OpenDIEL:
A Parallel Open Source Workflow Engine

Tristin Baker, Jordan Scott, Zachary Trzil

Advisor: Dr. Kwai Wong

Contents

Error! Bookmark not defined.
1. Introduction 1
1.1 Prior Work 4
1.2 Initial Work 4
2. Direct Communication 5
2.1 Introduction 5
2.2 Improvements 5
3. Tuple Space Communication 8
3.1 Introduction 8
3.2 Improvements 8
3.2 Results 10
4. Graphical User Interface 12
4.1 Introduction 12
4.2 Layout 12
4.3 Improvements 15
5. Conclusion Error! Bookmark not defined.

1. Introduction

The OpenDIEL (Open Distributive Interoperable Executive Library) has the goal of being able to run multiple loosely coupled systems concurrently. A loosely coupled system, while being serial, is a system which requires data from other systems in order to operate. OpenDIEL aims to give users a simplified method to facilitate this communication by converting individual loosely coupled systems into OpenDIEL modules. These modules can then communicate directly or by using the OpenDIEL tuple server. Module creation is done automatically by the OpenDIEL, so all the user has to do is specify which systems he or she wants to run, where they are located, and add the proper OpenDIEL function calls to their code.

The OpenDIEL uses a workflow configuration file that the OpenDIEL Driver reads. This driver calls the IELExecutive function, which is the main function for the OpenDIEL. IELExecutive reads in the data from the workflow file and uses this data to populate the IEL_exec_info_t struct. This data structure stores all of the needed data for the modules to be able to communicate. The OpenDIEL Driver will facilitate the execution of each module and continue doing so until each module has ran the amount of times specified within the workflow configuration file.

OpenDIEL uses Message Passing Interface in order to exchange data between different modules. OpenDIEL has several functions that are wrappers around the standard MPI function calls, such as IEL_Barrier, or IEL_bc_get and IEL_bc_put. MPICH is the specific implementation of MPI that the OpenDIEL uses.

Below is a diagram of how the OpenDIEL workflow is laid out.

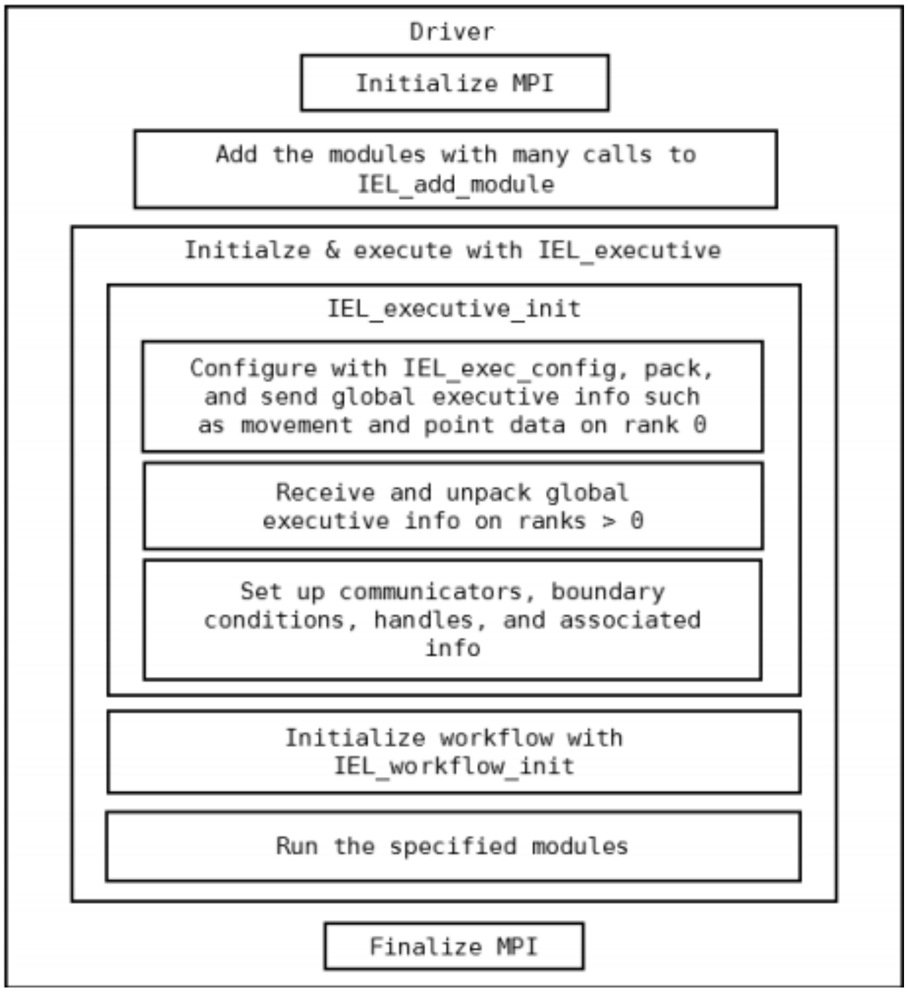

Driver
Initialize MPI
Add the modules with many calls to IEL_add_module
Initialze & execute with IEL_executive
IEL_executive_init
Configure with IEL_exec_config, pack, and send global executive info such as movement and point data on rank 0
Receive and unpack global executive info on ranks > 0
Set up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

Figure 1.a

There are two methods the modules can use to communicate data between one another, direct communication and tuple space communication. Direct communication gives each module a shared_bc array based on the num_shared_bc variable's size. This variable is specified within the workflow configuration file. The modules then are given indices within the shared_bc array in which they can read and write from, which are stored in the module's shared_bc_read and shared_bc_write variables, which are located in the module_depend_t data structure. These modules can then use IEL_bc_put or IEL_bc_get in order to communicate their data to one another.

The other form of communication is tuple space communication. The tuple space communication is indirect and nonblocking. This allows processes to send information regardless of the status of the receiving process, enabling the sending process to continue working instead of blocking. The tuple server operates by listening for any MPI_Send/Recv calls by calling MPI_Probe in a continuous loop. The data being exchanged with each MPI_Send/Recv call that the server detects is dealt with appropriately by storing, reading, or reading and deleting the data based on the tag used in the MPI_Send/Recv function. The data is stored in a red-black tree with each node in the tree corresponding to each unique tag. Each node contains a queue of the waiting messages. Two functions, *tput(...)* and *tget(..)* have been implemented as wrappers around the MPI_Send and MPI_Recv calls respectively. These functions allow you to specify the size of the data, a data tag, the server to send or receive from, and whether or not to delete the data (*tget* only).

The ultimate goal of the OpenDIEL is to be able for all projects in the The Research Experiences in Computational Science, Engineering, and Mathematics (RECSEM) REU to be able to run their projects through the OpenDIEL itself. This goal has not been realized yet, but we are steadily making improvements in order to see the goal as a reality.

1.1 Prior Work

The OpenDIEL has been under continuous development for many years and has seen multiple developers. During the last three years, a vast amount of work has gone into improving the existing systems the OpenDIEL already had implemented. The previous developer, Tanner Curren, worked from June 2015-November 2016 to greatly improve the workflow capabilities as well as completely revamp the direct communication. In 2014-2015 Jason Coan implemented the first iteration of tuple space communication.

1.2 Initial Work

When we first started working on the code, we noticed that there was a large portion of the code which was either poorly documented, or not documented at all. There was also quite a bit of legacy code which was no longer used or deprecated by other newer functions that had been implemented. This made navigating the code extremely confusing to sort through and understand what was going on under the hood. We decided that a good place to start was to write some real documentation, figuring that it would not only help us understand what was going on, but also any future developers of the OpenDIEL. The layout is fairly simple. The user can select whether they want to read the *comm* or *exec* side of the code, and then from there, select which function or data structure they want to know more about. From these pages, the user can see where the function is located, a description of the function, which functions the function calls, what the important variables are, and whether or not the function is completed, in progress, planned, or deprecated. This documentation is not complete yet, but is planned to be finished.

2. Direct Communication

2.1 Introduction

The OpenDIEL direct communication allows for modules to directly share data between them by using the shared_bc. The modules are given the indices where they can read from and write to by the user in the workflow configuration file. These indices can be accessed by using the IEL_bc_put and IEL_bc_get functions. The function prototypes are listed below.

```
int
IEL_bc_put(IEL_exec_info_t * exec_info,
                    char * module,
                    MPI_Request &request);
int
IEL_bc_get(IEL_exec_info_t * exec_info,
                    char * module,
                    MPI_Request &request);
```

These functions take the IEL_exec_info_t struct from the current module to get the indices where a module can read from and write to, the module to which the data will be read from or written to, and the MPI_Request, so MPI knows which modules need to be ready.

2.2 Improvements

Previously, the direct communication examples included in the OpenDIEL tarball did not show any extensive proof that the direct communication functions work. To do this, we decided to use an MPI example we were given and port it to the OpenDIEL. This example used Laplace Transformation Matrices to show how MPI_Send and MPI_Recv work. In the OpenDIEL implementation, three separate modules are used to spread the work out evenly. The shared_bc_write and shared_bc_read's are set in the workflow configuration file. Then in the modules, once the calculation has completed, each one will share its data by calling,

```
IEL_bc_exchange(exec_info, module, MPI_request);
```

and then use this received data to do the next round of Laplace Transformation calculations. This new example more adequately shows a new user of OpenDIEL how the direct communication works.

In the current implementation, a user will specify where each module can read and write from in its copy of the shared_bc array. This proves to be a waste of memory and is not scalable by any reasonable means. For example, if the num_shared_bc array is 10000 elements long, but a module only accesses 500 elements of its copy, there would be 9500 elements left unused, resulting in a big waste of memory. This problem only grows as the num_shared_bc and the number of modules grow, so it was determined that this needed to be changed.

A diagram of this model is shown below.

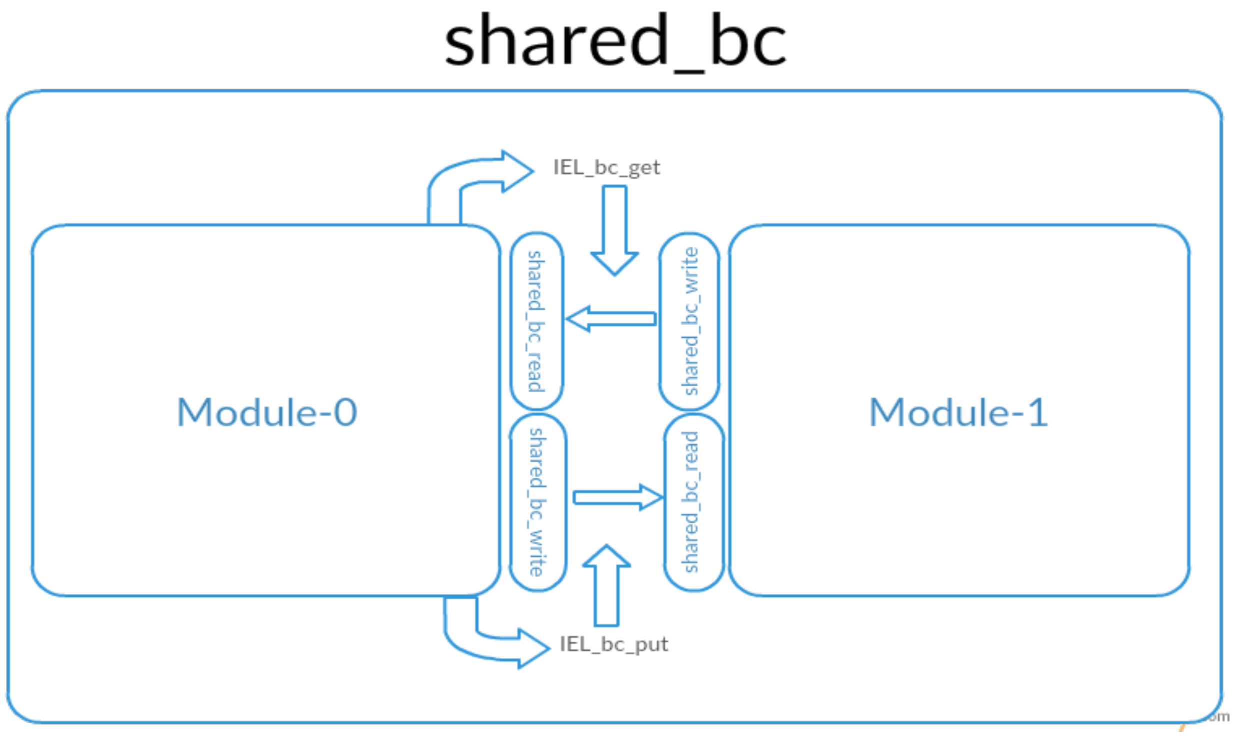


Figure 2.2.a

To solve this problem, we decided that each module would need to only have the exact amount of space they need for their calculations. Not only would this be better for saving memory, but it would also provide the user with a much simpler method of setting up their serial code to work as an OpenDIEL modules. For example, in the current implementation, if the shared_bc contains 2000 elements and there were two modules, each able to read and write 1000 elements, the user would need to set each and every element manually, usually with a for loop,

```
for(i = 0; i < 1000; i++) {
        shared_bc[i] = //value
}
```

and then call IEL_bc_put and IEL_bc_get to exchange the data as needed. The user also needs to specify that the one module can write to another module at specific indices, but also make sure that the second module can read from those indices. The same applies the other way around. Using the new method, the user would be able to call IEL_bc_copy,

```
IEL_bc_copy(/* data */, /* module to send to */)
```

which will copy the relevant data to the predefined readable section of the receiving module. This new method would improve the user experience drastically over what is currently implemented, but would also improve the speed of the OpenDIEL's direct communication and memory management.

A diagram of this model is shown below.

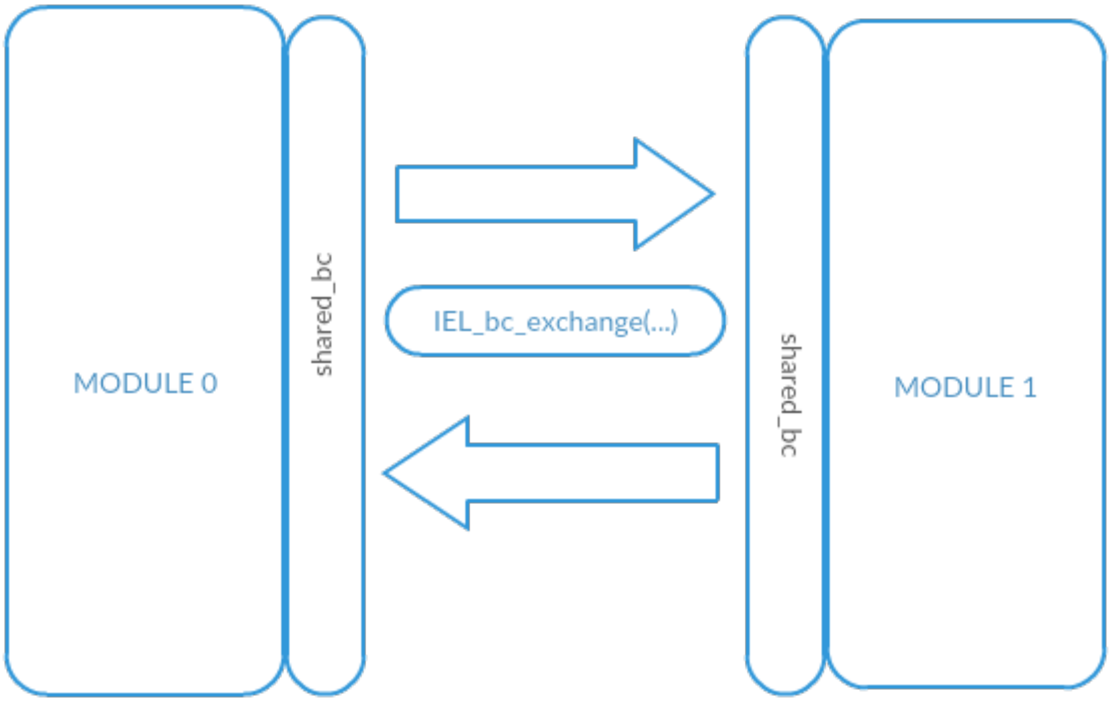


Figure 2.2.b

While work on this new direct communication method was attempted, it was determined that more work than was previously thought would be required due to the way the configuration file parser that is used, libconfig, is implemented with OpenDIEL. This problem was realized late into the research experience, and so not enough time was left to make a fully functional version of this implementation. Work on the improved direct communication will need to continue at a later date to fully realize this new implementation.

3. Tuple Space Communication

3.1 Introduction

The tuple space communication, like the direct communication, has been worked on extensively by the previous long-term developer. OpenDIEL was designed from the outset with the goal of being able to run multiple tuple servers. Using multiple tuple servers affords two important benefits to OpenDIEL. Distributing data across a set of tuple servers in an HPC environment may allow for tuple servers to run on multiple nodes, taking advantage of memory in each node. This vastly increases the working memory accessible by OpenDIEL and limits the amount of data that needs to be written to files; greatly decreasing running times. The other potential benefit is data resiliency. Long-running programs that depend on data from previous steps are common in HPC environments. It is imperative that programs be able to recover from system crashes without having to restart from the beginning. Using multiple tuple servers can allow for successful accessing of data should a single node become unavailable. Tuple servers can also be configured to backup data to disk at regular intervals and when they are not actively being used. This would allow for data to be saved at checkpoints, allowing for quick recovery with minimal performance penalties.

The existing code allowed for integrating the functionality of using distributed tuple servers with minimal changes to the core code. This allows for a user to chose to run a single server exactly as before or, if needed, to elect to use multiple servers. The existing tuple server code used MPI_Probe() with the MPI_ANY_TAG tag to listen for and intercept any MPI_Send/Recv call. Early in development, it became clear that using the MPI_ANY_TAG with multiple tuple servers was causing it to detect errant MPI_Send/Recv calls, causing a fatal MPI error as the data it detected often did not fit within the allotted buffer. The most frequent tag picked up was the RUN_READY tag. This indicated that an already initialized server was detecting MPI_Send calls used to indicate the status of a server during initialization. Changing the tag that MPI_Probe() listened for to SERVER_SEND, a tag used to indicate that the server is communicating via MPI_Send, fixed this issue. The only other change to the original code that was needed in order to develop a working prototype of the distributed tuple server communication was to make a call to an initializer function if multiple tuple servers were requested.

3.2 Improvements

The distributed array of tuple servers leverages the existing tuple server code, running the tuple server as a module for each server in the distributed array. Two out of the total number of tuple servers will be reserved. One of the reserved instances of the tuple server will act as a metadata server. For each communication, two arrays will be stored on the metadata server, one containing the rank of the server that each piece of data is stored on and the other containing the corresponding size of that piece of data. These are placed on the metadata server using the existing *tput* function. The other reserved tuple server calls an init function to initialize the metadata server and is reserved for any managerial tasks that may be needed. The framework for the distributed tuple server is implemented in the following functions:

- *int* manager_init (*int* number_of_servers),
- *int* IEL_dist_tput(*size_t* size, *const char* *tag, *void* *data),
- *int* IEL_dist_tget(*size_t* *size, *const char* *tag, *unsigned char* del, *void* **data),
- *int* get_server_rank(), and
- *unsigned long* get_hash(*const char* *str)

The *manager_init* function initializes a struct that contains the number of tuple servers being used and the rank of the last tuple server that was used to store data. It stores this struct on the metadata server with a tag of 0.

The *IEL_dist_tput* function is used to store data on the distributed array of tuple servers. When this function is called, the info struct is retrieved from the metadata server to initialize the arrays of metadata to be the correct size – the size is the number of tuple servers that exist minus the two used for management purposes since the data will be spread evenly across all existing tuple servers. The data being sent is then chopped into pieces and sent to each server and the corresponding indices of the meta data arrays are updated. The server that each piece of data is sent to is determined by the *get_server_rank* function. After all the data has been placed, the metadata server is then updated with the two arrays. These are stored at the same tag and the negative of the tag used in the *IEL_dist_tput* call to allow for easy lookup when the data is retrieved.

The *IEL_dist_tget* function is used to retrieve data stored on the distributed array of tuple servers. The function first pulls the meta data arrays from the metadata server, stored at the tag passed to *IEL_dist_tget*. The sizes of all the pieces of data are totaled up in order to allocate space for the array. The function then steps through the array of server ranks, and requests the data at each server with the corresponding tag. The TUPLE_TAKE or TUPLE_READ tags are used for reading and deleting from the server or reading and keeping the data on the server respectively. Keeping track of the increasing total size of the data as it is retrieved allows the array of data to be rebuilt properly. After the data is reassembled, if the user elected to delete the data from the servers, the corresponding meta data is also deleted.

The *get_server_rank* function simply returns the rank of the next server to use. It does this by pulling the info struct stored on the metadata server (deleting it from the server), checking the value of the last_used_server variable that it contains, updating the value by incrementing it, and returning this incremented value after placing the struct back onto the metadata server. This algorithm results in a simple round-robin access of each available tuple server.

The *get_hash* function is simply an implementation of the DJB2 hash function. This hash function was used in order to allow the users to uniquely tag their data with strings for increased legibility. This hash function was chosen for its qualities of being both simplistic to implement and and its computationally efficient design. It generates an unsigned long hash value, but the tuple server takes an integer as its tag. To minimize potential issues, the hash value returned as an unsigned long then the *bitwise and* of the hash value and INT_MAX is taken to mask off the higher order bits of the hash value, leaving only the bits that will fit into an integer.

Currently no collision resolution strategy has been implemented since the DJB hash function is quite good at randomization and uniformity. Further, the strings given to the hash function are user-defined, so it does not need to account for arbitrary strings colliding as the user can define strings that are known not to collide (very few strings of common words are known to collide and lists of these strings are readily available). If a collision resolution strategy is needed in the future, it can easily be added.

3.2 Results

A simple example was developed to test, debug, and benchmark the distributed tuple server implementation. Two modules, hello1 and hello2 were created. Hello1 simply creates an array of a user-defined number of random doubles. It then uses *IEL_dist_tput* to place the data on the tuple servers. Hello2 then calls *IEL_dist_tget* to pull the data from the tuple servers. These were then benchmarked for several dataset sizes. To compare against file I/O, the array of doubles was created in the same manner and written to a file. To compare against a single tuple server, the existing IEL_tput() and IEL_tget() functions were used to place the data on a single server. A script was developed to run each test fifty times so that an average could be obtained.

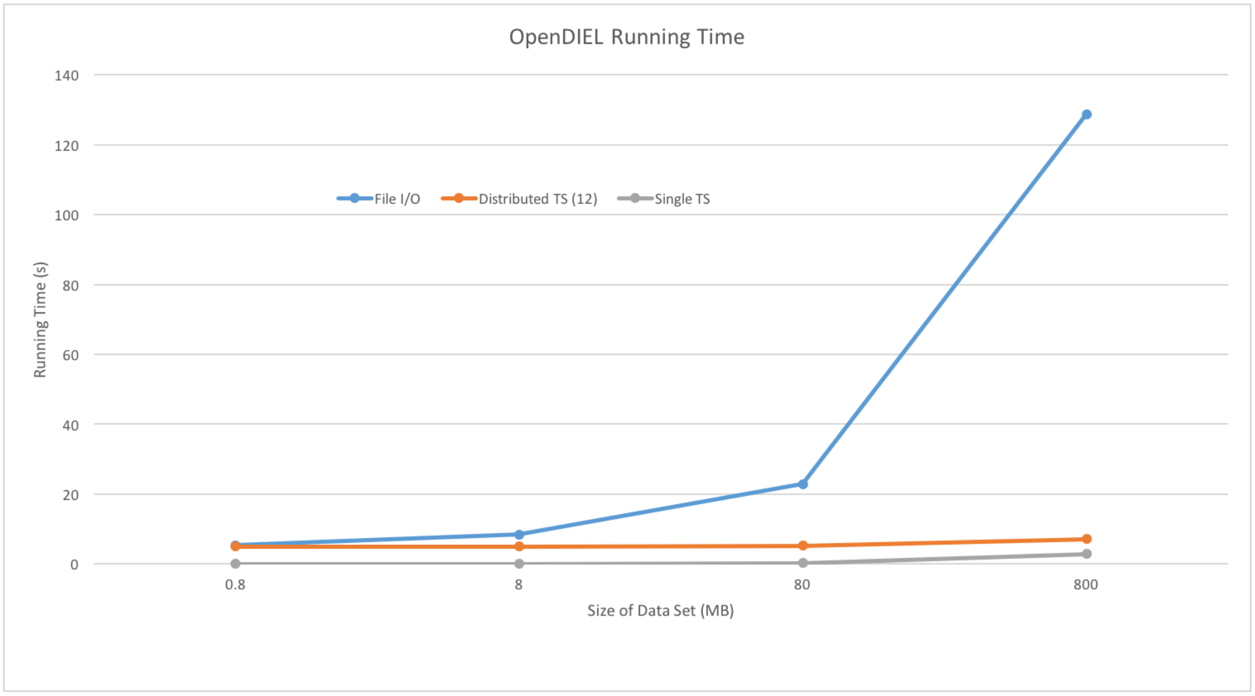


Figure 3.2.a

The results show a nearly constant running time for the single and distributed tuple servers while the file I/O increases exponentially with size, as expected. The mean running time of the largest data set, 800MB, using file I/O as the method of data transfer was 128.77 seconds over fifty runs with a standard deviation of 2.96 and a median of 127.27 seconds. The mean running time of the distributed tuple server array, using the same size data set and also averaged over fifty runs was 7.15 seconds with a standard deviation of 0.0045 and a median of 7.15 seconds. Using a single tuple server to facilitate communication resulted in slightly reduced mean running time of 2.89 seconds with a median of 2.88 and standard deviation of 0.09925. A table of all results can be found in Appendix A. All tests were run on the Comet supercomputer, using an interactive node consisting of 24 cores.

The difference in the running time of the single and distributed tuple server can be attributed to the time needed to initialize the extra processes through openDIEL. This is made evident by the fact that the difference in running time between the distributed and single tuple server tests is constant across data sizes and that the distributed tuple server and file I/O tests take roughly the same time to run -- while initializing 14 total process -- where the running time is dominated by setting up the program instead of by computation or data transfer.

4. Graphical User Interface

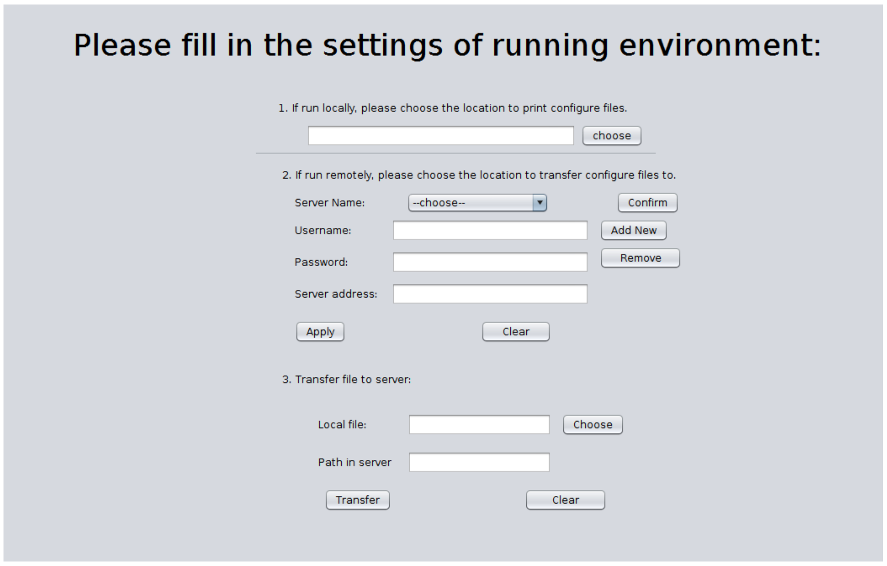
4.1 Introduction

The OpenDIEL Graphical User Interface (GUI) began development in 2015 and was paused until the summer of 2017. As of May 2017 the GUI had several thousand lines of compilable code and provided a suitable skeleton to be developed on. The purpose of the OpenDIEL GUI is to allow users to modify their code to use OpenDIEL functions, create modules/groups/sets graphically, and execute their code. The user should be able to do this without ever having to see any code themselves. The GUI aims to be a self-contained program that simplifies the user's job so that they can run their code with ease. The user can either have their code located locally or can sign in via SSH to a machine and modify it remotely.

4.2 Layout

The GUI has several different tabs to assist the user with ease of use. When a user first launches the GUI, they are greeted with the first tab, an introduction screen, which allows them to specify where their code is located, either local or remote. This tab is not currently fully implemented, but works with test cases. The second tab is a notepad. This tab allows for user to jot down any notes they may have during setup, execution, or results. This notes then can be saved to the current working directory and loaded again should the user close and reopen the GUI.

Introduction Tab



Please fill in the settings of running environment:

1. If run locally, please choose the location to print configure files.

2. If run remotely, please choose the location to transfer configure files to.

Server Name: --choose--

Username:

Password:

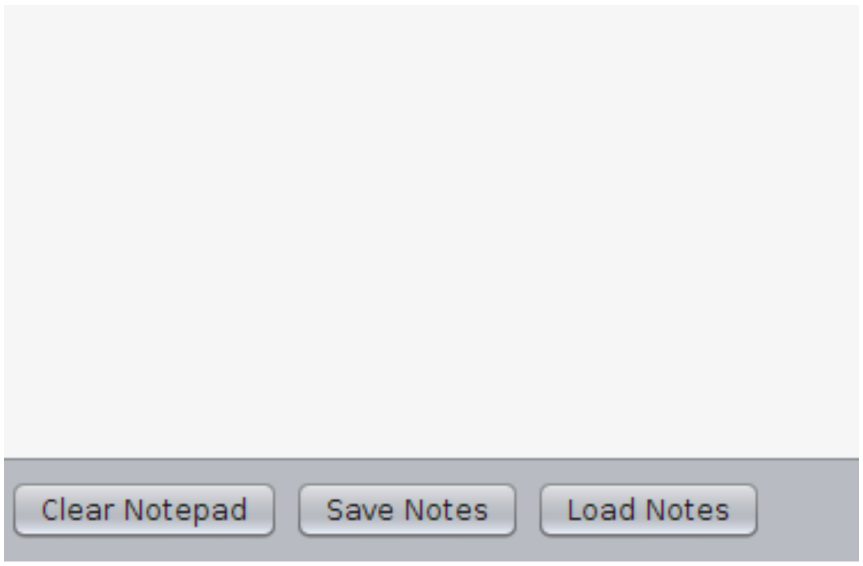
Server address:

3. Transfer file to server:

Local file:

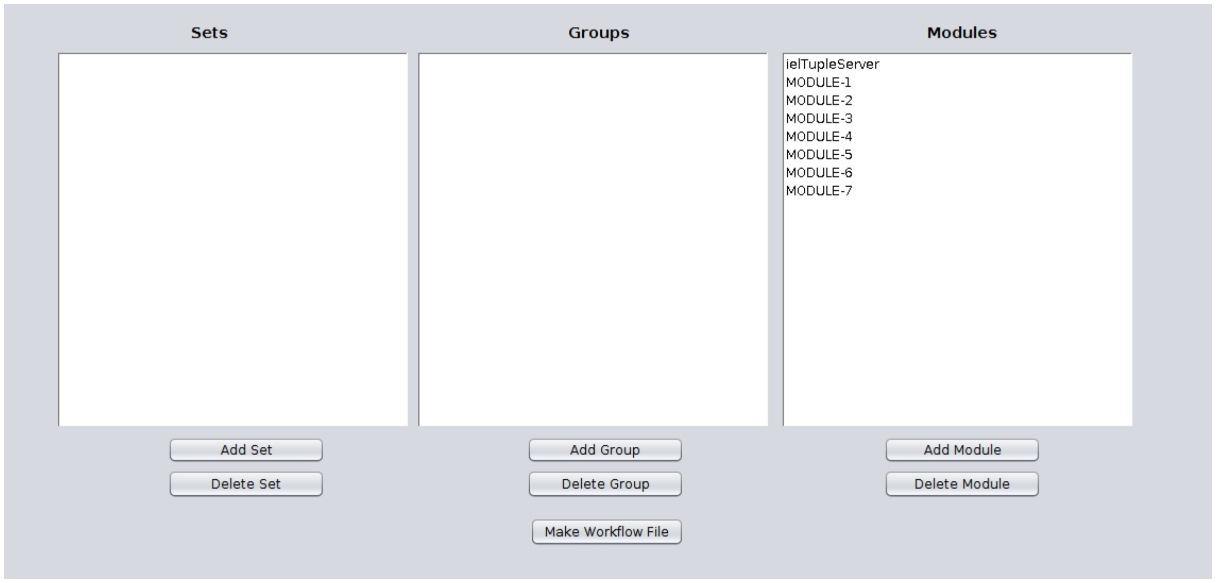
Path in server

Figure 4.1.a



Notepad Tab
Figure 4.2.b

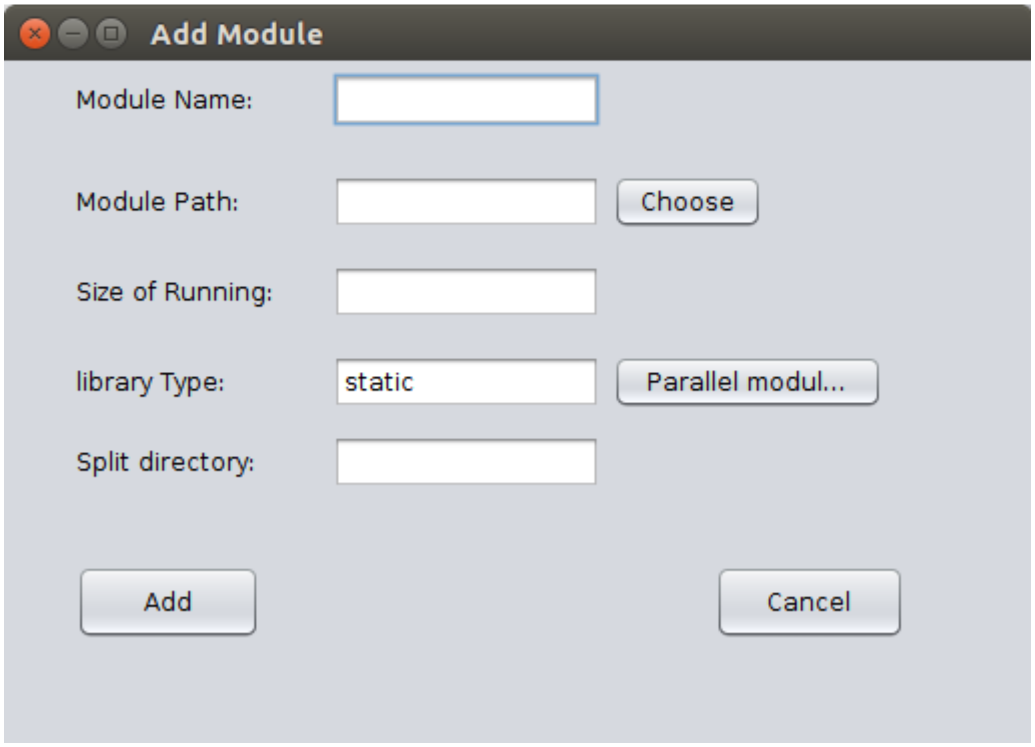
The third tab holds one of the most important functions of the GUI, the workflow configuration file creation. In this tab, users can create modules, groups, and sets that are specific to their projects. The modules can go into groups in any order the user specifies, and this also works with the groups going into sets. When creating a module, the user is prompted to enter any information the workflow file needs to know about the module, such as its name, location, and its size. This is similar to the way groups and sets are created. Once the user is satisfied with the modules, groups, and sets, they can click the “Make Configuration File” button, which automatically generates the workflow file the OpenDIEL Driver needs to function correctly.



ielTupleServer
MODULE-1
MODULE-2
MODULE-3
MODULE-4
MODULE-5
MODULE-6
MODULE-7

OpenDIEL Tab

New Module Window



Module Name:

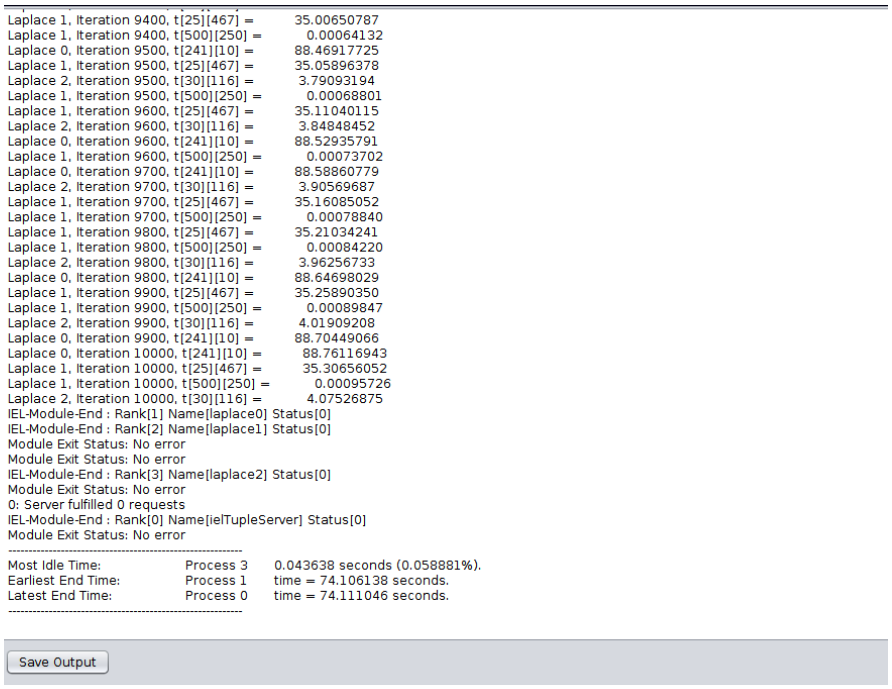
Module Path:

Size of Running:

library Type: static

Split directory:

The fourth tab is equally important. After a user runs their code, the results will be displayed to this tab. From here, the user can examine the results or choose to save them to examine at a later time.



```
Laplace 1, Iteration 9400, t[25][467] =        35.00650787
Laplace 1, Iteration 9400, t[500][250] =        0.00064132
Laplace 0, Iteration 9500, t[241][10] =        88.46917725
Laplace 1, Iteration 9500, t[25][467] =        35.05896378
Laplace 2, Iteration 9500, t[30][116] =         3.79093194
Laplace 1, Iteration 9500, t[500][250] =        0.00068801
Laplace 1, Iteration 9600, t[25][467] =        35.11040115
Laplace 2, Iteration 9600, t[30][116] =         3.84848452
Laplace 0, Iteration 9600, t[241][10] =        88.52935791
Laplace 1, Iteration 9600, t[500][250] =        0.00073702
Laplace 0, Iteration 9700, t[241][10] =        88.58860779
Laplace 2, Iteration 9700, t[30][116] =         3.90569687
Laplace 1, Iteration 9700, t[25][467] =        35.16085052
Laplace 1, Iteration 9700, t[500][250] =        0.00078840
Laplace 1, Iteration 9800, t[25][467] =        35.21034241
Laplace 1, Iteration 9800, t[500][250] =        0.00084220
Laplace 2, Iteration 9800, t[30][116] =         3.96256733
Laplace 0, Iteration 9800, t[241][10] =        88.64698029
Laplace 1, Iteration 9900, t[25][467] =        35.25890350
Laplace 1, Iteration 9900, t[500][250] =        0.00089847
Laplace 2, Iteration 9900, t[30][116] =         4.01909208
Laplace 0, Iteration 9900, t[241][10] =        88.70449066
Laplace 0, Iteration 10000, t[241][10] =        88.76116943
Laplace 1, Iteration 10000, t[25][467] =        35.30656052
Laplace 1, Iteration 10000, t[500][250] =        0.00095726
Laplace 2, Iteration 10000, t[30][116] =         4.07526875
IEL-Module-End : Rank[1] Name[laplace0] Status[0]
IEL-Module-End : Rank[2] Name[laplace1] Status[0]
Module Exit Status: No error
Module Exit Status: No error
IEL-Module-End : Rank[3] Name[laplace2] Status[0]
Module Exit Status: No error
0: Server fulfilled 0 requests
IEL-Module-End : Rank[0] Name[ielTupleServer] Status[0]
Module Exit Status: No error
------------------------------------------------------
Most Idle Time:            Process 3     0.043638 seconds (0.058881%).
Earliest End Time:         Process 1     time = 74.106138 seconds.
Latest End Time:           Process 0     time = 74.111046 seconds.
------------------------------------------------------
```

Output Tab (Post execution)

In order to use the OpenDIEL framework, source code must be translated into an OpenDIEL module. The ModMaker python package included in OpenDIEL allows users to input a C, C++, or Fortran file directory, and output a useable module. This package is being added the GUI currently. When fully operational, the user will be able to select a file from within the GUI, select to run the ModMaker package, and have the new module file in the same directory. When the file has been transformed the user will also have the ability to view and edit the OpenDIEL module from within the GUI.

4.3 Improvements

Over the course of the summer, several improvements were made to the GUI. These changes were made to improve the user experience and to make it so the user had even less work to do before they could get their code compatible and running with OpenDIEL. The OpenDIEL tab saw the most improvement in the GUI. When we first got the code, the way this tab was laid out was quite confusing and not very intuitive. Now, a user can add a new module and drag it to any group they want, and then reorder the modules in any order they want. This also applies to the groups into sets. Not only is this more intuitive for the user, but it also helps any future developer as it cleans up a lot of unnecessary code. Along with cleaning up unnecessary code, we also went about renaming certain GUI elements. Previous developers had used very arbitrary names for GUI elements, such as *JPanel1,* or *JButton3*. Obviously, this makes for a confusing and frustrating time in trying to figure out which GUI element does what. Instead, we adopted a naming convention that allows future developers to know exactly which element does what. For instance, *JPanel1* and *JButton3* might be renamed *ModulePanel* or *AddModuleButton*, respectively.

In addition to improving user and developer experience, we also have added new functionality to the GUI. One very important new feature is an execution button. This button allows for the user to locate where their OpenDIEL Driver is. This folder is also where the runscript for their project will be, which will eventually be generated by the GUI. The GUI will then run the user's project. This is facilitated by Java's *Process* and *ProcessBuilder* classes. These classes will use the currently set path and a set of instructions to execute a program through the shell. The output of this execution will go directly to the GUI's Output tab, as explained above. See Appendix B. for a code snippet showing how the *Process* and *ProcessBuilder* works with the Laplace Direct Communication example listed above.

This function will execute the user's runscript, which, again, will be generated by the GUI and placed in the folder containing the OpenDIEL driver. The *readFromExecution()* function simply reads the output from the *Process* and writes it to the GUI's output tab. This proves handy for the user, as they are able to save the results in order to view or compare them later. An improvement to this that could be made is in the Output tab. Currently, when the IELExecutive is running, the output panel shows the output until the *JTextArea* that the output gets printed to in the Output tab. However, if the output is longer than the length of the *JTextArea*, the output still gets printed, but the user cannot see the results until after the execution is completed. This can be solved by using multithreading, however, in its current state, this was considered a future goal due to the scope of the problem.

Appendix

Appendix A.

Running Time of Two Modules Using File I/O			
Size of Data Set	Mean (s)	Median (s)	Standard Deviation
800MB	128.773	127.272	2.958
80MB	22.877	19.964	5.560
8MB	8.380	6.597	3.586
0.8MB	5.337	5.336	0.0018
0.08MB	5.214	5.212	0.00357

Running Time of Two Modules Using Distributed Tuple Communication			
Size of Data Set	Mean (s)	Median (s)	Standard Deviation
800MB	7.153	7.153	0.00445
80MB	5.227	5.227	0.00112

8MB	5.029	5.028	0.00060
0.8MB	5.007	5.007	0.00034
0.08MB	----	-----	-----

Note: These values were obtained by running a total of twelve tuple servers, leaving ten tuple servers to distribute the data across.

Running Time of Two Modules Using Single Tuple Server Communication			
Size of Data Set	Mean (s)	Median (s)	Standard Deviation
800MB	2.894	2.878	0.0993
80MB	0.295	0.289	0.00581
8MB	0.0302	0.0296	0.00166
0.8MB	0.00405	0.00404	0.00004
0.08MB	0.00107	0.00104	0.00012

Note: The above data was collected on a different day from the previous two tables.

The results contained within this appendix were obtained by running each test on an interactive node on the Comet supercomputer a total of 50 times. It should be noted that even though these tests were run on an interactive node and precautions were taken to ensure that the testing environments were as identical as possible, tests run on different days differed in running time by as much as 38 times in some cases. These results should not be used to, and are not intended to gauge absolute running times of openDIEL. Instead, these times indicate relative differences in configurations of openDIEL with regards to tuple server configuration and file I/O.

Appendix B.

```
private void runIconActionPerformed(java.awt.event.ActionEvent evt) {

        String[] args = new String[] {"/bin/sh",
                                      "-c",
                                    "./runscript"};
    String line;
    Process p;
    try {
      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!");

    } catch (IOException ex) {
     Logger.getLogger(BuildGUI.class.getName()).log(Level.SEVERE,
                                                                        null,
                                                                            ex);
    }
}
```