

Finding Errors in HPC Applications

Andrew Emerson
SCAI, Cineca



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

- 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 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.

- Serial debuggers
 - gdb, dbx, idb, cuda-gdb etc
 - Clang static analyzer
- Memory debugging
 - Valgrind
 - cuda-memcheck (GPUs)
 - Intel Inspector*
- Parallel debuggers
 - Totalview *
 - DDT (Allinea)*
 - Intel Parallel Studio *

A wide range of tools are available, even though the most sophisticated ones tend to be commercial products. *

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.

- 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.
- Optimization 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`

- 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-running program started outside the debugger attaching to the process id of the program.
- Syntax:
- GNU gdb:

gdb

> attach process_id

gdb attach process_id

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 syntax:

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.

- **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.

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

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.

- **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 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.

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=0x7fffffffdd280)
   at serial_order_bug.f90:27
  1 Thread 0x2aaaaaf7d8b0 (LWP 1553)  MAIN__omp_fn.0
   (.omp_data_i=0x7fffffffdd280) 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.

- In the following OpenMP code, using the SECTIONS directive, two threads initialize their own array and then sum it to the other

```
PROGRAM lock
  INTEGER*8 LOCKA, LOCKB
  INTEGER NTHREADS, TID, I, OMP_GET_NUM_THREADS, OMP_GET_THREAD_NUM
  PARAMETER (N=1000000)
  REAL A(N), B(N), PI, DELTA
  PARAMETER (PI=3.1415926535)
  PARAMETER (DELTA=.01415926535)

  CALL OMP_INIT_LOCK(LOCKA)
  CALL OMP_INIT_LOCK(LOCKB)

  !$OMP PARALLEL SHARED(A, B, NTHREADS, LOCKA, LOCKB) PRIVATE(TID)

    TID = OMP_GET_THREAD_NUM()
  !$OMP MASTER
    NTHREADS = OMP_GET_NUM_THREADS()
    PRINT *, 'Number of threads = ', NTHREADS
  !$OMP END MASTER
    PRINT *, 'Thread', TID, 'starting...'
  !$OMP BARRIER
```

Debug openMP applications



```
!$OMP SECTIONS
!$OMP SECTION
PRINT *, 'Thread',TID,' initializing A()'
CALL OMP_SET_LOCK(LOCKA)
DO I = 1, N
  A(I) = I * DELTA
ENDDO
CALL OMP_SET_LOCK(LOCKB)
PRINT *, 'Thread',TID,' adding A() to B()'
DO I = 1, N
  B(I) = B(I) + A(I)
ENDDO
CALL OMP_UNSET_LOCK(LOCKB)
CALL OMP_UNSET_LOCK(LOCKA)
```

```
!$OMP SECTION

PRINT *, 'Thread',TID,' initializing B()'
CALL OMP_SET_LOCK(LOCKB)
DO I = 1, N
  B(I) = I * PI
ENDDO
CALL OMP_SET_LOCK(LOCKA)
PRINT *, 'Thread',TID,' adding B() toA()'
DO I = 1, N
  A(I) = A(I) + B(I)
ENDDO
CALL OMP_UNSET_LOCK(LOCKA)
CALL OMP_UNSET_LOCK(LOCKB)

!$OMP END SECTIONS NOWAIT

PRINT *, 'Thread',TID,' done.'

!$OMP END PARALLEL

END
```

Debugging OpenMP Applications



- **Compiling:**

```
gfortran -fopenmp -g -o omp_debug omp_debug.f90
```

- **Execution:**

- `export OMP_NUM_THREADS=2`

- `./omp_debug`

- The program produces the following output before hanging:

```
Number of threads =                2
```

```
Thread                0 starting...
```

```
Thread                1 starting...
```

```
Thread                0  initializing A()
```

```
Thread                1  initializing B()
```

- In the debugger:
 - List the source code from line 10 to 50:
 - Insert breakpoint at beginning of parallel region and run:

```
list 10,50  
b 20  
run
```

```
2 Thread 0x40200940 (LWP 8533)  MAIN__omp_fn.0  
  (.omp_data_i=0x7fffffffed2b0) at  
  openmp_bug2_nofix.f90:20
```

```
1 Thread 0x2aaaaaf7d8b0 (LWP 8530)  MAIN__omp_fn.0  
  (.omp_data_i=0x7fffffffed2b0) at  
  openmp_bug2_nofix.f90:20
```

- The print statements aren't executed so insert breakpoints in the two sections:

```
thread apply 2 b 35  
thread apply 1 b 49
```

- Restart execution:

```
thread apply all cont
```

- Execution hangs so ctrl-c and check where threads are:

```
thread apply all where
```

```
Thread 2 (Thread 0x40200940 (LWP 8533)) :  
0x00000000004010b5 in MAIN__.omp_fn.0  
(.omp_data_i=0x7ffffffffffd2b0) at  
openmp_bug2_nofix.f90:29
```

```
Thread 1 (Thread 0x2aaaaaf7d8b0 (LWP 8530)) :  
0x0000000000400e6d in MAIN__.omp_fn.0  
(.omp_data_i=0x7ffffffffffd2b0) at  
openmp_bug2_nofix.f90:43
```

Debugging OpenMP Applications



- Thread number 2 is stopped at line 29 on the statement:

```
CALL OMP_SET_LOCK (LOCKB)
```

- Thread number 1 is stopped at line 43 on the statement :

```
CALL OMP_SET_LOCK (LOCKA)
```

- So it's clear that the bug is in the calls to routines `OMP_SET_LOCK` that cause execution stopping
- Looking at the order of the routine calls to `OMP_SET_LOCK` and `OMP_UNSET_LOCK` it is clear there is an error.
- The correct order provides that the call to `OMP_SET_LOCK` must be followed by the corresponding `OMP_UNSET_LOCK`
- Arranging the order the code finishes successfully

- 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.

- 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”.

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

- 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:

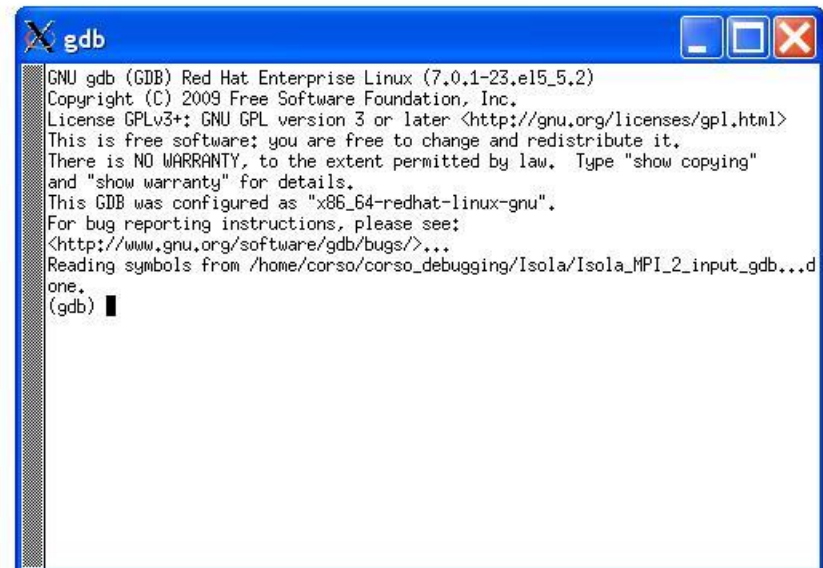
```
gdb attach 12513 (shell 1)
gdb attach 12514 (shell 2)
gdb attach 12515 (shell 3)
gdb attach 12516 (shell 4)
```
- Use debugger commands for each shell as in the serial case

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@corsill10 Isola]$ mpirun -np 2 xterm -e gdb ./Isola_MPI_2_input_gdb
```

MPI Run-time diagnostics



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

```
#!/bin/bash
```

```
#SBATCH --time=30
```

```
#SBATCH --tasks=4
```

```
module load autoload openmpi
```

```
mpirun --display-allocation --  
display-map exe
```

openmpi

```
===== 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
```

MPI Run-time diagnostics



```
#!/bin/bash
```

```
#SBATCH --time=30
```

```
#SBATCH --tasks=8
```

```
module load autoloader 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

- 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, PMPI_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.

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 with totalview



ProcessWindow <@node353>

File Edit View Group Process Thread Action Point Debug Tools Window Help

Group (Control) Go Halt Kill Restart Next Step Out Run To Record GoBack Prev UnStep Caller BackTo Live

Rank 0: mpiexec.hydra<poisson-default.exe>.0 (Stopped)
Thread 1 (47617747993328): poisson-default.exe (Stopped)

Stack Trace

Function	FP
PMI_Init	FP=7fffd1a6f990
iPMI_Init_Ext	FP=7fffd1a6f990
PMI_Init_Ext	FP=7fffd1a6f9b0
InitPG	FP=7fffd1a6fb70
MPID_Init	FP=7fffd1a6fe30
MPID_Init_thread	FP=7fffd1a70030
PMPI_Init	FP=7fffd1a700e0
pmi_init	FP=7fffd1a700f0
poisson	FP=7fffd1a704f0
main	FP=7fffd1a705a0
_libc_start_main	FP=7fffd1a70660
_start	FP=7fffd1a70670

Stack Frame

Function "poisson":
No arguments.
Local variables:
periods: (LOGICAL*4 (2))
dims: (INTEGER*4 (2))
type_ligne: 792727792 (0x2f4010f0)
code: 788562808 (0x2f008378)
comm2d: 794832944 (0x2f603030)
convergence: .false. (-777582968)
hy: 6.95331739009117e-310 <denormal:
hx: 2.61355912474366e-315 <denormal:
error: 0

Function poisson in poisson.F90

```

46 !MPI Initialization
47 CALL MPI_INIT(code)
48
49 #ifdef HPCT_HPM
50 CALL hpm_init()
51 CALL hpm_start('global')
52 #endif
53
54 CALL MPI_COMM_RANK( MPI_COMM_WORLD, rang, code)
55
56 CALL MPI_COMM_SIZE( MPI_COMM_WORLD, nb_procs, code)
57
58 OPEN (10, FILE='poisson.data', STATUS='OLD')
59 READ (10,*) nx, ny
60 READ (10,*) dims(1), dims(2)
61 READ (10,*) it_max
62 READ (10,*) prec
63 CLOSE (10)
64
65 if (rang == 0) then
66   print *, 'Grid dimensions: ', nx, ny
67   print *, 'it_max : ', it_max
68   print *, 'prec : ', prec

```

Action Points Processes Threads

STOP 1 poisson.F90#57 poisson+0x56

TotalView 8.14.0-21 <@node353>

File Edit View Tools Window Help

ID	Rank	Host	Status	Description
1	<local>	<local>	B	mpiexec.hydra (1 active threads)
2	0 <local>	<local>	T	mpiexec.hydra<poisson-default.exe>
3	1 <local>	<local>	T	mpiexec.hydra<poisson-default.exe>
4	2 <local>	<local>	T	mpiexec.hydra<poisson-default.exe>
5	3 <local>	<local>	T	mpiexec.hydra<poisson-default.exe>

- All programs have bugs.
- Parallel programs are particularly difficult because of the need to debug multiple processes and possibly, 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.
 - 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