LEGION: A VISION FOR FUTURE HPC PROGRAMMING SYSTEMS

Michael Bauer, NVIDIA Research
Patrick McCormick, Los Alamos National Laboratory
PROGRAMMING SYSTEM GOALS
What do we need to deliver?

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?

We can...

... at great cost: **Programmer Pain**

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)

---

**Task graph for one time step on one node...**

... of a **mini-app**

**Today: programmer’s responsibility**

**Tomorrow: programming system’s responsibility**
VISION FOR PROGRAMMING SYSTEMS
Intelligent Programming Model and System Design

These must be co-designed

Expressive High-Level Programming Models

Powerful Programming Systems

Static Analysis
Dynamic Analysis
Synthesis
Formal Methods

High Performance
Performance
Portability
Programmability
PROGRAMMING MODEL DESIGN

The difference is in understanding the data

Low-Level Programming Systems

Emphasize control abstractions

- MPI
- NVIDIA CUDA
- OpenACC
- OpenMP

Burden of data management falls on the programmer

High-Level Programming Systems

Emphasize data abstractions

- Legion
- Spark

Both are examples of strong data models

1. Complete understanding of the structure of program data
2. Enable sound program analysis and powerful tool construction

How does this help?
PARALLEL PROGRAMMING PROBLEMS
Why programming systems should have strong data models

- Functionally correct application code
- Mapping to target machine
- Extraction of parallelism
- Management of data transfers
- Task scheduling and Latency hiding
- Data-Dependent Behavior
PARALLEL PROGRAMMING PROBLEMS
Why programming systems should have strong data models

- Functionally correct application code
- Mapping to target machine
- Extraction of parallelism
- Management of data transfers
- Task scheduling and Latency hiding
- Data-Dependent Behavior

Compiler/Runtime understanding of data
LEGEN D PROGRAMMING MODEL

Leverage a Strong Data Model

Legion Programs

Extraction of parallelism

Task scheduling and Latency hiding

Legion Mappers

Management of data transfers

Data-Dependent Behavior

Legion Programming System
LEGION PROGRAMMING MODEL
Separating Correctness and Performance

Legion Program

Legion

Legion Mapper

Analysis!

Machine-Independent Specification

Defines application correctness

Sequential semantics

Programmatic interface for performing machine-specific and app-specific mapping

Only impacts performance

Yesterday: manual mapping

Today: programmatic mapping

Tomorrow: generated mappers
EVALUATION ON A REAL APPLICATION: S3D

Production run on Titan and Piz-Daint

Ported more than 100K lines of MPI+Fortran to Legion C++

Legion enables new chemistry via better scheduling to limit data footprint sizes

Ran on two of the world’s top 10 supercomputers for 1 month

- Porting time to Titan from Keeneland: 14 hours (normal tuning takes days - weeks)
- Porting time to Piz-Daint from Titan: 4 hours
COMPARISON WITH MPI+OPENACC

The power of program analysis

Weak scaling results on Titan out to 8K nodes

As application and machine complexity increases, the performance gap will grow.
THE IMPORTANCE OF ABSTRACTIONS
The Impact of Direct Control & Low-Level Data Abstractions

AsyncRecv(X, sizeof(X));
DoWork(Y);
Sync();
F(X);

How much work should we do? What happens in DoWork()? 
What resources are in use? Where is the data? Who is using it and how? (i.e. How do we schedule?)
Is this modular? Is it performance portable? Does it work for more than one app? Are we productive?

What’s the impact as we face new/emerging trends/challenges in programming?

Missed Opportunities!
IN SITU ANALYSIS & VISUALIZATION

Perfect Example of Resource Contention...

We can no longer afford to save “everything” for post run processing...

Would like to minimize impacts on performance and code structure.

Separation of concerns simplifies details and can minimize overheads.

System-dependent scheduling and mapping reduces analysis overhead to less than 1% of overall execution time.
EXTENDING THE MODEL TO STORAGE
Expose the Entire Memory Hierarchy

Deeper storage hierarchies, asynchronous I/O, versioning, etc. Yet another layer of complexity that developers must reason about...

Data movement is expensive, compute is free... But...

Idle processors are expensive! (e.g. Trinity: dumping data memory to disk spends 10X more in power waiting on the data to move than to move the data)
CONCLUSIONS

We need powerful programming systems

**Should be co-designed** - models, systems, applications, supporting infrastructure (avoid stove-piped solution spaces).

A strong **data model is a critical component** that ties the pieces together.

Thanks:

Alex Aiken, Eric Barton, Ben Bergen, Scot Britenfeld, Jackie Chen, David Daniel, Charles Ferenbaugh, Gary Grider, Sam Gutierrez, Zhihao Jia, Quincey Koziol, Wonchan Lee, Carlos Maltzahn, Galen Shipman, Elliott Slaughter, Ian Sohl, Christine Sweeney, Sean Treichler, Noah Watkins

*Funding acknowledgements: DOE SC Office of Advanced Scientific Computing Research DOE NNSA Advanced Simulation and Computing*
CONCLUSION

We need powerful programming systems

We must co-design programming models with programming systems to achieve our goals

Strong data models are a crucial component of programming model and system co-design

Legion shows the value of this approach and the benefits conferred to real applications