Workflow and Direct Communication in the Open Distributive Interoperable Executive Library

Tanner Curren, Nick Moran, and Kwai Wong

August 7, 2015

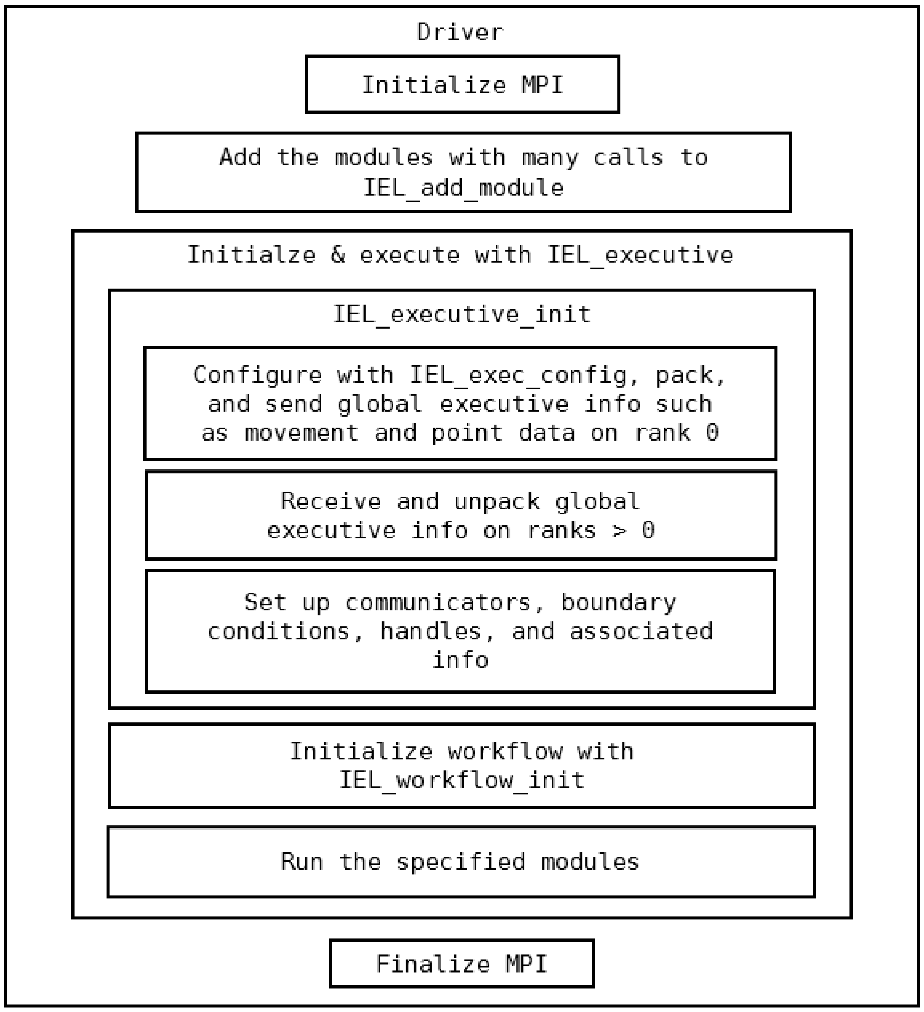
Contents

Chapter 1

Background

The open Distributive Interoperable Executive Library, or openDIEL, aims to allow for the concurrent and sequential operation of many loosely coupled systems. Loosely coupled systems are systems which operate in serial yet may require input data from other systems to operate; these loosely coupled systems, called modules, should depend only on their data input to operate. The openDIEL is a portable framework intended to facilitate the running of these systems and the transfer of their data via a library of MPI-based function calls. For example, the EnergyPlus module calculates the cost of a building's energy consumption given a set of parameters. The output of EnergyPlus can then be used by another module to analyze the energy consumption over different periods of time. If these two modules are run many times concurrently, the outputs can all be sent as input to a third module which applies statistical analysis principles to the data. These modules together form a loosely coupled simulation that the openDIEL might be utilized in.

The openDIEL aims to provide a method of running many of these modules and allowing them to operate in a multi-process environment established by its framework. The openDIEL initializes any associated modules via a configuration file created by the user using a specific template that identifies key operation parameters such as the number of processes to be used and what modules should be run. The framework executes these modules in succession and in parallel as determined by the user in the configuration file. The implementation of the workflow for these modules, which involves setting up the distribution and ordering of the modules, will be described in this report. The library aims to be portable enough to require minimal modification of serial code for implementation of communication functionality. The following diagram details the setup of the various aspects of the openDIEL:


Driver
Initialize MPI
Add the modules with many calls to IEL_add_module
Initialze & execute with IEL_executive
IEL_executive_init
Configure with IEL_exec_config, pack, and send global executive info such as movement and point data on rank 0
Receive and unpack global executive info on ranks > 0
Set up communicators, boundary conditions, handles, and associated info
Initialize workflow with IEL_workflow_init
Run the specified modules
Finalize MPI

The library facilitates data transfer via two major methods: tuple communication and direct communication. These two methods have both advantages and disadvantages that make them particularly situational. The tuple space communication requires an extra process dedicated to maintaining a tuple server, which stores transferred data prior to acquisition by the intended process as well as requests for data to be acquired. The tuple space requires minimal maintenance on the user end, but is not necessarily the most efficient way to transfer large amounts of data since the data must move from the source process to the tuple server and then from the tuple server to the target process. For transferring larger amounts of data, the library provides direct communication, which relies on a set of shared boundary conditions declared in the configuration file as a chunk of data that different processes have access to different sections of. One process can modify data in a shared boundary condition and send this change straight to another process via direct communication, preventing the need to move the data twice as the tuple space does. This report will also examine modifications to a previously implemented method of direct communication.

Chapter 2

Workflow

2.1 Goals

Before this project, the openDIEL did not have a way to specify *how* modules would run in relation to one another. This interaction is the role of the openDIEL's workflow engine. This system will be a comprehensive solution to the to the problem of scheduling modules within the openDIEL, based on both data dependencies and a user-defined workflow. To this end, the openDIEL will include new components to the master configuration file to allow the user to easily specify workflow and data I/O. Before implementation, we will need to decide on an acceptable model for specifying workflow in the openDIEL's configuration file. Then we will need to ensure that this information is readily available through the openDIEL's core data structure (IEL_exec_info_t) This will be implemented through various wrapper functions, extending the configuration file to enable these new options for workflow. The workflow configuration section will be separated from existing configuration options to ensure that existing processes are easy to convert to the new model of openDIEL simulations.

2.2 Implementation

- **Method 1 – IELStartWorkflow()**

  The first implementation of workflow is a wrapper function (IELStartWorkflow()) which replaces the current call to IELExecutive() in the driver of the openDIEL workflow module. This function reads the workflow definition described in the configuration file and generates a series of workflow steps to perform. Each workflow step represented one call to IELExecutive(), the workhorse function of the IEL. Since IELExecutive() requires a configuration file, IELStartWorkflow() generates a temporary configuration file based on the module's being executed in the current workflow step and writes it to disk before each IELExective() call. Since IELExecutive() contains collective MPI calls between all openDIEL-enabled ranks, each module in a workflow step has to completed before starting the next workflow step.

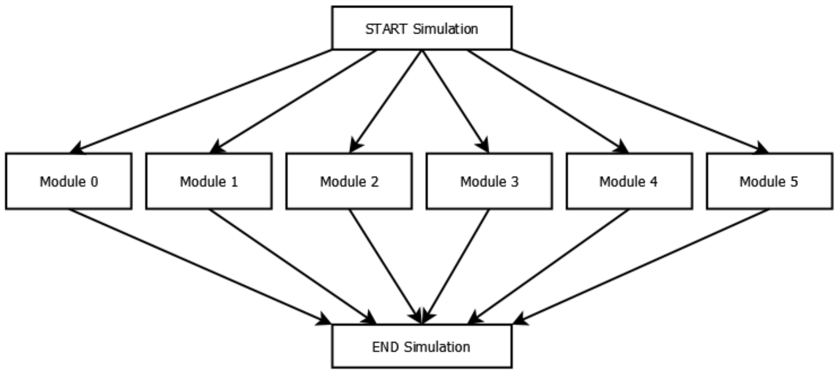
- **Method 2 – IELDispatchModule()/Tuple Server Signaling**

  The next implementation of workflow was restricted wrapper functions. Instead of wrapping around multiple IELExecutive() calls, this method wrapped around the actual module's entry points (the module functions themselves). To achieve this, the IELDispatchModule() was written. This function takes two arguments: the executive info struct and a function pointer. The function pointer is a reference to entry function of the module to be dispatched. IELDispatchModule will then read three additional parameters from the configuration file for each module: signal (integer), wait (integer), and dir (string). The signal/wait system allows users to describe rudimentary dependency rules. The dispatcher system waits for the ID specified in the "wait" parameter before executing a given module. Additionally, the openDIEL will use the "dir" parameter to determine whether a module needs to execute in a separate directory and changes to that working directory, either accompanied by other processes/modules or alone. Once the module has finished, a signal is sent to the next module, allowing it to start.

- **Method 3 – Executive Modification**

  The final implementation of workflow that was decided on is to modify the source code of the openDIEL directly, and change the way module running is handled. In this model, the configuration file was expanded to contain a new section titled "workflow". The workflow options allow users to partition modules into groups of sequentially running modules, which the openDIEL will then start simultaneously. In this way, processes can be reused for different modules and the user is presented with an easy to interpret interface which allows for flexible control over the execution of a simulation.

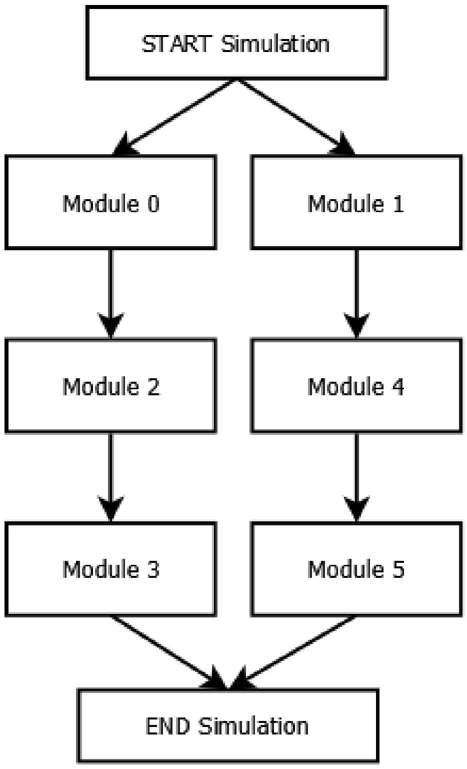
Before the workflow engine, all modules executed by the openDIEL were started at the same time. This flow of execution is as follows:



The workflow module added the workflow section to the configuration file. A full configuration file is shown below. Note the added workflow section.

```
shared_bc_sizes = []
tuple_space_size=0

modules=(
    {
    function="mod0"
    args=(/* Optional Arguments Here */)
    libtype="static"
    library="libmod0.a"
    splitdir="module-0"
    size=1 // can be anything
    points=()
    },
    {
    function="mod1"
    args=(/* Optional Arguments Here */)
    libtype="static"
    library="libmod1.a"
    splitdir="module-1"
    size=1 // can be anything
    points=()
    },
    {
    function="mod2"
    args=(/* Optional Arguments Here */)
    libtype="static"
    library="libmod2.a"
    splitdir="module-2"
    size=1 // can be anything
    points=()
    },
    {
    function="mod3"
    args=(/* Optional Arguments Here */)
    libtype="static"
    library="libmod3.a"
    splitdir="module-3"
    size=1 // can be anything
    points=()
    },
    {
    function="mod4"
    args=(/* Optional Arguments Here */)
    libtype="static"
    library="libmod4.a"
    splitdir="module-4"
    size=1 // can be anything
    points=()
    },
    {
    function="mod5"
    args=(/* Optional Arguments Here */)
    libtype="static"
    library="libmod5.a"
    splitdir="module-5"
    size=1 // can be anything
    points=()
    }
)

workflow:
{

  depend:
  {
  }

  groups:
  {
    group-0:
    {
      order=("mod0", "mod2", "mod3")
      iterations=1
    }
    group-1:
      order=("mod1", "mod4", "mod5")
      iterations=1
    {

    }
  }
}
```

With the new workflow options set in the configuration file, the openDIEL's flow will be reduced to the following.



One advantage is that each group of modules only requires one process, meaning that this simulation will only require two processes to complete as compared to the previous requirement of six processes. Additionally, as is evident in the configuration file example provided above, it is trivial to change the order of executions in a compiled simulation. Using the "iterations" parameter, a user is able to easily change how many times a group of modules executes. The "splitdir" option lets users choose the name of a subdirectory each module will use while execution. The subdirectory name us constructed to be ¡splitdir name¿-¡module rank¿, unless a modules "exec_mode" parameter is set to "parallel", in which case the subdirectory name for all processes is ¡splitdir name¿-0.

Chapter 3

Direct Communication

3.1 Goals

The direct communication portion of the openDIEL is currently intended for the transfer of large chunks of data. Prior to the implementation of the tuple communication, the openDIEL relied on direct communication as its primary format of communication. However, the original implementation of direct communication was poorly optimized for user-end use. This project aims to modify direct communication in a way that makes it both more efficient and more user-friendly. The new method of direct communication should be loosely based on the previously existing method while still making improvements and without modifying the previous method significantly enough to make code designed for the previous method no longer functional.

While the tuple communication is still recommended for the openDIEL's primary method of communication simply because of the lack of overhead required in the configuration file and the ease of implementation into user code, the direct communication should also be a viable and readily user-accessible method of communication. Ideally, the user should be able to choose between the use of tuple communication, direct communication, or both, and use both methods as appropriate for the situation. Thus, direct communication aims to provide an easy-to-use method of transferring data based on a set of shared boundaries that can be used upon the user's discretion along with the tuple communication.

Another goal of the new direct communication method is to rely on shared boundary conditions as parameters for input and output. Tuple communication, while able to utilize the boundary conditions, relies moreso on transferring chunks of data and having the user utilize that data at their own discretion. The new direct communication implementation, however, should handle any overhead dealing with input and output, assuming all inputs and outputs are shared boundary conditions, reducing the amount of buffer modifications that the user has to deal with.

The new direct communication method should be implemented as a set of function calls for the user to implement into code as a method of input/output. These function calls should allow the user to easily send and receive the boundary conditions as well as synchronize processes without having to deal with MPI-related overhead for communication. This implementation should provide the user with the basic utilities of multi-process communication as a baseline, and can be expanded in the future to allow for more flexibility in communication methods.

Additionally, a new configuration option should be added for use in direct communication: movement. By setting up movement parameters, the user should be able to declare where certain boundary conditions are meant to be sent to and from. This concept expands upon the idea of allowing access permissions as the points declaration does, and allows the user more direct control over data flow. Thus, the user will be able to predefine their planned movement and thus have to deal less with defining these movement parameters mid-program, allowing for cleaner adaptations of modules to the openDIEL.

3.2 Implementation

Direct communication is implemented in two ways: through the configuration file and through a set of function calls. The configuration file sets up the sizes and accessibility parameters of the shared boundary conditions to be used by direct communication. The implementation for the configuration setup remains from the previous implementation of direct communication, but has been altered slightly to assign boundary conditions somewhat differently. This new setup effectively grants the user full control over the shared boudnary conditions.

The new communication function calls added for direct communication will now be described, followed by their implementation methodology. These calls take two parameters: a pointer to an executive info structure, which is supplied as the parameter to the module's function and is the integral structure for most IEL operations, and a target process integer that the process will be communicating with. These functions return either an error or a success signifier.

- **IEL_send**: This function sends whatever data is in the shared boundary conditions to the process specified by the target. This call utilizes a standard non-blocking send. First, the function packs any shared boundary conditions that the calling process can access into a contiguous buffer that is set up during the configuration step automatically. This buffer is then sent via MPI_Isend to the target process.

- **IEL_recv**: This function receives whatever data was in the shared boundary conditions of the process specified by the target. This call utilizes a standard blocking receive. First, the function utilizes MPI_Probe to find the size of the boundary condition buffer being sent, and then creates a receive buffer to put the received data in. Then, the function receives the boundary condition data using MPI_Recv, and then unpacks this data into the appropriate positions in the shared boundary conditions based on where both communicating processes can access the shared boundary conditions.

These functions can be called at any time in the module's code, and allow the user to send and receive data at will. A non-blocking send and a blocking receive were chosen because while the sending process can pack and send the data in that order and then continue through other code, the receiving process must unpack the received data directly after receiving it. These functions also are included in yet another wrapper function, which abstracts out the sending and receiving calls when two processes intend only to swap their shared boundary conditions. This function, **IEL_move**, performs an IEL_send call and then an IEL_recv call sequentially on the target process passed as a parameter.

The library also now includes synchronization function calls, which are just wrappers around MPI barrier synchronization calls; these barrier synchronization calls halt the continuation of the calling process until all other processes in the specified MPI communicator also call the function. These functions take an executive info structure as a parameter similar to the data transfer functions described above and return either an error or success signifier. The synchronization functions are as follows:

- **IEL_barrier**: synchronizes all processes in the IEL
- **IEL_module_barrier**: synchronizes all processes in the calling process's module
- **IEL_group_barrier**: synchronizes all processes in the calling process's workflow group

These functions allow the user to synchronize code at will, allowing for more control over communication calls.

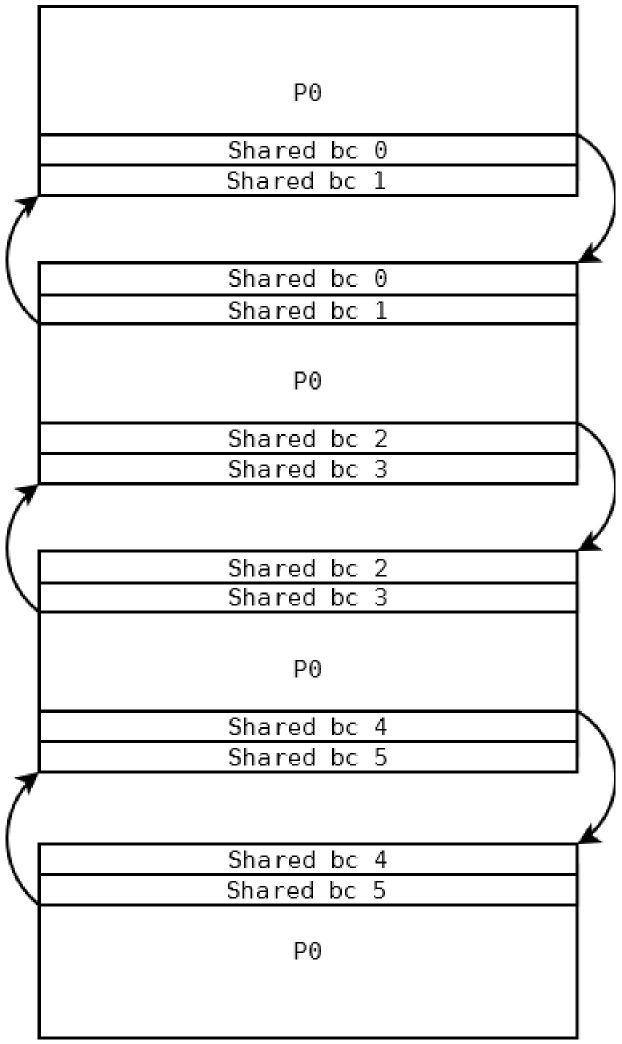
To aid in the transportation of data to and from the shared boundary condition, two new functions for direct communication are also contained in the openDIEL.

- **IEL_insert_bc** allows the user to place data from any buffer into any specified shared boundary condition.
- **IEL_copy_bc** allows the user to copy data from any specified shared boundary condition into any given buffer.

By using these functions, the user can exchange local data with the shared boundary conditions, and more easily facilitate data transfer without having to deal with intricacies of placing the data into the boundary conditions.

3.3 Jacobi Example

A popular example of a communication-based adaption of an algorithm is the Jacobi iteration method to solve a Laplace equation. This algorithm can be adapted to run in parallel by dividing a matrix among multiple processes, with each process performing calculations, communicating the boundaries of the matrix between processes, performing calculations again, and so on [1]. By adapting this algorithm to work with the openDIEL's direct communication, this method provides a usable example for direct communication. The following diagram represents the grids on each process with the shared boundary conditions:



We can see from this diagram that only certain arrays need to be communicated; these arrays can be declared as shared boundary conditions by the configuration files, then we can fill them with values from the matrix using IEL_insert_bc for sending and place values back into the grid using IEL_copy_bc after receiving. This simulation presents a concrete example of the use of boundary conditions in communication, and a situation in which direct communication would be appropriate.

Chapter 4

Future Plans

The openDIEL's workflow and direct communication now provide basic functionality, but we hope to improve these aspects in the future.

The task of scheduling workflow will be handed over to the tuple server to allow for the dynamic scheduling of modules. The tuple server running on MPI rank 0 (openDIEL Command Server) will operate in conjunction with the other openDIEL processes in a server-client relationship, such that the command server will determine (from a list of user defined dependencies) which modules are needed to run, and communicate the necessary information to any non-occupied worker processes. Worker processes will communicate information about the current job status back to the command server when they have finished executing.

In addition to dynamic task scheduling techniques, advanced data I/O will be integrated into the openDIEL as a part of the workflow engine. While the openDIEL includes a plethora of interprocess communication functions (Tuple Communications, Direct Communications), the user is still 100% responsible for file I/O. In the future, we will provide file I/O handlers which, based on the configuration file dependencies, can utilize high performance file I/O systems such as HDF5 to optimize data transfers.

We plan to improve direct communication by implementing different send and receive calls, such as possibly a blocking send and a nonblocking receive, so that we may provide a wider range of use for the user to cover any potential situation in which these calls might be necessary. We also plan to implement different data storage options into direct communication, such as HDF5.

[1] Urbanic, John, 2011: Laplace MPI Template Version. Pittsburgh Supercomputing Center.