices of region requirements that are already satisfied and do not need to be mapped by the callback.

The callback must fill in the following fields of the `output` structure:

- `target_procs` is a vector of processors. All processors must be on the same node and of the same kind (e.g., all LOCs or all TOCs). The runtime will execute the task on the first processor in the vector that becomes available.

- `chosen_variant` is the `VariantID` of a variant of the task. The chosen variant must be compatible with the chosen processor kind.

- For each region requirement, the `input` structure has a vector of valid instances of the region in the same order as region requirements are added to the task launcher. The entry of the `chosen_instances` field should be filled either with one or more instances from the corresponding entry of `valid_instances`, or the mapper can add newly created instances. A new instance is created by the runtime call `create_physical_instance`, which, in addition to other arguments,

takes a target memory in which the instance should be created and a vector of logical regions—physical instances can be created that hold the data of multiple logical regions. If new physical regions are created, the mapper calls `select_task_sources` to choose existing instances to be the source of data to fill those new instances (see below).

- For any regions that are strictly output regions (e.g, with `WRITE_DISCARD` privileges) where no input data will be loaded, the callback must fill in the `output_targets` with a memory for the corresponding region requirement. These memories must be visible to the selected processor(s).
- The callback should set a memory that will hold each future produced by the task in `future_locations`.
- Normally the runtime system will retain regions with valid data even if no tasks are known that will use those regions at the time the task finishes. This policy can lead to an accumulation of read-only regions that are never garbage colleted (since read-only regions are not invalidated by any write operations). This policy can be controlled by specifying a set of indices of read-only regions in `untracked_valid_regions`—these instances will be marked for garbage collection after the task is complete.
- Optionally the task may request that the `postmap_task` be invoked for this task once mapping is complete; see Section 7.2.8.

7.2.6 Creating Physical Instances

New phyiscal instances are created by the runtime call `create_physical_instance`:

```
1  bool MapperRuntime::create_physical_instance(
2                          MapperContext ctx, Memory target_memory,
3                          const LayoutConstraintSet &constraints,
4                          const std::vector<LogicalRegion> &regions,
5                          PhysicalInstance &result,
6                          bool acquire, GCPriority priority,
7                          bool tight_bounds, size_t *footprint,
8                          const LayoutConstraint **unsat) const
```

Besides the standard runtime context, the arguments to this function are:

- The `target_memory` is the memory where the instance will be created.
- The `constraints` specify the layout constraints of the region, such as whether it should be laid out in column-major or row-major order for 2D index spaces. Layout constraints are discussed in Section 3.4.

- The `regions` field is a vector of logical regions, all of which should be included in the created instance. The ability to have more than one logical region in an instance allows for colocation of data from multiple regions.
- The `result` field holds the newly created instance after the call returns; if successful the function returns true.
- If `tight_bounds` is true, then the call will select the most specific (tightest) solution to the constraints, if more than one solution is possible. Otherwise, the runtime is free to pick any valid solution.
- `footprint` holds the size of the allocated instance in bytes.
- `unsat` holds any constraints that could not be satisfied if the call fails.

The runtime function `find_or_create_physical_instance` provides higher level functionality that preferentially finds an existing physical instance satisfying some constraints or creates a new one if necessary. The default mapper also provides higher-level functions that wrap `create_physical_instance`; see `default_create_custom_instances` for an example.

7.2.7 Selecting Sources for New Physical Instances

When a new physical instance is created, if its contents may be read the mapper callback `select_task_sources` will be invoked to pick a source of data for the instance:

```
1  virtual void select_task_sources(const MapperContext ctx,
2    const Task& task,
3    const SelectTaskSrcInput& input,
4    SelectTaskSrcOutput& output) = 0;
5
6  struct SelectTaskSrcInput {
7    PhysicalInstance target;
8    std::vector<PhysicalInstance> source_instances;
9    unsigned region_req_index;
10 };
11
12 struct SelectTaskSrcOutput {
13   std::deque<PhysicalInstance> chosen_ranking;
14 };
```

An implementation of this callback fills in `chosen_ranking` with a queue of instances selected from `source_instances`, most preferred instance first. The default mapper, for example, ranks instances in order of bandwidth between the source instance and the target memory—see `default_policy_select_target_memory` in `default_mapper.cc`.

Despite its name, this callback is also used for other operations that create new physical instances, such as copy operations.

7.2.8 Postmapping

The callback `postmap_task` is called only if requested by `map_task` (see Section 7.2.5). The purpose of this callback is to allow additional copies of regions updated by a task to be made once the task has finished. As input the callback is given the mapped instances for each region requirement as well as the valid instances. The callback should fill in `chosen_instances` with a vector for each region requirement of additional copies to be made; possible sources of these copies are specified by `source_instances`.

```
1  virtual void postmap_task(
2      const MapperContext ctx,
3      const Task& task,
4      const PostMapInput& input,
5      PostMapOutput& output) = 0;
6
7  struct PostMapInput {
8      std::vector<std::vector<PhysicalInstance> > mapped_regions;
9      std::vector<std::vector<PhysicalInstance> > valid_instances;
10 };
11
12 struct PostMapOutput {
13     std::vector<std::vector<PhysicalInstance> > chosen_instances;
14     std::vector<std::vector<PhysicalInstance> > source_instances;
15 };
```

7.2.9 Using Virtual Mappings

A useful optimization is to use *virtual mapping* for a logical region argument that a task does not use itself but only passes as an argument to a subtask. A virtual mapping is just a way of recording that no physical instance will be created for the region argument, but the name and metadata for the region are still available so that it can be passed as an argument to subtasks.

The function `PhysicalInstances::get_virtual_instance()` returns a virtual instance which can be used as the chosen physical isntance of a region requirement. If a task variant is marked as an `inner` task (meaning that it does not access any of its regions and only passes them on to subtasks), the default mapper will use virtual instances for all of the region arguments, except for fields with reduction privileges, for which the Legion runtime always requires a real physical instance to be mapped. See `map_task` in `default_mapper.cc`.

7.3 Other Mapping Features

Custom policies for mapping tasks and their region requirements are the most common reasons for users to write their own mappers. In this section we cover a few other mapping features that can be included in custom mappers. This section is very incomplete; only a handful of calls relevant to other features covered in this manual are currently included.

7.3.1 Profiling Requests

Legion has a general interface to profiling through the type `ProfileRequest`, which has one public method, `add_measurement()`. Most Legion operations take an optional profile request that will turn on the gathering of profiling information for that specific operation. Most profiling is done in the Realm low-level runtime, and running a Legion program with the command-line flag `-lg:prof` will turn on profiling of many runtime operations; see `https://legion.stanford.edu/profiling/index.html#legion-prof` for an introduction to using the Legion profiler. Most users only use the Legion profiler, but `ProfileRequests` are available for users who want more selective control over profiling.

7.3.2 Mapping Acquires and Releases

The callback `map_acquire` is called for every `acquire` operation. Other than the possibility of adding a profiling request, `map_acquire` has no options to set.

For the callback `map_release` there is a policy decision to make:

```
1  virtual void select_release_sources(
2    const MapperContext ctx,
3    const Release& release,
4    const SelectReleaseSrcInput& input,
5    SelectReleaseSrcOutput&  output) = 0;
6
7  struct SelectReleaseSrcInput {
8    PhysicalInstance target;
9    std::vector<PhysicalInstance> source_instances;
10 };
11
12 struct SelectReleaseSrcOutput {
13   std::deque<PhysicalInstance> chosen_ranking;
14 };
```

Recall that the semantics of release is that it restores the copy restriction on a region with simultaneous coherence and any updates to the region are flushed to the original `target` instance. This call allows the mapper to

produce a ranking `chosen_ranking` of which of the valid instances of the region `source_instances` should be the source of the copy to the `target` at the point of the release.

7.3.3 Controlling Stealing

There are two callbacks for controlling how tasks are stolen. A mapper may try to steal tasks from another mapper using `select_steal_targets`, and a mapper can control which tasks it allows to be stolen using `permit_steal_request`.

Mappers that want to steal tasks should implement `select_steal_targets`. This callback sets `targets` to a set of processors from which tasks can be stolen. A `blacklist` is supplied as input, which records processors for which a previous steal request failed due to insufficient work. The blacklist is managed automatically by the runtime system, and processors are removed from the blacklist when they acquire additional work.

```
1  struct SelectStealingInput {
2    std::set<Processor> blacklist;
3  };
4
5  struct SelectStealingOutput {
6    std::set<Processor> targets;
7  };
8
9  virtual void select_steal_targets(
10   const MapperContext ctx,
11   const SelectStealingInput& input,
12   SelectStealingOutput& output) = 0;
```

When a mapper receives a steal request the `permit_steal_request` callback is invoked, notifying the mapper of the requesting processor (the `thief`) and the tasks the mapper has available to steal, from which the callback selects a set of `stolen_tasks`.

```
1  struct StealRequestInput {
2    Processor thief_proc;
3    std::vector<const Task*> stealable_tasks;
4  };
5
6  struct StealRequestOutput {
7    std::set<const Task*> stolen_tasks;
8  };
9
10 virtual void permit_steal_request(const MapperContext ctx,
11   const StealRequestInput& input,
12   StealRequestOutput& output) = 0;
```

7.4 Mappers Included with Legion

Several useful mappers are included in the Legion repository:

- The *default mapper* has already been discussed. The default mapper is a full implementation of the legion mapping API with reasonably heuristics for every mapping callback. The default mapper has grown over time—as users have found cases where the default mapper did not perform well, improvements have been made. As a result, the default mapper is a non-trivial mapper, even though it still does not come close to achieving optimal mappings for most complex applications.

- The *null mapper* is a base class that fails an assertion for every mapper API call. The null mapper is a useful starting point when writing a mapper from scratch, as the mapper will show exactly which API calls need to be implemented to support the application.

- The *replay mapper* can be used to replay mapping decisions recorded in a replay file by Legion Spy. The replay mapper is used mostly for ensuring that a failed computation can be deterministically replayed to help diagnose the source of bugs in the Legion runtime itself.

- The *logging wrapper* adds logging of mapping operations (which calls were made and with what arguments) to an existing mapper. To use the logging wrapper, replace any use of `new MyMapper(...)` in the application with `new LoggingWrapper(new MyMapper(...))` and run with the command line flag `-level mapper=2`.

- The *forwarding mapper* is a base class used to build mapper wrappers; the fowarding mapper simply forwards all mapper calls to another mapper. The logging wrapper is written using the forwarding mapper.

Chapter 8

Interoperation

In this chapter we briefly discuss the most common scenarios where Legion programs need to interoperate with other systems. We will rely in this chapter on examples from the `legion/examples` directory in the Legion repository.

8.1 MPI

Legion has well-developed support for interoperation with MPI. The essentials of the approach are:

- The top-level Legion task is control-replicated, with the number of shards equal to the number of ranks of MPI.
- Legion and MPI time-slice the machine: One of MPI or Legion is running at any given time, while the other runtime waits.
- Data can be shared between MPI and Legion by attaching an MPI buffer as a region with simultaneous coherence (with the correct layout constraint to ensure the buffer contents are interpreted correctly by Legion). The data can be moved back and forth between a shard of the top-level task and the corresponding MPI rank using the producer-consumer synchronization discussed in Section 6.2.

MPI interoperation is illustrated in `legion/examples/mpi_with_ctrl_repl/mpi_with_ctrl_repl.cc`:

```
1  MPILegionHandshake handshake;
2  ...
3  // This is the preferred way of using handshakes in Legion
4  IndexLauncher worker_launcher(WORKER_TASK_ID, launch_bounds, TaskArgument(NULL, 0), args_map);
5  // We can use our handshake as a phase barrier
```

```
6     // Record that we will wait on this handshake
7     worker_launcher.add_wait_handshake(handshake);
8     // Advance the handshake to the next version
9     handshake.advance_legion_handshake();
10    // Then record that we will arrive on this version
11    worker_launcher.add_arrival_handshake(handshake);
12    // Launch our worker task
13    // No need to wait for anything
14    runtime->execute_index_space(ctx, worker_launcher);
15
```

In this excerpt, we see that the synchronization between MPI and Legion is wrapped in a `MPILegionHandshake` object (line 1). The handshake encapsulates a phase barrier and is used similarly (see Section 6.2), but a handshake also knows how to work with MPI. An index task launcher is built to run the Legion-side work (line 4) and its execution is deferred until the MPI side signals it is done running (line 7). Just like a phase barrier, handshakes have generations so that they can be reused multiple times, typically across iterations of a loop. The handshake is advanced to the next generation (line 9) and when the index tasks are finished the (new generation of the) handshake is signaled to restart the MPI side (line 11).

The MPI side of the interface is symmetric. From the same example:

```
1   for (int i = 0; i < total_iterations; i++)
2   {
3     printf("MPI Doing Work on rank %d\n", rank); // MPI work goes here
4     if (strict_bulk_synchronous_execution)
5       MPI_Barrier(MPI_COMM_WORLD);
6     // Perform a handoff to Legion, this call is
7     // asynchronous and will return immediately
8     handshake.mpi_handoff_to_legion();
9     ..
10    // Wait for Legion to hand control back,
11    // This call will block until a Legion task
12    // running in this same process hands control back
13    handshake.mpi_wait_on_legion();
14    if (strict_bulk_synchronous_execution)
15      MPI_Barrier(MPI_COMM_WORLD);
16  }
```

MPI uses the same handshake object as Legion. Note that the call to `mpi_wait_on_legion` blocks until Legion arrives at the handshake; the other arrive/wait handshake methods are asynchronous. Because the MPI side blocks while it is waiting on Legion, it is not concerned with the generation of the handshake, so the generation should only be advanced by the Legion side to allow for deferred execution of Legion tasks.

8.2 OpenMP

Legion provides a straightfoward model of interoperation with OpenMP. Legion tasks may use OpenMP pragmas internally to exploit multiple threads in a single kernel. Legion tasks that use OpenMP should be mapped to OMP processors, which can be enforced by adding an OMP constraint when the task is registered.

Under the hood Legion interoperates with OpenMP by directly implementing OpenMP functionality. Only a subset of OpenMP is supported, but the support extends to the most commonly used features, particularly `omp parallel for`.

The program `legion/omp_saxpy` illustrates typical uses of OpenMP in Legion programs. In this code, the leaf tasks (the tasks that do not call other tasks) include OpenMP pragmas. For example, in `simple_blas.cc` a dot product operation is defined:

```
1  template ⟨⟩
2  float BlasTaskImplementations⟨float⟩::dot_task_cpu(const Task *task,
3                                                    const std::vector⟨PhysicalRegion⟩ &regions,
4                                                    Context ctx, Runtime *runtime)
5  {
6    IndexSpace is = regions[1].get_logical_region().get_index_space();
7    Rect⟨1⟩ bounds = runtime->get_index_space_domain(ctx, is);
8
9    const FieldAccessor⟨READ_ONLY,float,1,coord_t,
10         Realm::AffineAccessor⟨float,1,coord_t⟩ ⟩ fa_x(regions[0], task->regions[0].instance_fields[0]);
11   const FieldAccessor⟨READ_ONLY,float,1,coord_t,
12         Realm::AffineAccessor⟨float,1,coord_t⟩ ⟩ fa_y(regions[1], task->regions[0].instance_fields[0]);
13
14   float acc = 0;
15 #pragma omp parallel for reduction(+:acc) if(blas_do_parallel)
16   for(int i = bounds.lo[0]; i ⟨= bounds.hi[0]; i++)
17     acc += fa_x[i] * fa_y[i];
18   return acc;
19 }
```

Note that unlike most other examples in this manual, this code uses an `AffineAccessor` for the fields. Affine accessors support indexing into regions like arrays, which is necessary in this example because the different iterations of the loop will be split across multiple OpenMP threads—we cannot use an iterator here, as an iterator by definition defines a sequential order of access to the elements iterated over. The `axpy` task in the same file gives another example of using OpenMP pragmas within Legion tasks.

The code for registering the tasks that use OpenMP is in `simple_blas.inl`:

```
1    {
2      dot_task_id = Runtime::generate_static_task_id();
3      TaskVariantRegistrar tvr(dot_task_id);
4  #ifdef REALM_USE_OPENMP
```

```
5     tvr.add_constraint(ProcessorConstraint(Processor::OMP_PROC));
6   #else
7     tvr.add_constraint(ProcessorConstraint(Processor::LOC_PROC));
8   #endif
9     Runtime::preregister_task_variant<T, BlasTaskImplementations<T>::dot_task_cpu>(tvr, "dot (cpu)");
10  }
```

This code is parameterized on whether OpenMP is to be used or not; if it is used the processor constraint for the task is set to `OMP_PROC`, otherwise it is set to `LOC_PROC` (i.e., CPUs).

There are a few command-line flags that affect the execution of Legion programs using OpenMP:

- `-ll:ocpus n` sets the number of CPUs to be reserved for OpenMP to n.
- `-ll:othr t` sets the number of threads per CPU to t.
- `-ll:okindhack` exposes the master OpenMP thread as a CPU processor. This flag is useful when running with `-ll:cpus 0` to give an extra processor to the OpenMP runtime; if there are some remaining CPU tasks they can be sent to the procesor running the master OpenMP thread using `-ll:okindhack`.
- `-ll:onuma` ensures that OpenMP cores are grouped by NUMA domain; a warning is printed if NUMA support is not found.
- `-ll:ostack m` sets the OpenMP stack size to m bytes.

Finally, Legion is not compiled with OpenMP by default. To enable OpenMP, build Legion with `USE_OPENMP = 1`.

8.3 HDF5

HDF5 is a standard file format for storing very large files. HDF5 is hierarchical: files can be made of *groups*, which can be nested, with data sets stored at the leaves of the group structure. HDF5's hierarchical files map well on to Legion's regions and subregions.

Legion provides support for reading and writing HDF5 files using logical regions. We have already seen all of the mechanisms in Section 6.2: HDF5 files are treated as external resources that are attached to a region with simultaneous coherence. After acquiring the region (to release the copy restriction) copies can be made of the data; when the coherence is released any updates are flushed back to the original region instance.

A simple example of creating and writing checkpoints using an HDF5 file is `legion/examples/attach_file`. Most of the calls that manipulate HDF5 files in this program are actually direct calls to the HDF5 library, such as calls to `H5Fcreate`, `H5Gcreate2`, `H5Gclose`, `H5Sclose`, and `H5Fclose` in `generate_hdf_file`. See the HDF5 documentation for the semantics of these calls.

The place where the API intersects with HDF5 is in the use of an attach launcher to bind a region to an HDF5 file. The relevant excerpt from this example:

```
1  #ifdef LEGION_USE_HDF5
2    if(*hdf5_file_name) {
3      // create the HDF5 file first - attach wants it to already exist
4      bool ok = generate_hdf_file(hdf5_file_name, hdf5_dataset_name, num_elements);
5      assert(ok);
6      std::map<FieldID,const char*> field_map;
7      field_map[FID_CP] = hdf5_dataset_name;
8      printf("Checkpointing data to HDF5 file '%s' (dataset='%s')\n",
9             hdf5_file_name, hdf5_dataset_name);
10     AttachLauncher al(LEGION_EXTERNAL_HDF5_FILE, cp_lr, cp_lr);
11     al.attach_hdf5(hdf5_file_name, field_map, LEGION_FILE_READ_WRITE);
12     cp_pr = runtime->attach_external_resource(ctx, al);
13   } else
14 #endif
```

The only differences in this code from the discussion of attach launchers in Section 6.2 are the constant `LEGION_EXTERNAL_HDF5_FILE` for the external resource argument of the constructor on line 10 and the use of the `attach_hdf5` method with the access mode `LEGION_FILE_READ_WRITE` on line 11.

HDF5 is not included in the Legion build by default. Set `USE_HDF5=1` to build with HDF5 supoort. The variable `HDF_ROOT` can be set to the root directory of the HDF library if needed.

8.4 Kokkos

Legion supports running Kokkos tasks as part of a Legion execution. An example program is in the directory `legion/examples/kokkos_saxpy`. From the Legion point of view, Kokkos is most useful as a processor-agnostic portability layer for kernels, allowing the same kernel to target both GPUs and CPUs, for example. In Kokkos, the parameterization over the processor kind is handled by C++ templates over *execution spaces*. Instantiating the execution space with different arguments allows Kokkos to generate code for GPUs, GPUs, and OpenMP.

The need to parameterize Kokkos tasks on the execution space, combined some limitations of C++ templates, causes the implementation of Kokkos tasks in Legion to look quite different from other task implementations we have seen. For example, the initialization task in `kokkos_saxpy` is:

```
template ⟨typename execution_space⟩
class InitTask {
public:
  static void task_body(const Task *task,
                        const std::vector⟨PhysicalRegion⟩ &regions,
                        Context ctx, Runtime *runtime)
  {
    printf("kokkos(%s) init task on processor " IDFMT ", kind %d\n",
           typeid(execution_space).name(),
           runtime->get_executing_processor(ctx).id,
           runtime->get_executing_processor(ctx).kind());

    const float offset = *reinterpret_cast⟨const float *⟩(task->args);

    Rect⟨1⟩ subspace = runtime->get_index_space_domain(ctx,
                                                       task->regions[0].region.get_index_space());

    AccessorRW acc(regions[0], task->regions[0].instance_fields[0]);

    // you can use relative indexing for your own kernels too - just make
    //  sure you do the right thing when operating on a partitioned
    //  subregion!
    Kokkos::View⟨float *,
                Kokkos::LayoutStride,
                typename execution_space::memory_space⟩ view = acc.accessor;

    size_t n_elements = subspace.hi.x - subspace.lo.x + 1;
    Kokkos::RangePolicy⟨execution_space⟩ range(runtime->get_executing_processor(ctx).kokkos_work_space(),
                                               0, n_elements);
    Kokkos::parallel_for(range,
                         KOKKOS_LAMBDA (int i) {
                           // using a relative address, but value to store
                           //  is based on global index
                           // have to use a relative address!
                           view(i) = (i + subspace.lo.x) + offset;
                         });
  }
};
```

A Kokkos task is implemented as a C++ class with a method `task_body` that has the code for the task (and has the standard Legion signature for a task). The class definition is wrapped in a template with an execution space parameter.

Another point where Legion and Kokkos interact is in the mapping of Legion's region accessors into Kokkos views—for code generation in Kokkos to work well, it is important that Kokkos views be used to access data, though these views can often be Legion accessors cast to a suitable Kokkos

type (e.g., lines 21-25 above).

Finally, because Kokkos tasks are templated classes, registering a Kokkos task requires additional logic to instantiate the template with the kind of processor to be used and then registering the `task_body` function of the resulting class. In `kokkos_saxpy`, this logic is encapsulated in the function `preregister_kokkos_task`:

```
1  template ⟨template⟨typename⟩ class PORTABLE_KOKKOS_TASK⟩
2  void preregister_kokkos_task(TaskID task_id, const char *name)
3  {
4    // register a serial version on the CPU
5    {
6      TaskVariantRegistrar registrar(task_id, name);
7      registrar.add_constraint(ProcessorConstraint(Processor::LOC_PROC));
8      Runtime::preregister_task_variant⟨
9        PORTABLE_KOKKOS_TASK⟨Kokkos::Serial⟩::task_body ⟩(registrar, name);
10   }
11   ...
12 }
13 ...
14 preregister_kokkos_task⟨InitTask⟩(INIT_TASK_ID, "init");
```

Note that `pregister_kokkos_task` is templated on the class representing the task to be registered.

Enable `USE_Kokkos=1` to build with Kokkos support, which is not included in Legion by default. The compile-time flags `KOKKOS_ENABLE_CUDA`, `KOKKOS_ENABLE_OPENMP` and `KOKKOS_ENABLE_SERIAL` enables generation of kernels for GPUs, OpenMP, and CPUs, respectively.

8.5 Python

Legion provides extensive interoperation support with Python through Pygion, which implements the Legion programming model in Python. Pygion is essentially untyped Regent with Python syntax. All of the Legion programming features are available through Pygion.

One of the additional features that Pygion provides is the ability to define a Legion task in Python. This task, when run, will execute in a Python interpreter. Given the overheads of Python, tasks written in Python (as opposed to task simply called from Python) will generally be slow compared to the same task written in C++ or CUDA, but for small amounts of compute the convenience of having all of the Python facilities available outweighs what should be a negligible performance penalty.

Python tasks run on dedicated Python processors, for which Legion reserves a core to run the Python interpreter.

There are numerous examples of Pygion code illustrating the use of all of the Legion features in `legion/bindings/python/examples`. As an example, a simple "hello, world" program is in `legion/bindings/python/examples/hello.py`:

```
from pygion import task

@task
def main():
    print("Hello, Legion!")

if __name__ == '__main__':
    main()
```

More information on Pygion can be found in [SA19].

Bibliography

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

[SA19] Elliott Slaughter and Alex Aiken. Pygion: Flexible, scalable task-based parallelism with Python. In *Parallel Applications Workshop, Alternatives To MPI (PAW-ATM)*, 2019.