SAND2014-16477PE

Photos placed in horizontal position with even amount of white space between photos and header

Photos placed in horizontal position
with even amount of white space
between photos and header

Sandia National Laboratories

Exceptional service in the national interest

Kokkos Update

Programming Models and Apps Workshop

August 5, 2014

SAND2014-****P (Unlimited Release)

U.S. DEPARTMENT OF ENERGY

NNSA National Nuclear Security Administration

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

Increasingly Complex Heterogeneous Future

Sandia National Laboratories

¿ Performance Portable and Future Proof Codes?

Memory Spaces

- Bulk non-volatile (Flash?)
- Standard DDR (DDR4)
- Fast memory (HBM/HMC)
- (Segmented) scratch-pad on die

Execution Spaces

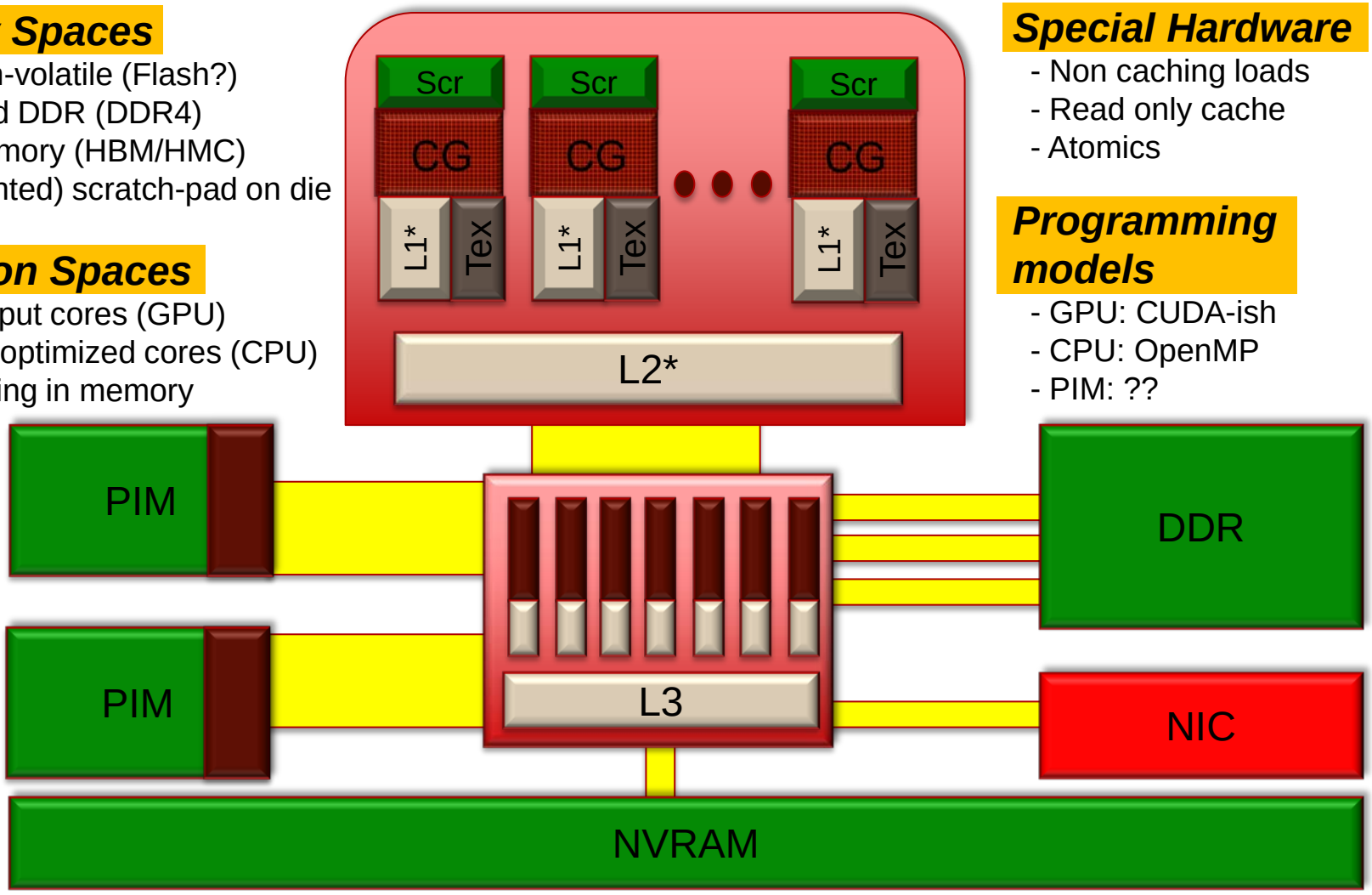
- Throughput cores (GPU)
- Latency optimized cores (CPU)
- Processing in memory

Special Hardware

- Non caching loads
- Read only cache
- Atomics

Programming models

- GPU: CUDA-ish
- CPU: OpenMP
- PIM: ??



Vision for Managing Heterogeneous Future

Sandia National Laboratories

- "MPI + X" Programming Model, separate concerns
  - Inter-node: MPI and domain specific libraries layered on MPI
  - Intra-node: Kokkos and domain specific libraries layered on Kokkos
- Intra-node parallelism, heterogeneity & diversity concerns
  - Execution spaces' (CPU, GPU, PIM, ...) diverse performance requirements
  - Memory spaces' diverse capabilities and performance characteristics
  - Vendors' diverse programming models for optimal utilization of hardware
- Desire standardized performance portable programming model
  - Via vendors' (slow) negotiations: OpenMP, OpenACC, OpenCL, C++17
  - Vendors' (biased) solutions: C++AMP, Thrust, CilkPlus, TBB, ArrayFire, ...
  - Researchers' solutions: HPX, StarPU, Bolt, Charm++, ...
- Necessary condition: address execution & memory space diversity
  - Execution { CPU, Xeon Phi, NVIDIA GPU }, Memory { GDDR, DDR, NVRAM }
  - SNL Computing Research Center's Kokkos (C++ library) solution
  - Engagement with ISO C++ Standard committee to influence C++17

Programmatics

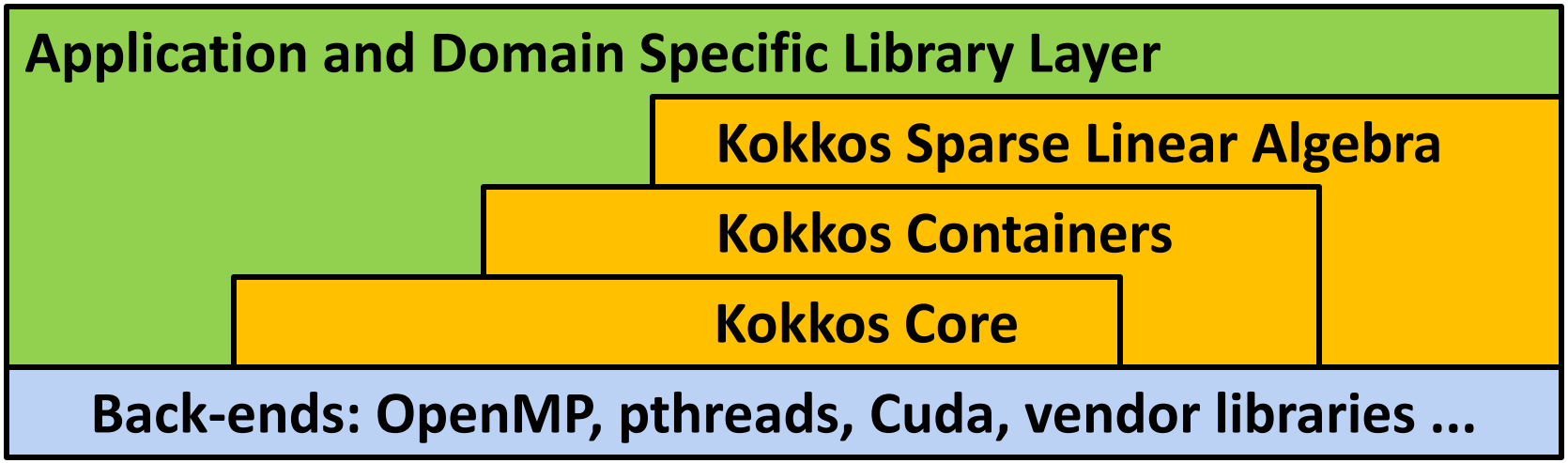
Sandia National Laboratories

- **ASC/CSSE (FY11 start): Heterogeneous Computing project**
  - Tight integration with co-design, mini-application, and testbed projects
  - Manycore (CPU, GPU, Xeon Phi, ...) performance portable "X" for MPI+X
    - Kokkos library is the "X" for fine grain data parallelism
  - 1.0-1.4 FTE split among ~2 staff + interns (FY14 @ 1.0 FTE)
- **ASCR/EASI : Sparse Linear Algebra Kernels on Manycore**
  - Some portion of this project also working on Kokkos core
- **LDRD (FY14 start): Unified Task+Data Manycore Parallelism**
  - For solver-preconditioners, finite elements, informatics, transport sweeps, ...
  - 0.9 FTE split among ~4 staff
- **Internal/external interests, and resource challenge ahead**
  - Trilinos, LAMMPS, SIERRA, other ASC codes (SNL, LANL, LLNL), AWE, ...
  - ISO C++ standards addressing fine grain parallelism (am a voting member)
  - ➢ Currently under-resourced for production-growth support

Kokkos: A Layered Collection of Libraries

Sandia National Laboratories

- Standard C++, Not a language extension
  - In *spirit* of TBB, Thrust & CUSP, C++AMP, LLNL's RAJA, ...
  - *Not* a language extension like OpenMP, OpenACC, OpenCL, CUDA, ...
- Uses C++ template meta-programming
  - Rely on C++1998 standard (supported everywhere except IBM's xlC)
  - Moving to C++2011 for concise lambda syntax (required by LLNL's RAJA)
    - Vendors slowly catching up to C++2011 language compliance



Performance Portability Challenge:

Device-Specific Memory Access Patterns are Required

- **CPUs (and Xeon Phi)**
  - Core-data affinity: consistent NUMA access (first touch)
  - Hyperthreads' cooperative use of L1 cache
  - Array alignment for cache-lines and vector units
- **GPUs**
  - Thread-data affinity: coalesced access with cache-line alignment
  - Temporal locality and special hardware (texture cache)
- **¿ "Array of Structures" vs. "Structure of Arrays" ?**

➢ **This has been the *wrong* question**

Right question: Abstractions for Performance Portability ?

Kokkos Performance Portability Answer

Sandia National Laboratories

- **Thread parallel computation**
  - Dispatched to an execution space
  - Operates on data in memory spaces
  - ➢ **Should use device-specific memory access pattern; how to portably?**
- **Multidimensional Arrays, *with a twist***
  - Layout mapping: multi-index (i,j,k,...) ↔ memory location
  - ➢ Choose layout to satisfy device-specific memory access pattern
  - Layout changes are invisible to the user code;
  - ➢ IF the user code uses Kokkos' simple array API: a(i,j,k,...)
- **Manage device specifics under simple portable API**
  - Dispatch computation to one or more execution spaces
  - Polymorphic multidimensional array layout
  - Utilization of special hardware; e.g., GPU texture cache

Sandia National Laboratories

Recent Publication

Kokkos: Enabling manycore performance portability through polymorphic memory access patterns, Journal of Parallel and Distributed Computing, July 2014

http://dx.doi.org/10.1016/j.jpdc.2014.07.003

Recent Use and Evaluations

Evaluate Performance Impact of Array Layout

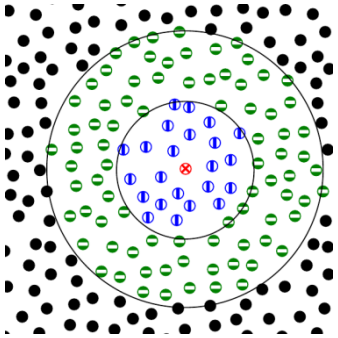
Sandia National Laboratories

- Molecular dynamics computational kernel in miniMD
- Simple Lennard Jones force model:

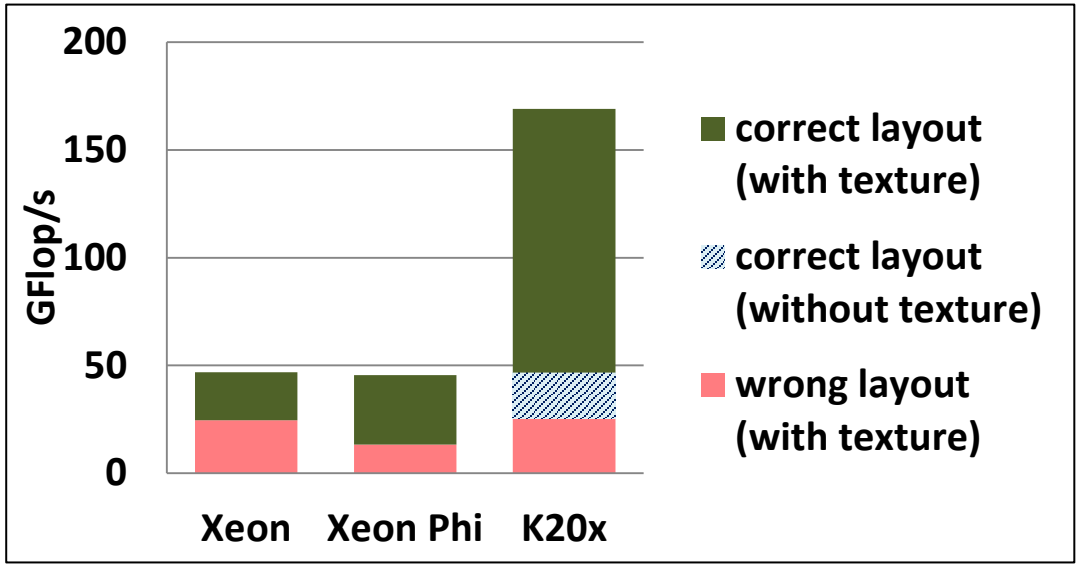
$$F_i = \sum_{j, r_{ij} < r_{cut}} 6\varepsilon \left[\left(\frac{\varsigma}{r_{ij}}\right)^{7} - 2\left(\frac{\varsigma}{r_{ij}}\right)^{13} \right]$$

- Atom neighbor list to avoid N^2 computations

```
pos_i = pos(i);
for( jj = 0; jj < num_neighbors(i); jj++) {
  j = neighbors(i,jj);
  r_ij = pos_i – pos(j); //random read 3 floats
  if (|r_ij| < r_cut) f_i += 6*e*((s/r_ij)^7 – 2*(s/r_ij)^13)
}
f(i) = f_i;
```



- Test Problem
  - 864k atoms, ~77 neighbors
  - 2D neighbor array
  - Different layouts CPU vs GPU
  - Random read ‘pos’ through GPU texture cache
- **Large performance loss with wrong array layout**

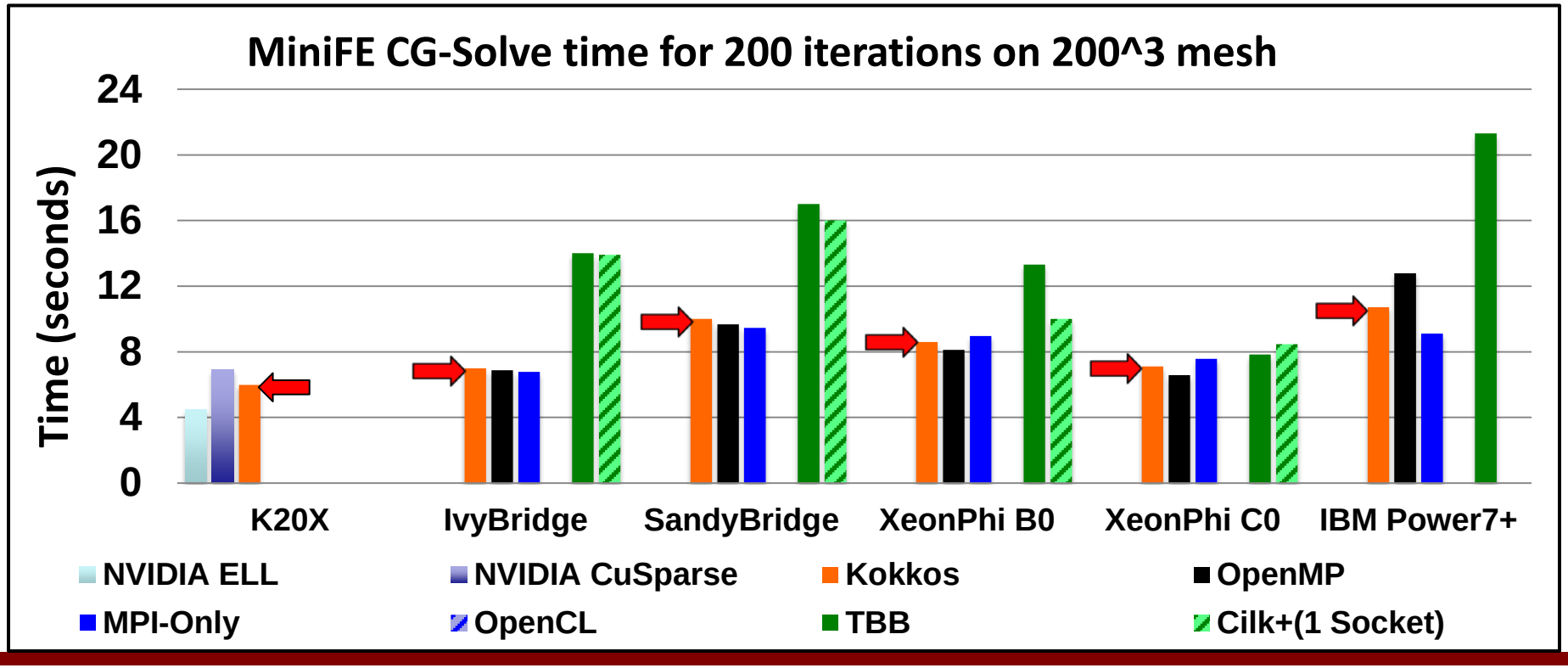


Evaluate Performance Overhead of Abstraction

Sandia National Laboratories

Kokkos competitive with native programming models

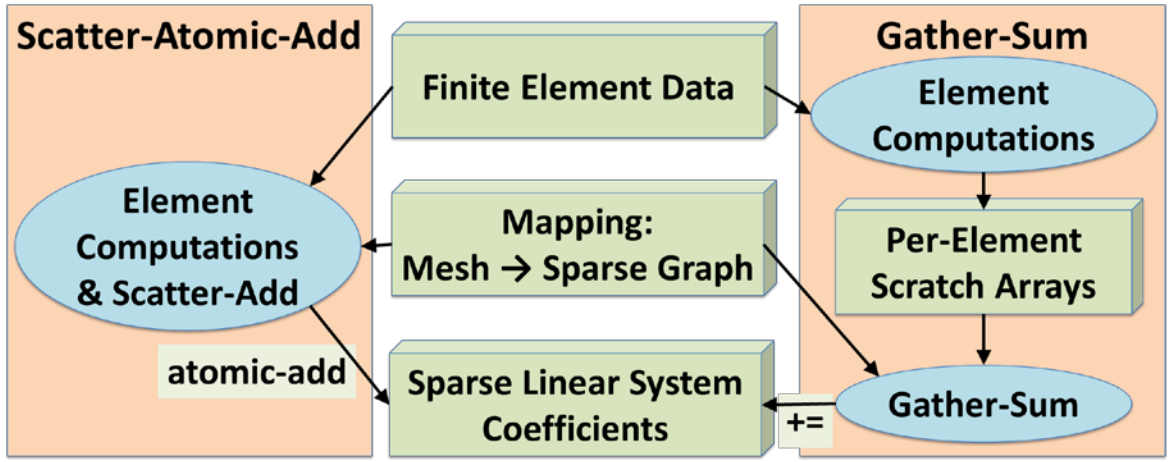
- MiniFE: finite element linear system iterative solver mini-app
- Compare to versions specialized for programming models
- Running on hardware testbeds

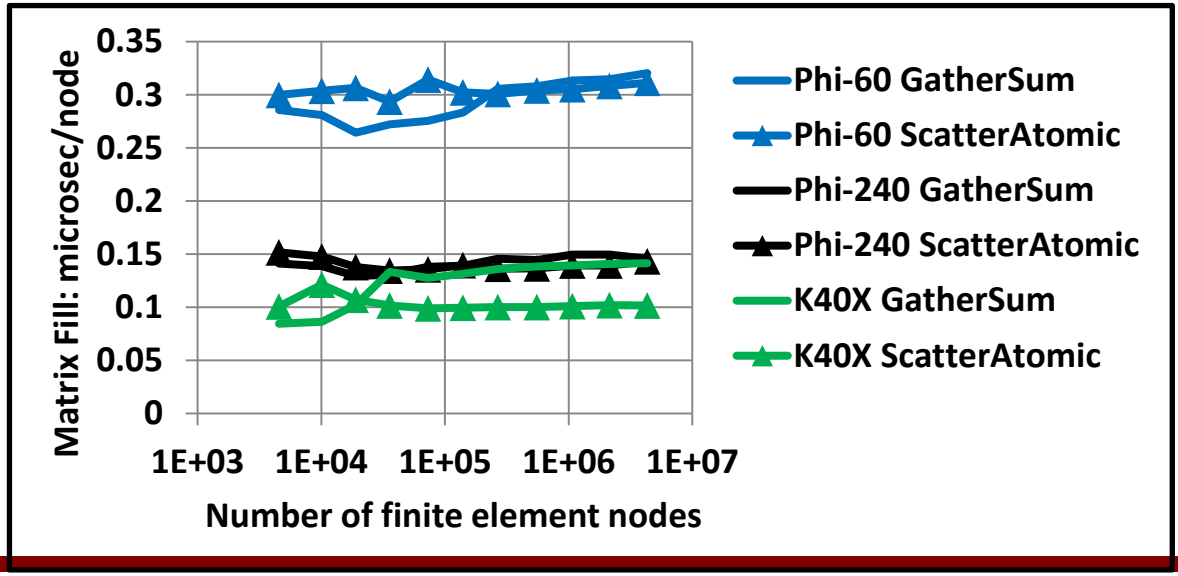


Thread-Scalable Fill of Sparse Linear System

Sandia National Laboratories

- MiniFENL: Newton iteration of FEM: $x_{n+1} = x_n - J^{-1}(x_n) r(x_n)$
- Thread-scalable pattern: Scatter-Atomic-Add or Gather-Sum ?
- Scatter-Atomic-Add
  - \+ Simpler
  - \+ Less memory
  - – Slower HW atomic
- Gather-Sum
  - \+ Bit-wise reproducibility
- Performance win?
  - Scatter-atomic-add
  - ~equal Xeon PHI
  - 40% faster Kepler GPU
- ✓ Pattern chosen
  - Feedback to HW vendors: performant atomics

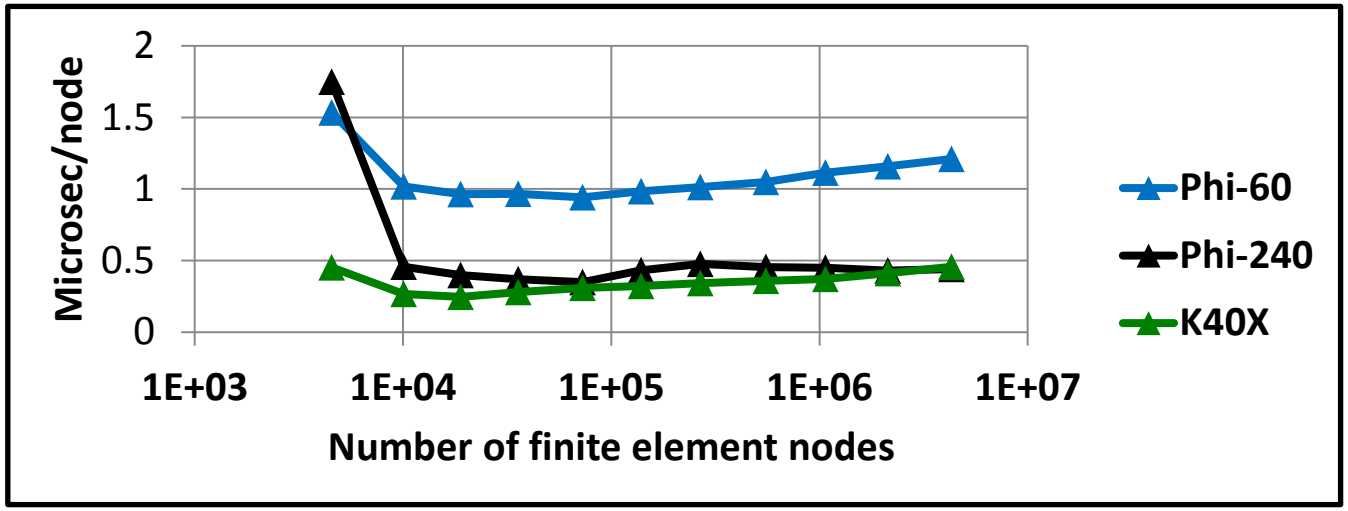




Thread-Scalable Sparse Matrix Construction

Sandia National Laboratories

- MiniFENL: Construct sparse matrix graph from FEM connectivity
- Thread scalable algorithm for constructing a data structure
  1. Parallel-for : fill Kokkos lock-free unordered map with FEM node-node pairs
  2. Parallel-scan : sparse matrix rows' column counts into row offsets
  3. Parallel-for : query unordered map to fill sparse matrix column-index array
  4. Parallel-for : sort rows' column-index subarray

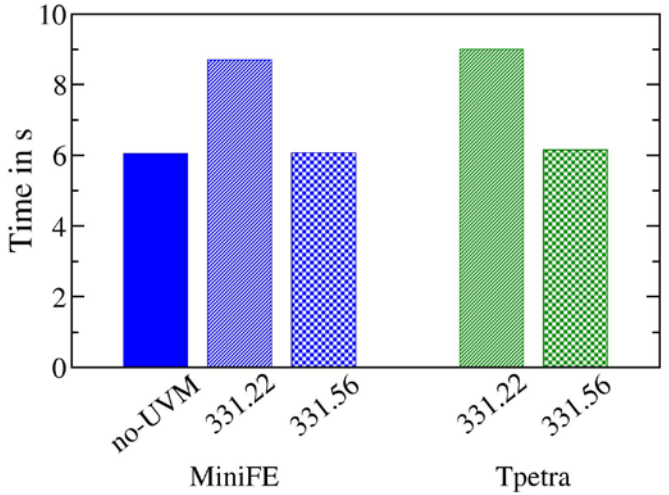


- Pattern and tools generally applicable to construction and dynamic modification of data structures

Tpetra: Domain Specific Library Layer for Sparse Linear Algebra Solvers

Sandia National Laboratories

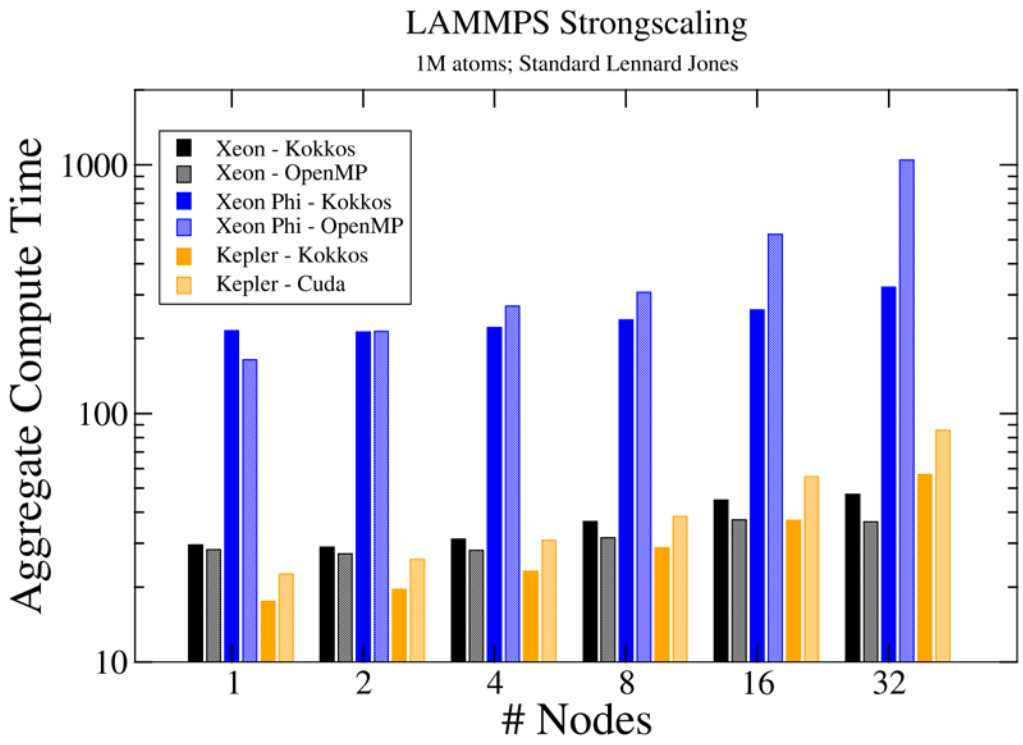
- Funded by ASC/Algorithms and ASCR/EASI
- Tpetra: Sandia's templated C++ library for sparse linear algebra
  - Templated on "scalar" type: float, double, automatic derivatives, UQ, ...
  - Incremental refactoring from pure-MPI to MPI+Kokkos
- CUDA UVM (unified virtual memory) codesign success
  - Sandia's early access to CUDA 6.0 via Sandia/NVIDIA collaboration
  - Hidden in Kokkos, can neglect memory spaces and maintain correctness
  - Enables incremental refactoring and testing
- Early access to UVM a win-win
  - Expedited refactoring + early evaluation
  - Identified performance issue in driver
  - NVIDIA fixed before their release



LAMMPS (molecular dynamics application) Porting to Kokkos has begun

Sandia National Laboratories

- Funded by LAMMPS' projects
- Enable thread scalability throughout code
  - Replace redundant hardware-specialized manycore parallel packages
- Current release has optional use of Kokkos
  - Data and device management
  - Some simple simulations can now run entirely on device
- **Performs as well or better than original hardware-specialized packages**



Sandia National Laboratories

Recent and In-Progress Enhancements to Programming Model Abstractions: Spaces, Policies, Defaults, C++11, and Tasks

Execution Space

Sandia National Laboratories

- Execution Space *Instance*
  - Hardware resources (e.g., cores, hyperthreads) in which functions execute
  - Functions may execute concurrently on those resources
  - Concurrently executing functions have coherent view to memory
  - Degree of potential concurrency determined at runtime
  - Number of execution space instances determined at runtime
- Execution Space *Type* (CPU, Xeon Phi, CUDA)
  - Functions compiled to execute on an instance of a specified type
  - Types determined at configure/compile time
- Host Space
  - The main process and its functions execute in the Host Space
  - One type, one instance, and is serial (potential concurrency == 1)
- Execution Space *Default*
  - Configure/build with one type – it is the default
  - Initialize with one instance – it is the default

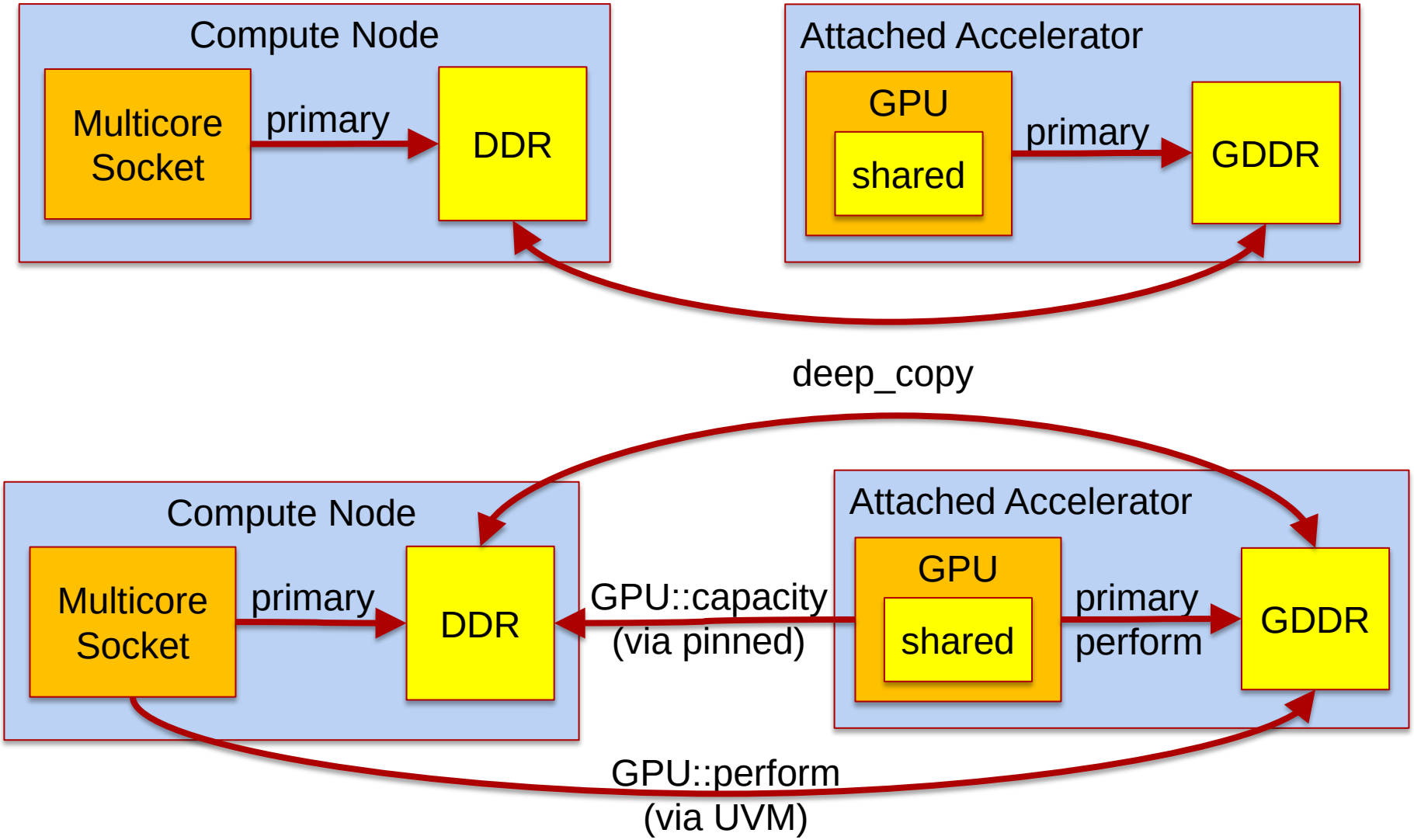
Memory Spaces

Sandia National Laboratories

- Memory Space *Types* (GDDR, DDR, NVRAM, Scratchpad)
  - The *type* of memory is defined with respect to an execution space type
  - Anticipated types, identified by their dominant usage
  - Primary: (default) space with allocable memory (e.g., can malloc/free)
    - Performant : best performing space (e.g., GDDR)
    - Capacity : largest capacity space (e.g., DDR)
    - Contemporary system: Primary == Performant == Capacity
  - Scratch : non-allocable *and* maximum performance
  - Persistent : usage can persist between process executions (e.g., NVRAM)
- Memory Space *Instance*
  - Has relationship with execution space instances (more later)
  - Directly addressable by functions in that execution space
  - Contiguous range of addresses
- Memory Space *Default*
  - Default execution spaces' default memory space

Examples of Execution and Memory Spaces

Sandia National Laboratories



Execution / Memory Space Relationships

Sandia National Laboratories

- (Execution Space , Memory Space , Memory Access Traits)
  - Accessibility : functions can/cannot access memory space
    - E.g., Host functions can never access GPU scratch memory
    - E.g., GPU functions can access Host capacity memory only if it is pinned
    - E.g., Host functions can access GPU performant memory only if it is UVM
  - Readable / Writeable
    - E.g., GPU performant memory using texture cache is read-only
  - Bandwidth : potential rate at which concurrent instructions can read or write
  - Capacity for views to (allocable) data
- Memory Access Traits (extension point) potential examples:
  - read-only, write-only, volatile/atomic, random, streaming, ...
  - Converting between "views" with same space and different traits
  - ➢ Default is simple readable/writeable – no special traits
- Future opportunity
  - Execution space access to remote memory space (similar to MPI 1-sided)

Views, Defaults, and Subviews

Sandia National Laboratories

- typedef View< ArrayType , Layout , Space , Traits > view_type ;
  - Omit Traits : no special compile-time defined access traits
  - Omit Space : default execution space's default memory space
  - Omit Layout : allocable memory space's default layout
  - default everything: View< ArrayType >
- view_type a(optional_traits , N0 , N1 , ...);
  - optional_traits : a collection of optional runtime defined traits
  - label trait : string used in error and warning messages, default none
  - initialize trait : default parallel_for(N0,[=](int i){ a(i,...) = 0 ; })
    - Default uses memory space's preferred execution space with static scheduling
    - Common override is to not initialize after allocating
- dst_view = subview< DestViewType >(src_view , ...args...)
  - Subviews of views increasingly important to users
  - Growing capability, challenging with polymorphic layout
  - C++11 'auto' type would help address this challenge

Execution Policy

Sandia National Laboratories

- How Potentially Concurrent Functions are Executed
  - Where : in what execution space (instance & type)
  - Parallel Work: current capabilities [0..N) or (#teams, #thread/team)
  - Scheduling : currently static scheduling of data parallel work
  - Map work function calls onto resources of the execution space
    - E.g., contiguous spans of [0..N) to a CPU thread for contiguous access pattern
    - E.g., strided subsets of [0..N) to GPU threads for coalesced access pattern
- Compose Pattern & Policy : parallel_for(policy , functor);
  - Policy::execution_space to replace Functor::device_type
  - Allows functor to be a C++11 lambda without impeding flexibility
  - Default Policy and Space for Simple Functors
  - Policy 'size_t N' is [0..N) with static scheduling and default execution space
  - E.g., parallel_for(N , [=](int i) { /* lambda-function body */ });

Execution Policies, Patterns, and Defaults

Sandia National Laboratories

- Patterns: parallel_for, parallel_reduce, parallel_scan
- parallel_*pattern*(policy , functor);
  - Execute on policy's execution space according to policy's scheduling
  - functor API requirements defined by pattern and policy
  - functor API omissions have defaults
- parallel_reduce functor API requirements and defaults
  - functor::init(value_type & update); // { new(& update) value_type(); }
  - functor::join(volatile value_type & update ,
    volatile const value_type & in) const ; // { update += in ; }
  - functor::final(value_type & update) const ; // {;}
- parallel_scan functor has similar requirements and defaults

Defaults enable C++11 Lambda for Functors

Sandia National Laboratories

- **Dot product becomes simple with C++11 lambda with defaults**

```
double dot( View<double*> x , View<double*> y ) {
  double d = 0 ;
  parallel_reduce( x.dimension_0() , [=](int i, double & v) { v += x(i) * y(i); } , d );
  return d ;
}
```

- **Execution Policy – how to execute**
- **Execution Policy's Execution Space – where to execution**
  - **Default for a single type and instance**
- **Parallel reduce and scan defaults**
  - **Reduction type – deduced from lambda's argument list**
  - **Initialize – default constructor**
  - **Join – operator +=**
- **Expect Cuda / nvcc version 7 to support C++11 lambda**
  - **Portability!**
- **Anecdote: our experienced developers prefer functors**

Execution Policy – an extension point

Sandia National Laboratories

- Policy calls functor's work function in parallel
  - PolicyType<ExecSpace>::member_type // data parallel work item
    void Func::operator()(PolicyType<...>::member_type) const ;
- Range policy (existing)
  - parallel_for(RangePolicy<ExecSpace>(0,N) , functor);
    void Func::operator()(integer_type i) const ;
- Thread team policy (existing)
  - parallel_for(TeamPolicy<ExecSpace>(#teams,thread/team) , functor);
    void Func::operator()(TeamPolicy<ExecSpace>::member_type team) const ;
  - Replaces "device" interface
- Extension point for new policies
  - Multi-indices [0..M)x[0..N)
  - Dynamic scheduling / work stealing
- **Parallel execution over Raja-like index sets is an execution policy**

Execution Policy, Functor with multiple '()'

Sandia National Laboratories

- Allow functors to have multiple parallel work functions
  - typedef PolicyType< ExecSpace , TagType > policy ;
  - parallel_*pattern*(policy(...) , functor);

  void FunctorType::operator()(const TagType &, policy::member_type) const ;
  - Parallel work functions differentiated by 'TagType'
    - TagType used instead of class' method name
- Motivations
  - Algorithm (class) with multiple parallel passes using the same data
    - miniFENL sparse matrix graph construction from FEM connectivity
  - Common need in LAMMPS
    - allow LAMMPS to remove "wrapper functors"

Execution Policy for Task Parallelism

Sandia National Laboratories

- Kokkos/Qthreads LDRD
- TaskManager< ExecSpace > execution policy
  - Policy object shared by potentially concurrent tasks

```
TaskManager<...> tm( exec_space , ... );
Future<> fa = spawn( tm , task_functor_a ); // single-thread task
Future<> fb = spawn( tm , task_functor_b );
```

  - Tasks may be data parallel

```
Future<> fc = spawn_for( tm.range(0..N) , functor_c );
Future<value_type> fd = spawn_reduce( tm.team(N,M) , functor_d );
wait( tm ); // wait for all tasks to complete
```

  - Destruction of task manager object waits for concurrent tasks to complete
- Task Managers
  - Define a scope for a collection of potentially concurrent tasks
  - Have configuration options for task management and scheduling
  - Manage resources for scheduling queue

Execution Policy for Task Parallelism

Sandia National Laboratories

- **Tasks' execution dependences**
  - Start a task only after other specified tasks have completed

    ```
    Future<> array_of_dep[ M ] = { /* future for other specified tasks */ };
    ```

  - Single threaded task:

    ```
    Future<> fx = spawn( tm.depend(M,array_of_dep) , task_functor_x  );
    ```

  - Data parallel task:

    ```
    spawn_for( tm.depend(M,array_of_dep).range(0..N) , task_functor_y );
    ```

  - Tasks and dependences define a directed acyclic graph (dag)

- **At most one active task manager on an execution space**
  - Well-defined scope and lifetime for collection of potentially current tasks
  - Don't consume resources when not in use