SCAI
SuperComputing Applications and Innovation

Debugging

Andrew Emerson, Paride Dagna and others
SCAI, Cineca

Contents

- Introduction
- Before using the debugger
  - checking tools
  - compiler options
- Preparing for the debugger
- Debugging a serial program with gdb
- Parallel Program debugging with gdb, PMPI and Totalview

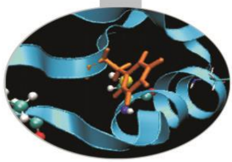
What is a bug ?

Sometimes called an “undocumented feature” but usually when something goes wrong..

As soon as we started programming, we found to our surprise that it wasn’t as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.!

Maurice Wilkes discovers debugging, 1949.

Notable software errors..



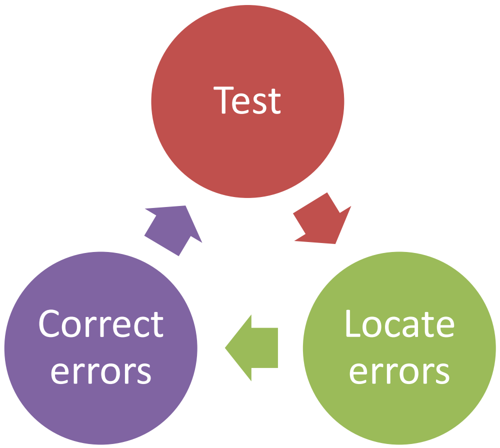
- Nasa Mars Climate Orbiter (1998, $125M)
  - Lost due to imperial <-> metric conversion
- Ariane 5 Flight 501 ($500M satellite)
  - Aborted due to non updated software (16bit variable for 64bit number).
- Heathrow Terminal 5
  - Bug in baggage handling control software. “Unexpected passenger behaviour” (i.e. retrieving something from a bag) brought down system.
- Mariner 1 spacecraft
  - Missing “-” caused craft to go of course and engineers aborted mission.
- “Smart ship” USS Yorktown was immobilised for 3 hours due to divide-by-zero.
- Amazon once had a book on sale about flies that cost **$23,698,655.93.**
  - Competing software agents raised the price to ca. $23M before someone noticed.

Taken from “10 of the most costly software errors in history, https://raygun.com/blog/10-costly-software-errors-history” ,Wikipedia and http://www.michaeleisen.org/blog/?p=358

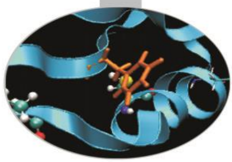
Avoiding bugs

- Don’t write them in the first place but good software practices help:
  - Structured or oriented programming techniques, minimising global variables, etc
  - Revisioning systems such as svn or git to keep track of changes and revert to working versions if things go wrong
  - Integrated Development Environments (IDEs) have many integrated features for writing programs.
  - HPC and parallel programming sometimes to leads to “performing code” instead of “safe code” so extra care is needed

For large programs it is worth developing *mini-apps*, smaller versions containing the main functionality



Practical debugging



- One of the most widely used methods to find out the reason of a strange behaviour in a program is the insertion of “printf” or “write” statements in the supposed critical area.
- However this kind of approach has a lot of limits and requires frequent code recompiling and becomes hard to implement for complex programs, above all if parallel. Moreover sometimes the error may not be obvious or hidden.
- Debuggers and other programs are very powerful tools able to provide, in a targeted manner, a high number of information facilitating the work of the programmer in research and in the solution of instability in the application.
- For example, with simple debugging commands you can have your program run to a certain line and then pause. You can then see what value any variable has at that point in the code.

Debugging process

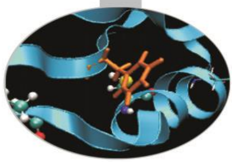
The debugging process can be divided into four main steps:

1. Start your program.
2. Make your program stop on specified conditions.
3. Examine what has happened, when your program has stopped.
4. Change things in your program, or its compilation, so you can experiment with correcting the effects of one bug and go on to learn about another.

Before starting the debugger

- Before starting the debugger, check your compiler documentation to see what compile or run-time checks are available.
- Some compiler options to try
  - switch down the optimisation level (e.g. from –O3). High or "aggressive" optimisations can cause code changes and introduce bugs.
  - turn on compiler options such as **`–C or –check-bounds`** to look for incorrect array indices.
  - for intel compilers try also –**`fpe0 –traceback`**, which stops the program if a floating point error is detected.
  - use options for uninitialised variable detection, etc.
- For performance reasons many run-time checks are switched off by default. Remember to switch them off again when debugging is complete.
- If possible also worth using a different compiler to see if the problem persists, or more useful error or warning messages are obtained.

Before starting the debugger..



Static Analysis tools for C, e.g.

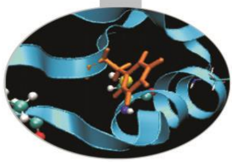
splint [-options] filename[s]

- Unused declarations
- Type inconsistences
- Infinite loops
- Possible Memory leaks
- many others

For Fortran tools include ftncheck, Forcheck, Cleanscape FortranLint,etc

SCAI SuperComputing Applications and Innovation

Before starting the debugger



Some advice

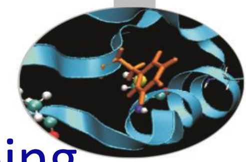
- Do not ignore compiler warnings, even if they appear to be harmless
- Use multiple compilers to check the code
- Try a static checker
- Re-run test cases frequently

Serious developers will use infrastructures implementating *Continuous Integration*, which run defined compilation and execution tests every time a new version of the code is uploaded to a repository. New code is not accepted until it passes all the tests.

Most popular debuggers

- Some debuggers are distributed with the compiler suite:
  - Commercial
    - Portland pgdbg
    - Intel
  - Free
    - Gnu gdb
- There are also some powerful, commercial debuggers from independent vendors:
  - DDT (Allinea)
  - Totalview (Rogue Wave Software)
  - Valgrind (particularly for Memory problems)

Debugger capabilities



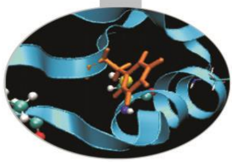
- The purpose of a debugger is to allow you to see what is going on “inside” another program while it executes or what another program was doing at the moment it crashed.
- Using specific commands, debuggers allow real-time visualization of variable values, static and dynamic memory state (stack, heap) and registers state.
- Common errors include:
  - pointer errors
  - array indexing
  - memory allocation
  - argument and parameter mismatches
  - communication deadlocks in parallel programming
  - I/O
- …

Compiling rules for debugging

- In order to debug a program effectively, the debugger needs debugging information which is produced compiling the program with the "-g" flag.
- This debugging information is stored in the object files fused in the executable; it describes the data type of each variable or function and the correspondence between source line numbers and addresses in the executable code.
- Opimization should be at –O0, -O1 or –O2 level.
- GNU compiler:
  - gcc/g++/gfortran –g [other flags] source –o executable
- INTEL compiler:
  - icc/icpc/ifort –g [other flags] source –o executable

SuperComputing Applications and Innovation

Execution



- The standard way to run the debugger is:
  - **debugger executable name or**
  - **debugger exe corefile**
- Otherwise it's possible to first run the debugger and then point to the executable to debug:

  GNU gdb:

gdb

> **file executable**

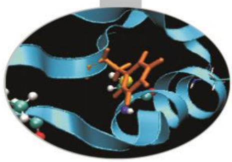
- It's also possible to debug an already-runnnig program started outside the debugger attaching to the process id of the program.
- Syntax:
- GNU gdb:

gdb

> **attach process_id**

gdb attach process_id

GDB command list



run: start debugged program

list: list specified function or line. Two arguments with comma between specify starting and ending lines to list.

```
list begin,end
```

break <line> <function> : set breakpoint at specified line or function, useful to stop execution before a critical point.

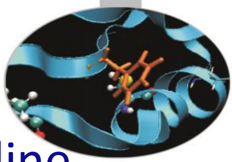
```
break filename:line
break filename:function
```

It's possible to insert a boolean expression with the sintax:

```
break <line> <function> condition
```

With no **<line> <function>**, uses current execution address of selected stack frame. This is useful for breaking on return to a stack frame.

GDB command list /2

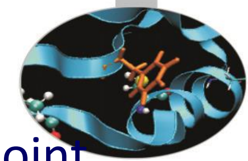


- **`clear <line> <func>`** : Clear breakpoint at specified line or function.
- **`delete breakpoints [num]`** : delete breakpoint number “num”. With no argument delete all breakpoints.
- **`If`** : Set a breakpoint with condition; evaluate the condition each time the breakpoint is reached, and stop only if the value is nonzero. Allowed logical operators: > , < , >= , <= , ==
- Example :

```
break 31 if i >= 12
```

- **`condition <num> < expression>`** : As the “if” command associates a logical condition at breakpoint number “num”.
- **`next <count>:`** continue to the next source line in the current (innermost) stack frame, or `count` lines.

GDB command list/3



`continue:` continue program being debugged, after signal or breakpoint

`where` : print backtrace of all stack frames, or innermost “count” frames.

`step` : Step program until it reaches a different source line. If used before a function call, allow to step into the function. The debugger stops at the first executable statement of that function

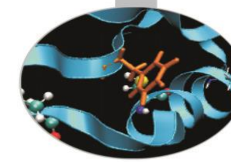
`step count` : executes **`count`** lines of code as the `next` command

`finish` : execute until selected stack frame or function returns and stops at the first statement after the function call. Upon return, the value returned is printed and put in the value history.

`set args` : set argument list to give program being debugged when it is started. Follow this command with any number of args, to be passed to the program.

`set var variable = <EXPR>:` evaluate expression `EXPR` and assign result to variable **`variable`**, using assignment syntax appropriate for the current language

GDB Command list/4



search <expr>: search for an expression from last line listed

reverse-search <expr> : search backward for an expression from last line listed

display <exp>: Print value of expression `exp` each time the program stops.

print <exp>: Print value of expression `exp`

This command can be used to display arrays:

`print array[num_el]` displays element `num_el`

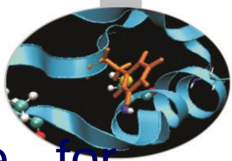
`print *array@len` displays the whole array

watch <exp>: Set a watchpoint for an expression. A watchpoint stops execution of your program whenever the value of an expression changes.

info locals: print variable declarations of current stack frame.

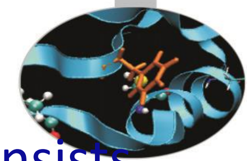
show values <number> : shows `number` elements of value history around item `number` or last ten.

GDB command list/5



- **backtrace** `<number,full>` **:** shows one line per frame, for many frames, starting with the currently executing frame (frame zero), followed by its caller (frame one), and on up the stack. With the `number` parameter print only the innermost `number` frames. With the `full` parameter print the values of the local variables also.
  - `#0 squareArray (nelem_in_array=12, array=0x601010) at variable_print.c:67`
  - `#1 0x00000000004005f5 in main () at variable_print.c:34`
- **frame** `<number>` **:** select and print a stack frame.
- **up** `<number>` **:** allow to go up `number` stack frames
- **down** `<number>` **:** allow to go up `number` stack frames
- **info** `frame` **:** gives all informations about current stack frame
- **detach:** detach a process or file previously attached.
- **quit:** quit the debugger

Using Core dumps for Postmortem Analysis



In computing, a core dump, memory dump, or storage dump consists of the recorded state of the working memory of a computer program at a specific time, generally when the program has terminated abnormally.

Core dumps are often used to assist in diagnosing and debugging errors in computer programs.

In most Linux Distributions core file creation is disabled by default for a normal user but it can be enabled using the following command :

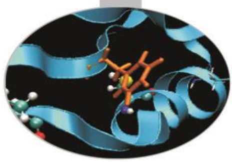
- **`ulimit -c unlimited`**

Once "ulimit –c" is set to "unlimited" run the program and the core file will be created

The core file can be analyzed with gdb using the following syntax:

- **`gdb -c core executable`**

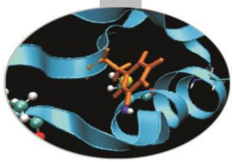
Parallel debugging



- Parallel debugging is more complex than serial because multiple processes need to be debugged simultaneously.
- Normally debuggers can be applied to multi-threaded parallel codes, containing OpenMP or MPI directives, or even OpenMP and MPI hybrid solutions.
- For OpenMP, the threads of a single program are akin to multiple processes except that they share one address space (that is, they can all examine and modify the same variables). On the other hand, each thread has its own registers and execution stack, and perhaps private memory.
- GDB provides some facilities for debugging OpenMP and MPI programs but usually a dedicated debugger such as Totalview is employed.

SCAI SuperComputing Applications and Innovation

Debugging OpenMP Applications



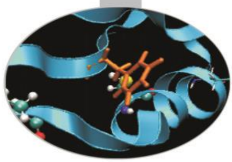
GDB facilities for debugging multi-threaded programs :

- automatic notification of new threads
- **thread <thread_number>** command to switch among threads
- **info threads** command to inquire about existing threads

```
(gdb) info threads

*   2 Thread 0x40200940 (LWP 5454)  MAIN__.omp_fn.0 (.omp_data_i=0x7fffffffd280)
    at serial_order_bug.f90:27

  1     Thread     0x2aaaaaf7d8b0     (LWP     1553)            MAIN__.omp_fn.0
    (.omp_data_i=0x7fffffffd280) at serial_order_bug.f90:27

thread apply <thread_number> <all> args allow to apply a command to apply a
    command to a list of threads.
```

- When any thread in your program stops, for example, at a breakpoint, all other threads in the program are also stopped by GDB.
- GDB cannot single-step all threads in lockstep. Since thread scheduling is up to your debugging target’s operating system (not controlled by GDB), other threads may execute more than one statement while the current thread completes a single step unless you use the command :set scheduler-locking on.
- GDB is not able to show the values of private and shared variables in OpenMP parallel regions.

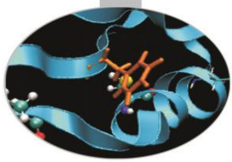
Debugging MPI applications



- Even more difficult than OpenMP since in principle could involve many thousands of tasks.
- Many MPI errors are possible including: invalid arguments, type matching, race conditions, deadlocks etc.
- Debugging communications is not easy. Some communication-related bugs may be hidden by MPI buffering such that they occur only for certain numbers of tasks or program inputs.
- Generally best to use the minimum no. of tasks necessary to reproduce the unexpected behaviour.

SuperComputing Applications and Innovation

Debugging MPI Applications



- There are two common ways to use serial debuggers such GDB to debug MPI applications

1. Attach to individual MPI processes after they are running using the “attach” method available for serial codes launching instances of the debugger to attach to the different MPI processes.

2. Open a debugging session for each MPI process through the command “mpirun”.

Debugging MPI Applications

Attach method

- Run the application in the usual way.

`mpirun -np 4 executable`

- From another shell, use the top command to find the MPI processes which bind to the executable:

```
top - 15:06:40 up 91 days,  4:00,  1 user,  load average: 5.31, 3.34, 2.66
Tasks: 198 total,   9 running, 188 sleeping,   0 stopped,   1 zombie
Cpu(s): 97.4%us,  2.3%sy,  0.0%ni,  0.2%id,  0.0%wa,  0.0%hi,  0.1%si,  0.0%st
Mem:  16438664k total,  3375504k used, 13063160k free,    72232k buffers
Swap: 16779884k total,    48328k used, 16731556k free,  1488208k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
12515 dagna     25   0  208m  10m 4320 R 99.8  0.1   0:10.23 Isola_MPI_2_inp
12516 dagna     25   0  208m  10m 4312 R 99.8  0.1   0:10.23 Isola_MPI_2_inp
12514 dagna     25   0  208m  10m 4320 R 99.5  0.1   0:10.15 Isola_MPI_2_inp
12513 dagna     25   0  235m  18m 4656 R 97.5  0.1   0:09.97 Isola_MPI_2_inp
 6244 dagna     15   0 82108 2660 1904 S  0.0  0.0   0:00.08 bash
 6428 dagna     15   0  101m 2472 1296 S  0.0  0.0   0:00.06 sshd
 6429 dagna     15   0 82108 2668 1908 S  0.0  0.0   0:00.08 bash
12512 dagna     15   0 74500 3396 2420 S  0.0  0.0   0:00.03 mpirun
12549 dagna     15   0 28792 2184 1492 R  0.0  0.0   0:00.01 top
```

PID executable MPI processes

Debugging MPI Applications

- Run up to “n” instances of the debugger in “attach” mode, where n is the number of the MPI processes of the application. Using this method you should have to open up to n shells.

- Referring to the previous slide we have to run four instances of GDB:

```
gdb attach 12513 (shell 1)
gdb attach 12514 (shell 2)
gdb attach 12515 (shell 3)
gdb attach 12516 (shell 4)
```

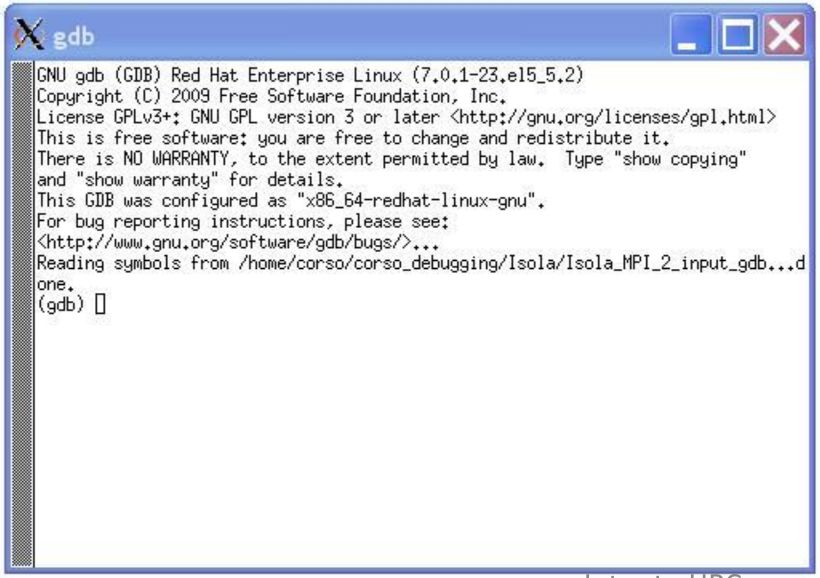
- Use debugger commands for each shell as in the serial case

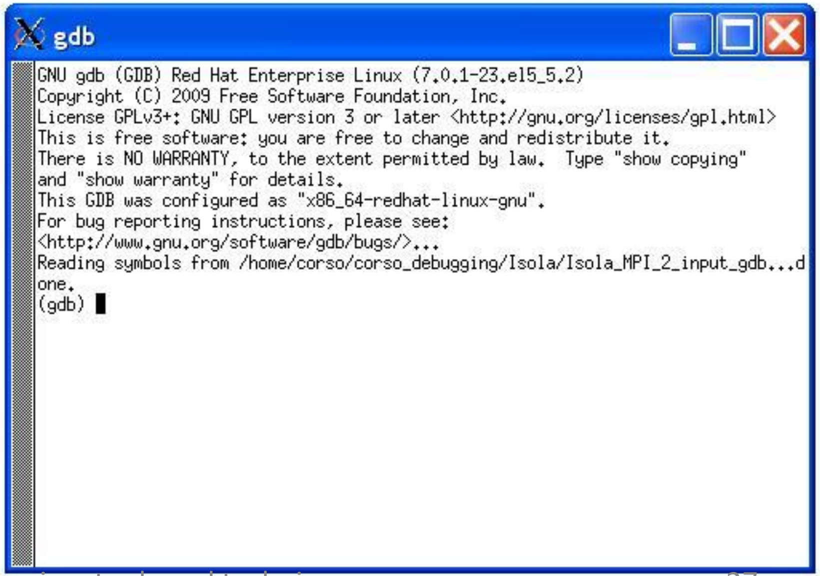
Debugging MPI Applications

- mpirun method
  - This technique launches a separate window for each MPI process in MPI_COMM_WORLD, each one running a serial instance of GDB that will launch and run your MPI application.

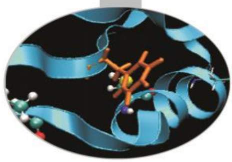
```
mpirun -np 2 xterm -e gdb nome_eseguibile
```

```
[corso@corsi110 Isola]$ mpirun -np 2 xterm -e gdb ./Isola_MPI_2_input_gdb
```





MPI Run-time diagnostics



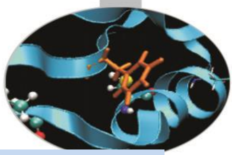
- Somtimes useful to know how the MPI tasks were created and on which physical nodes they were created (*binding*).

```
#!/bin/bash
#PBS -l walltime=30
#PBS -l select=2:ncpus=4:mpiprocs=4
#PBS -A cin_staff
#PBS -o out
#PBS -e err

cd $PBS_O_WORKDIR
module load autoload openmpi
mpirun --display-allocation --display-
    map exe
```

```
======================   ALLOCATED NODES
===============

 Data for node: Name: node102   Num slots: 4    Max slots: 0
 Data for node: Name: node103ib0       Num slots: 4    Max slots:
0

=================================================================
 ========================   JOB MAP
=====================
 Data for node: Name: node102   Num procs: 4
        Process OMPI jobid: [38452,1] Process rank: 0
        Process OMPI jobid: [38452,1] Process rank: 1
        Process OMPI jobid: [38452,1] Process rank: 2
        Process OMPI jobid: [38452,1] Process rank: 3

 Data for node: Name: node103ib0       Num procs: 4
        Process OMPI jobid: [38452,1] Process rank: 4
        Process OMPI jobid: [38452,1] Process rank: 5
        Process OMPI jobid: [38452,1] Process rank: 6
        Process OMPI jobid: [38452,1] Process rank: 7
```

openmpi

MPI Run-time diagnostics



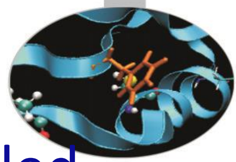
```
#!/bin/bash
#PBS -l walltime=30
#PBS -l select=2:ncpus=4:mpiprocs=4
#PBS -A cin_staff
#PBS -o out
#PBS -e err

cd $PBS_O_WORKDIR
module load autoload intelmpi

export I_MPI_DEBUG=5
mpirun ./spawnexample
```

[0] MPI startup(): Rank	Pid	Node name	Pin cpu
[0] MPI startup(): 0	18836	node102	{0,1,2}
[0] MPI startup(): 1	18837	node102	{3,4,5}
[0] MPI startup(): 2	18838	node102	{6,7,8}
[0] MPI startup(): 3	18839	node102	{9,10,11}
[0] MPI startup(): 4	32649	node103	{0,1,2}
[0] MPI startup(): 5	32650	node103	{3,4,5}
[0] MPI startup(): 6	32651	node103	{6,7,8}
[0] MPI startup(): 7	32652	node103	{9,10,11}

Intel mpi

Also possible via the **`MPI_Get_processor_name`** function call

Debugging MPI with PMPI



- MPI implementations also provide a profiling interface called PMPI.
- In PMPI each standard MPI function (MPI_) has an equivalent function with prefix PMPI_ (e.g. PMPI_Send, PMI_RECV, etc).
- With PMPI it is possible to customize normal MPI commands to provide extra information useful for profiling or debugging.
- Not necessary to modify source code since the customized MPI commands can be linked as a separate library during debugging. For production the extra library is not linked and the standard MPI behaviour is used.

SuperComputing Applications and Innovation

PMPI Examples

Profiling

```
// profiling example
static int send_count=0;
int MPI_Send(void*start,int count, MPI_Datatype datatype, int dest,
   int tag, MPI_Comm comm)
{
send_count++;
return PMPI_Send(start, count, datatype, dest, tag, comm);
}
```

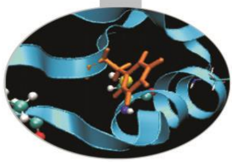
Debugging

```
! Unsafe uses of MPI_Send
! MPI_Send can be implemented as MPI_Ssend (synchronous send)
subroutine MPI_Send( start, count, datatype, dest,
 tag, comm, ierr )
 integer start(*), count, datatype, dest, tag, comm
 call PMPI_Ssend( start, count, datatype,
 dest, tag, comm, ierr )
end
```

Debugging MPI with totalview and RCM

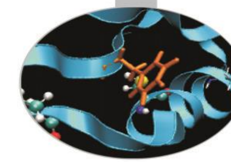
- Totalview is a powerful, sophisticated, programmable tool for debugging serial or parallel programs.
- Being a graphical tool, for best results recommended to use a remote visualization tool such as RCM (Remote Connection Manager), rather than just an X-display (slow).
- It is also a commercial product, so licenses are limited!

Debugging MPI with Totalview and RCM



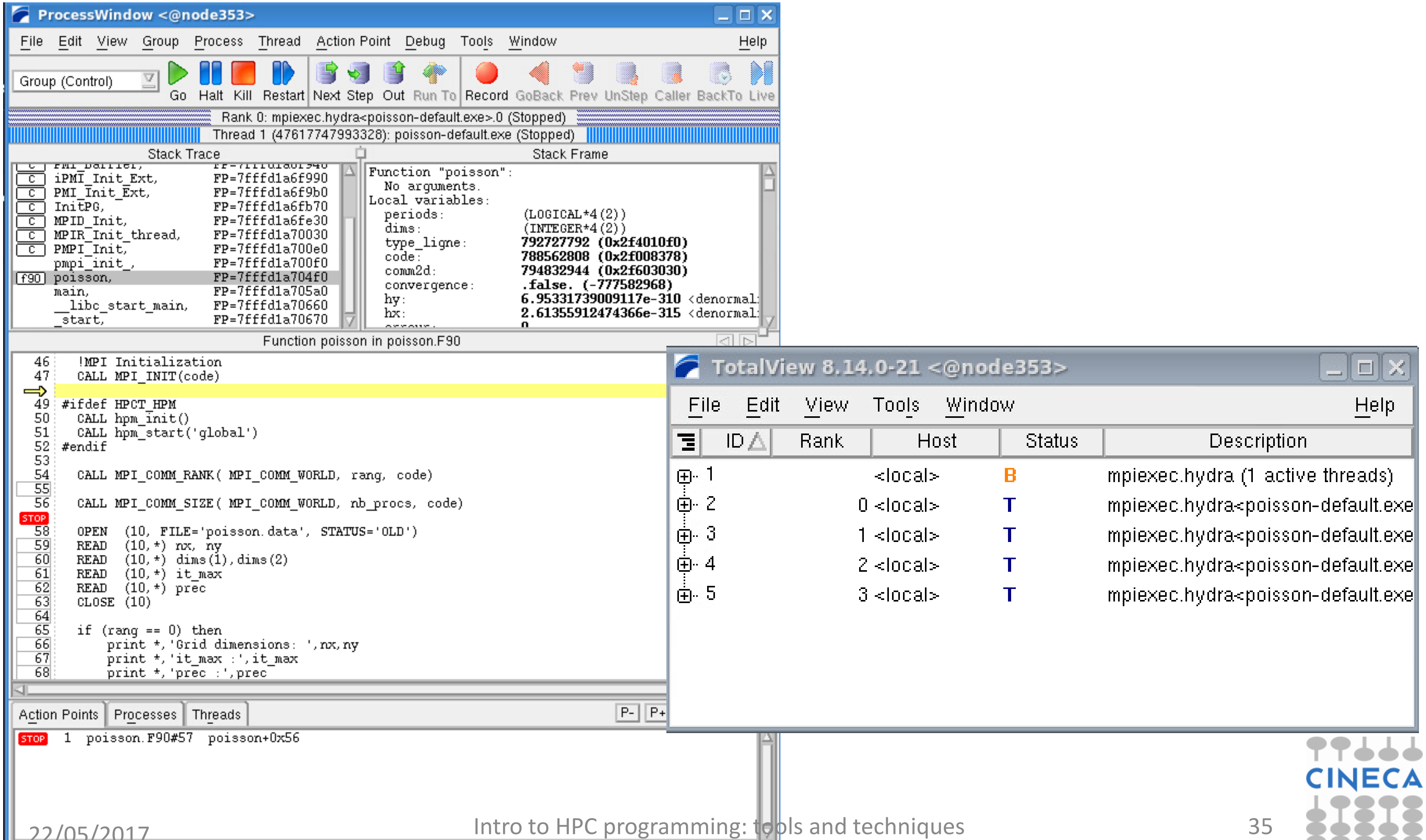
1. Download and install RCM on workstation:
   http://www.hpc.cineca.it/content/remote-visualization-rcm
2. Launch RCM and log on to Marconi. You will be given a Linux-style desktop.
3. Open a terminal and prepare a PBS/Loadleveler job script. Insert the DISPLAY number in the job script. Or open an interactive PBS session.

Debugging MPI with totalview and RCM - Example



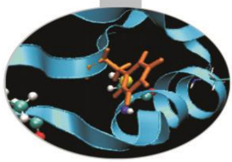
```
#!/bin/bash

#PBS -l walltime=00:30:00
#PBS -l select=1:ncpus=4:mpiprocs=4:mem=15gb
#PBS -N totalview
#PBS -o job.out
#PBS -e job.err
#PBS -q debug
####    account number (type saldo -b)
#PBS -A your_account_here

module load profile/advanced
module load autoload openmpi/1.6.3--gnu--4.7.2
module load totalview/8.12.0-1
  export DISPLAY=node097:1

  cd $PBS_O_WORKDIR
  mpirun -tv -n 4 poisson.exe
```

Debugging MPI with totalview and RCM



SuperComputing Applications and Innovation

Summary



- All programs have bugs.
- Parallel programs are particularly difficult because of the need to debug multiple processes and the interactions between them.
- A debugging strategy should include:
  - compiler options to lower side-effects of optimisation and increase the level of compile-time and run-time checking.
  - Static analysis tools
  - post-mortem analysis of stack traces and core files
  - run-time diagnostic options
  - the use of debuggers such as gdb or Totalview
  - in tandem with profilers or similar tools to understand better what the program is doing