Explicit Parallel Programming in Regent

CS315B
Lecture 9

Implicit Parallel Programming Template

```
while (…) do
  for R in Parts do
      task1(R)
  end
  for R in Parts do
      task2(R)
  end
end
```

How Do We Scale This Program?

```
while (…) do
  for R in Parts do
      task1(R)
  end
  for R in Parts do
      task2(R)
  end
end
```

- Make more Parts
- Make each subregion R smaller

Amdahl Strikes Back

- Recall Amdahl's law
  - Parallel speedup is limited by the sequential portion left un-parallelized
  - There is some sequential overhead to launching tasks on a single processor
- If we double the # of subregions
  - Each subregion is $\frac{1}{2}$ the size, so <= $\frac{1}{2}$ of the work
  - Launch overhead doubles
  - Useful compute/overhead ratio decreases by >= 4X

What Does That Mean?

```
while (…) do
  for R in Parts do
      task1(R)
  end
  for R in Parts do
      task2(R)
  end
end
```

- Can scale this program to 8 or 16 nodes
  - Should be more, but…
- We want to run on 100's or 1,000's of nodes

SPMD Programming Revisited

- Recall that SPMD programs
  - Launch 1 task per processor at program start-up
  - These tasks run for the duration of the program
  - Tasks explicitly communicate to exchange data
- Notice
  - SPMD programs launch the minimum # of tasks to keep the machine busy
  - These tasks run for the maximum amount of time
  - Best possible launch overhead/work ratio!

The Price

- SPMD programs minimize distributed overheads related to control
- The price is explicit parallel programming
  - Programmer must manage data transfers and synchronization

Data Transfers

- Simultaneous coherence
  - Two+ tasks share a region
    - With a single "master" instance
  - Both will read & write
- Acquire/Release
  - Acquire permits copies to be made
  - Release flushes updates back to "master" instance

Synchronization: Phase Barriers

- A task can *arrive* at a phase barrier
  - A barrier has a declared/expected # of arrivers
  - The barrier is *triggered* when all arrivers arrive
- A task can *await* a phase barrier
  - Block until the phase barrier triggers
- A task can *advance* a phase barrier
  - More on this shortly …

Phase Barriers: Await and Arrive

- Designed to support phases of computation
- To separate phase A and B for tasks 1 and 2:
  - Phase barrier P with 2 arrivers
  - Tasks 1 and 2 execute A
  - Tasks 1 and 2 arrive on phase 0 of P
  - Tasks 1 and 2 advance to phase 1 of P
  - Tasks 1 and 2 await (on start of) phase 1 of P
  - Tasks 1 and 2 execute B

Comments

- Note that arrive is non-blocking
  - Task continues executing
- An arrive followed immediately by await is closest to a traditional MPI barrier
  - All tasks stay at the await until all tasks reach the arrive
  - But typically phase barriers are used for pairs of tasks, not all tasks in the program

What is a Phase Barrier?

- A phase barrier is an unbounded sequence of *phases*
- Each phase has its own identity
  - An expected number of arrivers
  - An arbitrary number of awaiters
  - When the required # of arrivers have all arrived, the phase ends

About Advance

- A phase barrier has two components
  - A shared, global object, always in a specific phase
  - A component local to a task, holding the name of some phase
- Advance changes the local component
  - Refers to the nest phase
  - For that task only
- Arrive affects the global state
- Recall that Legion/Regent like to run ahead
  - Schedule tasks well into the future
  - Advance supports deferred execution

phase_barrier_immediate.rg

- Again,
  - advance gives the name of the next phase
  - await waits for the *start* of the phase
  - arrive contributes towards the *end* of the current phase

Must Parallelism

- Explicitly parallel programs require *must* parallelism
  - All tasks must be given resources and be able to run simultaneously
  - Otherwise the program may deadlock
  - E.g., because an arriver is not scheduled
- So far, we have only seen *may* parallelism
  - It's OK for tasks to run in parallel, but not required

Must Epoch Launch

```
must_epoch
  task1(…)
  task2(…)
end
```

- Evaluate the body of the must_epoch launch
- Gather all task calls until end
- Launch all tasks in the must_epoch simultaneously
  - Abort if there are insufficient resources

acquire.rg

- In language/tests/regent/run_pass
- Note that
  - awaits can be used as a precondition on acquire
  - arrives can be used as a postcondition on release
- acquire does not affect mapping
  - But calls to tasks on an acquired region can create new instances

copy.rg

- One can also perform explicit copies between regions
  - Regions must have the same index and field spaces
- copy(r,s) => copy region r to region s
- Copies can optionally include a reduction operator copy(r,s,+)
  - Value of one region is folded in to the other

copy_phase_barrier.rg

- Non-trivial use of advance
- Awaits used as a task precondition (preferred!)
- No use of acquire/release

- no_access_flag(r)
  - Declares the task will not access data in r
    - But subtasks may
  - Normally not needed
    - Only required here because of simultaneous coherence on the region
    - Tells the runtime that this task should not be the one to allocate the single master version of the region

Summary

- To write explicitly parallel SPMD programs in Regent, must use a combination of

- must_epoch launches
  - To generate simultaneously executing & communicating tasks
- Phase barriers
  - To synchronize between tasks in deferred execution style
- Simultaneous coherence
  - To allow simultaneous read/write access among tasks
  - Use acquire/release to relax the coherence (make copies)
- Explicit copies

How Do We Scale This Program?

```
while (...) do
  for R in Parts do
      task1(R)
  end
  for R in Parts do
      task2(R)
  end
end
```

```
must_epoch
  for i = 1,num_tasks do
    task(part[i],phaseb[i])
end
```

where

tasks know which other tasks they have to communicate with

Control Replication

- Regent can do this for you!
- __demand(_spmd)
- Takes a program in implicit parallel style, converts it to SPMD style
- Restrictions
  - Task launches must have the same index space
  - Regions cannot be allocated/deallocated

Control Replication

- We recommend using control replication for your project
  - Write in implicit style
- Should scale to 256-512 nodes
  - At least