Pro Regent Tips

Wonchan Lee
CS315B Guest Lecture
Nov 17th, 2016
Parallelize Your Initialization

```plaintext
```task toplevel()
    var cells = region(...)
    var p_cells = partition(equal, cells, ...)

    initialize_cells(cells)
    for steps = 0, 10000 do
        for c in p_cells.colors do
            do_something(p_cells[c])
        end
    end
end
```

This is not good

Runtime is smart enough to never make the instance for `cells`

```plaintext
```task toplevel()
    var cells = region(...)
    var p_cells = partition(equal, cells, ...)

    for c in p_cells.colors do
        initialize_cells(p_cells[c])
    end
    for steps = 0, 10000 do
        for c in p_cells.colors do
            do_something(p_cells[c])
        end
    end
end
```

This is better

Can't run this when `cells` is bigger than the memory on a node
Never Inline Access Big Regions

```plaintext
task toplevel()
    var cells = region(....)
    var p_cells = partition(equal, cells, ....)
    map(tokens)
    for e in p_cells[0] do @e = 0 end
    unmap(tokens)
    for c in p_cells.colors do
        do_something(p_cells[c])
    end
end
```

This is not good

Will request an instance for cells
Be Careful about File I/O

```plaintext
task dump(r : region(...))
where
  reads(r)
do
  var fd = fopen("dump.dat", "a")
  ...
end

task toplevel()
  ...
  for color in p.colors do
    dump(p[color])
  end
  ...

Will Crash!

regent
```

```plaintext
task dump(r : region(...))
where
  reads(r)
do
  var fd = fopen("dump.dat", "a")
  ...
end

task toplevel()
  ...
  dump(r)
  ...

Never Request Root
```

```plaintext
task dump(r : region(...), token : int)
where
  reads(r)
do
  var fd = fopen("dump.dat", "a")
  ...
  return 1
end

task toplevel()
  var token = 1
  ...
  for color in p.colors do
    token = dump(p[color], token)
  end
  ...

One Solution
```

regent
• Introduce spurious dependencies to ensure termination

```c
task block_task(r_image : region(ispace(int2d), Pixel))
where
  reads writes(r_image)
do
  return 1
end

task toplevel()
...
  var token = initialize(r_image, config.filename_image)
  wait_for(token)
  var ts_start = c.legion_get_current_time_in_micros()
  ...
  for color in p_private.colors do
    token += block_task(p_private[color])
  end
  wait_for(token)
  var ts_end = c.legion_get_current_time_in_micros()
  ...
```
Give Most Precise Privilege

Will unnecessarily create a reduction instance for all fields

```regent
task increment(r : region(...))
  where
    reduces+(r)
  do
    for e in r do
      e.f += 1
    end
  end

This is not good
```

```regent
task increment(r : region(...))
  where
    reduces+(r.f)
  do
    for e in r do
      e.f += 1
    end
  end

This is better
```

```regent
task increment(r : region(...))
  where
    reads writes(r.f)
  do
    for e in r do
      e.f += 1
    end
  end

Even better if possible
```
Avoid Returning Pointers from Tasks

create_array and toplevel can run on different nodes

create_array and toplevel will always run on the same node
Compact Your Unstructured Regions

- Overheads due to poor locality in
  - Iterator
  - Stride copies
  - Physical instances
Control Your Pointer Allocation

- Allocate one pointer at a time
- Dynamically cast integers to pointers

```plaintext
task by_new(r : region(some_fs))
where ...
do
  for i = 0, 100 do
    var p = new(ptr(some_fs, r))
    if i < 10 then p.boundary = true
       else p.boundary = false end
  end
end

task by_casting(r : region(some_fs))
where ...
do
  for i = 0, 100 do
    var p = dynamic_cast(ptr(some_fs, r), i)
    regentlib.assert(not isnull(p), "invalid pointer")
    if i < 10 then p.boundary = true
       else p.boundary = false end
  end
end
```
But... Dynamic Casting is Expensive

- Dynamic casting involves a runtime inclusion check
  - Use it only for debugging

```cpp
task initialize_graph(...) where
  reads writes(r_pages, r_links)
do
  ...
  var page_ids : uint32[2]
  for link in r_links do
    ...
    var src_page = unsafe_cast(ptr(Page, r_pages), page_ids[0])
    var dst_page = unsafe_cast(ptr(Page, r_pages), page_ids[1])
  ...
```

Graph initialization took 0.19s for smallrmat.dat

```cpp
var src_page = dynamic_cast(ptr(Page, r_pages), page_ids[0])
var dst_page = dynamic_cast(ptr(Page, r_pages), page_ids[1])
```

0.72s with dynamic casting
If your code died and you don't know why...

- This means you got a segfault:

  0  terra (JIT) 0x00000000000287db7 $<rank_page> + 1223

- Turn on bounds checks (flag `-fbounds-checks 1')

  pagerank_edge.rg:70: pointer ptr(Page(), $r_src_pages) is out-of-bounds

- Make sure you have no other invalid pointer accesses
If your code died and you don't know why...

- This means you got a segfault:
  
  ```
  0   terra (JIT)                         0x000000000d287db7 $<rank_page> + 1223
  ```

  - Turn on bounds checks (flag '-fbounds-checks 1')

  ```
  pagerank_edge rg:70: pointer ptr(Page(), $r_src_pages) is out-of-bounds
  ```

  - Make sure you have no other invalid pointer accesses

- This probably means you have a type error:

  ```
  ...objects/legion/language/src/rdir/plugin/src/regent/flow.t:243: assertion failed!
  stack traceback:
  [C]: in function 'assert'
  ...objects/legion/language/src/rdir/plugin/src/regent/flow.t:243: in function 'add_edge'
  ...gion/language/src/rdir/plugin/src/regent/flow_from_ast.t:406: in function 'add_input_edge'
  ...
  ```

  - Turn off flow analysis (flag '-fflow 0') and see if you get a better error message

  - Unfortunately, delta debugging is the best way here
Pretty Printer is More Useful than You'd Expect

- You can pretty print your code with '-fpretty 1'
- Pretty printer shows most of optimization results
Legion Prof is Your Friend

Live Demo
Legion Spy can be useful

• Use it for a small problem
  - Large graphs won't even be rendered

• Some useful flags
  - Flag '-d': Generate dataflow graph
  - Flag '-e': Generate event graph
  - Flag '-z': Make graphs verbose
  - Flag '-r': Generate region tree
• Any binary with a C header can be used

• Use `__physical` and `__fields` to pass raw pointers to regions
  - `__physical`: `(logical region) \times (field) \rightarrow (physical region)`
  - `__fields`: `(logical region) \times (field) \rightarrow (field id)`
  - `legion_physical_region_get_field_accessor_generic`: `(physical region) \times (field id) \rightarrow (accessor)`
  - `legion_accessor_generic_raw_rect_ptr_Nd`: `(accessor) \times (rectangle) \rightarrow (raw pointer)`

• Parallel Cholesky code passes raw pointers to BLAS functions in Fortran

• Stencil example even compiles a C code on the fly and load it
Questions?
Liszt-Regent

Wonchan Lee
CS315B Guest Lecture
Nov 17th, 2016
Lisztt

• A DSL (Domain Specific Language) for physical simulations
  - Embedded in the Lua language

• Runtime for parallel execution on both CPUs and GPUs
  - Via the Terra code with the Legion C API
Relational Data Model

- Data is store in a relation
  - Each element has several fields
  - Similar to Regent's logical region except fields are added dynamically

```lua
-- Create a new relation
local num_particles = 10
local particles = L.NewRelation {
    name = 'particles', size = num_particles,
}

-- Create and populate fields
particles:NewField('norm', L.double):Load(0.0)
local velF = particles:NewField('velocity', L.vec3f)
velF:Load(function(i) return {rand(), rand(), rand()} end)
```

- Or, in a multi-dimensional grid

```lua
-- Create a new grid
local grid = L.NewRelation {
    name = 'grid', dims = {100, 100},
}

grid:NewField('f', L.double):Load(0.0)
```
Data Parallel Kernels

- Kernels describe per-element computation
  - Take only a single relation as an argument
  - Should be data parallel

```ebb
-- Define a new kernel
ekern calculate_norm(p : particles)
   p.norm = sqrt(L.dot(p.velocity, p.velocity))
end

-- Run the kernel for all particles in parallel
particles:foreach(calculate_norm)
```
Global Variables

- Kernels can access global variables
  - Only read or reduction accesses are allowed

```
local weight = L.Global('count', L.double, 1.0 / N)
local avg   = L.Global('avg',   L.double, 0.0)

ebb kernel avg_norm(p : particles)
  avg += weight * sqrt(L.dot(p.velocity, p.velocity))
end

particles:foreach(avg_norm)
```
Coupling Relations

- Use relations as field types
  - Similar to region pointers in Regent

```lisp
local vertices = L.NewRelation { ... }
local edges    = L.NewRelation { ... }

vertices:NewField('pos', L.vec3f):Load(...)
edges:NewField('head', vertices):Load(...)
edges:NewField('tail', vertices):Load(...)

ebb kernel diff_pos(e : edges)
  var diff = e.tail.pos - e.head.pos
end
```

- Use macros to join two relations

```lisp
vertices:NewFieldMacro('edges', L.Macro(function(v)
  return ebb `L.Where(edges.tail, v)
end))
edges:GroupBy('tail')

ebb kernel visit_neighbors(v : vertices)
  for e in v.edges do
    var neighbor = e.head
  end
end
```
Uncentered Accesses and Subsets

- Grids support uncentered accesses in affine form

```lua
local grid = L.NewRelation { name = 'grid', dims = {100, 100} }
grid:NewField('f', L.double):Load(...)

grid:NewFieldMacro('__apply_macro', L.Macro(function(p, x, y)
    return ebb.L.Affine(grid, {{1, 0, x},
     {0, 1, y}}, p)
end))

ebb kernel three_point_stencil(p : grid)
    var avg = (p(0, 0).f + p(1, 0).f + p(-1, 0).f) / 3
end
```

- Grids can have named subsets
  - Similar to Regent's logical partition with single subregion

```lua
grid:NewSubset('interior', { {1, 98}, {1, 98} })
grid.interior:forall(three_point_stencil)
```
Domain Libraries

- Liszt provides libraries for widely used geometric domains
  - Triangular/Tetrahedral mesh

```lua
-- Load a tetrahedral mesh from a Vega mesh file
local Tetmesh = L.require 'domains.tetmesh'
local dragon = Tetmesh.Load('dragon.veg')

-- The mesh already has the 'edges' and 'vertices' relations,
dragon.edges:NewField('length', L.float):Load(...)
dragon.vertices:NewField('mass', L.float):Load(...)

-- and also some useful macros
ebb kernel calculate_force(v : dragon.vertices)
  for e in v.edges do ... end
  for n in v.neighbors do ... end
end
```

- 2D/3D Grid

```lua
local Grid = L.require 'domains.grid'
local grid = Grid.NewGrid3d {
  size = { 100, 100, 100 },
  origin = { 0, 0, 0 },
  width = { 2.0, 4.0, 2.0 },
  boundary_depth = { 0, 1, 0 },
  periodic_boundary = { true, false, true },
}

-- 'grid' already has the 'cells' and 'vertices' relations.
-- 'grid.cells' has two subsets, 'interior' and 'boundary'.
```
Phase Analysis

- Infers "phase" of each field and global access
  - Similar concept as privileges in Regent, but inferred by compiler

```ebb
ebb kernel avg_norm(p : particles)
  avg += weight * sqrt(L.dot(p.velocity, p.velocity))
end
```

- Raises an error if phases are incompatible
  - No write phases to globals

```ebb
ebb kernel avg_norm(p : particles)
  avg = weight * sqrt(L.dot(p.velocity, p.velocity))
end
```

- No uncentered writes

```ebb
ebb kernel shift(p : grid)
  p(1, 0).f = p(0, 0).f
end
```
Control Statements

- Reuses control statements in Lua

```lua
local initial_f = L.Global('initial_f', L.double, 0.0)

ebb kernel initialize(v : vertices)
    v.f = initial_f
end

ebb kernel visit_neighbors(v : vertices)
    for e in v.edges do ...
end
end

while i < 1000 do
    if i % 100 == 0 then
        initial_f:set(i / 100)
        vertices:foreach(initialize)
    end
    vertices:foreach(visit_neighbors)
    i = i + 1
end
```
Benefits of Liszt

- Sequential, but implicitly parallel
- Code specialization for a particular domain
- Code generation for both CPU and GPU
Issues in Liszt

• Compiler and runtime should take care of data partitioning and parallelization
  - Joins and arbitrary affine accesses are hard to parallelize
  - Kernels can change at runtime

• Inherent sequential bottleneck due to interpretation
Issues in Liszt

• Compiler and runtime should take care of data partitioning and parallelization
  - Joins and arbitrary affine accesses are hard to parallelize
  - Kernels can change at runtime

• Inherent sequential bottleneck due to interpretation

Soleil-X (Liszt-Legion)

Runtime overhead is growing!
Liszt-Regent Translation

- Regent already has nice features to reuse
  - Partitioning primitives
  - **SPMD transformation**
  - CPU/GPU code generation

- Translate a useful subset of Liszt to Regent
  - Target application: **Soleil-X**
Liszt-Regent Translation

• Regent already has nice features to reuse
  - Partitioning primitives
  - **SPMD transformation**
  - CPU/GPU code generation

• Translate a useful subset of Liszt to Regent
  - Target application: Soleil-X

```
Liszt code
  ↓
Serial Regent code
  ↓
Parallel Regent code
  ↓
SPMD Regent code
```
Soleil-X

- Multi-physics simulation for next generation compute systems

Fluid Flow

Particle

3D stencil on structured grid

unstructured, coupled with grid

3D sweep on structured grid

Written in Liszt
Liszt → Sequential Regent

- Record Liszt API calls instead of running them
- Meta-program Regent code

```lisp
local grid = Grid.NewGrid3d { ... }
grid.cells:NewField('pressure', L.double):Load(0)
sumPressure = L.Global('sumPressure', L.double, 0.0)
local ebb sum ( c : grid.cells )
  sumPressure += c.pressure
end

function Integrate(cells)
  cells:foreach(sum)
end
Integrate(grid.cells)

-- Finish recording and translate and run the code
A.translateAndRun()
```
Liszt ➔ Sequential Regent

- Record Liszt API calls instead of running them
- Meta-program Regent code

```liszt
local grid = Grid.NewGrid3d { ... }
grid.cells:NewField('pressure', L.double):Load(0)
sumPressure = L.Global('sumPressure', L.double, 0.0)
local ebb sum ( c : grid.cells )
  sumPressure += c.pressure
end

function Integrate(cells)
cells:foreach(sum)
end
Integrate(grid.cells)

-- Finish recording and translate and run the code
A.translateAndRun()
```

```regent
fspace columns
{
  pressure : double,
}

task sum(cells : region(ispace(int3d), columns))
where reads(cells.pressure)
do
  var acc : double = 0.0
  for c in cells do
    acc += c.pressure
  end
  return acc
end

task toplevel()
  var sumPressure : double = 0.0
  var cells = region(ispace(int3d, ...), columns)
  fill(cells.pressure, 0)
  var t = sum(cells)
  sumPressure += t
end
```
• Translate control statements manually when
  - The condition depends on a runtime value

  ```liszt
  while simTime:get() < final_time do
    ...
    simTime:set(simTime:get() + dt:get())
  end
  ```

  ```liszt
  M.WHILE(M.LT(simTime:get(), final_time))
  ...
  simTime:set(simTime:get() + dt:get())
  M.END()
  ```

  - Or, unrolling them increases the code size significantly

  ```liszt
  while rk_stage < 4 do
    -- RK sub-step with many task calls
    ...
    rk_stage = rk_stage + 1
  end
  ```

  ```liszt
  M.WHILE(M.LT(rk_stage:get(), 4))
  -- This loop body won't be unrolled
  ...
  rk_stage:set(rk_stage:get() + 1)
  M.END()
  ```
Sequential Regent $\rightarrow$ Parallel Regent

- Infer partitions
- Change single task launch to index launch

```regent
task toplevel()
  var sumPressure : double = 0.0
  var cells = region(ispace(int3d, ...), columns)
  fill(cells.pressure, 0)
  var t = sum(cells)
  sumPressure += t
end
```

Before Parallelization

Are we done?

```regent
task toplevel()
  var sumPressure : double = 0.0
  var cells = region(ispace(int3d, ...), columns)
  var p_cells = partition(equal, cells, ...)
  fill(cells.pressure, 0)
  var t = 0.0
  for c in p_cells.colors do t += sum(p_cells[c]) end
  sumPressure += t
end
```

After Parallelization
You Forgot Your Stencils, Bro

- Infer **ghost partitions**
- **Change leaf tasks** to use ghost regions for stencil accesses
- Change single task launch to index launch

```plaintext

task stencil(r : region(ispace(int3d), xy))
where reads(r.x), writes(r.y) do
  for e in r do
    e.x = (r[e + {1, 0, 0}].y + r[e + {-1, 0, 0}].y) / 2
  end
end

task toplevel()
  var r = region(ispace(int3d, ...), xy)
  stencil(r)
end
```

Stencil Code
You Forgot Your Stencils, Bro

- Infer **ghost partitions**
- **Change leaf tasks** to use ghost regions for stencil accesses
- Change single task launch to index launch

```plaintext
task stencil(r : region(ispace(int3d), xy))
where reads(r.x), writes(r.y) do
  for e in r do
    e.x = (r[e + {1, 0, 0}].y + r[e + {-1, 0, 0}].y) / 2
  end
end
task toplevel()
  var r = region(ispace(int3d, ...), xy)
  stencil(r)
end
```

How can we do these?
Inferring Ghost Partitions

Stencil Code

We need two ghost partitions

Ghost Partitions

```plaintext
task stencil(r : region(ispace(int3d), xy))
where reads(r.x), writes(r.y) do
  for e in r do
    e.x = (r[e + {1, 0, 0}].y + r[e + {-1, 0, 0}].y) / 2
  end
end
```

```plaintext
task toplevel()
  var r = region(ispace(int3d, ...), xy)

  -- Primary partition
  var p = partition(equal, r, ...)

  var coloring1 = create()
  for c in p.colors do
    color(coloring1, c, (p[c].bounds + {1, 0, 0}) - p[c].bounds)
  end
  var g1 = partition(aliened, r, coloring1)

  var coloring2 = create()
  for c in p.colors do
    color(coloring2, c, (p[c].bounds + {-1, 0, 0}) - p[c].bounds)
  end
  var g2 = partition(aliened, r, coloring2)
end
```
Stencil Analysis for Multiple Tasks

Task A

Task B

Task C

Merged stencil

Primary partition

Ghost partition 1

Ghost partition 2
Changing Leaf Tasks

```plaintext
task stencil(r : region(ispace(int3d), xy))
where reads(r.x), writes(r.y) do
  for e in r do
    e.x = (r[e + {1, 0, 0}].y + r[e + {-1, 0, 0}].y) / 2
  end
end

Stencil Code
```

```plaintext
task stencil(r : region(ispace(int3d), xy),
g1 : region(ispace(int3d), xy),
g2 : region(ispace(int3d), xy))
where reads(r.x, g1.x, g2.x), writes(r.y) do
  for e in r do
    var idx1 = e + {1, 0, 0}
    var v1
    if idx1 <= r.bounds then v1 = r[idx1].y else v1 = g1[idx1].y end
    var idx2 = e + {-1, 0, 0}
    var v2
    if idx2 <= r.bounds then v2 = r[idx2].y else v2 = g2[idx2].y end
    e.x = (v1 + v2) / 2
  end
end

Stencil Code
```
Multi-node Performance

Excalibur
33.6M cells per node
16+2 cores per node

Throughput per node
Efficiency

Weak Scaling

Excalibur
134.2M cells
16+2 cores per node

Strong Scaling

Throughput
Speedup
Multi-GPU Performance

Sapling
2.1M cells
2 runtime cores

Throughput (Mcells/s)

13X speed up

12.793

8 CPUs
1 GPU
2 GPUs