LA-UR-16-27057
Approved for public release; distribution is unlimited.

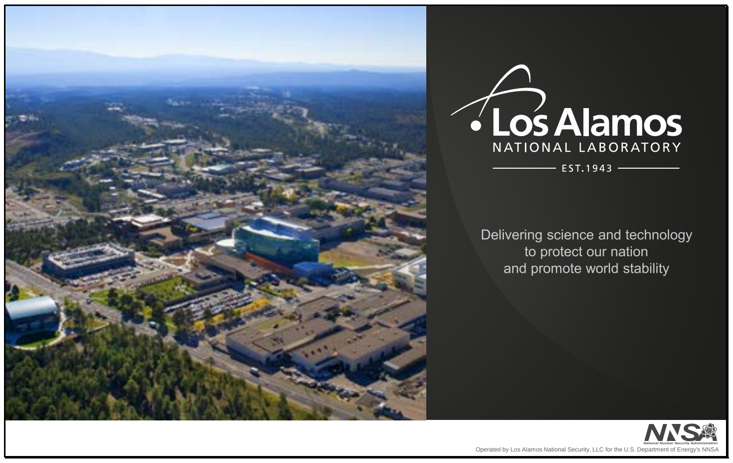
Title: FY16 ASC ATDM L2 Milestone : PARTISN Research and FleCSI Updates

Author(s): Womeldorff, Geoffrey Alan
Payne, Joshua Estes
Bergen, Benjamin Karl

Intended for: Report

Issued: 2016-09-15

Slide 1



Los Alamos
NATIONAL LABORATORY
EST. 1943

Delivering science and technology
to protect our nation
and promote world stability

Slide 2

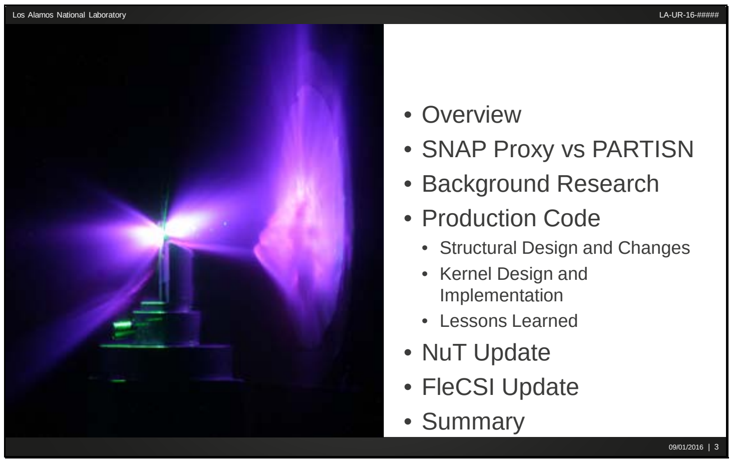
# FY16 ASC ATDM L2 Milestone

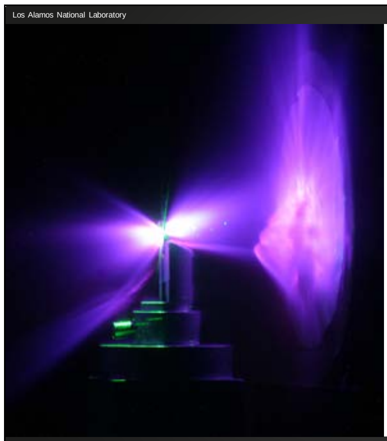
PARTISN Research and FleCSI Updates

**LANL Co-Design Team**

2016/09/01

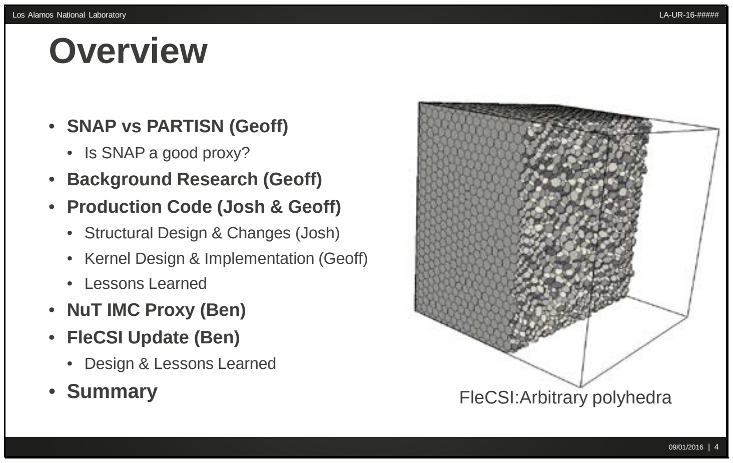
Slide 3





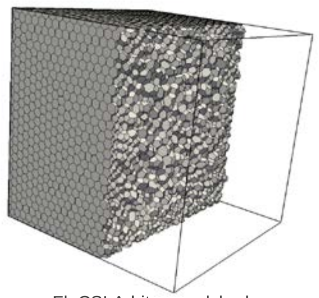
- Overview
- SNAP Proxy vs PARTISN
- Background Research
- Production Code
  - Structural Design and Changes
  - Kernel Design and Implementation
  - Lessons Learned
- NuT Update
- FleCSI Update
- Summary

Slide 4



## Overview

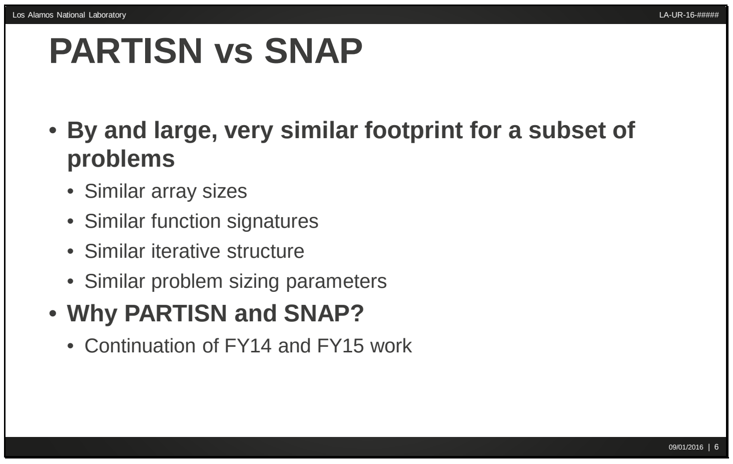
- **SNAP vs PARTISN (Geoff)**
  - Is SNAP a good proxy?
- **Background Research (Geoff)**
- **Production Code (Josh & Geoff)**
  - Structural Design & Changes (Josh)
  - Kernel Design & Implementation (Geoff)
  - Lessons Learned
- **NuT IMC Proxy (Ben)**
- **FleCSI Update (Ben)**
  - Design & Lessons Learned
- **Summary**


FleCSI:Arbitrary polyhedra

Slide 5

## SNAP Proxy for PARTISN

Slide 6

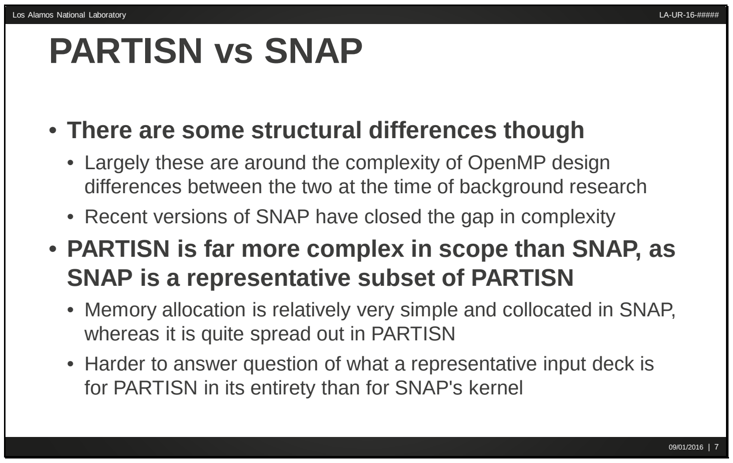


## PARTISN vs SNAP

- **By and large, very similar footprint for a subset of problems**
  - Similar array sizes
  - Similar function signatures
  - Similar iterative structure
  - Similar problem sizing parameters
- **Why PARTISN and SNAP?**
  - Continuation of FY14 and FY15 work

**SNAP is PARTISN without decades of development accretion and without the physics. FY14 efforts showed they are very similar in terms of flops/load. The "kernel" is very similar between the two codes.**

Slide 7


Los Alamos National Laboratory LA-UR-16-#####

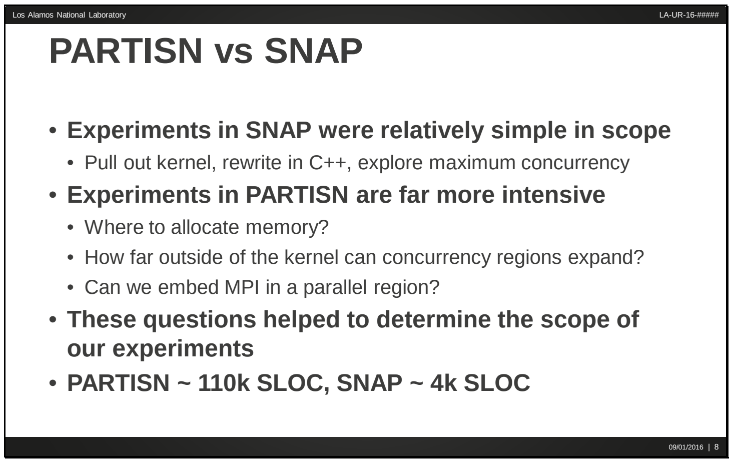
## PARTSN vs SNAP

- **There are some structural differences though**
  - Largely these are around the complexity of OpenMP design differences between the two at the time of background research
  - Recent versions of SNAP have closed the gap in complexity
- **PARTISN is far more complex in scope than SNAP, as SNAP is a representative subset of PARTISN**
  - Memory allocation is relatively very simple and collocated in SNAP, whereas it is quite spread out in PARTISN
  - Harder to answer question of what a representative input deck is for PARTISN in its entirety than for SNAP's kernel

09/01/2016 | 7

**However, PARTISN is more complex, and so the amount of changes able to be completed with available resources are smaller in scope than SNAP. The technical decision was made to focus on the interface layer and the C++ kernel as both of those would offer lasting value to the production code team. This limited available performance, by going with (the current version of) Kokkos versus native CUDA, and by not rewriting the threading section of PARTISN due to lack of man hours. The constraint, though, enabled the successful resolution of the L2 completion criteria.**

Slide 8



Los Alamos National Laboratory LA-UR-16-#####

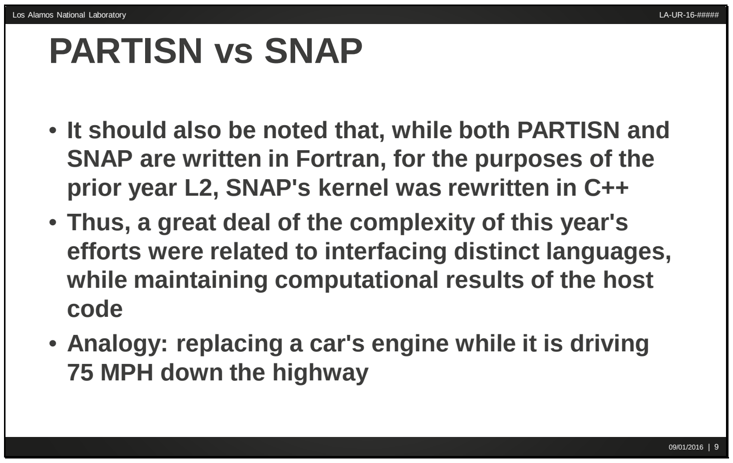
## PARTISN vs SNAP

- **Experiments in SNAP were relatively simple in scope**
  - Pull out kernel, rewrite in C++, explore maximum concurrency
- **Experiments in PARTISN are far more intensive**
  - Where to allocate memory?
  - How far outside of the kernel can concurrency regions expand?
  - Can we embed MPI in a parallel region?
- **These questions helped to determine the scope of our experiments**
- **PARTISN ~ 110k SLOC, SNAP ~ 4k SLOC**

09/01/2016 | 8

**In working with SNAP, there was a simple translation of kernel with generic input argument sets. In working with PARTISN, we worked backwards from the kernel arguments (both function signature and implicit via module), to trace memory allocation to its source. In addition, threaded regions in PARTISN were more complex, and, since working with a production code, and not a proxy, maintaining the convergence behaviour of the code was mandatory. With MPI embedded at the thread-level in PARTISN, and Kokkos not supporting (currently) multi-threaded access to CUDA-UVM, our choices were to either allow only single threaded behaviour, and use CUDA-UVM through Kokkos to prevent memory duplication (UVM is host-backed GPU memory that can be shared between Fortran and C++), or to allow multi-threaded access through native CUDA which would have a good chance of faster computational performance, but would greatly increase our engineering scope. An even greater chance of performance increase would have come from broadening the scope of concurrency explored to the start of the threaded region in PARTISN. However, this would have required embedding MPI in threaded regions on the CPU, and/or the GPU, which would in turn greatly increase our engineering scope and downselect again from our limited toolchain options.**

Slide 9


Los Alamos National Laboratory

LA-UR-16-#####

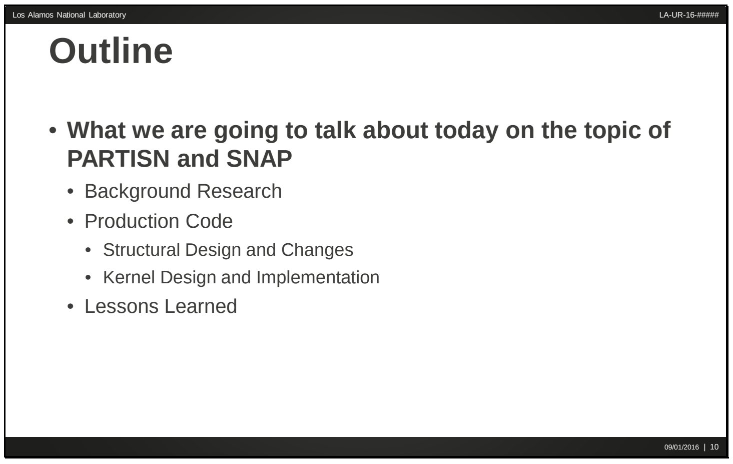
## PARTISN vs SNAP

- It should also be noted that, while both PARTISN and SNAP are written in Fortran, for the purposes of the prior year L2, SNAP's kernel was rewritten in C++
- Thus, a great deal of the complexity of this year's efforts were related to interfacing distinct languages, while maintaining computational results of the host code
- Analogy: replacing a car's engine while it is driving 75 MPH down the highway

09/01/2016 | 9

**Covered to some extent in prior two annotations. We were able to maintain convergence behaviour between the following three scenarios: 1) original Fortran version, 2) Fortran with C++ memory allocations (in CUDA-UVM through Kokkos), and 3) scenario 2 but running against the Kokkos opt3_sweep kernel. This added more kernel debugging complexity by requiring numerical results to be the same. The exercise helped to expose the intricacies and pitfalls in translating idiomatic programming logic between distinct programming languages.**

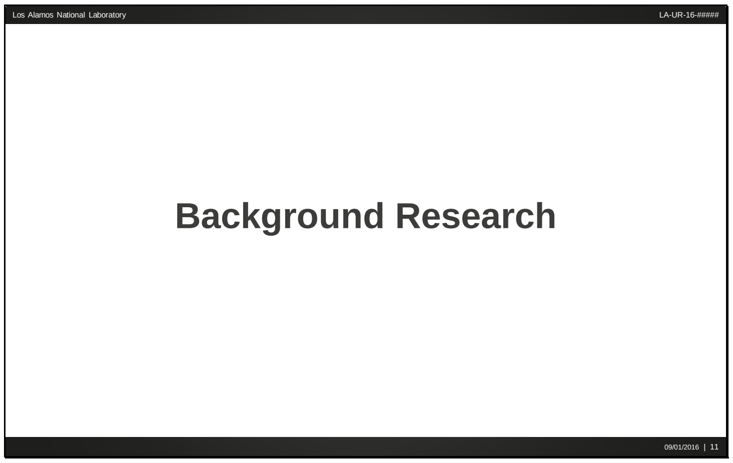
Slide 10



## Outline

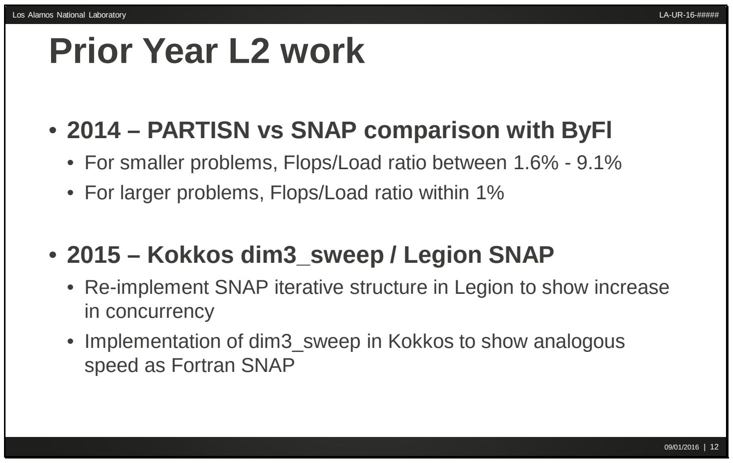
- **What we are going to talk about today on the topic of PARTISN and SNAP**
  - Background Research
  - Production Code
    - Structural Design and Changes
    - Kernel Design and Implementation
  - Lessons Learned

Slide 11



## Background Research

Slide 12


Los Alamos National Laboratory

LA-UR-16-#####

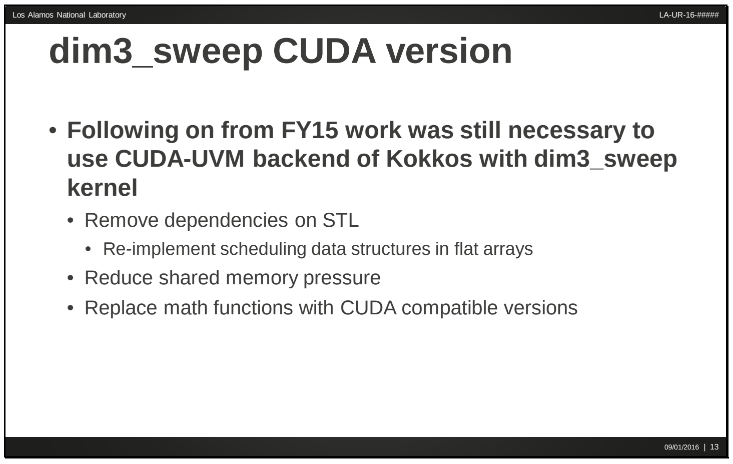
## Prior Year L2 work

- **2014 – PARTISN vs SNAP comparison with ByFl**
  - For smaller problems, Flops/Load ratio between 1.6% - 9.1%
  - For larger problems, Flops/Load ratio within 1%
- **2015 – Kokkos dim3_sweep / Legion SNAP**
  - Re-implement SNAP iterative structure in Legion to show increase in concurrency
  - Implementation of dim3_sweep in Kokkos to show analogous speed as Fortran SNAP

09/01/2016 | 12

**Above is a modicum of detail from prior year's L2s that led to this years work. First, PARTISN and its proxy SNAP were compared, and found to be similar at scale, at least in terms of Flops/Load. The next year, the concurrency of the KBA algorithm was compared to a Legion implementation of SNAP, and a Kokkos version of SNAP's hotspot, the dim3_sweep kernel, was created and studied.**

Slide 13

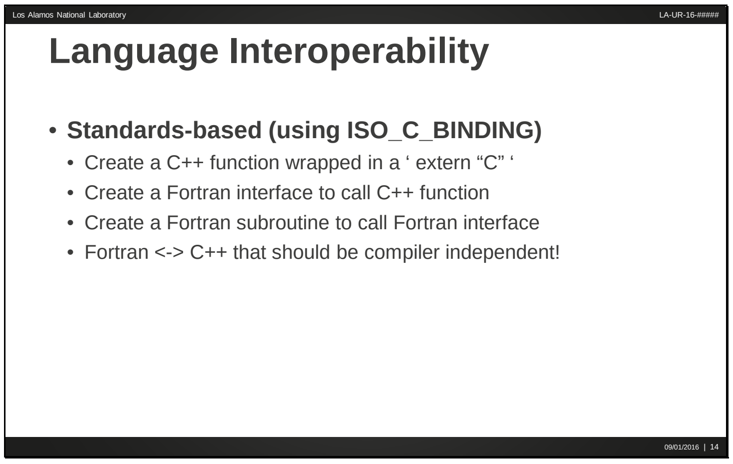


## dim3_sweep CUDA version

- **Following on from FY15 work was still necessary to use CUDA-UVM backend of Kokkos with dim3_sweep kernel**
  - Remove dependencies on STL
    - Re-implement scheduling data structures in flat arrays
  - Reduce shared memory pressure
  - Replace math functions with CUDA compatible versions

**As a follow on to last year's FY work, we continued to explore the dim3_sweep Kokkos kernel version. We made modifications, listed above, to allow it to function on a CUDA GPU. After running it on a GPU, it was found to have good amounts of speedup. However, this was with concurrency levels on the order of launching all energy groups of an octant sweep simultaneously.**

Slide 14


Los Alamos National Laboratory LA-UR-16-#####

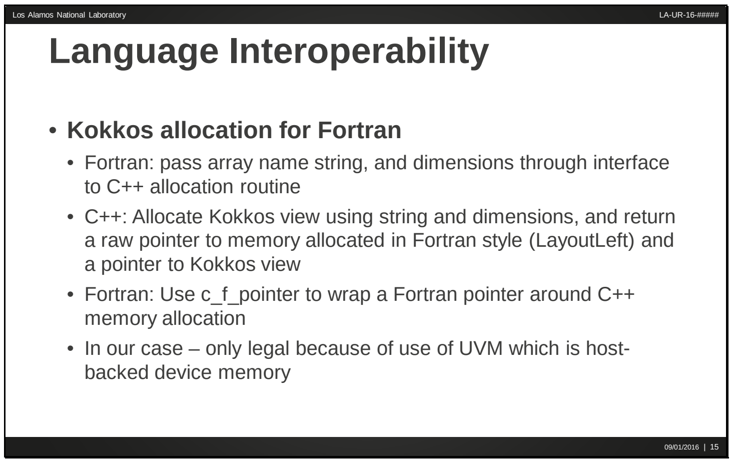
## Language Interoperability

- **Standards-based (using ISO_C_BINDING)**
  - Create a C++ function wrapped in a ‘ extern “C” ‘
  - Create a Fortran interface to call C++ function
  - Create a Fortran subroutine to call Fortran interface
  - Fortran <-> C++ that should be compiler independent!

09/01/2016 | 14

**We then developed a plan to fulfill the L2 completion criteria, by assessing how we would port the Kokkos work from SNAP to PARTISN. This would require some experimentation in programming language interoperability. We chose to use standards based interoperability in order to give ourselves the widest choice in compiler toolchains. All of the compiler toolchains we tested supported the ISO_C_BINDING interoperability module in Fortran.**

Slide 15


Los Alamos National Laboratory LA-UR-16-#####

## Language Interoperability

- **Kokkos allocation for Fortran**
  - Fortran: pass array name string, and dimensions through interface to C++ allocation routine
  - C++: Allocate Kokkos view using string and dimensions, and return a raw pointer to memory allocated in Fortran style (LayoutLeft) and a pointer to Kokkos view
  - Fortran: Use c_f_pointer to wrap a Fortran pointer around C++ memory allocation
  - In our case – only legal because of use of UVM which is host-backed device memory

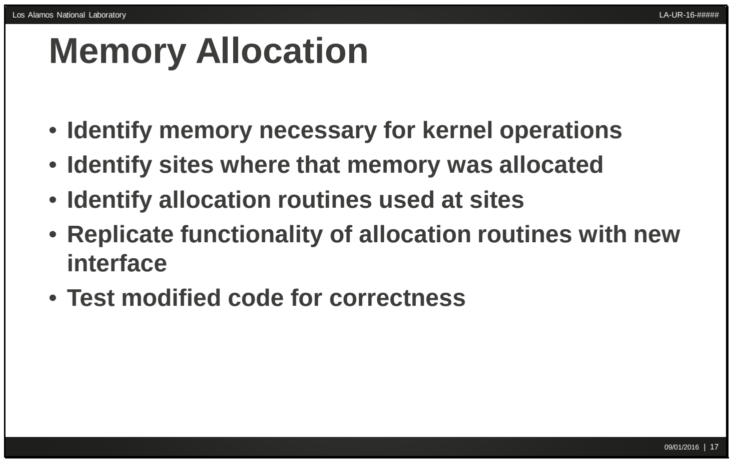
09/01/2016 | 15

**We tested a proof-of-concept experiment by creating code that would, as above, pass an array name string and dimensions to C++, Kokkos allocates C++ memory using the dimensions and string in a Fortran compatible layout,  C++ passes pointers to the view object created by Kokkos and a pointer to the memory allocated as its backing store back to Fortran, and then Fortran wraps the raw memory pointer in a Fortran pointer, and uses the C++ allocation as it would a native one.**

Slide 16

## Production Code: Structural Design and Changes

Slide 17


Los Alamos National Laboratory

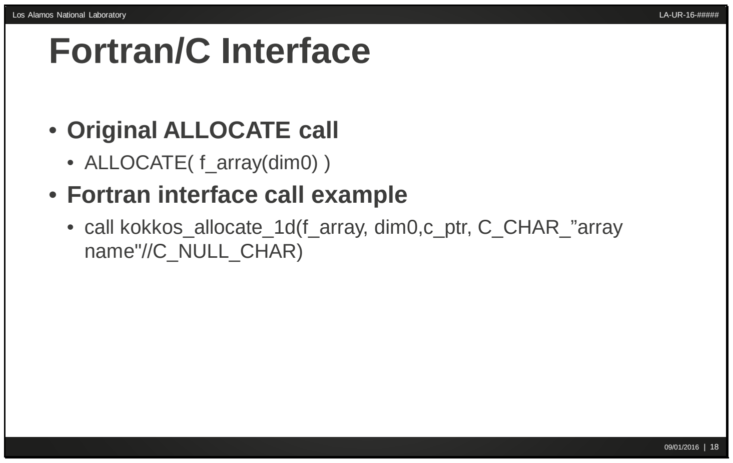
LA-UR-16-#####

## Memory Allocation

- Identify memory necessary for kernel operations
- Identify sites where that memory was allocated
- Identify allocation routines used at sites
- Replicate functionality of allocation routines with new interface
- Test modified code for correctness

09/01/2016 | 17

**From the structural changes side of the house, this was our plan. We started by identifying the memory that would need to be used by a GPU kernel. We traced that memory (read: arrays) back to its allocation sites, and made note of the native routines which allocated them. With the detective work completed, we replicated the functionality of the allocation routines with our own (substituting CPU allocated Fortran memory, for CUDA-UVM allocated by C++). Once the new allocations were in place, we tested the modified code with the new memory and the Fortran version of the kernel (opt3_sweep).**

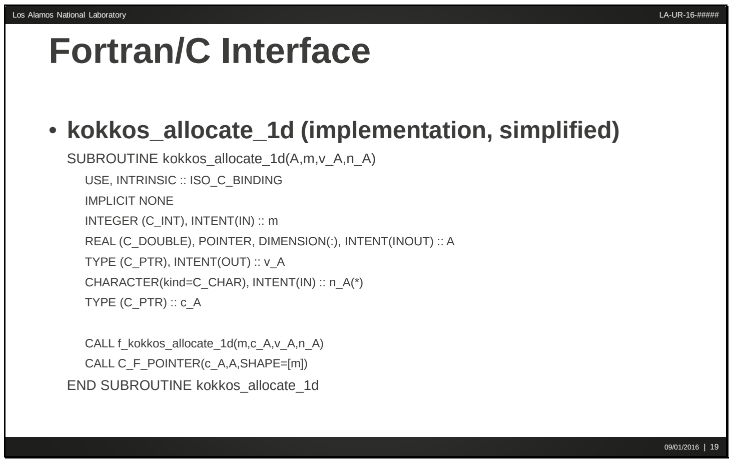


## Fortran/C Interface

- **Original ALLOCATE call**
  - ALLOCATE( f_array(dim0) )
- **Fortran interface call example**
  - call kokkos_allocate_1d(f_array, dim0,c_ptr, C_CHAR_"array name"//C_NULL_CHAR)

**Here is an example of the interface (bottom) and invocation of that interface (top) for allocating a 1D array. The allocation routine interface takes the dimensions (single, in this case) and a string of the array name, and returns a C pointer to the raw allocation (only safe on GPU memory because it is CPU backed UVM) and a C pointer to the view (retained to pass back in as a kernel argument later).**

Slide 19



Los Alamos National Laboratory LA-UR-16-#####

## Fortran/C Interface

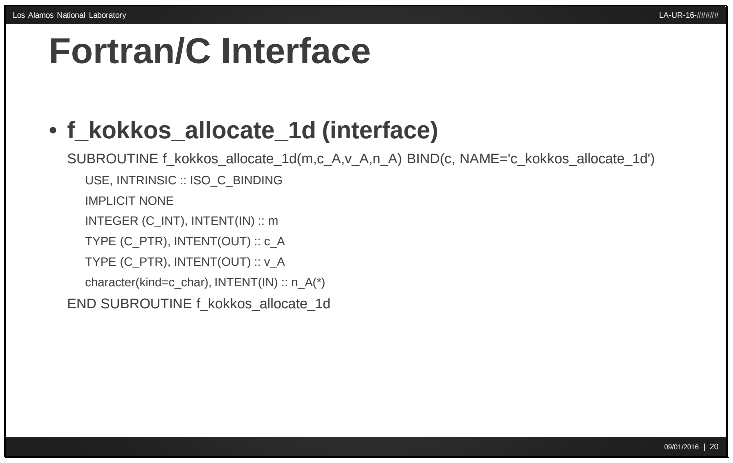
- **kokkos_allocate_1d (implementation, simplified)**

```
SUBROUTINE kokkos_allocate_1d(A,m,v_A,n_A)
   USE, INTRINSIC :: ISO_C_BINDING
   IMPLICIT NONE
   INTEGER (C_INT), INTENT(IN) :: m
   REAL (C_DOUBLE), POINTER, DIMENSION(:), INTENT(INOUT) :: A
   TYPE (C_PTR), INTENT(OUT) :: v_A
   CHARACTER(kind=C_CHAR), INTENT(IN) :: n_A(*)
   TYPE (C_PTR) :: c_A

   CALL f_kokkos_allocate_1d(m,c_A,v_A,n_A)
   CALL C_F_POINTER(c_A,A,SHAPE=[m])
END SUBROUTINE kokkos_allocate_1d
```

09/01/2016 | 19

**The implementation of the 1D allocate routine the pervious function being called, followed by an intrinsic Fortran subroutine c_f_pointer, of ISO_C_BINDING interoperability, which causes a Fortran pointer to be assigned to point to a C pointer (here, the C pointer points to the CPU side of the UVM memory allocation) given both pointers and the array shape. It is important to note that this requires that the C pointer point to memory which is laid out in the same fashion as would Fortran lay out a multidimensional array, as Fortran does not have the same facilities as C with respect to arbitrary layouts of multidimensional arrays.**

Slide 20



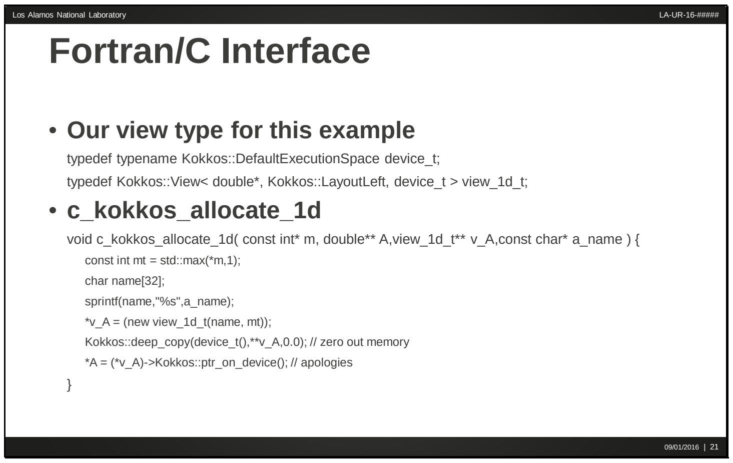
## Fortran/C Interface

- **f_kokkos_allocate_1d (interface)**

```
SUBROUTINE f_kokkos_allocate_1d(m,c_A,v_A,n_A) BIND(c, NAME='c_kokkos_allocate_1d')
    USE, INTRINSIC :: ISO_C_BINDING
    IMPLICIT NONE
    INTEGER (C_INT), INTENT(IN) :: m
    TYPE (C_PTR), INTENT(OUT) :: c_A
    TYPE (C_PTR), INTENT(OUT) :: v_A
    character(kind=c_char), INTENT(IN) :: n_A(*)
END SUBROUTINE f_kokkos_allocate_1d
```

**The implementation of the 1D allocate routine the pervious function being called, followed by an intrinsic Fortran subroutine c_f_pointer, of ISO_C_BINDING interoperability, which causes a Fortran pointer to be assigned to point to a C pointer (here, the C pointer points to the CPU side of the UVM memory allocation) given both pointers and the array shape. It is important to note that this requires that the C pointer point to memory which is laid out in the same fashion as would Fortran lay out a multidimensional array, as Fortran does not have the same facilities as C with respect to arbitrary layouts of multidimensional arrays.**

Slide 21



Los Alamos National Laboratory LA-UR-16-#####

## Fortran/C Interface

- **Our view type for this example**

```
typedef typename Kokkos::DefaultExecutionSpace device_t;
typedef Kokkos::View< double*, Kokkos::LayoutLeft, device_t > view_1d_t;
```

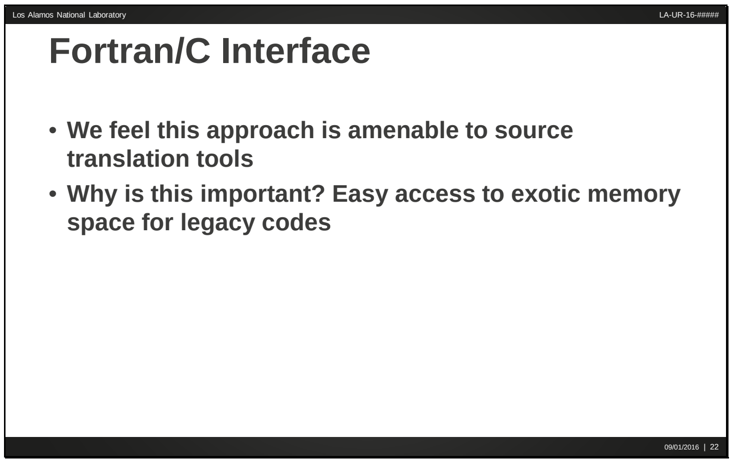
- **c_kokkos_allocate_1d**

```
void c_kokkos_allocate_1d( const int* m, double** A,view_1d_t** v_A,const char* a_name ) {
    const int mt = std::max(*m,1);
    char name[32];
    sprintf(name,"%s",a_name);
    *v_A = (new view_1d_t(name, mt));
    Kokkos::deep_copy(device_t(),**v_A,0.0); // zero out memory
    *A = (*v_A)->Kokkos::ptr_on_device(); // apologies
}
```

09/01/2016 | 21

**Here is the actual C++ allocation routine (1D shown). The variable mt, an integer, is created so that when a 0-length array dimension is passed from Fortran this routine will not error out (as 0 length allocations are legal in Fortran but not in C). Next the character array is stringified. Then, the string and array length are used to allocate the view. “deep_copy” is invoked to zero out the allocated memory on both sides. Finally, the raw pointer to the CPU portion of the UVM allocation is obtained and stored to be passed back to Fortran for use with c_f_pointer (as mentioned earlier). Not shown is that embedded in the “view_1d_t” type is the specification for memory allocation location (for this work, CUDA-UVM) and the layout order (LayoutLeft in Kokkos parlance) which maps to the Fortran memory order.**

Slide 22


Los Alamos National Laboratory LA-UR-16-#####

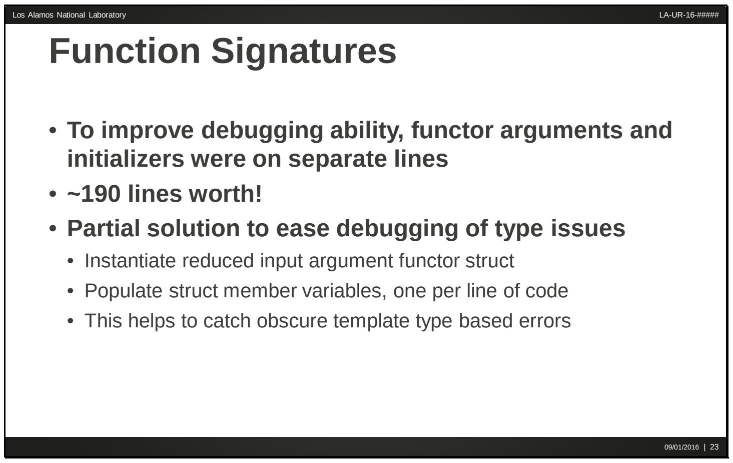
## Fortran/C Interface

- We feel this approach is amenable to source translation tools
- Why is this important? Easy access to exotic memory space for legacy codes

09/01/2016 | 22

**Here is the actual C++ allocation routine (1D shown). The variable mt, an integer, is created so that when a 0-length array dimension is passed from Fortran this routine will not error out (as 0 length allocations are legal in Fortran but not in C). Next the character array is stringified. Then, the string and array length are used to allocate the view. “deep_copy” is invoked to zero out the allocated memory on both sides. Finally, the raw pointer to the CPU portion of the UVM allocation is obtained and stored to be passed back to Fortran for use with c_f_pointer (as mentioned earlier). Not shown is that embedded in the “view_1d_t” type is the specification for memory allocation location (for this work, CUDA-UVM) and the layout order (LayoutLeft in Kokkos parlance) which maps to the Fortran memory order.**

Slide 23


Los Alamos National Laboratory LA-UR-16-#####

## Function Signatures

- **To improve debugging ability, functor arguments and initializers were on separate lines**
- **~190 lines worth!**
- **Partial solution to ease debugging of type issues**
  - Instantiate reduced input argument functor struct
  - Populate struct member variables, one per line of code
  - This helps to catch obscure template type based errors

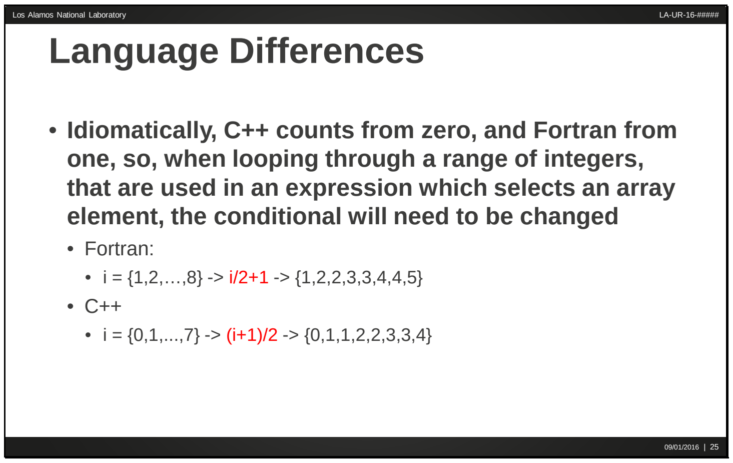
09/01/2016 | 23

**A combination of the fact that the were upwards of 90 arguments and approximately 10 template arguments led us to try to find ways to structure the kernel invocation in the C interface such that debugging difficulty was ameliorated somewhat. We had the idea to instantiate the functor struct using only the scalar values, then to populate the member Kokkos views one line at a time, after the struct was instantiated. This allowed us to much more easily catch template type mismatch errors in the member views.**

Slide 24

## Production Code:
## Kernel Design and Implementation

Slide 25

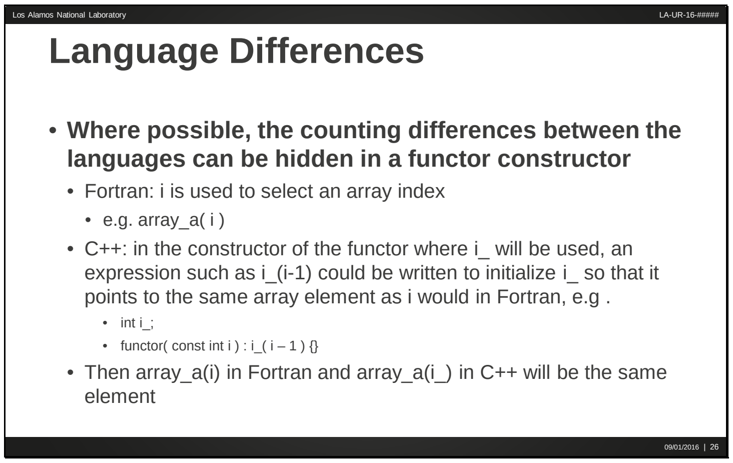


## Language Differences

- **Idiomatically, C++ counts from zero, and Fortran from one, so, when looping through a range of integers, that are used in an expression which selects an array element, the conditional will need to be changed**
  - Fortran:
    - i = {1,2,…,8} -> i/2+1 -> {1,2,2,3,3,4,4,5}
  - C++
    - i = {0,1,...,7} -> (i+1)/2 -> {0,1,1,2,2,3,3,4}

**It is a surprise to no one familiar with HPC that Fortran and C count differently. Usually this is straightforward when transliterating a piece of code from one to the other. More complicated is when a sequence produced by an expression must be changed to be "off by one". Above is an example of such.**

Slide 26


Los Alamos National Laboratory LA-UR-16-#####

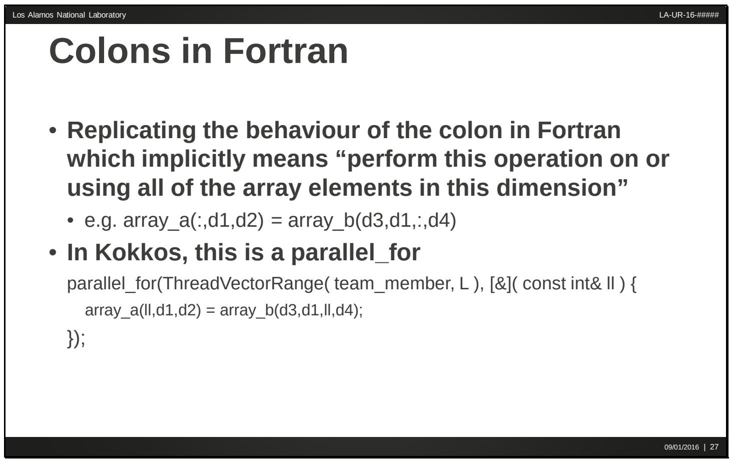
## Language Differences

- **Where possible, the counting differences between the languages can be hidden in a functor constructor**
  - Fortran: i is used to select an array index
    - e.g. array_a( i )
  - C++: in the constructor of the functor where i_ will be used, an expression such as i_(i-1) could be written to initialize i_ so that it points to the same array element as i would in Fortran, e.g .
      - int i_;
      - functor( const int i ) : i_( i – 1 ) {}
  - Then array_a(i) in Fortran and array_a(i_) in C++ will be the same element

09/01/2016 | 26

**We found one way to reduce the opportunity for “off by one” errors, and that was by subtracting one from the integer parameters in the C++ kernel in their construction (where relevant). For some integers, this was unnecessary, such as those used the same way one would a boolean variable. For others, used in counting expressions, dimensionality arguments, logical expressions, it was necessary to painstakingly count the combinations where these modified integers would be used together to ensure that both sides of the expression had one subtracted from their elements only once, and not multiple times, so that the expressions maintained the same articulation of their computational intent.**

Slide 27


Los Alamos National Laboratory LA-UR-16-#####

## Colons in Fortran

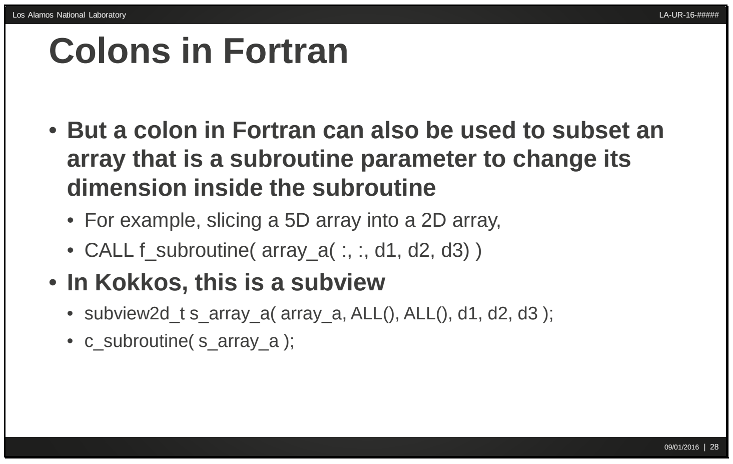
- **Replicating the behaviour of the colon in Fortran which implicitly means “perform this operation on or using all of the array elements in this dimension”**
  - e.g. array_a(:,d1,d2) = array_b(d3,d1,:,d4)
- **In Kokkos, this is a parallel_for**

```
parallel_for(ThreadVectorRange( team_member, L ), [&]( const int& ll ) {
    array_a(ll,d1,d2) = array_b(d3,d1,ll,d4);
});
```

09/01/2016 | 27

**Colons in Fortran mean “all of the array elements in this dimension”. This can be used in a handful of ways, that are transliterated distinctly. Above, we show how it can be used on both signs of an assignment operator, given that the dimensions referred to by the colons are compatible. To rewrite this type of usage in a parallel kernel, it is straightforward, simply adding an iterator index in place of the colon. It is important to note that ordering is not implicit in the parallel region.**

Slide 28


Los Alamos National Laboratory

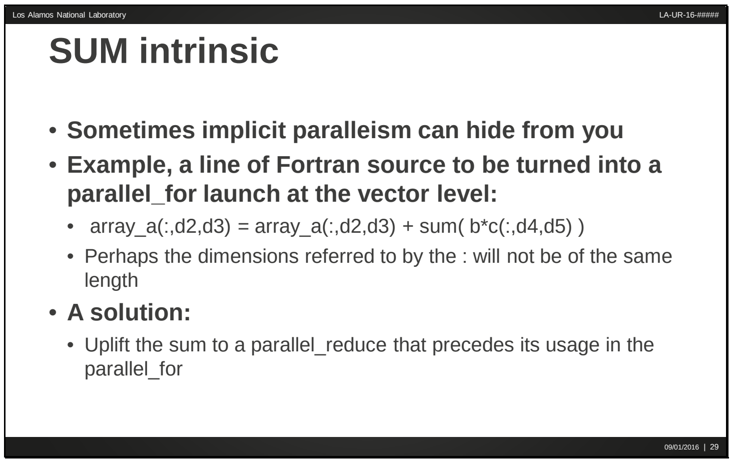
LA-UR-16-#####

## Colons in Fortran

- But a colon in Fortran can also be used to subset an array that is a subroutine parameter to change its dimension inside the subroutine
  - For example, slicing a 5D array into a 2D array,
  - CALL f_subroutine( array_a( :, :, d1, d2, d3) )
- In Kokkos, this is a subview
  - subview2d_t s_array_a( array_a, ALL(), ALL(), d1, d2, d3 );
  - c_subroutine( s_array_a );

09/01/2016 | 28

**Since the colon can be used to subset an array's dimension(s), it is idiomatic in Fortran to use this ability when reducing the dimensionality of an array between execution scopes. For example, a the driver of a routine might feature a 4D array, where one dimension is multiple time values for numerical methods requiring such, and the other three are spatial dimensions. When passing this 4D array to a physics routine, multiple time values might not be needed, and so the array is subsetted upon entry into the subroutine as above. In rewriting this behaviour in C++ from Fortran, we use a feature of Kokkos known as the subview. A subview is a full-class view, a reference counted pointer to memory, that can point to a subset of the view that is used to create the subview. Here we show the Kokkos syntax "ALL()" which replicates the functionality of the colon in Fortran (when it is used to reduce the dimensionality).**



## SUM intrinsic

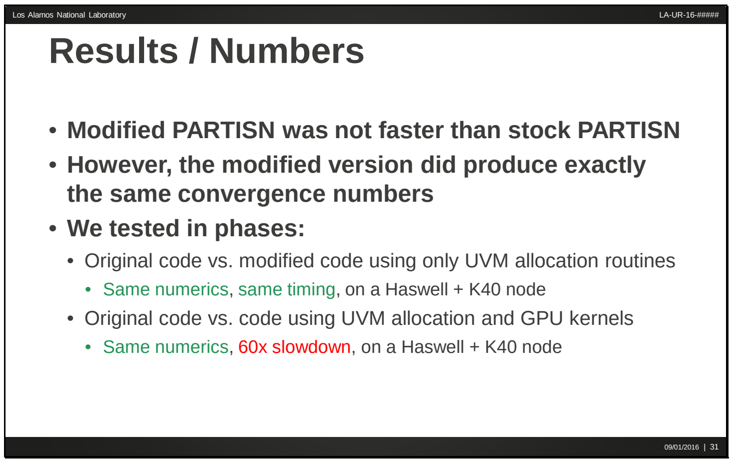
- **Sometimes implicit paralleism can hide from you**
- **Example, a line of Fortran source to be turned into a parallel_for launch at the vector level:**
  - array_a(:,d2,d3) = array_a(:,d2,d3) + sum( b*c(:,d4,d5) )
  - Perhaps the dimensions referred to by the : will not be of the same length
- **A solution:**
  - Uplift the sum to a parallel_reduce that precedes its usage in the parallel_for

**There is implicit parallelism in the Fortran SUM() intrinsic can be difficult if it is embedded at the parallelism level of vectors. SUM is vectorized already, so it must be promoted to outside the vectorized region, and performed ahead of time.**

Slide 30

## Lessons Learned

Slide 31

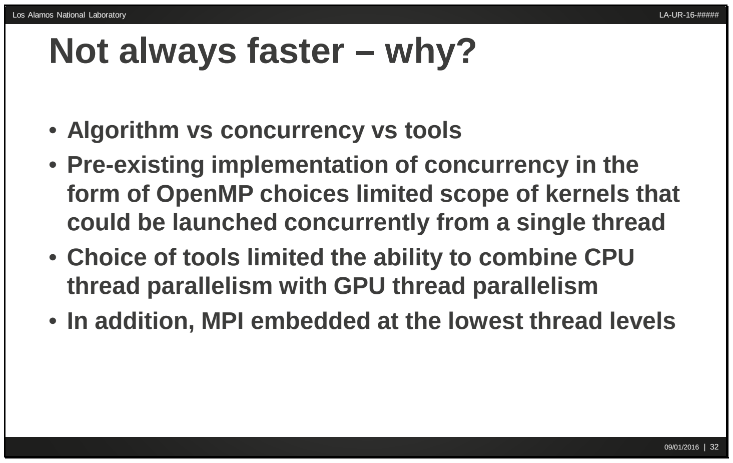


## Results / Numbers

- **Modified PARTISN was not faster than stock PARTISN**
- **However, the modified version did produce exactly the same convergence numbers**
- **We tested in phases:**
  - Original code vs. modified code using only UVM allocation routines
    - Same numerics, same timing, on a Haswell + K40 node
  - Original code vs. code using UVM allocation and GPU kernels
    - Same numerics, 60x slowdown, on a Haswell + K40 node

**We tested the modified PARTISN code and showed it provide compatible physical results with the original code. This was a great engineering success to us. It lets us know it is literally possible to run PARTISN in a GPU environment. We tested both the modified version (with C++ backed memory allocations) both with and without the GPU kernel enabled, and showed success in both cases. In our experiments, we found the modified version to be much slower than the original version, because of lack of threaded concurrency, and greatly reduced kernel launch sizing parameters.**

Slide 32


Los Alamos National Laboratory LA-UR-16-#####

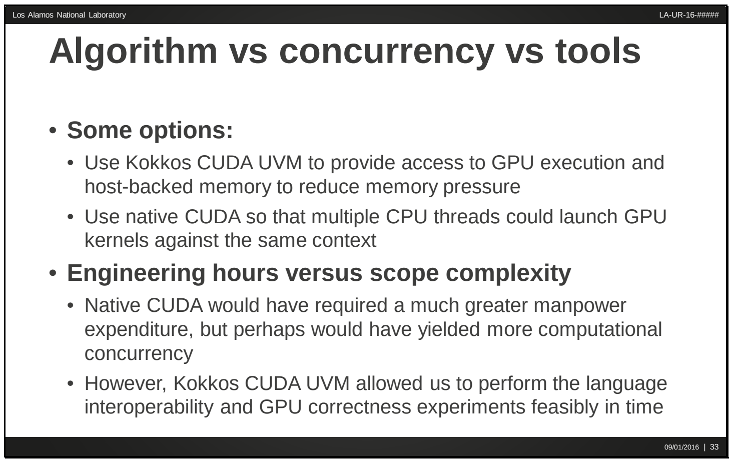
## Not always faster – why?

- Algorithm vs concurrency vs tools
- Pre-existing implementation of concurrency in the form of OpenMP choices limited scope of kernels that could be launched concurrently from a single thread
- Choice of tools limited the ability to combine CPU thread parallelism with GPU thread parallelism
- In addition, MPI embedded at the lowest thread levels

09/01/2016 | 32

**We would have needed a larger time and engineering allocation in order to increase the scope of our efforts to include the entirety of the threaded region in PARTISN. Had we done this, we would have expected similar or improved performance in the modified vice native code. This would have required in-development (but not ready at the time of work) features in Kokkos, or rewriting the Kokkos portion of the code in CUDA (which would obviate much of the development done over the last few years in learning it). Either one of these would have exposed more available concurrency. It is noted that both options would also have required embedding MPI in the kernel launch and also language interoperability with respect to MPI. The calls to MPI inside were mostly asynchronous, so it would likely have been possible, but it would have reduced speed without necessary multi-threaded or GPU MPI support. It should also be noted that GPU communication improvements are to be expected in future ASC hardware.**

Slide 33

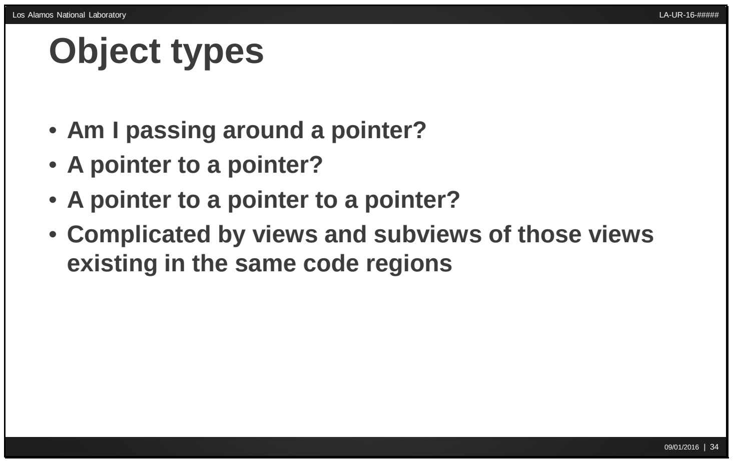


## Algorithm vs concurrency vs tools

- **Some options:**
  - Use Kokkos CUDA UVM to provide access to GPU execution and host-backed memory to reduce memory pressure
  - Use native CUDA so that multiple CPU threads could launch GPU kernels against the same context
- **Engineering hours versus scope complexity**
  - Native CUDA would have required a much greater manpower expenditure, but perhaps would have yielded more computational concurrency
  - However, Kokkos CUDA UVM allowed us to perform the language interoperability and GPU correctness experiments feasibly in time

**Described in detail on previous slide.**

Slide 34


Los Alamos National Laboratory

LA-UR-16-#####

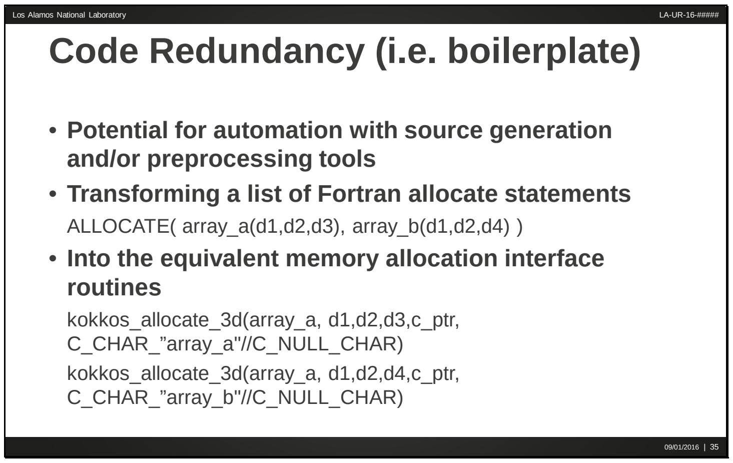
## Object types

- Am I passing around a pointer?
- A pointer to a pointer?
- A pointer to a pointer to a pointer?
- Complicated by views and subviews of those views existing in the same code regions

09/01/2016 | 34

Some care had to be taken to ensure the correct level of pointer or dereferencing was being used at various stages of interface between Fortran and C++. One great trick we learned was to test the pointer level of a Kokkos view by calling a member function. If the member function was linked correctly, then we knew we had the correct level of “pointerness” to use in a parallel_for construct. Subviews are pointers to views, and this technique helped with them especially. (A view is essentially, in some senses, a fancy reference counted pointer to a memory allocation, with additional state stored in its object.)

Slide 35



## Code Redundancy (i.e. boilerplate)

- **Potential for automation with source generation and/or preprocessing tools**
- **Transforming a list of Fortran allocate statements**

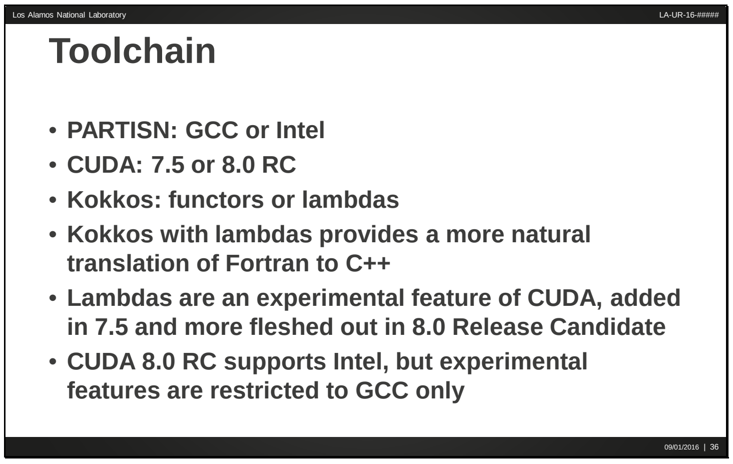
  ALLOCATE( array_a(d1,d2,d3), array_b(d1,d2,d4) )
- **Into the equivalent memory allocation interface routines**

  kokkos_allocate_3d(array_a, d1,d2,d3,c_ptr, C_CHAR_"array_a"//C_NULL_CHAR)

  kokkos_allocate_3d(array_a, d1,d2,d4,c_ptr, C_CHAR_"array_b"//C_NULL_CHAR)

**We felt many of the array allocations could have been created with automated tools. Scripts and or parsers could have been written to separate the above representative ALLOCATE statement into component allocation routines. While not necessary in our experiment, such would be invaluable to a production team.**

Slide 36


Los Alamos National Laboratory LA-UR-16-#####

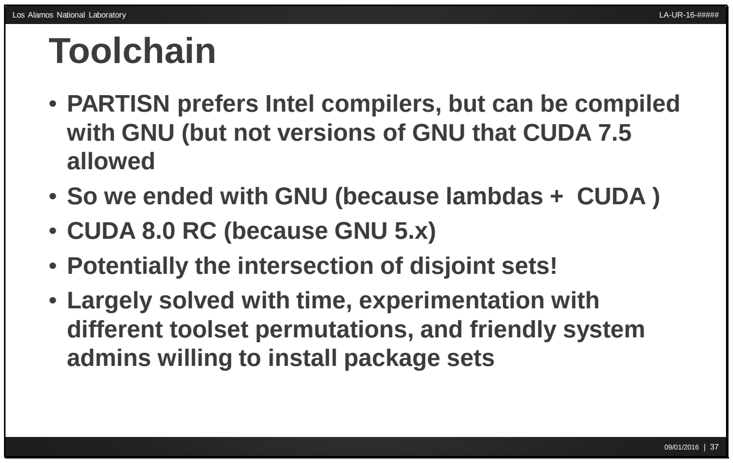
## Toolchain

- PARTISN: GCC or Intel
- CUDA: 7.5 or 8.0 RC
- Kokkos: functors or lambdas
- Kokkos with lambdas provides a more natural translation of Fortran to C++
- Lambdas are an experimental feature of CUDA, added in 7.5 and more fleshed out in 8.0 Release Candidate
- CUDA 8.0 RC supports Intel, but experimental features are restricted to GCC only

09/01/2016 | 36

**One of the trickier aspects of this work was in matching up production code team desires with experimental team desires, with the toolchain between the two groups. The production team prefers the Intel compiler, but was willing to revive GCC support for us. The experimental team (us) preferred rewriting kernel sections as lambdas, instead of functors, because we feel this preserves, to the greatest extent possible, the look and feel of the code. (This improves maintainability, if the work is picked up by the production team.)**

Slide 37


Los Alamos National Laboratory LA-UR-16-#####

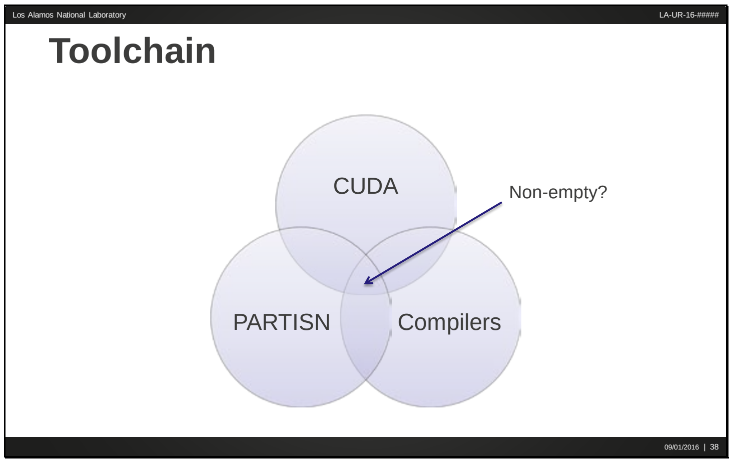
## Toolchain

- PARTISN prefers Intel compilers, but can be compiled with GNU (but not versions of GNU that CUDA 7.5 allowed
- So we ended with GNU (because lambdas + CUDA )
- CUDA 8.0 RC (because GNU 5.x)
- Potentially the intersection of disjoint sets!
- Largely solved with time, experimentation with different toolset permutations, and friendly system admins willing to install package sets

09/01/2016 | 37

**Lambdas in CUDA are an experimental feature in CUDA 7.5, and still experimental in CUDA 8.0 Release Candidate (current as of time of work), added at the behest of the SNL Kokkos developers by NVIDIA. In 7.5, nvcc (the CUDA compiler) only worked with GNU compilers (and only less than major version 5), while in 8.0 it works with Intel (and others) as well. However, experimental features (which we needed for lambdas) were only available in GCC. In addition, we required nested lambdas (to write the kernel in the most future-proof fashion, were we able to expose greater concurrency in the kernel and surrounding regions of PARTISN code), and this mandated the 8.0 RC of CUDA**

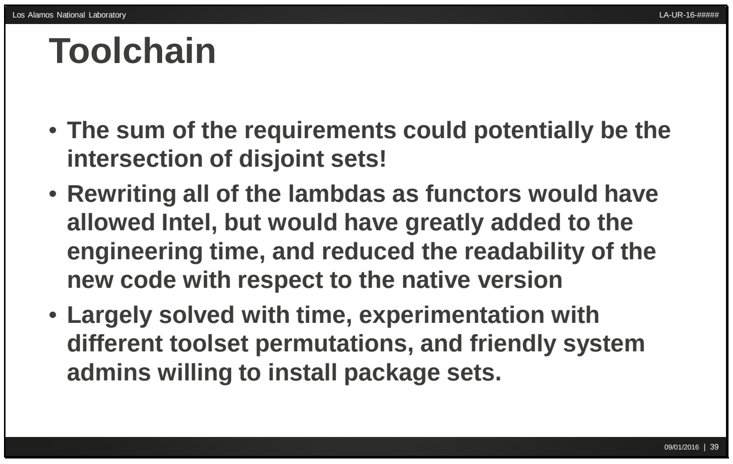
Slide 38



## Toolchain

**We were pleased to find a minimum of one combination of toolchain selections which supported our endeavours. We have every reason to believe that the number of choices will expand with time, as NVIDIA has shown willingness to increase compiler support in nvcc from 7.5 to 8.0.**

Slide 39


Los Alamos National Laboratory LA-UR-16-#####

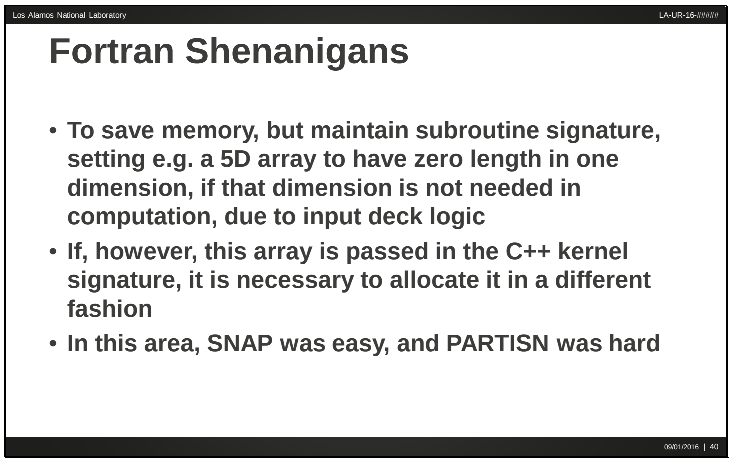
## Toolchain

- The sum of the requirements could potentially be the intersection of disjoint sets!
- Rewriting all of the lambdas as functors would have allowed Intel, but would have greatly added to the engineering time, and reduced the readability of the new code with respect to the native version
- Largely solved with time, experimentation with different toolset permutations, and friendly system admins willing to install package sets.

09/01/2016 | 39

**We were pleased to find a minimum of one combination of toolchain selections which supported our endeavours. We have every reason to believe that the number of choices will expand with time, as NVIDIA has shown willingness to increase compiler support in nvcc from 7.5 to 8.0.**

Slide 40


Los Alamos National Laboratory LA-UR-16-#####

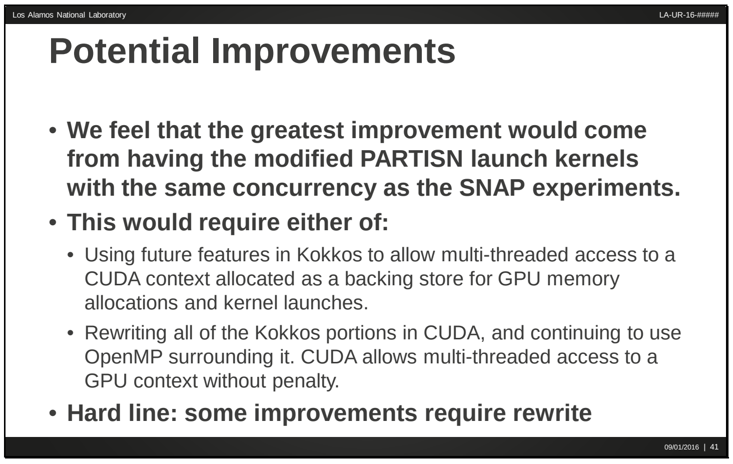
## Fortran Shenanigans

- To save memory, but maintain subroutine signature, setting e.g. a 5D array to have zero length in one dimension, if that dimension is not needed in computation, due to input deck logic
- If, however, this array is passed in the C++ kernel signature, it is necessary to allocate it in a different fashion
- In this area, SNAP was easy, and PARTISN was hard

09/01/2016 | 40

**In supporting many different physics solvers and modes, PARTISN has a fairly complex set of memory allocation sites. There are places where an array, if not needed by a particular physics mode, is allocated with zero length dimensions. This preserves array dimensionality, for subroutine arguments, but reduces memory allocation demands for efficiency. This is perfectly correct idiomatic Fortran, but it presents difficulty in transliterating because the same idiom is not present in C++ (the least reasons of which is that C++ does not have multi-dimensional array as a first-class data type). To work around this, we identified areas where this mattered for our work (certainly not exhaustive of all areas) and modified them to produce arrays with length one in dimensions instead of zero. Then, as in the native PARTISN code, we ignored those shortened arrays**

Slide 41


Los Alamos National Laboratory LA-UR-16-#####

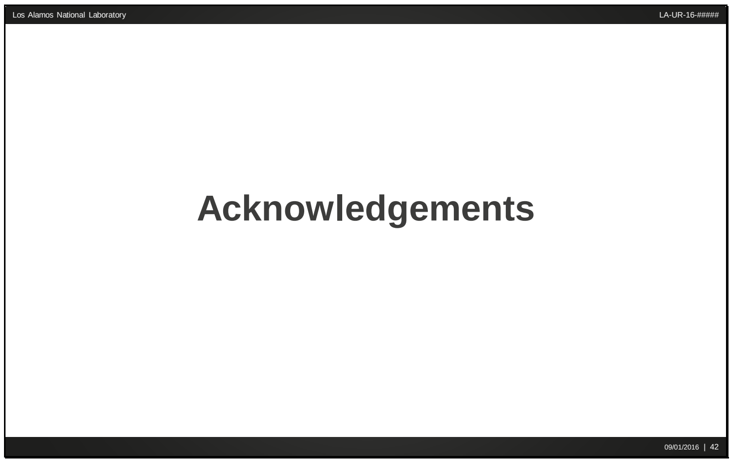
## Potential Improvements

- **We feel that the greatest improvement would come from having the modified PARTISN launch kernels with the same concurrency as the SNAP experiments.**
- **This would require either of:**
  - Using future features in Kokkos to allow multi-threaded access to a CUDA context allocated as a backing store for GPU memory allocations and kernel launches.
  - Rewriting all of the Kokkos portions in CUDA, and continuing to use OpenMP surrounding it. CUDA allows multi-threaded access to a GPU context without penalty.
- **Hard line: some improvements require rewrite**

09/01/2016 | 41

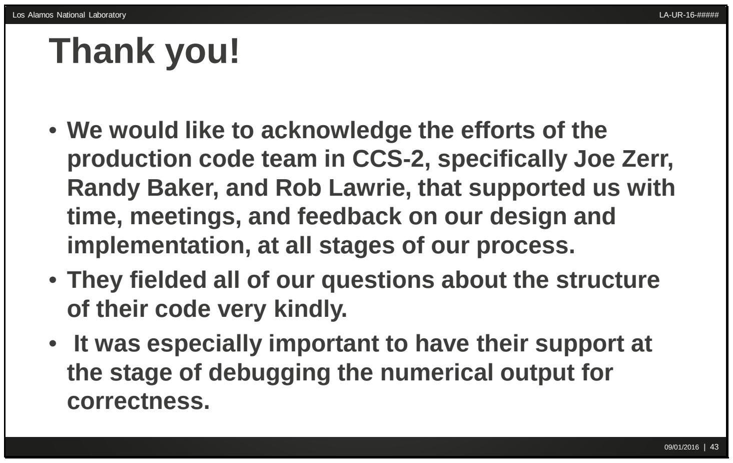
**As described elsewhere and above, we feel we could achieve similar results in PARTISN, given time and effort. In the scope of the L2, though, we were greatly pleased to show a) that our interoperability experiments were feasible b) that the modified code produced a correct result, and finally c) how to continue down the road for performance. We consider all of this to be useful feedback and or a roadmap for the production team, if they are so interested in continuing with this work.**

Slide 42



## Acknowledgements

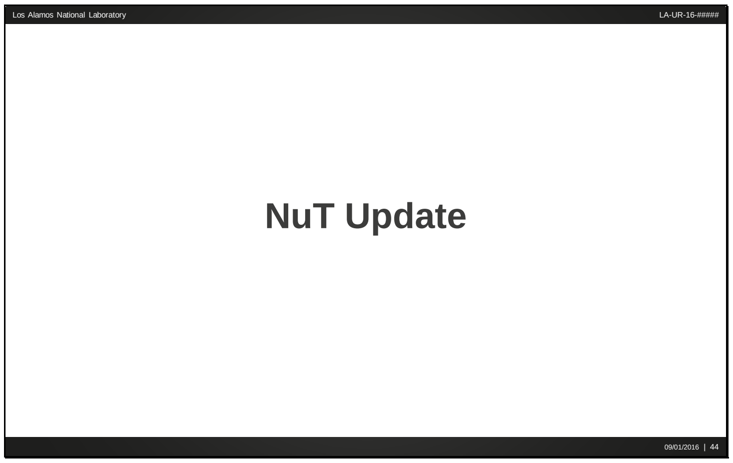
Slide 43



## Thank you!

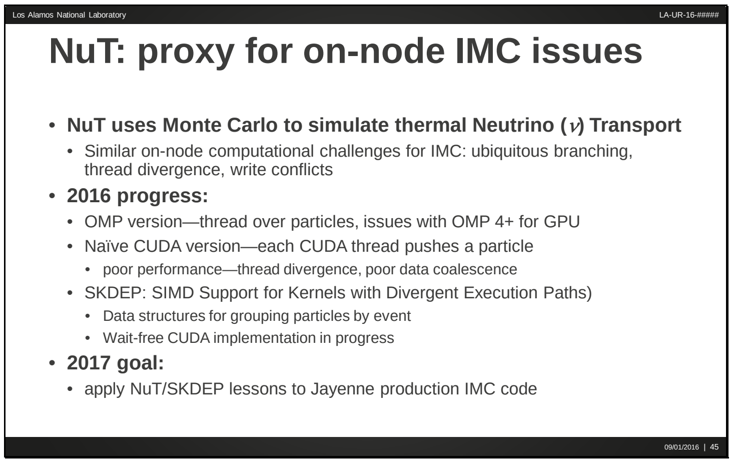
- We would like to acknowledge the efforts of the production code team in CCS-2, specifically Joe Zerr, Randy Baker, and Rob Lawrie, that supported us with time, meetings, and feedback on our design and implementation, at all stages of our process.
- They fielded all of our questions about the structure of their code very kindly.
- It was especially important to have their support at the stage of debugging the numerical output for correctness.

Slide 44



## NuT Update

Slide 45



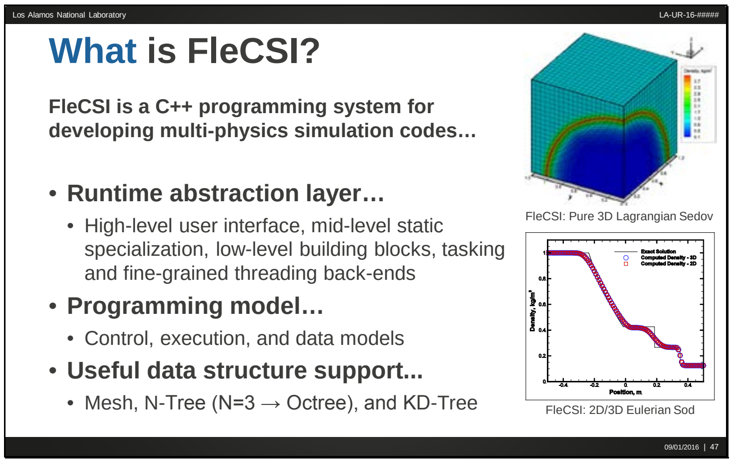
## NuT: proxy for on-node IMC issues

- **NuT uses Monte Carlo to simulate thermal Neutrino ($\nu$) Transport**
  - Similar on-node computational challenges for IMC: ubiquitous branching, thread divergence, write conflicts
- **2016 progress:**
  - OMP version—thread over particles, issues with OMP 4+ for GPU
  - Naïve CUDA version—each CUDA thread pushes a particle
    - poor performance—thread divergence, poor data coalescence
  - SKDEP: SIMD Support for Kernels with Divergent Execution Paths)
    - Data structures for grouping particles by event
    - Wait-free CUDA implementation in progress
- **2017 goal:**
  - apply NuT/SKDEP lessons to Jayenne production IMC code

Slide 46

## FleCSI Update

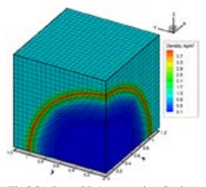
Slide 47

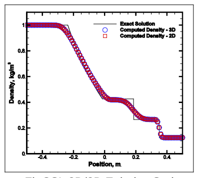

Los Alamos National Laboratory LA-UR-16-#####

## What is FleCSI?

**FleCSI is a C++ programming system for developing multi-physics simulation codes…**

- **Runtime abstraction layer…**
  - High-level user interface, mid-level static specialization, low-level building blocks, tasking and fine-grained threading back-ends
- **Programming model…**
  - Control, execution, and data models
- **Useful data structure support...**
  - Mesh, N-Tree (N=3 → Octree), and KD-Tree


FleCSI: Pure 3D Lagrangian Sedov



FleCSI: 2D/3D Eulerian Sod

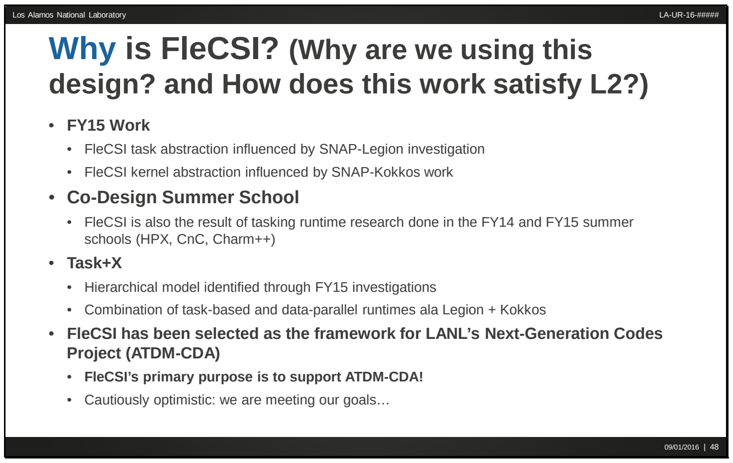
09/01/2016 | 47

**FleCSI, the Flexible Computational Science Infrastructure (FleCSI) is a C++ framework to aid in the development of application interfaces and tools for creating and maintaining multi-physics simulation codes. The primary structure of FleCSI is hierarchical, exposing low-level, mid-level, and high-level interfaces that are appropriate for different sets of users. The normal use pattern for FleCSI is that a computer or computational scientist creates a specialization layer for an application using the low-level FleCSI interface. This mid-level layer provides the high-level interface that the end user actually uses to develop their physics simulation.**

**FleCSI provides control, execution, and data models that are consistent with modern task-based and functional models.**

**FleCSI also provides support for several important data structures and algorithms that can be statically customized as part of the creation of the mid-level interface. Examples of these include mesh and tree topology types.**

Slide 48


Los Alamos National Laboratory LA-UR-16-#####

## Why is FleCSI? (Why are we using this design? and How does this work satisfy L2?)

- **FY15 Work**
  - FleCSI task abstraction influenced by SNAP-Legion investigation
  - FleCSI kernel abstraction influenced by SNAP-Kokkos work
- **Co-Design Summer School**
  - FleCSI is also the result of tasking runtime research done in the FY14 and FY15 summer schools (HPX, CnC, Charm++)
- **Task+X**
  - Hierarchical model identified through FY15 investigations
  - Combination of task-based and data-parallel runtimes ala Legion + Kokkos
- **FleCSI has been selected as the framework for LANL's Next-Generation Codes Project (ATDM-CDA)**
  - **FleCSI's primary purpose is to support ATDM-CDA!**
  - Cautiously optimistic: we are meeting our goals…

09/01/2016 | 48

**The design of the FleCSI framework was directly motivated by experiments and lessons-learned from previous co-design milestones. In particular, work on the SNAP and Pennant proxy applications established the viability of the Legion and Kokkos programming models for handling distributed-memory parallelism and data-parallelism, respectively.**

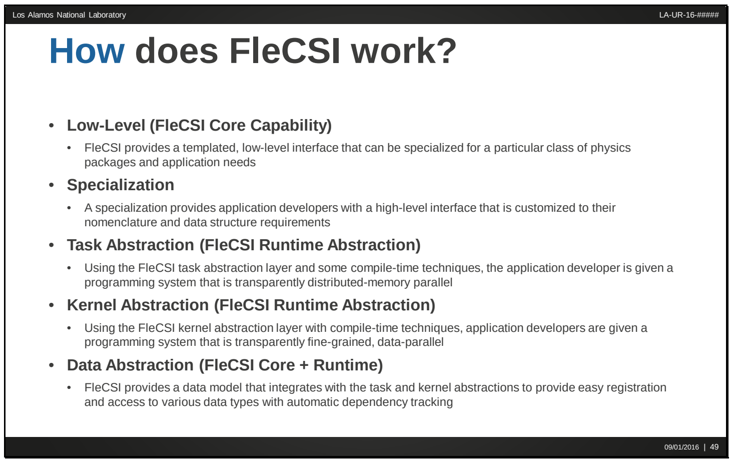
**The hierarchical runtime approach developed through our FY15 investigations make up the foundation of the FleCSI task-based programming model. Technical lessons from these early experiments are also being used to improve the Legion interface and capabilities (contract with Nvidia and collaboration and support of Stanford), and to make it compatible with various node-level runtimes. LANL's support of Nvidia and Stanford is designed to harden Legion for production use, and to address specific design and performance issues identified during the FY15 milestone work. Particular areas identified for improvement and enhancements include, parallel scalability, an improved mapping interface, and support for new processor types (e.g., Kokkos processor type: work performed at LANL in collaboration with Stanford).**

**Our work on understanding and using fine-grained data-parallel programming models (through work on SNAP with Kokkos) is**

**critical to the design of FleCSI's kernel interface. Participation by the Kokkos team (and several LANL contibutors) on the C++17 standards committee has helped to push changes to the C++ standard library that will dramatically improve our ability to achieve portable performance across the diverse architecture landscape of the future.**

**FleCSI is directly supporting production code development through LANL's ATDM project (currently called 'NGC' for Next-Generation Codes). Results from work on the co-design FY16 milestone give us restrained confidence that we can realize many or all of the primary goals of the FleCSI project: separation of concerns, performance portability, code sustainability, scalability and resilience. Our investigations into modern programming models were fundamental to the design and development of the FleCSI framework.**

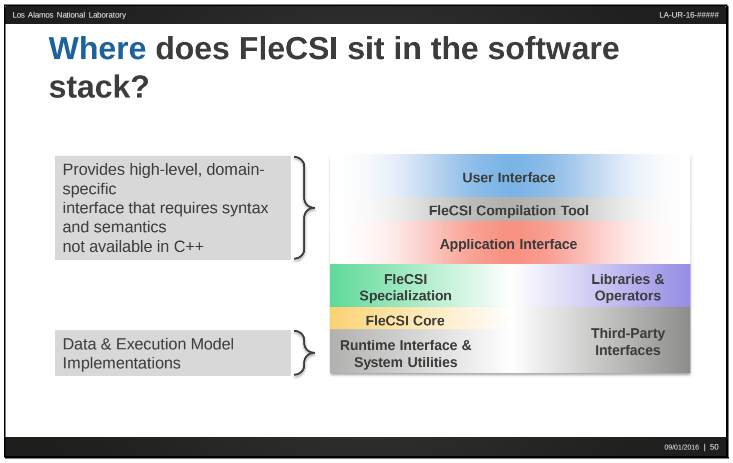
Slide 49



## How does FleCSI work?

- **Low-Level (FleCSI Core Capability)**
  - FleCSI provides a templated, low-level interface that can be specialized for a particular class of physics packages and application needs
- **Specialization**
  - A specialization provides application developers with a high-level interface that is customized to their nomenclature and data structure requirements
- **Task Abstraction (FleCSI Runtime Abstraction)**
  - Using the FleCSI task abstraction layer and some compile-time techniques, the application developer is given a programming system that is transparently distributed-memory parallel
- **Kernel Abstraction (FleCSI Runtime Abstraction)**
  - Using the FleCSI kernel abstraction layer with compile-time techniques, application developers are given a programming system that is transparently fine-grained, data-parallel
- **Data Abstraction (FleCSI Core + Runtime)**
  - FleCSI provides a data model that integrates with the task and kernel abstractions to provide easy registration and access to various data types with automatic dependency tracking

Slide 50



## Where does FleCSI sit in the software stack?

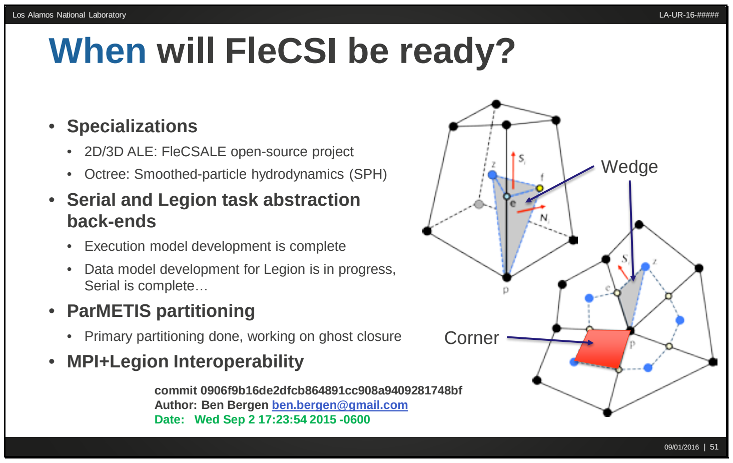
Provides high-level, domain-specific
interface that requires syntax and semantics
not available in C++

Data & Execution Model Implementations

**This slide gives a visual depiction of where the core FleCSI library sits in the software stack. Notice that FleCSI itself may directly interface other third-party interfaces. There is a well-defined interface to low-level runtime drivers and system utilities. A FleCSI specialization will likely interface libraries that are either third-party or that have been written with the FleCSI programming model.**

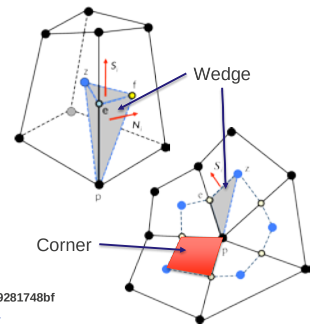
**Application developers use the FleCSI specialization layer to generate an application interface. We are looking at ways to include static optimization between the user interface and the application layer. An example of this approach would be to generate source for the application layer based on the user inputs. This would allow better performance tuning by exposing more information to the compiler.**

Slide 51



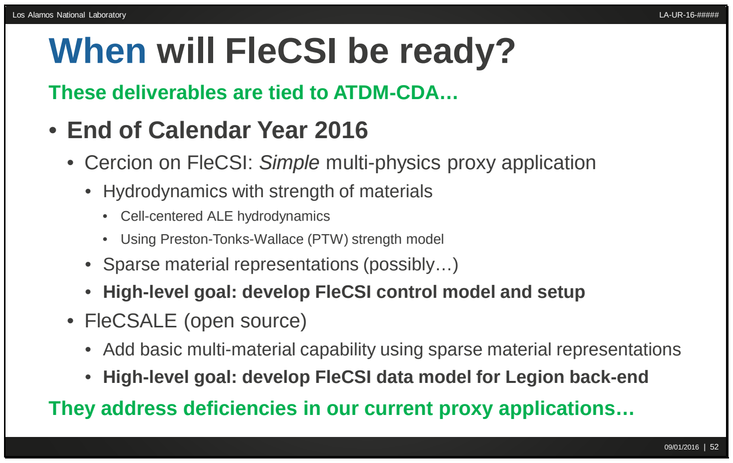
## When will FleCSI be ready?

- **Specializations**
  - 2D/3D ALE: FleCSALE open-source project
  - Octree: Smoothed-particle hydrodynamics (SPH)
- **Serial and Legion task abstraction back-ends**
  - Execution model development is complete
  - Data model development for Legion is in progress, Serial is complete…
- **ParMETIS partitioning**
  - Primary partitioning done, working on ghost closure
- **MPI+Legion Interoperability**



**commit 0906f9b16de2dfcb864891cc908a9409281748bf**
**Author: Ben Bergen ben.bergen@gmail.com**
**Date:   Wed Sep 2 17:23:54 2015 -0600**

Slide 52



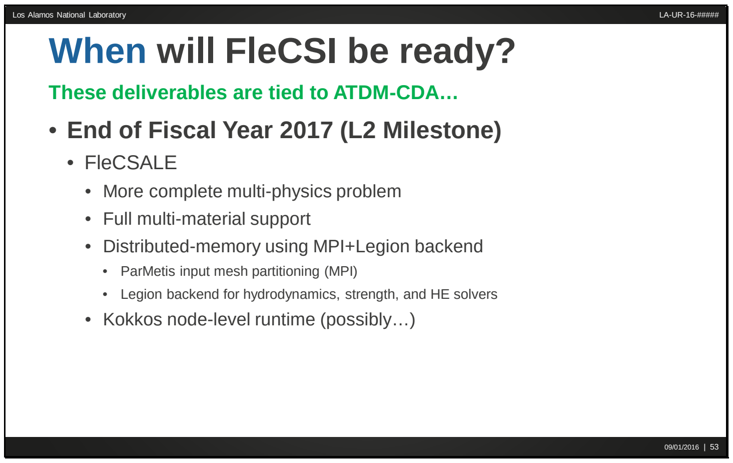
## When will FleCSI be ready?

These deliverables are tied to ATDM-CDA…

- **End of Calendar Year 2016**
  - Cercion on FleCSI: *Simple* multi-physics proxy application
    - Hydrodynamics with strength of materials
      - Cell-centered ALE hydrodynamics
      - Using Preston-Tonks-Wallace (PTW) strength model
    - Sparse material representations (possibly…)
    - **High-level goal: develop FleCSI control model and setup**
  - FleCSALE (open source)
    - Add basic multi-material capability using sparse material representations
    - **High-level goal: develop FleCSI data model for Legion back-end**

They address deficiencies in our current proxy applications…

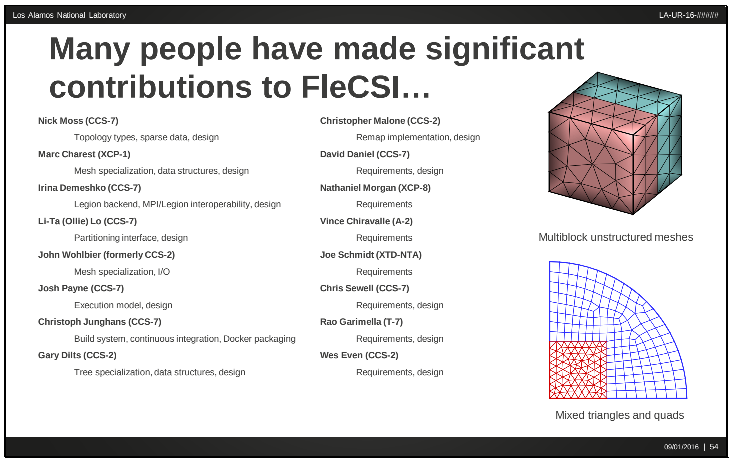
Slide 53



## When will FleCSI be ready?

**These deliverables are tied to ATDM-CDA…**

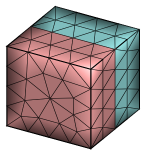
- **End of Fiscal Year 2017 (L2 Milestone)**
  - FleCSALE
    - More complete multi-physics problem
    - Full multi-material support
    - Distributed-memory using MPI+Legion backend
      - ParMetis input mesh partitioning (MPI)
      - Legion backend for hydrodynamics, strength, and HE solvers
    - Kokkos node-level runtime (possibly…)



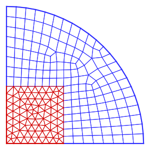
## Many people have made significant contributions to FleCSI…

**Nick Moss (CCS-7)**
Topology types, sparse data, design
**Marc Charest (XCP-1)**
Mesh specialization, data structures, design
**Irina Demeshko (CCS-7)**
Legion backend, MPI/Legion interoperability, design
**Li-Ta (Ollie) Lo (CCS-7)**
Partitioning interface, design
**John Wohlbier (formerly CCS-2)**
Mesh specialization, I/O
**Josh Payne (CCS-7)**
Execution model, design
**Christoph Junghans (CCS-7)**
Build system, continuous integration, Docker packaging
**Gary Dilts (CCS-2)**
Tree specialization, data structures, design

**Christopher Malone (CCS-2)**
Remap implementation, design
**David Daniel (CCS-7)**
Requirements, design
**Nathaniel Morgan (XCP-8)**
Requirements
**Vince Chiravalle (A-2)**
Requirements
**Joe Schmidt (XTD-NTA)**
Requirements
**Chris Sewell (CCS-7)**
Requirements, design
**Rao Garimella (T-7)**
Requirements, design
**Wes Even (CCS-2)**
Requirements, design



Multiblock unstructured meshes



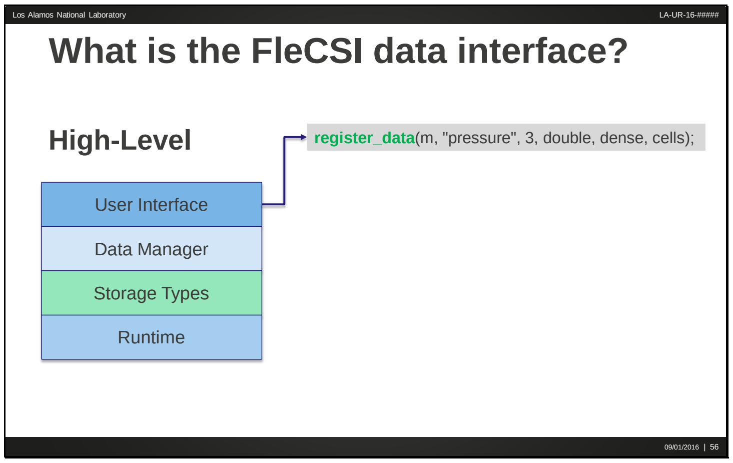
Mixed triangles and quads

**FleCSI was developed as part of the ATDM ASD project. It represents the collaborative efforts of many people across many disciplines.**

Slide 55

## FleCSI Data Model

Slide 56



## What is the FleCSI data interface?

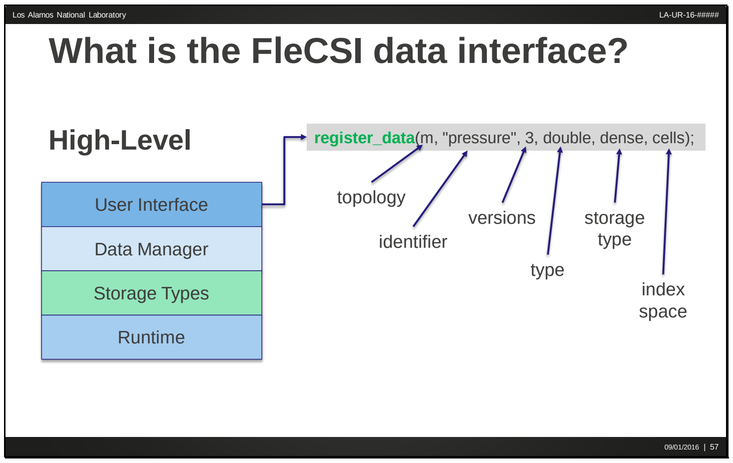
High-Level

User Interface
Data Manager
Storage Types
Runtime

```
register_data(m, "pressure", 3, double, dense, cells);
```

The FleCSI data model provides a high-level interface that can be used to register and access data that are associated with a data_client_t and an index space. The user interface does not expose any metaprogramming or templates, and is intended to allow very clean looking implementations of the physics methods being described.

Slide 57



Los Alamos National Laboratory LA-UR-16-#####

## What is the FleCSI data interface?

### High-Level

```
register_data(m, "pressure", 3, double, dense, cells);
```

09/01/2016 | 57

**The elements of the high-level interface are used by lower leves of FleCSI to select how the data should be registered.**

**topology: the data_client_t instance with which the registered data will be associated.**

**identifier: a string identifier that will be hashed to create a unique size_t id.**

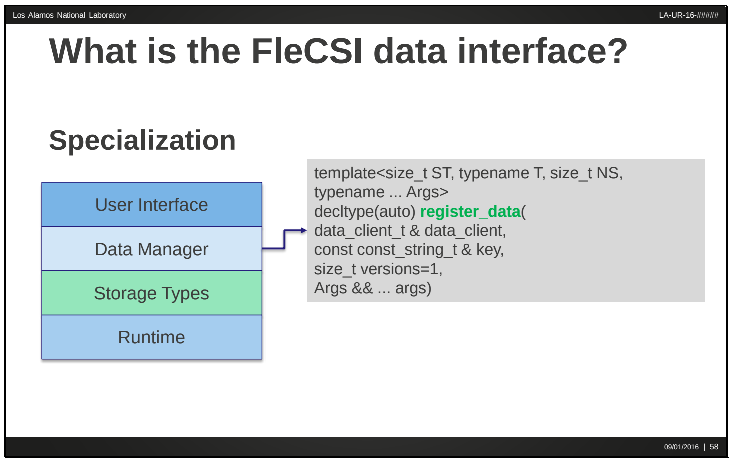
**versions: FleCSI supports multiple state versions under a single identifier. This is useful for new and old state, or for predictor-corrector methods.**

**type: the intrinsic or user-defined type of the data to be registered.**

**storage type: a hint to the framework that tells how the user intends to access the data.**

**index space: an index space that is either defined by the user, or that is defined by the framework itself, e.g., a mesh topology.**

Slide 58



## What is the FleCSI data interface?

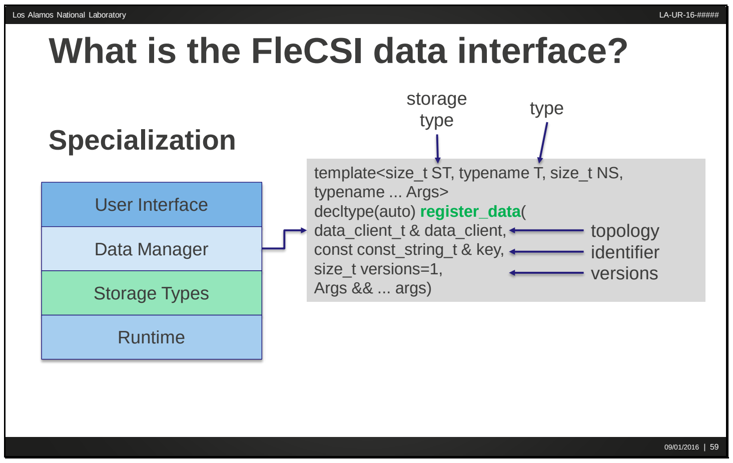
### Specialization

User Interface
Data Manager
Storage Types
Runtime

```
template<size_t ST, typename T, size_t NS,
typename ... Args>
decltype(auto) register_data(
data_client_t & data_client,
const const_string_t & key,
size_t versions=1,
Args && ... args)
```

**The specialization layer may add new backend support and storage types that modify what the default implementation does to register data. If this level of the framework is not specialized, the data registration falls through to a particular backend that determines how each storage type should be handled.**

Slide 59


Los Alamos National Laboratory LA-UR-16-#####

## What is the FleCSI data interface?

### Specialization

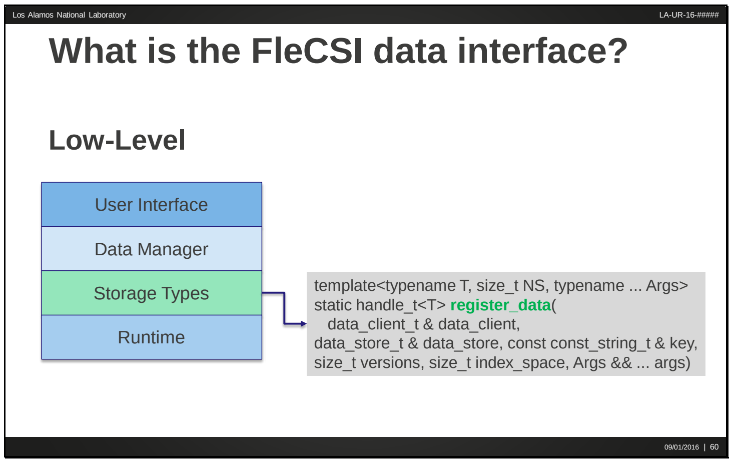
User Interface
Data Manager
Storage Types
Runtime
storage type
type
topology
identifier
versions

```
template<size_t ST, typename T, size_t NS,
typename ... Args>
decltype(auto) register_data(
data_client_t & data_client,
const const_string_t & key,
size_t versions=1,
Args && ... args)
```

09/01/2016 | 59

**This level of the data model uses some of the high-level inputs to apply static specialization to the low-level types. In this case, storage type and type are passed as template parameters to select specific low-level implementations.**

Slide 60


Los Alamos National Laboratory

LA-UR-16-#####

## What is the FleCSI data interface?

### Low-Level

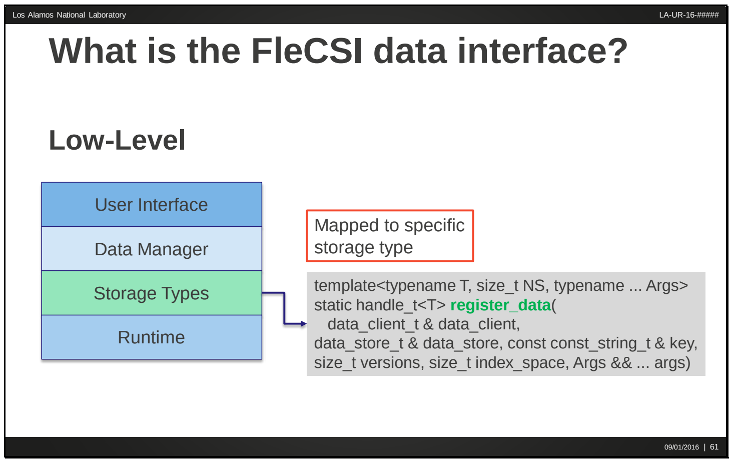
User Interface
Data Manager
Storage Types
Runtime

```
template<typename T, size_t NS, typename ... Args>
static handle_t<T> register_data(
   data_client_t & data_client,
data_store_t & data_store, const const_string_t & key,
size_t versions, size_t index_space, Args && ... args)
```

09/01/2016 | 60

**The low-level interface, having been specialized on type and storage type, makes use of the backend interface to actually register the data.**

Slide 61



## What is the FleCSI data interface?

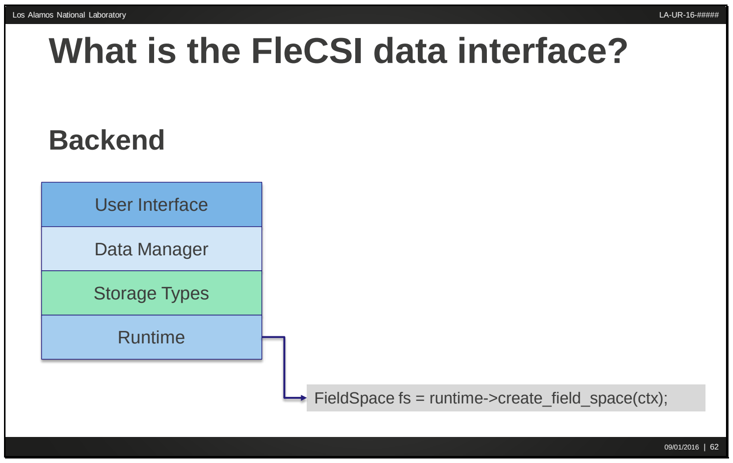
### Low-Level

User Interface
Data Manager
Storage Types
Runtime
Mapped to specific storage type

```
template<typename T, size_t NS, typename ... Args>
static handle_t<T> register_data(
   data_client_t & data_client,
data_store_t & data_store, const const_string_t & key,
size_t versions, size_t index_space, Args && ... args)
```

**This concept is covered in the previous two slides annotations.**

Slide 62


Los Alamos National Laboratory LA-UR-16-#####

## What is the FleCSI data interface?

### Backend

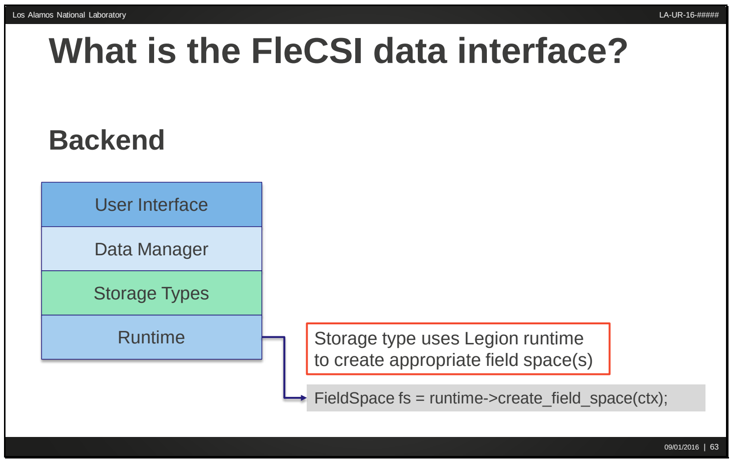
User Interface
Data Manager
Storage Types
Runtime

```
FieldSpace fs = runtime->create_field_space(ctx);
```

09/01/2016 | 62

**The particular runtime backend is selected by a data policy. At this level, FleCSI uses the specific low-level runtime interface to register the data.**

Slide 63



## What is the FleCSI data interface?

### Backend

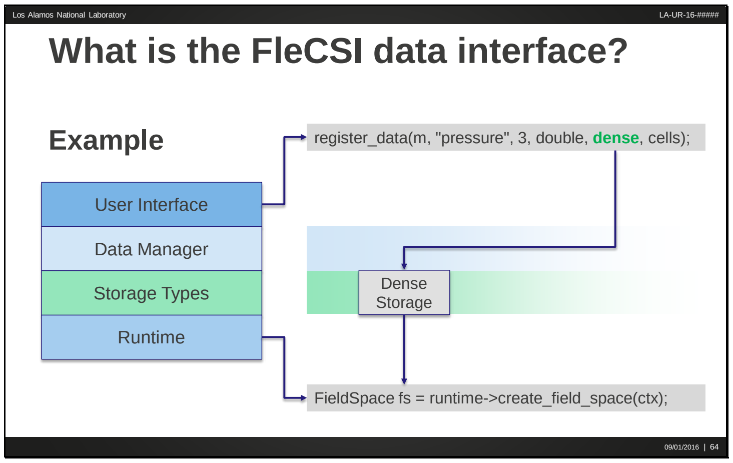
User Interface
Data Manager
Storage Types
Runtime

Storage type uses Legion runtime to create appropriate field space(s)

```
FieldSpace fs = runtime->create_field_space(ctx);
```

**This example shows the Legion backend. Data registration translates into the creation of a field space. In the following slides, we will show more details about how the data model and execution models are tied together.**

Slide 64



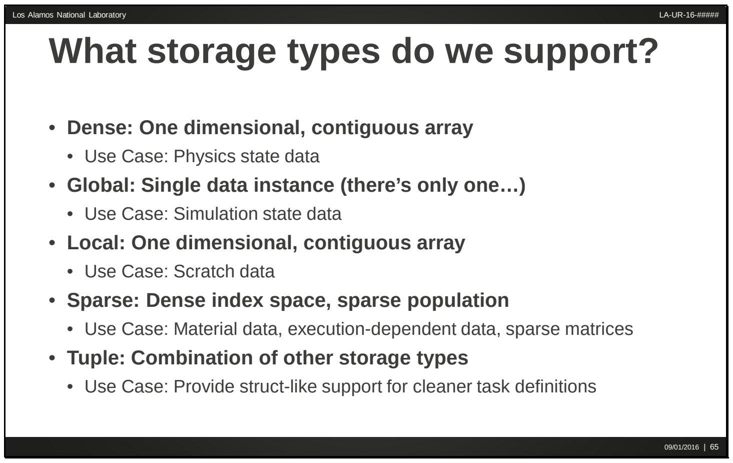
## What is the FleCSI data interface?

### Example

```
register_data(m, "pressure", 3, double, dense, cells);
FieldSpace fs = runtime->create_field_space(ctx);
```

**From top to bottom, the selection of a storage type, type, and data policy determine how data are registered for a particular combination of attributes.**

Slide 65


Los Alamos National Laboratory LA-UR-16-#####

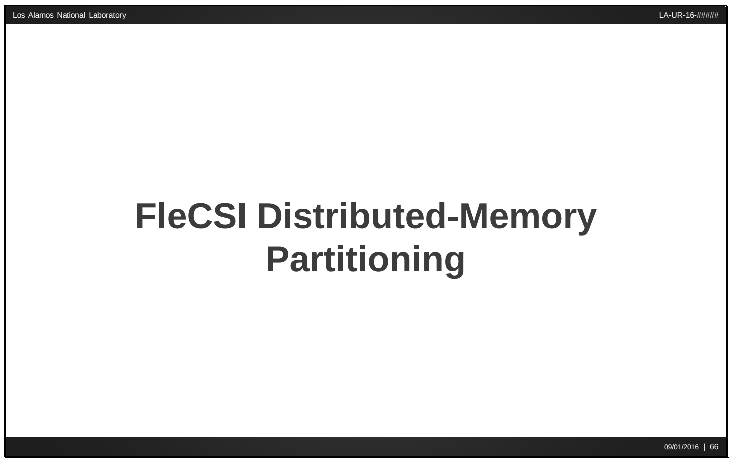
## What storage types do we support?

- **Dense: One dimensional, contiguous array**
  - Use Case: Physics state data
- **Global: Single data instance (there's only one…)**
  - Use Case: Simulation state data
- **Local: One dimensional, contiguous array**
  - Use Case: Scratch data
- **Sparse: Dense index space, sparse population**
  - Use Case: Material data, execution-dependent data, sparse matrices
- **Tuple: Combination of other storage types**
  - Use Case: Provide struct-like support for cleaner task definitions

09/01/2016 | 65

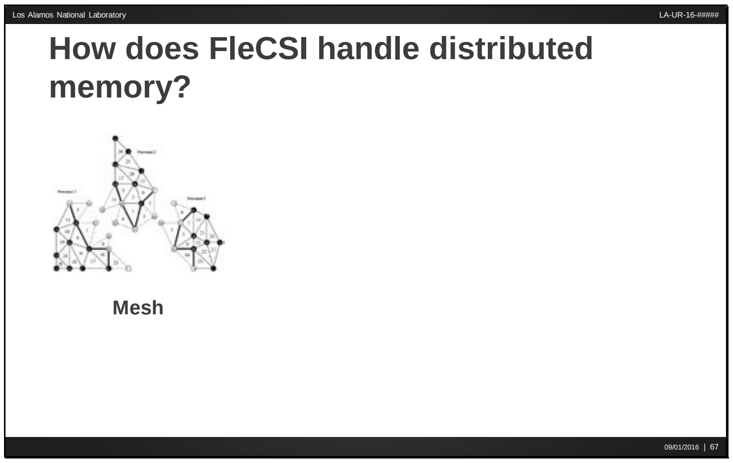
**This slide simply spells out the various storage types that we currently support. Additional types may be added. As stated earlier, these storage types are used by FleCSI to determine how data should be stored and accessible. The actual data may be of any type that satisfies certain constraints, primarily that the data can be serialized, and that they do not directly reference addresses in a particular address space.**

Slide 66



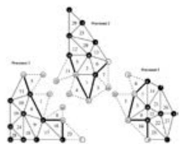
## FleCSI Distributed-Memory Partitioning

Slide 67



Los Alamos National Laboratory LA-UR-16-#####

## How does FleCSI handle distributed memory?

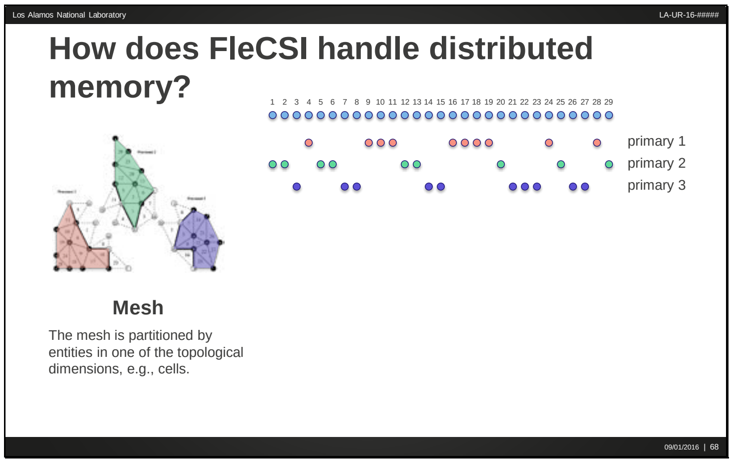


Mesh

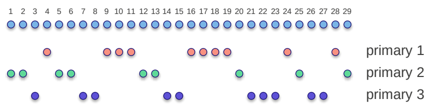
09/01/2016 | 67

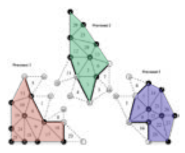
FleCSI uses conventional techniques to generate initial partitionings of mesh and tree entities. Selecting a particular entity type (in this case the mesh cells), FleCSI generates a primary partition of the cells into disjoint collections using ParMetis.

Slide 68



## How does FleCSI handle distributed memory?



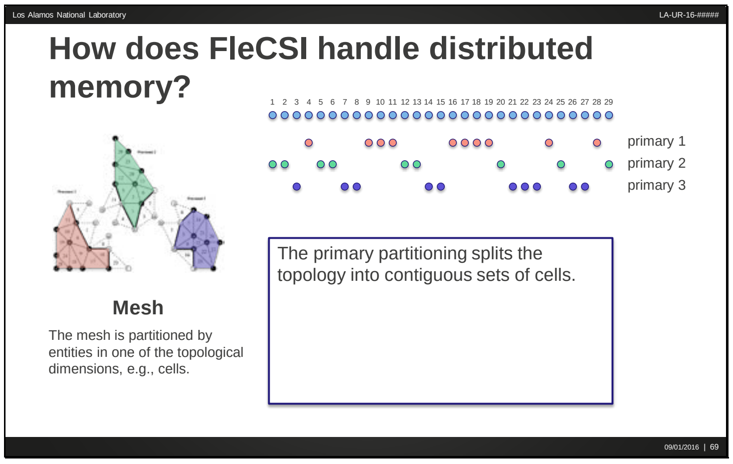


**Mesh**

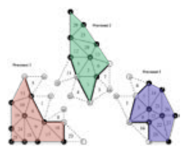
The mesh is partitioned by entities in one of the topological dimensions, e.g., cells.

**FleCSI uses conventional techniques to generate initial partitionings of mesh and tree entities. Selecting a particular entity type (in this case the mesh cells), FleCSI generates a primary partition of the cells into disjoint collections using ParMetis.**

Slide 69

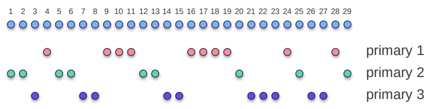

Los Alamos National Laboratory LA-UR-16-#####

## How does FleCSI handle distributed memory?



**Mesh**

The mesh is partitioned by entities in one of the topological dimensions, e.g., cells.

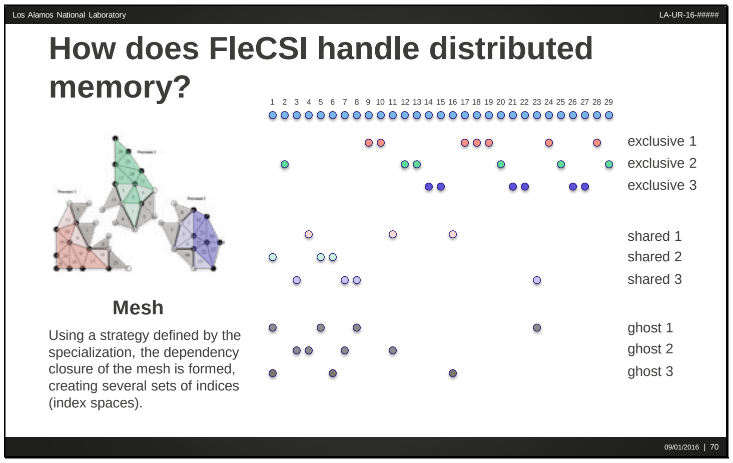


The primary partitioning splits the topology into contiguous sets of cells.

09/01/2016 | 69

**FleCSI uses conventional techniques to generate initial partitionings of mesh and tree entities. Selecting a particular entity type (in this case the mesh cells), FleCSI generates a primary partition of the cells into disjoint collections using ParMetis.**

Slide 70



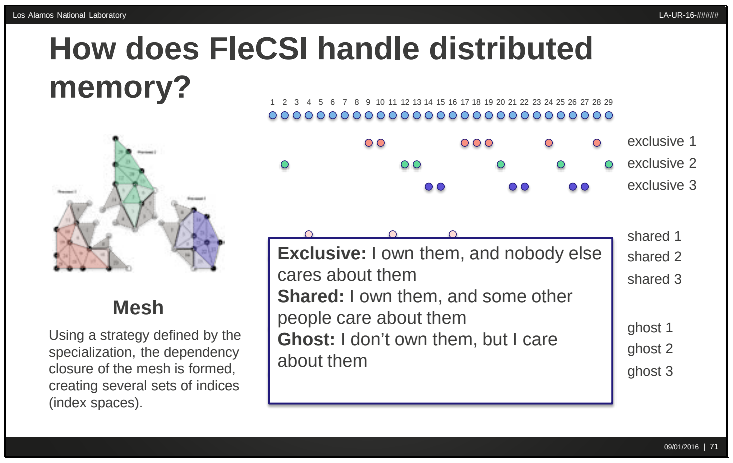
## How does FleCSI handle distributed memory?

**Mesh**

Using a strategy defined by the specialization, the dependency closure of the mesh is formed, creating several sets of indices (index spaces).

A FleCSI dependency closure creates several index spaces on each rank that provide a complete set of dependency information for that rank. The index spaces (local, shared, and ghost) contain topological indices that correspond to owned data, owned and shared data, and dependent data, respectively. The mesh partitioning image on the left shows each rank with its respective index spaces shaded to indicate local (dark), shared (light), and ghost (gray) indices. The logic used to define the dependency closure is part of the particular mesh specialization being used. The low-level FleCSI topology types support storage and manipulation of several dependency closures per specialization.

Slide 71



## How does FleCSI handle distributed memory?

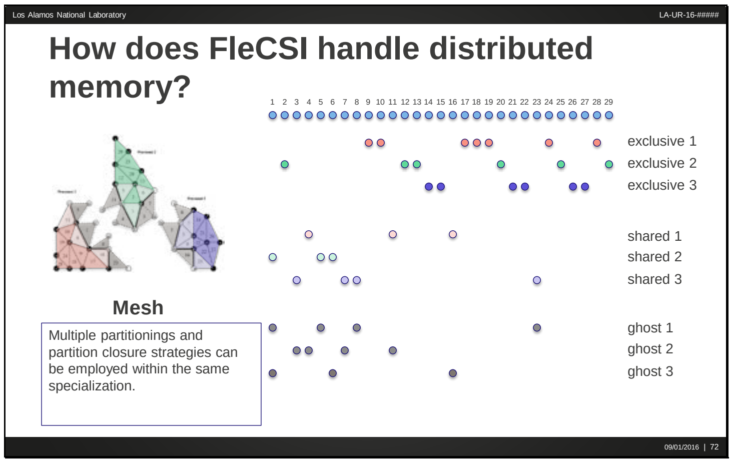
**Mesh**

Using a strategy defined by the specialization, the dependency closure of the mesh is formed, creating several sets of indices (index spaces).

**Exclusive:** I own them, and nobody else cares about them
**Shared:** I own them, and some other people care about them
**Ghost:** I don't own them, but I care about them

**This slide simply attempts to clarify the concepts of local, shared, and ghost.**

Slide 72

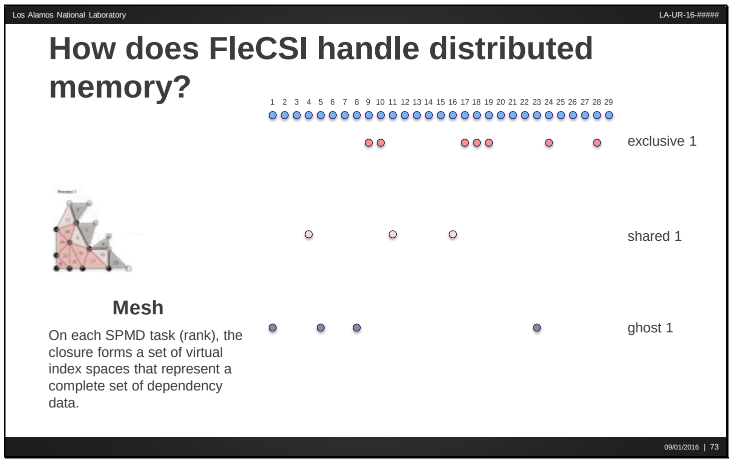


## How does FleCSI handle distributed memory?

Multiple partitionings and partition closure strategies can be employed within the same specialization.

**FleCSI supports multiple partitionings and closure strategies per specialization, e.g., a specialization might partition with respect to cells and with respect to vertices, forming two independent partition schemes and closures.**

Slide 73



## How does FleCSI handle distributed memory?

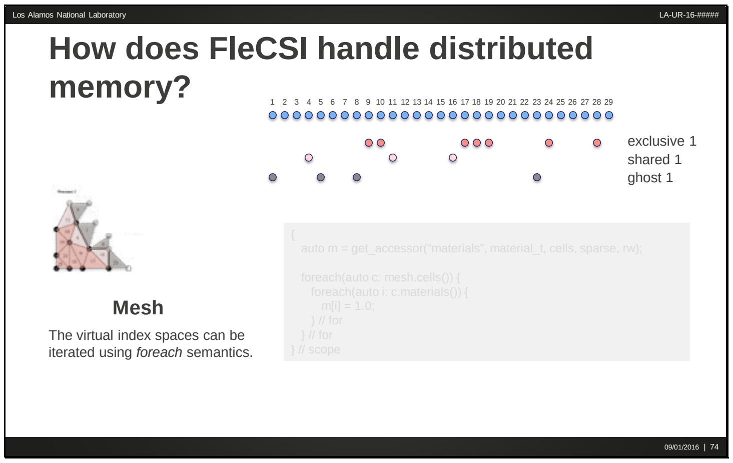
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
exclusive 1
shared 1
ghost 1

**Mesh**

On each SPMD task (rank), the closure forms a set of virtual index spaces that represent a complete set of dependency data.

**Each task or rank has a full set of its local, shared, and ghost data. The indices of these data make up a virutal index space. In the following slides, several subsets of this virtual index space are shown. Users can use these subset index spaces to iterate through particular logical subsets of the mesh or tree entities. In this case, the union of the index spaces on this slide create a virtual index space of mesh cells. The user can iterate over all of the cells, only the cells that are local to the rank or task, the local and shared cells, or the ghost cells. This provides flexibility to the user, and maintains a clean code interface.**

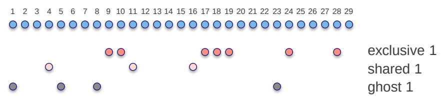
Slide 74

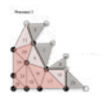


Los Alamos National Laboratory

LA-UR-16-#####

## How does FleCSI handle distributed memory?





**Mesh**

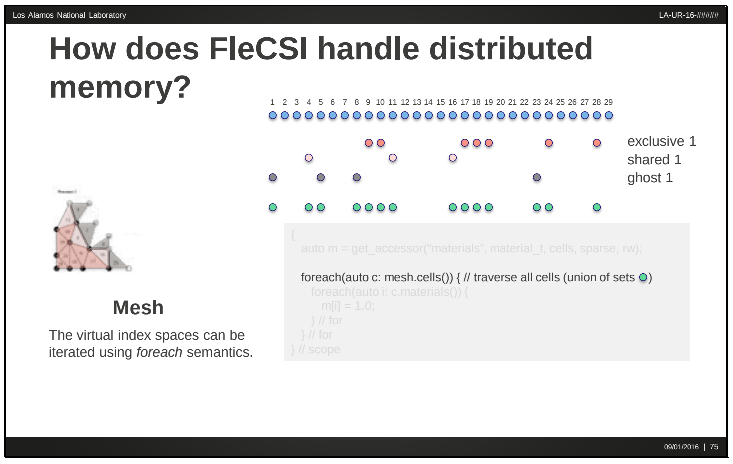
The virtual index spaces can be iterated using *foreach* semantics.

```
{
   auto m = get_accessor("materials", material_t, cells, sparse, rw);

   foreach(auto c: mesh.cells()) {
      foreach(auto i: c.materials()) {
         m[i] = 1.0;
      } // for
   } // for
} // scope
```

09/01/2016 | 74

**Virtual index space animation.**

Slide 75



## How does FleCSI handle distributed memory?

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
exclusive 1
shared 1
ghost 1

**Mesh**

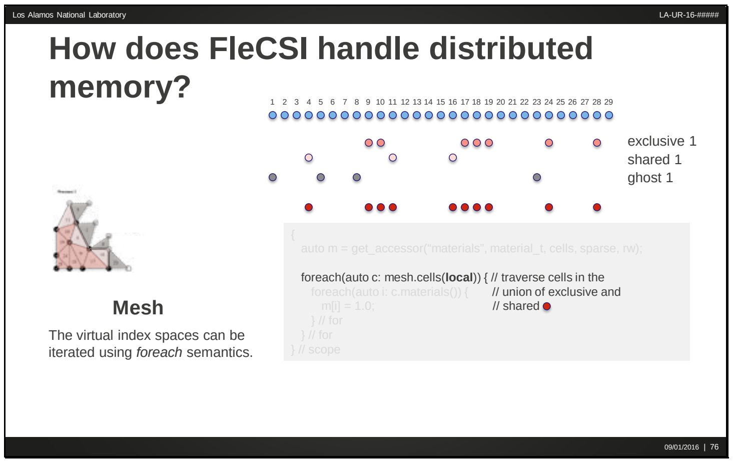
The virtual index spaces can be iterated using *foreach* semantics.

```
{
  auto m = get_accessor("materials", material_t, cells, sparse, rw);

  foreach(auto c: mesh.cells()) { // traverse all cells (union of sets ●)
    foreach(auto i: c.materials()) {
      m[i] = 1.0;
    } // for
  } // for
} // scope
```

**Virtual index space animation.**

**Showing iteration over all cells in the virtual index space.**

Slide 76



## How does FleCSI handle distributed memory?

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
exclusive 1
shared 1
ghost 1

```
{
  auto m = get_accessor("materials", material_t, cells, sparse, rw);

  foreach(auto c: mesh.cells(local)) { // traverse cells in the
    foreach(auto i: c.materials()) {       // union of exclusive and
      m[i] = 1.0;                          // shared ●
    } // for
  } // for
} // scope
```

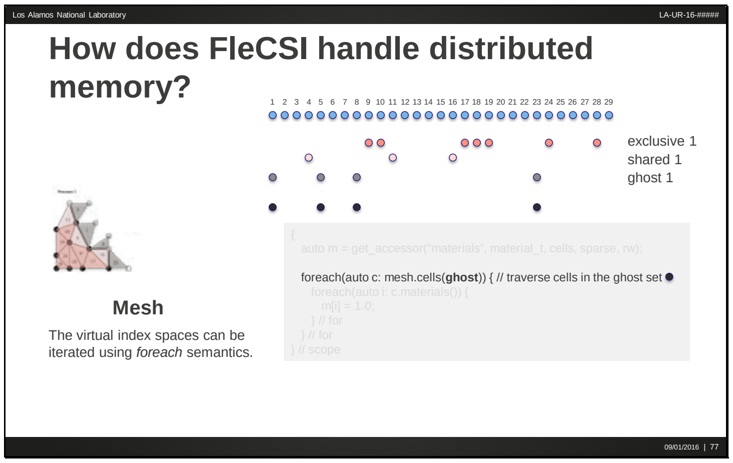
**Mesh**

The virtual index spaces can be iterated using *foreach* semantics.

**Virtual index space animation.**

**Showing iteration over the local and shared indices of the virutal index space.**

Slide 77



## How does FleCSI handle distributed memory?

**Mesh**

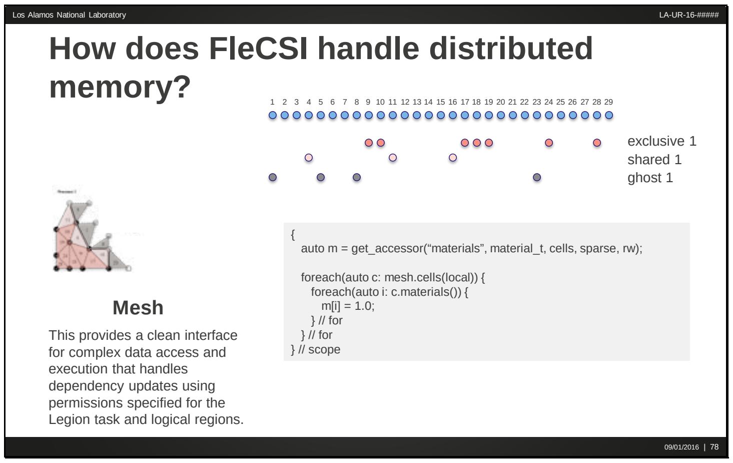
The virtual index spaces can be iterated using *foreach* semantics.

```
{
  auto m = get_accessor("materials", material_t, cells, sparse, rw);

  foreach(auto c: mesh.cells(ghost)) { // traverse cells in the ghost set ●
    foreach(auto i: c.materials()) {
      m[i] = 1.0;
    } // for
  } // for
} // scope
```

**Virtual index space animation.**

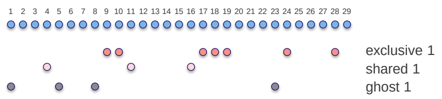
**Showing iteration over the ghost indices of the virutal index space.**

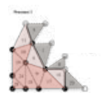
Slide 78



Los Alamos National Laboratory LA-UR-16-#####

## How does FleCSI handle distributed memory?





**Mesh**

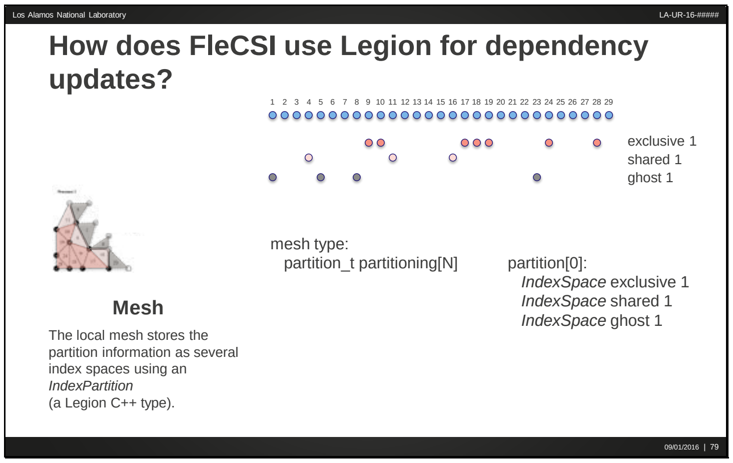
This provides a clean interface for complex data access and execution that handles dependency updates using permissions specified for the Legion task and logical regions.

```
{
   auto m = get_accessor(“materials”, material_t, cells, sparse, rw);

   foreach(auto c: mesh.cells(local)) {
      foreach(auto i: c.materials()) {
         m[i] = 1.0;
      } // for
   } // for
} // scope
```

09/01/2016 | 78

**This slide illustrates the utility of our approach and demonstrates that the user can develop very clean code that is semantically serial on a distributed-memory system.**

Slide 79


Los Alamos National Laboratory LA-UR-16-#####

## How does FleCSI use Legion for dependency updates?

**Mesh**

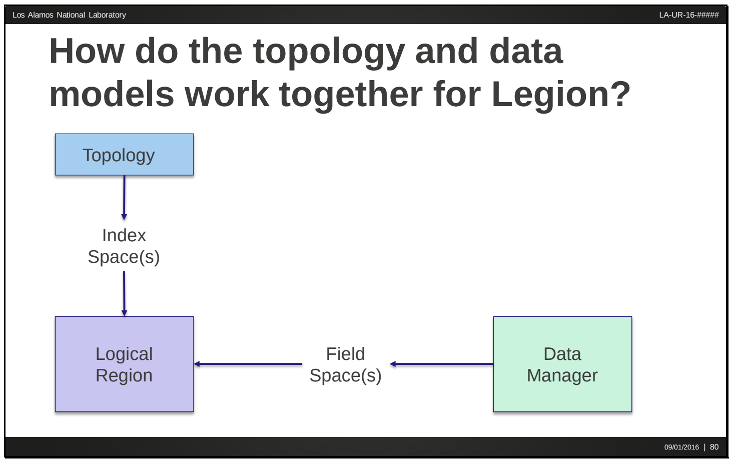
The local mesh stores the partition information as several index spaces using an *IndexPartition* (a Legion C++ type).

mesh type:
partition_t partitioning[N]

partition[0]:
*IndexSpace* exclusive 1
*IndexSpace* shared 1
*IndexSpace* ghost 1

09/01/2016 | 79

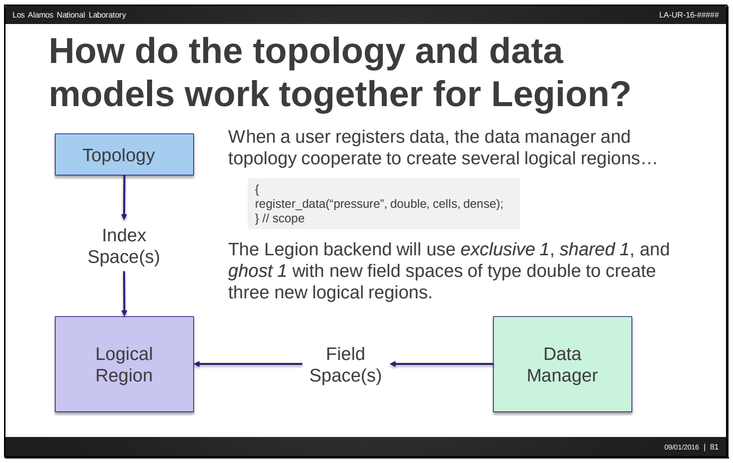
**The Legion backend to FleCSI uses Legion IndexSpace and IndexPartition data structures to store the index spaces for a virtual index space. These are stored with the Legion runtime context (FleCSI data structure) in a generalized partition representation. Each partition data object holds the information for the local, shared, and ghost index spaces. FleCSI uses this information to create logical regions that are stored as part of the context.**



## How do the topology and data models work together for Legion?

When a user registers data, the data manager creates and appropriate set of field spaces for the virtual index space that has been specified by the user. The topology instance (a data_client_t), e.g., the mesh, provides the index partition and index space information.

Slide 81



## How do the topology and data models work together for Legion?

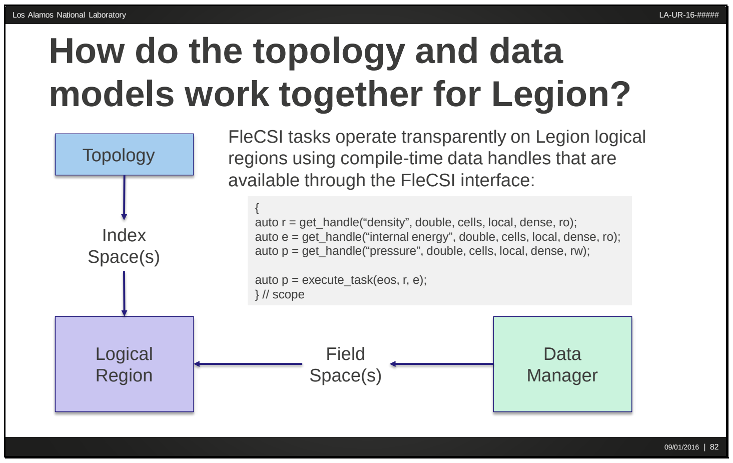
When a user registers data, the data manager and topology cooperate to create several logical regions…

```
{
register_data("pressure", double, cells, dense);
} // scope
```

The Legion backend will use *exclusive 1*, *shared 1*, and *ghost 1* with new field spaces of type double to create three new logical regions.

**Together, the index spaces and field spaces are used to create a logical region. The logical region is stored by the FleCSI context, and is available for tasks to use transparent to the user.**

Slide 82



## How do the topology and data models work together for Legion?

Topology
Index Space(s)
Logical Region
Field Space(s)
Data Manager

FleCSI tasks operate transparently on Legion logical regions using compile-time data handles that are available through the FleCSI interface:

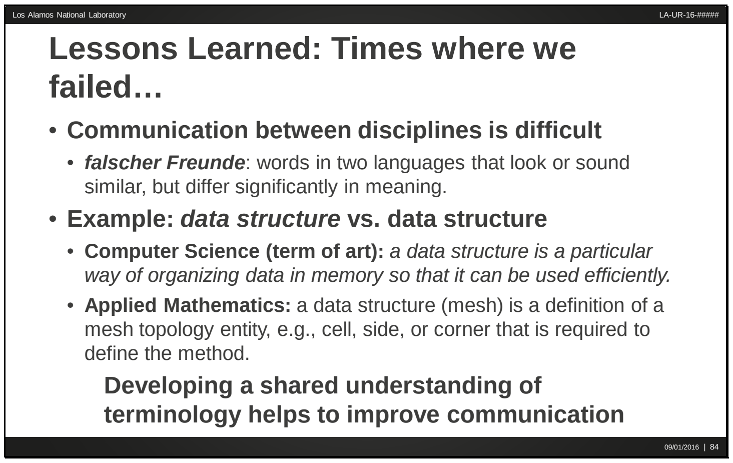
```
{
auto r = get_handle("density", double, cells, local, dense, ro);
auto e = get_handle("internal energy", double, cells, local, dense, ro);
auto p = get_handle("pressure", double, cells, local, dense, rw);

auto p = execute_task(eos, r, e);
} // scope
```

**Users implement their tasks using data handles that, internally, are connected to the correct index space and field space of a logical region. Using some static metaprogramming techniques, these data are mapped and transformed into accessors. The accessors act like C language arrays, e.g., arg[i] = 1.0, so that the user can directly read or mutate the data. Permissions are granted through the handle interface, i.e., the user specifies the required permissions when they request a handle to data.**

Slide 83

## FleCSI Lessons Learned

Slide 84

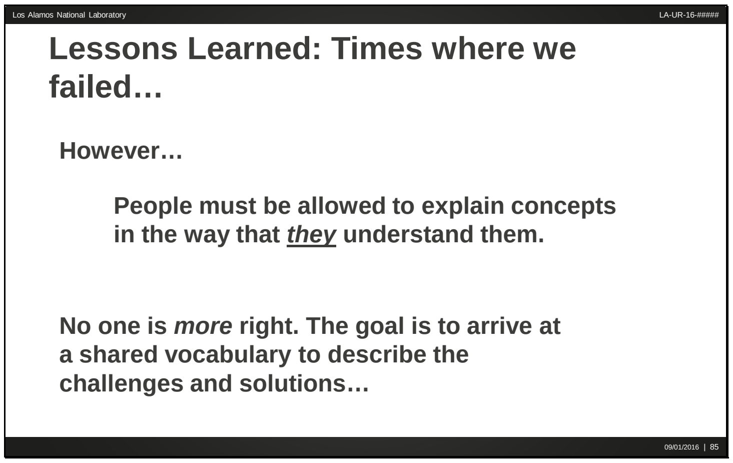


## Lessons Learned: Times where we failed…

- **Communication between disciplines is difficult**
  - ***falscher Freunde***: words in two languages that look or sound similar, but differ significantly in meaning.
- **Example: *data structure* vs. data structure**
  - **Computer Science (term of art):** *a data structure is a particular way of organizing data in memory so that it can be used efficiently.*
  - **Applied Mathematics:** a data structure (mesh) is a definition of a mesh topology entity, e.g., cell, side, or corner that is required to define the method.

**Developing a shared understanding of terminology helps to improve communication**

Slide 85



## Lessons Learned: Times where we failed…

However…

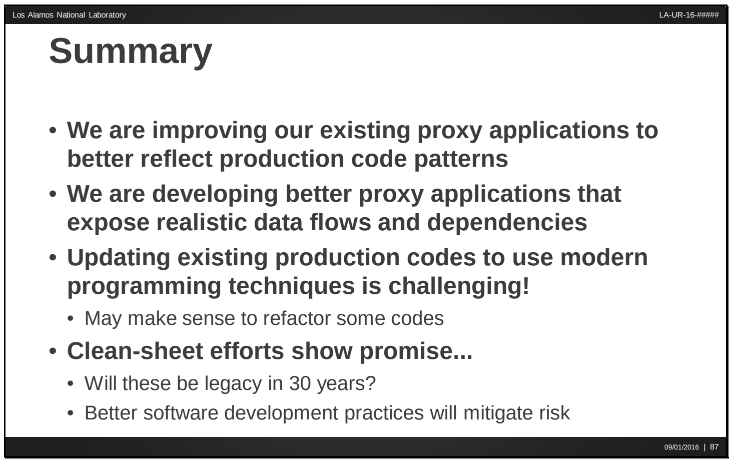
People must be allowed to explain concepts in the way that ***they*** understand them.

No one is ***more*** right. The goal is to arrive at a shared vocabulary to describe the challenges and solutions…

Slide 86

## Summary

Slide 87

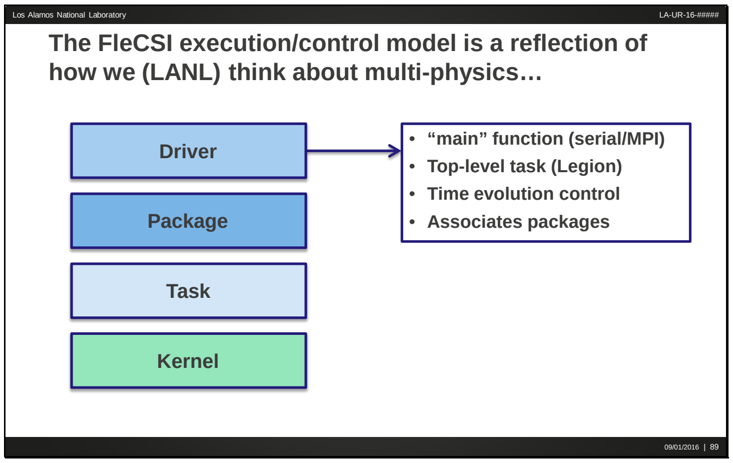


## Summary

- **We are improving our existing proxy applications to better reflect production code patterns**
- **We are developing better proxy applications that expose realistic data flows and dependencies**
- **Updating existing production codes to use modern programming techniques is challenging!**
  - May make sense to refactor some codes
- **Clean-sheet efforts show promise...**
  - Will these be legacy in 30 years?
  - Better software development practices will mitigate risk

Slide 88

## FleCSI Control & Execution Models



Los Alamos National Laboratory LA-UR-16-#####

## The FleCSI execution/control model is a reflection of how we (LANL) think about multi-physics…

Driver
Package
Task
Kernel

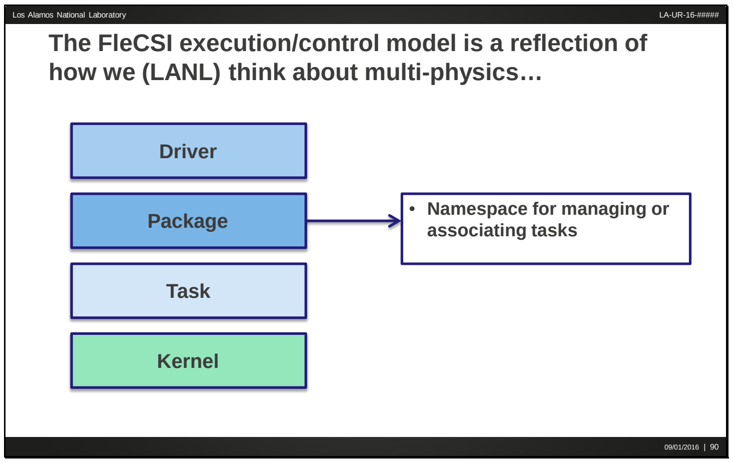
- "main" function (serial/MPI)
- Top-level task (Legion)
- Time evolution control
- Associates packages

09/01/2016 | 89

FleCSI has an intuitive execution model, which includes a control layer (control model is under development), a task layer, and a kernel layer. There is additional support for defining functions that may be called from within a task or kernel. In the simple case, a function call is trivially executed directly. However, in some cases (think virtual function support) more steps are need to insure that a function call is valid in any address space in which it may be executed.

Driver Layer: This is where the top-level control logic of the simulation lives.

Slide 90



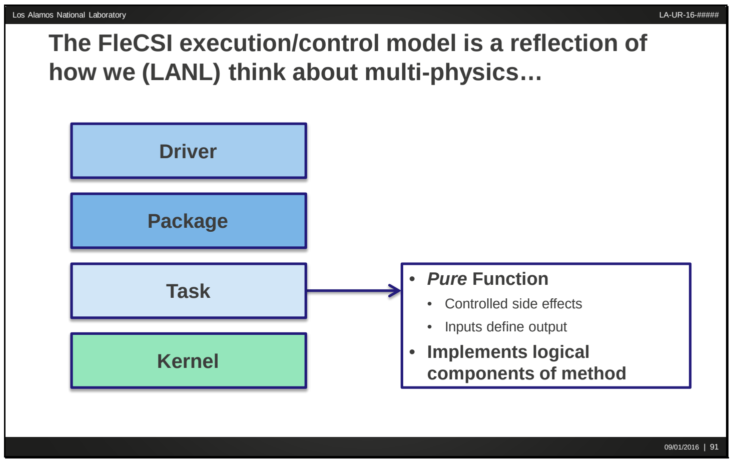
## The FleCSI execution/control model is a reflection of how we (LANL) think about multi-physics…

Driver
Package
Task
Kernel

- Namespace for managing or associating tasks

The package layer is simply a namespace to allow users to logically group different tasks that have a common purpose.

Slide 91


Los Alamos National Laboratory LA-UR-16-#####

## The FleCSI execution/control model is a reflection of how we (LANL) think about multi-physics…

Driver
Package
Task
Kernel

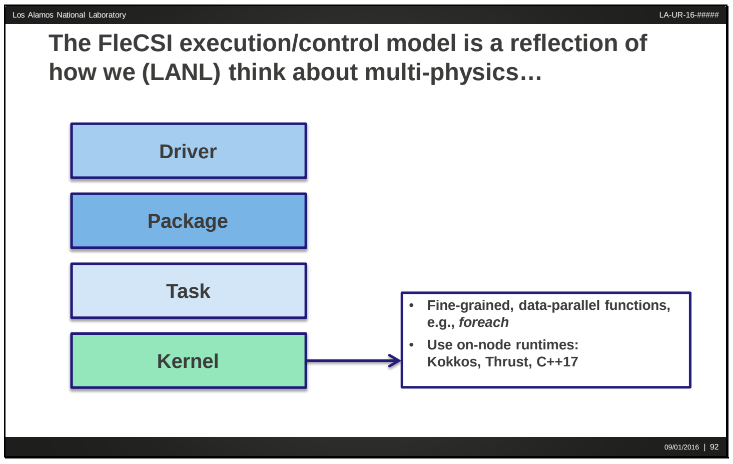
- *Pure* Function
  - Controlled side effects
  - Inputs define output
- Implements logical components of method

09/01/2016 | 91

**FleCSI tasks have controlled side effects, i.e., those, about which the runtime can reason. Within a task, a developer or user can assume that execution is happening in a single address space.**

**The task abstraction in FleCSI is based on previous co-design research on task-based runtime models, e.g., Legion, STAPL, Charm++, and OCR (Intel). FleCSI's task layer was also influenced by discussions between LANL, Intel and Stanford (Tim Mattson of Intel organized a discussion group to investigate this topic. Ben Bergen and David Daniel both participated in these discussions. Ben Bergen, Pat McCormick, et. al participated in discussions with Stanford and Intel on requirements and design for task-based runtimes at Intel's Hillsboro location.**

Slide 92


Los Alamos National Laboratory LA-UR-16-#####

## The FleCSI execution/control model is a reflection of how we (LANL) think about multi-physics…

- Fine-grained, data-parallel functions, e.g., *foreach*
- Use on-node runtimes: Kokkos, Thrust, C++17

09/01/2016 | 92

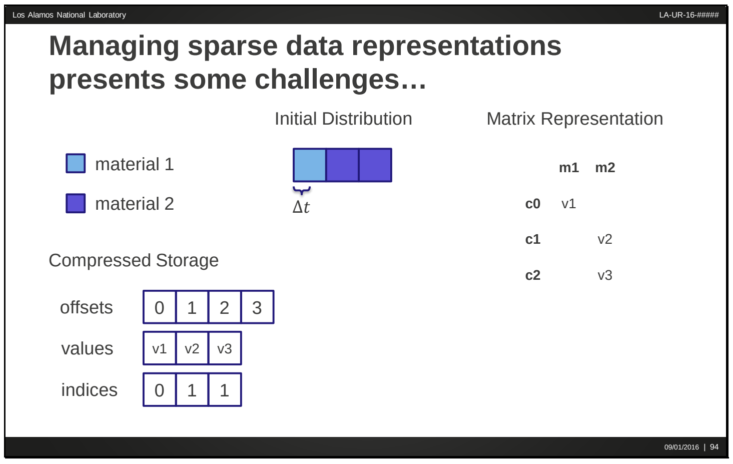
**A FleCSI kernel is a fine-grained, data-parallel unit of execution. Our current implementation strategy is to use Kokkos until the C++17 standard is available. Kernels execute in a relaxed consistency mode, although some operations may depend on sequential consistency when it is possible to reason about and expect that the underlying hardware supports sequential consistency.**

**The FleCSI kernel abstraction is based on experience with Kokkos, OpenCL and CUDA, although it is most similar to the Kokkos model. C++17 will directly support many of the interface requirements of the FleCSI kernel model (Thanks to the Kokkos team for their efforts on the C++17 committee!). Our experience with the Kokkos programming model during the FY15 milestone was extremely influential in the design of the FleCSI kernel abstraction.**

Slide 93

## Sparse Data Representations

Slide 94



## Managing sparse data representations presents some challenges…

| | |
|---|---|
| 🟦 | material 1 |
| 🟪 | material 2 |

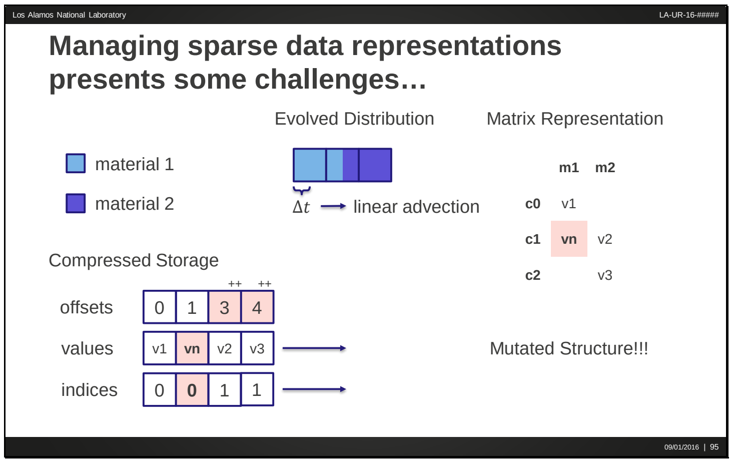
Matrix Representation

$$\begin{array}{c|cc} & \mathbf{m1} & \mathbf{m2} \\ \hline \mathbf{c0} & v1 & \\ \mathbf{c1} & & v2 \\ \mathbf{c2} & & v3 \end{array}$$

Compressed Storage

| | | | | |
|---|---|---|---|---|
| offsets | 0 | 1 | 2 | 3 |
| values | v1 | v2 | v3 | |
| indices | 0 | 1 | 1 | |

Slide 95



## Managing sparse data representations presents some challenges…

Evolved Distribution

Matrix Representation

🟦 material 1

🟪 material 2

$\Delta t \longrightarrow$ linear advection

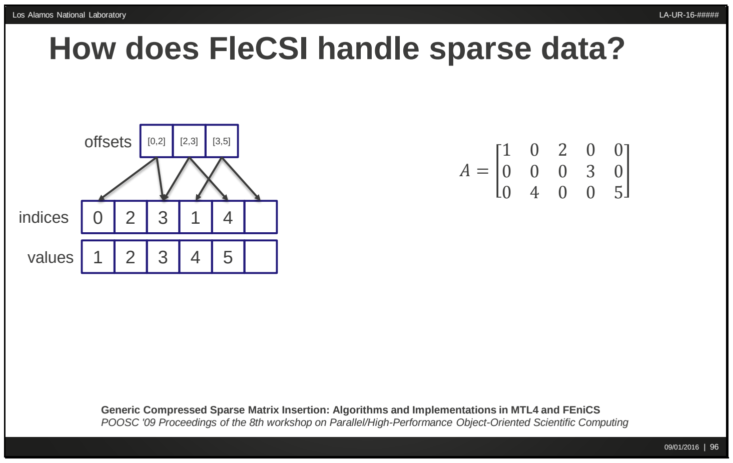
| | m1 | m2 |
|---|---|---|
| c0 | v1 | |
| c1 | vn | v2 |
| c2 | | v3 |

Compressed Storage

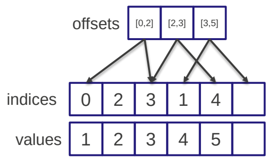
| | | | ++ | ++ |
|---|---|---|---|---|
| offsets | 0 | 1 | 3 | 4 |
| values | v1 | vn | v2 | v3 |
| indices | 0 | 0 | 1 | 1 |

Mutated Structure!!!

Slide 96



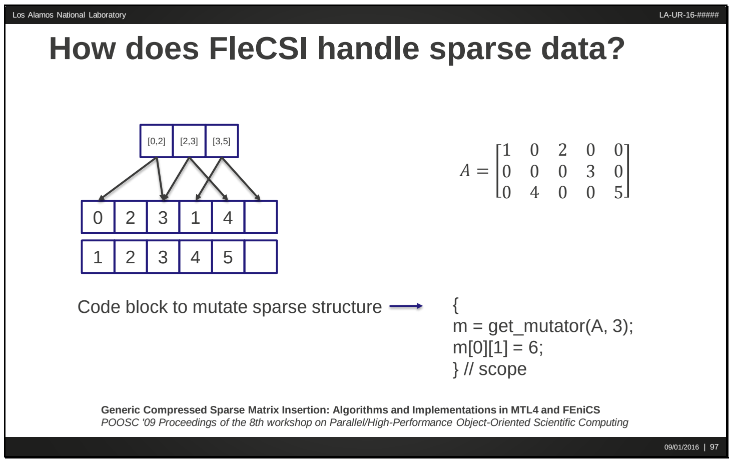
## How does FleCSI handle sparse data?



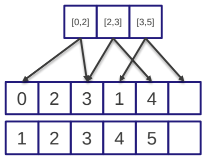
$$A = \begin{bmatrix} 1 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \\ 0 & 4 & 0 & 0 & 5 \end{bmatrix}$$

**Generic Compressed Sparse Matrix Insertion: Algorithms and Implementations in MTL4 and FEniCS**
*POOSC '09 Proceedings of the 8th workshop on Parallel/High-Performance Object-Oriented Scientific Computing*

Slide 97



## How does FleCSI handle sparse data?



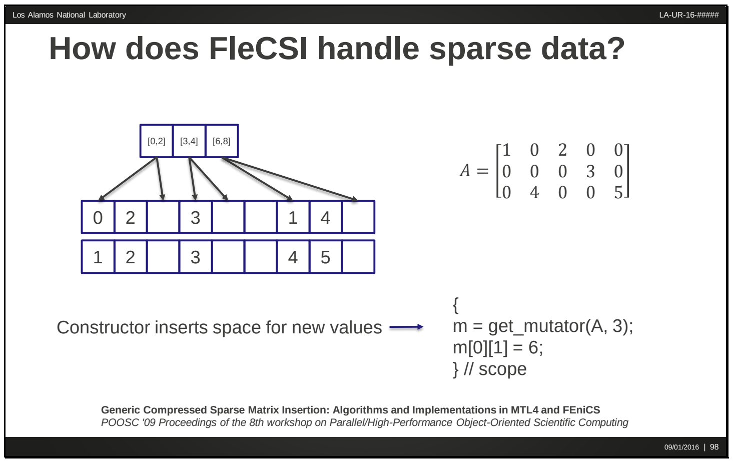
$$A = \begin{bmatrix} 1 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \\ 0 & 4 & 0 & 0 & 5 \end{bmatrix}$$

Code block to mutate sparse structure $\longrightarrow$

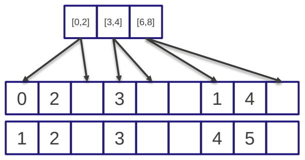
```
{
m = get_mutator(A, 3);
m[0][1] = 6;
} // scope
```

**Generic Compressed Sparse Matrix Insertion: Algorithms and Implementations in MTL4 and FEniCS**
*POOSC '09 Proceedings of the 8th workshop on Parallel/High-Performance Object-Oriented Scientific Computing*

Slide 98



## How does FleCSI handle sparse data?



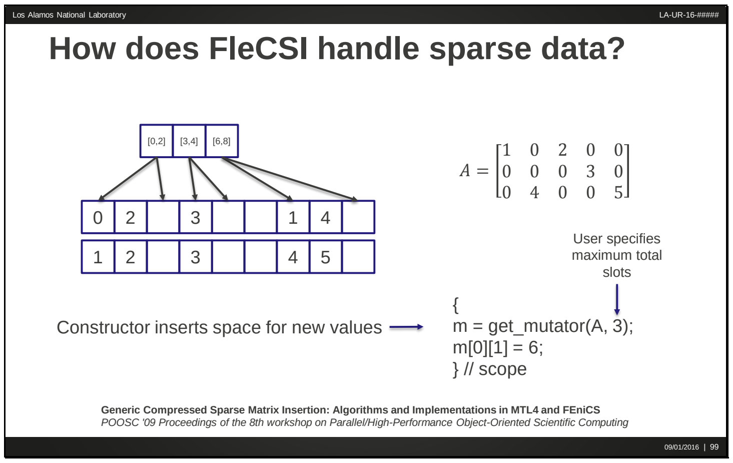
$$A = \begin{bmatrix} 1 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \\ 0 & 4 & 0 & 0 & 5 \end{bmatrix}$$

Constructor inserts space for new values $\longrightarrow$

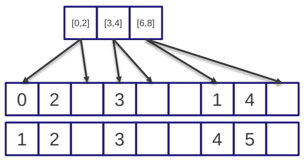
```
{
m = get_mutator(A, 3);
m[0][1] = 6;
} // scope
```

**Generic Compressed Sparse Matrix Insertion: Algorithms and Implementations in MTL4 and FEniCS**
*POOSC '09 Proceedings of the 8th workshop on Parallel/High-Performance Object-Oriented Scientific Computing*

Slide 99



## How does FleCSI handle sparse data?



$$A = \begin{bmatrix} 1 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \\ 0 & 4 & 0 & 0 & 5 \end{bmatrix}$$

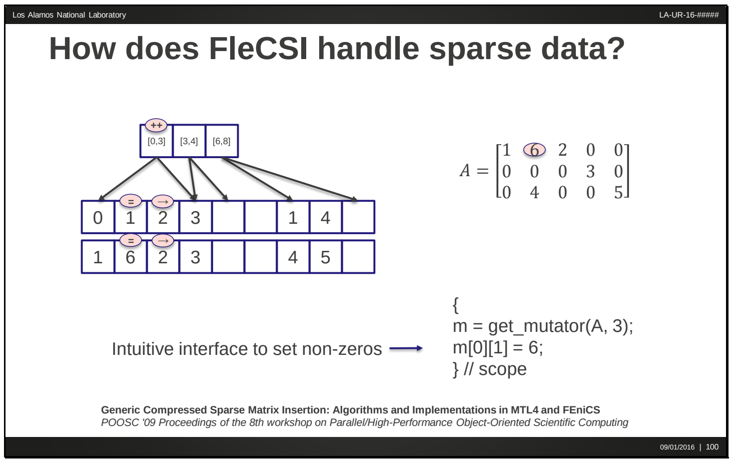
User specifies maximum total slots

Constructor inserts space for new values

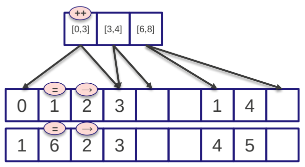
```
{
m = get_mutator(A, 3);
m[0][1] = 6;
} // scope
```

**Generic Compressed Sparse Matrix Insertion: Algorithms and Implementations in MTL4 and FEniCS**
*POOSC '09 Proceedings of the 8th workshop on Parallel/High-Performance Object-Oriented Scientific Computing*

Slide 100



## How does FleCSI handle sparse data?



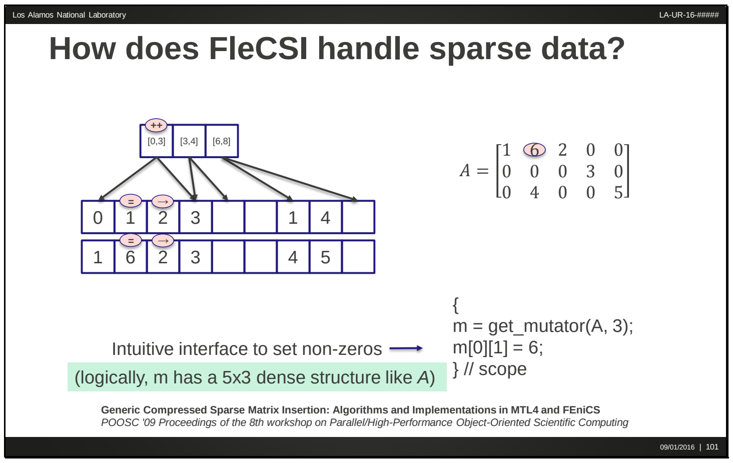
$$A = \begin{bmatrix} 1 & 6 & 2 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \\ 0 & 4 & 0 & 0 & 5 \end{bmatrix}$$

Intuitive interface to set non-zeros $\longrightarrow$

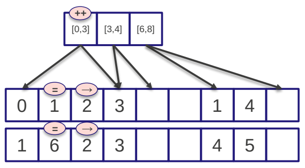
```
{
m = get_mutator(A, 3);
m[0][1] = 6;
} // scope
```

**Generic Compressed Sparse Matrix Insertion: Algorithms and Implementations in MTL4 and FEniCS**
*POOSC '09 Proceedings of the 8th workshop on Parallel/High-Performance Object-Oriented Scientific Computing*

Slide 101



## How does FleCSI handle sparse data?



$$A = \begin{bmatrix} 1 & 6 & 2 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \\ 0 & 4 & 0 & 0 & 5 \end{bmatrix}$$

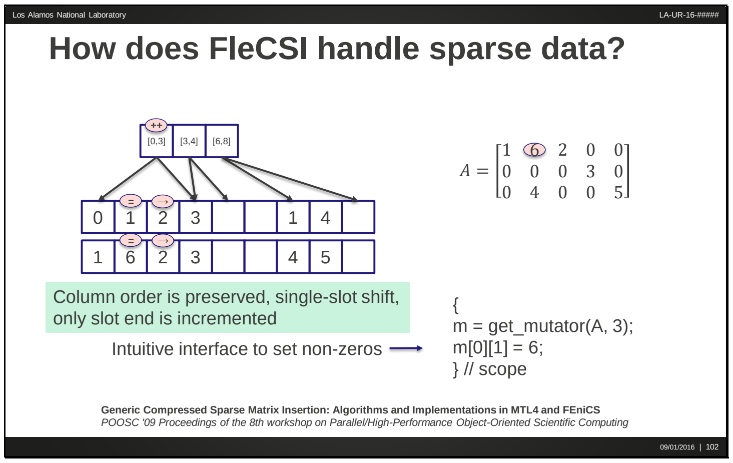
Intuitive interface to set non-zeros →

```
{
m = get_mutator(A, 3);
m[0][1] = 6;
} // scope
```

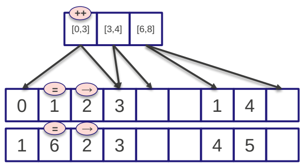
(logically, m has a 5x3 dense structure like *A*)

**Generic Compressed Sparse Matrix Insertion: Algorithms and Implementations in MTL4 and FEniCS**
*POOSC '09 Proceedings of the 8th workshop on Parallel/High-Performance Object-Oriented Scientific Computing*

Slide 102



## How does FleCSI handle sparse data?



$$A = \begin{bmatrix} 1 & 6 & 2 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \\ 0 & 4 & 0 & 0 & 5 \end{bmatrix}$$

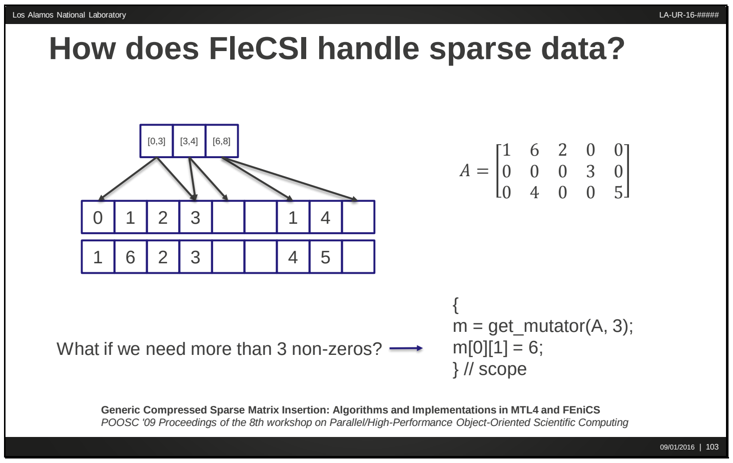
Column order is preserved, single-slot shift, only slot end is incremented

Intuitive interface to set non-zeros →

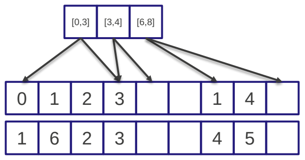
```
{
m = get_mutator(A, 3);
m[0][1] = 6;
} // scope
```

**Generic Compressed Sparse Matrix Insertion: Algorithms and Implementations in MTL4 and FEniCS**
*POOSC '09 Proceedings of the 8th workshop on Parallel/High-Performance Object-Oriented Scientific Computing*

Slide 103



## How does FleCSI handle sparse data?



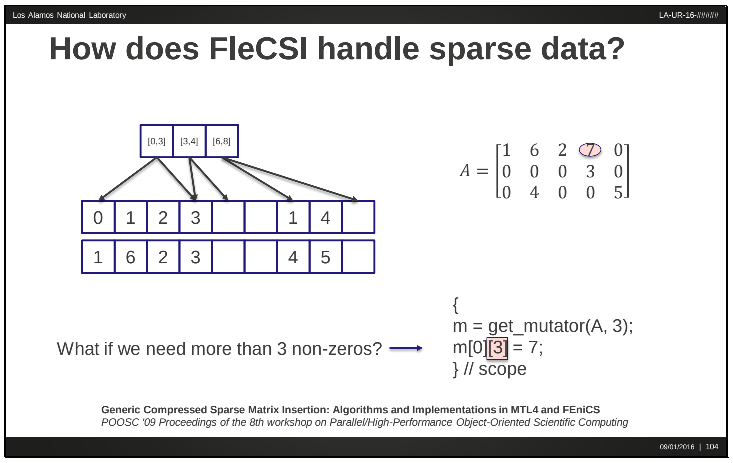
$$A = \begin{bmatrix} 1 & 6 & 2 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \\ 0 & 4 & 0 & 0 & 5 \end{bmatrix}$$

What if we need more than 3 non-zeros? $\longrightarrow$

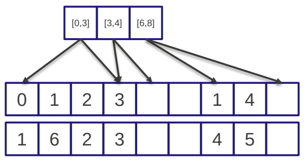
```
{
m = get_mutator(A, 3);
m[0][1] = 6;
} // scope
```

**Generic Compressed Sparse Matrix Insertion: Algorithms and Implementations in MTL4 and FEniCS**
*POOSC '09 Proceedings of the 8th workshop on Parallel/High-Performance Object-Oriented Scientific Computing*

Slide 104



## How does FleCSI handle sparse data?



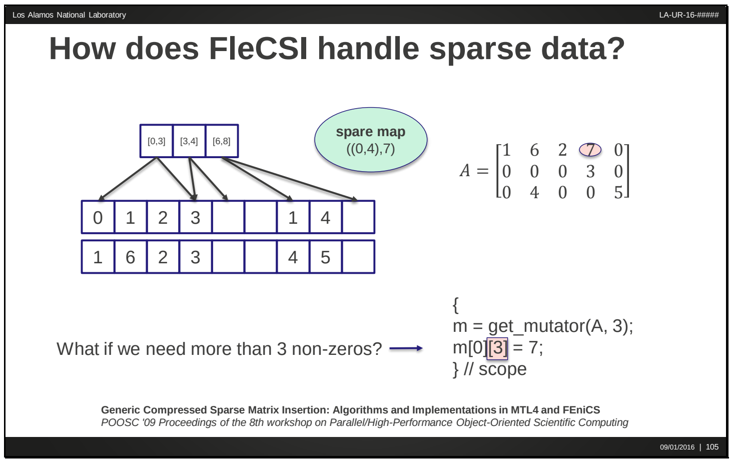
$$A = \begin{bmatrix} 1 & 6 & 2 & 7 & 0 \\ 0 & 0 & 0 & 3 & 0 \\ 0 & 4 & 0 & 0 & 5 \end{bmatrix}$$

What if we need more than 3 non-zeros? $\longrightarrow$

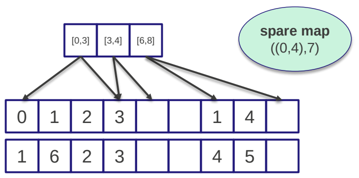
```
{
m = get_mutator(A, 3);
m[0][3] = 7;
} // scope
```

**Generic Compressed Sparse Matrix Insertion: Algorithms and Implementations in MTL4 and FEniCS**
*POOSC '09 Proceedings of the 8th workshop on Parallel/High-Performance Object-Oriented Scientific Computing*

Slide 105



## How does FleCSI handle sparse data?



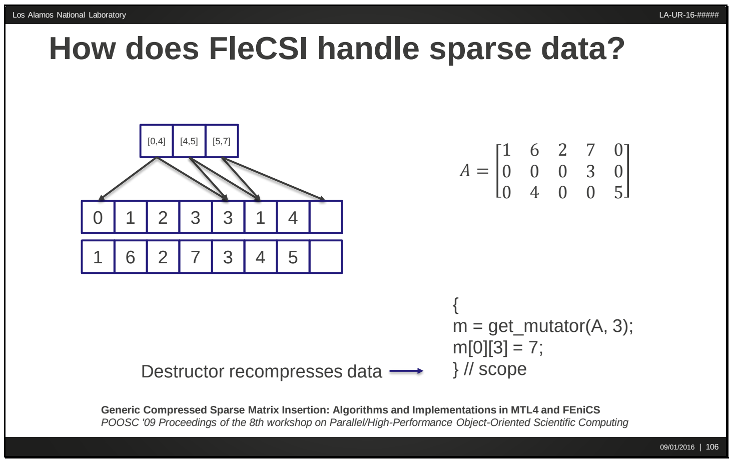
$$A = \begin{bmatrix} 1 & 6 & 2 & 7 & 0 \\ 0 & 0 & 0 & 3 & 0 \\ 0 & 4 & 0 & 0 & 5 \end{bmatrix}$$

What if we need more than 3 non-zeros? $\longrightarrow$

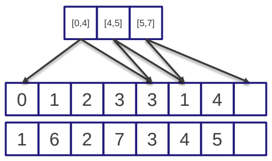
```
{
m = get_mutator(A, 3);
m[0][3] = 7;
} // scope
```

**Generic Compressed Sparse Matrix Insertion: Algorithms and Implementations in MTL4 and FEniCS**
*POOSC '09 Proceedings of the 8th workshop on Parallel/High-Performance Object-Oriented Scientific Computing*

Slide 106



## How does FleCSI handle sparse data?



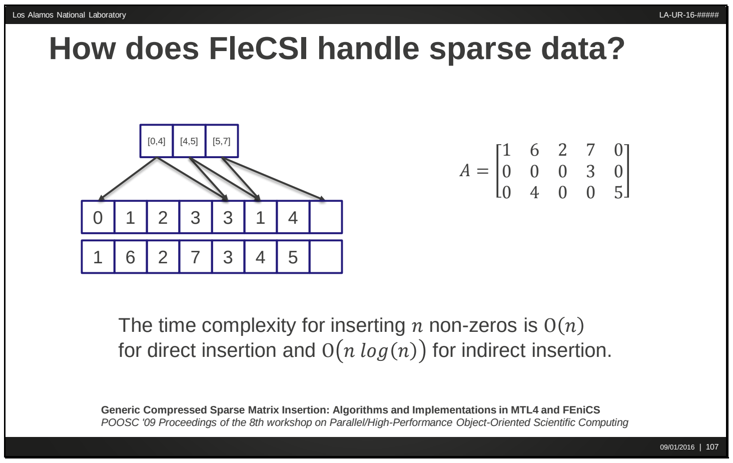
$$A = \begin{bmatrix} 1 & 6 & 2 & 7 & 0 \\ 0 & 0 & 0 & 3 & 0 \\ 0 & 4 & 0 & 0 & 5 \end{bmatrix}$$

```
{
m = get_mutator(A, 3);
m[0][3] = 7;
} // scope
```

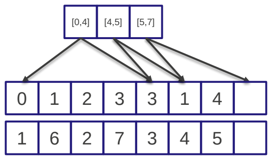
Destructor recompresses data $\longrightarrow$

**Generic Compressed Sparse Matrix Insertion: Algorithms and Implementations in MTL4 and FEniCS**
*POOSC '09 Proceedings of the 8th workshop on Parallel/High-Performance Object-Oriented Scientific Computing*

Slide 107



## How does FleCSI handle sparse data?

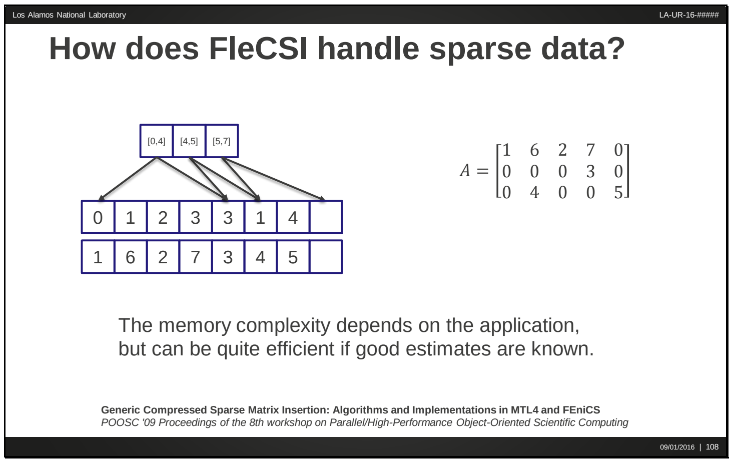


$$A = \begin{bmatrix} 1 & 6 & 2 & 7 & 0 \\ 0 & 0 & 0 & 3 & 0 \\ 0 & 4 & 0 & 0 & 5 \end{bmatrix}$$

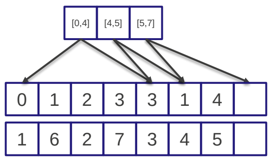
The time complexity for inserting $n$ non-zeros is $O(n)$ for direct insertion and $O\big(n\ log(n)\big)$ for indirect insertion.

**Generic Compressed Sparse Matrix Insertion: Algorithms and Implementations in MTL4 and FEniCS**
*POOSC '09 Proceedings of the 8th workshop on Parallel/High-Performance Object-Oriented Scientific Computing*

Slide 108



## How does FleCSI handle sparse data?

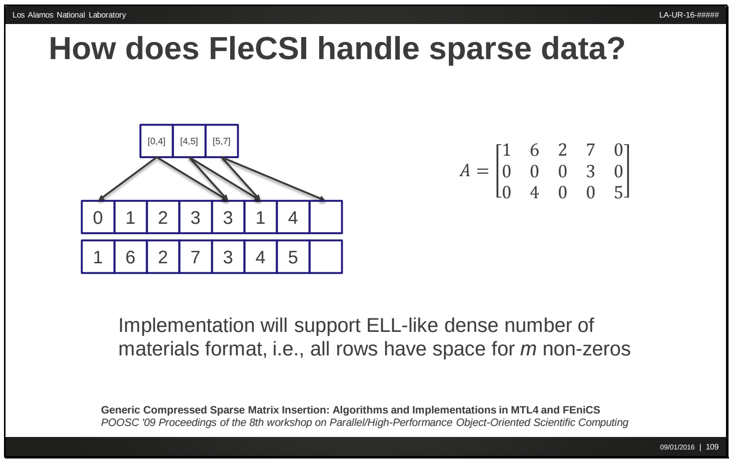


$$A = \begin{bmatrix} 1 & 6 & 2 & 7 & 0 \\ 0 & 0 & 0 & 3 & 0 \\ 0 & 4 & 0 & 0 & 5 \end{bmatrix}$$

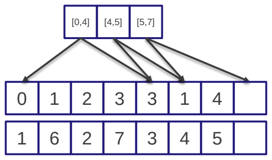
The memory complexity depends on the application, but can be quite efficient if good estimates are known.

**Generic Compressed Sparse Matrix Insertion: Algorithms and Implementations in MTL4 and FEniCS**
*POOSC '09 Proceedings of the 8th workshop on Parallel/High-Performance Object-Oriented Scientific Computing*

Slide 109



## How does FleCSI handle sparse data?

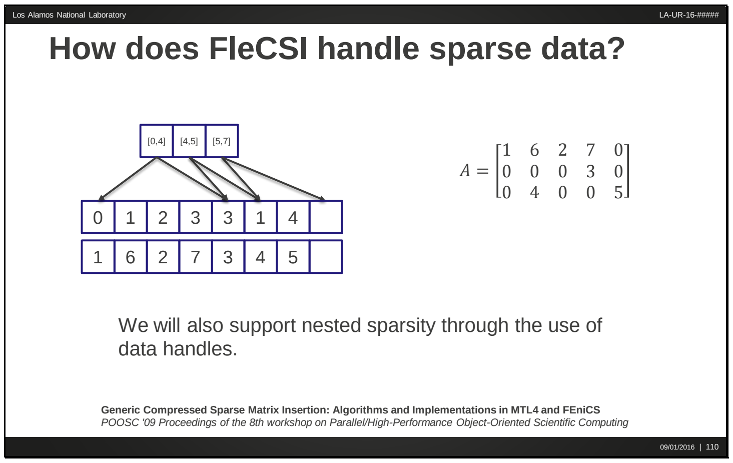


$$A = \begin{bmatrix} 1 & 6 & 2 & 7 & 0 \\ 0 & 0 & 0 & 3 & 0 \\ 0 & 4 & 0 & 0 & 5 \end{bmatrix}$$

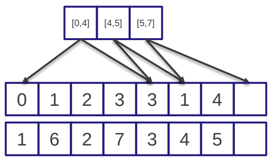
Implementation will support ELL-like dense number of materials format, i.e., all rows have space for *m* non-zeros

**Generic Compressed Sparse Matrix Insertion: Algorithms and Implementations in MTL4 and FEniCS**
*POOSC '09 Proceedings of the 8th workshop on Parallel/High-Performance Object-Oriented Scientific Computing*

Slide 110



## How does FleCSI handle sparse data?

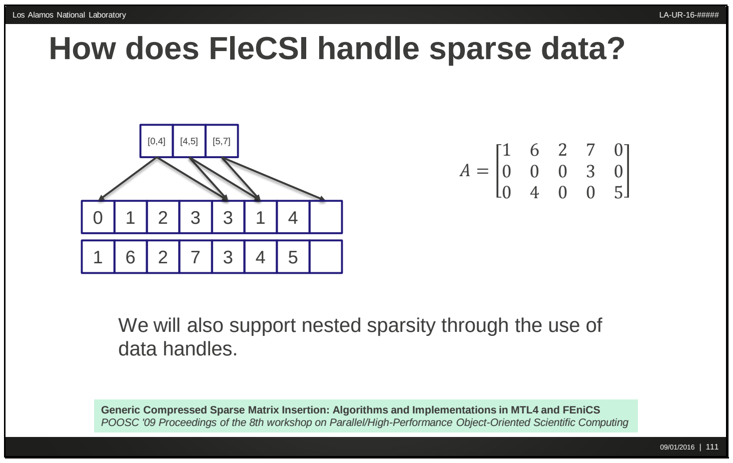


$$A = \begin{bmatrix} 1 & 6 & 2 & 7 & 0 \\ 0 & 0 & 0 & 3 & 0 \\ 0 & 4 & 0 & 0 & 5 \end{bmatrix}$$

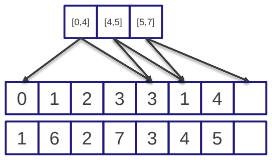
We will also support nested sparsity through the use of data handles.

**Generic Compressed Sparse Matrix Insertion: Algorithms and Implementations in MTL4 and FEniCS**
*POOSC '09 Proceedings of the 8th workshop on Parallel/High-Performance Object-Oriented Scientific Computing*

Slide 111



## How does FleCSI handle sparse data?



$$A = \begin{bmatrix} 1 & 6 & 2 & 7 & 0 \\ 0 & 0 & 0 & 3 & 0 \\ 0 & 4 & 0 & 0 & 5 \end{bmatrix}$$

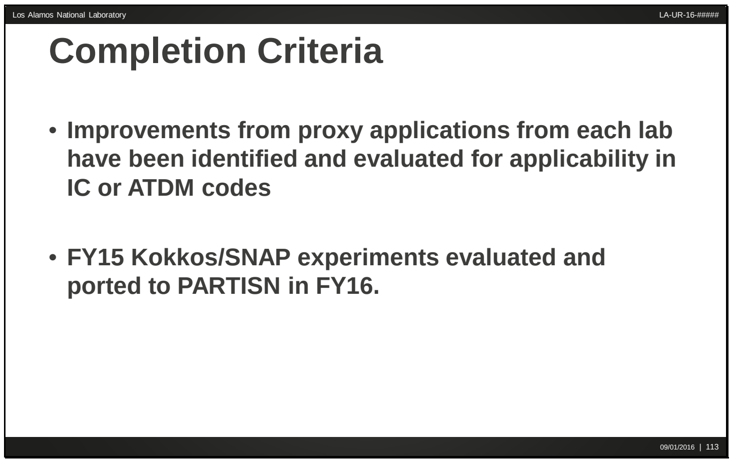
We will also support nested sparsity through the use of data handles.

**Generic Compressed Sparse Matrix Insertion: Algorithms and Implementations in MTL4 and FEniCS**
*POOSC '09 Proceedings of the 8th workshop on Parallel/High-Performance Object-Oriented Scientific Computing*

Slide 112

## Summary

Slide 113



Los Alamos National Laboratory

LA-UR-16-#####

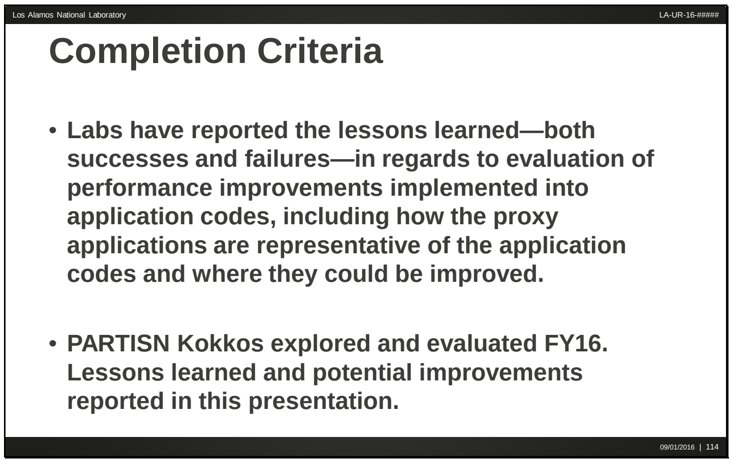
## Completion Criteria

- Improvements from proxy applications from each lab have been identified and evaluated for applicability in IC or ATDM codes
- FY15 Kokkos/SNAP experiments evaluated and ported to PARTISN in FY16.

09/01/2016 | 113

**Kokkos was shown to be effective in FY15 in implementing a C++ version of SNAP's kernel. This same methodology was applied to a production IC code, PARTISN. This was a much more complex endeavour than in FY15 for many reasons; a C++ kernel embedded in Fortran, overloading Fortran memory allocations, general language interoperability, and a fully fleshed out production code versus a simplified proxy code.**

Slide 114


Los Alamos National Laboratory LA-UR-16-#####

## Completion Criteria

- Labs have reported the lessons learned—both successes and failures—in regards to evaluation of performance improvements implemented into application codes, including how the proxy applications are representative of the application codes and where they could be improved.
- PARTISN Kokkos explored and evaluated FY16. Lessons learned and potential improvements reported in this presentation.

09/01/2016 | 114

**Lessons learned are Legion. In no particular order: Interoperability between Fortran and C++ was really not that hard, and a useful engineering effort. Tracking down all necessary memory allocations for a kernel in a production code is pretty hard. Modifying a production code to work for more than a handful of use cases is also pretty hard. Figuring out the toolchain that will allow a successful implementation of design decisions is quite hard, if making use of "bleeding edge" design choices. In terms of performance, production code concurrency architecture can be a virtual showstopper; being too complex to easily rewrite and test in a short period of time, or depending on tool features which do not exist yet. Ultimately, while the tools used in this work were not successful in speeding up the production code, they helped to identify how work would be done, and provide requirements to tools.**