Debugging tools on the CINECA BG/Q (FERMI)

Andrew Emerson and Alessandro Marani
a.emerson@cineca.it, a.marani@cineca.it

Dept. SCAI
CINECA

Contents

- Introduction
- FERMI Architecture
- Compilation flags
- Core files and addr2line
- GDB
- Totalview
- Other tools
- Hands-on

Debugging on multi-core architectures

- Many programmers start off debugging by adding **`printf`** (C) or **`write(*,*)`** (FORTRAN);
- For multi-task, parallel MPI programs this becomes complex;
- Further complicated by multi-threaded codes – **`printf`** or **`write`** doesn't scale to multi-thousand core, threaded systems. (**`printf`** has been known to bring down filesystems in extreme cases.)

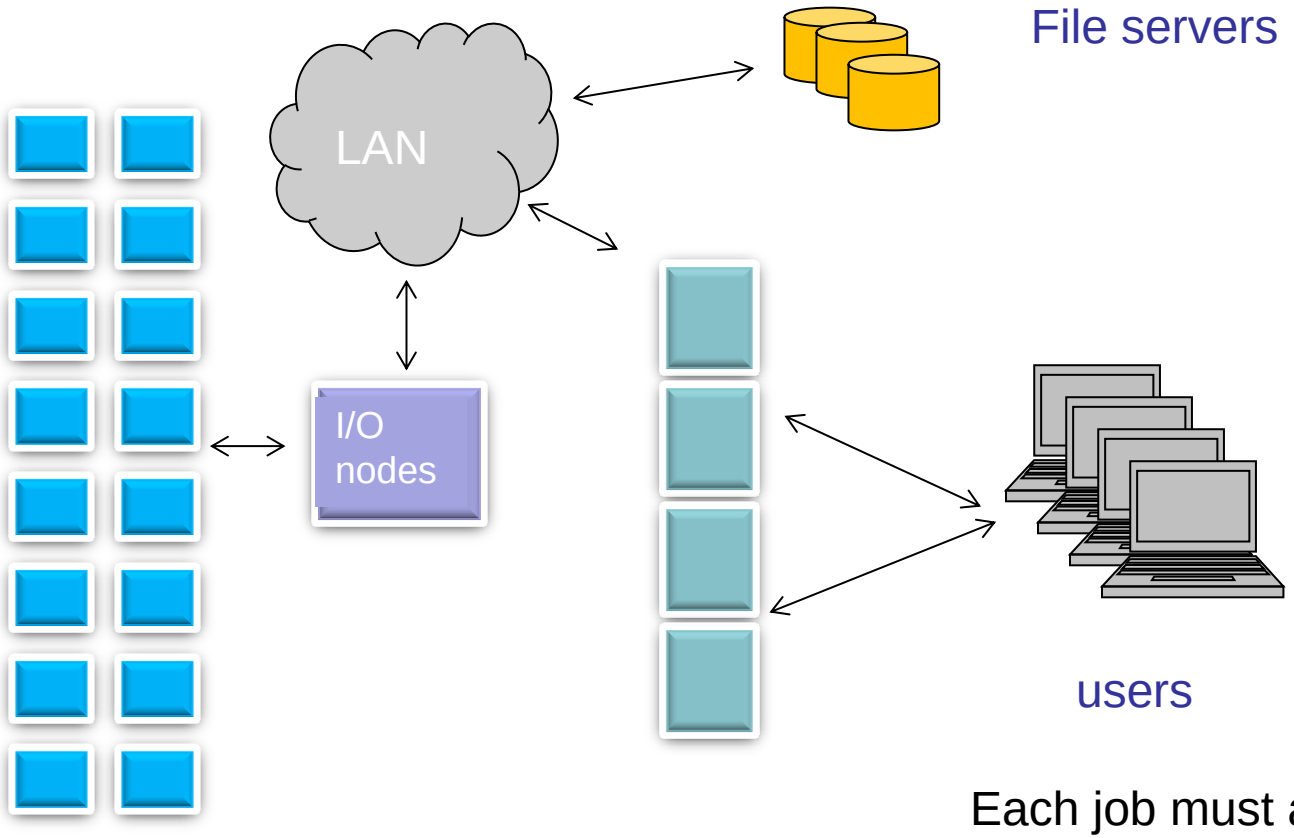
Debugging on FERMI – before you start

Finding program bugs on a BG/Q no exception:

- FERMI jobs may have many tens of thousands of cores;
- it is not possible to log directly on the compute nodes;
- error messages are often vague, and core files may be rather incomprehensible.

There are some tools that can help but you need to understand the system architecture first..

Simplified BG/Q architecture



Each job must at have at least one I/O node.
On BG/Q gdb debugging is done via the I/O nodes.

Compiling for a debug session

3 flags (IBM compiler) are required for compiling a program that can be analyzed by 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

Other useful flags

-qcheck Helps detecting some array-bound violations, aborting with SIGTRAP at runtime

-qflttrap Helps detecting some floating-point exceptions, aborting with SIGTRAP at runtime

-qhalt=<sev> Stops compilation if encountering an error of the specified lever of severity

-qformat Warns of possible problems with I/O format specification (C/C++) (printf,scanf…)

-qkeepparm ensures that function parameters are stored on the stack even if the application is optimized.

FERMI debugging tools

addr2line

GDB

GDB
The GNU Project Debugger

Totalview

TOTALVIEW
TECHNOLOGIES

Core files

Blue Gene core files are lightweight text files

Hexadecimal addresses in section STACK describe function call chain until program exception. It's the section delimited by tags: +++STACK / —STACK

```
+++STACK
Frame Address    Saved Link Reg
0000001fffff5ac0  000000000000001c
0000001fffff5bc0  00000000018b2678
0000001fffff5c60  00000000015046d0
0000001fffff5d00  00000000015738a8
0000001fffff5e00  00000000015734ec
0000001fffff5f00  000000000151a4d4
0000001fffff6000  00000000015001c8
---STACK
```

In particular, "Saved Link Reg" column is the one we need!

addr2line

If nothing is specified, an unsuccesful job generates a text core file for the processes that caused the crash…

…however, those core files are all but easily readable!

```
+++PARALLEL TOOLS CONSORTIUM LIGHTWEIGHT COREFILE FORMAT version 1.0
+++LCB 1.0
Program   : array_err.x
Job ID    : 61743
Personality:
   ABCDET coordinates : 0,0,0,0,1,1
   Rank               : 5
   Ranks per node     : 4
   DDR Size (MB)      : 16384
+++ID Rank: 5, TGID: 113, Core: 4, HWTID:0 TID: 113 State: RUN
***FAULT Encountered unhandled signal 0x00000005 (5) (SIGTRAP)
Generated by interrupt.....................0x00000005 (Program Exception IP=0x0000000001000438  ESR=0x0000000002000000: Trap )
While executing instruction at.............0x0000000001000438
Dereferencing memory at....................0x0000000000000000
Fault occurred at timebase.................0x00017349a2a0943d
Tools attached (list of tool ids).........None
Currently running on hardware thread....Y
General Purpose Registers:
  r00=0000000000000014 r01=0000001fbfffb9e0 r02=000000000015af7c0 r03=0000000000000005 r04=0000001fbfffba60 r05=0000000001548f60 r06=0000000001548f60 r07=000
000000000001
  r08=ffffffffeff0000 r09=000000000000005f r10=8080808080808080 r11=0000000000000000 r12=0000000010003f8 r13=0000001ec0507700 r14=0000000000000000 r15=000
0000000000
  r16=0000000000000000 r17=0000000000000000 r18=0000000000000000 r19=0000000000000000 r20=0000000000000000 r21=0000000000000000 r22=0000000000000000 r23=000
0000000000
  r24=0000000000000000 r25=0000000000000000 r26=000000000013cb4c0 r27=0000001fbfffc100 r28=0000000001565b90 r29=0000000000000000 r30=000000000015a1e68 r31=000
0000015a1e80
Special Purpose Registers:
  lr=0000000001000408 cr=0000000084000222 xer=0000000020000000 ctr=000000000000000d
  msr=000000008002f000 dear=0000000000000000 esr=0000000002000000 fpscr=0000000000008000
  sprg0=0000000000000000 sprg1=0000000000000000 sprg2=0000000000000000 sprg3=0000000000000000 sprg4=0000000000000000
  sprg5=0000000000000000 sprg6=000000000044e800 sprg7=0000000000000000 sprg8=0000000000000000
  srr0=0000000001000438 srr1=000000008002f000 csrr0=0000000000000000 csrr1=0000000000000000  mcsrr0=0000000000000000 mcsrr1=0000000000000000
  dbcr0=0000000000000000 dbcr1=0000000000000000 dbcr2=0000000000000000 dbcr3=0000000000000000 dbsr=0000000000000000
Floating Point Registers:
  f00=5500002000000000 1000008800200001  0000000000000000 0000000000000000  f01=0000000000000000 0000000000000000  0000000000000000 0000000000000000
  f02=0000000000000000 0000000000000000  0000000000000000 0000000000000000  f03=0000000000000000 0000000100000000  0000000000000000 0000000000000000
  f04=0000000000000000 0000000000000001  0000000000000001 0000000000000003  f05=0000000000000000 0000000000000000  0000000000000000 0000000000000000
  f06=0000000000000000 0000000000000000  0000000000000000 0000000000000000  f07=0000000000000000 0000000000000000  0000000000000000 0000000000000000
  f08=0000000000000000 0000000000000000  0000000000000000 0000000000000000  f09=0000000000000020 0000000000000020  0000000000000020 0000000000000020
  f10=4059000000000000 4059000000000000  4059000000000000 4059000000000000  f11=4059000000000000 4059000000000000  4059000000000000 4059000000000000
  f12=3fe0000000000000 3fe0000000000000  3fe0000000000000 3fe0000000000000  f13=0000000000000001 0000000000000000  0000000000000000 0000000000000000
  f14=0000000000000000 0000000000000000  0000000000000000 0000000000000000  f15=0000000000000000 0000000000000000  0000000000000000 0000000000000000
  f16=0000000000000000 0000000000000000  0000000000000000 0000000000000000  f17=0000000000000000 0000000000000000  0000000000000000 0000000000000000
  f18=0000000000000000 0000000000000000  0000000000000000 0000000000000000  f19=0000000000000000 0000000000000000  0000000000000000 0000000000000000
  f20=0000000000000000 0000000000000000  0000000000000000 0000000000000000  f21=0000000000000000 0000000000000000  0000000000000000 0000000000000000
  f22=0000000000000000 0000000000000000  0000000000000000 0000000000000000  f23=0000000000000000 0000000000000000  0000000000000000 0000000000000000
  f24=0000000000000000 0000000000000000  0000000000000000 0000000000000000  f25=0000000000000000 0000000000000000  0000000000000000 0000000000000000
  f26=0000000000000000 0000000000000000  0000000000000000 0000000000000000  f27=0000000000000000 0000000000000000  0000000000000000 0000000000000000
```

addr2line is an utility that permits to get from this file information about where the job crashed

using addr2line

From the core file output, save only the addresses in the Saved Link Reg column:

```
000000000000001c
00000000018b2678
00000000015046d0
00000000015738a8
00000000015734ec
000000000151a4d4
00000000015001c8
```

Replace the first eight 0s with 0x:

00000000018b2678 => 0x018b2678

If you load the module “superc”, a simple script called “a2l-translate” is capable of doing the replacement for you:

```
a2l-translate corefile
```

Launch addr2line:

```
addr2line -e ./myexe 0x018b2678
addr2line -e ./myexe < addresses.txt
```

GDB

GDB
The GNU Project Debugger

On FERMI, GDB is available both for front-end and back-end applications

Front-end: **`gdb <exe>`**
Back-end: **`/bgsys/drivers/ppcfloor/gnu-linux/bin/powerpc64-bgq-linux-gdb <exe>`**

GDB

It is possible to make a post-mortem analysis of the binary core files generated by the job:

```
/bgsys/drivers/ppcfloor/gnu-linux/bin/powerpc64-bgq-linux-gdb <exe> <corefile>
```

To generate binary core files, add the following envs to runjob:

```
--envs BG_COREDUMPONEXIT=1
--envs BG_COREDUMPBINARY=*
```

'*' means "all the processes". It is possible to indicate which ranks generate its core by specifying its number.

Note that core files are not always generated by errors related to insufficient memory (more memory allocated than available).

GDB – remote access

The Blue Gene/Q system includes support for using GDB real-time with applications running on compute nodes.

IBM provides a simple debug server called gdbserver. Each running instance of GDB is associated with one process or rank (also called GDB client).

Each instance of a GDB client can connect to and debug one process. To debug multiple processes at the same time, run multiple GDB tools at the same time. A maximum of four GDB tools can be run on one job.

…so, how to do that?

Using GDB on running applications

1) First of all, submit your job as usual;

```
llsubmit <jobscript>
```

2) Then, get your job ID;

```
llq -u $USER
```

3) Use it for getting the BG Job ID;

```
llq -l <jobID> | grep "Job Id"
```

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>
```

5) Get the IP address for your process;

```
dump_proctable --id <BG Job ID> --rank <rank #> --host sn01-io
```

Using GDB on running applications

6) Launch GDB! (back-end version);

```
/bgsys/drivers/ppcfloor/gnu-linux/bin/powerpc64-bgq-linux-gdb
./myexe
```

7) Connect remotely to your job process;

```
(gdb) target remote <IP address>:10000
```

8) Start debugging!!!

(Although you aren't completely free…for example, command 'run' does not work)

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.

Running a Totalview execution in back-end can be a bit tricky, as it requires connection from FERMI to your local machine via ssh tunneling to VNC server.

Using Totalview: preliminaries

In order to use Totalview, first you need to have downloaded and installed VNCviewer on your local machine.
(http://www.realvnc.com/download/viewer/)

Windows users will also find useful Cygwin, a Linux-like environment for Windows. During installation, be sure to select “openSSH” from the list of available packages.
(http://cygwin.com/setup.exe)

Once all the required softwares are installed, we are ready to start preparing our Totalview session!

Using Totalview: preparation

1) On FERMI, load tightvnc module;

```
module load tightvnc
```

2) Execute the script vncserver_wrapper;

```
vncserver_wrapper
```

3) Instructions will appear. Copy/paste to your local machine (Cygwin shell if Windows) this line from those instructions:

```
ssh -L 59xx:localhost:59xx -L 58xx:localhost:58xx -N
  <username>@login<no>.fermi.cineca.it
```

where xx is your VNC display number, and <no> is the number of the front-end node you're logged into (01,02,07 or 08)

4) Open VNCViewer. On Linux, use another local shell and type:

```
vncviewer localhost:xx
```

On Windows, double click on VNCviewer icon and write localhost:xx when asked for the server. Type your VNC password (or choose it, if it's your first visit)

Using Totalview: job script setting

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
```

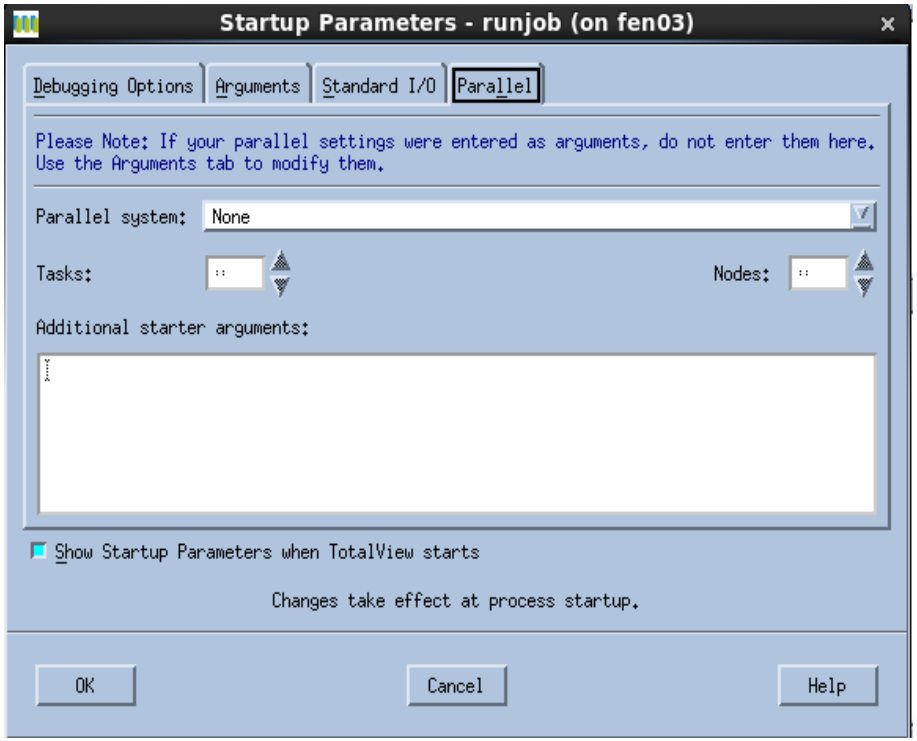
where xx and <no> are as the above slide (you'll find the correct DISPLAY name to export in vncserver_wrapper instructions)

6) Totalview execution line (inside your LoadLeveler script) will be as follows:

```
totalview runjob -a <runjob arguments: --np, --exe, --args…>
```

7) Launch the job. When it will start running, you will find a Totalview window opened on your VNCviewer display! Closing Totalview will also kill the job.

Using Totalview: start debugging



Select “BlueGene” as a parallel system, and a number of tasks and nodes according to the arguments you gave to runjob during submission phase.

Click “Go” (the green arrow) on the next screen and your application will start running.

WARNING: due to license issues, you are NOT allowed to run Totalview sessions with more than 1024 tasks simultaneously!!!

Out from Totalview

When you've finished using Totalview, please follow this procedure in order to close the session safely:

Close VNCviewer on your local machine;

2) Kill the VNCserver on FERMI:

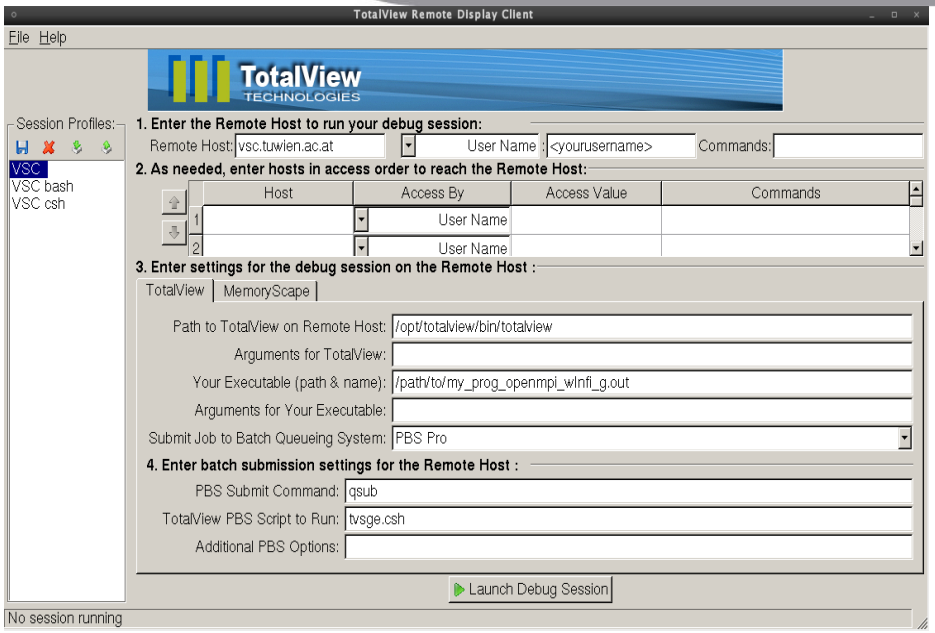
```
vncserver kill :x
```

x is the usual VNC display number, without the initial 0 (if present);

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

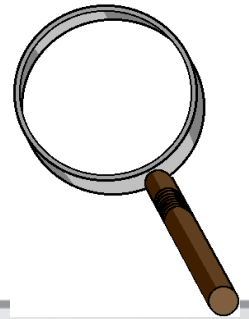
Totalview Remote Display Client

An easier (and maybe safer) way to use Totalview is Totalview RDC (Remote Display Client), a simple tool that helps with submitting a job already set with the proper characteristics (and with no VNC involved)



RDC procedure isn't fully operative yet, since we encountered some firewall issues that lead to different behaviours depending on the single workstation settings

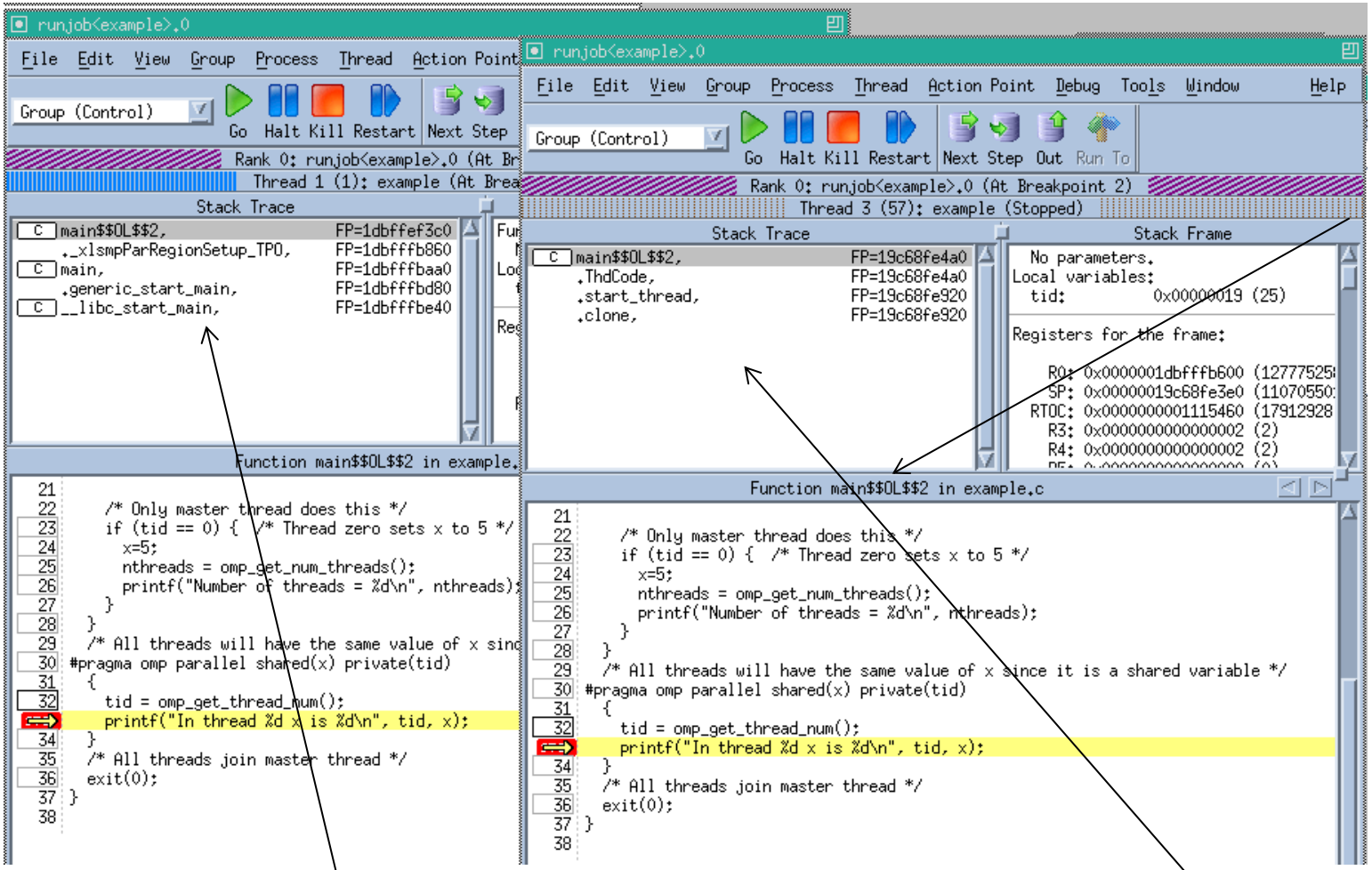
Our System Administrators are looking into it. Connecting with RDC will be soon a possibility!!



Totalview and Openmp programs

- Debugging OpenMP code is similar to multi-threaded (MPI) code.
- But since OpenMP compilers alters your code in particular ways (by creating ***outlined routines*** from parallel regions) there are things you should know:
  - Since a compiler can generate multiple outlined routines from a single parallel region a single line of source code can generate multiple machine blocks in different functions.
  - You can't step into /out of a parallel region. Set a breakpoint inside and let the process run into it. You can then single step in it.
  - For loops, best to use *static* scheduling rather than dynamic. Knowing which threads execute each iteration aids debugging.

Totalvew and OpenMP



Outlined routine name

Master thread

Worker thread

OpenMP Private and Shared variables

- Private variables are held by the compiler in the outlined routine and treated like local variables.
- Shared variables are kept in the stack frame of the master thread's routine.
- In Totalview you can view shared variables either by looking at the master thread or via an OpenMP worker thread. In the latter case Totalview will use the master thread context to find the shared variable.

Totalview licensing

Totalview costs money and we only have limited licenses:

- 1024 Totalview_Team tokens bluegene-power (FERMI);
- 128 Totalview_TeamPlus tokens for linux-x86 (PLX, ARPA);
- 128 CUDA tokens for linux-x86 for debugging with GPU (PLX);
- 128 Replay tokens for linux-x86 for Reverse Debugging with ReplayEngine.

1 core = 1 token so on FERMI it is easy to use the entire license!

Totalview licensing

Some considerations:

-Use the minimum number of cores and time necessary for the debugging;

-You can check the token usage of all users directly on Fermi:

- `module load totalview`
- `lmstat -c $LM_LICENSE_FILE -a`

According to experience, we may introduce limits to prevent excessive usage.

Other tools

Valgrind (PLX only)

- Useful for debugging the memory, e.g. memory leaks.

Allinea DDT (recently installed on Fermi)

- Graphical debugger
- Strong feature set ideal for hybrid applications.
  - Memory debugging
  - Data analysis

Debugging Hands-on

- *Now it's your turn....*