OpenDIEL

Supported by The National Science Foundation

Tristin Baker, Jordan Scott, and Zachary Trzil
Mentor: Dr. Kwai Wong

Introduction

What is OpenDIEL?

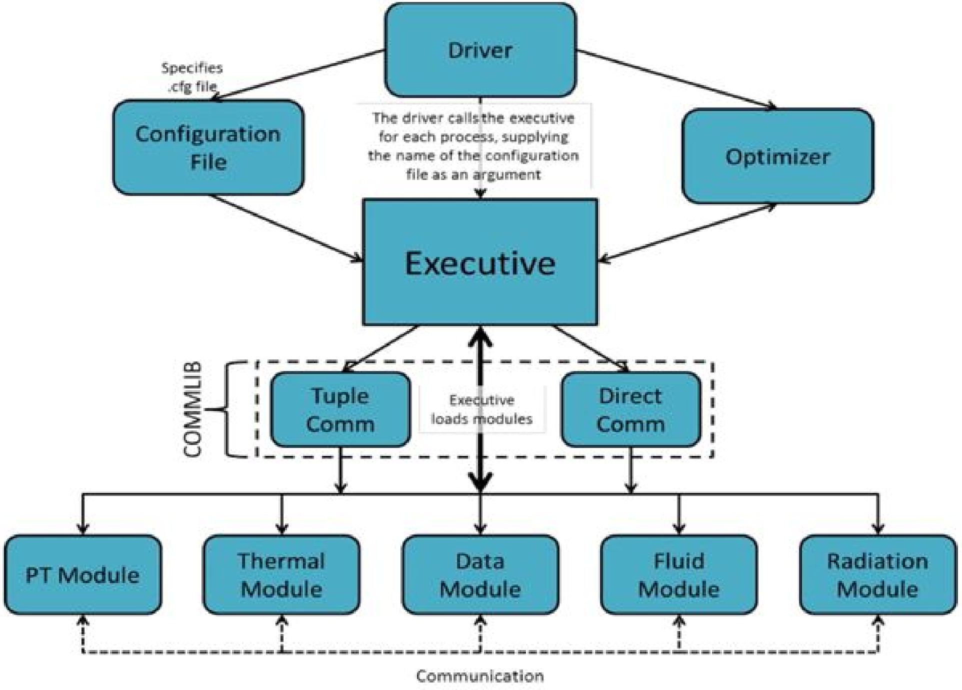
- Lightweight workflow framework for HPC's to run multiple parallel softwares as OpenDIEL Modules from one executable
    - Allows for communication and data transfer between the different modules
- Uses a driver to drive the IELExecutive
    - Driver reads a workflow file to identify the different modules/groups/sets the IELExecutive will use in order to run
- Uses MPI (Message Passing Interface) to facilitate transfer of data and information the different modules need
- Has two forms of communication
    - Direct Communication and Tuple Space Communication.

What is OpenDIEL? (cont.)

- Used for simulating system-wide scientific applications
- Comparison to other Scientific workflow managers
  - Main use is for multidisciplinary work
  - Scalable computational performance
  - DIEL’s 2 types of communication
- **Open**DIEL

OpenDIEL Requirements

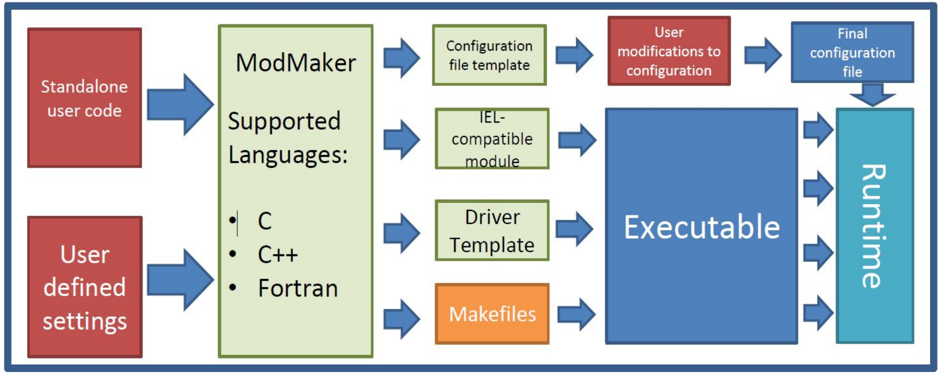
- Driver File
- Config File
- User source code
  - ModMaker



ModMaker

- Python Package

- Transforms a C, C++, or Fortran file (or directory of files) into an openDIEL module(s)



OpenDIEL Workflow: Modules

- An openDIEL compatible piece of code.
- Each module is given parameters
- Can be Parallel or Serial

```
modules=(
    function="laplace0"
    args=()
    libtype="static"
    shared_bc_read=([500,999])
    shared_bc_write=([0,499])
    size=1
```

OpenDIEL Workflow: Groups

- Allows user to specify which order to run modules in
- Groups can have dependencies to other groups
  - The specified group must finish before the dependent one can run
- Modules contained in groups always run in serial
- User can also specify how many iterations each group should run

```
group2:
{
    order=("HELLO1, HELLO2")
    iterations=1
    depends=("group1")
}
```

OpenDIEL Workflow: Sets

- Allows user to specify which groups should run concurrently
  - Each group's modules run in serial, but the groups can run concurrently
- Allows user to specify the number of iterations a set will run
  - This determines how many times each group in the set will run

```
set1:
 {
   num_set_runs=1
   group1:
   {
     order=("laplace0")
     iterations=1
   }
   group2:
   {
     order=("laplace1")
     iterations=1
   }
   group3:
   {
     order=("laplace2")
     iterations=1
   }
 }
}
```

Graphical User Interface

GUI and Workflow

- Interface is used to organize modules, sets, and groups and to create the “workflow.cfg” file that is used by the OpenDIEL driver to run
- This will be done by using Java’s DnD (Drag and Drop) functions
- User can use GUI to execute code once workflow file is written

```
num_shared_bc=2000
tuple_space_size=1
modules=(
 {
  function="ielTupleServer";
  args=();
  libtype="static";
  library="libIELexec.a";
  size=1;
 },
 {
  function="laplace0"
  args=()
  libtype="static"
  shared_bc_read=([500,999])
  shared_bc_write=([0,499])
  size=1
 },
 {
  function="laplace1"
  args=()
  libtype="static"
  shared_bc_read=([0,499],[1500, 2000])
  shared_bc_write=([500,999],[1000, 1499])
  size=1
 },
 {
  function="laplace2"
  args=()
  libtype="static"
  shared_bc_read=([1000,1499])
  shared_bc_write=([1500, 2000])
  size=1
 }
)
```

```
workflow:
{
 tuple_set:
 {
  tuple_group:
  {
   order=("ielTupleServer")
   iterations=1
  }
 }
 set1:
 {
  num_set_runs=1
  group1:
  {
   order=("laplace0")
   iterations=1
  }
  group2:
  {
   order=("laplace1")
   iterations=1
  }
  group3:
  {
   order=("laplace2")
   iterations=1
  }
 }
}
```

Using GUI for Execution

- User can now execute code through the GUI using Java’s ProcessBuilder and Process classes.
- The output of the program will be sent to the OpenDIEL’s output tab, which has also seen improvements this summer.
- User specifies where script is.

```
private void
runIconActionPerformed(java.awt.event.ActionEvent
evt) {
         String[] args = new String[] {"/bin/sh",
               "-c",
               "./runscript"};
        String line;
        Process p;
        ProcessBuilder pb = new
        ProcessBuilder(args);
        pb.redirectErrorStream(true);
        pb.directory(new
File(System.getProperty("user.dir")));
        tabPanel.setSelectedIndex(3);
        p = pb.start();
        InputStream is = p.getInputStream();
        InputStreamReader isr = new
InputStreamReader(is);
        BufferedReader input = new
BufferedReader(isr);
        System.out.flush();
        while((line = input.readLine()) != null) {
            this.output.readFromExecution(line);
        }
        input.close();
        JOptionPane.showMessageDialog(this,
"Done!");
```

Output Tab

- Now has the ability to show output of execution of programs.
- User can save output into a text file for later viewing.
- Gives live feedback of output

- Improvements:
    - Make this process multithreaded so output will show even when scrolling down.

Live Demonstration

Not Yet Implemented

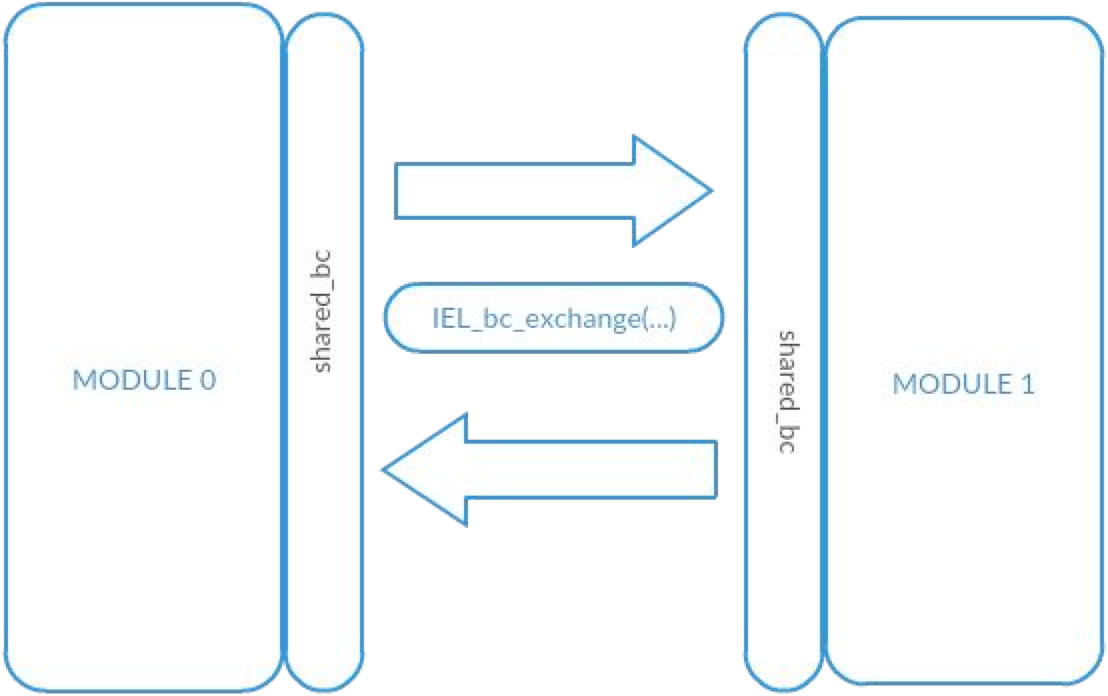
- The Drag-and-Drop features does not yet work, but the framework is implemented in the GUI and does not require much more work to get functioning.
- Allowing the user to launch code remotely via SSH.
- Allowing the user to convert their code to OpenDIEL compatible modules by using the “ModMaker”.

Direct Communication

Direct Communication

- Method for modules to share data directly between one another
- Facilitated by a shared boundary condition
  - Main method of direct communication
  - Found in IEL_exec_info_t data structure as double * shared_bc
    - Each module has shared_bc_write and shared_bc_read
      - Sizes modified in workflow configuration file
  - shared_bc size is set in the workflow configuration file
- IEL_bc_put and IEL_bc_get
  - Method the modules use for their shared_bc_read and shared_bc_write to communicate
  - Wrappers around MPI_Send and MPI_Receive
  - IEL_bc_exchange

Direct Communication Visualized

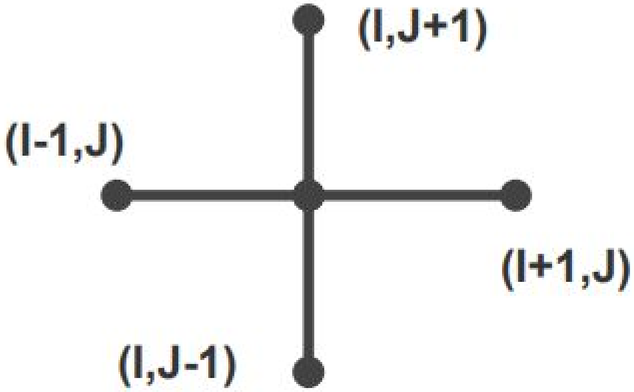


Using Direct Communication For Laplace Example

- OpenDIEL implementation of MPI Laplace example program
- In the original, the 1000x1000 matrix existed in one function.
- With the OpenDIEL implementation, the matrix can be any size and can be split into however many functions the user wants thanks to the shared_bc.
- Within the workflow configuration file, the user sets where the modules can read from and write to.
    - In this example, the user will want the ‘shared_bc_write’ to be equal to its top and bottom rows, and its ‘shared_bc_read’ field to be able to read from the module above and below’s ‘shared_bc_write’ field.

Laplace Example Visualized

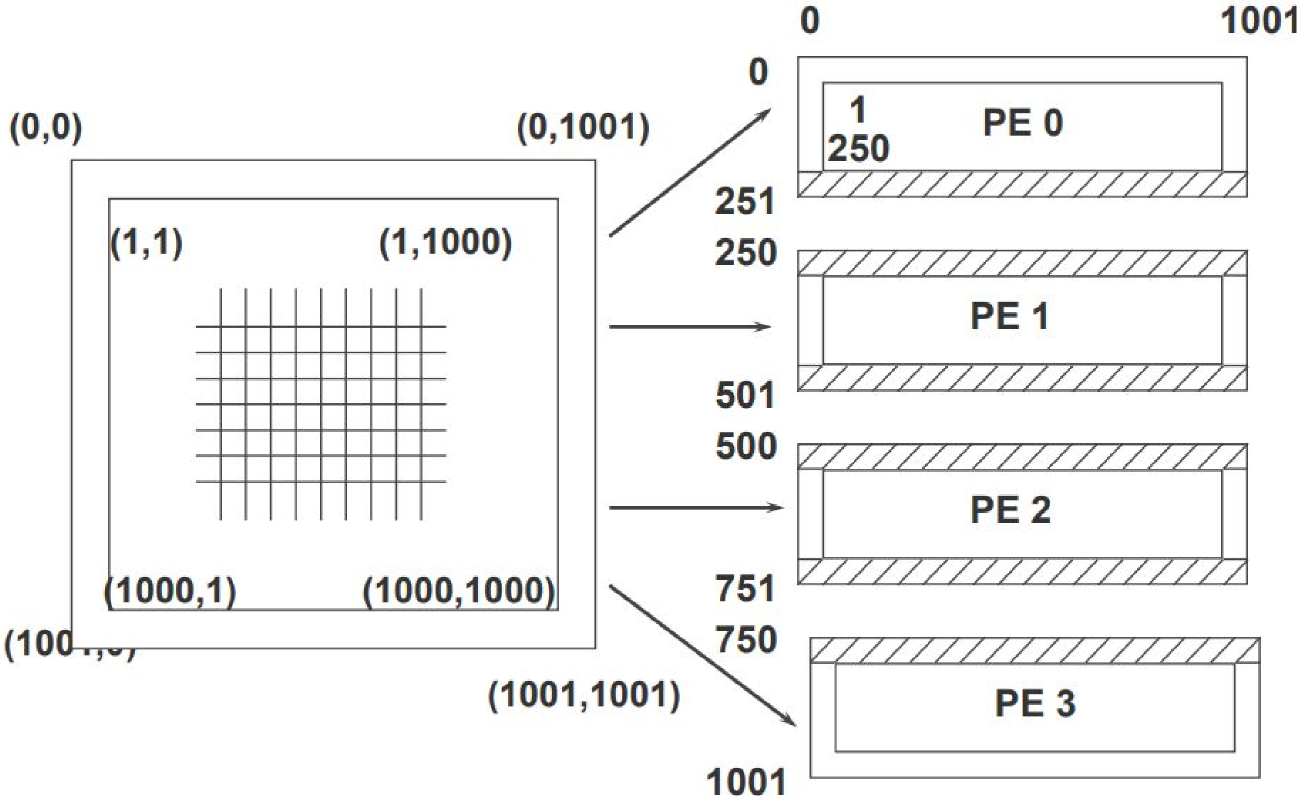
5- POINT FD STENCIL



POINT JACOBI ITERATION

```
T( I , J ) =  0.25 * { Told( I -1 , J  )
                      +( Told( I+1 , J )
                      +( Told( I , J  -1 )
                      +( Told( I , J + 1) }
```

Laplace Example Visualized



Laplace Example Visualized

```
{
  function="laplace1"
  args=()
  libtype="static"
  shared_bc_read=([0,499],[1500, 2000])
  shared_bc_write=([500,999],[1000, 1499])
  size=1
},
```

```
/* Set the shared_bc write equal to the top row of t.
 * shared_bc write is specified by the user in the workflow.cfg file */
for(i = 500; i < 1000; i++) {
  exec_info->shared_bc[i] = t[0][i%500];
}

/* This function sends the shared_bc 'write' values to the
 * "laplace0" shared_bc 'read' values and 'reads' the shared_bc
 * 'write' values from the "laplace0" module.
 */
IEL_bc_exchange(exec_info, "laplace0", &request);

for(i = 1000; i < 1500; i++) {
  exec_info->shared_bc[i] = t[nr-1][i%1000];
}

/* This function sends the shared_bc 'write' values to the
 * "laplace2" shared_bc 'read' values and 'reads' the shared_bc
 * 'write' values from the "laplace2" module.
 */
IEL_bc_exchange(exec_info, "laplace2", &request);

/* Set the top row of t equal to what is received by the IEL_bc
 * exchange from laplace 0. */
for(i = 500; i < 1000; i++) {
  t[0][i%500] = exec_info->shared_bc[i];
}

for(i = 1000; i < 1500; i++) {
    t[nr-1][i%1000] = exec_info->shared_bc[i];
 }
```

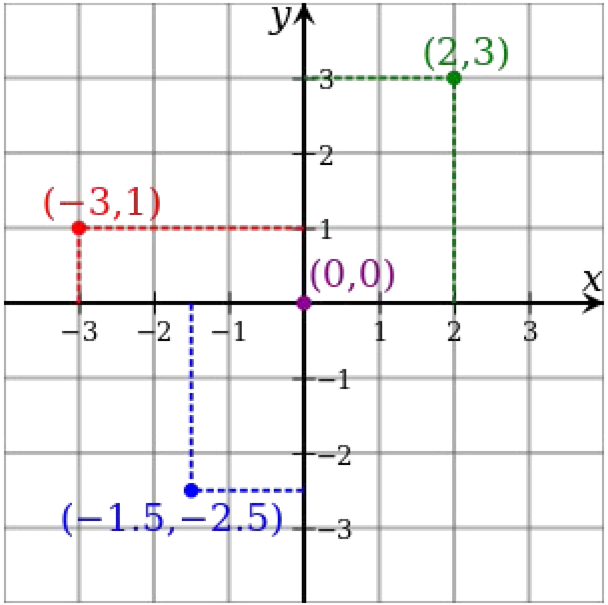
To Do:

- Remove Global shared_bc
    - In its current implementation, if direct communication is being used, OpenDIEL designates a shared_bc array of size num_shared_bc to each module. This is a waste of memory for the most part, and is not scalable.

Tuple Space Communication

Tuple Space - What is it?

- In short, a tuple is a list
  - Example: Cartesian Coordinate system -- a “repository” of two-tuples
  - We use three-tuples to store data. A value that identifies the server (our ‘x coordinate’), a value that identifies the data location on that server (our ‘y coordinate’), and the data itself
  - Imagine if at every point there was a ‘bucket’ in which data could be dumped. This would be a representation of our 3-dimensional tuple space.



Tuple Space - What is it?

- In more technical terms, a tuple space is an associative memory paradigm for distributed/parallel computing
  - Repository of tuples that can be accessed concurrently
  - Processes can put, read, and delete tuples from the repository
  - Goals: Minimize blocking communication and maximize scalability and usability
    - All processes communicate with each other through the tuple space

Tuple Space Communication

- Requirements:
    - Non-blocking -- Sender does not wait for message to be received

```
|---Axxxxx--|    |------------D-|               |---A--|  |----------D-|
|--------Bxx-|   |-------Exx---|       VS       |--------B-|  |------E--|        x = wait time for blocking I/O
|-------------C| |--Fxxxxxx--|                  |-------------C|  |--F--|
```

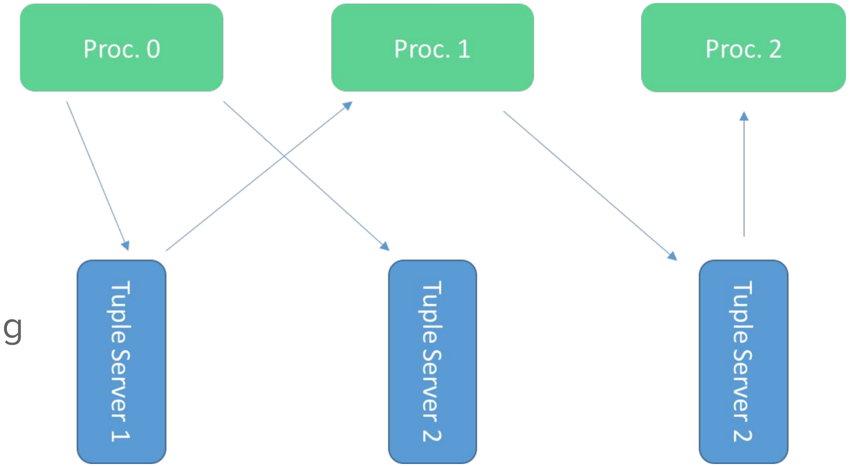
    - Asynchronous -- Concurrent processing in multithreaded environments is necessary for parallel computing
    - Reliability -- Scientific studies require repeatability. Asynchrony can lead to race conditions if not handled properly

Tuple Space - Previous Work

- Facilitated by the Tuple Server
    - The tuple server listens for and intercepts all MPI_Send/Recv calls with MPI_Probe
        - Used the MPI_ANY_TAG to listen
    - Flags are used by the sender to let the tuple server know how to respond to requests and are used during server initialization
        - These flags were errantly picked up by already initialized servers
    - Data is stored in a RB-Tree of arbitrary size. Each individual node has a “data tag” (hash)
    - Within each node, there is a queue of messages to be read
    - Data profile: [data tag (hash) I data size I data]

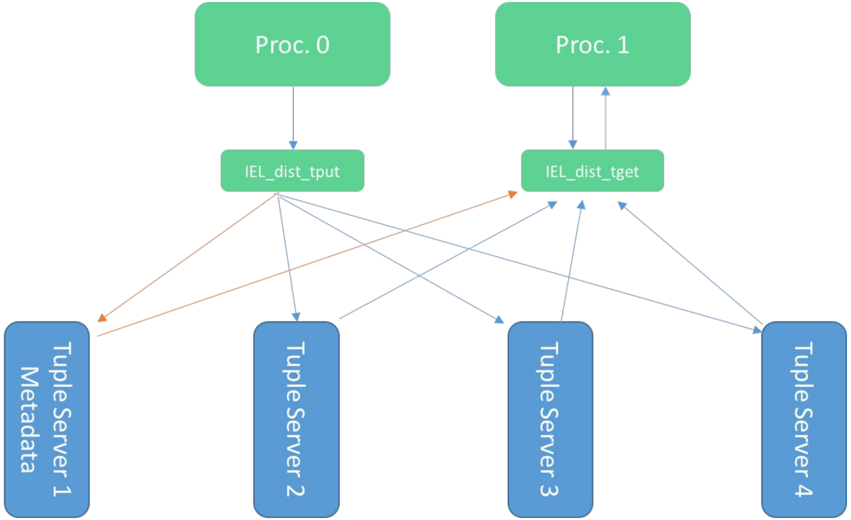
Distributed Tuple Space

- Modules may use a distributed array of tuple servers to store data in system memory
    - The user may specify a single server to place data on and may access multiple tuple servers concurrently
    - A process that calls for data must wait until the data is put on the server -- blocking



Distributed Tuple Space

- Modules may use a distributed array of tuple servers to store data in system memory
    - The user may instead request that the data be distributed evenly among the tuple servers
    - The user does not need to choose which server to use
    - Both methods may be used in conjunction with each other
    - Does not block! Returns immediately if the data is not present

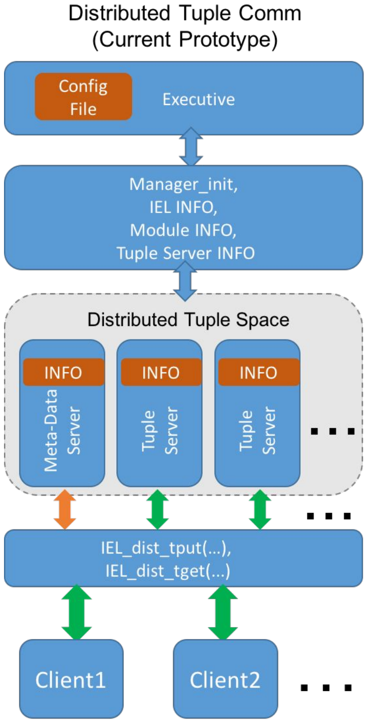


Distributed Tuple Space - Why?

- Speed of communication
    - Each server is its own process with its own memory. Large data transfers can take place on multiple tuple servers simultaneously instead of proceeding serially on a single tuple server
- Data Resiliency
    - Data can be striped across tuple servers to prevent data loss if connection to a node is lost
    - Each server can write critical data to disk when not in use, protecting from system crashes

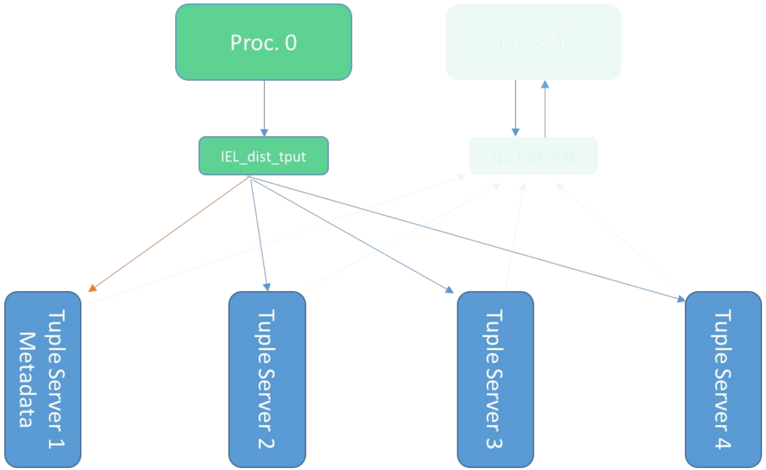
Distributed Tuple Space Implementation

- Each instance of a tuple server is run as a module
  - Driver file and workflow file must be consistent with the number of tuple servers being used
- Server 0 initializes the metadata server and is reserved for managerial tasks
- Server 1 is the metadata server and is reserved
  - Server 1 also contains a struct that stores relevant information about the state of the servers
- All other tuple servers are available for data storage



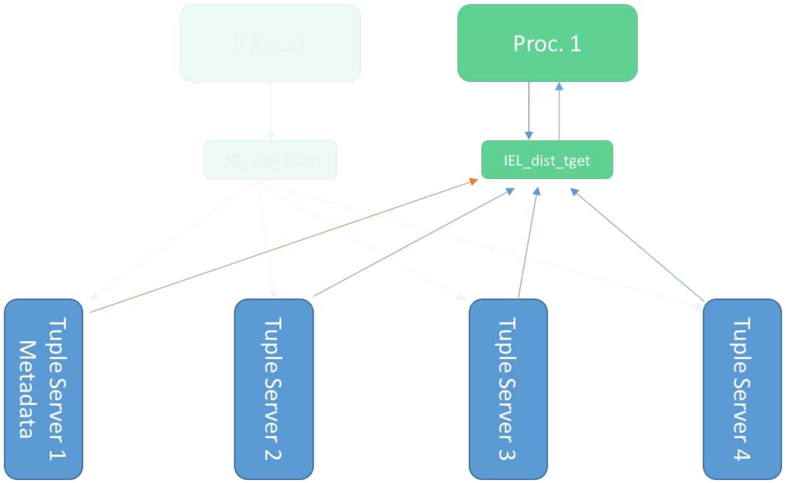
Distributed Tuple Space Implementation

- Sending data
  - A client can send data to the distributed array of tuple servers by calling IEL_dist_tput()
  - The data is distributed evenly among the available tuple servers
  - Sets up two arrays of meta data:
    - The server rank in the order used
    - The size of the corresponding piece of data sent to the tuple server
  - Stores the metadata on the first tuple server



Distributed Tuple Space Implementation

- Receiving data
  - A client can receive data stored on the distributed array of tuple servers by calling IEL_dist_tget()
  - Uses the metadata to pull the data from the servers in the order in which it was stored
  - Reconstructs the data into an array that the caller has access to



Distributed Tuple Space API

- IEL_dist_tput (**size_t** size, **const char** *tag, **void** *data)
    - Size is the size of the data to be sent
    - The tag is a user-defined string to uniquely identify the data (NOTE: it is expected that the string is NULL terminated, otherwise unexpected behavior may occur.)
    - The data is the data to store on the server. If this data is stored in an array, simply pass the array as the parameter

Distributed Tuple Space API

- IEL_dist_tget (**size_t** *size, **const char** *tag, **unsigned char** del, **void** **data)
    - Size will be set by the function call and is the size of the data returned to the user
    - The tag is the user-defined string to identify the data -- the same tag passed to IEL_dist_tput()
    - The del variable is a TRUE/FALSE (1/0) value indicating to delete the data from the server(s) if TRUE and to keep it in place if FALSE.
    - The data is the memory address of an UNALLOCATED pointer to the data that the function will fill in

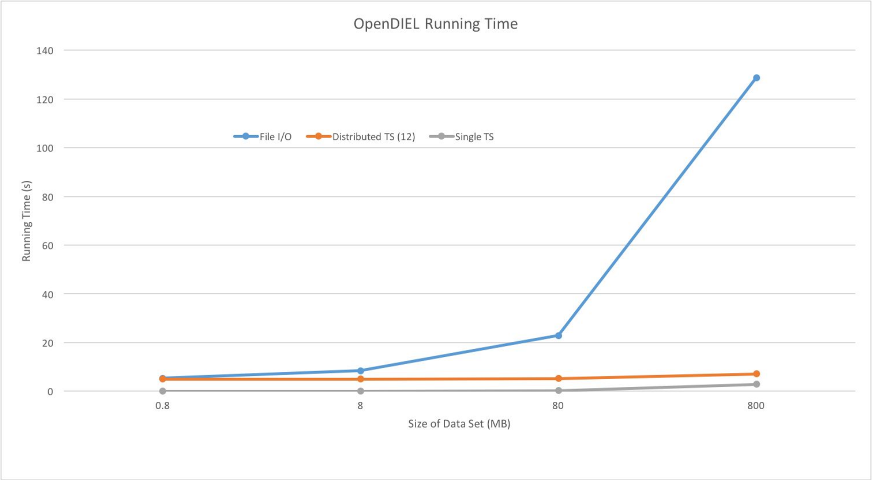
Distributed Tuple Space API

- The original tput and tget functions can be used to access tuple servers concurrently!
  - IEL_tput(**size_t** size, **int** tag, **int** serverRank, **void** *data)
  - IEL_tget(**size_t** *size, **int** tag, **int** serverRank, **unsigned char** del, **void** **data)

- These currently do not interact with the metadata server. In the future, the metadata server can keep track of these calls as well so that a non-blocking version of each function can be created

Distributed Tuple Space Testing

- Methodology
  - The running time of openDIEL was benchmarked with two modules communicating using a single tuple server, distributed tuple servers, and file I/O
- Results
  - The distributed tuple server performance was comparable with the single tuple server performance with a constant small overhead for openDIEL to initialize the extra processes



Future Work

- Create non-blocking versions of the original tput/tget functions
- Develop an algorithm to stripe data across the distributed tuple servers
- Develop a scheme to tag data as critical and write this critical data to disk at certain checkpoints when the tuple server is not in use
    - Create a restore feature to relaunch after a failure
- Create easy to follow documentation and user-guides so that end users can begin using openDIEL for their projects
- Release an alpha version of openDIEL

Acknowledgements

- Funding
  - The National Science Foundation (NSF)

- Facilities
  - The University of Tennessee (UTK) & The Joint Institute for Computational Sciences (JICS)

- Program director/Mentor
  - Dr. Kwai Wong