Procedia Computer Science

Volume 80, 2016, Pages 2332–2337

ICCS 2016. The International Conference on Computational Science

Best practices in debugging Kepler workflows

Michał Owsiak[1], Marcin Płóciennik[1], Bartek Palak[1], Tomasz Zok[1] , and Olivier Hoenen[2]

[1] Poznań Supercomputing and Networking Center IBCh PAS, Poznan, Poland
michalo@man.poznan.pl, marcinp@man.poznan.pl, bartek@man.poznan.pl, tzok@man.poznan.pl
[2] Max-Planck-Institut für Plasmaphysik, Garching, Germany
olivier.hoenen@ipp.mpg.de

Abstract

In this paper we present various techniques related to Kepler development, debugging, and *JVM* customisation. We highlight some aspects of development process that may help people to perform better while working with Kepler (especially in case they develop new components for the Kepler platform). We present knowledge and ideas that were gained over the time while working with Kepler tools throughout various projects and different applications of Kepler into existing environments. These ideas are presented for the sake of saving time and effort by other people who just start their experience with Kepler project.

Keywords: Kepler, debugging, development, DevOps

1 Introduction

Development in Kepler platform may be challenging at some aspects - especially when you develop for variety of legacy code and combine numerous components within workflows [3]. Our experience comes from many projects like PLGridPlus, EUFORIA or EUROFusion[2][1 2], with many complex scenarios orchestrated by Kepler.

In our research we work at the edges of various technologies that overlap just partially. Very often we have to develop components that has no precedence and make numerous languages (like Fortran, C, C++, Python, Java) and technologies work together. In addition to that, we have to combine them within a workflow platform - Kepler.

This paper is divided into three main sections, each describing different aspects of Kepler related developments. The first section focuses on workflow level, the second section puts actors into focus, and the last one describes JVM internals that helps to debug Kepler. Each section

[1]Part of this work has been carried out within the framework of the EUROfusion Consortium and has received funding from the Euratom research and training programme 2014-2018 under grant agreement No 633053.

[2]Part of this work has been co-funded by the Horizon 2020 Framework Programme through the INDIGO-DataCloud Project, RIA-653549.

 Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2016

doi:10.1016/j.procs.2016.05.433

is independent and highlights just part of a process. Of course, we just cover a small fraction of what can be done with Kepler and its ecosystem.

2 Workflows

Sometimes the workflow just stops and the perception of what has been called first is lost. You can encounter this issue while working with *DDF* based workflows. Quite often this is related to data stream draining. It means that data are produced at one place, but are not properly consumed somewhere else.

2.1 Looking for floating tokens

You can always find floating data by using *ptolemy.vergil.actor.lib.MonitorReceiverContents*. This way, you can spot all the tokens that are still pending. They have not been consumed by actors, yet. If you add *ptolemy.vergil.actor.lib.MonitorReceiverContents* into the canvas and run workflow, you can easily spot the place where data are not transferred correctly[3].

2.2 Checkpoints

If you are developing large workflow, with lots of parallel components, you should pay attention to proper handling of data flow. Sometimes you want to stream data in a particular way. We find *Expression* or *Constant* actors really helpful in this case. To stream data flow we simply connect all outputs of actors that have to finish their job before we can proceed and, after all inputs are ready, we simply pass data to other part of the workflow[4].

2.3 Using Multiple Tab Display to control text output

The simple *Display* actor has one disadvantage - it can not be plugged into workflow in a synchronised way. It means it is hard to tell whether it fired before or after particular actor. This can lead to incorrect interpretation of results. You can easily solve that by replacing the Display by a Multiple Tab Display. Just put it somewhere in the workflow, where you want to check output, and connect it with following actor[5].

3 Actors

3.1 Prototyping with Groovy

In case you want to test some ideas for an actor and develop it quickly, there is a great, new actor in Kepler 2.5 - Groovy. It allows you to embed Groovy code directly in the Kepler actor.[6]

3.2 Prototyping with Jython

Python is widely used within scientific projects. A few possibilities exist in order to execute a Python script from a Kepler workflow. By default, you can use *Jython* implementation of

[3] https://raw.githubusercontent.com/mkopsnc/keplerhacks/master/workflows/lock.kar
[4] https://raw.githubusercontent.com/mkopsnc/keplerhacks/master/workflows/synchronize.kar
[5] https://raw.githubusercontent.com/mkopsnc/keplerhacks/master/workflows/mtd.kar
[6] https://raw.githubusercontent.com/mkopsnc/keplerhacks/master/workflows/Groovy.kar

Python in Kepler. This approach is quite efficient when it comes to simple activities that do not require access to JVM data (e.g. shared libraries). However, while using this approach, we have faced numerous compatibility issues and we had to develop different approach for handling Python based scripts. Issues were related to accessing native code via the *cython* based bridge. Incompatibilities and lack of support for cython[7] based types lead to developing solution based on running Python as external process.

3.3 Prototyping with Python

If you want to overcome Jython limitations, you can always call Python directly from Kepler via *java.lang.ProcessBuilder*. This way, you can run everything as if you run it directly in Python. However, there are few limitations in this solution. First of all, you are running your code in separate process. If are using shared libraries then they all will be duplicated, which may have increase memory consumption. In addition, they will not be shared between the JVM and Python - this might be the issue in case you want to share some low level components from shared libraries. Another issue with invoking Python as an external process is the startup time of Python. Each time you start new Python process, you have to load Python environment and all the modules that are used by your code. This might have huge impact on execution time. Running workflow that contains code invoking Python triggers huge performance issues. This affects mostly parallel executions. For a simple Python code dealing with NumPy based data types, we have observed execution times as presented in Table 1. These kind of results were not acceptable. In order to reduce times, we had to developed different approach for Kepler based executions. We have decided to run Python via direct *JNI*[7] calls to Python library.

In Table 1 we present some timing tests executed for exactly the same Python script, however, this time we call it directly from the JVM via a call to the Python shared library. To call Python library we use JNI mechanism. However, it is also possible to use JNA for the same purpose. JNA library has slightly different approach in terms of calling shared libraries. You can check the differences by looking at sample code implementing simple Ḧello worldäpplication[8].

In the first case (first column), we have executed the Python script as an external process (as shown in previous section) for various numbers of CPUs being used. The second column shows results for the same Python script called via JNI call, the third column shows results that are based on JNI and, in addition, benefit from internal caching mechanism of one of the libraries. Thanks to running Python in exactly the same process space as Java process we could benefit from optimisations done at shared library level. Its internal cache was fully accessible for the Python script as it was executed in the same process space as JVM itself. In this case, the gain is huge and we benefit from some internals that are project specific[1]. Of course, this is not the solution for all Python based cases[9].

CPUs	Python process [s]	JNI open [s]	JNI cache [s]
1	385	186	13
2	415	200	16
4	564	235	14
8	1365	902	21
16	6557	3800	50

Table 1: Different execution times for python code extracted from SYCOMORE workflow

[7] http://www.jython.org/archive/22/userfaq.html
[8] JNI vs. JNA - https://github.com/mkopsnc/keplerhacks/tree/master/jnijna
[9] Sample code - https://github.com/mkopsnc/keplerhacks/tree/master/python

While calling native code via JNI or JNA one must pay attention to segmentation faults in the native code. They will have impact on JVM. We are discussing this topic later in the text, in section 4.5.

3.4 Embedding multiple versions of a native library

When a native code (Fortran or C++ in our case) is turned into a library and executed as an actor through JNI, developers might want to switch easily between different versions of the native library. One of such use case is to switch from an optimised version of the library to a more verbose version when debugging is required directly from the workflow. As Java does not bring the capability to safely unload such native library, we need to use the dynamic linking loader C API (`dlfcn.h`).

In the actor Java class, a simple boolean parameter or a multiple choice string parameter can be added to allow different versions of the native library to be selected. In the JNI code, additional steps consist of: loading the specific dynamic library (exact path/name is required, depending on choice made in actor's parameter), getting the address of the targeted symbol in memory (pointer on function to be executed), unloading the dynamic library (when its reference count drops to zero). Of course this flexibility comes with a small overhead, but in the development phase much time can be saved by allowing such run-time choice.

4 Working close to the JVM

When working with native code, especially legacy code, you can experience various issues that may impact workflow execution: calls to *exit()*, segmentation faults, runtime errors and bugs in the native code. These issues are hard to handle as they are embedded (typically) inside code you cannot access or require debugging of the code outside JVM. However, there are ways to overcome these issues.

4.1 Customising startup parameters

The startup procedure of Kepler consists of two steps. In the first step you run a startup class, then it creates a new JVM instance and starts it with the specified parameters. Unfortunately, not all the parameters can be passed to the second instance as its creation process is statically. However, there is a way to pass separate and highly customised parameters for the JVM via the *_JAVA_OPTIONS* environment variable[10]. Let us say we want to quickly set some properties for JVM, e.g. we want to customize memory settings. It is as easy as exporting an environment variable with proper settings, just before running Kepler.

```
export _JAVA_OPTIONS="-Xmx2G -Xms1G -Dmy_propert='some value'"
```

In addition, you can specify separate settings for the startup process and for the the main Kepler module. By altering the *environment.txt*[11] file inside your main module it is possible to specify custom settings for the _JAVA_OPTIONS environment variable. It can be quite useful in case you want to attach a debugger to your JVM before Kepler starts. You can also fine tune memory settings or diagnostic related information provided by JVM (e.g. JNI being verbose).

[10] This environment variable is not mentioned by official JVM documentation
[11] Each module can have separate environment.txt file that defines system variables

4.2 Debugging Kepler in Eclipse without projects

Debugging Kepler in Eclipse might be a quite time consuming process especially in case we want to make quick checks on some actors. It can get even worse if you have number of Kepler versions being used at the same time. There is, however, a way that can save you lots of time while working with Kepler sources.

Instead of building Eclipse projects (based on description in *Generating Project Files for Your Favorite IDE*[12])

```
cd build-area; ant eclipse
```

and importing them into Eclipse (with all the necessary configurations, e.g. adding external jars, etc.) you can do a dirty debugging by creating empty project in Eclipse, adding source path with Java classes, you want to debug, and attaching Eclipse to already running process. This way, you can quickly and easily make some tests in your code without building proper, Eclipse based, environment for Kepler. All you have to do is to export _JAVA_OPTIONS variable

```
export _JAVA_OPTIONS="-agentlib:jdwp=transport=dt_socket,\
      server=y,suspend=y,address=8000"
```

before running Kepler and then, attach to already running Java process from Eclipse *Run > Debug Configurations > Remote Java Application*. This way, you can focus on the source code in which you are interested.

4.3 Debugging Kepler in gdb

Debugging Kepler under *gdb*[5] can be very useful in case you want to debug some native code that is used by Kepler (e.g. shared library, JNI code). You can easily attach gdb to running Kepler session by attaching JVM that runs Kepler. You have to make sure to connect to correct Java process. To get its *PID* you should examine process tree. It is possible to get this information either with *pstree* or with *ps f*.

```
shell> jps
17220 Jps
9502 Kepler
9468 Kepler
shell> ps f
5636 pts/5    Ss     0:00 -bin/tcsh
9462 pts/5    S+     0:00  \_ /bin/bash
9468 pts/5    Sl+    0:01      \_ java
9502 pts/5    Sl+    0:24          \_ java
```

As you can see, process *9502* is a child of process *9468* (startup process). Process *9502* is the one that will actually run workflows. Once you know it, you can attach debugger to Kepler by running gdb.

```
> gdb /proc/9502/exe 9502
```

This way, you can debug the whole JVM together with all loaded shared libraries.

4.4 Handling exit calls

Handling exit calls can be done using *atexit*[4] function. It is possible to combine atexit, *siglongjmp* and *sigsetjmp* to simulate try/catch approach inside JNI code. This way, legacy code will not terminate JVM. Sometimes it might be very useful, especially in case we do not have access to source code of the library that calls exit() function. All we have to do is: define

[12] https://kepler-project.org/developers/teams/build/documentation/build-system-instructions

a method to call when exit function is called, set handler for the exit call via atexit and then use sigsetjmp to handle incorrect call to exit()[13].

Another approach to handle exit() calls is to set custom handler via *-Dexit=new_handler* option for the compiler: each call to exit() function is replaced by the *new_handler* code. This solution is even better as it allows the setting of different exit handlers for different libraries. However, it requires an access to legacy code source. The main difference, comparing to solution based on atexit is the way code is compiled and the way we define the handler for exit call[14].

4.5 Handling SIGSEGV

Segmentation faults generated by native code cause the JVM process to exit prematurely[6]. In case JVM encounters SIGSEGV it produces fatal error log and terminates. Sometimes you would like to avoid this kind of behavior. You would like to produce informative message and use some better means of informing user that something went wrong. In the case of *SIGSEGV* you can apply a similar technique as described above. This way, you can handle Segmentation Faults and proceed with JVM execution. In order to set the handler one must do following: define signal handler for signal SIGSEGV, setup new signal handler in the code, use sigsetjmp to handle incorrect call to exit, and eventually bring back original signal handler[15].

5 Conclusions

We have shown various techniques that can help people speed up and ease up Kepler based developments. All these techniques can be applied without too much effort but, at the same time, may lead to better understanding of problems that occur during development process. Of course, we presented just a small subset of various debugging techniques choosing solutions we find most useful and tightly related to Kepler based development.

References

[1] L. Di Gallo and C. Reux et al. "Coupling between a multiphysics workflow engine and an optimization framework". *Computer Physics Communications 00*. 2015, pp. 1–19.

[2] M. Płóciennik et al. "Tools, Methods and Services Enhancing the Usage of the Kepler-based Scientific Workflow Framework". *Procedia Computer Science, Volume 29*. 2014, pp. 1733–1744.

[3] M. Płóciennik et al. "Approaches to Distributed Execution of Scientific Workflows in Kepler". *Fundamenta Informaticae 128*. 2013, pp. 1–22.

[4] "The Open Group Base Specifications Issue 7". http://pubs.opengroup.org/onlinepubs/9699919799/nframe.html. 2013.

[5] "8 gdb tricks you should know". https://blogs.oracle.com/ksplice/entry/8_gdb_tricks_you_should. 2011.

[6] "Troubleshooting Guide for Java SE 6 with HotSpot VM". http://www.oracle.com/technetwork/java/javase/crashes-137240.html. 2010.

[7] Sheng Liang. *The Java Native Interface: Programmer's Guide and Specification*. 1999.

[13] atexit sample - https://github.com/mkopsnc/keplerhacks/tree/master/atexit
[14] dexit sample - https://github.com/mkopsnc/keplerhacks/tree/master/dexit
[15] SIGSEGV handler sample - https://github.com/mkopsnc/keplerhacks/tree/master/sigsegv