Legion: Programming Heterogeneous, Distributed Parallel Machines

Michael Bauer
Zhihao Jia
Elliott Slaughter
Sean Treichler
Wonchan Lee
Alex Aiken

Pat McCormick
Sam Guiterrez
Galen Shipman
Modern Supercomputers

- Heterogeneity
  - Processor kinds
  - Relative performance

- Distributed Memory
  - Non-uniform
  - Distinct from processors

- Growing disparities
  - FLOPS >> bandwidth
  - bandwidth >> latency
Programming System Goals

High Performance
We must be fast

Performance Portability
Across many kinds of machines and over many generations

Programmability
Sequential semantics, parallel execution
Can We Fulfill These Goals Today?  
Yes … at great cost:

Do you want to schedule that graph?  
(High Performance)

Do you want to re-schedule that graph for every new machine?  
(Performance Portability)

Do you want to be responsible for generating that graph?  
(Programmability)

Today: programmer’s responsibility

Tomorrow: programming system’s responsibility

Task graph for one time step on one node…

… of a mini-app
The Crux

- How do we describe the data?
  - Most programming systems focus on control
  - Minimal facilities for organization/structure of data

- Why?

  Answer: Solve the *aliasing problem*
  - Can two references refer to the same data?

  Answer: Decouple naming data from layout/location
Legion Approach

- Capture the structure of program data
- Decouple specification from *mapping*

Automate
  - data movement
  - parallelism discovery
  - synchronization
  - hiding long latency
Overview

- Legion Programming Model
- Legion Runtime System
- A Real Application: S3D
- Performance Results
- Conclusion
Legion Programming Model
Writing Legion Applications

- Logical regions

- Writing tasks
  - `calc_currents(piece[0], p_0, s_0, g_0);`
  - `calc_currents(piece[1], p_1, s_1, g_1);`
  - `distribute_charge(piece[0], p_0, s_0, g_0);`
  - `distribute_charge(piece[1], p_1, s_1, g_1);`

- Mapping Legion applications to hardware

Machine Independent Program Specification
Example: Circuit Simulation

Challenges

- Requires online partitioning
- Iterative solver, may need dynamic load balancing
- Data and task placement non-obvious, architecture dependent
Example: Circuit Simulation

```plaintext
task simulate_circuit (Region[Node] N, Region[Wires] W)
{

Tasks are the unit of parallel execution.

Logical regions are (typed) collections

Logical:
  no implied layout
  no implied location
}
```
Partitioning
Partitioning

task simulate_circuit(Region[Node] N, Region[Wires] W)
{
    [ P, S ] = partition(ps_map, N)
}

}
Partitioning

task simulate_circuit(Region[Node] N, Region[Wires] W)
{
    Array[Region[Node]] private, shared, ghost;
    ...
    [ P, S ] = partition(ps_map, N)
    private = partition(private_map, P)
    shared = partition(shared_map, S)
    ghost = partition(ghost_map, S)
}
Region Trees
Region Trees

disjoint

Locality
Independence
Region Trees

disjoint

Locality
Independence
Region Trees

Locality
Independence
Aliasing
Region Trees

Locality
Independence
Aliasing
Legion Tasks

task simulate_circuit(Region[Node] N, Region[Wires] W) :
{
  ...
  calc_currents(piece[0]);
  calc_currents(piece[1]);
  distribute_charge(piece[0]);
  distribute_charge(piece[1]);
  ...
}

subtasks

task calc_currents(Piece p) :

task distribute_charge(Piece p) :
Legion Tasks

```plaintext
task simulate_circuit(Region[Node] N, Region[Wires] W) :
  N, W
{
  ...
  calc_currents(piece[0], \textcolor{blue}{p_0}, \textcolor{red}{s_0}, \textcolor{green}{g_0});
  calc_currents(piece[1], \textcolor{blue}{p_1}, \textcolor{red}{s_1}, \textcolor{green}{g_1});
  distribute_charge(piece[0], \textcolor{blue}{p_0}, \textcolor{red}{s_0}, \textcolor{green}{g_0});
  distribute_charge(piece[1], \textcolor{blue}{p_1}, \textcolor{red}{s_1}, \textcolor{green}{g_1});
  ...
}

task calc_currents(Piece p) :
  p.private, p.shared, p.ghost, p.wires

task distribute_charge(Piece p) :
  p.private, p.shared, p.ghost, p.wires
```

A task must declare the regions it will use.

Subtask containment: A subtask can only use (sub)regions accessible to its parent task.

Tasks appear to execute in program order.
Interference

Informal definition: Two tasks $T_1$ and $T_2$ are non-interfering $T_1 \# T_2$ if there is no dependence between them on their region arguments.

If $T_1 \# T_2$ then $T_1$ and $T_2$ can execute in parallel.
Execution Model

task simulate_circuit(Region[Node] N, Region[Wires] W) :
{
    ...
    calc_currents(piece[0], p_0, s_0, g_0);
    calc_currents(piece[1], p_1, s_1, g_1);
    distribute_charge(piece[0], p_0, s_0, g_0);
    distribute_charge(piece[1], p_1, s_1, g_1);
    ...
}

Tasks are issued in program order.
Privileges

task simulate_circuit(Region[Node] N, Region[Wires] W) : ReadWrite(N,W)
{
    ...  
calc_currents(piece[0], p_0, s_0, g_0);
calc_currents(piece[1], p_1, s_1, g_1);
distribute_charge(piece[0], p_0, s_0, g_0);
distribute_charge(piece[1], p_1, s_1, g_1);
    ...
}

task calc_currents(Piece p) :
    ReadWrite(p.shared), Read(p.private, p.shared, p.ghost)
p.wires

task distribute_charge(Piece p) :
    ReadOnly(p.shared), ReadGhost(p.private, p.shared, p.ghost)
p.wires
Non-Interference Dimensions

Several dimensions of `#` operator
- Entries (rows)
- Privileges
- Fields (columns)

Logical regions are a relational data model
- Partitioning is selection (`σ`)
- Field-slicing is projection (`π`)
- Don’t support all relational operators
Legion Summary

- Logical regions: a relational data model
  - Support partitioning and slicing
  - Convey locality, independence, aliasing

- Implicit task parallelism
  - Task may have arbitrary sub-tasks
  - Tasks declare region usage including privileges and fields

- Tasks appear to execute in program order
  - Execute in parallel when non-interference established

- Machine independent specification of application
Legion Runtime System
Legion Runtime System

Tasks : Regions :: Instructions : Registers

A Distributed Hierarchical Out-of-Order Task Processor
task simulate_circuit(Region[Node] N, Region[Wires] W) :
  ReadWrite(N,W)
{
  ... 
  calc_currents(piece[0],
  calc_currents(piece[1],
  distribute_charge(piece[0],
  distribute_charge(piece[1],
  ... 
}

task calc_currents(Piece p) :
  ReadWrite(p.wires), Read(p.private, p.shared, p.ghost)

task distribute_charge(Piece p) :
  ReadOnly(p.wires), Reduce(p.private, p.shared, p.ghost)
Dependence Analysis

task simulate_circuit(Region[Node] N, Region[Wires] W) :
  ReadWrite(N,W)
{
  ...
  calc_currents(piece[0],
  calc_currents(piece[1],
  distribute_charge(piece[0],
  distribute_charge(piece[1],
  ...
}

task calc_currents(Piece p) :
  ReadWrite(p.wires),
  Read(p.private, p.shared, p.ghost)

task distribute_charge(Piece p) :
  ReadOnly(p.wires),
  Reduce(p.private, p.shared, p.ghost)

read-after-write of wires w/CC[0]

DC[0]  CC[1]  CC[0]
Mapping Interface

- Programmer selects:
  - Where tasks run
  - Where regions are placed

- Mapping computed dynamically

- Decouple correctness from performance
Correctness Independent of Mapping

task simulate_circuit(Region[Node] N, Region[Wires] W) :
    ReadWrite(N,W)
{
    ...  
    calc_currents(piece[0], p[0]);
    calc_currents(piece[1], p[1]);
    distribute_charge(piece[0], p[0]);
    distribute_charge(piece[1], p[1]);
    ...
}

task calc_currents(Piece p) :
    ReadWrite(p.wires), Read(p.private, p.shared, p.ghost)

task distribute_charge(Piece p) :
    ReadOnly(p.wires), Reduce(p.private, p.shared, p.ghost)
Distribution

- After tasks are mapped they are distributed to target node.
- Task execution can generate sub-tasks.
- Do we need inter-node dependence checks?

Subtask containment:
A subtask can only use (sub)regions accessible to its parent task.
Independence Theorem

Let $t_1$ be a subtask of $T_1$ and $t_2$ be a subtask of $T_2$. Then

$$T_1 \# T_2 \Rightarrow t_1 \# t_2$$
Independence Theorem

Let $t_1$ be a subtask of $T_1$ and $t_2$ be a subtask of $T_2$. Then

$$T_1#T_2 \Rightarrow t_1#t_2$$

Proof: Use subtask containment.

Observation: It is sufficient to test interference only of sibling tasks.

Note: Similar property holds in functional languages, but it holds in Legion even though we may imperatively mutate regions.
Runtime Summary

- A distributed hierarchical out-of-order task processor
  - Analogous to hardware processors

- Can exploit parallelism implicitly:
  - Task-, data-, and nested-parallelism

- Runtime builds task graph ahead of execution to hide latency and costs of dynamic analysis

- Decouples mapping decisions from correctness
  - Enables efficient porting and (auto) tuning
A Real Application: S3D
S3D

- Production combustion simulation
- Written in ~200K lines of Fortran
- Direct numerical simulation using explicit methods
S3D Versions

- Supports many chemical mechanism
  - DME (30 species)
  - Heptane (52 species)

- Fortran + MPI
  - Vectorizes well
  - MPI used for multi-core

- “Hybrid” OpenACC
  - Recent work by Cray/Nvidia/DoE

- Legion interoperates with MPI

---

Recent work by Cray/MPI used for multi-core

- Vectorizes DME (30 species)

- OpenACC

---

Supports many chemical mechanisms

- Thousands of elementary reaction steps accumulated to global species reaction rates
- Often the target for model reductions or algorithmic improvements
- How fast can we compute detailed chemical kinetics on accelerators?

---

Detailed chemical kinetics are expensive

- How fast can we compute detailed chemical kinetics on

---

Planning the science simulation

- Lacking a systematic, first-principle procedure to identify all

---

Fig. 10.

- Auto-Ignition Delay (sec)

---

Recent 3D simulation on Jaguar

- Will need a month on 12,000

---

Auto-Ignition Delay (sec)

- Thousands of elementary reaction steps accumulated to global species reaction rates

---

Table: Number of reactions, $I$

<table>
<thead>
<tr>
<th>Chemistry</th>
<th>Number of reactions, $I$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$CH_4$ (Leeds)</td>
<td>2</td>
</tr>
<tr>
<td>$C_2H_4$ (San Diego)</td>
<td>4</td>
</tr>
<tr>
<td>USC C2H4</td>
<td>8</td>
</tr>
<tr>
<td>USC C1-C4</td>
<td>10</td>
</tr>
<tr>
<td>GRI1.2</td>
<td>10</td>
</tr>
<tr>
<td>1,3-Butadiene</td>
<td>10</td>
</tr>
<tr>
<td>iso-o</td>
<td>10</td>
</tr>
<tr>
<td>$CH_4$ (K)</td>
<td>2</td>
</tr>
<tr>
<td>CH4 (Gh et al.)</td>
<td>2</td>
</tr>
<tr>
<td>GR13.0</td>
<td>4</td>
</tr>
<tr>
<td>Decanoate</td>
<td>8</td>
</tr>
<tr>
<td>Methyl</td>
<td>10</td>
</tr>
<tr>
<td>Skeletal $n$-heptane</td>
<td>10</td>
</tr>
<tr>
<td>Skeletal iso-octane</td>
<td>10</td>
</tr>
</tbody>
</table>

---

Fig. 11.

- Number of reactions, $I$

---

DME chemistry, a biomass fuel surrogate, consists of 3036

---

Fig. 12.

- DME chemistry
Parallelism in S3D

- Data is large 3D cartesian grid of cells

- Typical per-node subgrid is $48^3$ or $64^3$ cells
  - Nearly all kernels are per-cell
  - Embarrassingly data parallel

- Hundreds of tasks
  - Significant task-level parallelism

- Except...
  - Computational intensity is low
  - Large working sets per cell (1000s of temporaries)
  - Performance limiter is data, not compute
S3D Tasks in Legion

\[ R_q \]
\[ \rho \ddot{v}, \rho E, \rho, \rho Y_i, \ldots \]

\[ R_{int} \]
\[ \text{vol} = \text{CalcVolume}(\rho) \]
\[ T = \text{SolveForTemp}(\rho E, \rho, \ldots) \]
\[ P = \text{CalcPressure}(T, \ldots) \]
\[ \nabla T = \text{CalcGradient}(T) \]
\[ \ldots \]
\[ \text{rr} \downarrow i = \text{ReactionRates}(Y \downarrow i, \ldots) \]
\[ \ldots \]
\[ \text{for } i \text{ in } [1..\text{nspec}]: \]
\[ \frac{\partial}{\partial t} \rho Y_i \equiv f(\text{rr} \downarrow i, \ldots) \]

\[ R_{state} \]
\[ T, \lambda, \nu, D_{s_i}, \ldots \]

\[ R_{rhs} \]
\[ \frac{\partial}{\partial t} \rho \ddot{v}, \frac{\partial}{\partial t} \rho E, \frac{\partial}{\partial t} \rho, \frac{\partial}{\partial t} \rho Y_i, \ldots \]
S3D Task Parallelism

- One call to Right-Hand-Side-Function (RHSF) as seen by the Legion runtime
  - Called 6 times per time step by Runge-Kutta solver
  - Width == task parallelism
  - H2 mechanism (only 9 species)
  - Heptane (52 species) is significantly wider
- Manual task scheduling would be difficult!
S3D Execution Profile

- Task executions on different processors
  - Copies not shown
  - “Bubbles” caused by critical path
Mapping for Heptane 48³

- Dynamic Analysis for \((\text{rhsf+2})\)
- Clean-up/meta tasks

4 AMD Interlagos Integer cores for Legion Runtime

8 AMD Interlagos FP cores for application

NVIDIA Kepler K20
Heptane Mapping for $96^3$

- Handle larger problem sizes per node
  - Higher computation-to-communication ratios
  - More power efficient
- Not enough room in 6 GB GPU framebuffer
  - OpenACC requires code changes
- Legion analysis is independent of problem size
  - Larger tasks -> fewer runtime cores
Performance Results
Legion S3D DME Performance

- 1.71X - 2.33X faster between 1024 and 8192 nodes
- Larger problem sizes have higher efficiency
Legion Heptane Performance

- 1.73X - 2.85X faster between 1024 and 8192 nodes
- Higher throughput on Keeneland (balanced CPU+GPUs)
Legion PRF Performance

- 116 species mechanism, >2X as large as heptane
- Legion uses different mapping approach
Conclusion
Conclusion

- Describing structure of program data is important!

- Logical regions provide a relational data model
  - Dynamic, expressive, multiple views

- Supports dynamic runtime analysis
  - Implicit parallelism enables sequential reasoning
  - Automates movement of program data

- Decouples correctness and performance of mapping
  - Enables efficient tuning and porting on new architectures
Legion

- Legion website: http://legion.stanford.edu
- Github repo: http://github.com/stanfordlegion

Questions?