DEBUGGING ON FERMI

Debugging your application on a system based on a BG/Q architecture like FERMI could be an hard task due to the following problems:

- the core files generated by a crashing job on FERMI aren't easily readable;
- direct access to the back-end nodes is not straightforward.

In this document we discuss three useful tools that can help with debugging your applications on FERMI: **GDB**, **addr2line** and **Totalview**.

PREPARING A DEBUGGABLE APPLICATION

Three compilation flags are mandatory when compiling an application suited for debugging tools:

- **-g** : integrates debugging symbols on your code, making them “human readable” when analyzed from debuggers
- **-O0** : avoids any optimization on your code, making it execute the instructions in the exact order they’re implemented
- **-qfullpath** : Causes the full name of all source files to be added to the debug information (xl compilers only)

Other useful flags for xl compilers may be:

- **-qcheck / -qflttrap** : help detecting some particular errors at run-time, thus stopping the execution with a SIG-TRAP (the first flag deals with array-bound violation, the second with floating point exceptions)
- **-qhalt=<sev>**: stops the compilation if it encounters an error with the degree of severity you indicate. Possible degrees are: *i* (informational messages), *w* (warning messages), *e* (error messages, C only), *s* (severe error messages, default value)
- **-qkeepparm** : ensures that function parameters are stored on the stack even if the application is optimized. As a result, parameters remain in the expected memory location, providing access to the values of these incoming parameters to debuggers

GDB

GDB, the Gnu Project Debugger, is available as a default application for debugging both front-end or back-end applications, compiled either with gnu or xl. With GDB is possible to observe the behavior of your software at runtime, making it break on certain conditions and manipulate its parameters to look at the effects. It is also possible to make a post-mortem analysis, by making it read binary core files that can help with searching the point of your code where the problem arose. However, all these functionalities aren't fully exploitable on FERMI: in particular, since there is no support for parallel debugging, with GDB you can only analyze “serial” codes or single threads of your parallel code.
For general instructions about how to use GDB, you can check the following webpage (http://sourceware.org/gdb/current/onlinedocs/gdb/) or this quick reference (http://www.stanford.edu/class/cs107/other/gdbrefcard.pdf).

GDB on front-end nodes

For applications compiled for front-end execution, it's easy to launch GDB:
`gdb ./myexe`
Inside GDB, you can look at your program with all the commands and functionalities the debugger has to offer, from running the application real-time to read its source code at the moment of its crash.

GDB on back-end nodes

Back-end compiled applications need their specific version of GDB:
/bgsys/drivers/ppcfloor/gnu-linux/bin/powerpc64-bgq-linux-gdb ./myexe
Since you aren't actually running on a back-end node, you can't use GDB in its full functionality; more precisely, you can't run a simulation inside the debugger.

You can, however, analyze the **core file** created by your failing code, to detect the source line where your application has stopped. Such core files can't be created by GDB itself (since it can't execute your program), but can be generated by your job execution and then be used in order to debug your application "post-mortem". In order to generate the core files (that need to be in binary format), you have to add some environment variables to your *runjob* command line:
runjob --envs BG_COREDUMPONEXIT=1 --envs BG_COREDUMPBINARY= … : myexe*
You can specify the process whose core file you want to generate by replacing the asterisk (which means 'all the processes') with its rank number. If you want the core from more than one process, you can separate their ranks with commas:
runjob --envs BG_COREDUMPONEXIT=1 --envs BG_COREDUMPBINARY=1,3 … : myexe

Once you have the binary core file, you can launch GDB to analyze it, by specifying the core file as a parameter:
/bgsys/drivers/ppcfloor/gnu-linux/bin/powerpc64-bgq-linux-gdb ./myexe <corefile>
The debugger will point at the last instruction it tried to execute. You can use commands like "print", "list" or "where" in order to understand what happened to your software.

GDB on back-end nodes on running applications

It is possible to attach GDB to a back-end running process by associating to it a tool called "gdbserver". It can be associated only to one process (rank) at a time: for debugging multiple processes you have to launch multiple tools (up to 4) at a time.
Below is explained the complete procedure for attaching GDB to a running process. There is also an easier method provided by a module (see the following paragraph, "gdb-setup").

1) First of all, submit your job as usual;
llsubmit <jobscript>

2) Then, get your jobID (*fen04.29286.0* in the example);

```
llq -u $USER
Id                       Owner      Submitted   ST PRI Class        Running On
------------------------ ---------- ----------- -- --- ------------ -----------
fen04.29286.0            amarani0   12/18 12:19 R  50  parallel     fen06
```

3) Use it for getting the BG Job ID (`78473` in the example);
llq -l <jobID> | grep "Job Id"
A number called "BG Job ID" (different from the usual job ID) will be displayed.

```
llq -l  fen04.29286.0 | grep "Job Id"
          BG Job Id: 78473
```

4) Start the GDB-server tool;
start_tool --tool /bgsys/drivers/ppcfloor/ramdisk/distrofs/cios/sbin/gdbtool --args "--rank=<rank #> --listen_port=10000" --id <BG Job ID>
<rank #> is the rank of the process you want to attach to. The listen port value is the default one, and there is no reason for changing it.

```
start_tool --tool /bgsys/drivers/ppcfloor/ramdisk/distrofs/cios/sbin/gdbtool
           --args “--rank=0 --listen_port=10000” --id 78473
```

5) Get the IP address for your process (`10.15.0.81` in the example);
dump_proctable --id <BG Job ID> --rank <rank #> --host sn01-io
sn01-io is a default value, and it doesn't need to be changed. An IP address will be displayed, referring to the IO node of the card where your process is running.

```
dump_proctable --id 78473 --rank 0  --host sn01-io

Rank     I/O node IP address                    pid

0        10.15.0.81                             0x00000000
```

6) Launch GDB (back-end version);
/bgsys/drivers/ppcfloor/gnu-linux/bin/powerpc64-bgq-linux-gdb ./myexe

7) Inside GDB, connect remotely to your job process;
(gdb) target remote <IP address>:10000
<IP address> is the value you got from dump_proctable; 10000 is the default listen port.

```
(gdb) target remote 10.15.0.81:10000
```

You are now connected to your running application, on the process you asked for. Even this time GDB functionalities are limited and commands like 'run' or 'continue' won't work. You can, however, get information about the current state of your run (useful, for example, if you're stuck on a deadlock), with commands like “print”, “list” or “where”.

gdb-setup

You can complete the setup procedure for using GDB on running processes in an easier and more “user-friendly” way, by loading a proper module containing a script that executes automatically some of the steps of the procedure described above (*gdb-setup*) and a symbolic link to the back-end version of the debugger (*be-gdb*). In order to use it, you need to load the module “gdb”:
module load gdb
when the module is loaded, you can use the setup script with:
gdb-setup -j <job_id> -r <rank #>
where <*job_id*> is the jobID of the running job you want to debug and <*rank #*> is the rank (MPI task) of the specific process you want to examinate.

```
module load gdb
gdb-setup -j fen03.46368 -r 3
```

The script automatically finds the BG Job ID, launches gdbtool and finds the IP address of the I/O node with *dump_proctable*. Then, it prints the instructions for the connection from GDB:

```
Setup has finished succesfully! You can now start your debugging session following
these instructions:
Open gdb (back-end version), indicating the path and the name of the executable
running inside your job (what you want to debug):
> be-gdb ./myexe
Inside gdb, input the following:
```

```
  target remote 10.15.0.52:10019
WARNING: quitting gdb while remotely debugging will also kill the job!
You should open a different terminal if you want to analyze a different rank (i.e. a
different gdbtool) for the same job
```

You can now launch the back-end version of GDB (by using the short-cut provided by the module) and remotely connect to the job, with the “target remote” line printed by gdb-setup. All the functionality limitations caused by the back-end nodes still stand.

ADDR2LINE

Addr2line is a simple unix command that translates program addresses into file names and line numbers. It gets informations from core files to figure out the source file and the line on which your program stopped its execution.

When there are no particular environment variables specified, if an application stops abruptly a core file is generated, in lightweight text file format. It is quite hard to extract some informations from the simple reading of the file, but there is a section that can be interesting: it's the list of hexadecimal addresses written between the tags “+++**STACK**” and “**---STACK**”, in particular the ones under the column “**Saved Link Reg**”. They represent the function call chain up to the point your executable stopped. It is worth noticing that, in case of multithreading (OpenMP) applications, multiple function chains can be found in the core file, one for each thread.

This is the procedure for reading the hexadecimal addresses with addr2line:

1) Copy the lines of the “Saved Link Reg” column in a new text file;

2) Replace in such lines the first eight 0s with “0x” and save;

Example: `0000000001b2678 --> 0x018b2678`

3) Use addr2line giving in input the executable and this text file:

addr2line -e ./myexe < addresses.txt

A list of functions will be displayed, representing the history of the last nested calls your job executed until its crash. Each line provides the full path of the source code file where the function resides, and the line number of such file where it called the subsequent function (or crashed, if it is the last one on the list).

You can also ask addr2line to tell you the path for a single address, for example:

addr2line -e ./myexe 0x018b2678

a2l-translate

By loading the module “superc” you can use a Perl script, a2l-translate, capable to read the core file and generate the text file with the addresses already edited and ready to be used with addr2line. You can use it like this:

module load superc
a2l-translate <corefile>

You will then find in your directory one or more text files, one for each eventual OpenMP thread, called *<corefile>.t0* (or *.t1*, *.t2*,...) that you can give directly as an input to addr2line, as explained above in step 3.

TOTALVIEW

TotalView is a GUI-based source code-defect-analysis-tool that gives you control over processes and thread execution and visibility into program state and variables. It allows you to debug one or many processes and/or threads with complete control over program execution. You can reproduce and troubleshoot difficult problems that can occur in concurrent programs that take advantage of threads, OpenMP, MPI, or computational accelerators.
For more informations about Totalview's usage, you can check the official User Guide_ (http://www.roguewave.com/documents.aspx?entryid=451&command=core_download).

In order to launch a Totalview session on FERMI, you need to estabilish a tunneled connection to its VNC server, that opens a graphical display on your local machine. In order to do that, a specific procedure needs to be followed, and it's explained below.

Preparations

In order to use Totalview properly, one software (two for Windows Users) has to be installed in user's local workstation:

- The usage of Totalview from a remote host requires a **VNC** (Virtual Network Computing) connection. VNC is a graphical desktop sharing system to remotely control another computer. While the VNC Server is already placed on FERMI, the user has to download and install **VNCviewer**, available on its website (http://www.realvnc.com/download/viewer/).
- There are many ways to establish a remote connection on Windows, where “ssh” isn't available. What we suggest is to download **Cygwin**, a Linux-like environment for Windows (available at http://cygwin.com/setup.exe). During installation, be sure to select “**openSSH**” from the list of available packages (use its internal search engine if you don't find it in the packages list).

Estabilish a VNC connection

1) On FERMI, load tightvnc module;
module load tightvnc

2) Execute the script vncserver_wrapper;
vncserver_wrapper
VNCserver will be opened, and a display number will be assigned to you. Also, instructions will appear on the screen, containing all the informations necessary to proceed with the connection establishment.

Instructions example:

```
New display: 4

On your personal workstation, open a ssh tunneling connection:
MyPC# ssh -L 5904:localhost:5904 -L 5804:localhost:5804 -N
amarani0@login01.fermi.cineca.it

(don't wait for the prompt: the command has to continue until you finish your

work with VNC).

Windows users need to install Cygwin (Linux-like environment for Windows):

download setup.exe from www.cygwin.com , add Openssh package from the mirror you have
```

```
selected

On your personal workstation, either,

in another shell, run the vncviewer client:

MyPC# vncviewer localhost:4

(on your personal workstation the vnc client can have a different name)

OR

in your browser (with java plugin installed), look at

http://localhost:5804

WARNING: you need to export the DISPLAY variable in the job script

         to redirect X stream to your login/submit node:

         export DISPLAY=fen01:4

When your work is completed, you have to

1) close the vnc client on your pc
2) kill the vnc server on login01.fermi.cineca.it
login01.fermi.cineca.it# vncserver -kill :4
3) close the ssh tunneling connection by issuing a ^C.
```

NOTE: If it is the first time ever you use this module, you will be asked to choose a VNC password. Remember that password, since you will use it every time you open the VNC viewer on your local machine.

3) Open a local terminal on your machine (if Linux) or a Cygwin shell (if Windows). Copy/paste in there the following line:
ssh -L 59xx:localhost:59<xx> -L 58<xx>:localhost:58<xx> –N <username>@login<no>.fermi.cineca.it
<xx> is your VNC display number, <username> is your username on FERMI and <no> is the number of the front-end node you're logged into (01,02,07 or 08). You will find the correct line to copy/paste on your vncserver_wrapper instructions, with all the variables already settled.
NOTE: if asked for a password, input the password you normally use for connecting on FERMI.

```
ssh -L 5904:localhost:5904 -L 5804:localhost:5804 -N amarani0@login01.fermi.cineca.it
```

4) Open VNCViewer.
On Linux: open another local shell and type:
vncviewer localhost:<xx> (<xx> is your VNC display number, 04 in the example)
On Windows: double click on VNCviewer icon (created on your desktop by installation) and write "*localhost:<xx>*" when asked for the server (<xx> is your VNC display number, 04 in the example).
In both cases, you will be asked for your VNC password.

Set and launch a Totalview batch job

5) Inside your job script, you have to load the proper module and export the DISPLAY environment variable:
module load profile/advanced totalview
export DISPLAY=fen<no>:<xx>
where <xx> and <no> are as above (you'll find the correct DISPLAY name to export in vncserver_wrapper instructions)

```
module load profile/advanced/totalview
export DISPLAY=fen01:04
```
(also fen01:4 works)

6) Totalview execution line will be as follows:
totalview runjob –a <runjob arguments: --np, --exe, --args...>
after the mandatory flag -a you have to insert every runjob option and argument you need, in the same way you're used to.

```
totalview runjob -a -n 128 --ranks-per-node 4 --exe ./myexe.x
```

7) Submit your job. When it will start running, you will find a Totalview window opened on your VNCviewer display, where you can start your debugging session. Closing Totalview will also kill the job, and vice versa.

You can use Totalview with all the functionalities described in its User Guide (Replay Engine, Memory check...). Just remember, when specifying the parallel settings, to select “Blue Gene” as your parallel system, and to choose the number of tasks and nodes according to what you asked for on your job script.

WARNING: due to license issues, you are **NOT** allowed to run Totalview sessions with more than 1024 tasks simultaneously! An error message will appear if you try to do so. You can always check the status of the licenses with the command:
lmstat -c $LM_LICENSE_FILE -a

Close the VNC session

When you're done with your debugging session, please follow these instructions in order to close your VNC session properly:

1) Close VNCviewer on your local machine;

2) Kill the VNCserver on FERMI:
vncserver -kill :<x>
<x> is the usual VNC display number, without the initial 0 (if it was present). Once again, the correct command to input is written on your vncserver_wrapper instructions;

```
vncserver -kill :4
```
(:04 won't work. Also, notice the space between “-kill” and “:4”)

3) On your first local shell (Cygwin shell if Windows), close the ssh tunneling connection with CTRL+C.

Totalview Remote Display Client

An useful tool that helps in launching Totalview sessions in a rather easier way is the Remote Display Client (RDC), developed by Totalview staff.
RDC can be downloaded on your local workstation. It provides a simple graphical interface where, by setting a friendly page of parameters, it is possible to launch Totalview jobs bypassing completely the need of a VNC connection.
However, some problems have been encountered while making some tests on launching jobs with RDC: in fact, it looks like its functionality is strictly dependent on the workstation it is located at. Our System Administrators are currently working in order to fix this problem: this guide will be updated as soon as this issue will be resolved and connecting RDC to FERMI will become a completely portable procedure.