Workflow and Direct Communication in the openDIEL

(Distributive Interoperable Executive Library)

By Tanner Curren and Nicholas Moran

Mentor: Kwai Wong

What is the openDIEL?

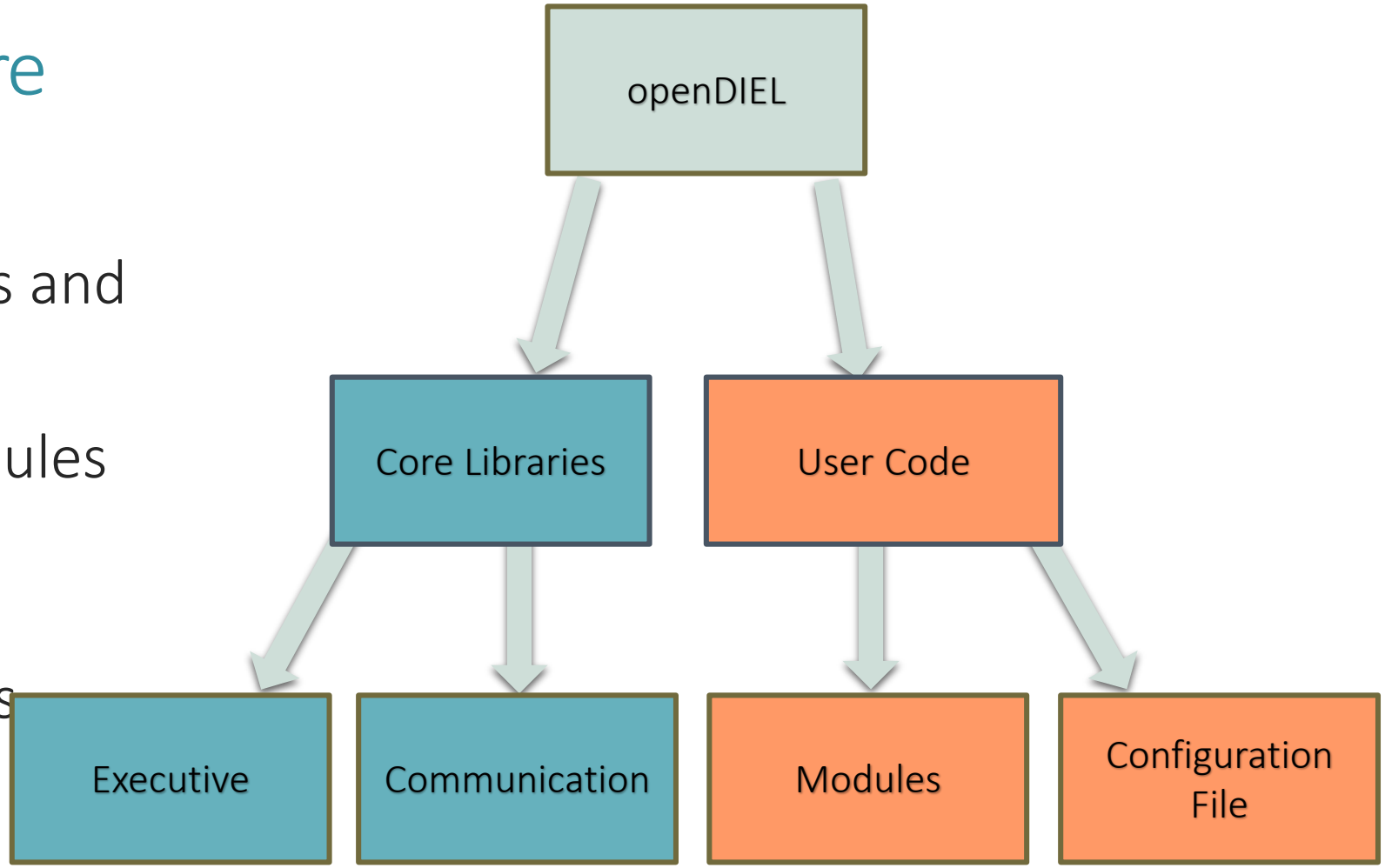
- Open Distributive Interoperable Executive Library
- Provides a framework for using many components (modules) of a loosely coupled system
- Allows ease of access for communication between modules
- Portable and easy to implement into user code

Loosely Coupled Systems

- Systems with components that can require input from other components and output to other components
- Adaptions of serial code, so each module is self-contained aside from I/O
- Called modules; organized by workflow and communicate through Tuple Communication and Direct Communication

openDIEL Structure

- Split into two main sections: core libraries and user code
- Written in C, but modules can be in C, C++, and Fortran
- Configuration file uses libconfig library



openDIEL runtime organization

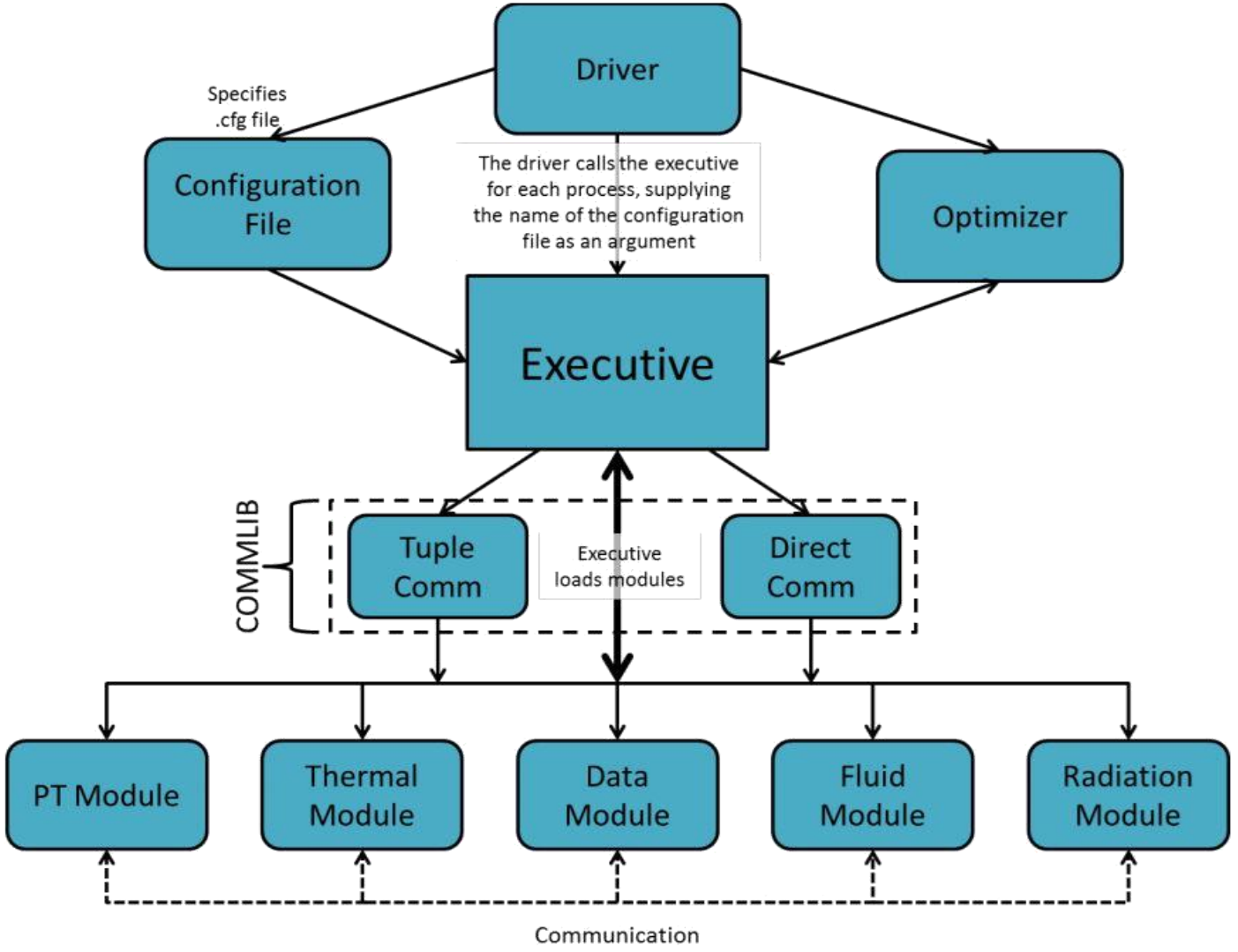


*Diagram by Jason Coan

Using the openDIEL

- Create a configuration file
- Implement openDIEL communication functions into modules
  - Modmaker simplifies this process

- Example configuration file:

```
shared_bc_sizes = []
tuple_space_size=0
```

Global Options

Specification of shared boundary condition array sizes and the tuple space size (number of processes)

```
modules=(
  {
    function="modep7"
    args=()
    libtype="static"
    library="libmodep7.a"
    splitdir="ep7-rv-workflow"
    size=512
  },
  {
    function="modreadvars"
    args=("in.rvi", "monthly")
    libtype="static"
    splitdir="ep7-rv-workflow"
    library="libmodreadvars.a"
    size=512
  },
  {
    function="RAnalysis"
    args=()
    libtype="static"
    library="libRAnalysis.a"
    splitdir="ep7-rv-r-workflow"
    size=1
  }
)
```

Module Specific Options

Information specific to each module, such as identifiers, the amount of processes each module requires, and any arguments that should be passed

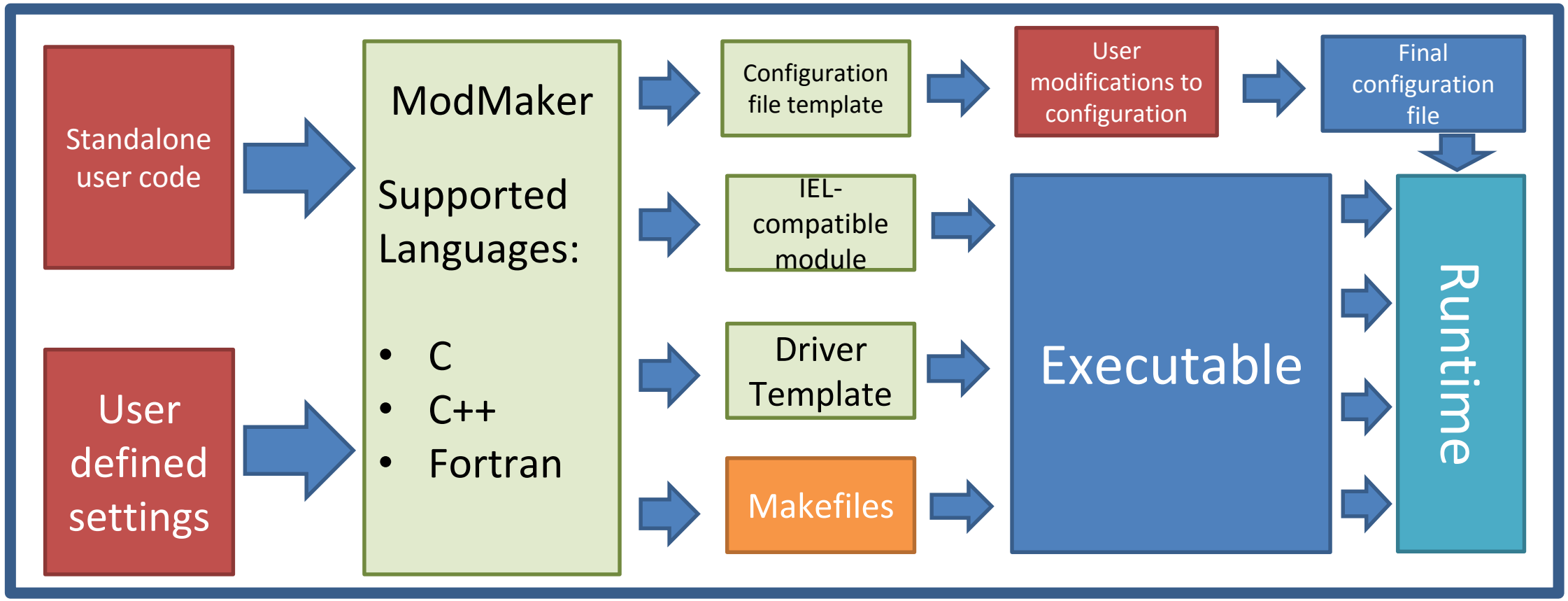
```
workflow:
{

  groups:
  {
    ep7-readvars:
    {
      order=("modep7", "modreadvars")
    }
    RAnalysis:
    {
      order=("RAnalysis")
    }
  }
```

Workflow Specification

Description of the workflow to be executed

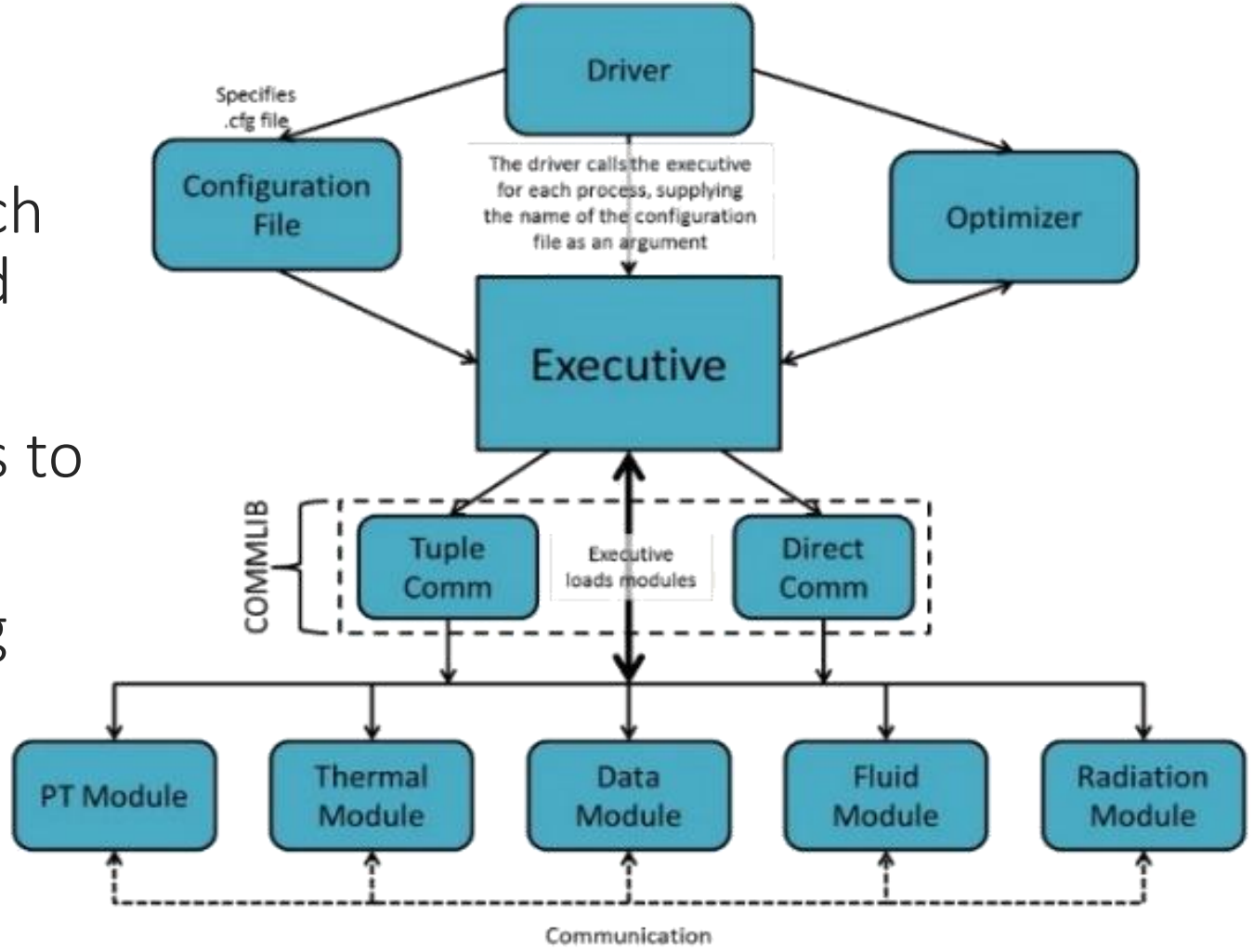
ModMaker

Easily adapts user code into modules for a loosely coupled system



Workflow (Goals)

- openDIEL previously would launch each module simultaneously, and only once.
- Needed to devise a way for users to specify a workflow
- Retain compatibility with existing openDIEL components

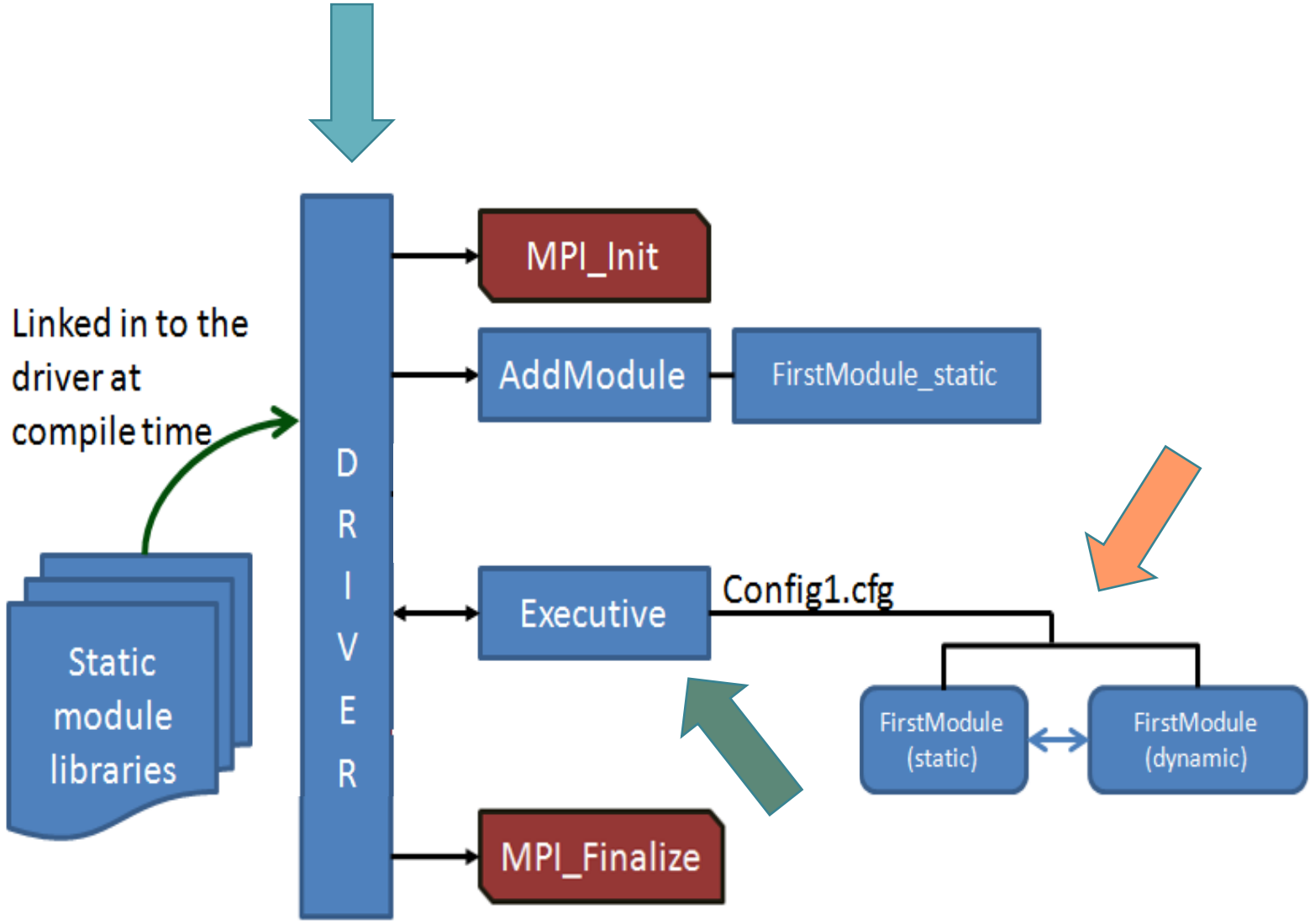


Methods

Method 1

Method 2

Method 3

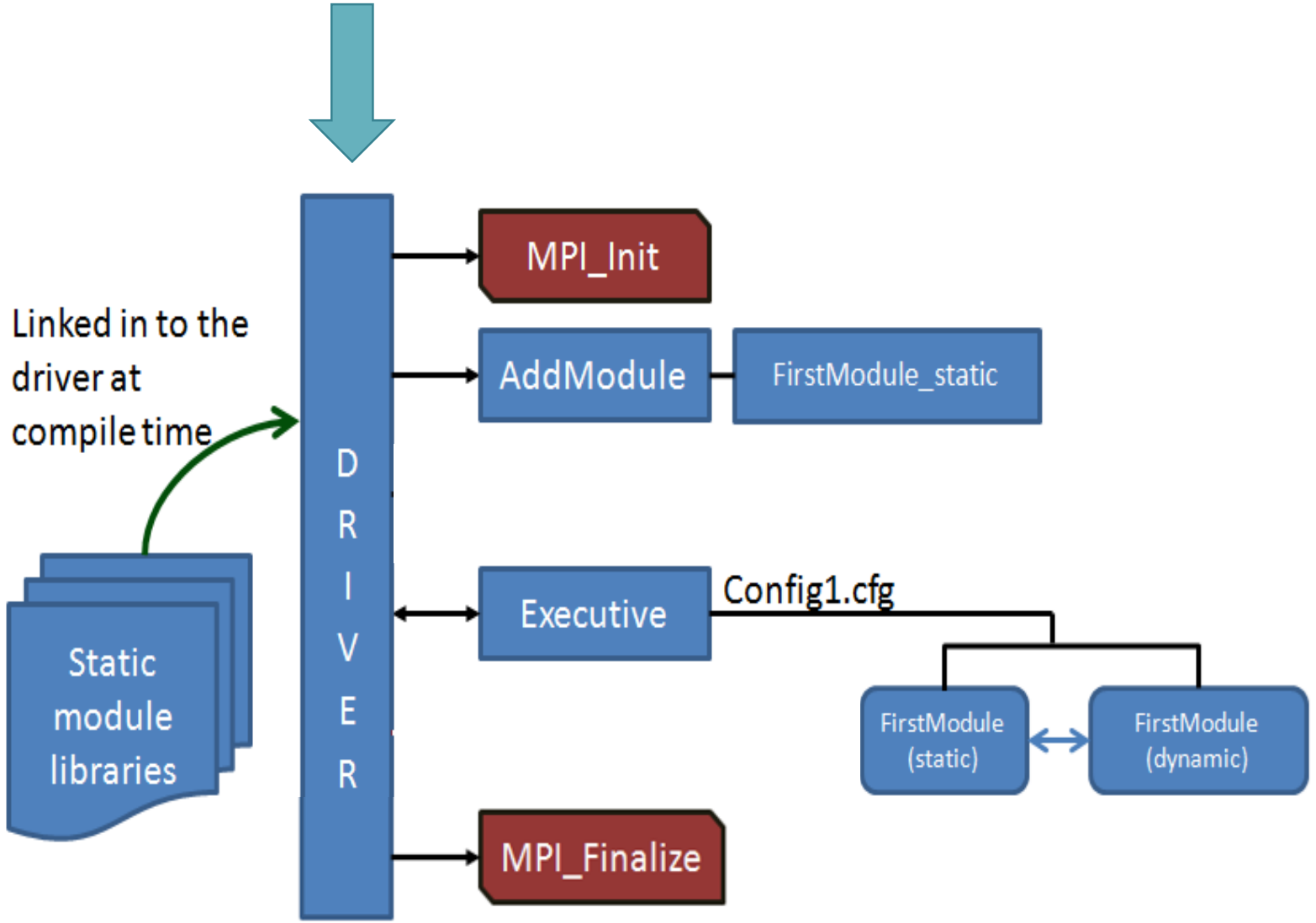


Methods

Method 1

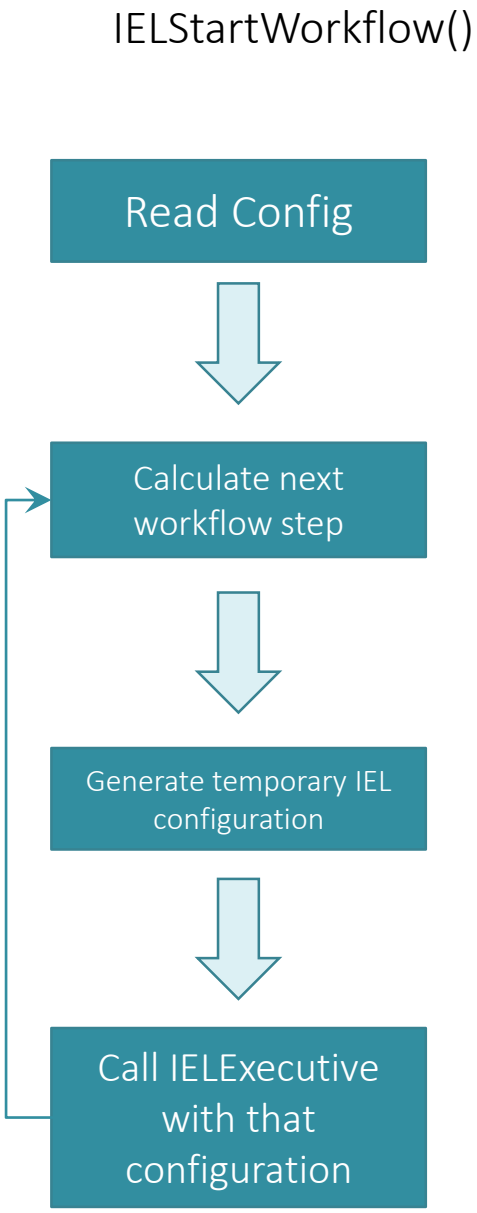
Method 2

Method 3



Workflow (Method 1)

- Achieves a workflow system without changing any code
- Adds a new function (IELStartWorkflow()) to replace the previous call to IELExecutive()
- IELStartWorkflow() will repeatedly call IELExecutive()

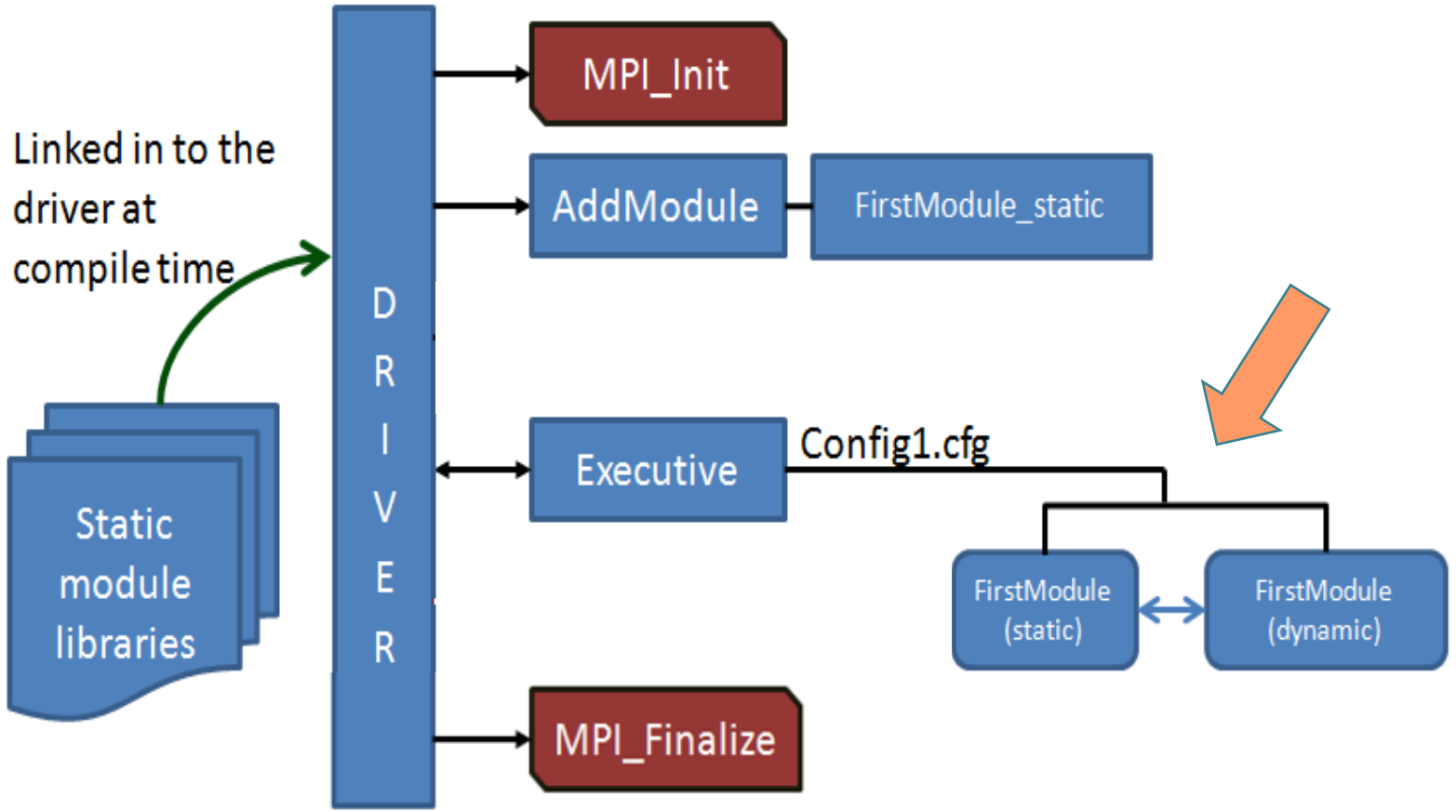


Methods

Method 1

Method 2

Method 3

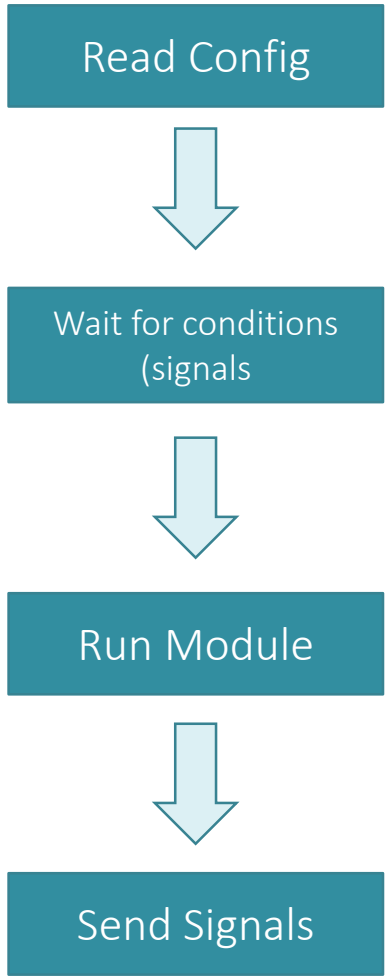


Workflow (Method 2)

- Still not able to change any code
- Develops a dispatcher function
  - Dispatcher uses the Tuple Server to coordinate modules to start/stop via signals, and accesses the function map to run modules
- All modules are started at the same time, and are therefore not able to reuse processes

Workflow (Method 2 continued)

- Dispatcher function reads configuration file
- Waits for preconditions to be satisfied
- Runs module
- Signals next modules

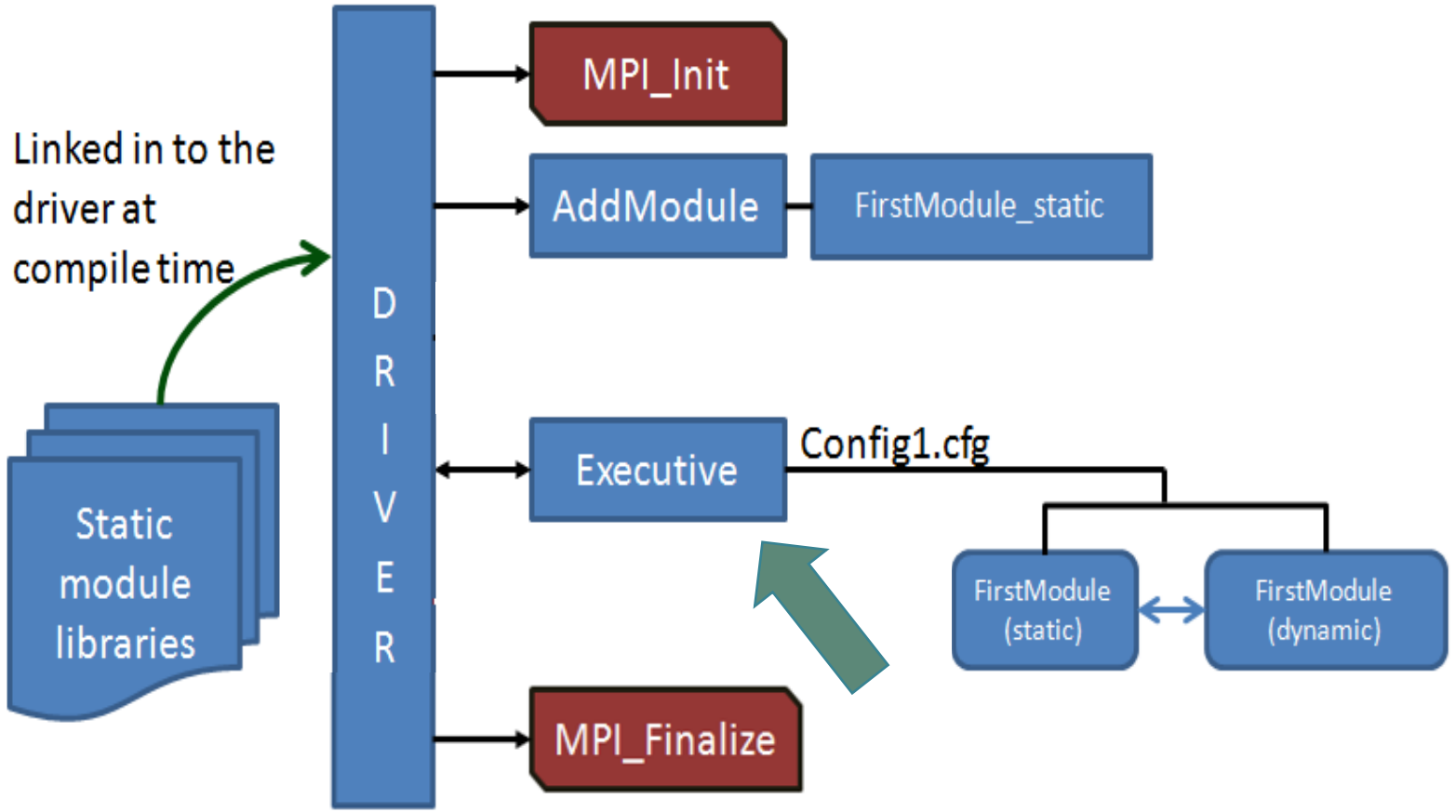


Methods

Method 1

Method 2

Method 3



Workflow (Method 3)

- Makes changes to how the IEL Executive works
- Changes the configuration file to include a workflow section
- Splits modules into groups. Each represents a collection of modules that will execute in the user specified order

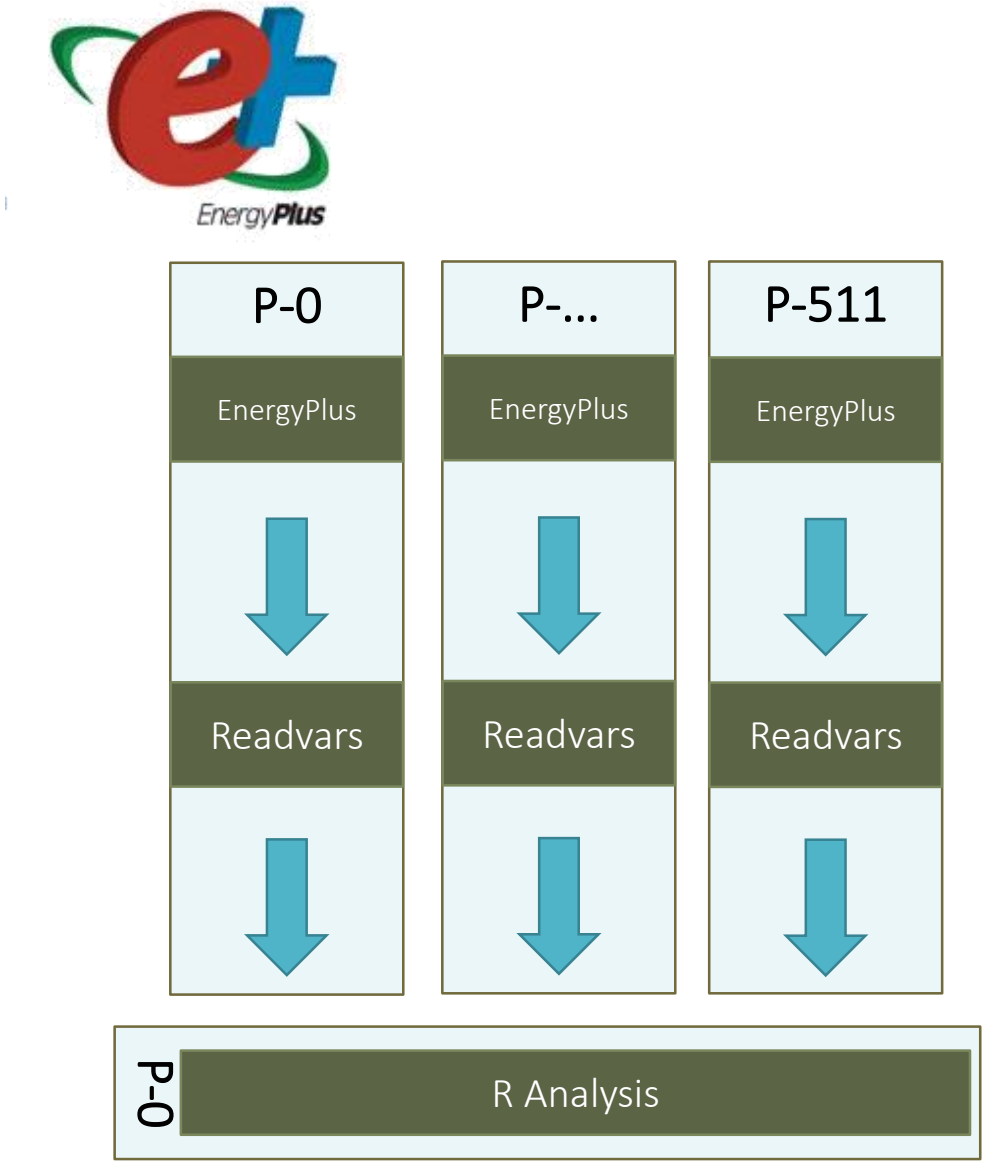
Workflow (Method 3 continued)

- User can specify any number of groups
- Each group will take on the size (# mpi ranks) of the maximum sized module within
- The “iterations” property can be set to determine how many times a group of modules will repeat. A module can access the current group iteration.

```
workflow:
{
  groups:
  {
    ep7-readvars:
    {
      order=("modep7", "modreadvars")
    }
    RAnalysis:
    {
      order=("RAnalysis")
    }
  }
```

Workflow Use Case

- EnergyPlus produces ouput data
- ReadVars extracts the necessary variable based on arguments passed into the openDIEL
- An R script performs statistical analysis on the resulting dataset



Communications in openDIEL

- Currently utilizes two methods of communication: tuple and direct
- Tuple communication sends user-specified data intended for transfer to a process set aside as a tuple server, then other processes take data from the tuple server's queue
- Direct communication sends data stored as boundary conditions directly to other processes; more useful for sending larger chunks of data
- All communications use MPI (Message Passing Interface) wrappers

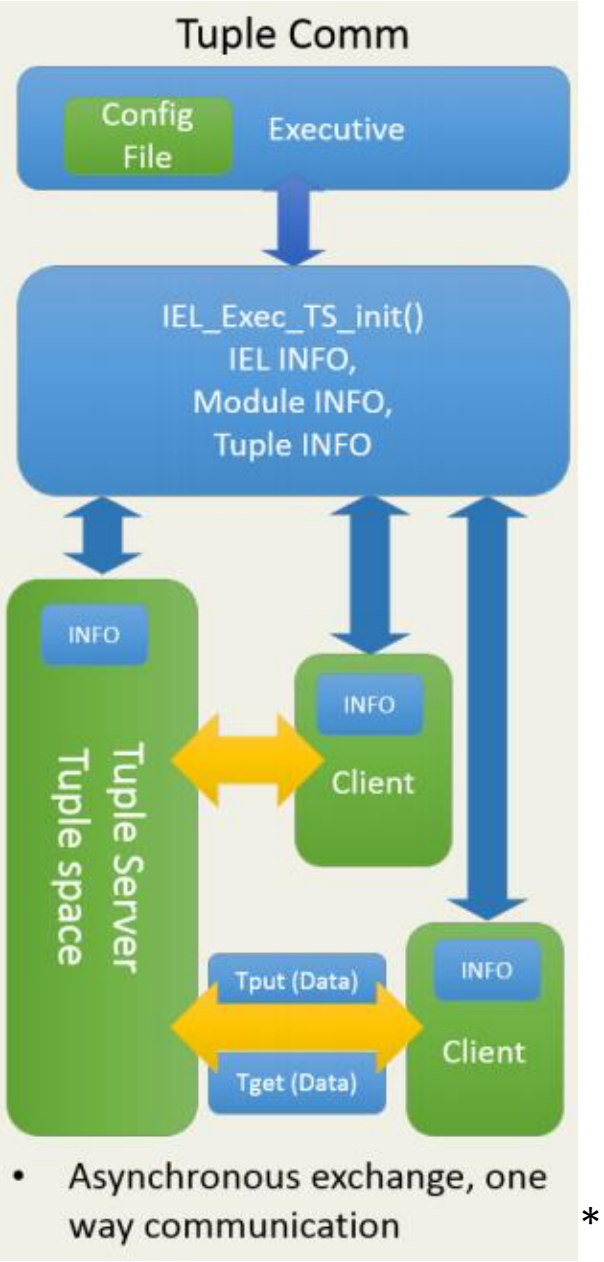


*Diagram by Jason Coan

Jacobi Example: Shared Boundaries

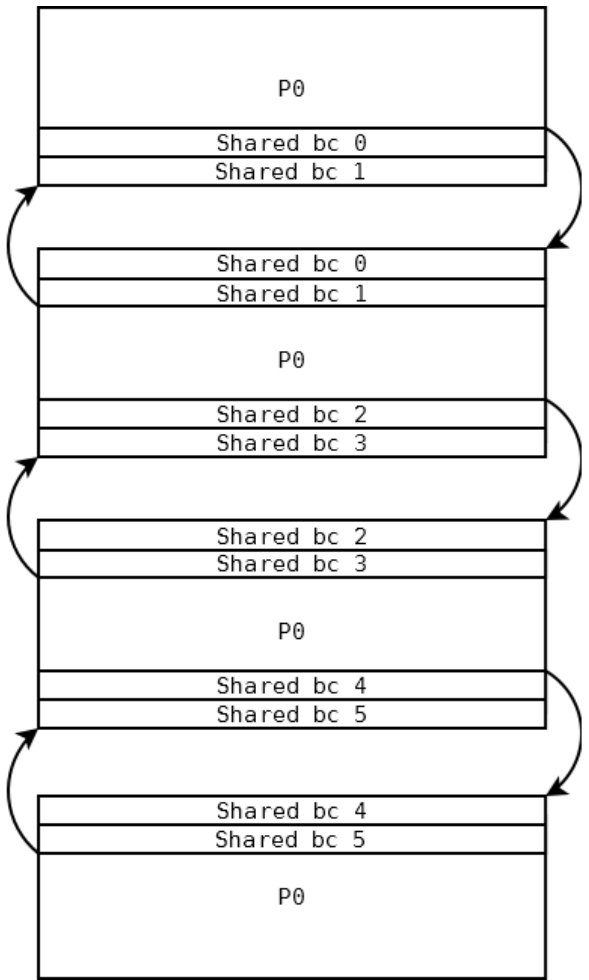
- Based on an already-existing implementation, now has many improvements
- Utilizes shared boundary conditions to move data
- Set up via configuration file; usable like a two-dimensional array
- Processes given certain access to certain boundary conditions to modify, send, and receive
- Movement parameters specify which conditions are sent to where

```
shared_bc_sizes = [16, 16, 16, 16, 16, 16]
tuple_space_size = 0;

modules = (
{
    function = "jacobi";
    args = ();
    libtype = "static";
    library = "libjacobi.a";
    size = 4;
    points = (
        (([0,16]),([0,16]),([0,0]),([0,0]),([0,0]),
         ([0,0])),

        (([0,16]),([0,16]),([0,16]),([0,16]),([0,0]),
         ([0,0])),

        (([0,0]),([0,0]),([0,16]),([0,16]),([0,16]),
         ([0,16])),

        (([0,0]),([0,0]),([0,0]),([0,0]),([0,16]),
         ([0,16]))
    );
}
)

movement = (
    (0, ([0, 16]), (0), (1)),
    (1, ([0, 16]), (1), (0)),
    (2, ([0, 16]), (1), (2)),
    (3, ([0, 16]), (2), (1)),
    (4, ([0, 16]), (2), (3)),
    (5, ([0, 16]), (3), (2))
)
```

Jacobi Example: Shared Boundaries

Use of Jacobi iterations to solve a Laplace equation

Each process works with part of a Matrix

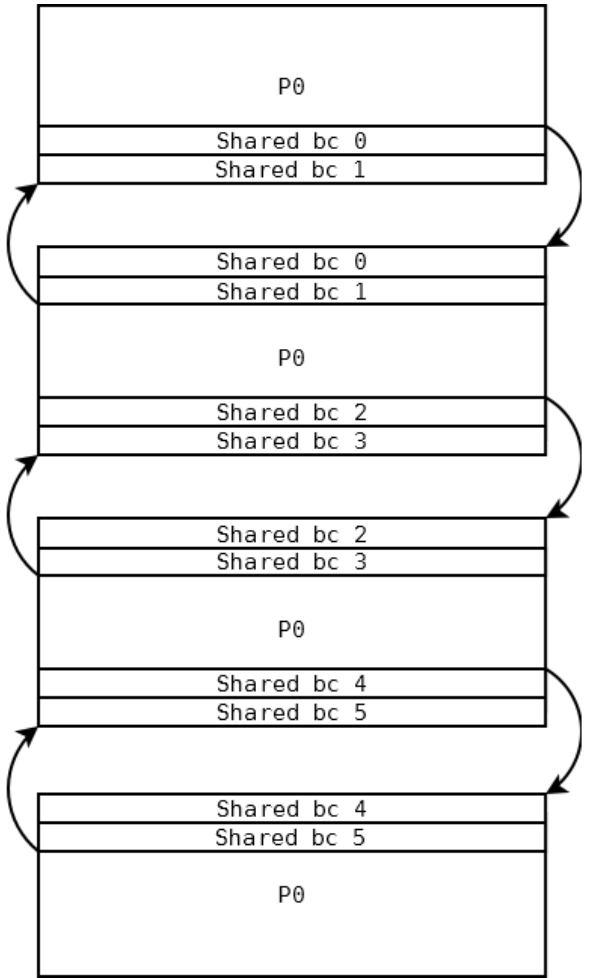



```
shared_bc_sizes = [16, 16, 16, 16, 16, 16]
tuple_space_size = 0;

modules = (
{
    function = "jacobi";
    args = ();
    libtype = "static";
    library = "libjacobi.a";
    size = 4;
    points = (
        (([0,16]),([0,16]),([0,0]),([0,0]),([0,0]),
         ([0,0])),

        (([0,16]),([0,16]),([0,16]),([0,16]),([0,0]),
         ([0,0])),

        (([0,0]),([0,0]),([0,16]),([0,16]),([0,16]),
         ([0,16])),

        (([0,0]),([0,0]),([0,0]),([0,0]),([0,16]),
         ([0,16]))
    );
}
)

movement = (
    (0, ([0, 16]), (0), (1)),
    (1, ([0, 16]), (1), (0)),
    (2, ([0, 16]), (1), (2)),
    (3, ([0, 16]), (2), (1)),
    (4, ([0, 16]), (2), (3)),
    (5, ([0, 16]), (3), (2))
)
```

Jacobi Example: Shared Boundaries

Use of Jacobi iterations to solve a Laplace equation

Each process works with part of a Matrix

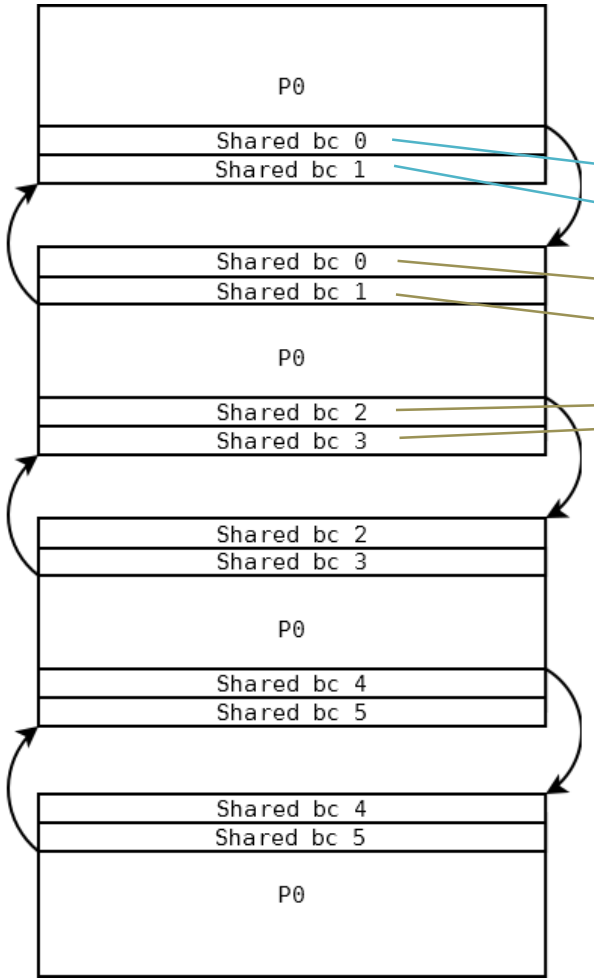



```
shared_bc_sizes = [16, 16, 16, 16, 16, 16]
tuple_space_size = 0;                                  ...

modules = (
{
     function = "jacobi";
     args = ();
     libtype = "static";
     library = "libjacobi.a";
     size = 4;
     points = (
          (([0,16]),([0,16]),([0,0]),([0,0]),([0,0]),
           ([0,0])),

          (([0,16]),([0,16]),([0,16]),([0,16]),([0,0]),
           ([0,0])),

          (([0,0]),([0,0]),([0,16]),([0,16]),([0,16]),
           ([0,16])),

          (([0,0]),([0,0]),([0,0]),([0,0]),([0,16]),
           ([0,16]))
     );
}
)

movement = (
     (0, ([0, 16]), (0), (1)),
     (1, ([0, 16]), (1), (0)),
     (2, ([0, 16]), (1), (2)),
     (3, ([0, 16]), (2), (1)),
     (4, ([0, 16]), (2), (3)),
     (5, ([0, 16]), (3), (2))
)
```

Jacobi Example: Shared Boundaries

Use of Jacobi iterations to solve a Laplace equation

Each process works with part of a Matrix

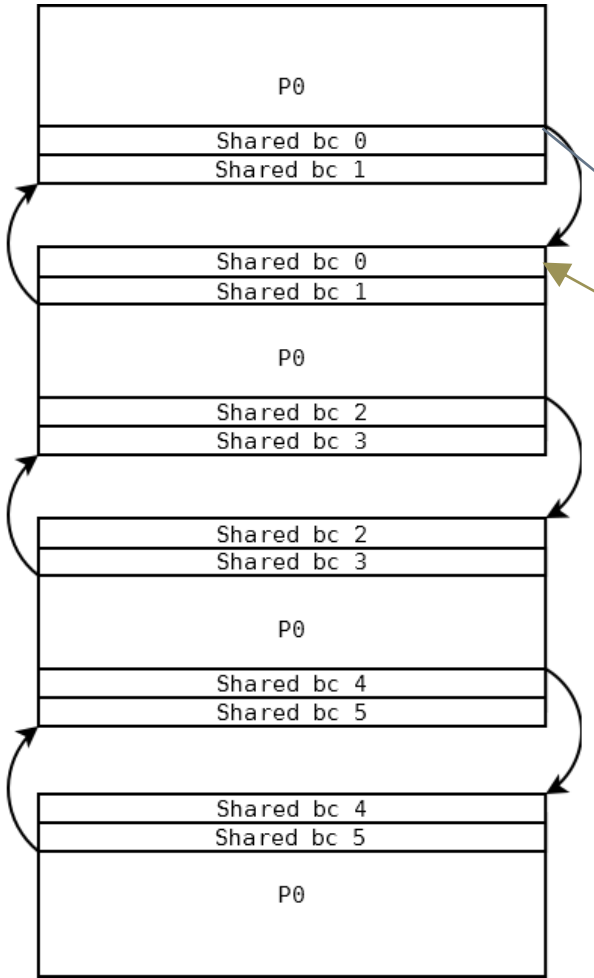



```
shared_bc_sizes = [16, 16, 16, 16, 16, 16]
tuple_space_size = 0;

modules = (
{
     function = "jacobi";
     args = ();
     libtype = "static";
     library = "libjacobi.a";
     size = 4;
     points = (
          (([0,16]),([0,16]),([0,0]),([0,0]),([0,0]),
           ([0,0])),

          (([0,16]),([0,16]),([0,16]),([0,16]),([0,0]),
           ([0,0])),

          (([0,0]),([0,0]),([0,16]),([0,16]),([0,16]),
           ([0,16])),

          (([0,0]),([0,0]),([0,0]),([0,0]),([0,16]),
           ([0,16]))
     );
}
)

movement = (
     (0, ([0, 16]), (0), (1)),
     (1, ([0, 16]), (1), (0)),
     (2, ([0, 16]), (1), (2)),
     (3, ([0, 16]), (2), (1)),
     (4, ([0, 16]), (2), (3)),
     (5, ([0, 16]), (3), (2))
)
```

Jacobi Example: Shared Boundaries

Use of Jacobi iterations to solve a Laplace equation

Each process works with part of a Matrix



Source

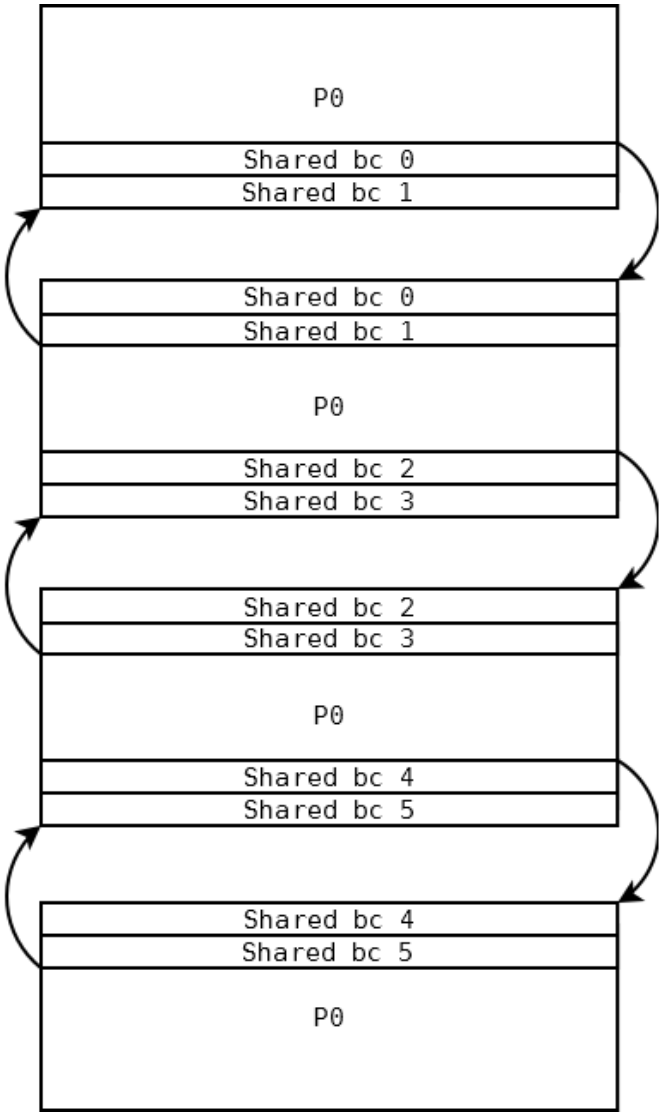
Destination

```
shared_bc_sizes = [16, 16, 16, 16, 16, 16]
tuple_space_size = 0;

modules = (
{
     function = "jacobi";
     args = ();
     libtype = "static";
     library = "libjacobi.a";
     size = 4;
     points = (
          (([0,16]),([0,16]),([0,0]),([0,0]),([0,0]),
           ([0,0])),

          (([0,16]),([0,16]),([0,16]),([0,16]),([0,0]),
           ([0,0])),

          (([0,0]),([0,0]),([0,16]),([0,16]),([0,16]),
           ([0,16])),

          (([0,0]),([0,0]),([0,0]),([0,0]),([0,16]),
           ([0,16]))
     );
}
)

movement = (
     (0, ([0, 16]), (0), (1)),
     (1, ([0, 16]), (1), (0)),
     (2, ([0, 16]), (1), (2)),
     (3, ([0, 16]), (2), (1)),
     (4, ([0, 16]), (2), (3)),
     (5, ([0, 16]), (3), (2))
)
```

Direct Communication Data Transfer

- The openDIEL uses a set of functions for direct communication
- These functions are placed into modules
  - **IEL_send, IEL_recv,** and **IEL_move** transfer data
  - **IEL_send** is nonblocking while **IEL_recv** is blocking; thus, when receiving, a process will wait until data is sent
  - **IEL_move** will both send and receive, making communication synchronous
- These functions require only two parameters: executive info (which is passed as the module's argument) and target process

Jacobi Example: Communication Algorithm



1. Set up matrices on each process
2. Perform calculations on each process
3. Put data into boundary conditions using **IEL_insert_bc**
4. Move data between each set of adjacent processes with **IEL_move**
5. Put data back into matrices using **IEL_copy_bc**
6. Synchronize processes with **IEL_barrier**
7. Repeat for a specified number of iterations

Credit to John Urbanic of PSC for original parallel Jacobi algorithm

Differences Between openDIEL Module and Original MPI Code

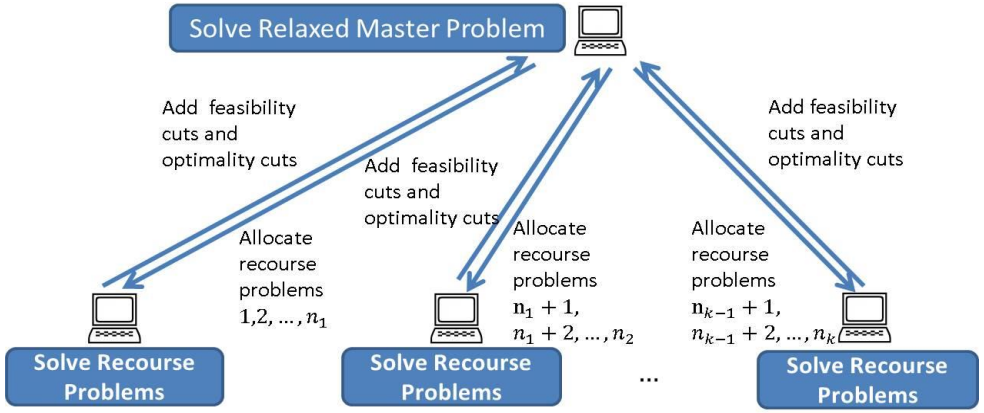
```
int jacobi(IEL_exec_info_t *exec_info)
{
  /* Variable declarations would go here */

  // Set edges & shared boundary conditions
  initialize(t);

  copy_into_bc(t, exec_info);

  /* Calculations code would go here*/

    // Copy values into boundary conditions before communicating
    copy_into_bc(t, exec_info);

    // Exchange data with the process below
    if (myRank != NPES - 1)
      IEL_move(exec_info, myRank + 1);
    // Exchange data with the process above
    if (myRank != 0)
      IEL_move(exec_info, myRank - 1);

    // Place the communicated values back into t
    copy_from_bc(t, exec_info);

    // Synchronize all processes in the IEL
    IEL_barrier(exec_info);
  }

  return EXIT_SUCCESS;
}
```

Done by executive

```
int main( int argc, char **argv ){
  /* Variable declarations would go here */

  /*  Initialize MPI */
  MPI_Init(&argc, &argv);
  /*  Determine size of global communicator */
  MPI_Comm_size(MPI_COMM_WORLD, &npes);
  /*  Determine my rank in the global communicator */
  MPI_Comm_rank(MPI_COMM_WORLD, &mype );

  if ( npes != NPES ){                    /* Currently hardcoded */
    MPI_Finalize();
    if( mype == 0 )
      fprintf(stderr, "The example is only for %d PEs\n", NPES);
    exit(1);
  }

  initialize(t);                    /* Give initial guesss of 0. */

  /* Calculations code would go here*/

    if( mype < npes-1 )
/* Send my data down to the processor below me; Only npes-1 do this */
      MPI_Send(&t[NRL][1], NC, MPI_FLOAT, mype+1, DOWN, MPI_COMM_WORLD);
    if( mype != 0 )
/* Sending my data up to the processor above me ; Only npes-1 do this  */
      MPI_Send(&t[1][1], NC, MPI_FLOAT, mype-1, UP, MPI_COMM_WORLD);
    if( mype != 0 )
/*  Receive new data from UP processor of any source */
      MPI_Recv(&t[0][1], NC, MPI_FLOAT, MPI_ANY_SOURCE, DOWN,
               MPI_COMM_WORLD, &status);
    if( mype != npes-1 )
/*  Receive new data from DOWN processsor of any source */
      MPI_Recv(&t[NRL+1][1], NC, MPI_FLOAT, MPI_ANY_SOURCE, UP,
               MPI_COMM_WORLD, &status);

/*  All processors in the global communicator wait at the barrier */
    MPI_Barrier( MPI_COMM_WORLD );

  }

  MPI_Finalize();       /*  Finalize MPI */
}
```

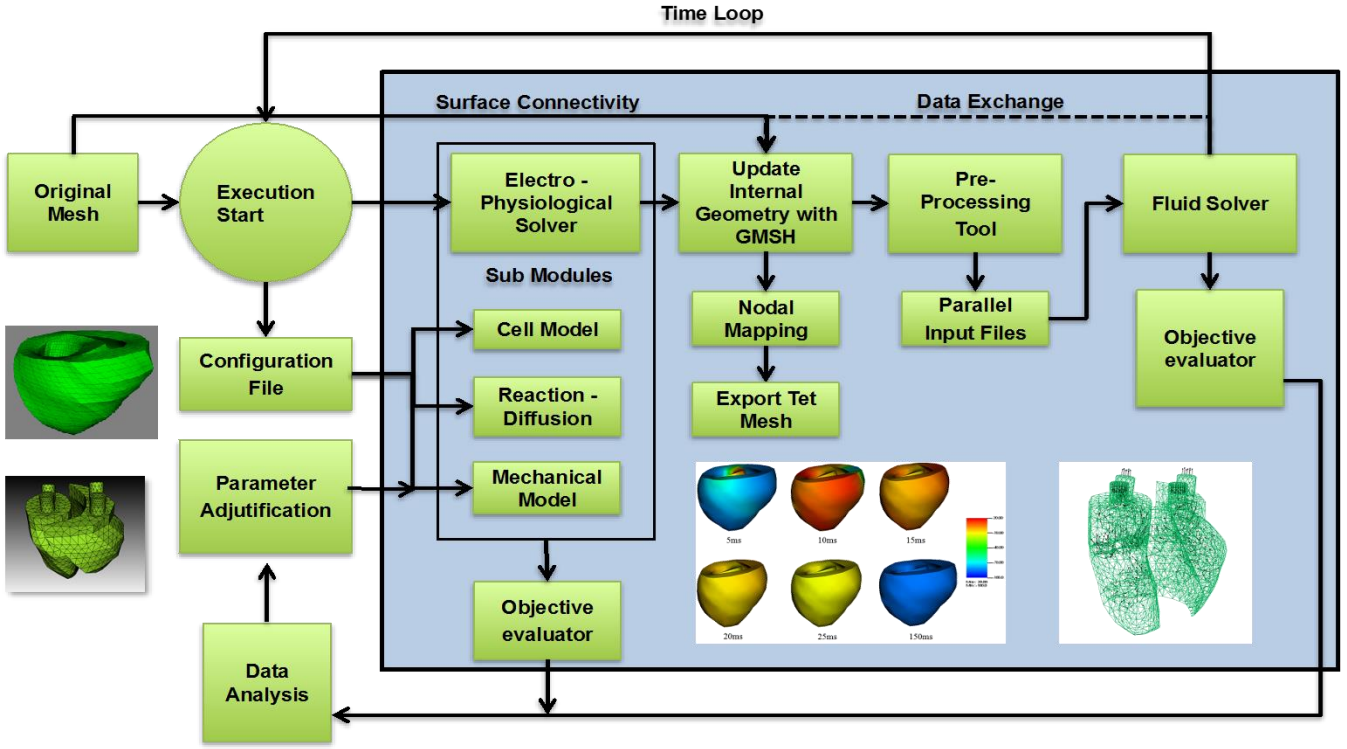
Different Multiphysics Simulations

Stochastic Disaster Planning with Symphony and PSUADE



Agent-Based Modeling with Repast

Cardiac Electrophysiology Modeling



Conclusions

- The openDIEL provides a framework for multiphysics simulations set up in a loosely coupled system of modules.
- The openDIEL is now able to be configured to run concurrently and sequentially via workflow options in the configuration file
- The openDIEL is also now able to communicate large contiguous amounts of data through shared boundary conditions via direct communication functions.

Future Plans for openDIEL

- Adding more user options for direct communications
  - Nonblocking receives, many-to-many communication
- Updating Tuple communication to work with scaling and to cooperate with direct communication
- Expanding Tuple server to contain a makefile-like list of workflow dependencies
- Implementing a working GUI that contains all of the newly implemented features

Acknowledgements

Mentor: Kwai Wong

Additional Assistance: Jason Coan

Original Parallel Jacobi Algorithm: John Urbanic, PSC

NSF

JICS

UTK

ORNL