

# Kokkos Library

## Enabling performance portability across manycore architectures

**H. Carter Edwards (presenting)**  
**Christian Trott and Daniel Sunderland**

**JOWOG-34 : August 8, 2013**  
**Unclassified Unlimited Release**



*Exceptional  
service  
in the  
national  
interest*



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

Diversity of devices and associated performance requirements

## Device Dependent Memory Access Patterns

- Performance heavily depends upon device specific requirements for memory placement, blocking, striding, ...
- CPUs with NUMA and vector units
  - Core-data affinity: first touch and consistent access
  - Alignment for cache-lines and vector units
- GPU Coalesced Access *with* cache-line alignment
- “Array of Structures” vs. “Structure of Arrays” ?
  - This has been the *wrong* question

Right question: Abstraction for Performance Portability ?

# Programming Model Concept

## Two foundational ideas

- **Manycore Device**
  - Distinct execution and memory spaces (physical or logical)
  - Dispatch parallel work to device : computation + data
- **Classic Multidimensional Arrays, *with a twist***
  - Map multi-index (i,j,k,...)  $\leftrightarrow$  memory location *on the device*
    - Efficient : index computation and memory use
  - Map is derived from an array Layout
  - Choose Layout for device-specific memory access pattern
  - Make layout changes transparent to the user code;
  - IF the user code honors the simple API:  $a(i,j,k,...)$

Separate user's index space from memory layout

# Kokkos Library (libraries)

- **Standard C++ Library, not a Language extension**
  - *In spirit of Intel's TBB, NVIDIA's Thrust & CUSP, MS C++AMP, ...*
  - *Not a language extension: OpenMP, OpenACC, OpenCL, CUDA*
- **Uses C++ template meta-programming**
  - *Compile-time polymorphism for devices and array layouts*
  - *C++1998 standard; would be nice to *require* C++2011 ...*
- **Kokkos is becoming a hierarchy of manycore-tuned libraries**
  - **CORE: Arrays and Parallel Dispatch**
  - **Containers: vector, unordered map, compressed sparse row**
  - **Sparse linear algebra kernels**
  - **? Mesh or grid**

# Core : Array Allocation, Access, and Layout

- Allocation and access

```
View< double * * [3][8] , Device > a("a",N,M);
```

- Dimension [N][M][3][8] ; two runtime, two compile-time
  - a(i,j,k,l)** : access data via multi-index with device-specific map
- 'View' API is the same for both host and device code
- Access checks correctness
  - Host ↔ device memory access – catch error before “seg fault”
  - Runtime array bounds checking – in debug mode
    - Capability on the GPU as well
- View semantics (shared pointer semantics)
  - Multiple view objects reference the same array; a.k.a., shared ownership
  - Last view deallocates array data

# Core : Allocation, Access, and Layout

- Advanced : specify array layout

```
View<double**[3][8], Layout , Device> a("a",N,M);
```

- Override default layout; e.g., force row-major or column-major
  - Multi-index access is unchanged in user code
- *Layout* is an extension point for blocking, tiling, etc.

- Advanced : specify memory access attributes

```
View< const double**[3][8], Device, RandomRead > x = a ;
```

- Use special hardware, if available
  - E.g., access 'x' data through GPU texture cache
- Advanced : integrate aggregate 'scalar' types into the layout
    - Stochastic variables
    - Automatic differentiation variables

# Core : Deep Copy

**NEVER** have a hidden, expensive deep-copy

- Deep-copy *only* when explicitly instructed by user code
- Basic : mirror the layout in Host memory space
  - **Avoid transpose or permutation of data: simple, fast deep-copy**

```
typedef class View<...,Device> MyViewType ;  
MyViewType a("a",...);  
MyViewType::HostMirror a_host = create_mirror( a );  
deep_copy( a , a_host ); deep_copy( a_host , a );
```

- Advanced : avoid unnecessary deep-copy

```
MyViewType::HostMirror a_host = create_mirror_view( a );
```

- If Device uses host memory then 'a\_host' is simply a view of 'a'
- **deep\_copy becomes a no-op**

# Core : Parallel Dispatch

## `parallel_for( nwork , functor )`

- **Functor : Function + its calling arguments**

```
template< class DeviceType > // template on device type
```

```
struct AXPY {
```

```
    typedef DeviceType device_type ; // run on this device
```

```
    void operator()(int iw) const { Y(iw) += A * X(iw); } // shared function
```

```
    const double A ;
```

```
    const View<const double*,device_type> X ;
```

```
    const View<          double*,device_type> Y ;
```

```
};
```

```
parallel_for( nwork , AXPY<device>( a , x , y ) ); // parallel dispatch
```

- **Functor is shared and called by NP threads ( $NP \leq nwork$ )**
- **Thread parallel call to 'operator()(iw)' :  $iw \in [0, nwork)$**
- **Access array data with 'iw' to avoid race conditions**



# Core : Parallel Dispatch

## `parallel_reduce( nwork , functor , result )`

- Similar to `parallel_for`, with a *Reduction Argument*

```
template< class DeviceType >
```

```
struct DOT {
```

```
    typedef DeviceType  device_type ;
```

```
    typedef double value_type ; // reduction value type
```

```
    void operator()( int iw , value_type & contrib ) const
```

```
        { contrib += y(iw) * x(iw); } // this thread's contribution
```

```
    const View<const double*,device_type> x , y ;
```

```
    // ... to be continued ...
```

```
};
```

```
parallel_reduce( nwork , DOT<device>(x,y) , result ); }
```

➤ Value type can be a scalar, 'struct', or dynamic array

- Result is a value or View to a value on the device

# Core : Parallel Dispatch

## `parallel_reduce( nwork , functor , result )`

- Initialize and join threads' individual contributions

```
struct DOT { // ... continued ...
```

```
    static void init( value_type & contrib ) { contrib = 0 ; }
```

```
    static void join( volatile value_type & contrib ,  
                    volatile const value_type & input )
```

```
    { contrib = contrib + input ; }
```

```
};
```

- Join threads' contrib via `Functor::join`
  - 'volatile' to prevent compiler from optimizing away the join
- Deterministic result ← highly desirable
  - Given the same device and # threads
  - Aligned memory prevents variations from vectorization

# Core : Advanced Parallel Dispatch (under development)

```
template< class DeviceType >
struct MyFunctor {
    void operator()( DeviceType dev ) const ; // shared function
};
parallel_for( WorkRequest , MyFunctor<device>( ... ) ); // parallel dispatch
```

- DeviceType abstracts thread hierarchy, shared memory, ...
  - OpenMP 4.0 vocabulary: team of threads, league of teams
  - Teams work cooperatively using transient team-shared memory
  - Teams have synchronization primitives (e.g., barrier)
  - E.g., Cuda's grid-block-thread = league-team-thread
- WorkRequest *requests* league, team, shared memory sizes
  - Actual sizes may vary according to device's capabilities
  - E.g., maximum team size limited by NUMA, #cores, #hyperthreads

# Core : Atomic Operations

## wrapper around 'native' atomic operations

- NOT the C++11 'atomic<T>' functionality and interface
- Fundamental operations on intrinsic data types
  - 32 and 64 bit integer and floating point types,
  - `old_val = atomic_exchange( address, new_val );`
  - `atomic_compare_exchange_strong( address, old_val , new_val );`
    - If `*address == old_val` then exchange
  - `old_val = atomic_fetch_add( address , value );`
    - `old_val = *address ; *address += value ;`
- Likely to have non-deterministic results ← **warning!**
  - Non-deterministic ordering of atomic operations
  - Floating point addition is NOT associative

# Core : Devices

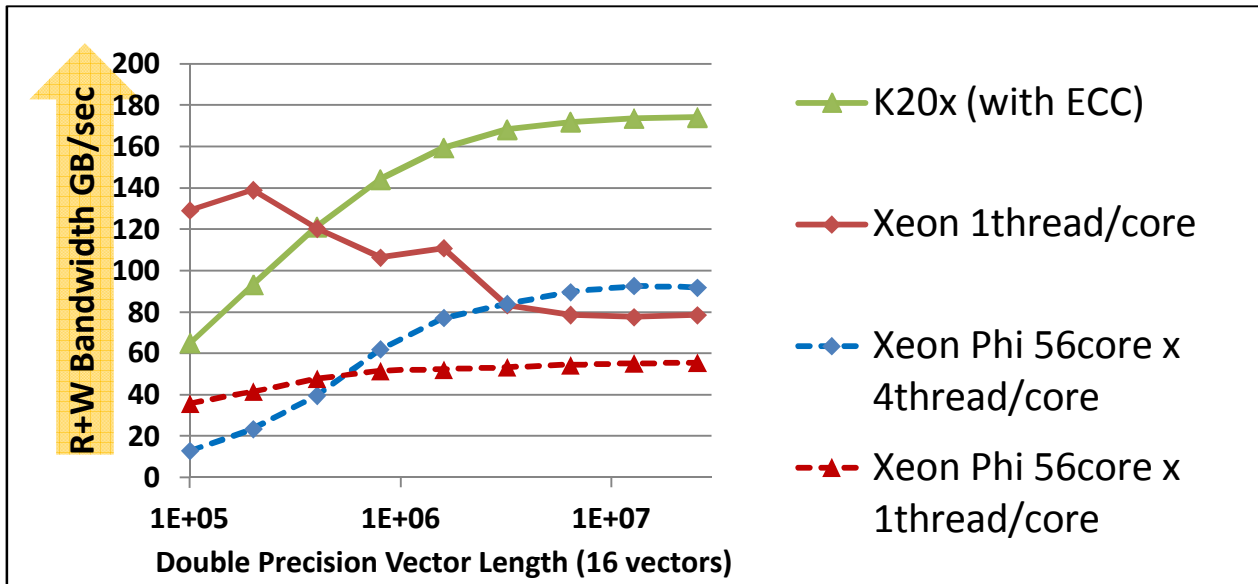
- **‘Threads’ Device : pthreads or C11 threads**
  - Pool of threads created once and pinned to cores
  - Hardware detection and core pinning via hardware locality library (hwloc)
  - Fan-in collective for deterministic reductions
  - Teams cannot span NUMA regions
- **‘OpenMP’ Device : wrapper on OpenMP**
  - Attempt to pin to cores via hwloc
  - Cannot use both ‘Threads’ and ‘OpenMP’ – compete for cores
- **‘Cuda’ Device : wrapper on NVidia’s CUDA 5.0 (or better)**
  - Currently require Fermi (or better); eventually require Kepler
  - Unified Virtual Memory (UVM) capability will define more memory spaces
    - Device resident and host accessible
    - Host resident and device accessible

# Performance Evaluation

- **Using Sandia Computing Research Center Testbed Clusters**
  - **Compton: 32nodes**
    - 2x Intel Xeon E5-2670 (Sandy Bridge), hyperthreading enabled
    - 2x Intel Xeon Phi 57core (pre-production)
    - ICC 13.1.2, Intel MPI 4.1.1.036
  - **Shannon: 32nodes**
    - 2x Intel Xeon E5-2670, hyperthreading disabled
    - 2x NVidia K20x
    - GCC 4.4.5, Cuda 5.5, MVAPICH2 v1.9 with GPU-Direct
- **Absolute performance “unit” tests**
  - Evaluate parallel dispatch/synchronization efficiency
  - Evaluate impact of array access patterns and capabilities
- **Mini-application : Kokkos vs. ‘native’ implementations**
  - Evaluate cost of portability

# Performance Test: Modified Gram-Schmidt

## Simple stress test for bandwidth and reduction efficiency



Intel Xeon: E5-2670 w/HT  
Intel Xeon Phi: 57c @ 1.1GHz  
Nvidia K20x

*Results presented here are for pre-production Intel Xeon Phi co-processors (codenamed Knights Corner) and pre-production versions of Intel's Xeon Phi software stack. Performance and configuration of the co-processors may be different in final production releases.*

- Simple sequence of vector-reductions and vector-updates
  - To orthonormalize 16 vectors
- Performance for vectors > L3 cache size
  - NVIDIA K20x : 174 GB/sec = ~78% of theoretical peak
  - Intel Xeon : 78 GB/sec = ~71% of theoretical peak
  - Intel Xeon Phi : 92 GB/sec = ~46% of achievable peak

# Performance Test: Molecular Dynamics

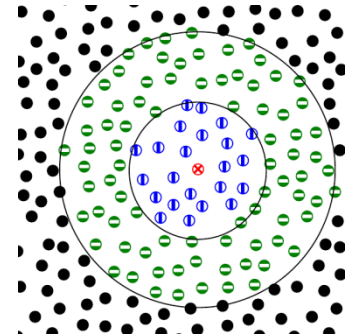
## Lennard Jones force model using atom neighbor list

- Solve Newton's equations for  $N$  particles

- Simple Lennard Jones force model: 
$$F_i = \sum_{j, r_{ij} < r_{cut}} 6 \epsilon \left[ \left( \frac{s}{r_{ij}} \right)^7 - 2 \left( \frac{s}{r_{ij}} \right)^{13} \right]$$

- Use 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;
```



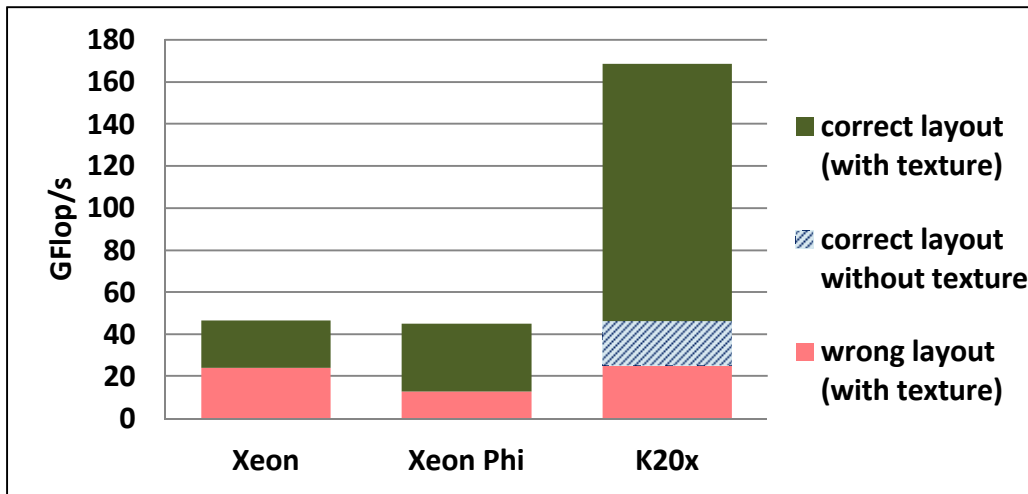
- Moderately compute bound computational kernel
- On average 77 neighbors with 55 inside of the cutoff radius



# Performance Test: Molecular Dynamics

## Lennard Jones (LJ) force model using atom neighbor list

- Test Problem (#Atoms = 864k, ~77 neighbors/atom)
  - Neighbor list array with correct vs. wrong layout
    - Different layout between CPU and GPU
  - Random read of neighbor coordinate via GPU texture fetch



Intel Xeon: E5-2670 w/HT  
Intel Xeon Phi: 57c @ 1.1GHz  
NVidia K20x

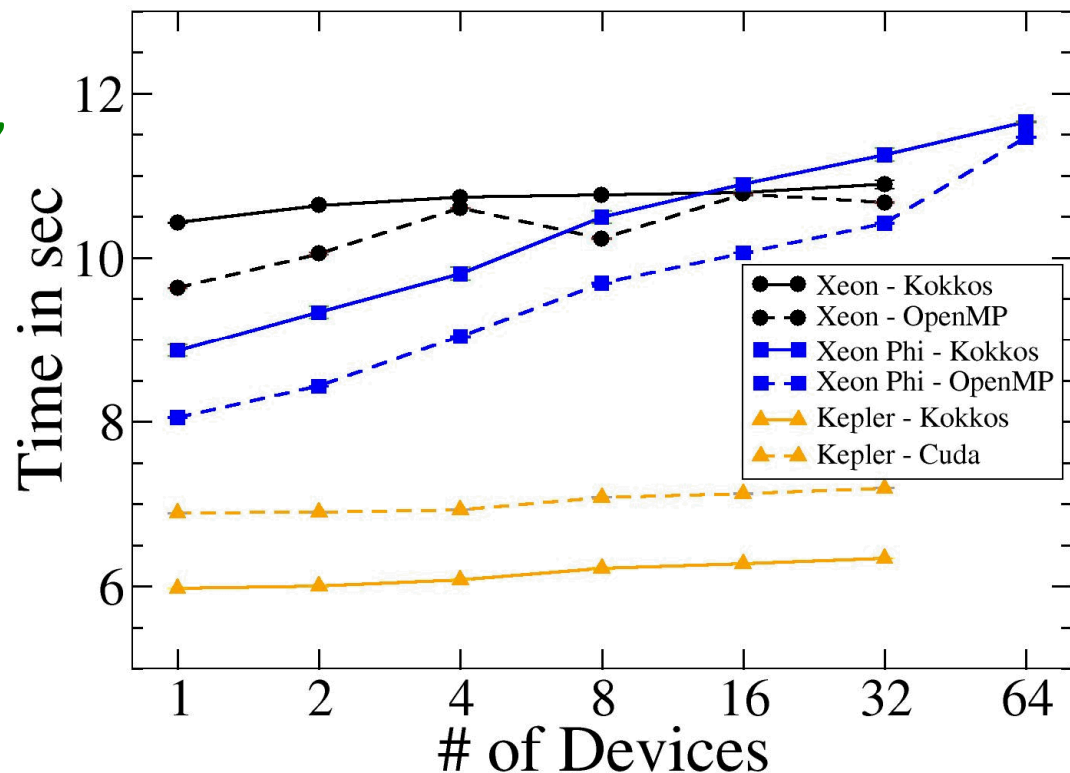
*Results presented here are for pre-production Intel Xeon Phi co-processors (codenamed Knights Corner) and pre-production versions of Intel's Xeon Phi software stack. Performance and configuration of the co-processors may be different in final production releases.*

- Large loss in performance with wrong layout
  - Even when using GPU texture fetch

# MPI+X Performance Test: MiniFE

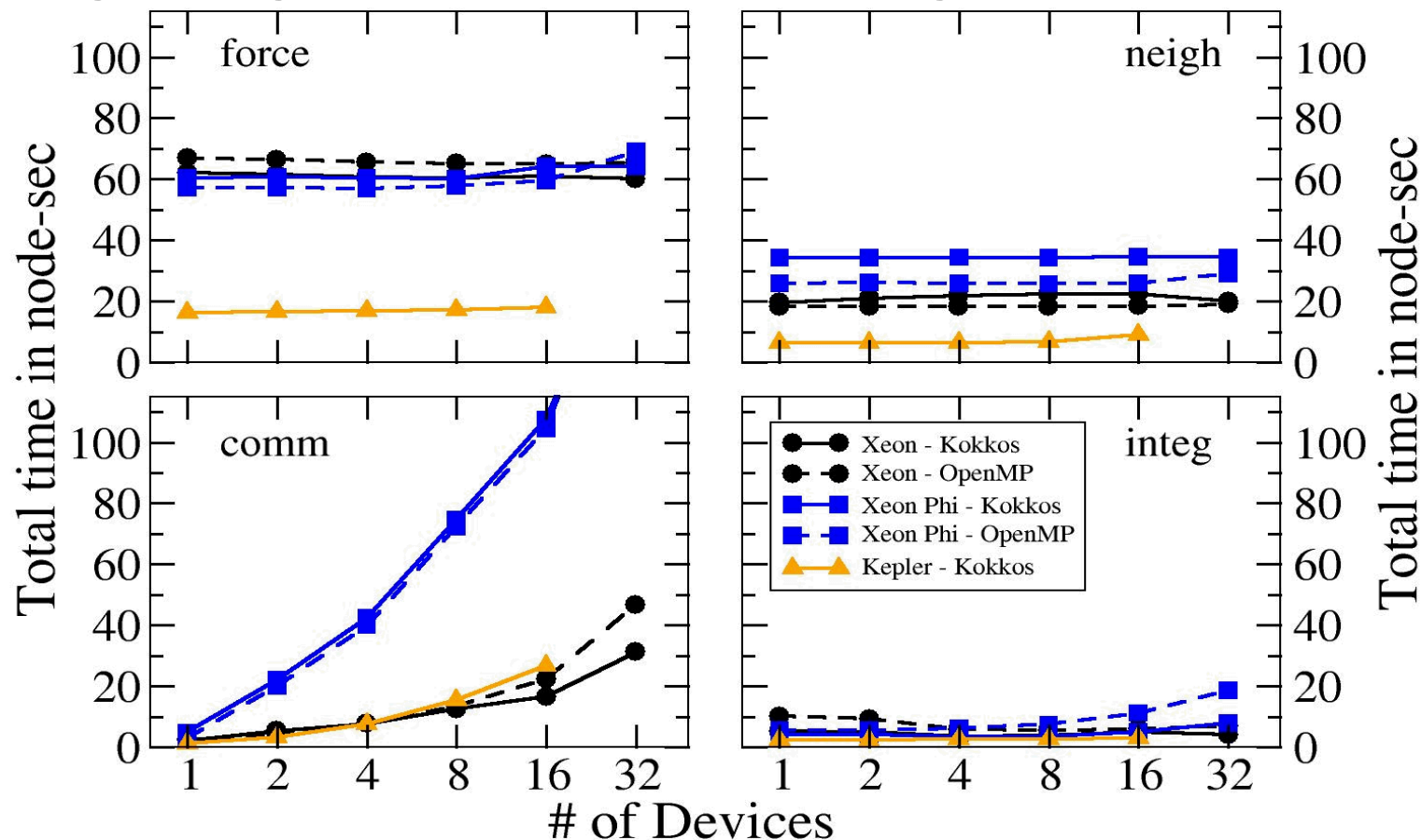
## Conjugate Gradient Solve of a Finite Element Matrix

- Comparing X = Kokkos, OpenMP, Cuda (GPU-direct via MVAPICH2)
- Weak scaling with one MPI process per device
  - Except on Xeon: OpenMP requires one process/socket due to NUMA
  - 8M elements/device
- Kokkos performance
  - 90% or better of “native”
  - Improvements ongoing



# MPI+X Performance: MiniMD

- Comparing X = OpenMPI vs. Kokkos , one MPI process / device
  - Using GPU-direct via MVAPICH2; no native Cuda version to compare
- Strong scaling test: 864k atoms, ~77 neighbors/atom



# Incremental Migration Strategy

## For C++ Applications & Libraries

- Replace array allocations with Views (in Host space)
  - Specify layout(s) to match existing array layout(s)
  - Extract pointers to allocated array data and use them in legacy code
- Replace array access with Views
  - Replace legacy array data structure(s) with View
  - Access data members via View API
- Replace functions with Functors, run in parallel on Host
  - Hard part: finding and extracting your functions' hidden states
    - improve code quality
  - Hard part: finding and fixing remaining thread-unsafe (race) conditions
    - most easily using atomic operations
- Set device to 'Cuda' and run on GPU
  - Hard part: thread scalability, some functors may require redesign

## Manycore Performance Portability

- Solved: “array of structs” vs. “struct of arrays” ?
  - The right question: what abstractions are required ?
  - Answer: multidimensional arrays with device-polymorphic layout
  - and coordinated parallel dispatch of computational kernels
- Kokkos C++ core library, not a language extension
  - Performance evaluation “unit tests” and mini-applications
  - Multicore CPU, NVidia GPU, Intel Xeon Phi coprocessor
  - 90% or better of device-specialized “native” implementation
- Plans : First release Sept’13 (next month)
  - Enable Trilinos MPI+X through Tpetra
  - Researching hierarchical task-data parallelism
- Applications are investigating Kokkos for MPI+X migration
  - LAMMPS
  - Climate modeling LDRD