Interfaces for openDIEL

Efosa Asemota, Frank Betancourt, Quindell Marshall
Mentor: Dr. Kwai Wong

August 3, 2018

Contents

1 Introduction

Open Interoperable Distributive Executive Library (openDIEL) is a workflow engine designed for usage in high performance computing environments. It consists of a set of C code, combined with openMPI functions to unify many different modules of computation under a single executable that can then be run in an MPI environment.

The user specifies a list of modules, and defines a workflow in a Configuration File. The user then creates a driver that reads in the configuration file, calls MPI_Init(), and calls IEL member functions to start modules and run the specified workflow. The driver can then simply be run with mpirun.

1.1 Workflow Configuration Files

At the center of openDIEL is the Workflow Configuration File. The file is divided into two different sections: module specification, and workflow specification.

The first section of the configuration file is module specification. This is where you are able to specify exactly what they want to run, with details such as the number of MPI processes, the number of openMP threads to create, and the number of GPUs the module will be allocated.

The second section of the configuration file is workflow specification; you specify exactly how modules should run, such as the order in which modules should run, and what kind of dependencies exist between them, and how many iterations.

1.2 Intermodular Communication

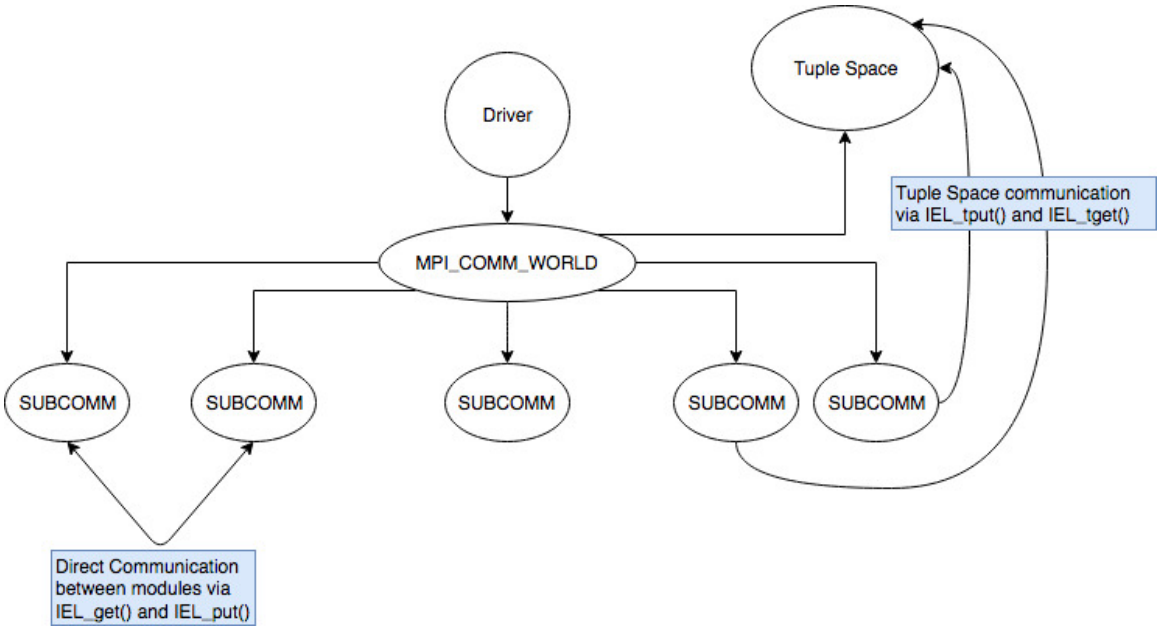

Figure 1: Graph showing inter-modular communication

Modules can communicate via either direct synchronous communication via IEL_get() and IEL_put(), or via asynchronous tuple space communication via IEL_tput() and IEL_tget() functions.

1.3 Driver

When running a workflow under openDIEL, all that is needed is a single driver executable. In order to create the driver executable, parallel modules code must be converted into a function, compiled as a library, and linked to the driver. Serial code can simply be started with fork() and exec(). The driver calls MPI_Init(), starts all of the user specified modules, and splits each of the modules into their own subcommunicator.

1.4 Managed and Automatic Modules

For module specification, there are two type of modules: Managed Modules, and Automatic Modules. Automatic modules consist of serial code; they do not call MPI_Init() or utilize any kind of parallelization. Automatic modules can simply be run with fork() and exec().

However, modules that utilize openMPI functions must be compiled as libraries and linked with the driver, and called as functions. Since driver should make the only call to MPI_init(), and the driver will create MPI_COMM_WORLD, managed modules must remove any calls to MPI_init() and MPI_Finalize(), as well as references to MPI_COMM_WORLD, and replace them with the proper subcommunicators of the MPI_COMM_WORLD created by the driver.

2 Interface

As previously discussed, openDIEL allows for the specification of modules and workflows via Workflow Configuration files. This is what the driver reads in order to figure out how to allocate resources, run modules, and work out dependencies.

2.1 Module Keywords

On a basic level, the way that resource usage is specified for each module of computation is through the usage of keywords:

2.1.1 function

This keyword is a string that specifies the name of the module. This string is used to refer to the module in the workflow section.

2.1.2 args

In the case of automatic modules, this keyword specifies the command that will be executed. For managed modules, this is a list of command line arguments that will be passed to the module when it is run.

2.1.3 copies

The copies keyword specifies how many copies of a module need to be run. This can be used in conjunction with a *splitdir* keyword that specifies the name of directories that input will come from, so that many copies of the same module can be working on different data.

2.1.4 processes_per_copy

Used in conjunction with copies, specifies how many MPI processes each copy will need. For example, you could specify that you want two copies of a LAMMPS simulation to run, with each of the copies to have 6 processes per copy.

2.1.5 threads_per_process

This keyword specifies how many openMP threads will be created when openMP parallel sections are encountered. This works by calling omp_set_num_threads(*threads_per_process*) before running each module that specifies this setting.

2.1.6 size

This keyword specifies the total number of MPI processes a module will require. When *processes_per_copy* and copies are specified, this value defaults to *processes_per_copy* * *copies*.

2.1.7 stdin

Names a file from which input will be redirected from. This is useful to allow you to run many different copies of the same module on different input data.

2.1.8 num_gpu

This specifies the number of GPUs that a module will require. If 2 GPUs are available, and 2 modules request 1 GPU, each module will be allocated separate GPUs. This works by setting the CUDA_VISIBLE_DEVICES environment variable.

2.1.9 cores

How many cores the module will require in total. This is calculated by *size* * *threads_per_process*

2.1.10 splitdir

This keyword specifies the name of the directories in which input and output will go for each module or copy of the module. If *splitdir* is set to "module-splitdir", for each copy of the module, it will receive it's input from files named by the previously specified keyword *stdin*, in directories named "module-splitdir-0,...,module-splitdir-[N-1]", where N is the number of copies of the module.

2.2 Workflow Organization

2.2.1 groups

Groups contain an ordered list of modules that they are to run.

order: This keyword is specified for groups, and is assigned a list of modules that are to be run

depends: This keyword is specified in groups, and is assigned a list of groups

that must be finished running before the group can begin.
iterations: This simply specifies the number of iterations the group will run

2.2.2 sets

Sets contains groups. All sets begin running in parallel. Dependencies can only be specified between groups that are members of the same set.

2.3 Application: LAMMPS

LAMMPS is popular simulation software for classical molecular dynamics that utilizes openMPI, openMP, and CUDA for parallelization. As such it is well suited to be run as a module under openDIEL. After compiling as a static library, LAMMPS can then be linked with a driver and called as a function.

2.3.1 Modules

The following LAMMPS modules were specified:

```
{
   function="lammps_mod"
   stdin="in.lj"
   splitdir="lj-melt"
   copies=2
   processes_per_copy=6
   size=12
}
```

Figure 2: Module 1 as it appears in a workflow file

Module 1 runs two copies of LAMMPS on different input parameters, utilizing 6 MPI processes each. This kind of module was specified with keywords *copies*=2, *processes_per_copy*=6, *splitdir*=lj-melt, *stdin*=in.lj.

```
{
   function="lammps_mod_gpu"
   args=("","-sf","gpu")
   stdin="in.friction"
   splitdir="friction"
   copies=1
   processes_per_copy=1
   size=1
   num_gpu=1
}
```

Figure 3: Module 2 as it appears in a workflow file

Module 2 runs with 1 MPI process and 1 GPU on a single file for input. This module was specified with they keywords *size*=1, *splitdir*=friction, *stdin*=in.friction, *num_gpu*=1.

2.3.2 Workflow

The workflow was specified with one set containing three groups:
tuple_group: specified with *order*=("ielTupleServer") and *iterations*=1. A tuple server is needed any time there are multiple groups that need to be started, or dependencies need to be managed
Group 1: specified with *order*=("lammps_mod") and *iterations*=1
Group 2: specified with *order*=("lammps_mod_gpu") and *iterations*=1
Set 1: contains **group1** and **group2**
Since all of the modules are in their own groups, and no dependencies are specified, all modules will begin at the same time

The resulting workflow at it appears in the workflow configuration file is as follows:

```
set1:
{
  tuple_group:
  {
     order=("ielTupleServer")
     iterations=1
  }
  group1:
  {
     order=("lammps_mod")
     iterations=1
  }
  group2:
  {
     order=("lammps_mod_gpu")
     iterations=1
  }
}
```

Figure 4: Module 2 as it appears in a workflow file

Under openDIEL, the process of running many different parallel LAMMPS simulations with varying input parameters is made easy, and allows for a lot of flexibility as to what resources will be used for each simulation, while at the same time being able to submit everything as one job.

2.4 Application: GREP in parallel

Another useful application of openDIEL is to search large amounts of data in parallel. Existing serial code can easily be parallelized with openDIEL by splitting up the data to be processed into multiple smaller pieces, running many copies of the same code on the smaller data, and then combining the output of the copies.

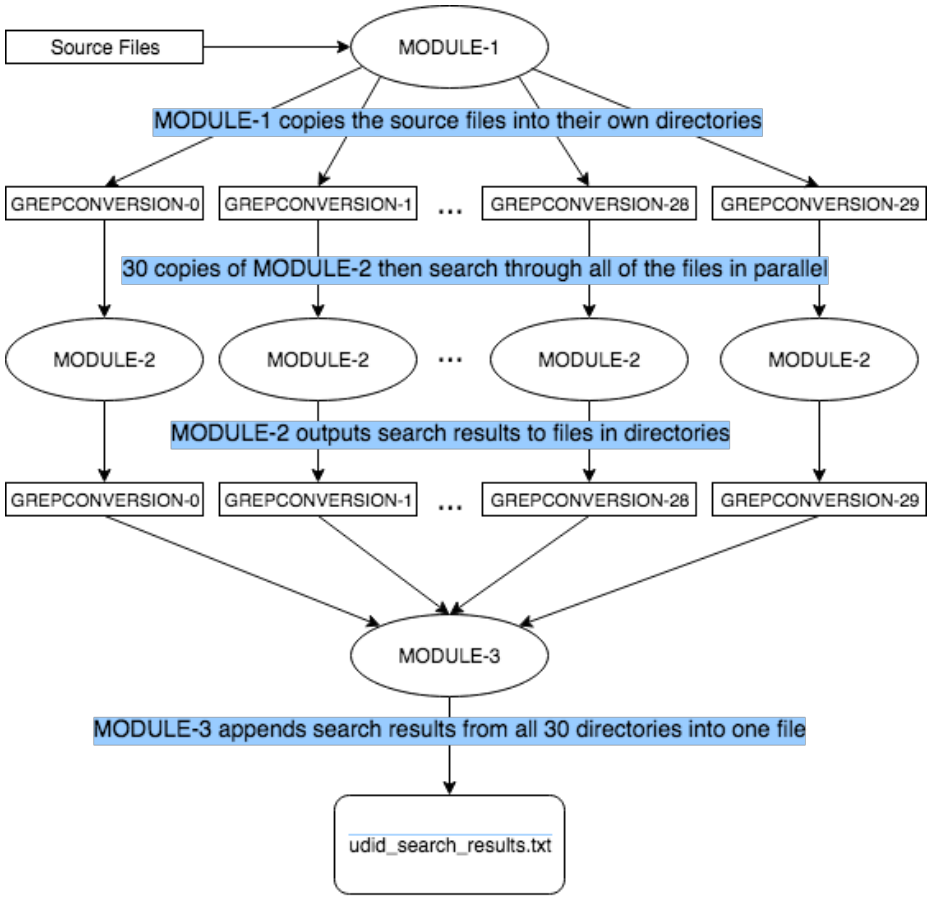

Figure 5: Diagram of entire workflow

2.4.1 The dataset

The dataset is 30 .csv files, totaling at about 180 million rows and 31 GB. Each row has 15 columns, and the goal is to search for and save all of the rows whose "UDID" column value matches the UDID value being searched for.

2.4.2 Modules

There are 3 modules that were specified:

2.4.3 Module: MODULE-1

The first module is simply a shell script that is given a path to a directory containing the 30 files that need to be processed. It takes all of the files, and creates 30 directories and copies each file in it's own directory. It then places a file containing a list of UDIDs that need to be searched for in each .csv file called "udids.txt". The resulting module is as follows:

```
{
  function="MODULE-1"
  args=("sh", "prepare_directories.sh", "data/April2016")
  libtype="static"
  size=1
},
```

Figure 6: MODULE-1 as it appears in the configuration file

2.4.4 Module: MODULE-2

The second module is another shell script that uses GREP to search for each UDID in "udids.txt", and redirect all of those lines to a file. The result is one file for each udid in "udids.txt", containing all of the lines that match in the .csv being search. This script is run with 30 copies so that all 30 .csv files can be searched for in parallel. The resulting module is as follows:

```
{
  function="MODULE-2"
  args=("sh","../search_udids.sh")
  libtype="static"
  size=30
  splitdir="GREPCONVERSION"
},
```

Figure 7: MODULE-2 as it appears in the configuration file

2.4.5 Module: MODULE-3

The third and final module looks through all 30 directories for each udid's search results file, and concatenates the results into a file that contains all of the lines for

the udid that were found in all 30 .csv files. The resulting module is as follows:

```
  {
    function="MODULE-3"
    args=("sh","location_concatenate.sh")
    libtype="static"
    size=1
  }
```

Figure 8: MODULE-3 as it appears in the configuration file

2.4.6 Workflow

The workflow only requires that the modules be run in order: MODULE-1 creates the directories containing the .csv files needing searching, MODULE-2 then searches the files and outputs a file in each directory with search results for each UDID, MODULE-3 then concatenates all of the search results for each UDID in all 30 directories into a single file containing all of the search results. The resulting workflow is specified as follows:

```
workflow:
{
  groups:
  {
    tuple_group:
    {
      order=("ielTupleServer")
      iterations=1
    }
    group1:
    {
      order=("MODULE1","MODULE-2", "MODULE-3")
      iterations=1
    }
  }
}
```

Figure 9: The complete workflow file

2.4.7 Results

With the original Python script, executing a search through all 30 files took roughly 40 minutes for a single UDID. Using GREP and parallelizing with openDIEL, the time was able to be reduced to an average of about 2 seconds per UDID to search through all 30 files.

Search Method	Time (minutes:seconds)
Python Script	40:00
SQLite Query	01:26
GREP	04:13
GREP with openDIEL	00:02

Figure 10: Comparison of time to search for 1 UDID in all files

3 Necessity for a GUI

Currently, the process of executing openDIEL in both managed and automatic modes has proven to be a very tedious and time consuming task, and if one were to try to execute this process without the assistance of a Graphical User Interface, they would need to follow each of the following steps.

First, the user would need to take their code(s) and run it through the python file ModMaker.py. The purpose of doing this step is because in order for a user to run their programs in parallel, they would need to first convert their code(s) into modules (also known as functions) in order for it to execute. Currently, ModMaker.py can only convert C code, C++, and Fortran code, but with future development, it will look to convert codes from different programming languages as well. The following figure details just what openDIEL is doing with a user's program.

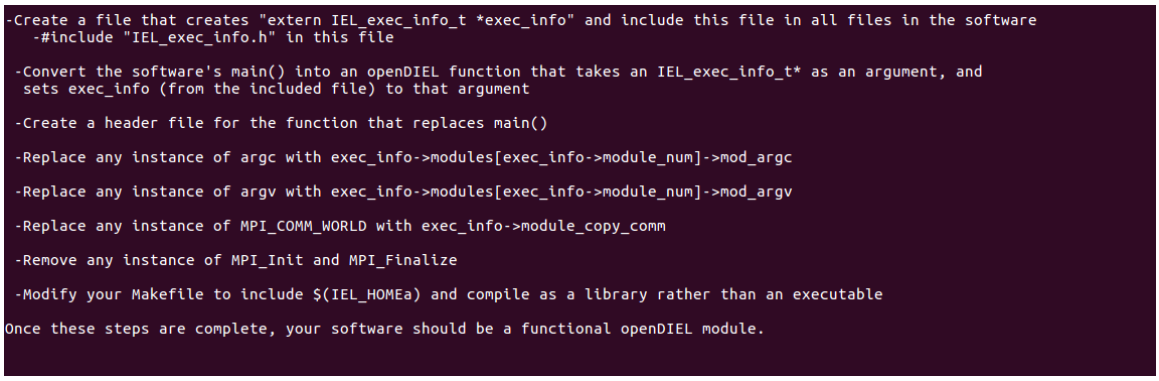

Figure 11: ModMaker.py's functionality

The following figures illustrate just how ModMaker.py modifies a simple Hello World program.

```
int main(void)
{
     FILE *fp;

     fp = fopen("file.txt", "w+");

     printf("i\n");
     fflush(stdout);
     fprintf(fp, "%s %s %s ", "HELLO-", "FROM-", "I-" );

     fclose(fp);
     return 0;
}
```

Figure 12: Original Code

```
#include "helloi.h"

int helloi(IEL_exec_info_t *exec_info)
{
     FILE *fp;

     fp = fopen("file.txt", "w+");

     printf("i\n");
     fflush(stdout);
     fprintf(fp, "%s %s %s ", "HELLO-", "FROM-", "I-" );

     fclose(fp);
     return IEL_SUCCESS;
}
```

Figure 13: Code converted to module form with ModMaker.py

As seen from both figures, the original code was successfully modified by ModMaker.py. The code was remade into a function called "helloi", and it was given the structure "IEL_exec_info_t *exec_info" as a parameter. It also returns "IEL_SUCCESS"

as a return statement. The user would need to repeat this step for as many modules as they would want to create.

The next step would be for the user to compile each module as a library. To do this, the user would need to enter in two commands for each module as shown in the following figure.

```
mpicc -c -I/home/reuub06user1/opendiel/INC/REUUB-06 helloi.c
ar -rcs libmodhelloi.a helloi.o
```

Figure 14: Commands for creating a library

The next step would be for the user to create a header file for their newly formatted modules. This is a very simple step, as all the user would need to do is follow this layout shown in the following figure. As stated previously, the user would also need to repeat this step for as many modules as they would want to create.

```
#include "IEL_exec_info.h"

#ifndef _MAINMOD_helloi_H
#define _MAINMOD_helloi_H

int helloi(IEL_exec_info_t *exec_info);

#endif
```

Figure 15: Header File

The next step is the most important step, as the user would need to create the workflow configuration file. The workflow configuration file is divided into two sections: the module section and the workflow section. The purpose of the module section is to define the existence of each module, and the user would do this by simply entering in the name of the module, giving the module some input arguments, giving a library type (either static or dynamic), including the name of the library that they compiled, giving the name of the split directory that will hold all of the output from that module, and finally assigning a number of processors to that module. The following figure shows just what the user would need to do in the module section.

```
modules=(
   {
     function="MODULE-0";
     args=("../../i-serial/helloiexe");
     libtype="static";
     splitdir="HELLOI";
     size=5
   },
   {
     function="helloi";
     args=();
     libtype="static";
     library="libmodhelloi.a";
     splitdir="HELLOI"
     size=5
   },
   {
     function="hellome";
     args=();
     libtype="static";
     splitdir="HELLOME"
     library="libmodhellome.a";
     size=5
   },
   {
     function="hellomy"
     args=()
     libtype="static"
     library="libmodhellomy.a"
     splitdir="HELLOMYSELF"
     size=1
   }
)
```

Figure 16: Module section of the Workflow Configuration File in Managed Mode

The next section is the workflow section, and the purpose of the workflow section of the configuration file is to define how each module will run once openDIEL has been executed. The user would do this by defining the order in which each module will run, the dependencies if needed, and also how many iterations that will occur. Once they've completed this step, the user would have created a group. The user can create as many groups as they'd want, but once they've created enough groups, they

would need to save those groups into a set. The user can also create as many sets as they'd like as well. The following figure shows that concept, and just what the user would need to do in this section. As seen from the figure below, the user defined two groups into a set, and listed an order and a number of iterations to go into each group.

```
workflow:
{
  groups:
  {
    group1:
    {
      # Note that both serial and parallel code can be run in the
      # same group
      order=("MODULE-0","hellome", "helloi")
      iterations=2
    }
    group2:
    {
      order=("hellomy")
      iterations=2
    }
  }
}
```

Figure 17: The Workflow section of the Workflow Configuration File

If the user is in managed mode, the next step would be for the them to edit the Driver.c code and include all of the header files that they created to correspond to each module that they created.

```
#include <stdlib.h>
#include <stdio.h>
#include "IEL.h"
#include "libconfig.h"
#include "IEL_exec_info.h"
#include "modexec.h"
#include "helloi.h"
#include "hellome.h"
#include "hellomy.h"
#include "tuple_server.h"
```

Figure 18: Module header files included in the Driver.c

Next, the user would go down into the code to IELAddModule function call and pass as arguments a function pointer to their module and the name of their module as a string argument.

```
  // Add all non-serial modules manually via IELAddModule
  IELAddModule(&helloi,"helloi");
  IELAddModule(&hellome,"hellome");
  IELAddModule(&hellomyself,"hellomy");
  IELAddModule(ielTupleServer, "ielTupleServer");
```

Figure 19: IELAddModule function calls for each module

```
#include <stdlib.h>
#include <stdio.h>
#include "IEL.h"
#include "libconfig.h"
#include "IEL_exec_info.h"
#include "modexec.h"
#include "helloi.h"
#include "hellome.h"
#include "hellomy.h"
#include "tuple_server.h"

#define MOD_STRING_LENGTH 20

void ConfigFile(void);

int main(int argc, char* argv[])
{
  int rc, rank, num_modules, i, size;
  char mod_name[MOD_STRING_LENGTH];
  config_t cfg;
  config_setting_t *setting;

  // Initialize basic MPI settings
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Comm_size(MPI_COMM_WORLD,&size);

  // ----------------- Timer ---------------------
  timestamp ("Begin", "driver.c", 1);
  // ----------------- Timer ---------------------

  // Add all non-serial modules manually via IELAddModule
  IELAddModule(&helloi,"helloi");
  IELAddModule(&hellome,"hellome");
  IELAddModule(&hellomyself,"hellomy");
  IELAddModule(ielTupleServer, "ielTupleServer");
  ...
```

Figure 20: Driver.c

Next, the user would need to link all of the compiled libraries for each module to the Driver and compile the Driver. Lastly, they would run the driver executable with

the Workflow Configuration file.

With a user-friendly Graphical User Interface in place, the issue of having the user being tasked with repeating each various step with little or no mistakes would be mitigated, as the goal of the GUI will be to simply handle most of this responsibility from the user, and to also speed up the process of preparing and running modules through openDIEL.

4 How GUI Solves Issues

The graphical user interface handles each of the current tasks that must be done manually in tabs, from a greeting tab with general instructions and constructing the workflow configuration file - modules and workflow sections are seperate - to creating and compiling a new driver, to running the driver and workflow, and printing out the results.

Upon the GUI being run, the first tab to open will be the "Introduction" tab, which houses basic information about the program for the user. Inside this tab there is simply generic information about the program, such as the creators, their respective establishments, as well as the National Science Foundation, and brief directions about each other tab for first-time users.

4.1 Module Tabs

The first and second actual tabs of the GUI are "Module Keywords" and "Module Attributes", taking care of the modules half of the configuration file. "Module Keywords" is short, and it is meant as a preemptive storage for the input arguments of a module. Here, any commands or files to be searched for a module (or multiple modules) can be given or a 'keyword' to be identifiable later in the application. These 'keywords' can be saved and updated as needed.

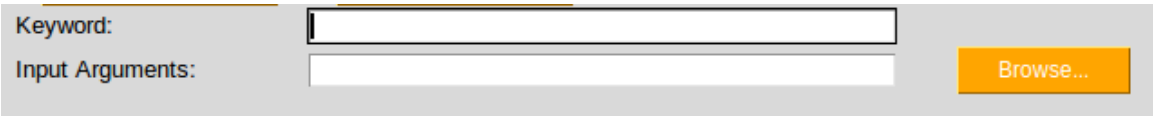

Figure 21: Module Keywords

Inside the "Module Attributes" tab is where the large chunk of information for modules is input. Before any information is stored, however, a mode for each module must be selected, between 'Automatic', meant for running only serial code, such as an executable or a file through an interpreter, or 'Managed', meant for running parallel code, which will require a few more steps in the driver creation. The mode is determined by which type of module is created, but although the decision is reversable, it will modify certain option entries for a switch to 'Automatic': the names, library

types, source code links and execution mode of all existing modules will be changed and then entry for these fields will be disabled unless the mode is reverted.

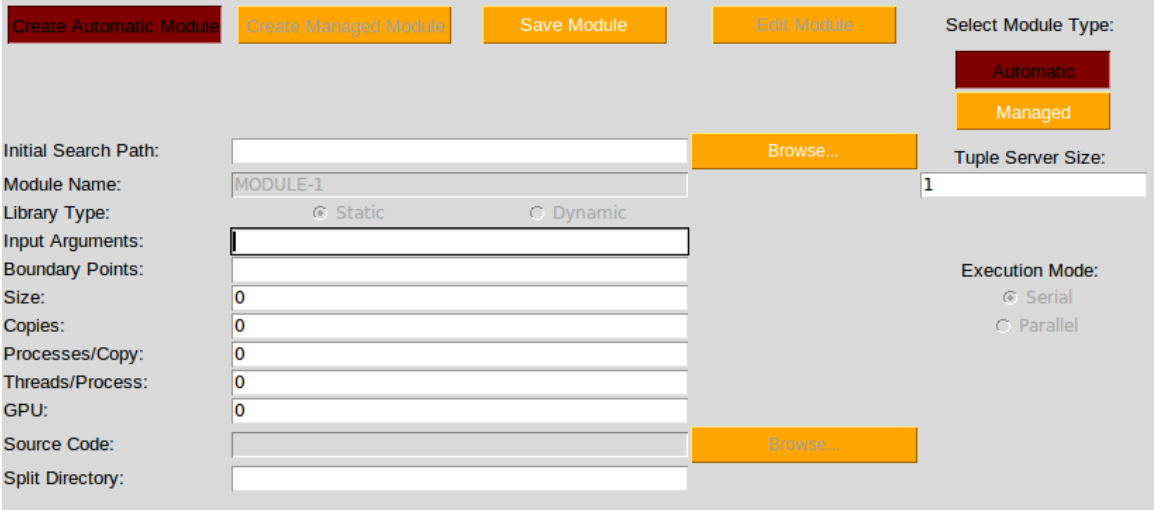

Figure 22: Module Attributes

After mode selection comes the bulk of the input (in order):

- File Search Path (Used for File Search Options in GUI as Initial Directory),
- Tuple Server Size (Tuple Communication),
- Module Name (Managed only),
- Module Library Type (Static or Dynamic),
- Module Input Arguments (May Take Input From 'Module Functions'),
- Module Boundary Points (Direct Communication),
- Number of Total Processors to Allocate for Module (Module Size),
- Module Execution Mode (Parallel or Serial),
- Number of Copies of Module,
- Processes Per Copy of Module,
- Threads Per Process of Module,
- Number of GPU to Allocate for Module,
- Path to Source Code File for Module (Managed only), and

- Name for Split Directories of Module.

Tuple Server Size: Number of processors to be allocated to the Tuple Server(s), which are run similarly to other modules, except this(these) will always come before created modules in the 'modules' half of the configuration file. Despite appearing for every module, this setting is not module dependant, and if set to 0, there will be no tuple server, and the 'dependencies' option in the 'Create Workflow' tab will be disabled. This value is listed at the top of the configuration file, as 'tuple_space_size'.

Module Name: This field is what will be stored as the 'function' parameter in the workflow configuration file, and must contain no spaces or special characters. For 'Automatic' mode, this will be preset to the name 'MODULE-N' where N is the current module number. For 'Managed' mode, if this module has a source code file that is modified in the 'Modify Driver' tab, this will become the name of the function of the new code, as well as the header file created from this new code.

Module Library Type: A choice between 'static' or 'dynamic', stored as the 'lib-type' parameter in the workflow configuration file.

Module Input Arguments: The input inside this entry box will be stored as the 'args' parameter of the module in the configuration file, broken up as a tuple of strings. Information inside this field must be entered just as the terminal command for it would be, including any tags, file references, or interpreter calls. Using the buttons placed on the left side from the 'Module Keywords' tab, the user can copy their stored data into this field with easy repeatability.

Module Boundary Points: This field takes in the boundary points for modules to have direct communication, and places them in the 'shared_bc_read' and 'shared_bc_write' parameters of the configuration file (Not Yet Supported) (if applicable).

Module Size: The number of processors the module needs to run, stored in the 'size' parameter in the workflow configuration file. This value is derived by the driver from multiplying the number of copies of the module by the number of processors per copy of the module (if applicable) when left blank. If the sufficient amount of processors are not available to be allocated , or are not given by the system, the workflow will not execute it at all.

Module Execution Mode: The execution mode for the module, stored in the 'exec_mode' parameter of the configuration file. This is a choice between 'serial' or 'parallel', and has bearing on the protocol for all the options related to module copies, excluding total module size. If this field is not 'parallel', it is not displayed in

the configuration file, and is inferred by the driver to be 'serial'.

Module Copies: The amount of copies of the module that are made at runtime. This is stored in the 'copies' parameter of the configuration file, and is used to derive the 'size' parameter by the driver if the size is left blank. If this value is set to 1 for 'serial' execution mode, or is equal to the 'size' parameter for 'parallel' execution mode, it will not be displayed explicitly in the configuration file.

Processes Per Copy: The number of processes each copy of the module receives at runtime. This is stored in the 'processes_per_copy' parameter of the configuration file, and is used to derive the 'size' parameter by the driver if the size is left blank. If this value is set to 1 for 'parallel' execution mode, or is equal to the 'size' parameter for 'serial' execution mode, it will not be displayed explicitly in the configuration file. If 'size', 'copies', and 'processes_per_copy' are all left blank on the configuration file, the module will not be executed at all.

Threads Per Process: The number of openMP threads each process of the module creates at runtime. This is stored in the 'threads_per_process' parameter of the configuration file, is used to calculate the 'cores' parameter along with 'size' implicitly, and is required for any modules that utilize openMP. If this is left blank inside the configuration file, it will default to 1.

Number of GPU: The number of GPU to allocate for the module. This is stored in the 'num_gpu' parameter of the configuration file (if applicable). This will default to 0 unless specified.

Source Code File: The file containing the unmodified source code for the module. This file is copy converted by ModMaker into a function with all of it's MPI calls modified to operate in a sub-communicator. This file location is necessary for any 'Managed' modules that wish to call program files that cannot be run with a simple command.

Module Split Directory: The copied name of the directories that house the input, output, and code for the module to be run. When the driver begins a module, it splits the work over its processors, creating a directory for each of them to work on files from. The split directories names will have a '-N' on the ends, where N represents the processor of that module ('HELLO-0' would belong to the first processor, 'HELLO-1' to the second, and so on).

4.2 Workflow Tab

The third tab is "Create Workflow", and it is here that the directions for the engine are established. The workflow itself can be compared to an employee schedule, except the employees in this case are the processors. Inside the configuration file, the second half is one large set labeled 'workflow'. Inside this set exist 'sets', and each set houses one or more 'groups', each with their own properties. Aside from the nested sets titled 'groups', each 'set' also contains a value for its iterations, titled 'num_set_runs' in the configuration file.

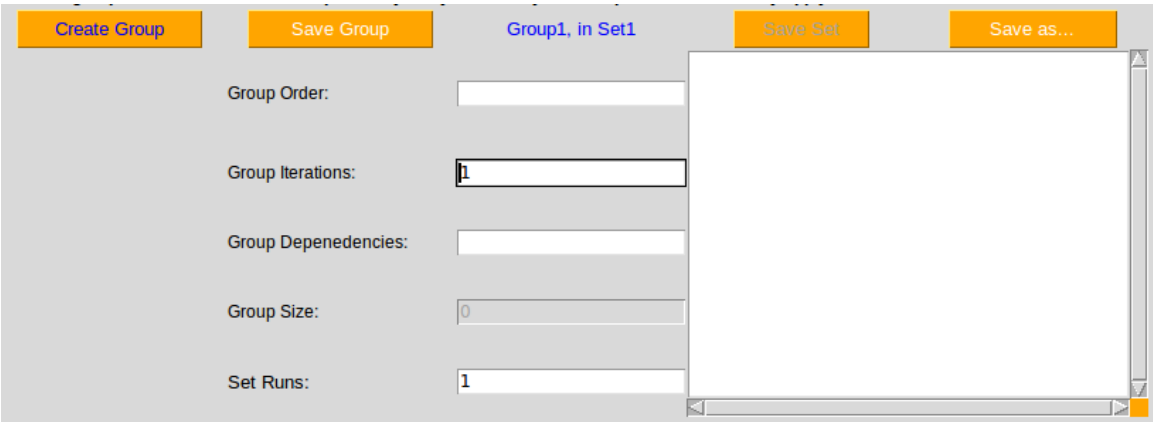

Figure 23: Create Workflow

Inside of each 'group' exists:

- The Group's Module Order,
- The Group's Iteration Count,
- The Group's Dependencies, and
- The Group's Total Processor Count (Group Size).

Group Order: The order the modules will run inside a group. The modules listed here run one after another sequentially, until it is completed. This may be run multiple times with varying group/set iterations, and the order determines the total amount of processors. Inside the workflow configuration file, this is listed as 'order'.

Group Iterations: The number of times the group will be run. This must be at least 1, and each run will execute the entire order once. Inside the configuration file, this is named 'iterations'.

Group Dependencies: The list of other groups the group is dependent on. This option is listed as 'depends' inside the configuration file, and until every group in this

list has run, the group will not start a single module. Groups can only be dependent on groups in the same set, and if either (1) a name is given for a group that does not exist in the current set, or (2) there is a dependency circle, where 2 or more groups are dependent on each other, the workflow engine will reject the configuration file and self-terminate.

Group Size: The amount of processors a group requires for all of its modules. Based on the entry of modules (which can be done manually or through the module buttons), this value will change once a group is saved. This is not displayed in the configuration file, but is used by the GUI to determine how many processors to call when the driver is executed.

Once the user is finished creating their groups and sets, there is a 'Save as' button, which allows them to write all their stored information about the modules and the workflow to a configuration file, where it is formatted and then displayed in the text box. If there are any errors, the user can either correct them inside the file itself, or edit the options in the GUI and overwrite it by saving again.

4.3 Driver Tab

The fourth, and most automated tab is "Modify Driver", where most of the GUI-side of work for 'Managed' mode occurs. When the user hits the 'Create Driver' button, they are prompted to inform them about what files will be overwritten if the GUI is successful. If they accept, several processes will occur behind the scenes, resulting in the driver being modified:

Figure 24: Modify Driver

The subsequent processes are:

- Turning Main Codes into Module Function Codes(Managed only),
- Creating Header Files for each Managed Module(Managed only),
- Creating Libraries for each Module Function Code,
- Including each Header File in the Driver, and
- Compiling the Driver Code and Linking Libraries.

Make Module Function: The source codes the user entered previously are not usab le by the driver in their uncompiled states, so they are first converted into functions (with the function name matching the module name given) that take in a pointer to the struct 'IEL_exec_info_t' from the driver when called. The original files are preserved in their previous locations, unless they were in the list of names of files to be overwritten. The converted and unconverted files are then listed in the textbox of the Driver Tab separately. Note that this step does not occur (1) for automatic mode, or (2) if there are no managed modules (modules with names that are not 'MODULE-N').

Make Header File: For all modules that are managed, there need to be header files that go into the driver, in order to connect their functions to the workflow later on. Instead of their normal includes and such, all that is used is "#include 'IEL_exec_info.h" to pass the struct pointer 'exec_info' as each function's argument. For each header created, there will be a small message listed in the Driver Tab textbox. This step does not occur for non-Managed modules, as they are covered by 'modexec.h'.

Make Library: In addition to header files, libraries must be made out of the new 'Module Function' files, to be linked to the driver code near the end. For each new file converted, they are compiled through 'mpicc', a wrapper around gcc made for MPI, with the declaration of the struct pointer 'exec_info', which contains all the variables and input arguments the function will need. Once it is compiled into an object, it must be archived as a library, and placed into the 'GUI_MODS' directory for use in driver compilation. In addition to all managed modules, this occurs for the file 'modexec.c', which is the function code used to run 'Automatic' modules. This step occurs in both modes.

Include and Link: Lastly, the new headers and libraries are connected to the driver through '-I' and '-L' commands, and based on the modules, a driver is compiled ('driverAM' or 'driverMM').

4.4 Launch and Output Tabs

The last two tabs are "Launch-DIEL" and "DIEL-Result", which serve as launching and output displaying tabs, respectively. Inside the 'Launch' tab, there is a command listed, and a run button. The command is automatically generated based on the created groups and sets, so all the user can do here is hit 'Run', and the driver will get started.

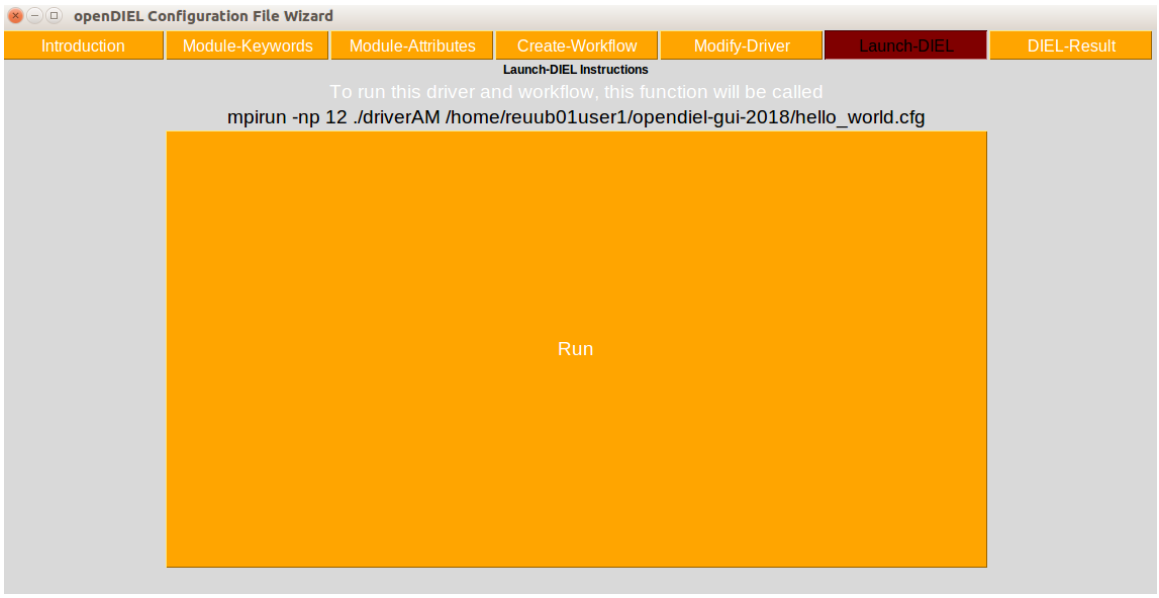

Figure 25: Launch DIEL

Once it is finished, the 'Result' tab will change color, signifying the user should proceed. Once the final tab is opened, it will revert back to its original color, and the output of the driver running will be displayed, along with any internal or extrenal errors that occur.

5 Future Work

- Automatic generation of SLURM batch scripts for running on several XSEDE supercomputers
- Add support for using LAMMPS in the GUI, as well as other pre-prepared libraries
- Further improvements to the useability and user friendliness of the interface
- Increased error checking in the GUI