Writing Legion Abstractions

Alex Aiken

Scope

- Cover some common approaches to structuring Legion code
- Just based on our own experience
- And not comprehensive …

Launcher Class (legion.h)

```
struct TaskLauncher {
public:
  TaskLauncher(void);
  TaskLauncher(Processor::TaskFuncID tid,
               TaskArgument arg,
               Predicate pred = Predicate::TRUE_PRED,
               MapperID id = 0,
               MappingTagID tag = 0);
public:
  inline void add_index_requirement(const IndexSpaceRequirement &req);
  inline void add_region_requirement(const RegionRequirement &req);
  inline void add_field(unsigned idx, FieldID fid, bool inst = true);
public:
  inline void add_future(Future f);
  inline void add_grant(Grant g);
  inline void add_wait_barrier(PhaseBarrier bar);
  inline void add_arrival_barrier(PhaseBarrier bar);
public:
  inline void set_predicate_false_future(Future f);
  inline void set_predicate_false_result(TaskArgument arg);
public:
  Processor::TaskFuncID              task_id;
  std::vector<IndexSpaceRequirement> index_requirements;
  std::vector<RegionRequirement>     region_requirements;
  std::vector<Future>                futures;
  std::vector<Grant>                 grants;
  std::vector<PhaseBarrier>          wait_barriers;
  std::vector<PhaseBarrier>          arrive_barriers;
  TaskArgument                       argument;
  Predicate                          predicate;
  MapperID                           map_id;
  MappingTagID                       tag;
  DomainPoint                        point;
```

Launcher Class (legion.h)

- Create a new launcher class per task
  - Inherit from one of the launcher base classes
  - (There is also a base class for index launches)

- Put all the code for a task in its launcher object
  - Registration
  - Launching
  - Mapping
  - Task variants can be static methods

Example

```
class CalcDiffusionTask : public IndexLauncher {
public:
  CalcDiffusionTask(S3DRank *rank,
                    Domain domain,
                    TaskArgument global_arg,
                    ArgumentMap arg_map,
                    Predicate pred = Predicate::TRUE_PRED,
                    bool must = false,
                    MapperID id = 0,
                    MappingTagID tag = 0,
                    bool add_requirements = true);
public:
  void launch_check_fields(Context ctx, HighLevelRuntime *runtime);
public:
  static const char * const TASK_NAME;
  static const int TASK_ID = CALC_DIFFUSION_TASK_ID;
  static const int MAPPER_TAG = RHSF_MAPPER_PRIORITIZE;
  ...
public:
  static void cpu_base_impl(S3DRank *rank, const Task *task, const Rect<3> &subgrid_bounds,
                            const std::vector<RegionRequirement> &reqs,
                            const std::vector<PhysicalRegion> &regions,
                            Context ctx, HighLevelRuntime *runtime);
  static void gpu_base_impl(S3DRank *rank, const Task *task, const Rect<3> &subgrid_bounds,
                            const std::vector<RegionRequirement> &reqs,
                            const std::vector<PhysicalRegion> &regions,
                            Context ctx, HighLevelRuntime *runtime);
};
```

Another Example

```
class CalcViscosityTask : public IndexLauncher {
public:
  CalcViscosityTask(S3DRank *rank,
                    Domain domain,
                    TaskArgument global_arg,
                    ArgumentMap arg_map,
                    Predicate pred = Predicate::TRUE_PRED,
                    bool must = false,
                    MapperID id = 0,
                    MappingTagID tag = 0,
                    bool add_requirements = true);
public:
  void launch_check_fields(Context ctx, HighLevelRuntime *runtime);
public:
  S3DRank *rank;
public:
  static const char * const TASK_NAME;
  static const int TASK_ID = CALC_VISCOSITY_TASK_ID;
  static const int MAPPER_TAG = 0;
...
public:
  static void cpu_base_impl(S3DRank *rank, const Task *task, const Rect<3> &subgrid_bounds,
                            const std::vector<RegionRequirement> &reqs,
                            const std::vector<PhysicalRegion> &regions,
                            Context ctx, HighLevelRuntime *runtime);
  static void gpu_base_impl(S3DRank *rank, const Task *task, const Rect<3> &subgrid_bounds,
                            const std::vector<RegionRequirement> &reqs,
                            const std::vector<PhysicalRegion> &regions,
                            Context ctx, HighLevelRuntime *runtime);
};
```

So What?

- If each task is encapsulated in an object …
- With the same interface …
- Then task launches are easily templated
  - Write the boilerplate code for launching tasks in one place
  - Parameterized on the type of task T
  - Different templates for different situations
    - E.g., launch CPU variant vs. GPU variant

Example

```
template<typename T>
void dispatch_task(T &launcher, FutureMap& fm, Context ctx,
                   HighLevelRuntime *runtime, int stage)
{
  launcher.tag |= (RHSF_MAPPER_FORCE_RANK_MATCH | (stage & RHSF_MAPPER_STAGE_MASK));
  fm = runtime->execute_index_space(ctx, launcher);
  if (S3DRank::get_perform_waits())
    fm.wait_all_results();
  // See if we need to perform any checks
  if (S3DRank::get_perform_all_checks())
    launcher.launch_check_fields(ctx, runtime);
}
```

- **Note: These are also convenient places to include any extra, related tasks you want to launch**
  - **E.g., integrity checking, some in-situ analysis**

Another Example

```
template<typename T>
void base_cpu_wrapper(const Task *task,
                      const std::vector<PhysicalRegion> &regions,
                      Context ctx, HighLevelRuntime *runtime)
{
  Point<3> rank_point = task->index_point.get_point<3>();
  if (S3DRank::get_show_progress())
    printf("%s CPU task, pt = (%d, %d, %d), proc = " IDFMT "\n", T::TASK_NAME,
           rank_point[0], rank_point[1], rank_point[2],
           runtime->get_executing_processor(ctx).id);

  S3DRank *rank = S3DRank::get_rank(rank_point, true);

  Blockify<3> grid2proc_map(rank->local_grid_size);
  Rect<3> my_subgrid_bounds = grid2proc_map.preimage(rank_point);

  T::cpu_base_impl(rank, task, my_subgrid_bounds, task->regions, regions, ctx, runtime);
}
```

And Another Example

```
template<typename T>
void base_gpu_wrapper(const Task *task,
                      const std::vector<PhysicalRegion> &regions,
                      Context ctx, HighLevelRuntime *runtime)
{
  Point<3> rank_point = task->index_point.get_point<3>();
  if (S3DRank::get_show_progress())
    printf("%s GPU task, pt = (%d, %d, %d), proc = " IDFMT "\n", T::TASK_NAME,
           rank_point[0], rank_point[1], rank_point[2],
           runtime->get_executing_processor(ctx).id);

  S3DRank *rank = S3DRank::get_rank(rank_point, true);

  Blockify<3> grid2proc_map(rank->local_grid_size);
  Rect<3> my_subgrid_bounds = grid2proc_map.preimage(rank_point);

  T::gpu_base_impl(rank, task, my_subgrid_bounds, task->regions, regions, ctx, runtime);
}
```

Task Registration

```
template<typename T>
void register_task(void)
{
  HighLevelRuntime::register_legion_task<base_cpu_wrapper<T> >(T::TASK_ID, Processor::LOC_PROC,
                                                                false/*single*/, true/*index*/,
                                                                RHSF_CPU_LEAF_VARIANT,
                                                                TaskConfigOptions(T::CPU_BASE_LEAF),
                                                                T::TASK_NAME);
}
```

- **And two more versions**
  - **For registering a task with a non-void return type**
  - **For registering a task with both CPU & GPU variants**

Mapping Abstractions

```
class MachineQueryInterface {
public:
  MachineQueryInterface(Machine *m);
public:
  /**
   * Find a memory visible to all the processors
   */
  Memory find_global_memory(void);
  static
  Memory find_global_memory(Machine *machine);
  /**
   * Get the memory stack for a given processor sorted
   * by either throughput or latency.
   */
  void find_memory_stack(Processor proc,
          std::vector<Memory> &stack, bool latency);
  static void find_memory_stack(Machine *machine, Processor proc,
                                std::vector<Memory> &stack, bool latency);
```

- **MachineQueryInterface implements common machine queries (and caches more expensive ones)**

Mapping Abstractions

```
class MappingMemoizer {
public:
  MappingMemoizer(void);
public:
  bool has_mapping(Processor target, const Task *task,
                   unsigned index) const;
  bool recall_mapping(Processor target, const Task *task,
      unsigned index, std::vector<Memory> &ranking) const;
  Memory recall_chosen(Processor target, const Task *task,
                       unsigned index) const;
  void record_mapping(Processor target, const Task *task,
        unsigned index, const std::vector<Memory> &ranking);
  void notify_mapping(Processor target, const Task *task,
                      unsigned index, Memory result);
  void commit_mapping(Processor target, const Task *task);
```

- **MappingMemoizer provides an interface for memoizing task mapping decisions on a processor**

Mapping Abstractions

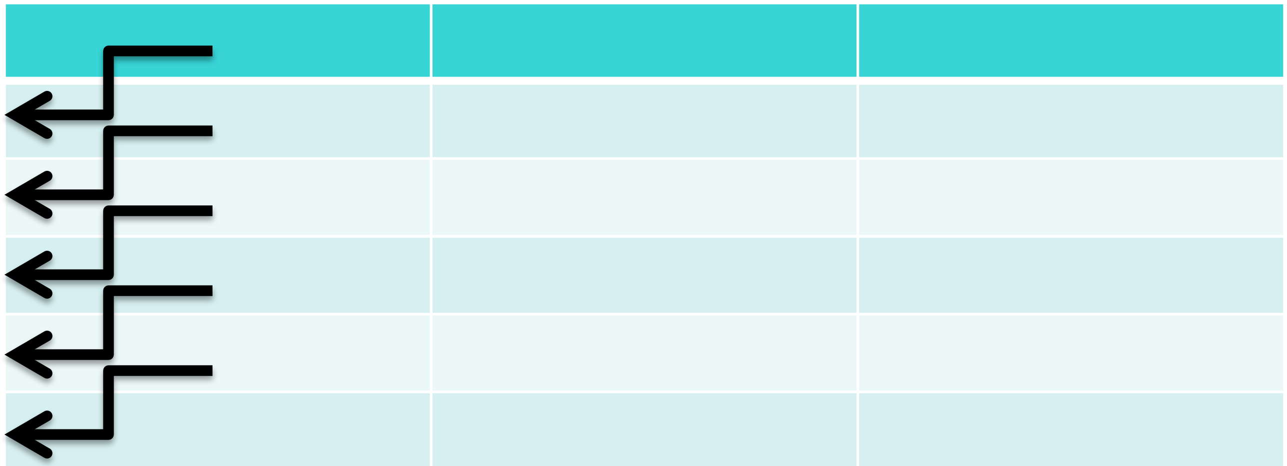
```
class MappingProfiler {
public:
  MappingProfiler(void);
public:

  void set_needed_profiling_samples(unsigned num_samples);
  void set_max_profiling_samples(unsigned max_samples);
  bool profiling_complete(const Task *task) const;

  Processor::Kind best_processor_kind(const Task *task) const;
  Processor::Kind next_processor_kind(const Task *task) const;

  void update_profiling_info(const Task *task, Processor target,
                             Processor::Kind kind,
                             const Mapper::ExecutionProfile &profile);
```

- Cycles through all variants of a task
  - And remembers the one with the best performance

A Word on Data Structures

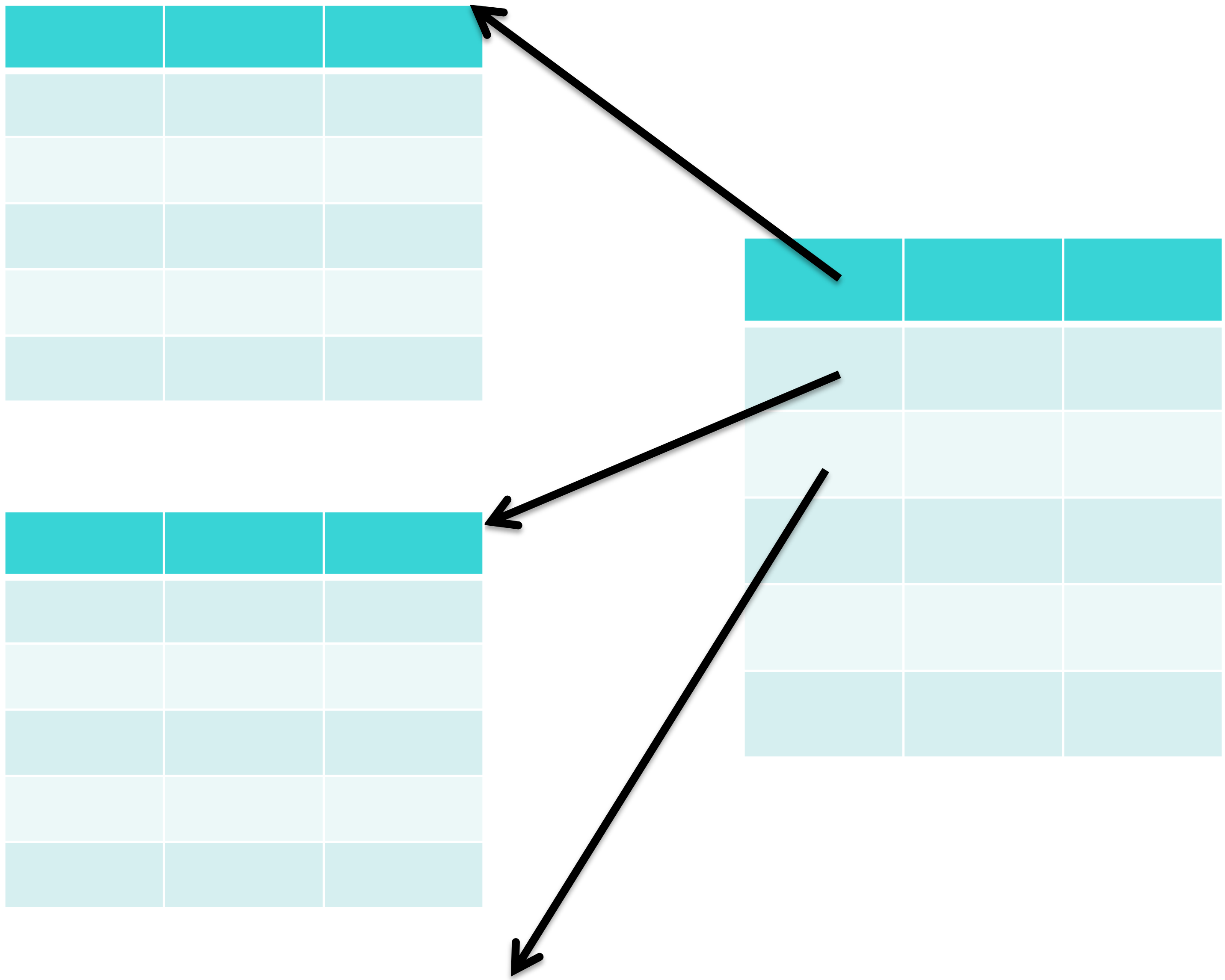
- A field can hold a “pointer” to a region
  - Or a region element
  - Implemented as a pair: the region pointed to, and the index within the region
- Guaranteed to be valid anywhere
  - Regions can be moved from place to place and all region references remain valid
- Caveat
  - To dereference a region pointer, the region must be mapped
  - Which ensures a valid copy of the data is in an addressable memory
  - Implies region pointer dereferences are always local

A Linked List



- Can also build trees, graphs, irregular meshes …

Pointers to Regions



Questions?