ParFUM Tutorial

Presented by:
Terry Wilmarth

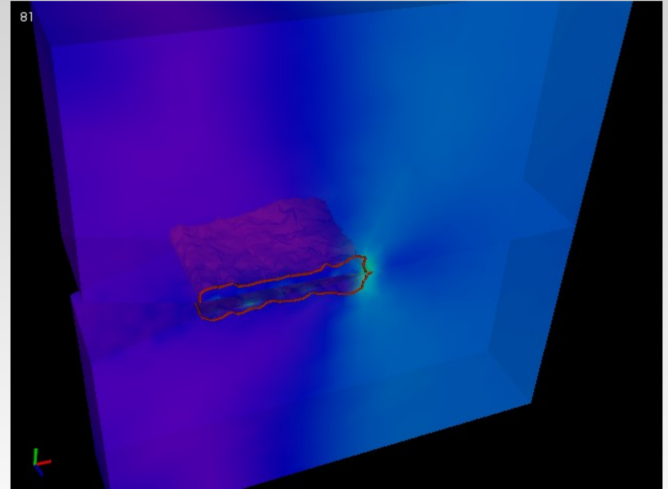
Parallel Programming Laboratory
and
Center for Simulation of Advanced Rockets
University of Illinois at Urbana-Champaign

Why use ParFUM?

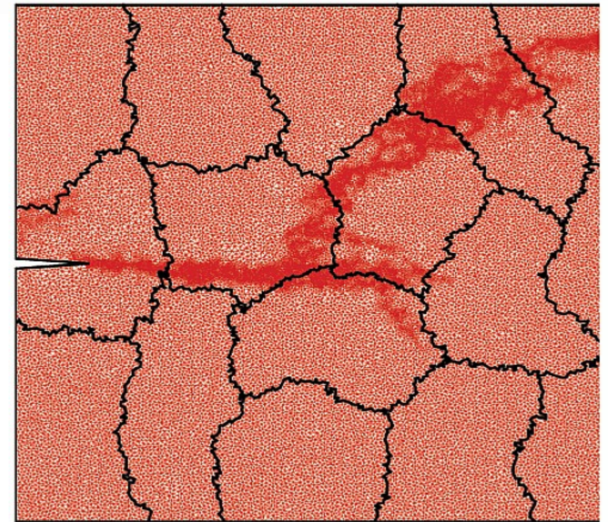
- Excellent performance for irregular (and regular) problems (load balancing, etc.)
- Ease of use:
  - Fast conversion of serial codes to parallel
  - Fast conversion of MPI codes to benefit from load balancing and other features
  - FORTRAN, C/C++ bindings
- Extremely portable, freely available
- Development is collaboration-driven

Features

- Flexible “ghost” layer creation
- Parallel partitioning
- Simple collective calls to update mesh entities on partition boundaries
- Mesh adaptivity
- Supports multiple element types and mixed elements
- Topological adjacencies
- Solution transfer
- Collision detection



3D Fractography in ParFUM



Dynamic Fracture in ParFUM

Tutorial Outline

- ParFUM philosophy and terminology
- Program structure
- Data ownership
- Ghost creation and synchronization
- Topological relationships
- Mesh adaptivity
- Obtaining, building and running ParFUM programs
- ParFUM Libraries
  - Solution Transfer
  - Collision Detection
  - Visualization
- A simple example program

ParFUM Philosophy

- Designed to be flexible and general
- Provides a few general operations common to a wide variety of applications
- Makes as few assumptions about the problem as possible
  - e.g. no restriction on number of spatial dimensions: node locations are just another type of node data

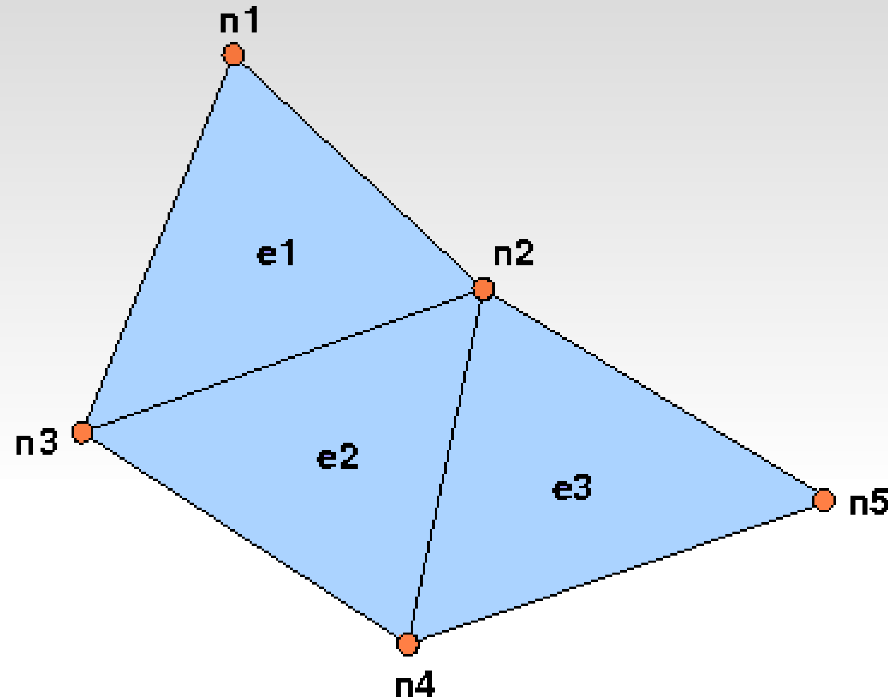
ParFUM Terminology

- ParFUM manages *mesh entities*
- Minimally, these are *elements* and *nodes*
- Element (aka *cell*): a portion of a problem domain, e.g. triangle, tetrahedron, hexahedron, etc.
- Node: a point in the problem domain, often a vertex of one or more elements
- Mesh: collection of nodes and elements

ParFUM Terminology, cont'd

- Elements have connectivity, a list of nodes that make up the element
- A mesh is decomposed into multiple partitions for parallel processing. Typically, there is one partition per processor, but in the case of ParFUM, there may be many partitions per processor
- Mesh entities are given local numbers on each partition

ParFUM Terminology, cont'd

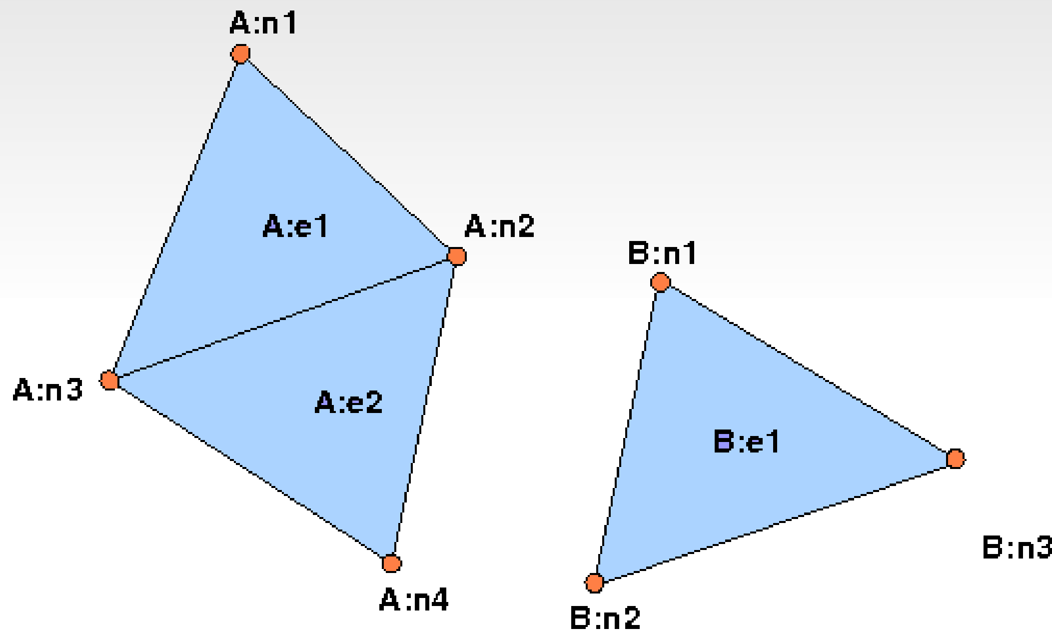


Conn:
e1: n1, n2, n3
e2: n2, n4, n3
e3: n5, n4, n2

- Example: material dynamics time-loop:

```
begin time-loop
  element loop – Element deformation applies forces to surrounding nodes
  node loop – Forces and boundary conditions change node positions
end time-loop
```

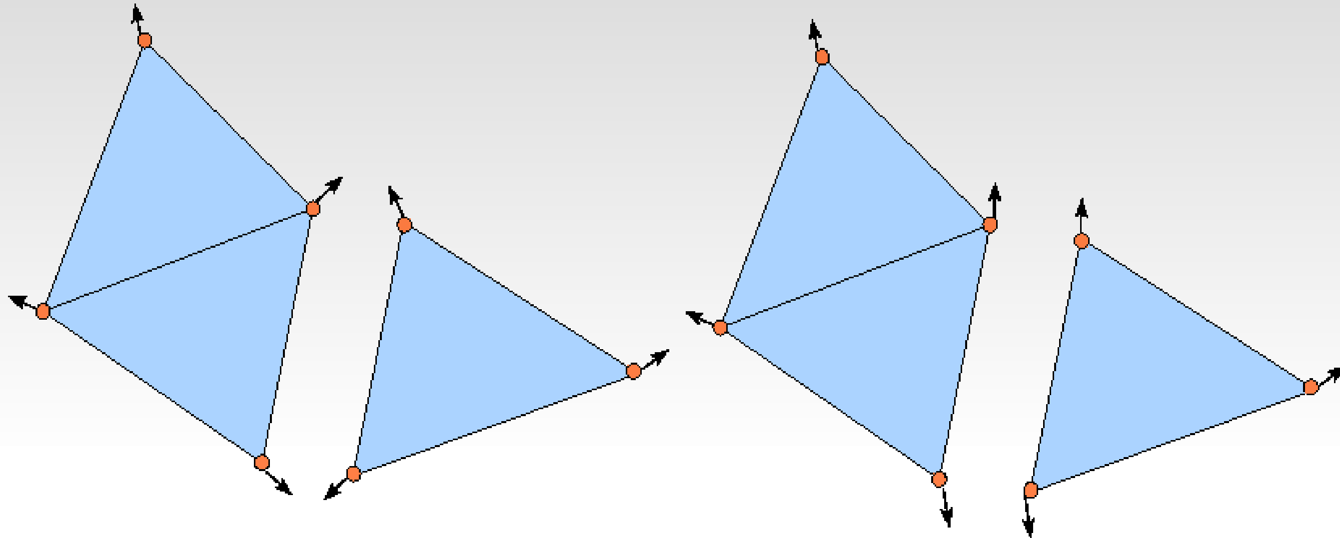
ParFUM Terminology, cont'd



Conn for partition A:
e1: n1, n2, n3
e2: n2, n4, n3

Conn for partition B:
e1: n1, n2, n3

ParFUM Terminology, cont'd



- Example: hydrostatic forces applied across partitions

```
begin partition-time-loop
  element loop – Element deformation applies forces to surrounding nodes
  update forces on shared nodes
  node loop – Forces and boundary conditions change node positions
end partition-time-loop
```

ParFUM Program Structure

- Traditional:

```
subroutine init
  read serial mesh and configuration data
end subroutine
// after init, ParFUM automatically partitions the mesh
subroutine driver
  get local mesh partition
  begin time-loop
    solver computations
    communication to update entities along partition boundaries
    more solver computations
  end time-loop
end subroutine
```

- `init` called on virtual processor (VP) 0; `driver` called on all VPs

ParFUM Program Structure

- MPI-style:

```
main program
  MPI_Init();
  FEM_Init(MPI_COMM_WORLD);
  if (I am master processor)
    read mesh
  partition mesh
  time-loop
    solver computations
    communication to update entities along partition boundaries
    more solver computations
  end time-loop
end main program
```

  - User needs to partition explicitly
  - User needs to create mesh and set as default

Data Ownership

- Two modes: ParFUM-owned, user-owned
- ParFUM-owned:

```
void FEM_Mesh_data(int mesh, int entity, int attribute, void *data,
  int first, int length, int datatype, int width);
```

- Only one routine for getting/setting data on a mesh entity's attributes
  - To set:

```
void FEM_Mesh_become_set(int mesh);
```

  - To get:

```
void FEM_Mesh_become_get(int mesh);
```

Data Ownership, cont'd

- Example: setting triangle element connectivity

```
FEM_Mesh_become_set(mesh);
int conn = new int[3*nElems];
// Fill out conn...
FEM_Mesh_data(mesh,          // the mesh to add the elements to
              FEM_ELEM+1,    // this is an attribute of element type 1
              FEM_CONN,      // this is the connectivity attribute
              conn,          // this is the data
              0,             // data is indexed from 0
              nElems,        // this is the length of the data
              FEM_INDEX_0,   // this is the type of the data
              3);            // this is the width of the data

! F90 version:
CALL FEM_Mesh_become_set(mesh);
ALLOCATE(conn(3,nElems))
! Fill out conn...
CALL FEM_Mesh_data(mesh, FEM_ELEM+1, FEM_CONN, conn, 1, nElems,
  FEM_INDEX_1, 3)
```

Data Ownership, cont'd

- Example: adding material property data to nodes

```
FEM_Mesh_become_set(mesh);
double matProp = new double[2*nNodes];
// Fill out matProp...
FEM_Mesh_data(mesh,          // the mesh to add the node attribute to
              FEM_NODE,      // this is an attribute of nodes
              FEM_DATA_13,   // this is a user attribute with tag 13
              matProp,       // this is the data
              0,             // data is indexed from 0
              nNodes,        // this is the length of the data
              FEM_DOUBLE,    // this is the type of the data
              2);            // there are two matProps per node

! F90 version:
CALL FEM_Mesh_become_set(mesh);
ALLOCATE(matProp(2,nNodes))
! Fill out matProp...
CALL FEM_Mesh_data(mesh, FEM_NODE, FEM_DATA+13, matProp, 1, nNodes,
  FEM_DOUBLE, 2)
```

Data Ownership, cont'd

- Example: getting 3D node coordinate data

```
FEM_Mesh_become_get(mesh);
double *nodeCoords = new double[3*nNodes];
FEM_Mesh_data(mesh,          // the mesh to get the node coordinates from
              FEM_NODE,      // this is an attribute of nodes
              FEM_COORD,     // this is the coordinate attribute
              nodeCoords,    // this is where the data is put
              0,             // data is indexed from 0
              nNodes,        // this is the length of the data
              FEM_DOUBLE,    // this is the type of coordinate data
              3);            // there are three doubles per node coordinate
// use nodeCoords now

! F90 version:
CALL FEM_Mesh_become_get(mesh);
ALLOCATE(nodeCoords(3,nNodes))
CALL FEM_Mesh_data(mesh, FEM_NODE, FEM_COORD, nodeCoords, 1, nNodes,
  FEM_DOUBLE, 3)
! use nodeCoords now
```

Data Ownership, cont'd

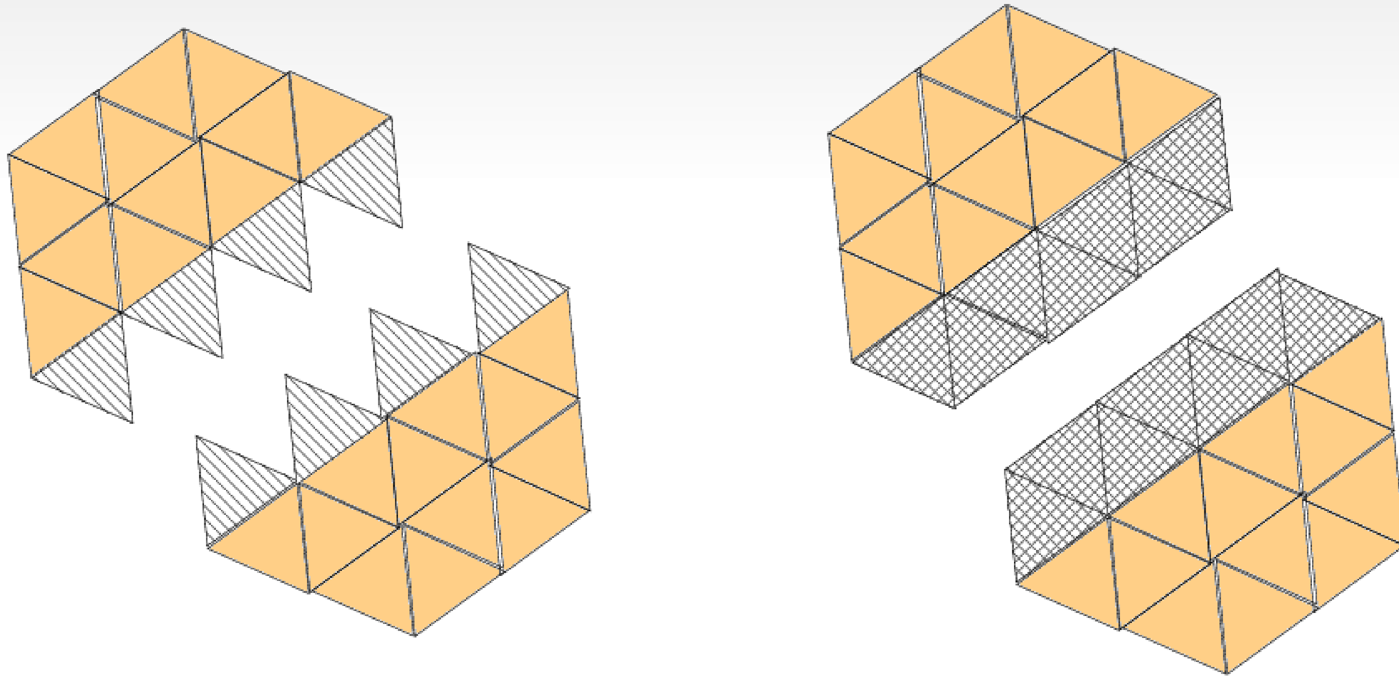
- User owns data:

```
void FEM_Register_entity(int mesh, int entity, void *data, int length,
  int max, FEM_Mesh_alloc_fn fn);
void FEM_Register_array(int mesh, int entity, int attribute, void *data,
  int datatype, int width);
```

  - Add entities with an initial length and pass a resize function
  - Add attributes subsequently, length is taken from the entity length when it was initially added

Ghost Creation

- Example: single layer of edge- and node-neighbors



Ghost Creation

```
void FEM_Add_ghost_layer(int nodesPerFace, int doAddNodes);
```

- *`nodesPerFace`* defines "neighboring" relationship
  - e.g. nodesPerFace=1, include all elements from neighboring partitions that share at least 1 node with an element of the local partition
  - e.g. nodePerFace=3, include all elements that share 3 nodes (such as sharing a tetrahedral face)
- *`doAddNodes`* is a flag for the inclusion of nodes in the ghost layer, i.e. includes all the nodes on the ghost elements that are not present locally

Ghost Creation

```
void FEM_Add_ghost_elem(int elemType, int facesPerElem,
  const int*elemToFace);
```

- Add elements of *`elemType`* to current ghost layer
- *`facesPerElem`* and *`elemToFace`* specify which faces of the elements are to be considered (where a face was defined by *`nodesPerFace`* for the current layer)
- *`elemToFace`* enumerates the possible faces
  - Example: Consider a quad on which you want only two sides (edges) to be considered for determining the neighboring status

```
const static int myElemToFace[] = {0,1,  2,3};
```

Ghost Creation

- Can have multiple ghost layers, with mixed elements, each defined differently
- Example: In a mixed mesh of triangles and quads, we want two ghost layers, one with all node-neighboring elements and the second only with edge-neighboring quads. Only the first layer needs node ghosts.

```
FEM_Add_ghost_layer(1, 1);
const static int triElemToFace[] = {0, 1, 2};
FEM_Add_ghost_elem(0, 3, triElemToFace);
const static int quadElemToFace[] = {0, 1, 2, 3};
FEM_Add_ghost_elem(1, 4, quadElemToFace);

FEM_Add_ghost_layer(2, 0);
const static int quad2ElemToFace[] = {0,1, 1,2, 2,3, 3,0};
FEM_Add_ghost_elem(1, 4, quad2ElemToFace);
```

Accessing Ghosts

- The entity tag for the ghosts of a particular entity is:

```
FEM_GHOST+<entity_tag>
```

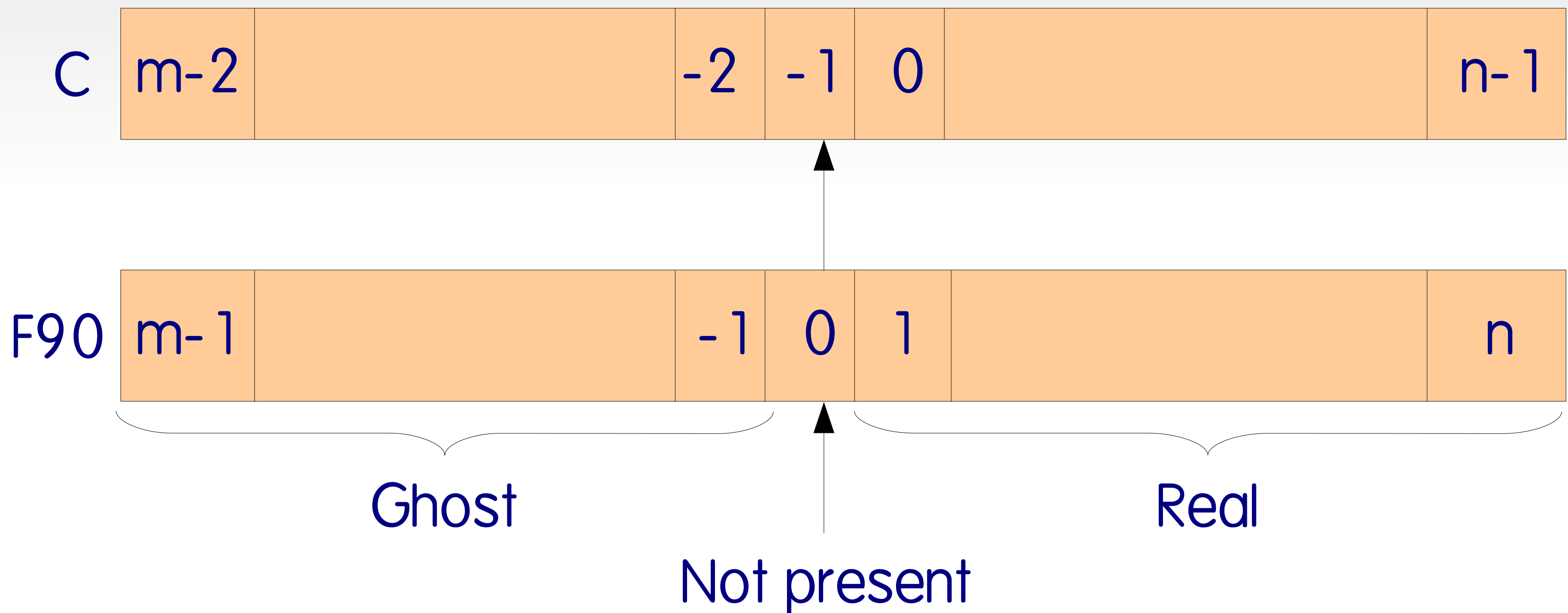
- Example: the tag for the ghosts of element type 1:

```
FEM_GHOST+FEM_ELEM+1
```

- Real elements (local) will have all real nodes, but ghost elements may have some real nodes, but also some nodes that are either not present, or are included in the ghost nodes
- Ghost element connectivity contains -1 (C) or 0 (FORTRAN) for non-present nodes, and negative integers for ghost nodes

Accessing Ghosts

- Real and ghost node numberings in C and FORTRAN with *n* real nodes and *m* ghost nodes:



Partitioning

- Ghosts are specified in the `init`
- Choice of serial (METIS) or parallel (memory-efficient algorithm using ParMETIS) partitioning:

```
extern int FEM_Partition_Mode;
// ...
FEM_Partition_Mode = 1;    // serial partitioning
FEM_Partition_Mode = 2;    // parallel partitioning
```

- When using the traditional program structure, partitioning is automatically performed after `init` and before the `driver`

Partitioning

- In the MPI-style structure, it must be explicitly called, after specifying the ghost layers

```
int parMesh;
if (rank == masterRank) {
  int serialMesh;
  // put mesh in serialMesh...
  // add ghost layers...
  parMesh = FEM_Mesh_broadcast(serialMesh, masterRank, comm);
  FEM_Mesh_deallocate(serialMesh);
}
else {
  parMesh = FEM_Mesh_broadcast(-1, masterRank, comm);
}
```

Node & Ghost Synchronization

- ParFUM can combine data on shared nodes
- ParFUM can update data on ghost nodes and elements
- ParFUM doesn't know the *meaning* of data attributes on nodes and elements; it only knows *location* and *type* of the data
- Create a *layout* for data that is to be exchanged – this specifies type and how the data is organized

Node & Ghost Synchronization

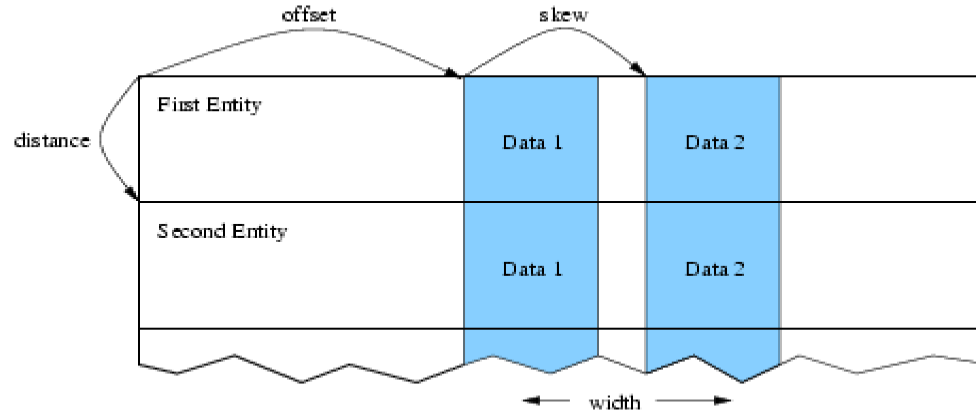
```
IDXL_Layout_t IDXL_Layout_create(int type, int width);
```

- Example: want to combine data for three values of force per node:

```
double *force = new double[3*n];
IDXL_Layout_t force_layout IDXL_Layout_create(IDXL_DOUBLE, 3);
```

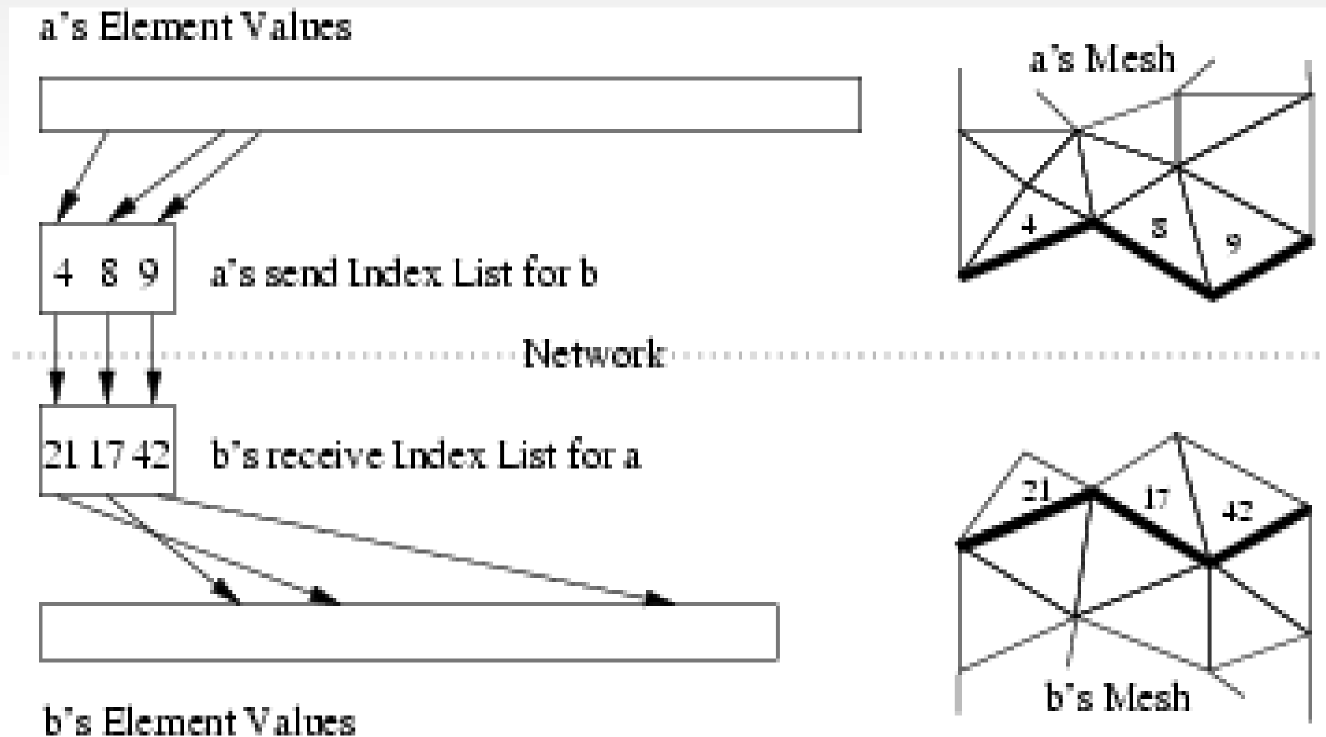
- Can create a field on complex data structures

```
IDXL_Layout_t IDXL_Layout_offset(int type, int width, int offsetBytes,
  int distanceBytes, int skewBytes);
```



Node & Ghost Synchronization

- ParFUM uses *indexed lists* (IDXL) for specifying locations of data on opposite sides of partition boundaries



Node & Ghost Synchronization

- The IDXL for shared nodes has identical send/recv lists:

```
IDXL_t FEM_Comm_shared(int mesh, int entity);
```

- The IDXL for ghosts will have differing send/recv lists:

```
IDXL_t FEM_Comm_ghost(int mesh, int entity);
IDXL_t IDXL_Create(void);
void IDXL_Combine(IDXL_t dest, IDXL_t src, int startSend,
  int startRecv);
```

Node & Ghost Synchronization

- We have acquired location and specified layouts for our data, now how do we communicate?

```
void IDXL_Comm_sendsum(IDXL_Comm_t comm, IDXL_t indices,
  IDXL_Layout_t layout, void *data);

void IDXL_Comm_sendrecv(IDXL_Comm_t comm, IDXL_t indices,
  IDXL_Layout_t layout, void *data);
```

- *comm*=0 is blocking, *comm* != 0 is non-blocking

Node & Ghost Synchronization

- Example: summing three force values on shared nodes

```
// C++
double *force = new double[3*nNodes];
// assume force has values from local solver computation
IDXL_Layout_t force_layout = IDXL_Layout_create(IDXL_DOUBLE, 3);
IDXL_t shared_indices = FEM_Comm_shared(mesh, FEM_NODE);

  ...in the time-loop...
  IDXL_Comm_sendsum(0, shared_indices, force_layout, force);

! F90
double precision, allocatable :: force(:,:)
integer :: force_layout, shared_indices
ALLOCATE(force(3,nodes))
! assume force has values from local solver computation
force_layout=IDXL_Layout_create(IDXL_DOUBLE, 3)
shared_indices=FEM_Comm_shared(mesh, FEM_NODE)

  ...in the time loop...
  CALL IDXL_Comm_sendsum(0, shared_indices, force_layout, force)
```

Node & Ghost Synchronization

- Example: updating seven double solution values on element ghosts

```
// C++
double *elemData = new double[7*(nElems+nGhostElems)];
// assume elemData has values from local solver computation
IDXL_Layout_t elemData_layout = IDXL_Layout_create(IDXL_DOUBLE, 7);
IDXL_t ghost_orig = FEM_Comm_ghost(mesh, FEM_ELEM+1);
IDXL_t ghost_shift = IDXL_Create();
IDXL_Combine(ghost_shift, ghost_orig, 0, nElems);

  ...in the time-loop...
  IDXL_Comm_sendrecv(0, ghost_shift, elemData_layout, elemData);

! F90
double precision, allocatable :: elemData(:,:)
integer :: elemData_layout, ghost_orig, ghost_shift
ALLOCATE(elemData(7,nElems+nGhostElems))
! assume elemData has values from local solver computation
elemData_layout=IDXL_Layout_create(IDXL_DOUBLE, 7)
ghost_orig=FEM_Comm_ghost(mesh, FEM_ELEM+1)
ghost_shift=IDXL_Create()
CALL IDXL_Combine(ghost_shift,ghost_orig, 1, nElems+1)

  ...in the time loop...
  CALL IDXL_Comm_sendrecv(0, ghost_shift, elemData_layout, elemData)
```

Node & Ghost Synchronization

- We have specified layouts for our data, now how do we communicate?
- For shared entities:

```
void FEM_Update_field(IDXL_Layout_t field, void *nodeData);

void FEM_Reduce_field(IDXL_Layout_t field, const void *nodeData,
  void *out, int op);
```

- *op* can be: *FEM_SUM, FEM_MIN, FEM_MAX*

Node & Ghost Synchronization

- For ghost entities:

```
void FEM_Update_ghost_field(IDXL_Layout_t field, int entityType,
  void *data);
```

- *entityType* is the element type for element ghosts, or -1 (in C) or 0 (in FORTRAN) to indicate node ghosts

Adjacencies

- ParFUM can derive other adjacency relationships from element connectivity:
  - Element-to-element
  - Node-to-element
  - Node-to-node
- User requests only those needed

Adjacencies

- User defines adjacency relationship (like neighboring relationship for ghosts)

```
void FEM_Add_elem2face_tuples(int mesh, int elemType, int nodesPerTuple,
  int tuplesPerElem, const int *elem2tuple);
```

- Example: define triangle adjacencies

```
const int triangleFaces[6] = {0,1, 1,2, 2,0};
FEM_Add_elem2face_tuples(mesh, 0, 2, 3, triangleFaces);

// Then create whichever adjacencies are desired
FEM_Mesh_create_elem_elem_adjacency(mesh);
FEM_Mesh_create_node_elem_adjacency(mesh);
FEM_Mesh_create_node_node_adjacency(mesh);
```

Adjacencies

- Adjacency tables are added to the ParFUM mesh as special system attributes
- Can be accessed via `FEM_Mesh_data` or through several simple access functions

```
e2e_getAll(int elem, int *neighbors);

n2e_getAll(int node, int **adjElems, int *sz);
```

- Mixed element meshes not supported (yet)

Mesh Adaptivity

- Two approaches in ParFUM:
  - Incremental Adaptivity (2D triangle meshes)
    - edge bisection, edge contraction, edge flip
    - supported in meshes with one layer of edge-neighboring ghosts
    - each individual operation leaves mesh consistent
  - Bulk Adaptivity (2D triangle, 3D tetrahedral meshes)
    - edge bisection, edge contraction, edge flips
    - supported in meshes with any or no ghost layer
    - operations performed in bulk; ghosts and adjacencies updated at end

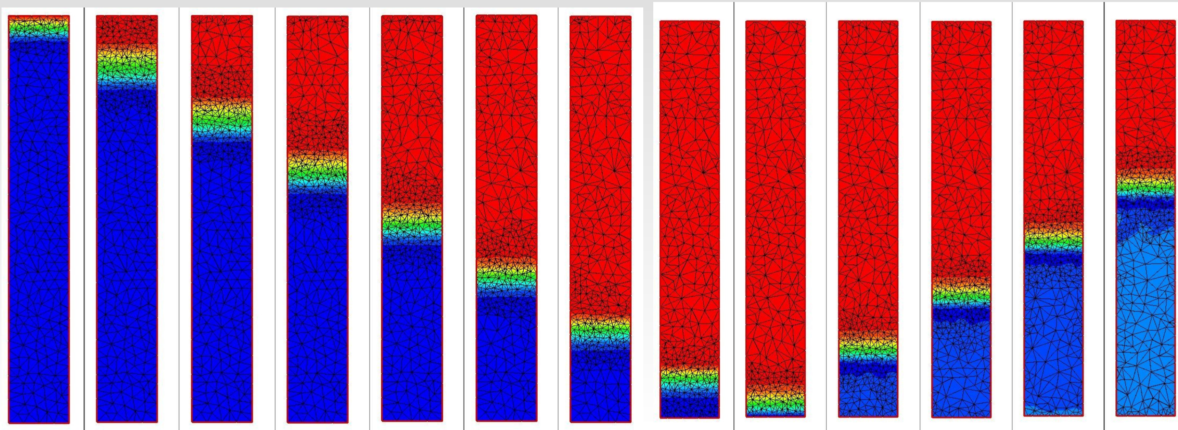
Mesh Adaptivity

- Current high-level adaptivity algorithms use incremental adaptivity; currently transitioning to the bulk approach

```
void FEM_AdaptMesh(int qual, int method, double factor, double *sizes);
void FEM_Refine(int qual, int method, double factor, double *sizes);
void FEM_Coarsen(int qual, int method, double factor, double *sizes);
void FEM_Smooth(int qual, int method);
void FEM_Repair(int qual);
void FEM_Remesh(int qual, int method, double factor, double *sizes);
void FEM_SetReferenceMesh();
void FEM_GradateMesh(double smoothness);
void FEM_SetMeshSize(int method, double factor, double *sizes);
```

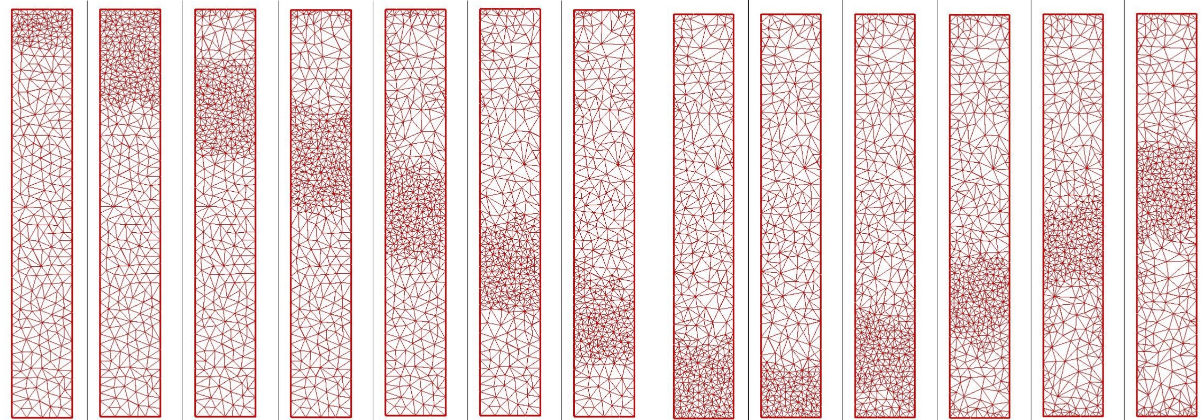
- *`qual`* specifies a quality measure, *`method`* specifies sizing method
  - `UNIFORM` (from `factor`), `USE_SIZES`, `SCALE_CURRENT` (by `factor`), `SCALE_SETTINGS_BY_SIZES`, `SCALE_SETTINGS` (by `factor`), `USE_SETTINGS`

Refinement and Coarsening in a 2D ParFUM Application



Shock propagation and reflection down the length of the bar

Adaptive mesh modification to capture the shock propagation



Download and Install Charm

- ParFUM is included in the Charm CVS

```
[In CSH:]
> setenv CVSROOT ":pserver:checkout@charm.cs.uiuc.edu:/cvsroot"
[In BASH:]
> export CVSROOT=":pserver:checkout@charm.cs.uiuc.edu:/cvsroot"
> cvs login
[hit ENTER at password prompt]
> cvs co -P charm
```

- Build "LIBS" for your machine:

```
> cd charm
[Try smartbuild:]
> ./smartbuild
[Or do a custom build, for example, net-linux:]
> ./build LIBS net-linux -O
```

Compiling and Linking

- Use *`charmc`* in `charm/bin` to compile
  - A multi-lingual compiler driver, builds C/C++ and F90 codes
  - Knows where relevant Charm modules and libraries are
  - Portable across machines and compilers
- Use `-language ParFUM` to link C/C++
- Use `-language ParFUMf` to link F90

Program Execution

- Use *`charmrun`* in `charm/bin` for net versions, varies for other versions
  - A portable parallel job execution script
  - Specify number of processors: +p<N>
  - Specify number of VPs (aka partitions): +vp<K>
  - Net versions of Charm need a *`nodelist`*

```
> cat .nodelist
group main ++shell ssh
  host cool1.cs.uiuc.edu
  host cool2.cs.uiuc.edu
  host cool3.cs.uiuc.edu
  host cool4.cs.uiuc.edu
> ./charmrun ./pgm +p4 +vp32
```

What's this +vp<K> business?

- Virtualization:
  - Applications on the Charm Run-time System are built using migratable objects
  - Each ParFUM mesh partition maps to an AMPI thread which in turn maps to a migratable object
  - Object-based virtualization = multiple migratable objects per processor
  - Enables load balancing, adaptive overlap of communication/computation, better cache performance

ParFUM

Adjacency Generation

User's Solver

Solution Transfer

Bulk Adaptivity

Ghost Layer Generation

pTopS

Contact

Collision Detection

Partitioning

I FEM

Incremental Adaptivity

Multi-phase Shared Arrays

AMPI

Charm++

User View

System View

Charm Run-time System

Load Balancing Framework

Communication Optimizations

Example Programs

- In the Charm distribution:
  - charm/examples/ParFUM
  - A good place to start: simple2D
    - simple structural dynamics program
    - can be run with NetFEM for live visualization

Related Topics

- Many useful libraries associated with ParFUM:
  - Load Balancing
  - Collision Detection
  - Solution Transfer
  - NetFEM (Mesh visualization)
  - IFEM: Iterative Solver Library
  - Multi-block Framework