SAND2015-1754C

Sandia National Laboratories

Exceptional service in the national interest

DHARMA: Distributed asyncHronous Adaptive Resilient Management of Applications

Janine Bennett, Robert Clay, John Floren, Ken Franko, Saurabh Hukerikar, Samuel Knight, Hemanth Kolla, Greg Sjaardema, Nicole Slattengren, Keita Teranishi, **Jeremiah Wilke**

SOS 19, Park City, UT
March 3, 2014

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. SAND NO. 2011-XXXXP

Mission

- Address challenges at extreme-scale that seem intractable with current PMs
  - Minimize data movement
  - Performance portability
  - Composability + fault-tolerance
- Focus #1: Evaluate existing PMs
  - Uintah: SPMD structured mesh with on-node DAG
  - Legion: Decoupling of logical algorithm and physical implementation, DAG automation
  - Charm++: Communicating parallel objects
- Focus #2: Develop AMT capability to fill potential gaps in existing PMs

Why pursue yet another AMT? 3 key efficiency/productivity challenges

- Overdecomposition and latency hiding
  - Data pipelining – operate on data as soon it is ready to use, not when entire giant chunk arrives
  - Programmer productivity: No more deciding how much work between MPI_Isend and MPI_Wait
- No universal data structures – leave app-specific
- Fault-tolerance

Why pursue yet another AMT?
3 key efficiency/productivity challenges

- Overdecomposition and latency hiding
- No universal data structures – leave app-specific
  - Make it possible to use Kokkos, Raja, TiDA, or whatever else app developers dream up
  - Flexible C++- transport layer with flexible protocols and data structure slicing/subsets
- Fault-tolerance

Why pursue yet another AMT? 3 key efficiency/productivity challenges

- Overdecomposition and latency hiding
- No universal data structures – leave app-specific
- Fault-tolerance
  - "Virtualization" beyond just pointers - semantic/logical names
  - Assume SPMD structure dominates problem – task collection approach of Krishanmoorthy et al.
  - Efficient global agreement collective – simplify failure/recovery model to assume every agrees at the same time on who has failed

Why pursue yet another AMT?
3 key efficiency/productivity challenges

- Overdecomposition and latency hiding
- No universal data structures – leave app-specific
- Fault-tolerance

All three unified through a key-value store providing asynchronous communication, data flow, and fault-tolerance

Better, faster, cheaper

Food for thought:

1) How far would changes propagate to make optimizations to a single compute kernel in your large code? E.g. Do you have to blow up the entire code to do better cache blocking or tiling?
2) You *may* do anything in MPI. But *can* you?

- Better = Faster = Cheaper = more productive programmers
- Better = Faster = Cheaper = express more about your code to give compilers, runtime more to work with
- Case study of Legion + S3D

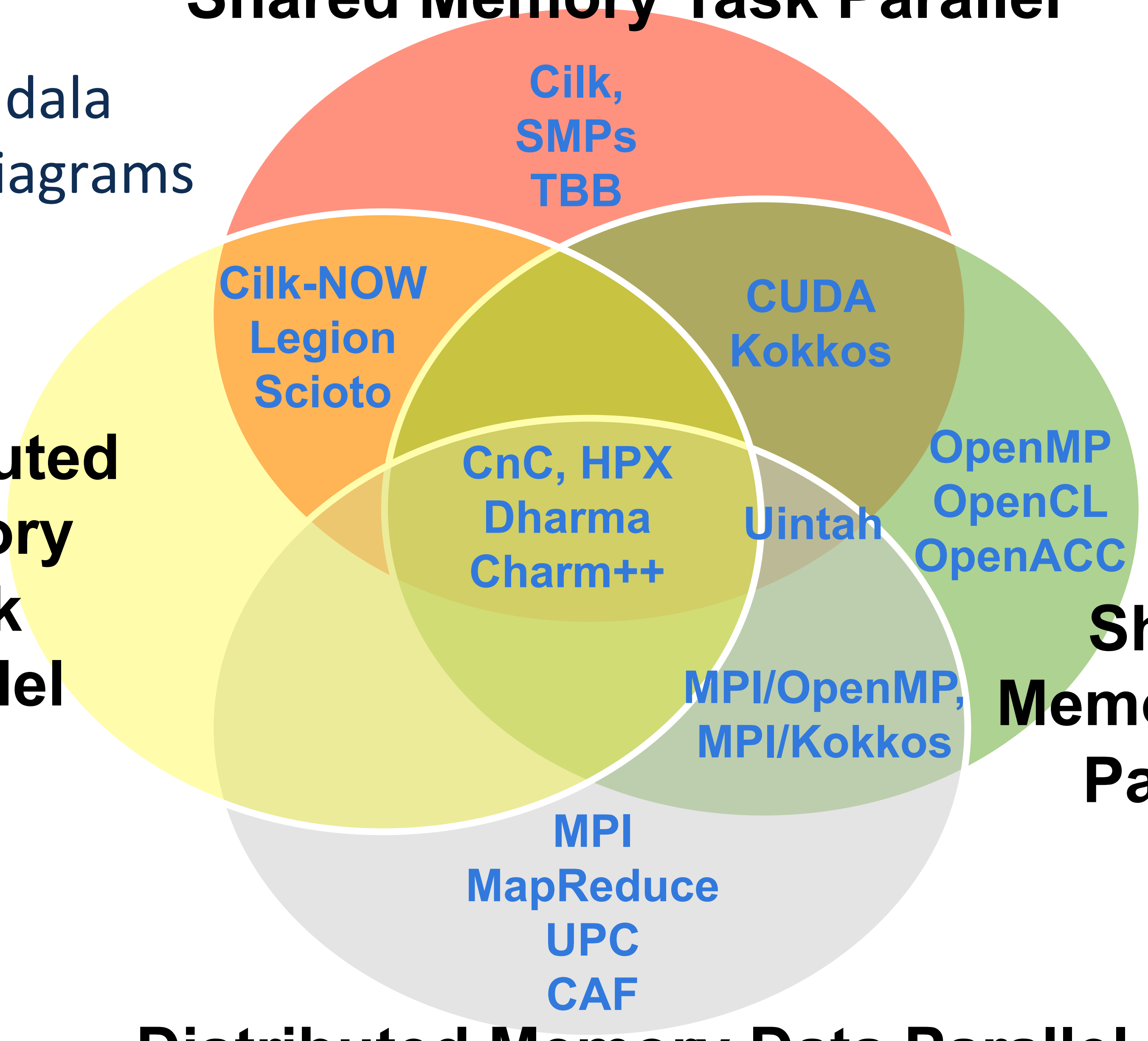
Development platform of the future?

- Whichever code makes it easy to express your algorithm correctly AND makes it easy to tune hardware mapping
- Don’t just rely on DSL or compiler to bridge usability gap

Answer your question with another question:

- Legion runtime overheads? Does it map well to SPMD? How difficult will the mapper interface be in practice? Fault-tolerance even with non-idempotent tasks?
- Uintah: Domain constrained? Internode load balance?
- Charm++: Works great for MD at large scale/contact app at medium scale. Large, unstructured mesh problems?
- Dharma: KV-store overheads? Burden on programmer?

Mandala Venn Diagrams



Shared Memory Task Parallel

Cilk,
SMPs
TBB

Distributed Memory Task Parallel

Cilk-NOW
Legion
Scioto

CUDA
Kokkos

CnC, HPX
Dharma
Charm++

Uintah

Shared Memory Data Parallel

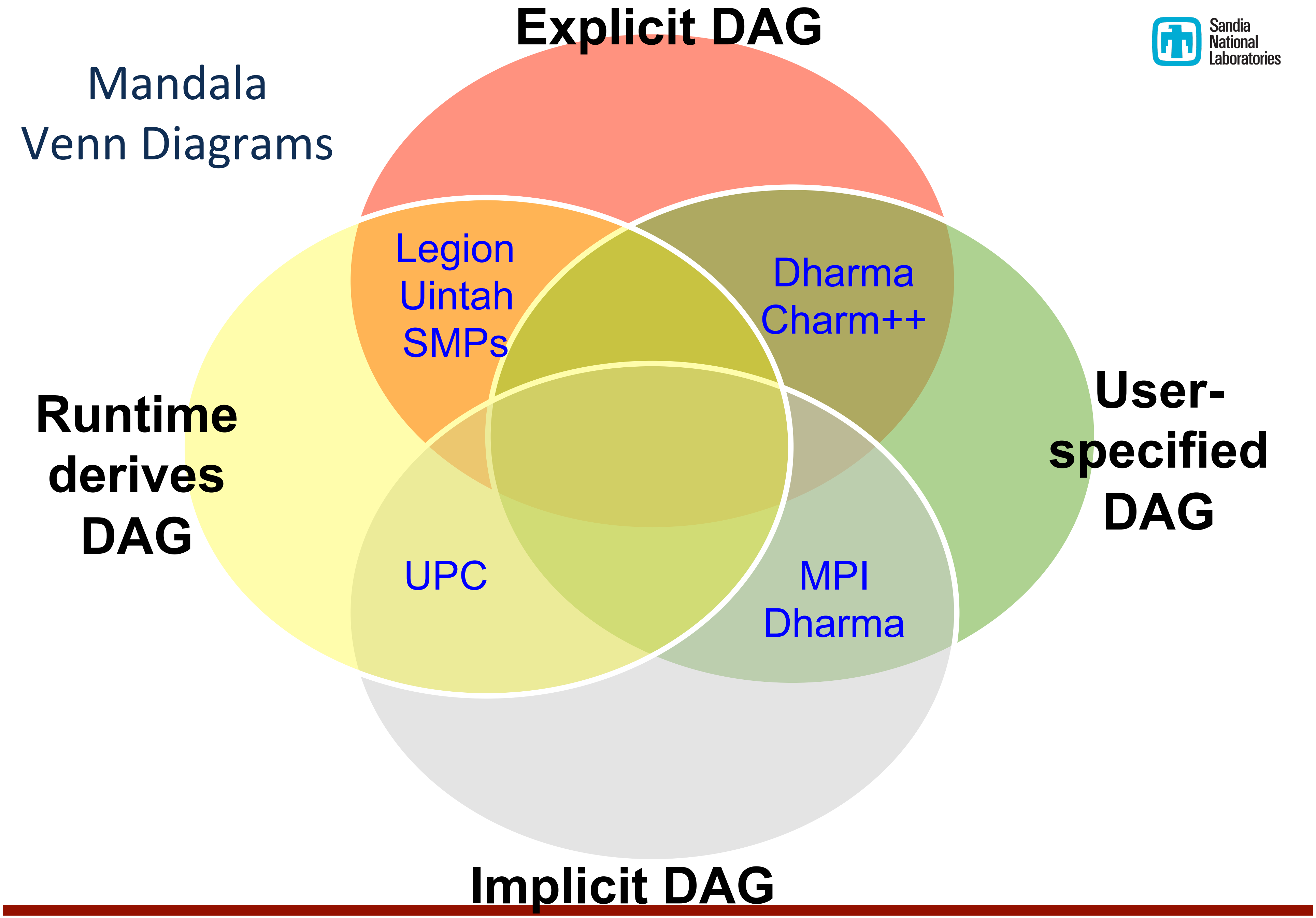
OpenMP
OpenCL
OpenACC

MPI/OpenMP,
MPI/Kokkos

MPI
MapReduce
UPC
CAF

Distributed Memory Data Parallel

Mandala Venn Diagrams



Explicit DAG

Runtime derives DAG

User-specified DAG

Implicit DAG

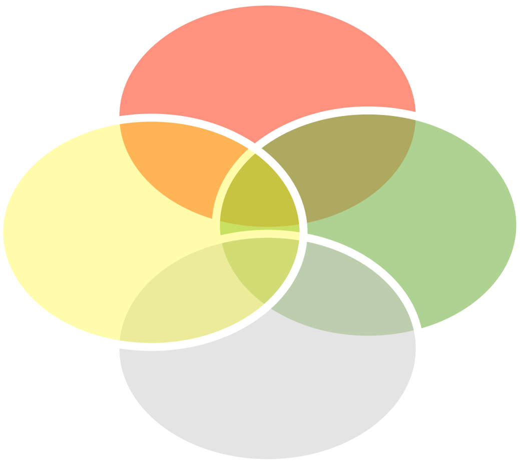
Legion
Uintah
SMPs

Dharma
Charm++

UPC

MPI
Dharma

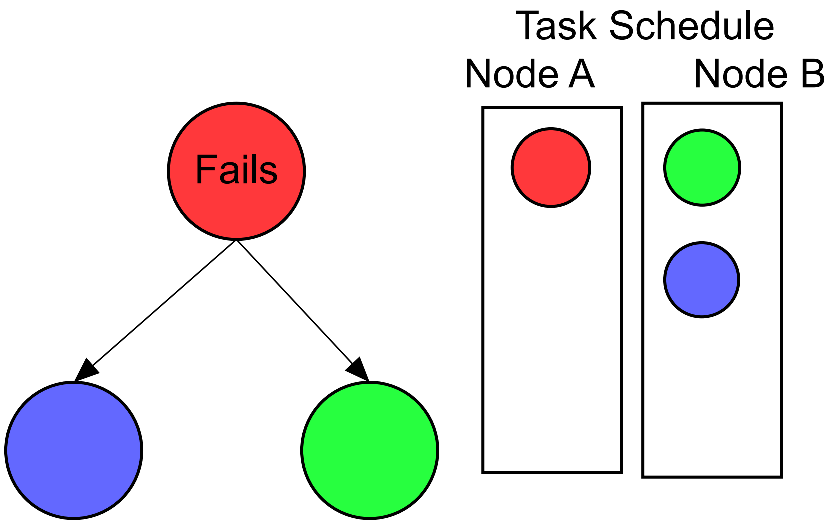
What features help and what features get in the way?

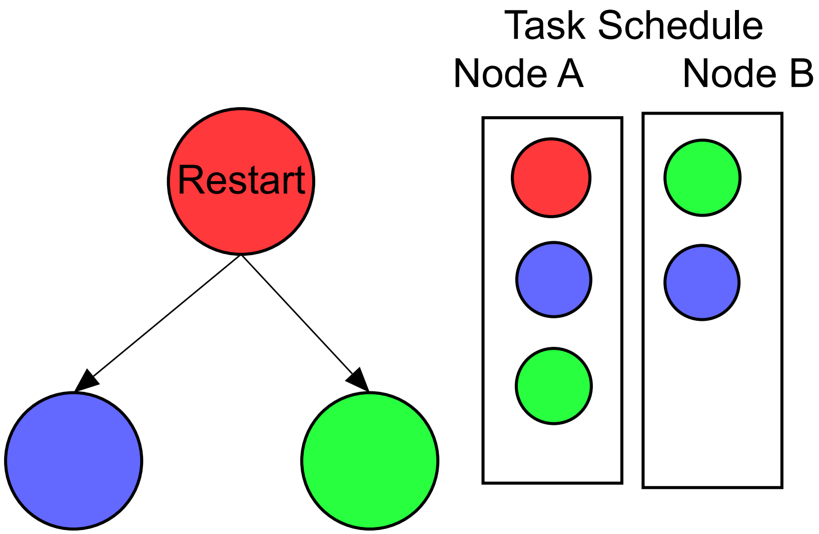


- Explicit vs. implicit DAG?
- User-defined or runtime-derived DAG?
- Runtime-specific data structures?
- Pointers or higher-level logic?
- Tasks communicate or isolated kernels?
- Direct collectives or collectives DAG-unrolled as part of DAG?
- Checkpoint strategy? Cascading rollback?

Restrictions make most sense in light of fault tolerance

- If nothing fails, you don't need to restrict the design
- Can relax restrictions with bookkeeping and fine-grained checkpoints, but is that too much bureaucracy?
- Burst buffers/tools like SCR make AMT a LOT easier than before – "asynchronous" checkpoint





Concluding thought:

Each AMT runtime is not just a tool. It is a *hypothesis*.

Each new application/science domain is an *experiment*.

Best AMT design will be decided ex post facto, not ab initio

1) Assert hypothesis
2) Controlled experiment
3) Refine hypothesis
4) Repeat