LA-UR-16-26040
Approved for public release; distribution is unlimited.

Title: FleCSI connection to Legion

Author(s): Bergen, Benjamin Karl

Intended for: ASC L2 Milestone Review

Issued: 2016-08-04

Los Alamos
NATIONAL LABORATORY
EST.1943

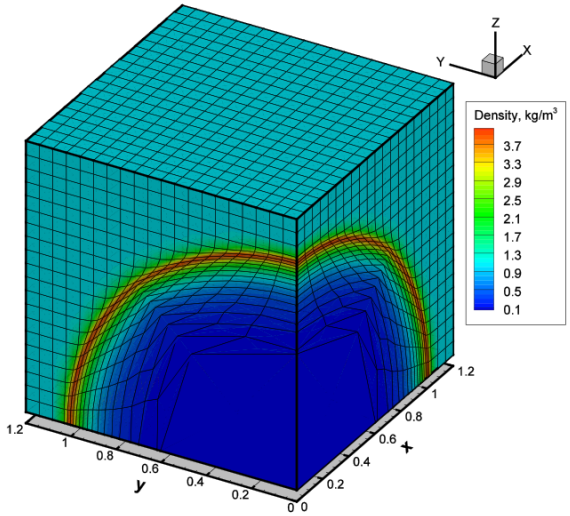
Delivering science and technology
to protect our nation
and promote world stability

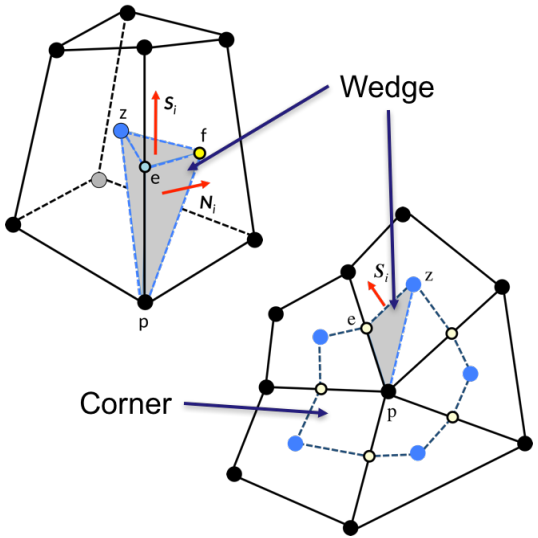
FleCSI: Connection to Legion

Ben Bergen

Wednesday, August 3rd 2016

NNSA
National Nuclear Security Administration
Operated by Los Alamos National Security, LLC for the U.S. Department of Energy's NNSA





- Acknowledgements
- Legion Backend
- Distributed-Memory Partitioning
- Sparse Data Representations
- MPI–Legion Interoperability

Many people have made significant contributions to FleCSI…

Nick Moss (CCS-7) 17,721 ++ / 10,167 -- **(0.25 FTE)**

Topology types, sparse data, design

Marc Charest (XCP-1) 19,582 ++ / 6,074 --

Mesh specialization, data structures, design

Irina Demeshko (CCS-7)

Legion backend, MPI/Legion interoperability, design

John Wohlbier (formerly CCS-2)

Mesh specialization, I/O

Josh Payne (CCS-7)

Execution model, design

Gary Dilts (CCS-2)

Tree specialization, data structures, design

Nathaniel Morgan (XCP-8), **Vince Chiravalle (A-2), Joe Schmidt (XTD-NTA), Rao Garimella (T-7), Wes Even (CCS-2)**

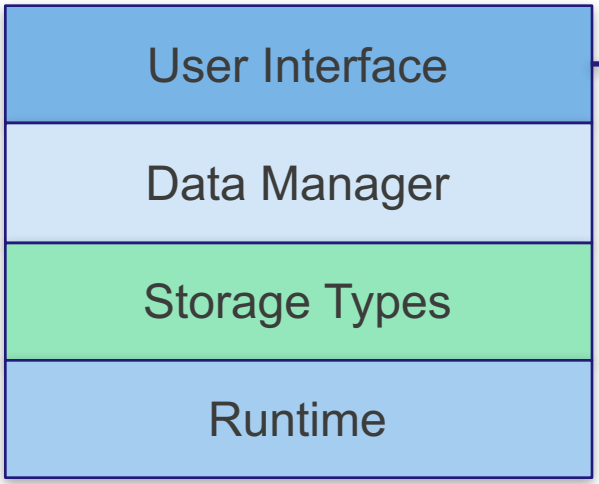
Requirements, mesh data structures, design

Legion Backend

What is the FleCSI data interface?

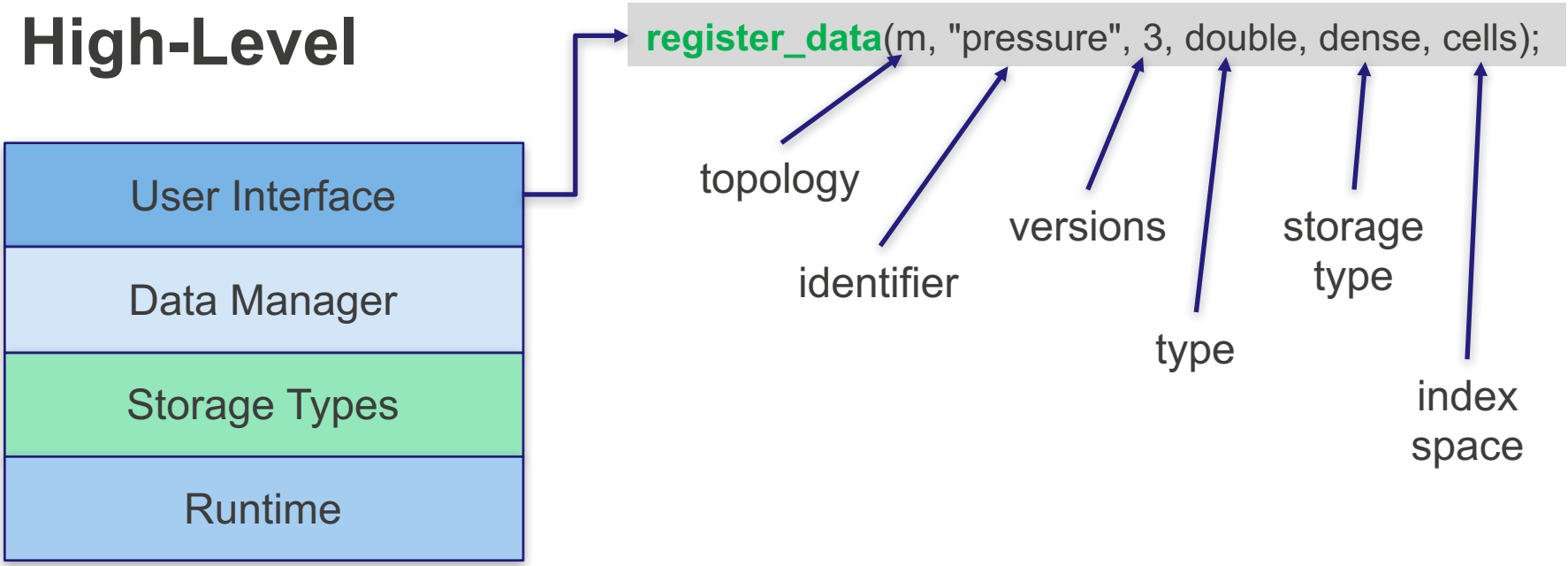
High-Level

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



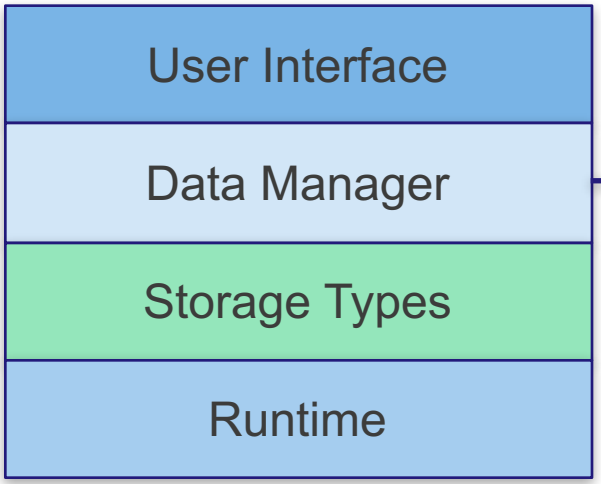
What is the FleCSI data interface?

High-Level



What is the FleCSI data interface?

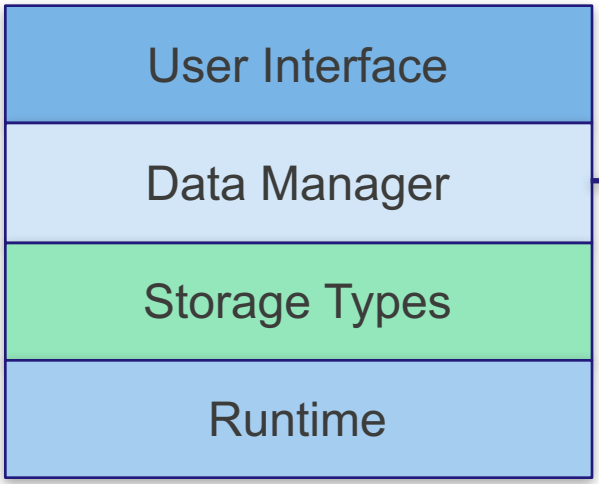
Specialization




```
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)
```

What is the FleCSI data interface?

Specialization



storage type

type

```
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)
```

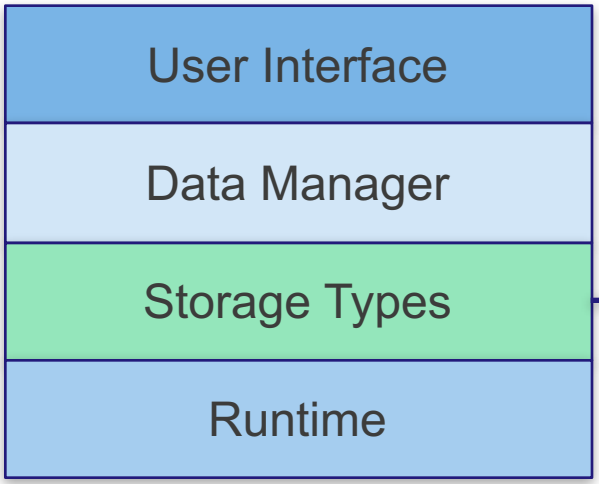
topology

identifier

versions

What is the FleCSI data interface?

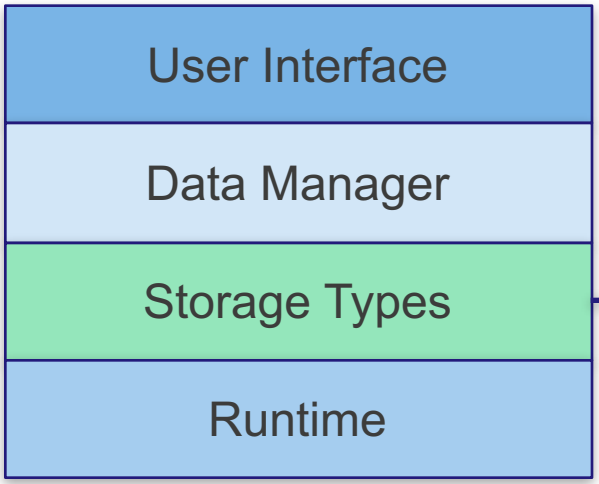
Low-Level




```
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)
```

What is the FleCSI data interface?

Low-Level

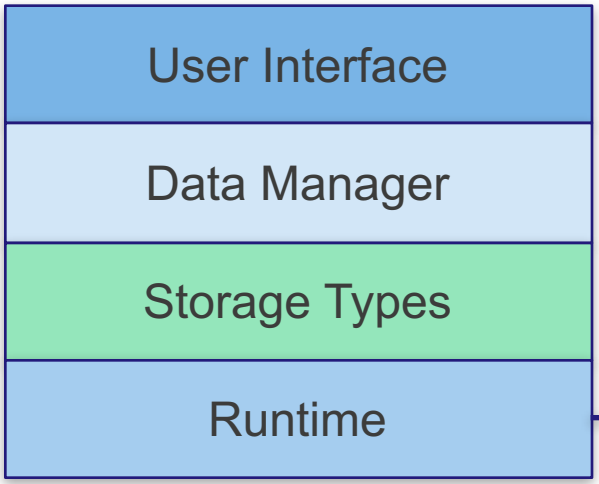


Mapped to specific storage type

```
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)
```

What is the FleCSI data interface?

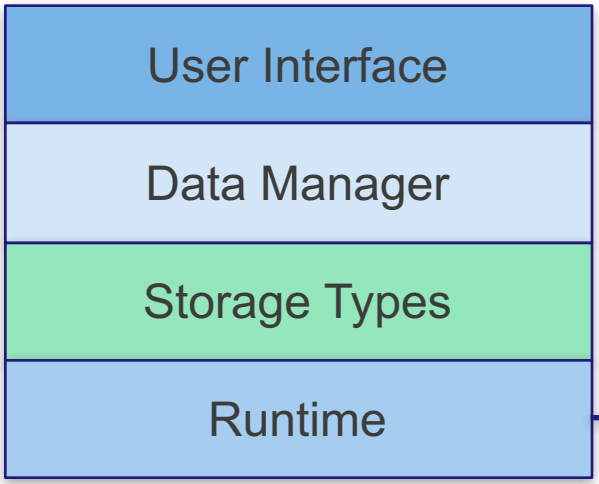
Backend




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

What is the FleCSI data interface?

Backend

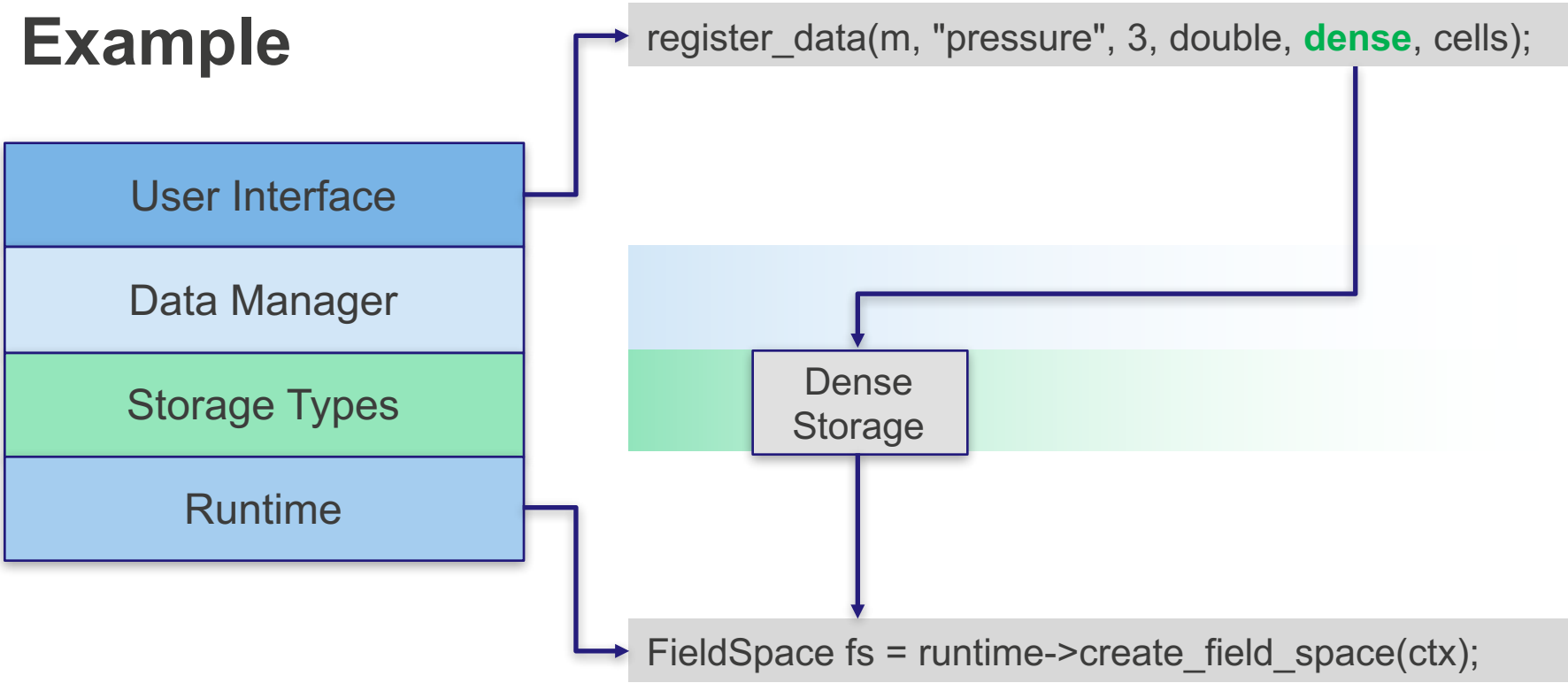


Storage type uses Legion runtime to create appropriate field space(s)

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

What is the FleCSI data interface?

Example

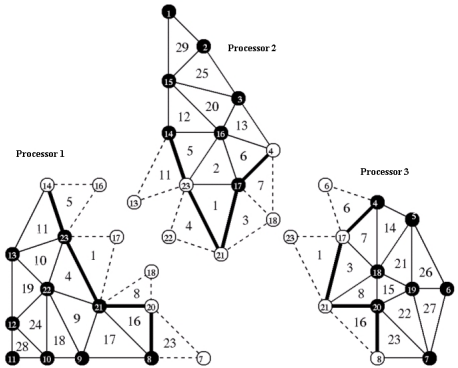


What storage types do we support?

- **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

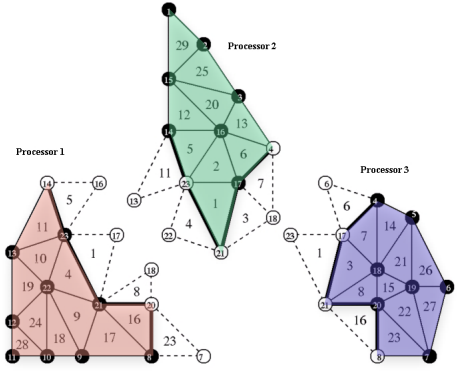
Distributed-Memory Partitioning

How does FleCSI handle distributed memory?



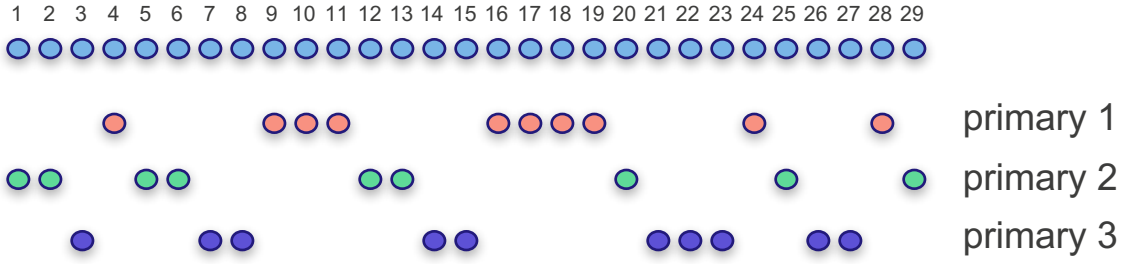
Mesh

How does FleCSI handle distributed memory?

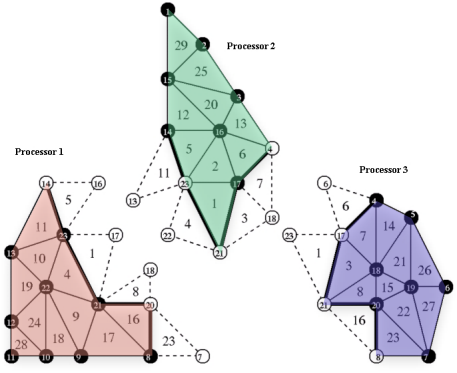


Mesh

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

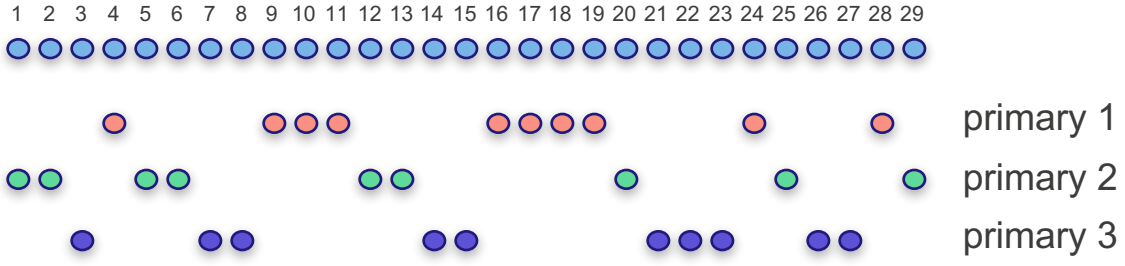


How does FleCSI handle distributed memory?



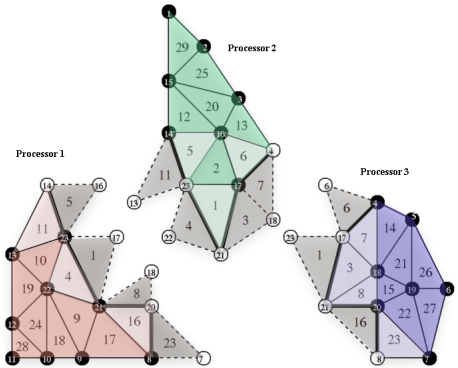
Mesh

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



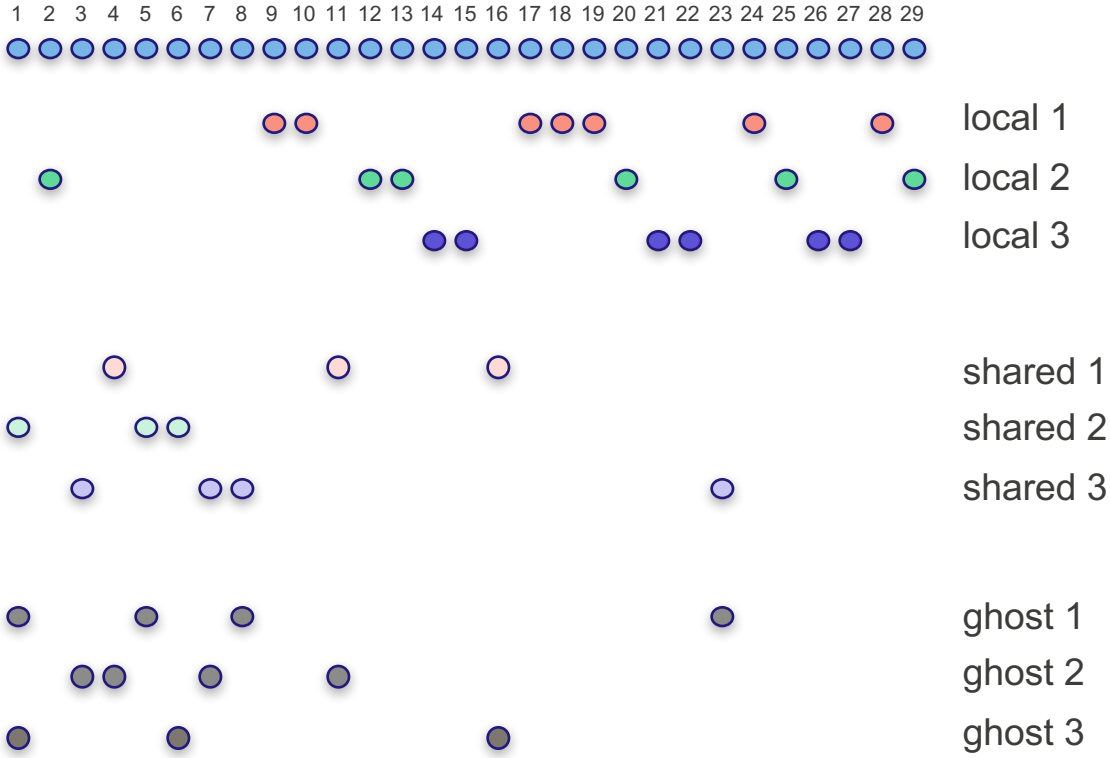
The primary partitioning splits the topology into contiguous sets of cells.

How does FleCSI handle distributed memory?

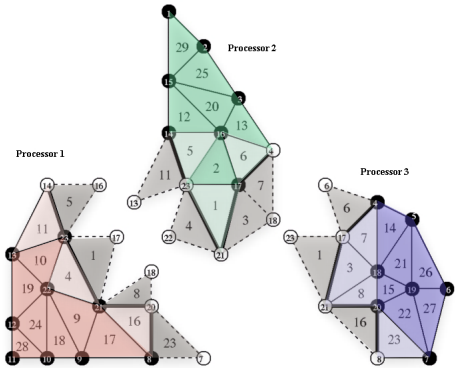


Mesh

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

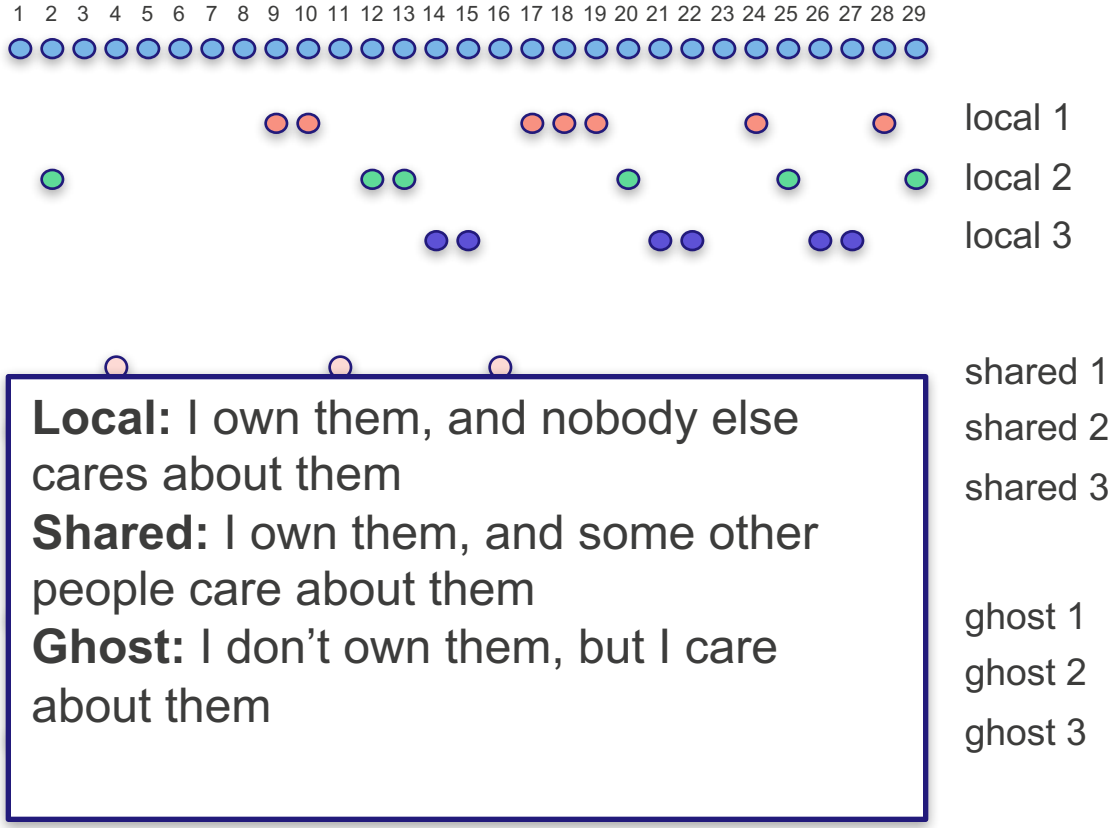


How does FleCSI handle distributed memory?

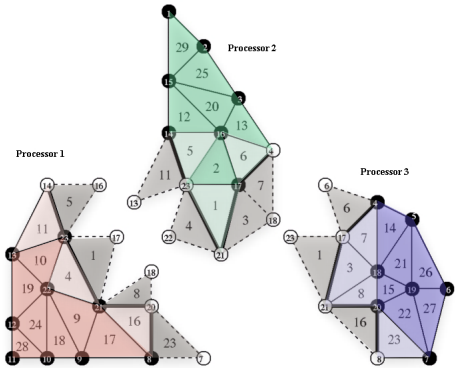


Mesh

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

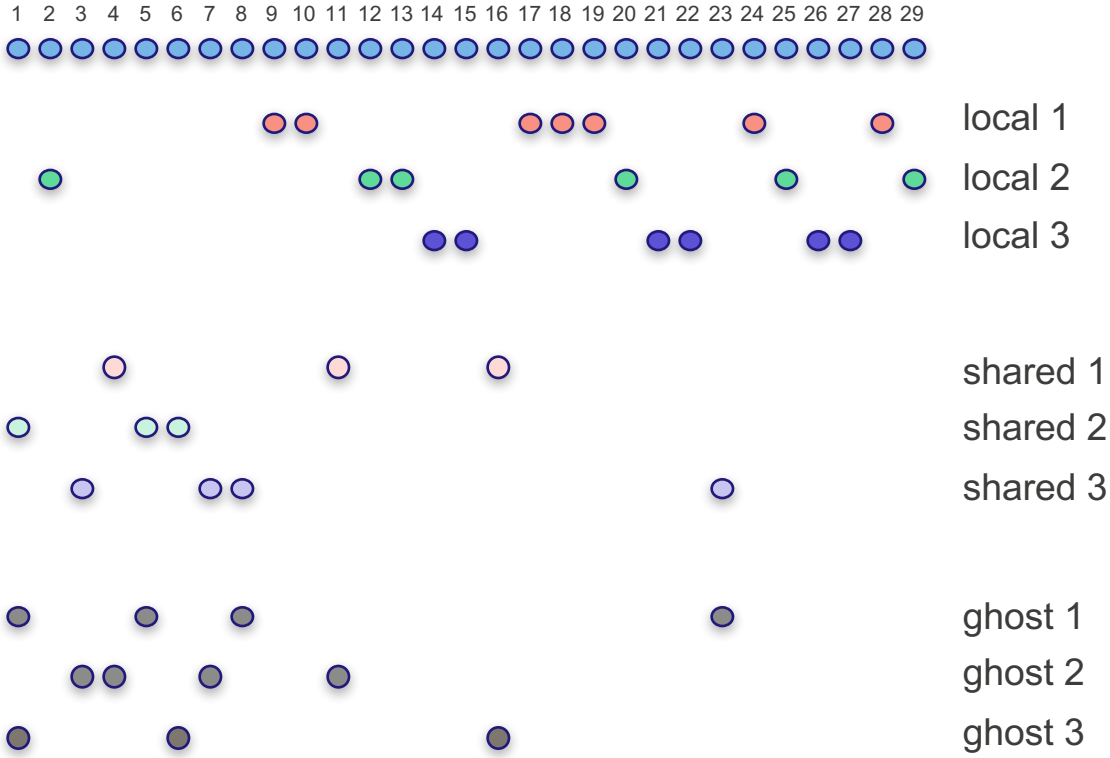


How does FleCSI handle distributed memory?

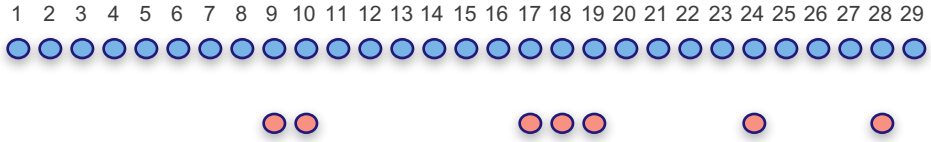


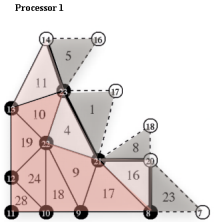
Mesh

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



How does FleCSI handle distributed memory?

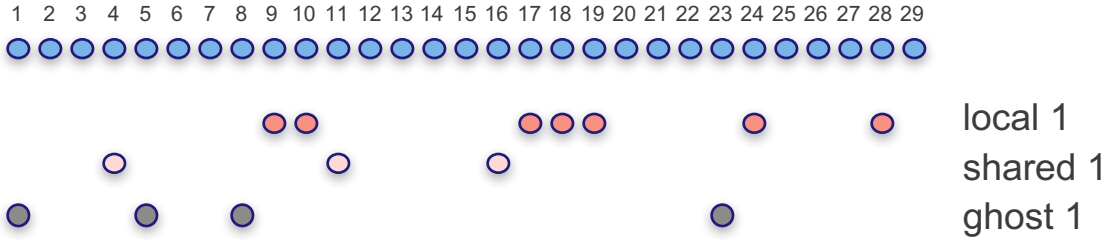


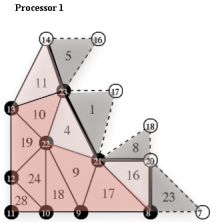


Mesh

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

How does FleCSI handle distributed memory?



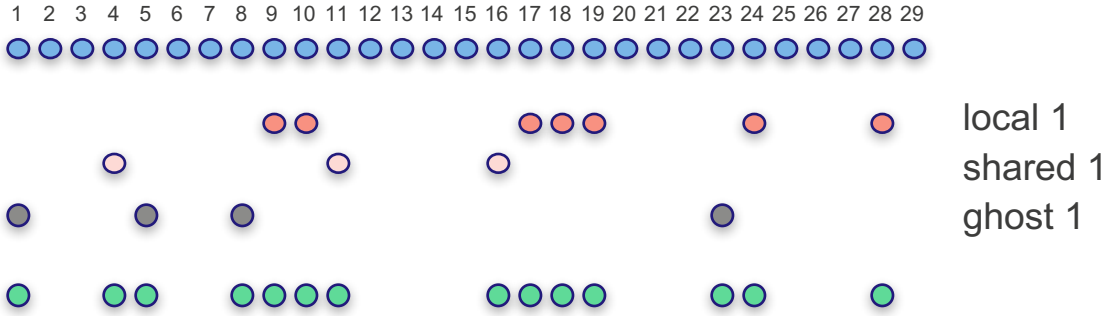


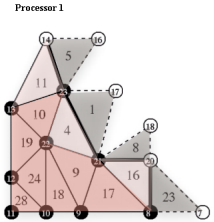
Mesh

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
```

How does FleCSI handle distributed memory?



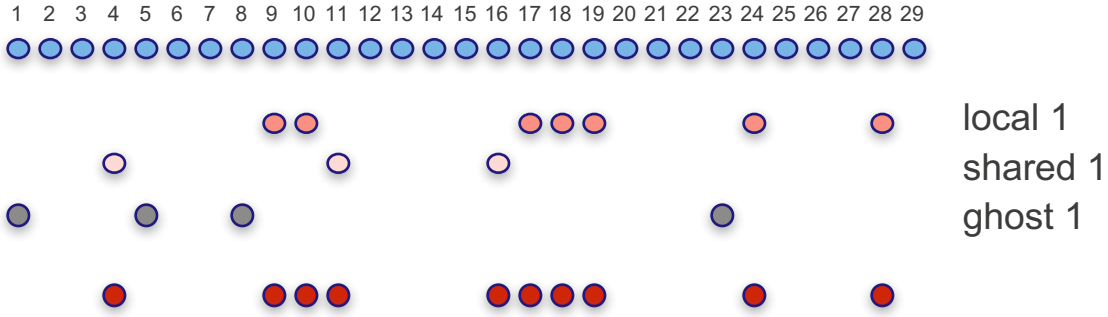


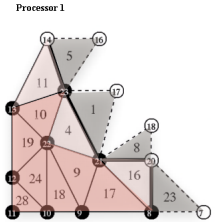
Mesh

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

How does FleCSI handle distributed memory?



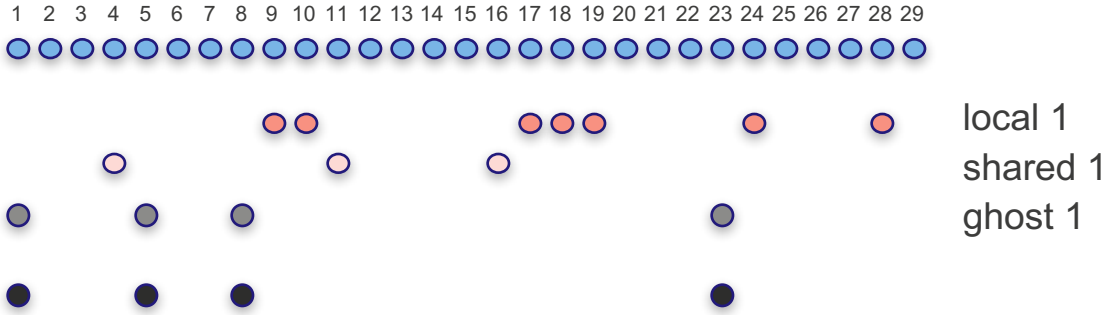


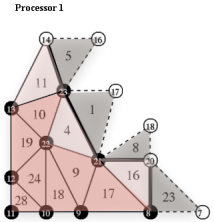
Mesh

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

How does FleCSI handle distributed memory?



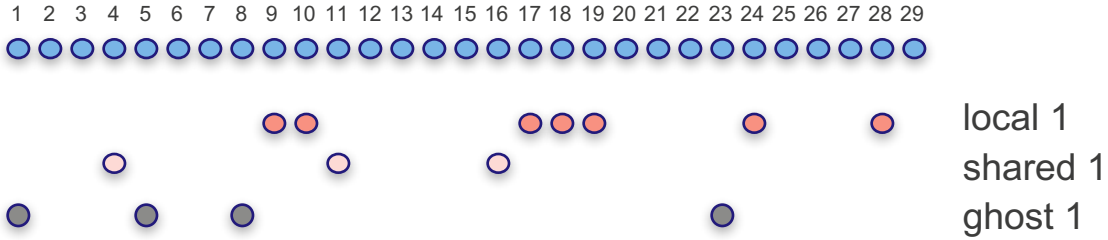


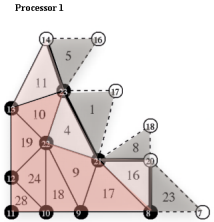
Mesh

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
```

How does FleCSI handle distributed memory?



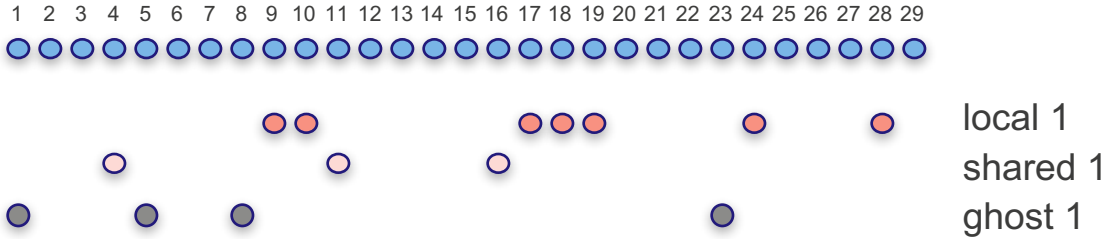


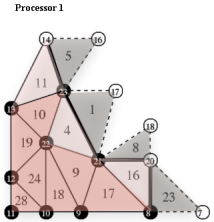
Mesh

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

How does FleCSI use Legion for dependency updates?





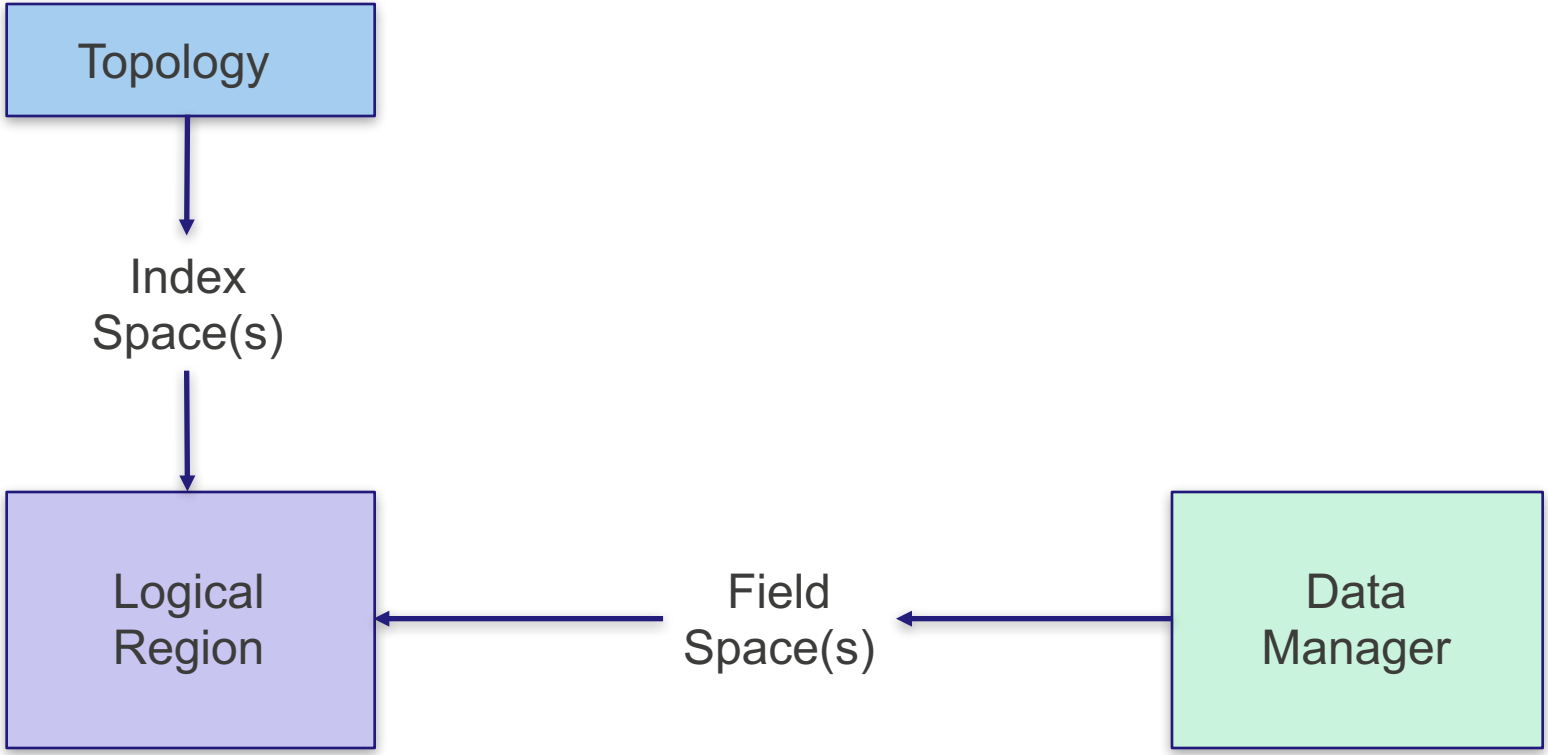
topology type:
partition_t * partition

partition[0]:
IndexSpace local 1
IndexSpace shared 1
IndexSpace ghost 1

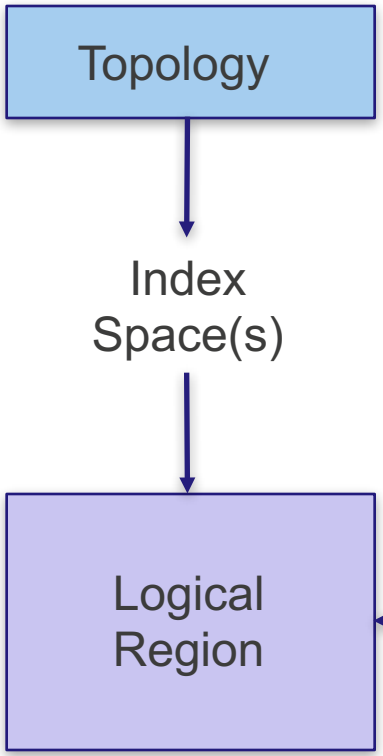
Mesh

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

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



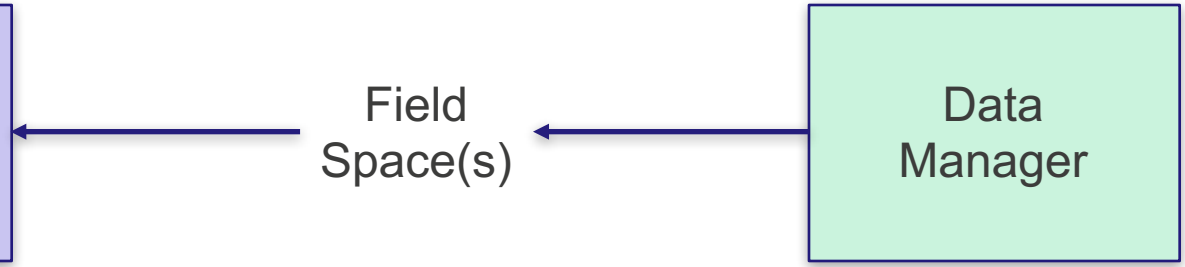
How do the topology and data models work together for Legion?



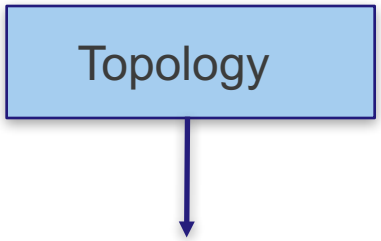
When a user registers data, the data manager and topology cooperate to create several logical regions…

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

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



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



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(eos, r, e);
} // scope
```

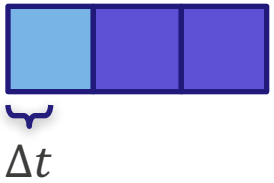
Sparse Data Representations

Managing sparse data representations presents some challenges…

material 1

material 2

Initial Distribution



Matrix Representation

	m1	**m2**
c0	v1	
c1		v2
c2		v3

Compressed Storage

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

Managing sparse data representations presents some challenges…

material 1

material 2

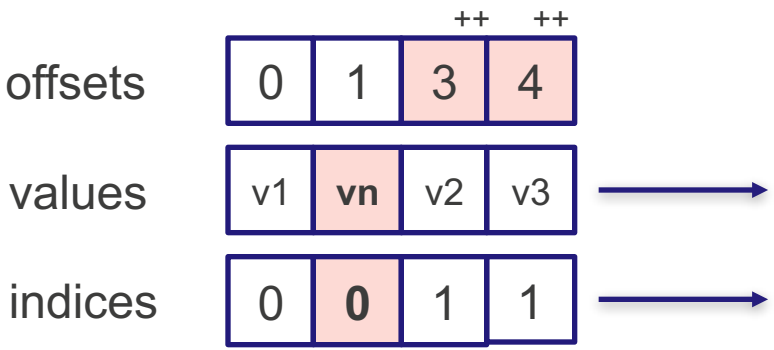
Evolved Distribution

Δt → linear advection

Matrix Representation

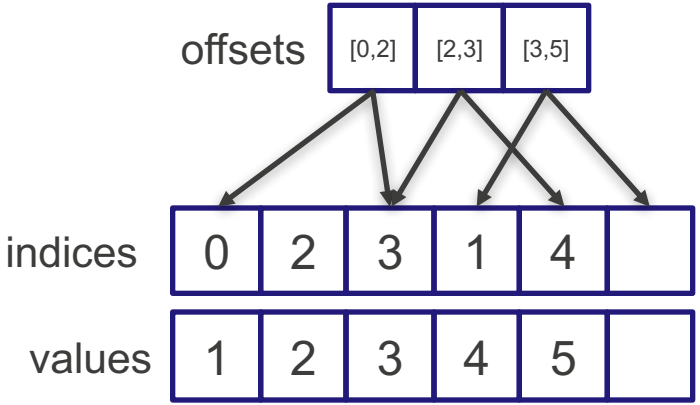
	m1	m2
c0	v1	
c1	vn	v2
c2		v3

Compressed Storage



Mutated Structure!!!

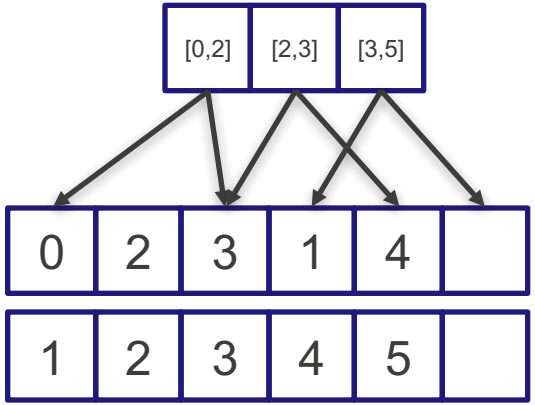
How does FleCSI handle sparse data?



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

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

How does FleCSI handle sparse data?



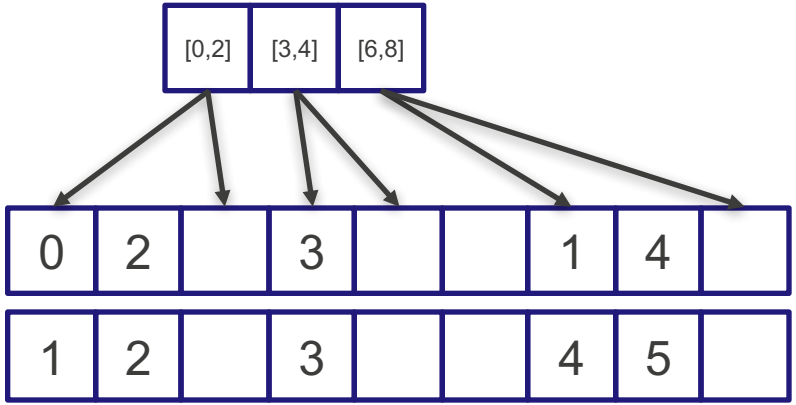
$$A = \begin{bmatrix} 1 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \\ 0 & 4 & 0 & 0 & 5 \end{bmatrix}$$

Code block to mutate sparse structure →

```
{
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

How does FleCSI handle sparse data?



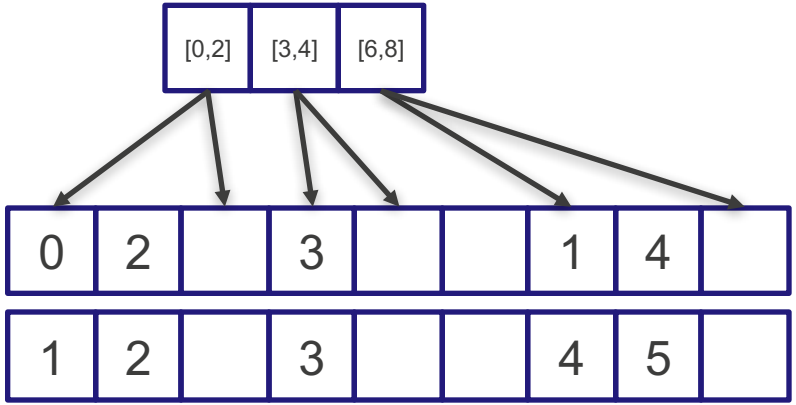
$$A = \begin{bmatrix} 1 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \\ 0 & 4 & 0 & 0 & 5 \end{bmatrix}$$

Constructor inserts space for new values →

```
{
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

How does FleCSI handle sparse data?



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

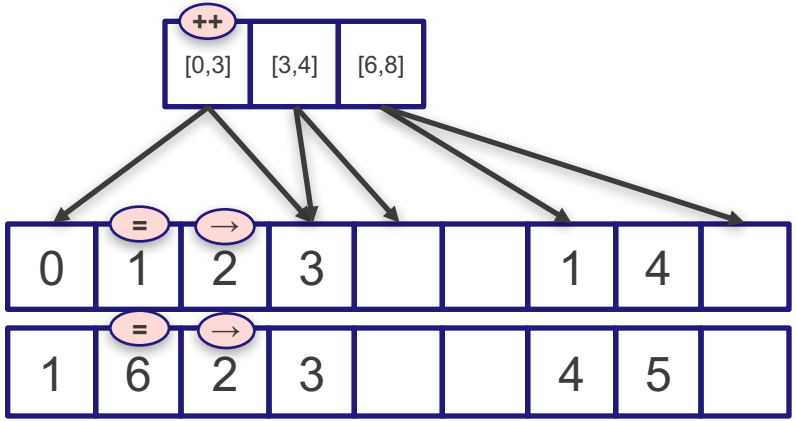
User specifies maximum total slots

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

How does FleCSI handle sparse data?



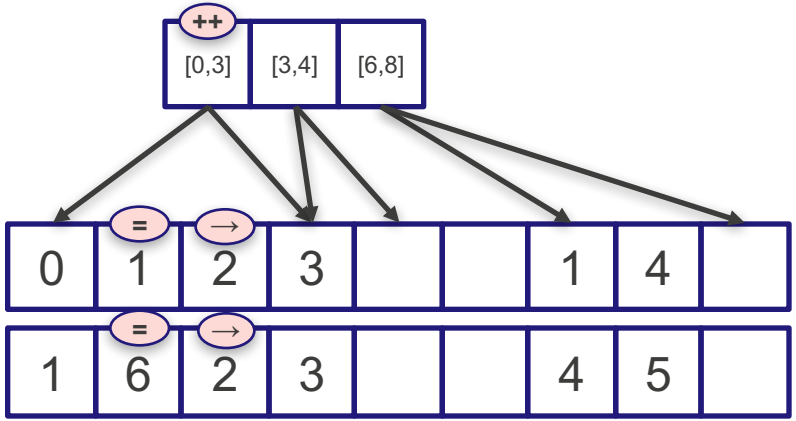
$$A = \begin{bmatrix} 1 & 6 & 2 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \\ 0 & 4 & 0 & 0 & 5 \end{bmatrix}$$

Intuitive interface to set non-zeros →

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

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

How does FleCSI handle sparse data?



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

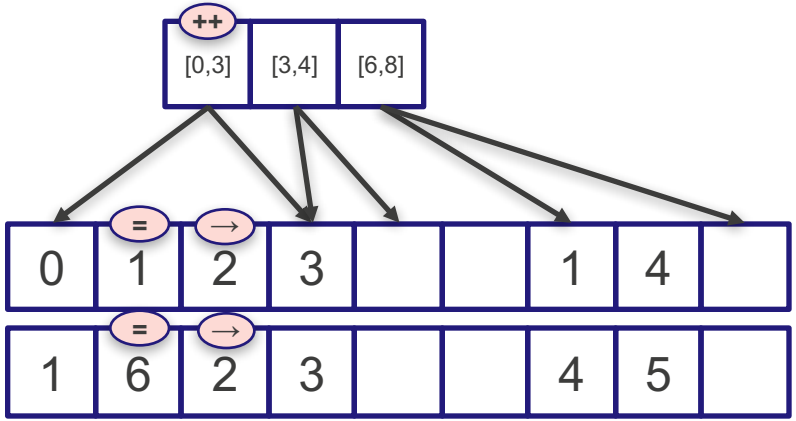
Intuitive interface to set non-zeros →

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

```
{
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

How does FleCSI handle sparse data?



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

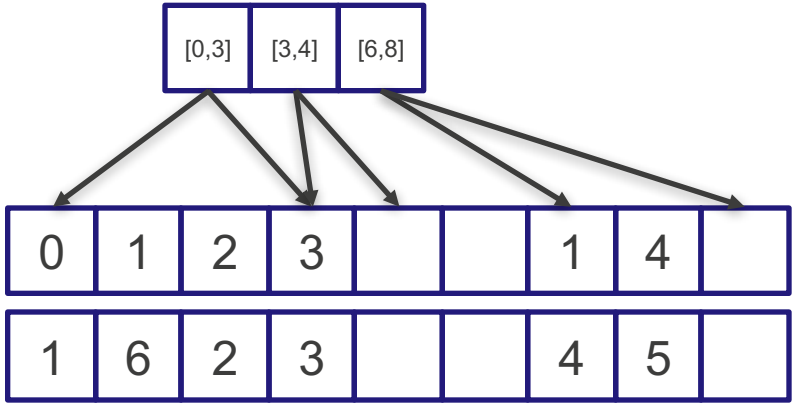
Column order is preserved, single-slot shift, only slot end is incremented

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

How does FleCSI handle sparse data?



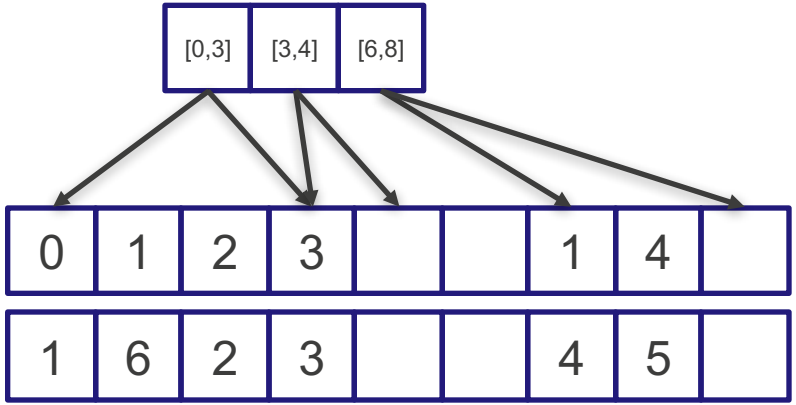
$$A = \begin{bmatrix} 1 & 6 & 2 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \\ 0 & 4 & 0 & 0 & 5 \end{bmatrix}$$

What if we need more than 3 non-zeros? →

```
{
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

How does FleCSI handle sparse data?



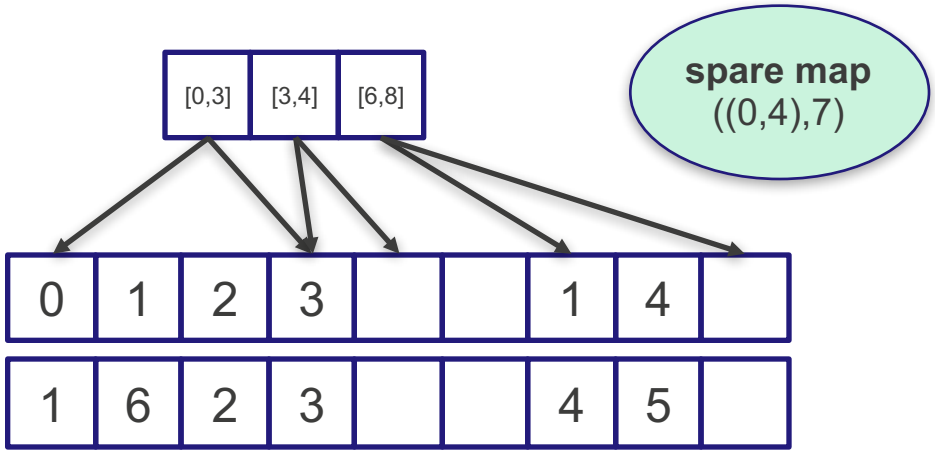
$$A = \begin{bmatrix} 1 & 6 & 2 & 7 & 0 \\ 0 & 0 & 0 & 3 & 0 \\ 0 & 4 & 0 & 0 & 5 \end{bmatrix}$$

What if we need more than 3 non-zeros? →

```
{
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

How does FleCSI handle sparse data?



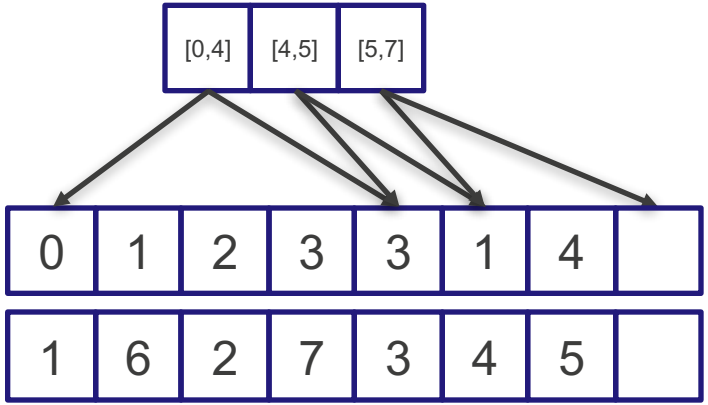
$$A = \begin{bmatrix} 1 & 6 & 2 & 7 & 0 \\ 0 & 0 & 0 & 3 & 0 \\ 0 & 4 & 0 & 0 & 5 \end{bmatrix}$$

What if we need more than 3 non-zeros? →

```
{
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

How does FleCSI handle sparse data?



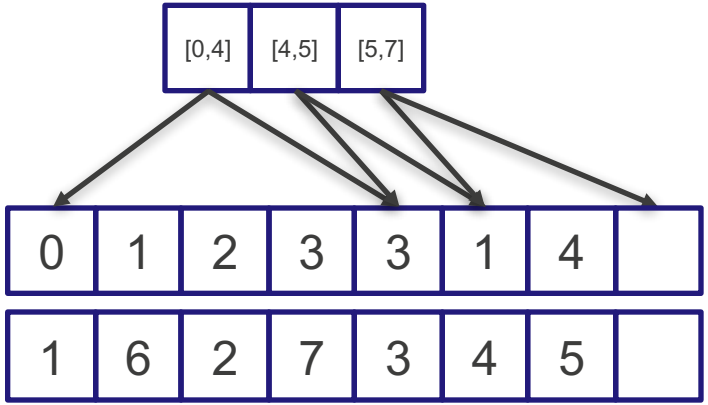
$$A = \begin{bmatrix} 1 & 6 & 2 & 7 & 0 \\ 0 & 0 & 0 & 3 & 0 \\ 0 & 4 & 0 & 0 & 5 \end{bmatrix}$$

Destructor recompresses data →

```
{
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

How does FleCSI handle sparse data?

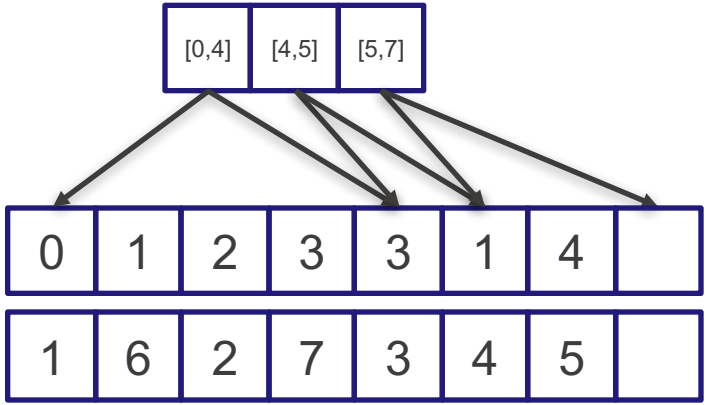


$$A = \begin{bmatrix} 1 & 6 & 2 & 7 & 0 \\ 0 & 0 & 0 & 3 & 0 \\ 0 & 4 & 0 & 0 & 5 \end{bmatrix}$$

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

How does FleCSI handle sparse data?

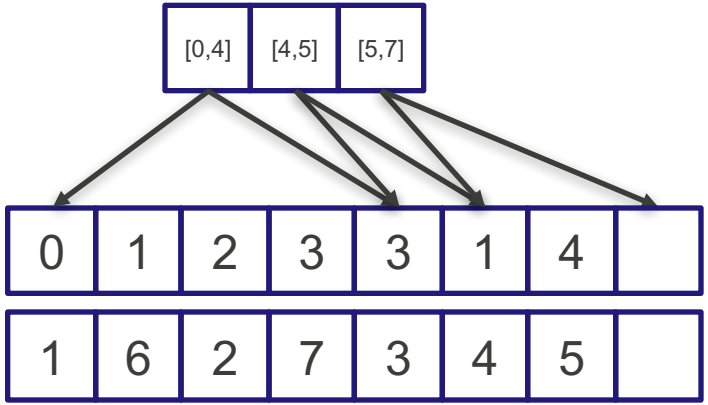


$$A = \begin{bmatrix} 1 & 6 & 2 & 7 & 0 \\ 0 & 0 & 0 & 3 & 0 \\ 0 & 4 & 0 & 0 & 5 \end{bmatrix}$$

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

How does FleCSI handle sparse data?

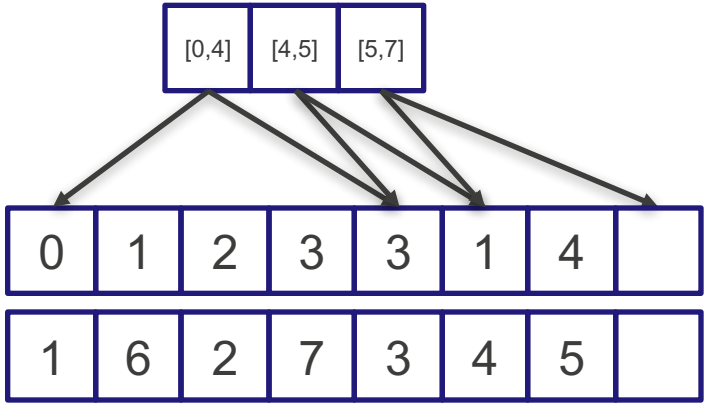


$$A = \begin{bmatrix} 1 & 6 & 2 & 7 & 0 \\ 0 & 0 & 0 & 3 & 0 \\ 0 & 4 & 0 & 0 & 5 \end{bmatrix}$$

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

How does FleCSI handle sparse data?

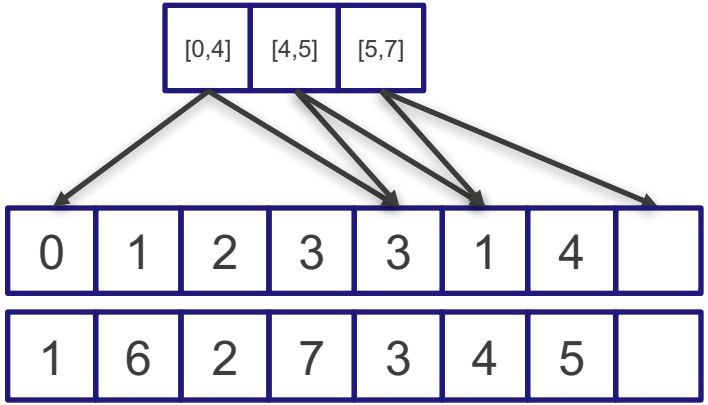


$$A = \begin{bmatrix} 1 & 6 & 2 & 7 & 0 \\ 0 & 0 & 0 & 3 & 0 \\ 0 & 4 & 0 & 0 & 5 \end{bmatrix}$$

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

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

Where are we at?

- **Sparse data type is implemented and in *friendly user* mode**
  - Serial implementation done
  - Working on Legion backend
- **Most other storage types have been implemented in serial**
  - Legion backend implementation is straightforward
- **Legion task implementation is being finalized**
  - Execution model for tasks is in place, i.e., *friendly user* mode
  - Data model structure is in place, but requires more work
- **ParMETIS partitioning is in progress…**
  - Primary partitioning in place
  - Working on dependency closure abstractions
- **MPI+Legion interoperability**
  - Working prototype is being integrated into FleCSI

MPI–Legion Interoperability

Initialization begins with MPI runtime…

Initialization begins with MPI runtime…

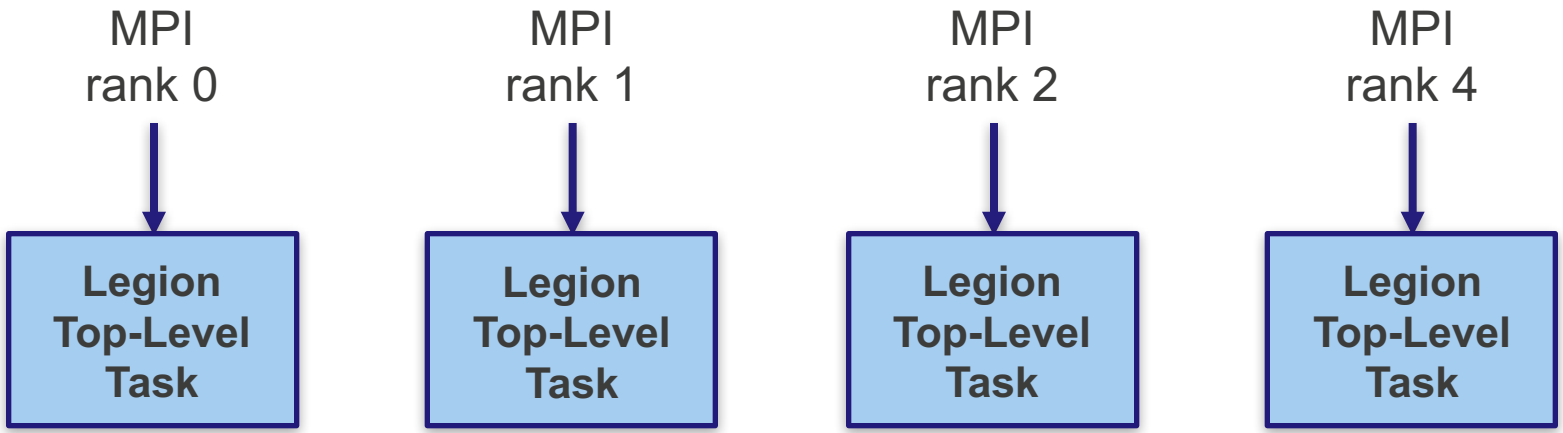
MPI rank 0

MPI rank 1

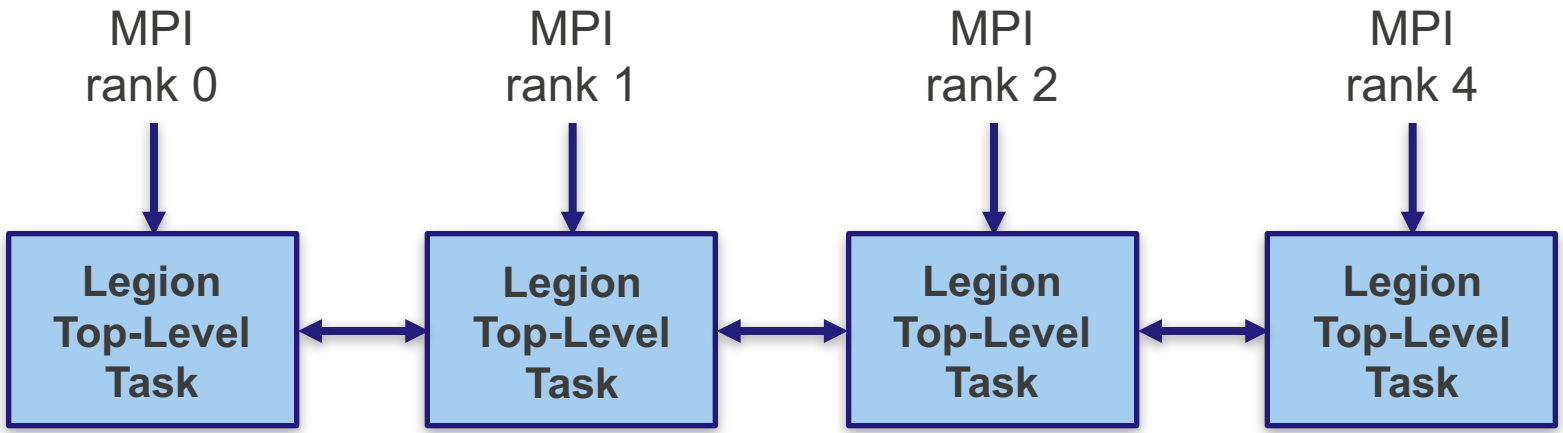
MPI rank 2

MPI rank 4

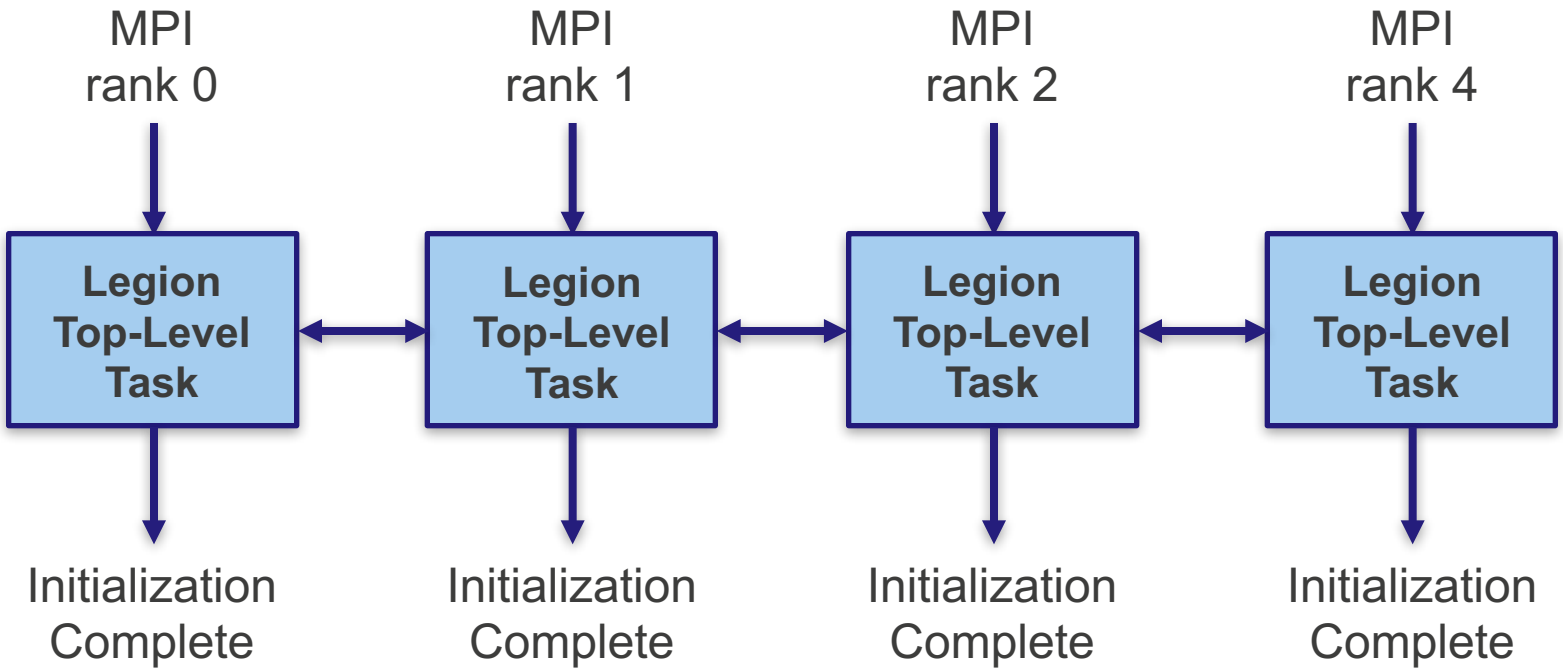
Each MPI rank starts the top-level Legion task…



Legion uses PGAS–MPI interoperability mode to synchronize tasks…



Once the Legion runtime has initialized, a SPMD task is launched and the system is ready…



Subsequent calls to the MPI runtime can be made from the SPMD tasks…

driver.h (Legion Runtime)

```
#include “mpi_tasks.h”

int driver(int argc, char ** argv) {

   execute_mpi(mpi_task_1, data);

   execute_task(legion_task_1, data);

   execute_mpi(mpi_task_2, data);
}
```

mpi_tasks.h (MPI Runtime)

```
int mpi_task_1(double * data) {
   int rank;
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   for(i=start_range; i<end_range; ++i) {
      data[i] = math();
   } // for
}

int mpi_task_2(double * data) {

   // algorithm logic

}
```

Subsequent calls to the MPI runtime can be made from the SPMD tasks…

driver.h (Legion Runtime)

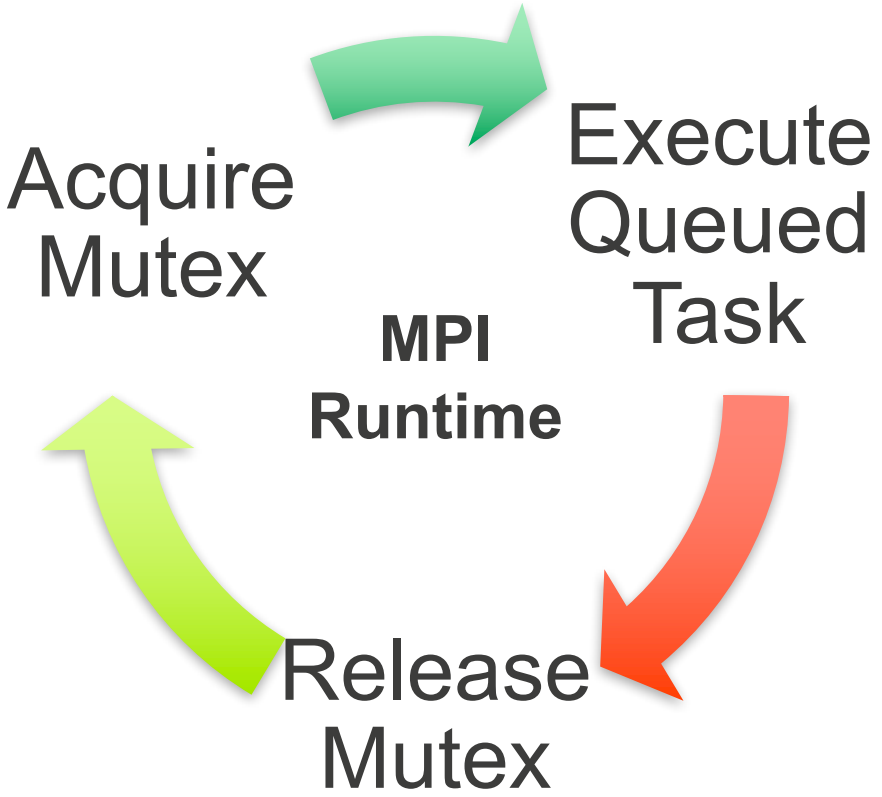
```
#include “mpi_tasks.h”

int driver(int argc, char ** argv) {

   execute_mpi(mpi_task_1, data);

   execute_task(legion_task_1, data);

   execute_mpi(mpi_task_2, data);
}
```

mpi_tasks.h (MPI Runtime)

```
int mpi_task_1(double * data) {
   int rank;
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   for(i=start_range; i<end_range; ++i) {
      data[i] = math();
   } // for
}

int mpi_task_2(double * data) {

   // algorithm logic

}
```

Each runtime has exclusive execution (bulk synchronous between runtimes)

How do the runtimes manage the interaction?

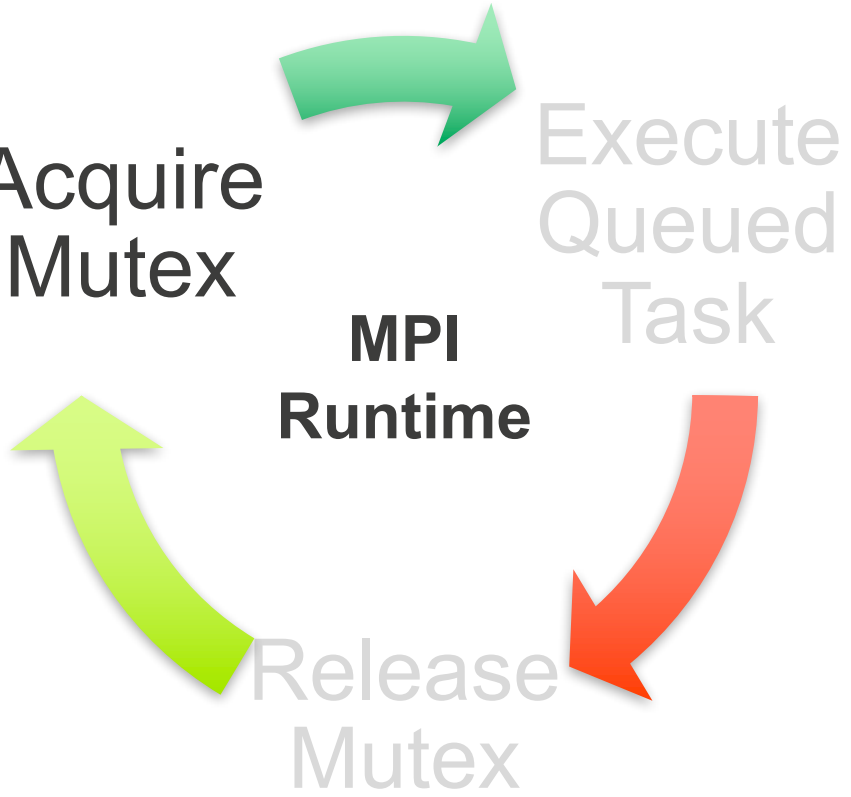


How do the runtimes manage the interaction?

execute_mpi(mpi_task)

How do the runtimes manage the interaction?

execute_mpi(mpi_task) →



How do the runtimes manage the interaction?

How do the runtimes manage the interaction?

How do the runtimes manage the interaction?