Creating a GUI to define workflows in openDIEEL and adding support for GPU Deployment

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

- openDIEL (Open Distributive Interoperable Executive Library) is a lightweight workflow engine designed for usage in HPC environments.
- Allows for the unification of many different modules into a single executable
- Written in C, and uses openMPI
Workflow configuration files specify the workflow and what resources are to be used by each module.

- First section contains modules: This is where you define what resources are needed by each module, such as the number of MPI processes, the number of GPUs, as well as defining where input and output needs to come from if needed.

- Second section contains the workflow: This is where the order of the modules is defined, and dependencies are specified, as well as the number of iterations that each group of modules needs to run.
Intermodular Communication

There are two forms of communication:

- Direct, synchronous module-to-module communication with IEL functions IEL_put() and IEL_get()

- Indirect, asynchronous tuple space communication with IEL functions IEL_tput() and IEL_tget.

- Also supports using multiple tuple servers with a variety of different API calls for modules to communicate with tuple servers in different ways.
Driver

- As previously mentioned, all modules are unified under a single executable, referred to as the driver.
- Driver makes calls to MPI_Init(), MPI_Finalize(), and makes relevant calls to IEL member functions to read the configuration files, add modules, and start the entire IEL system
Figure 1: Simplified overview of MPI process usage
Managed and Automatic modules

There are two basic types of modules:

-Automatic: These are serial modules, with no calls to MPI functions, and they can simply be run in openDIEL with fork() and exec()

-Managed: These are parallel modules with calls to MPI functions. These require that any MPI_Init() or MPI_Finalize() calls are removed, since the driver will make these calls. Additionally, the MPI_COMM_WORLD created must be removed and replaced with the proper subcommunicator of the MPI_COMM_WORLD created by the driver.
As previously mentioned, the interface is basically just a configuration file specified by the user.

- Resources are specified on a per module basis through keywords, such as copies, processes_per_copy, num_gpu, threads_per_process, size, stdin, and splitdir.

- Workflows are broken into two divisions: sets and groups.
Application: GREP in parallel

- An application of openDIEL is parallel data searching

- Existing serial code can easily be parallelized under openDIEL by the following process:

  1) Split data to be processed into smaller pieces
  2) Run several unmodified copies of the script on different subsets of the data and direct the results of the script to files
  3) Collect the output from the files and combine into a single file
Transit UDID search

-The need for such an application arose when existing Python scripts for a data search were not sufficiently fast for the data size.

-The goal was to be able to search through a dataset of 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.
Figure 2: Graph of parallel search with openDIEL
Workflow in openDIEL

```json
{
  function="MODULE-1"
  args="(sh, prepare_directories.sh, data/April2016)"
  libtype="static"
  size=1
},
{
  function="MODULE-2"
  args="(sh, ../search_udids.sh)"
  libtype="static"
  size=30
  splitdir="GREPCONVERSION"
},
{
  function="MODULE-3"
  args="(sh, location_concatenate.sh)"
  libtype="static"
  size=1
}
}
```

Figure 3: Parallel search workflow configuration file
Results

<table>
<thead>
<tr>
<th>Search Method</th>
<th>Time (minutes:seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python Script</td>
<td>40:00</td>
</tr>
<tr>
<td>SQLite Query</td>
<td>01:26</td>
</tr>
<tr>
<td>GREP</td>
<td>04:13</td>
</tr>
<tr>
<td>GREP with openDIEL</td>
<td>00:02</td>
</tr>
</tbody>
</table>

Figure 4: Comparison of various methods to search for 1 UDID

Further improvements could likely be made by directing search results to an openDIEL tuple server, rather than conducting file I/O for saving and collecting search results.
Why do we need a GUI?

- The purpose of the GUI is to provide a much more user-friendly interface in order to execute openDIEL.

- Currently, in order for a user to execute openDIEL, they would have to go through a lot of tedious and time consuming tasks.
Why do we need a GUI? (contd.)

- One of the first steps that the user would have to do is to first convert their source code into a module/function by using the Python program ModMaker.py.
- The reason for this step is because in order for openDIEL to be executed, the user must convert their code(s) into a function.
- The user would need to repeat this step for as many modules that they would want to create.
Figure 5: This figure is a simple example of a Hello World program.
Converted Code from ModMaker.py

Figure 6: This is the result of the Hello World program after it has been converted by ModMaker.py
Why do we need a GUI? (contd.)

- The next step would be for the user to also compile each module as library.
- In order to do that, the user would need to enter the following commands as shown in the following figure.
- Again, the user would need to repeat this step for as many modules that they would want to create.

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

*Figure 7: Commands for compiling a module as a library.*
Why do we need a GUI? (contd.)

- The next step would be for the user to create a header file for each module that can be included into the Driver.c code.
- This step is very simple as all the user would need to do is follow this layout as shown in the following figure.
- The user again would repeat this step for as many modules that they would want to create.
/*
 * Copyright (c) 2015 University of Tennessee
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
 * OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
 * NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
 * HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
 * WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS
 * IN THE SOFTWARE.
 */

#include "IEL_exec_info.h"

#ifndef _MAINMOD_helloi_H
#define _MAINMOD_helloi_H

int helloi(IEL_exec_info_t *exec_info);

#endif
Why do we need a GUI? (contd.)

- The next step would be for the user to create a 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 of the configuration file is to define the existence of each module.
- The purpose of the workflow section is to define how each module will run.
- The following figures illustrate the format of a typical configuration file.
Figure 9: Module Section of the Configuration File.

tuple_space_size=0
modules=
{
  function="MODULE-0"
  args="././l-serial/helloexe"
  libtype="static"
  splitdir="HELLO"
  size=5
},
{
  function="hello1"
  args=()
  libtype="static"
  library="libmodhelloworld.a"
  splitdir="HELLO"
  size=5
},
{
  function="hellome"
  args=()
  libtype="static"
  library="libmodhelloworld.a"
  splitdir="HELLO"
  size=5
},
{
  function="hellomyself"
  args=()
  libtype="static"
  library="libmodhelloworld.a"
  splitdir="HELLOMYSELF"
  size=1
}
Figure 10: The workflow section of the Configuration File
Why do we need a GUI? (contd.)

- The next step would be for the user to edit the Driver.c code by including each of the header files that correspond to each module they created.
- They would also go down in the code to the IELAddModule function call and pass as arguments a function pointer to the module and the name of the module as a string argument.
- The following figures detail these steps.
```c
#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 "modscript.h"
#include "tuple_server.h"

//@ Add all non-serial modules manually via IELAddModule
IELAddModule(&helloi, "helloi");
IELAddModule(&hellome, "hellome");
IELAddModule(&hellomyself, "hellomy");
//@ IELAddModule(&modscript, "modscript");
IELAddModule(ielTupleServer, "ielTupleServer");
```
Why do we need a GUI? (contd.)

- Next, the user would need to link all of the compiled libraries to the Driver.c code and compile the Driver.
- Lastly, the user would need to run the Driver executable with the configuration file.

- With the assistance of a Graphical User Interface, the trouble of trying to do each step would be mitigated, as the GUI will seek to handle most of the responsibility for the user and to also speed up the process of preparing and running modules through openDIEL.
How the GUI Solves These Issues!

The openDIEL Graphical User Interface, constructed in Python, provides a much simpler, faster, well-laid-out alternative to the module construction process. Be it inside the Introduction tab or another section, there are short instructions to guide the user on what they need to do.

Figure 13: Introduction Tab
Keyword and Module Tabs

Figure 14: Module Keywords Tab

Figure 15: Module Attributes Tab
Workflow Tab

Figure 16: Create Workflow Tab
Driver Tab

Figure 17: Modify Driver Tab
Launch and Output Tabs

Figures 18 and 19: Launch-DIEL and DIEL-Result Tabs
Any Questions?