Regent: Tasks

CS315B
Lecture 3

Design Goals

- Sequential semantics
  - The better to understand what you write
  - Parallelism is extracted automatically
- Throughput-oriented
  - The latency of a single thread/process is (mostly) irrelevant
  - The overall time is what matters
- Runtime decision making
  - Because machines are unpredictable/dynamic

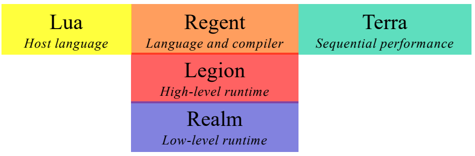
Throughput-Oriented

- Keep the machine busy
- How? Ideally,
  - Every core has a queue of independent work to do
  - Every memory unit has a queue of transfers to do
  - At all times
- C.f., bulk-synchronous model

Consequences

- Highly asynchronous
  - Minimize synchronization
  - Esp. global synchronization
- Sequential semantics but support for parallelism
- Emphasis on describing the structure of data
  - Next lecture

Regent Stack



Examples 0 & 1

- Embedded in Lua
  - Popular scripting language in the graphics community
- Excellent interoperation with C
  - And with other languages
- Python-ish syntax
  - For both Lua and Regent

Tasks

- Tasks are Regent's unit of parallel execution
  - Distinguished functions that can be executed asynchronously
- No preemption
  - Tasks will run until they block or terminate
  - And ideally they don't block …

Examples 2 & 3

- Tasks can call subtasks
- Nested parallelism
  - To arbitrary depth
- Terminology: *parent* and *child* tasks

If a parent task inspects the result of a child task, the parent task blocks pending completion of the child task.

Blocking

- *Blocking* means a task cannot continue
  - So the task stops running
- Blocking does not prevent independent work from being done
  - If the processor has something else to do
- But it does prevent the thread from continuing and launching more tasks

Examples 4 & 5

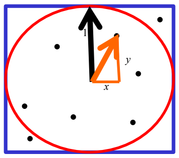
- "for all" style parallelism
- Note the order of completion of the tasks
  - main() finishes first (or almost first)!
  - All subtasks managed by the runtime system
  - Subtasks execute in non-deterministic order
- How?
  - Regent notices that the tasks are *independent*
  - In 4, no task depends on another task for its inputs

Runtime Dependence Analysis

- Example 5 is more involved
  - Positive tasks (print a positive integer)
  - Negative tasks (print a negative integer)
- Some tasks are dependent
  - The task for -5 depends on the task for 5
  - Note loop in main() does *not* block on the value of j!
- Some are independent
  - Positive tasks are independent of each other
  - Negative tasks are independent of each other

Computing the Area of a Unit Circle

- A Monte Carlo simulation to compute the area of a unit circle inscribed in a square
- Throw darts
  - Fraction of darts landing in the circle = ratio of circle's area to square's area



Computing the Area of a Unit Circle

- Example 6
  - Slow!
  - Why?
- Example 7
  - Faster!

Leaf Tasks

- *Leaf tasks* call no other tasks
  - The “leaves” of the task tree
- Leaf tasks are sequential programs
  - And generally where the heavy compute will be
- Thus, leaf tasks should be optimized for latency, not throughput
  - Want them to finish as fast as possible!

Terra

- Terra is a low-level, typed language embedded in Lua
- Designed to be like C
  - And to compile to similarly efficient code
- Also supports vector intrinsics
  - Not illustrated today

Considerations in Writing Regent Programs

- The granularity of tasks must be sufficient
  - Don’t write very short running tasks
- Don’t block in tasks that launch many subtasks
- Heavy sequential computations should be written as Terra functions

Profiling

- Is the performance any good?
  - You need to profile the code to find out
- Learn to use legion_prof
  - And use it early!
- Example 5 again …

Making Improvements

- If you don't like the profile, what can you do?
- Change the program
  - Remove dependencies that cause control tasks to block
  - Rewrite slow leaf tasks as Terra functions
- Next time
  - Improve memory/communication use
  - Change the *mapping*

Mapping

- Mapping is
  - The assignment of tasks to cores
  - The assignment of data to memories
  - … and many other policy decisions …
- Mapping is under programmer control
  - Completely programmable
- Programs use the *default mapper* if no other mapper is supplied.
- More on mapping next week …