Legion Execution Model

Elliott Slaughter

Tasks

What is a Task?

- Simple answer:
- Function
  - Single threaded execution context
  - Takes regions (futures, predicates, …) as arguments
  - Computes a result (region, future, …)
  - Optionally launches subtasks

What is a Task?

- More sophisticated answer:
- Unit of control
- What that means depends on processor type:
  - CPU: Single thread
  - GPU: Host function (single thread) with an attached CUDA context
  - "OpenMP" processor: Multiple threads on a CPU
  - "OpenGL" processor: Host function (single thread) with an attached graphics context
- Coarse-grained parallelism between tasks
- Fine-grained parallelism within tasks (optional)

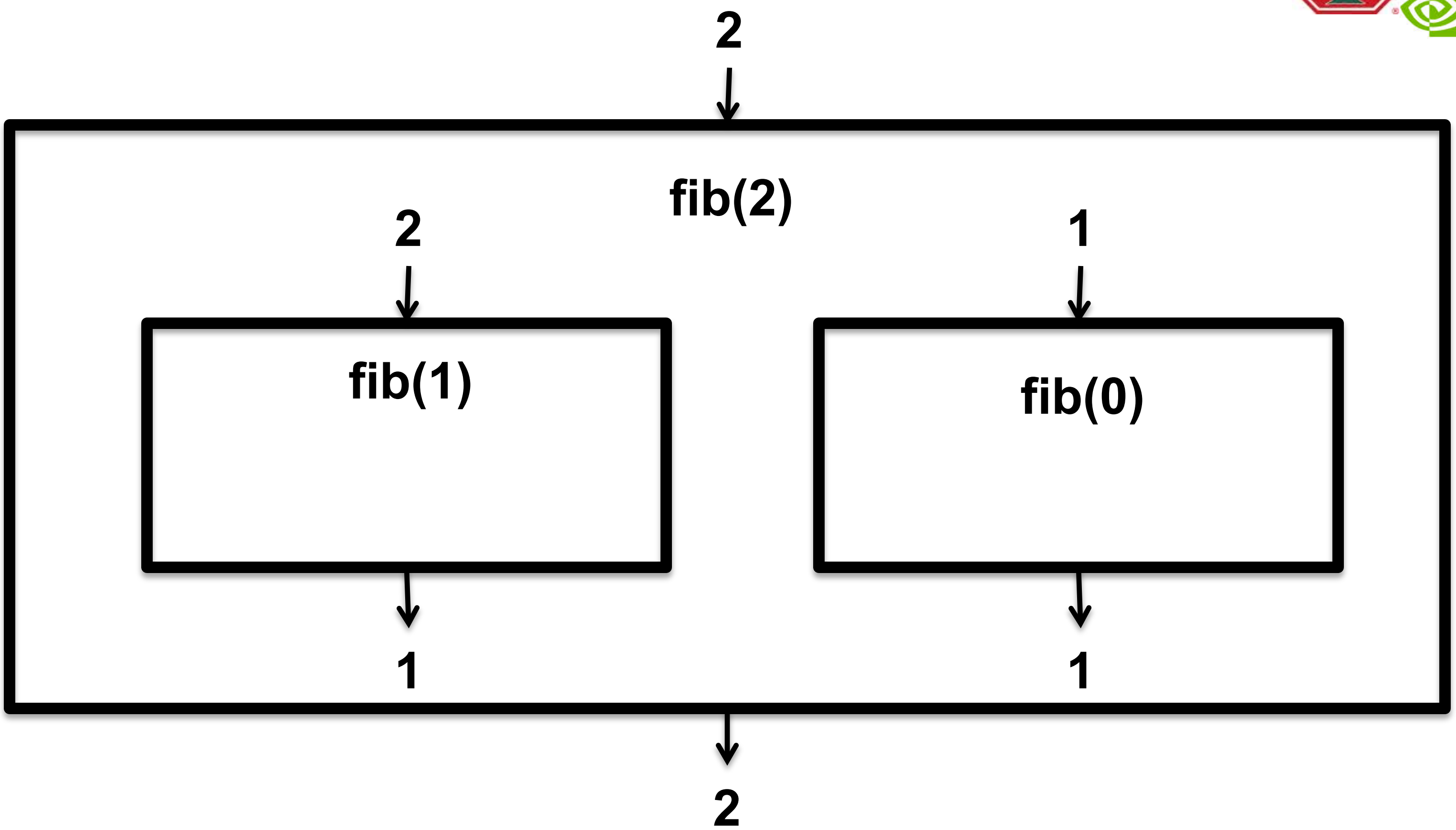
Task Do's and Don'ts

- Tasks do:
  - Explicitly declare inputs and outputs (regions*)
  - Wait for inputs (regions*) to be ready before starting
  - Exclusively** read and modify regions
  - Launch subtasks
- Tasks do NOT:
  - Communicate while running**
  - Stop for anything once started**

*Also futures, predicates, phase barriers, …

**There are exceptions; see Mike's talk on advanced features

Isolation



- **All* synchronization, communication happens at task boundaries**
  - ***Again, there are exceptions**

Kinds of Tasks

- Single Tasks
  - Like a single function call
- Index Space Tasks
  - Like a (potentially nested) for loop around a function call
  - Requires that all invocations be independent
  - Amortize dynamic analysis costs (vs many single tasks)

Declaring Tasks

```
void task_top(const Task *task,
  const std::vector<PhysicalRegion> &regions,
  Context ctx, HighLevelRuntime *runtime)
{ /* ... */ }

int task_fib(const Task *task,
  const std::vector<PhysicalRegion> &regions,
  Context ctx, HighLevelRuntime *runtime)
{
  assert(task->arglen == sizeof(int));
  int arg = *static_cast<int *>(task->args);
}
```

***These tasks are statically compiled; see also Sean's talk on Terra and dynamic compilation**

Registering Tasks

```
enum { TASK_TOP = 100, TASK_FIB };

int main(int argc, char **argv) {
  HighLevelRuntime::register_legion_task<task_top>(
    TASK_TOP,
    Processor::LOC_PROC, true /*single*/, false /*index*/,
    AUTO_GENERATE_ID, TaskConfigOptions(), "top");

  HighLevelRuntime::register_legion_task<int, task_fib>(
    TASK_FIB,
    Processor::LOC_PROC, true /*single*/, false /*index*/,
    AUTO_GENERATE_ID, TaskConfigOptions(), "fib");

  HighLevelRuntime::set_top_level_task_id(TASK_TOP);

  return HighLevelRuntime::start(argc, argv);
}
```

Registering Task Variants

```
HighLevelRuntime::register_legion_task<int, task_fib_cpu>(
  TASK_FIB,
  Processor::LOC_PROC, true /*single*/, false /*index*/,
  AUTO_GENERATE_ID, TaskConfigOptions(), "fib");

HighLevelRuntime::register_legion_task<int, task_fib_gpu>(
  TASK_FIB,
  Processor::TOC_PROC, true /*single*/, false /*index*/,
  AUTO_GENERATE_ID, TaskConfigOptions(), "fib");
```

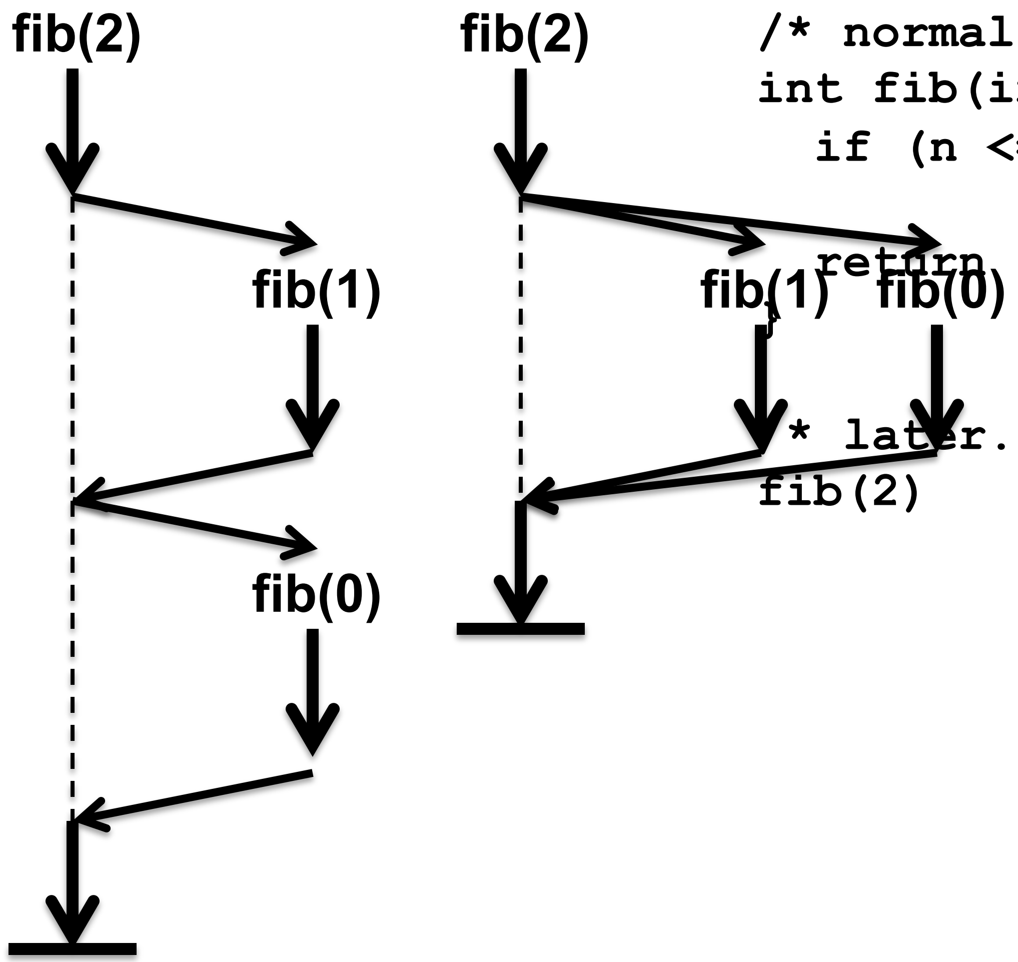
*Mapper chooses which task to run at runtime; see Mike's talk on mapping

TaskConfigOptions

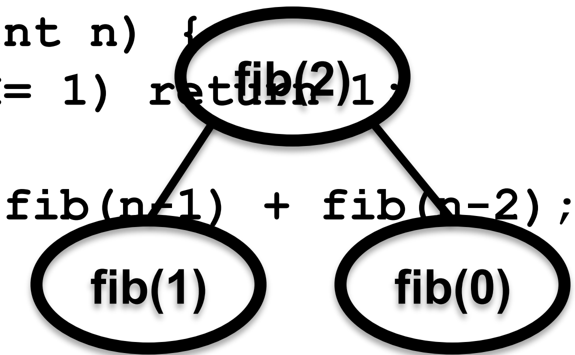
- Tasks may be (zero or more of):
- Leaf
  - Must not call into the runtime
- Inner
  - Must not read or modify any regions
  - Usually, inner tasks are tasks that just spawn other tasks
  - Similar to Sequoia's inner task qualifier
- Idempotent
  - Must have no externally-visible side-effects (e.g. disk I/O, dispensing money out of an ATM, …)
  - Useful for speculation/resilience
    - Implies that tasks can be re-run automatically

Execution Model

Implicit Parallelism with Explicit Serial Semantics




```
/* normal, serial C code */
int fib(int n) {
  if (n <= 1) return 1;
  return fib(n-1) + fib(n-2);
}
/* later... */
fib(2)
```



Invoking Tasks

```
int arg = *static_cast<int *>(task->args);

int arg1 = arg - 1;
TaskLauncher fib1(TASK_FIB,
                  TaskArgument(&arg1, sizeof(arg1)));
Future future1 = runtime->execute_task(ctx, fib1);

int arg2 = arg - 2;
TaskLauncher fib2(TASK_FIB,
                  TaskArgument(&arg2, sizeof(arg2)));
Future future2 = runtime->execute_task(ctx, fib2);

return future1.get_result<int>() +
  future2.get_result<int>();
```

Deferred Execution

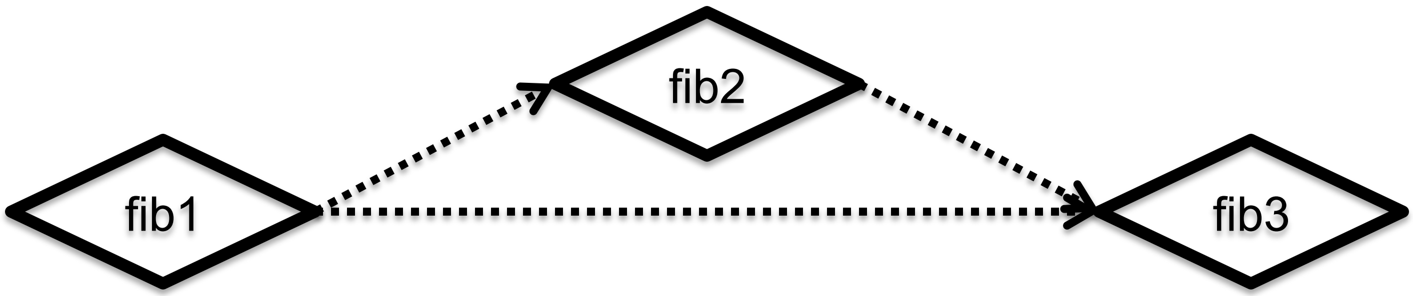
- Deferred is not the same as asynchronous
  - Deferred operations are composable
  - Think OpenGL, not MPI_Isend
- Deferred means:
  - Operations run asynchronously, return handle to result
  - That handle can be passed to other operations
- Deferred execution allows Legion to hide latency
  - Communication
  - Dynamic analysis
- Critical to performance in Legion!

Explicit Dataflow with Futures

```
TaskLauncher fib1(TASK_FIB, TaskArgument());
Future future1 = runtime->execute_task(ctx, fib1);

TaskLauncher fib2(TASK_FIB, TaskArgument());
fib2.add_future(future1);
Future future2 = runtime->execute_task(ctx, fib2);

TaskLauncher fib3(TASK_FIB, TaskArgument());
fib3.add_future(future1);
fib3.add_future(future2);
Future future3 = runtime->execute_task(ctx, fib3);
```



-Launcher All The Things!

- TaskLauncher (for single tasks)
- IndexLauncher (for index space tasks)
- InlineLauncher (for inline mappings)
- CopyLauncher (for explicit copies)
- …

- All follow the same pattern

TaskLauncher

```
/* legion.h: struct TaskLauncher */
TaskLauncher(Processor::TaskFuncID tid,
             TaskArgument arg,
             Predicate pred = Predicate::TRUE_PRED,
             MapperID id = 0,
             MappingTagID tag = 0);

void add_index_requirement(const IndexSpaceRequirement &);
void add_region_requirement(const RegionRequirement &);
void add_future(Future);
void add_grant(Grant);
void add_wait_barrier(PhaseBarrier);
void add_arrival_barrier(PhaseBarrier);
```

Questions?