Regent: Optimizing Implicitly Parallel Programs for Efficient Execution on Distributed Memory Machines

Elliott Slaughter
Heterogeneous Heterogeneity

Titan
~20,000

Aurora
~50,000

Trinity / Cori
~10,000

Summit
~3,500

November 29, 2016

http://regent-lang.org
Legion Runtime [Bauer et al.]

Key idea: system understanding of control & data

Programmer responsibilities:
- Divide control flow into tasks (functions)
- Divide data into regions (collections of objects)

System provides:
- Sequential semantics, implicit parallelism
- Automatic discovery of dependencies from program order + task arguments
- Automatic data movement
Example: PENNANT

-- Main simulation loop:
for i = 0, 3 do
    calc_forces(
        p_sides[i], points)
end
for i = 0, 3 do
    adv_pos_full(p_points[i])
end

-- Tasks:
task calc_force(sides, points)
where reduces +(points.f) do
    ...
end
task adv_pos_full(points)
where reads(points.{f, m, u0}), writes(points.u) do
    ...
end

dependence graph

http://regent-lang.org
Legion C++ API

You don’t get to write this code…

```markdown
for i = 0, 3 do
    calc_forces(p_sides[i], points)
end
for i = 0, 3 do
    adv_pos_full(p_points[i])
end
```

Instead you have to write this:

```c
runtime->unmap_region(ctx, physical_points);
runtime->unmap_region(ctx, physical_sides);
Domain domain = Domain::from_rect<1>(
    Rect<1>(Point<1>(0), Point<1>(2)));
IndexLauncher launcher_CF(TASK_CF, domain, 
    TaskArgument(), ArgumentMap());
launcher_CF.add_region_requirement(  
    RegionRequirement(p_sides, 0 /* projection */,  
    READ_WRITE, EXCLUSIVE, points));
launcher_CF.add_region_requirement(  
    RegionRequirement(points,  
    REDUCE_PLUS_VEC2, EXCLUSIVE, points));
launcher_CF.add_field(1, FIELD_F);
runtime->execute_index_space(ctx, launcher_CF);  
IndexLauncher launcher_AP(TASK_AP, domain, 
    TaskArgument(), ArgumentMap());
launcher_AP.add_region_requirement(  
    RegionRequirement(p_points, 0 /* projection */,  
    READ_WRITE, EXCLUSIVE, points));
launcher_AP.add_region_requirement(  
    RegionRequirement(points, 0 /* projection */,  
    REDUCE_ONLY, EXCLUSIVE, points));
launcher_AP.add_field(0, FIELD_U);
launcher_AP.add_region_requirement(  
    RegionRequirement(p_points, 0 /* projection */,  
    READ_ONLY, EXCLUSIVE, points));
launcher_AP.add_field(1, FIELD_M);
launcher_AP.add_field(1, FIELD_U0);
runtime->execute_index_space(ctx, launcher_AP);
runtime->map_region(ctx, physical_points);
runtime->map_region(ctx, physical_sides);
```

No compile-time checks for correctness

Needs manual optimizations for acceptable performance

And this code still won’t scale (scalable version is longer and no longer has sequential semantics)

http://regent-lang.org
Regent Programming Language

Make this code actually work:
for i = 0, 3 do
  calc_forces(p_sides[i], points)
end
for i = 0, 3 do
  adv_pos_full(p_points[i])
end

Regent is a language for implicit parallelism
Simplifies the Legion programming model
Enables compile-time checks and static analysis
Optimizes automatically for best performance:
  - On-node performance
  - Scalability to large numbers of nodes*

*Expanding the set of programs that can achieve high performance on large machines
Part 1
Lowering Optimizations
Execution of an Unoptimized Program

```
for i = 0, 3 do
    calc_forces(…, points)
end
for i = 0, 3 do
    adv_pos_full(p_points[i])
end
…
```
Mapping: Unoptimized

for \(i = 0, 3\) do
  \texttt{unmap(points)}
  \texttt{calc_forces(\ldots, points)}
  \texttt{map(points)} \textit{-- blocks}
end

for \(i = 0, 3\) do
  \texttt{unmap(points)}
  \texttt{adv_pos_full(p_points[i])}
  \texttt{map(points)}
end

dependence graph

\begin{center}
\begin{tikzcd}
cf & cf & cf
\end{tikzcd}
\end{center}
Mapping: Optimized

definition:

unmap(points)
for i = 0, 3 do
    calc_forces(…, points)
end
for i = 0, 3 do
    adv_pos_full(p_points[i])
end
map(points) -- blocks
Leaf Tasks: Unoptimized

execution timeline

<table>
<thead>
<tr>
<th>time</th>
<th>app thread</th>
<th>runtime thread</th>
<th>app thread</th>
<th>app thread</th>
<th>app thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>don’t know until here</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>analysis delayed</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>how many subtasks?</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

unmap(points)
for i = 0, 3 do
  calc_forces(…, points)
end
for i = 0, 3 do
  adv_pos_full(p_points[i])
end
map(points) -- blocks

dependence graph

cf → cf → cf
ap
Leaf Tasks: Optimized

unmap(points)
for i = 0, 3 do
  calc_forces(…, points)
end
for i = 0, 3 do
  adv_pos_full(p_points[i])
end
map(points) -- blocks

dependence graph

http://regent-lang.org
Index Launches: Unoptimized

```
unmap(points)
for i = 0, 3 do
    calc_forces(..., points)
end
for i = 0, 3 do
    adv_pos_full(p_points[i])
end
map(points) -- blocks
```

dependence graph

```
app thread
runtime thread
app thread
app thread
app thread
```

November 29, 2016

http://regent-lang.org
Index Launches: Optimized

unmap(points)

for i = 0, 3: calc_forces(…, points)

for i = 0, 3: adv_pos_full(p_points[i])

map(points) -- blocks

dependence graph
Other Optimizations

- Futures
- Pointer Check Elision
- Dynamic Branch Elision
- Vectorization
- GPU Code Generation
Experimental Setup

Applications:
- Circuit simulation (unstructured graph)
- MiniAero: compressible Navier Stokes (3D unstructured)
- PENNANT: Lagrangian hydrodynamics (2D unstructured)

Machine:
- Certainty: Infiniband, 2x Xeon (2x6 cores), 36 GB per node
Impact of Optimizations

Circuit

GFLOPS

Zones per second (in millions)

none
all-idx-map
all-idx-leaf
all-idx
all-map
all-leaf
all-vec
all

best single-threaded performance

individual optimizations disabled

http://regent-lang.org

November 29, 2016
All Optimizations Enabled

execution timeline

unmap(points)

for i = 0, 3: calc_forces(…, points)

for i = 0, 3: adv_pos_full(p_points[i])

map(points) -- blocks
unmap(points)
for i = 0, 3 do
    calc_forces(…, points)
end
for i = 0, 3 do
    adv_pos_full(p_points[i])
end
map(points) -- blocks
All Optimizations Except Mapping

unmap(points)

for $i = 0, 3$: calc_forces(…, points)
map(points) -- blocks
unmap(points)
unmap(points)
for $i = 0, 3$: adv_pos_full(p_points[i])

map(points) -- blocks
for i = 0, 3 do
  unmap(points)
  calc_forces(..., points)
  map(points) -- blocks
end

for i = 0, 3 do
  unmap(points)
  adv_pos_full(p_points[i])
  map(points)
end
Initial Strong Scaling Experiments

**Circuit**

- GFLOPS vs. Total CPUs
- Regent vs. Legion

**PENNANT**

- Zones per second (in millions)
- Regent vs. OpenMP

**MiniAero (Single Node)**

- Cells per second (in millions)
- Regent vs. MPI+Kokkos

**MiniAero (Multi-Node)**

- Cells per second (in millions)
- Regent vs. MPI+Kokkos

Certainty supercomputer up to 8 nodes

is good
Problem: Scalability

Soleil-X (Legion implementation)

Runtime overhead is growing

All Regent applications exhibited similar behavior

http://regent-lang.org
Part 2
Control Replication
Launching tasks is a sequential bottleneck.

Task Launch Overhead

every factor of N in scale:

Nx more tasks

Nx overhead

strong scaling: 1/N work per node ratio: $O(N^2)$

weak scaling: same work per node ratio: $O(N)$
**Single Program Multiple Data (SPMD)**

<table>
<thead>
<tr>
<th>Execution Timeline</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>app thread</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Strong scaling:**
  - 1/N work per node
  - Ratio: $O(N)$

- **Weak scaling:**
  - Same work per node
  - Ratio: $O(1)$

**Every factor of N in scale:**
- $Nx$ more tasks
- Fixed overhead
Control Replication

- Control replication converts:
  - From implicit parallelism
    - With implicit data movement
  - To explicit SPMD-style parallelism
    - With explicit communication and synchronization
Execution with Control Replication

<table>
<thead>
<tr>
<th>execution timeline</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>app thread</td>
<td>runtime thread</td>
</tr>
<tr>
<td>app thread</td>
<td>app thread</td>
</tr>
<tr>
<td>app thread</td>
<td>app thread</td>
</tr>
<tr>
<td>app thread</td>
<td>app thread</td>
</tr>
<tr>
<td>app thread</td>
<td>runtime thread</td>
</tr>
<tr>
<td>app thread</td>
<td>app thread</td>
</tr>
<tr>
<td>app thread</td>
<td>app thread</td>
</tr>
<tr>
<td>app thread</td>
<td>app thread</td>
</tr>
<tr>
<td>app thread</td>
<td>runtime thread</td>
</tr>
<tr>
<td>app thread</td>
<td>app thread</td>
</tr>
<tr>
<td>app thread</td>
<td>app thread</td>
</tr>
</tbody>
</table>

http://regent-lang.org
Execution with Control Replication

execution timeline

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

time

| app thread | runtime thread | app thread | app thread | app thread | app thread | app thread | runtime thread | app thread | app thread | app thread | app thread | app thread | app thread | runtime thread | app thread | app thread |
Detecting Implicit Data Movement

Implicit data movement is the result of:

- A write to a region
- Followed by a read to an aliased region
Aliased Partitions

-- Main simulation loop:
for i = 0, 3 do
    calc_forces(p_sides[i], points[i])
end
for i = 0, 3 do
    adv_pos_full(p_master[i])
end
Multiple Partitions

-- Main simulation loop:
for i = 0, 3 do
  calc_forces(
    p_sides[i], p_ghost[i])
end

-- implicit data movement
for i = 0, 3 do
  adv_pos_full(p_master[i])
end

gather data to least common ancestor
scatter to destination subregions
Detecting Implicit Data Movement

Implicit data movement is the result of:
- A write to a region
- Followed by a read to an aliased region

Compile-time alias analysis is known hard problem
- Secret weapon: region tree

For any two nodes
- Find least common ancestor
- If region or aliased partition, then nodes are aliased
Program Transformation

Main loop before transformation:

while t < T do
    for i = 0, 3 do
        calc_forces(…, p_ghost[i])
    end
    for i = 0, 3 do
        adv_pos_full(p_master[i])
    end
    t += dt
end

Main loop after transformation:

task shard(…, LO, HI)
    while t < T do
        for i = LO, HI do
            calc_forces(…, p_ghost[i])
        end
        for i = LO, HI do
            adv_pos_full(p_master[i])
        end
        t += dt
    end
end
Program Transformation

Main loop before transformation:

```plaintext
while t < T do
    for i = 0, 3 do
        calc_forces(…, p_ghost[i])
    end
    -- implicit data movement
    for i = 0, 3 do
        adv_pos_full(p_master[i])
    end
    t += dt
end
```

Main loop after transformation:

```plaintext
task shard(…, LO, HI)
    while t < T do
        for i = LO, HI do
            calc_forces(…, p_ghost[i])
        end
        -- explicit communication required
        for i = LO, HI do
            adv_pos_full(p_master[i])
        end
        t += dt
    end
end
```
Main loop before transformation:

\[
\text{while } t < T \text{ do}
\]

\[
\text{for } i = 0, 3 \text{ do}
\]

\[
\text{calc_forces}(\ldots, p_{\text{ghost}}[i])
\]

\end{verbatim}

\end{verbatim}

\[
\text{end}
\]

\[
\text{for } i = 0, 3 \text{ do}
\]

\[
\text{adv_pos_full}(p_{\text{master}}[i])
\]

\end{verbatim}

\end{verbatim}

\[
\text{end}
\]

\[
t + dt
\]

\end{verbatim}

\end{verbatim}

Main loop after transformation:

\[
\text{task shard}(\ldots, \text{LO, HI})
\]

\[
\text{while } t < T \text{ do}
\]

\[
\text{for } i = \text{LO, HI} \text{ do}
\]

\[
\text{calc_forces}(\ldots, p_{\text{ghost}}[i])
\]

\end{verbatim}

\end{verbatim}

\[
\text{end}
\]

\[
\text{for } i = \text{LO, HI} \text{ do}
\]

\[
\text{for } j = 0, 3 \text{ do}
\]

\[
\text{copy}(p_{\text{ghost}}[i], p_{\text{master}}[j], +)
\]

\end{verbatim}

\end{verbatim}

\[
\text{end}
\]

\[
\text{end}
\]

\[
\text{for } i = \text{LO, HI} \text{ do}
\]

\[
\text{adv_pos_full}(p_{\text{master}}[i])
\]

\end{verbatim}

\end{verbatim}

\[
\text{end}
\]

\[
t + dt
\]

\end{verbatim}

\end{verbatim}

\end{verbatim}

end
Main loop before transformation:

```java
while t < T do
    for i = 0, 3 do
        calc_forces(…, p_ghost[i])
    end
    for i = 0, 3 do
        adv_pos_full(p_master[i])
    end
    t += dt
end
```

Main loop after transformation:

```java
for i = LO, HI do
    calc_forces(…, p_ghost[i])
end

for i = LO, HI do
    for j = 0, 3 do -- only for overlapping j
        copy(p_ghost[i], p_master[j], +)
    end
end

for i = LO, HI do
    adv_pos_full(p_master[i])
end

for i = LO, HI do
    adv_pos_full(p_master[i])
end

for i = LO, HI do
    adv_pos_full(p_master[i])
end

for i = LO, HI do
    adv_pos_full(p_master[i])
end

for i = LO, HI do
    adv_pos_forces(…, p_ghost[i])
end
end
```
Program Transformation

Main loop before transformation:

```pseudocode
while t < T do
  for i = 0, 3 do
    calc_forces(…, p_ghost[i])
  end
  for i = 0, 3 do
    adv_pos_full(p_master[i])
  end
  t += dt
end
```

Main loop after transformation:

```pseudocode
task shard(…, LO, HI)

while t < T do
  for i = LO, HI do
    calc_forces(…, p_ghost[i])
  end
  for i = LO, HI do
    for j = 0, 3 do -- only for overlapping j
      copy(p_ghost[i], p_master[j], +,
           awaits(b0[i][j]), arrives(b1[i][j]))
    end
  end
  b1 = advance(b1)
  for i = LO, HI do
    adv_pos_full(p_master[i],
                 awaits(b1[i]), arrives(b0[i]))
  end
  b0 = advance(b0)
end
```

http://regent-lang.org
Region-based Dataflow IR (RDIR)

- Need to capture implicit data movement
- RDIR: Novel dataflow representation
- Explicit support for regions and partitions
- Inputs/outputs to tasks are explicit data nodes
- Captures implicit data movement as gather/scatter nodes between regions
while $t < T$ do
  for $i = 0, 3$ do -- $Red+(p_{ghost})$
    calc_forces(…, $p_{ghost}[i]$) -- $Red+(p_{ghost}[i])$
  end
  for $j = 0, 3$ do -- $R(p_{master})$
    adv_pos_full($p_{master}[i]$) -- $R(p_{master}[i])$
  end
end

while $t < T$ do
  for $i = 0, 3$ do
    Red+($p_{ghost}$)
    gather
    $pts.f$
    scatter
    R
    $W$
    $R$
    $W$
    $R$
    $W$
  end
end
while $t < T$

```plaintext
for i = L, H
    full_in  R  advance  W  full_in
    Red+

RDIR: Control Replication

for j = L, H
    empty_in  R  advance  W  empty_in
    Signal

p_grh.f2  R  fill(0)  W  p_grh.f
          R  copy  R  p_ma.f
          Red+

full_out  R  advance  W  full_out
empty_out  R  advance  W  empty_out
```

http://regent-lang.org
Experimental Setup

Applications:
- Circuit simulation (unstructured graph)
- MiniAero: compressible Navier Stokes (3D unstructured)
- PENNANT: Lagrangian hydrodynamics (2D unstructured)
- Stencil: radius-2 star-shaped stencil (2D structured)
- Soleil-X: compressible Navier Stokes (3D structured)

Machines:
- Piz Daint: Cray XC30, 1x Xeon (8 cores), 32 GB per node
- Excalibur: Cray XC40, 2x Xeon (2x16 cores), 128 GB per node
Runtime Analysis Overhead (Soleil-X)

![Graph showing runtime analysis overhead for Liszt and Liszt with Regent on a supercomputer. The graph plots time (s) on the y-axis and nodes on the x-axis, with a linear increase in overhead as the number of nodes increases.]
Circuit: Weak Scaling

Throughput per node ($10^3$ wires/s)

- Regent (with CR)
- Regent (w/o CR)

Total Nodes

- Piz Daint supercomputer is good
- 96% parallel efficiency at 512 nodes
MiniAero: Weak Scaling

Regent (with CR)  Regent (w/o CR)  Regent (manual SPMD)  MPI+Kokkos

93% parallel efficiency at 512 nodes

Piz Daint supercomputer

is good
PENNANT: Weak Scaling

- Regent (with CR)
- Regent (w/o CR)
- MPI+OpenMP (7T)
- MPI+OpenMP (16T)

Throughput per node (10^6 zones/s) vs Total Nodes

- 76% parallel efficiency at 512 nodes
- 94% parallel efficiency at 128 nodes

Piz Daint supercomputer is good
Stencil: Weak Scaling

96% parallel efficiency at 512 nodes

Piz Daint supercomputer
Soleil-X: Weak Scaling

96% parallel efficiency at 256 nodes

DSL compiles to Regent

Excalibur supercomputer

is good
Conclusion

- Regent is a language for implicit parallelism
- Simplifies the Legion programming model
- Enables compile-time checks and static analysis
- Optimizes automatically for best performance:
  - On-node performance
  - Scalability to large numbers of nodes*

*Expanding the set of programs that can achieve high performance on large machines

http://regent-lang.org
Thank You

Questions?
# Related Work: Languages

<table>
<thead>
<tr>
<th>Paradigm</th>
<th>Discovery of Parallelism</th>
<th>Distrib.</th>
<th>Unstruct.</th>
<th>Partitioning</th>
<th>Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regent</td>
<td>Hierarchical Tasks &amp; Regions</td>
<td>Both</td>
<td>Yes</td>
<td>Yes</td>
<td>Multiple</td>
</tr>
<tr>
<td>DPJ</td>
<td>Hierarchical Tasks &amp; Regions</td>
<td>Static</td>
<td>No</td>
<td>Yes</td>
<td>Single</td>
</tr>
<tr>
<td>Sequoia</td>
<td>Hierarchical Tasks &amp; Arrays</td>
<td>Static</td>
<td>Yes</td>
<td>No</td>
<td>Multiple</td>
</tr>
<tr>
<td>X10</td>
<td>Hierarchical Tasks w/ Explicit Mapping</td>
<td>Explicit</td>
<td>Yes</td>
<td>Yes</td>
<td>Single</td>
</tr>
<tr>
<td>Jade</td>
<td>Implicit</td>
<td>Dynamic</td>
<td>Yes</td>
<td>Yes</td>
<td>None</td>
</tr>
<tr>
<td>Chapel</td>
<td>Data-parallel*</td>
<td>Static*</td>
<td>Yes</td>
<td>No*</td>
<td>Single*</td>
</tr>
<tr>
<td>HPF</td>
<td>Data-parallel</td>
<td>Static</td>
<td>Yes</td>
<td>No</td>
<td>Single</td>
</tr>
<tr>
<td>Uintah</td>
<td>Domain-Specific</td>
<td>Static</td>
<td>Yes</td>
<td>No</td>
<td>Single</td>
</tr>
<tr>
<td>I/E Comp.</td>
<td>Sequential</td>
<td>Dynamic</td>
<td>Yes</td>
<td>Yes</td>
<td>Single**</td>
</tr>
<tr>
<td>OMP-&gt;MPI</td>
<td>Shared Memory</td>
<td>Dynamic</td>
<td>Yes</td>
<td>Slow</td>
<td>Single</td>
</tr>
</tbody>
</table>

*Also supports low-level features for explicit distributed programming.
**Partitions loop iterations, not data.
#Supports consumer-computes via optimization.

http://regent-lang.org
# Related Work: Languages

<table>
<thead>
<tr>
<th>Paradigm</th>
<th>Discovery of Parallelism</th>
<th>Distrib.</th>
<th>Unstruct.</th>
<th>Partitioning</th>
<th>Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regent</td>
<td>Hierarchical Tasks &amp; Regions</td>
<td>Both</td>
<td>Yes</td>
<td>Multiple</td>
<td>Dynamic</td>
</tr>
<tr>
<td>DPJ</td>
<td>Hierarchical Tasks &amp; Regions</td>
<td>Static</td>
<td>No</td>
<td>Single</td>
<td>No</td>
</tr>
<tr>
<td>Sequoia</td>
<td>Hierarchical Tasks &amp; Arrays</td>
<td>Static</td>
<td>Yes</td>
<td>Multiple</td>
<td>Static</td>
</tr>
<tr>
<td>X10</td>
<td>Hierarchical Tasks w/ Explicit Mapping</td>
<td>Explicit</td>
<td>Yes</td>
<td>Single</td>
<td>Explicit</td>
</tr>
<tr>
<td>Jade</td>
<td>Implicit</td>
<td>Dynamic</td>
<td>Yes</td>
<td>None</td>
<td>No</td>
</tr>
<tr>
<td>Chapel</td>
<td>Data-parallel*</td>
<td>Static*</td>
<td>Yes</td>
<td>Single*</td>
<td>Owner-comp.*</td>
</tr>
<tr>
<td>HPF</td>
<td>Data-parallel</td>
<td>Static</td>
<td>Yes</td>
<td>Single</td>
<td>Owner-comp.</td>
</tr>
<tr>
<td>Uintah</td>
<td>Domain-Specific</td>
<td>Static</td>
<td>Yes</td>
<td>Single</td>
<td>No</td>
</tr>
<tr>
<td>I/E Comp.</td>
<td>Sequential</td>
<td>Dynamic</td>
<td>Yes</td>
<td>Single**</td>
<td>Graph Part.</td>
</tr>
<tr>
<td>OMP-&gt;MPI</td>
<td>Shared Memory</td>
<td>Dynamic</td>
<td>Yes</td>
<td>Single</td>
<td>Owner-comp.#</td>
</tr>
</tbody>
</table>

*Also supports low-level features for explicit distributed programming.

**Partitions loop iterations, not data.

#Supports consumer-computes via optimization.
## Related Work: Languages

<table>
<thead>
<tr>
<th>Paradigm</th>
<th>Discovery of Parallelism</th>
<th>Distrib.</th>
<th>Unstruct.</th>
<th>Partitioning</th>
<th>Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regent</td>
<td>Hierarchical Tasks &amp; Regions</td>
<td>Both</td>
<td>Yes</td>
<td>Yes</td>
<td>Multiple</td>
</tr>
<tr>
<td>DPJ</td>
<td>Hierarchical Tasks &amp; Regions</td>
<td>Static</td>
<td>No</td>
<td>Yes</td>
<td>Single</td>
</tr>
<tr>
<td>Sequoia</td>
<td>Hierarchical Tasks &amp; Arrays</td>
<td>Static</td>
<td>Yes</td>
<td>No</td>
<td>Multiple</td>
</tr>
<tr>
<td>X10</td>
<td>Hierarchical Tasks w/ Explicit Mapping</td>
<td>Explicit</td>
<td>Yes</td>
<td>Yes</td>
<td>Single</td>
</tr>
<tr>
<td>Jade</td>
<td>Implicit</td>
<td>Dynamic</td>
<td>Yes</td>
<td>Yes</td>
<td>None</td>
</tr>
<tr>
<td>Chapel</td>
<td>Data-parallel*</td>
<td>Static*</td>
<td>Yes</td>
<td>No*</td>
<td>Single*</td>
</tr>
<tr>
<td>HPF</td>
<td>Data-parallel</td>
<td>Static</td>
<td>Yes</td>
<td>No</td>
<td>Single</td>
</tr>
<tr>
<td>Uintah</td>
<td>Domain-Specific</td>
<td>Static</td>
<td>Yes</td>
<td>No</td>
<td>Single</td>
</tr>
<tr>
<td>I/E Comp.</td>
<td>Sequential</td>
<td>Dynamic</td>
<td>Yes</td>
<td>Yes</td>
<td>Single**</td>
</tr>
<tr>
<td>OMP-&gt;MPI</td>
<td>Shared Memory</td>
<td>Dynamic</td>
<td>Yes</td>
<td>Slow</td>
<td>Single</td>
</tr>
</tbody>
</table>

*Also supports low-level features for explicit distributed programming.
**Partitions loop iterations, not data.
#Supports consumer-computes via optimization.
## Related Work: Languages

<table>
<thead>
<tr>
<th>Paradigm</th>
<th>Discovery of Parallelism</th>
<th>Distrib.</th>
<th>Unstruct.</th>
<th>Partitioning</th>
<th>Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regent</td>
<td>Hierarchical Tasks &amp; Regions</td>
<td>Both</td>
<td>Yes</td>
<td>Yes</td>
<td>Multiple</td>
</tr>
<tr>
<td>DPJ</td>
<td>Hierarchical Tasks &amp; Regions</td>
<td>Static</td>
<td>No</td>
<td>Yes</td>
<td>Single</td>
</tr>
<tr>
<td>Sequoia</td>
<td>Hierarchical Tasks &amp; Arrays</td>
<td>Static</td>
<td>Yes</td>
<td>No</td>
<td>Multiple</td>
</tr>
<tr>
<td>X10</td>
<td>Hierarchical Tasks w/ Explicit Mapping</td>
<td>Explicit</td>
<td>Yes</td>
<td>Yes</td>
<td>Single</td>
</tr>
<tr>
<td>Jade</td>
<td>Implicit</td>
<td>Dynamic</td>
<td>Yes</td>
<td>Yes</td>
<td>None</td>
</tr>
<tr>
<td>Chapel</td>
<td>Data-parallel*</td>
<td>Static*</td>
<td>Yes</td>
<td>No*</td>
<td>Single*</td>
</tr>
<tr>
<td>HPF</td>
<td>Data-parallel</td>
<td>Static</td>
<td>Yes</td>
<td>No</td>
<td>Single</td>
</tr>
<tr>
<td>Uintah</td>
<td>Domain-Specific</td>
<td>Static</td>
<td>Yes</td>
<td>No</td>
<td>Single</td>
</tr>
<tr>
<td>I/E Comp.</td>
<td>Sequential</td>
<td>Dynamic</td>
<td>Yes</td>
<td>Yes</td>
<td>Single**</td>
</tr>
<tr>
<td>OMP-&gt;MPI</td>
<td>Shared Memory</td>
<td>Dynamic</td>
<td>Yes</td>
<td>Slow</td>
<td>Single</td>
</tr>
</tbody>
</table>

*Also supports low-level features for explicit distributed programming.

**Partitions loop iterations, not data.

#Supports consumer-computes via optimization.
Related Work: Runtimes

- Legion
- StarPU
- Charm++
- HPX
- Realm
- OCR
Regent Example: PENNANT

- Lagrangian hydrodynamics proxy app from LANL
- 2D unstructured mesh: zones, points, sides

![Diagram of zones, points, and sides](image.png)
Regent Example: PENNANT

- Lagrangian hydrodynamics proxy app from LANL
- 2D unstructured mesh: zones, points sides

![Diagram of 2D unstructured mesh with zones, points, and sides]
Regent Example: PENNANT

Main simulation loop:

```regent
while t < T do
    -- calculate forces
    for s in sides do
        s.p1.f += ...
        s.p2.f += ...
    end

    -- advect points
    for p in points do
        var a = p.f/p.m
        p.u = p.u0 + dt*a
    end

    t += dt
end
```

Tasks:

```regent
task calc_force(sides, points)
where reduces +(points.f) do
    for s in sides do
        s.p1.f += ...
        s.p2.f += ...
    end
end
```

```regent
task adv_pos_full(points)
where reads(points.{f, m, u0}),
      writes(points.u) do
    for p in points do
        var a = p.f/p.m
        p.u = p.u0 + dt*a
    end
end
```

Regent Example: PENNANT

zones

sides

points

Main simulation loop:

```regent
while t < T do
    -- calculate forces
    for s in sides do
        s.p1.f += ...
        s.p2.f += ...
    end

    -- advect points
    for p in points do
        var a = p.f/p.m
        p.u = p.u0 + dt*a
    end

    t += dt
end
```

Tasks:

```regent
task calc_force(sides, points)
where reduces +(points.f) do
    for s in sides do
        s.p1.f += ...
        s.p2.f += ...
    end
end
```

```regent
task adv_pos_full(points)
where reads(points.{f, m, u0}),
      writes(points.u) do
    for p in points do
        var a = p.f/p.m
        p.u = p.u0 + dt*a
    end
end
```
Main simulation loop:

```
while t < T do

    -- calculate forces
    calc_forces(
        sides,
        points)

    -- advect points
    adv_pos_full(
        points)

    t += dt

end
```

Tasks:

```
task calc_force(sides, points)
where reduces +(points.f) do
    for s in sides do
        s.p1.f += ...
        s.p2.f += ...
    end
end

task adv_pos_full(points)
where reads(points.{f, m, u0}),
    writes(points.u) do
    for p in points do
        var a = p.f/p.m
        p.u = p.u0 + dt*a
    end
end
```
Regent Example: PENNANT

Warning: Naive approach

Main simulation loop:

```
while t < T do
    -- calculate forces
    calc_forces(sides, points)

    -- advect points
    adv_pos_full(points)

    t += dt
end
```

http://regent-lang.org
Regent Example: PENNANT

Warning: Naive approach

Main simulation loop:

while $t < T$ do
  for $i = 0, 3$ do
    calc_forces($p_{\text{sides}}[i], p_{\text{ghost}}[i]$)
  end

  for $i = 0, 3$ do
    adv_pos_full($p_{\text{master}}[i]$)
  end

  $t += dt$
end

http://regent-lang.org
Regent Example: PENNANT

Warning: Naive approach

Main simulation loop:

```java
while t < T do
    for i = 0, 3 do
        calc_forces(
            p_sides[i],
            p_ghost[i])
    end

    for i = 0, 3 do
        adv_pos_full(
            p_master[i])
    end

    t += dt
end
```

Warning: Naive approach

communication gathers data to least common ancestor

and scatters to destination subregions

wasted communication on points used by only one color
Regent Example: PENNANT

Optimized approach

Main simulation loop:

while t < T do

for i = 0, 3 do
    calc_forces( p_sides[i], p_ghost[i] )
end

for i = 0, 3 do
    adv_pos_full( p_master[i] )
end

t += dt
end
while t < T do
    for i = 0, 3 do
        calc_forces(p_sides[i], p_private[i], p_ghost[i])
    end
    for i = 0, 3 do
        adv_pos_full(p_private[i])
    end
    for i = 0, 3 do
        adv_pos_full(p_master[i])
    end
    t += dt
end
Lines of Code

![Bar chart showing non-comment, non-blank lines of code for Circuit, PENNANT, and MiniAero, with Regent and Reference categories.]

- Circuit: Regent 825, Reference 1,701
- PENNANT: Regent 1,789, Reference 2,416
- MiniAero: Regent 2,836, Reference 3,993
Partitions

N

W

P

S

p_1 p_2 p_3 s_1 s_2 s_3 g_1 g_2 g_3 w_1 w_2 w_3

http://regent-lang.org
Optimizing Index Launches

- Iterations must be independent
- Static dependence analysis finds eligible loops
- Index launch amortizes dynamic analysis cost
- Runtime will find parallelism regardless

for i = 0, 3 do
  B(p[i])
end

forall i = 0, 3: B(p[i])

dynamic region tree

static region tree
Optimizing Leaf Tasks

Structural analysis finds leaf tasks

-- Not leaf task: has calls
A(r)
for i = 0, 3 do
  B(p[i])
end
C(r)

-- Leaf task: no calls
for x in r do
  x.pos += dt * x.vel
end
Optimizing Mapping

- Flow-sensitive analysis to find spans of regions usage

\[
\text{unmap}(r) \\
A(r) \\
\text{map}(r) \\
\text{for } i = 0, 3 \text{ do} \\
\quad \text{unmap}(r) \\
\quad B(p[i]) \\
\quad \text{map}(r) \\
\text{end} \\
\text{unmap}(r) \\
C(r) \\
\text{map}(r)
\]

- Emit calls only on boundaries between usage modes

\[
\text{unmap}(r) \\
A(r) \\
\text{for } i = 0, 3 \text{ do} \\
\quad B(p[i]) \\
\quad \text{end} \\
\text{C}(r) \\
\text{map}(r)
\]
Optimizing Futures

- Runtime stalls if task results are used immediately

  ```
  var x : int = get(f(...))
  g(x) – stalls until x ready
  h(…) – can’t run (stalled)
  ```

- Compiler automatically flows futures through variables to avoid blocking

  ```
  var x : future(int) = f(…)
  g(x) – runs when x ready
  h(…) – runs immediately
  ```
Optimizing Pointer Check Elision

- Runtime pointer checks:
  - Per access (expensive)
  - Or not at all (dangerous)

- Compiler check pointers statically

```
for x in r do
  – x has type ptr(…, r)
  – so this access is safe
  @x = ...
end
```
Optimizing Dynamic Branches

- Ghosts stored separately from private region
- Requires dynamic check per element

for w in wires do
  - private or ghost?
  - need to check
  w.out_ptr = ...
end

If mapper co-locates regions, avoid the dynamic check
Vectorization

Loops over regions may be vectorizable

for x in r do
  x.f = 0
end

Compiler has improved info on independence, costs, etc.

for x : vector(...) in r do
  x.f = 0
end
Terra & Lua

- Lua is a dynamic language designed for embedding
- Terra is a C-like language which JITs to LLVM
- Enables meta-programming, embedded DSLs
- Automatic support for vectorization, etc.
Pushing the Performance Envelope with Static Analysis

![Diagram showing the relationship between task granularity and scale, with Dynamic Analysis and Static Analysis curves.](http://regent-lang.org)