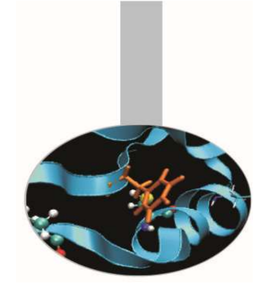
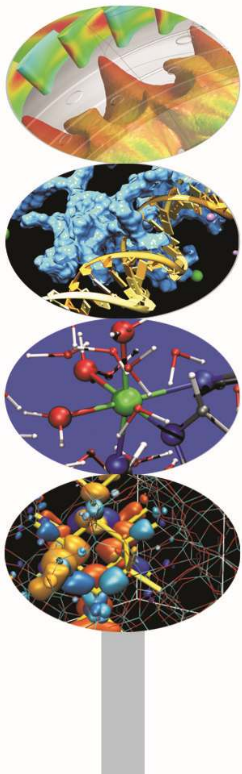
CINECA SCAI
SuperComputing Applications and Innovation



Debugging

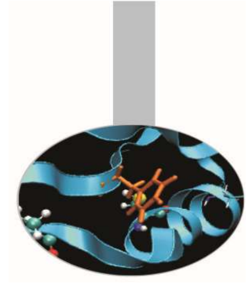


P.Dagna, M.Cremonesi

May 2015

CINECA

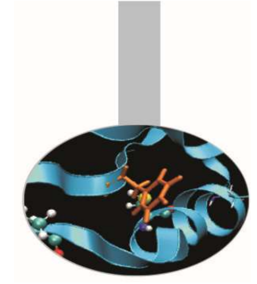
Introduction



One of the most widely used methods to find out the reason of a strange behavior 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.

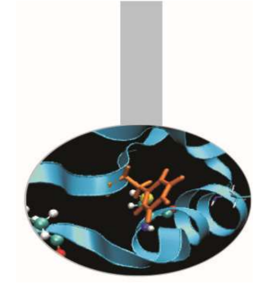
Introduction



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 three 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.

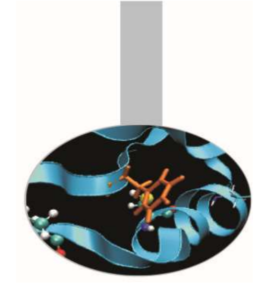
Debugging process



The debugging process can be divided into four main steps:

1. Start your program, specifying anything that might affect its behavior.
2. Make your program halt on specified conditions.
3. Examine what has happened, when your program has stopped.
4. Correct the program and go on to learn about another possible bug.

Most popular debuggers



Debuggers are generally distributed within the compiler suite.

Commercial:

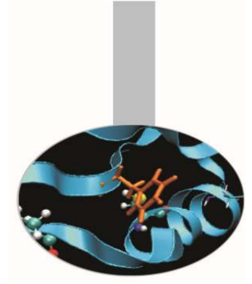
- Portland pgdbg
- Intel idb

Free:

- GNU gdb

Moreover there are companies specialized in the production of very powerful debuggers , among them most popular are:

- Allinea DDT
- Totalview

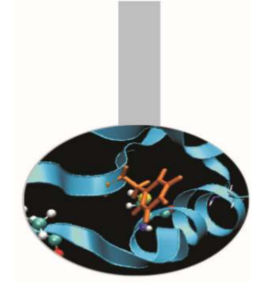


Debugger capabilities

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.

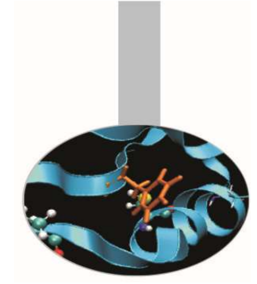
Debugger capabilities



Most common errors are:

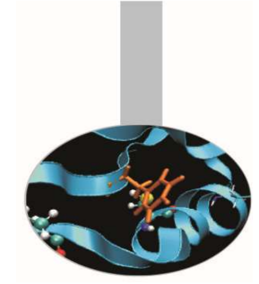
1. pointer errors
2. array indexing errors
3. allocation errors
4. dummy and actual arguments mismatch in calling routines
5. infinite loops
6. I/O errors

Compiling for debugging



To debug a program effectively, the debugger needs debugging information which is produced compiling the program with the "-g" flag.

Debugging information is stored in the object file; it describes the data type of each variable or function and the correspondence between source line numbers and addresses in the executable code.



Compiling for debugging

- **GNU compiler**:

  `gcc/g++/gfortran -g [other flags] source -o executable`

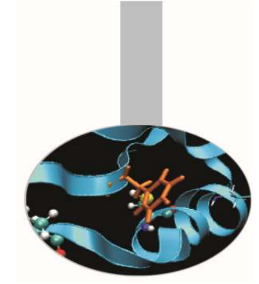
- **PGI compiler**:

  `pgcc/pgCC/pgf90 -g [other flags] source -o executable`

- **INTEL compiler**:

  `icc/icpc/ifort -g [other flags] source -o executable`

Execution



The **standard way** of running the debugger is:

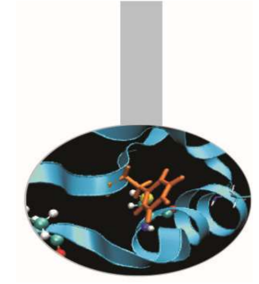
- `debugger_name executable`

Otherwise it's possible to first run the debugger and then point to the executable to debug:

- GNU gdb:
  - `gdb`

    `> file executable`

SCAI
SuperComputing Applications and Innovation



Execution

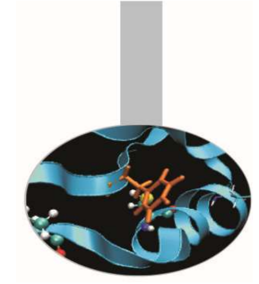
It's even possible to **debug an already-runnig** program started outside the debugger **attaching** to the **process id** of the program.

- GNU gdb:
  - gdb

    > attach process_id
  - gdb attach process_id

CINECA

Command list



- `run:` start program to be debugged
- `list:` list specified function or line. Two arguments parted by a comma specify starting and ending lines to list.

  ```
  list begin,end
  ```

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

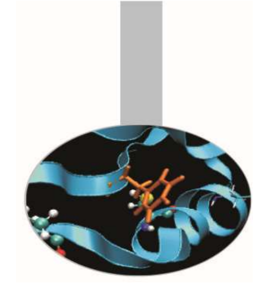
  ```
  break filename:line
  break filename:function
  ```

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.

Command list

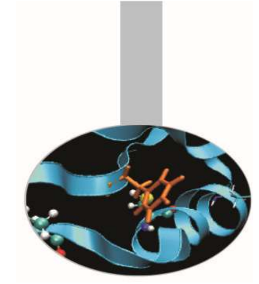


- `clear <line> <func>`: Clear a breakpoint at specified line or function.
- `delete breakpoints [num]` : delete breakpoint number “num”. With no argument delete all breakpoints.
- `If:` Set a breakpoint with condition; evaluate the condition each time the breakpoint is reached, and stop only if the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

  Example :

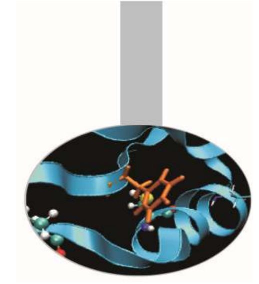
  ```
  break 31 if i >= 12
  ```
- `condition <num> < expression>` : As the “if” command associates a logical condition at breakpoint number “num”.
- `next <count>`: continue to the next source line in the current (innermost) stack frame, or `count` lines.

Command list

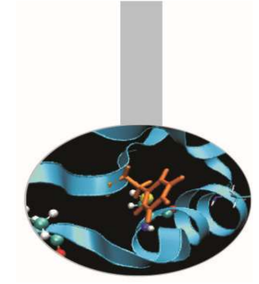


- `continue`: continue program being debugged, after signal or breakpoint
- `where` : print backtrace of all stack frames, or innermost "count" frames.
- `step` : Step program until it reaches a different source line. If used before a function call, allow to step into the function. The debugger stops at the first executable statement of that function
- `step count` : executes `count` lines of code as the `next` command

Command list



- `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.

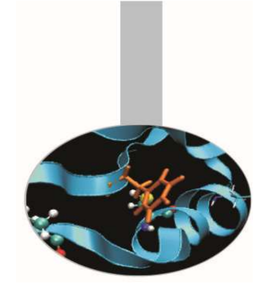


Command list

- `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[num_el]@count` displays "`count`" elements starting from position `num_el`.
    - `print *array@len` displays the whole array

SCAI
SuperComputing Applications and Innovation

Command list

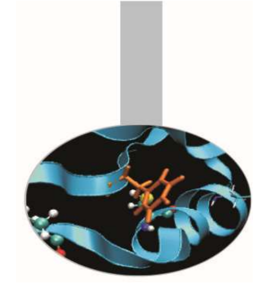


- `watch <exp>:` Set a watchpoint for an expression. A watchpoint stops execution of your program whenever the value of an expression changes.
- `info locals:` print variable declarations of current stack frame.
- `backtrace <number,full> :` shows one line per frame, for many frames, starting with the 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`

CINECA

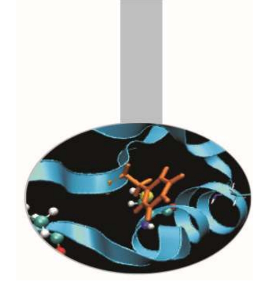
SuperComputing Applications and Innovation

Command list



- `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

CINECA



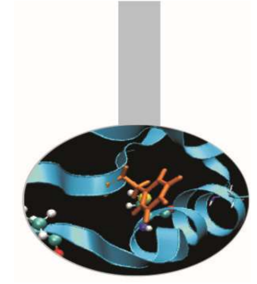
Debugging Serial Program

"pointer error" example

Program that:

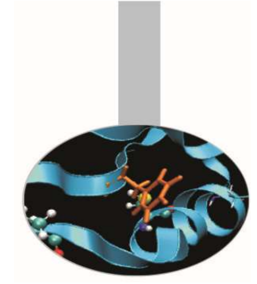
1. constructs an array of 10 integers in the variable `array1`
2. gives the array to a function `squareArray` that executes the square of each element of the array and stores the result in a second array named `array2`
3. After the function call, it's computed the difference between `array2` and `array1` and stored in array `del`. The array `del` is then written to standard output
4. Code execution ends without error messages but the elements of array `del` printed on standard output are all zeros.

Debugging Serial Program



```
#include <stdio.h>
#include <stdlib.h>
int indx;
void initArray(int nelem_in_array, int *array);
void printArray(int nelem_in_array, int *array);
int squareArray(int nelem_in_array, int *array);
int main(void) {
    const int nelem = 12;
    int *array1, *array2, *del;
    array1 = (int *)malloc(nelem*sizeof(int));
    array2 = (int *)malloc(nelem*sizeof(int));
    del = (int *)malloc(nelem*sizeof(int));
    initArray(nelem, array1);
    printf("array1 = "); printArray(nelem, array1);
    array2 = array1;
    squareArray(nelem, array2);
```

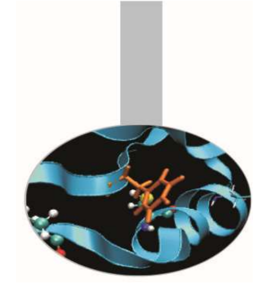
Debugging Serial Program



```
for (indx = 0; indx < nelem; indx++)
  {
    del[indx] = array2[indx] - array1[indx];
  }
  printf("La fifferenza fra array2 e array1 e':  ");
  printArray(nelem, del);
  free(array1);
  free(array2);
  free(del);
  return 0;}
void initArray(const int nelem_in_array, int *array)
{
  for (indx = 0; indx < nelem_in_array; indx++)
  {
    array[indx] = indx + 2;}
}
```

SCAI SuperComputing Applications and Innovation

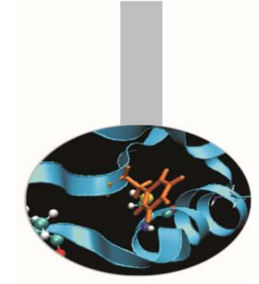
Debugging Serial Program



```
int squareArray(const int nelem_in_array, int *array)
{
  int indx;
  for (indx = 0; indx < nelem_in_array; indx++)
  {
    array[indx] *= array[indx];}
  return *array;
}
void printArray(const int nelem_in_array, int *array)
{
  printf("[  ");
  for (indx = 0; indx < nelem_in_array; indx++)
  {
    printf("%d  ", array[indx]); }
  printf("]\n\n");
}
```

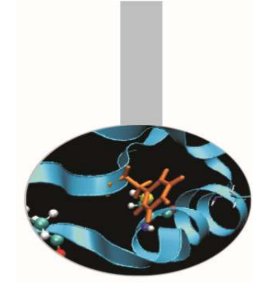
CINECA

Debugging Serial Program



- **Compiling:** `gcc -g -o arr_diff arr_diff.c`
- **Execution:** `./arr_diff`
- **Expected result:**
  - `del = [ 2 6 12 20 30 42 56 72 90 110 132 156 ]`
- **Real result**
  - `del = [ 0 0 0 0 0 0 0 0 0 0 0 0 ]`

Debugging Serial Program



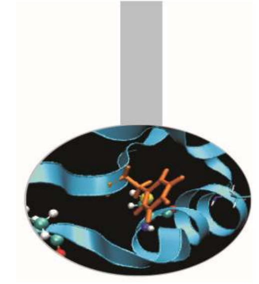
Debugging

- **Run the debugger gdb** -> `gdb ar_diff`

- **Step1**: possible coding error in function `squareArray()`
  - Procedure: list the code with the `list` command and insert a breakpoint at line 16 "`break 16`" where there is the call to `squareArray()`. Let's start the code using the command `run`. Execution stops at line 16.

    Let's check the correctness of the function `squareArray()` displaying the elements of the array `array2` using the command `disp`, For example (`disp array2[1] = 9`) produces the expected value.

Debugging Serial Program



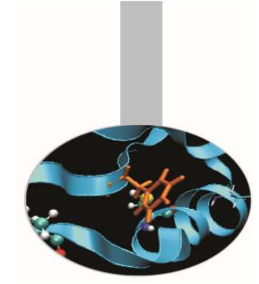
- **Step2**: check of the difference between the element values in the two arrays
  - For loop analysis:

```
 #35: for (indx = 0; indx < nelem; indx++)
(gdb) next
37          del[indx] = array2[indx] - array1[indx];
(gdb) next
35        for (indx = 0; indx < nelem; indx++)
```

  - Visualize array after two steps in the for loop:

```
(gdb) disp array2[1]
array2[1]=9
(gdb) disp array1[1]
array1[1]=9
```

Debugging Serial Program



As highlighted in the previous slide the values of the elements of `array1` and `array2` are the same. But this is not correct because array, `array1`, was never passed to the function `squareArray()`. Only array2 was passed in line 38 of our code. If we think about it a bit, this sounds very much like a "**pointer error**".

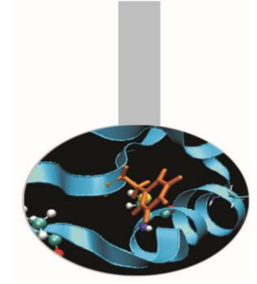
To confirm our suspicion, we compare the memory address of both arrays:

```
(gdb) disp array1
  1: array1 = (int *) 0x607460
(gdb) disp array2
  2: array2 = (int *) 0x607460
```

We find that the two addresses are identical.

SuperComputing Applications and Innovation

Debugging Serial Program



The error occurs in the statement: `array2 = array1` because the two names are pointers and the operation makes the two arrays share the same memory addresses.

Solution:

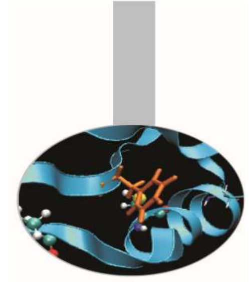
To solve the problem we just have to change the statement

```
array2 = array1;
```

in

```
for (int indx = 0; indx < nelem; indx++)
{
  array2[indx]  =  array1[indx]
}
```

Parallel debugging



Normally debuggers can be applied to **multi-threaded parallel codes**, containing OpenMP or MPI directives, or even **OpenMP and MPI** hybrid solutions.

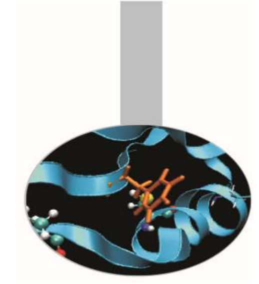
In general the threads of a single program are akin to multiple processes except that they share one address space (that is, they can all examine and modify the same variables). On the other hand, each thread has its own registers and execution stack, and perhaps private memory.

GDB provides some facilities for debugging multi-threaded programs.

Although specific commands are not provided, gdb still allows a very powerful approach for codes parallelized using MPI directives. For this reason it's widely used by programmers also for these kind of codes.

SuperComputing Applications and Innovation

Debug OpenMP Applications



- GDB facilities for debugging multi-thread programs :
  - automatic notification of new threads
  - `thread <thread_number>` command to switch among threads
  - `info threads` command to inquire about existing threads

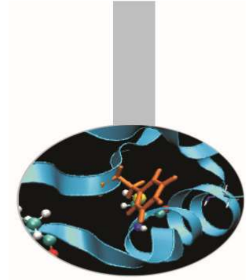
```
(gdb) info threads
*  2 Thread 0x40200940 (LWP 5454)  MAIN__.omp_fn.0
   (.omp_data_i=0x7fffffffd280) at
   serial_order_bug.f90:27
   1 Thread 0x2aaaaaf7d8b0 (LWP 1553)
   MAIN__.omp_fn.0 (.omp_data_i=0x7fffffffd280) at
   serial_order_bug.f90:27
```

- `thread apply <thread_number> <all> args`

  allows to apply a command to a list of threads.

CINECA

Debug OpenMP Applications



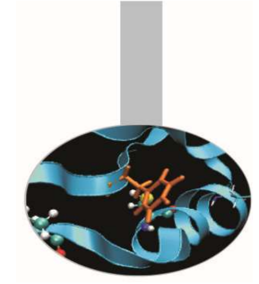
- When **any thread in your program stops**, for example, at a breakpoint, **all other threads in the program are also stopped** by GDB.
- GDB **cannot single-step all threads** in lockstep. Since thread scheduling is up to your debugging target's operating system (not controlled by GDB), **other threads 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.

SCAI
SuperComputing Applications and Innovation

Debug OpenMP Applications



- **Example of “hung process”**

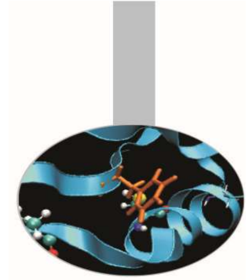
  In the following OpenMP code, using the SECTIONS directive, two threads initialize their own array and than sum it to the other.

```
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
```

CINECA

CINECA SCAI
SuperComputing Applications and Innovation

Debug OpenMP Applications



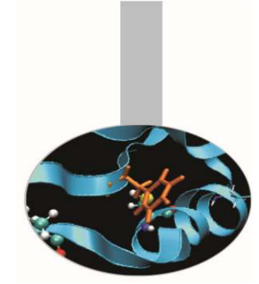
```
!$OMP SECTIONS
!$OMP SECTION
  PRINT *, 'Thread',TID,' initializing A()'
  CALL OMP_SET_LOCK(LOCKA)
      DO I = 1, N
         A(I) = I * DELTA
      ENDDO
  CALL OMP_SET_LOCK(LOCKB)
  PRINT *, 'Thread',TID,' adding A() to B()'
      DO I = 1, N
         B(I) = B(I) + A(I)
      ENDDO
   CALL OMP_UNSET_LOCK(LOCKB)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

CINECA

SCAI
SuperComputing Applications and Innovation

Debug OpenMP Applications



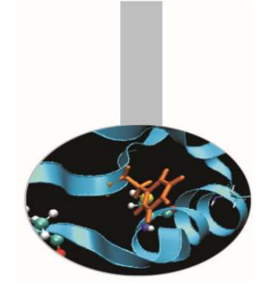
- Compiling:

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

- Esecution:
  - `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()
```

CINECA



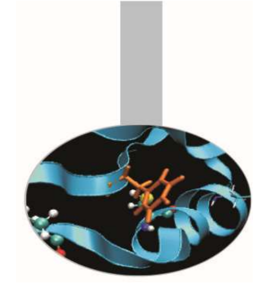
Debug OpenMP Applications

- Debugging
- List the source code from line 10 to 50 using the command:
  `list 10,50`
- Insert a breakpoint at the beginning of the parallel region:

```
(gdb) b 20
```

- run the executable with the command:

```
(gdb) run
```



Debug OpenMP Applications

- Check the threads are at the breakpoint:

```
(gdb) info threads
* 2 Thread 0x40200940 (LWP 8533)  MAIN__.omp_fn.0
   (.omp_data_i=0x7fffffffd2b0) at openmp_bug2_nofix.f90:20
  1 Thread 0x2aaaaaf7d8b0 (LWP 8530)  MAIN__.omp_fn.0
   (.omp_data_i=0x7fffffffd2b0) at openmp_bug2_nofix.f90:20
```

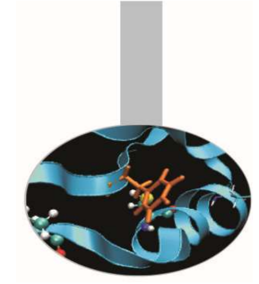
- Looking at the source it's clear that in the SECTION region the threads don't execute the statements:

```
PRINT *, 'Thread',TID,' adding A() to B()'
PRINT *, 'Thread',TID,' adding B() to A()`
```

- Insert a breakpoint in the two sections:

```
(gdb) thread apply 2 b 35
(gdb) thread apply 1 b 49
```

Debug OpenMP Applications

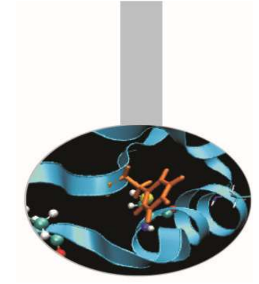


- Restart the execution:

```
   (gdb) thread apply all cont
Continuing.
 Thread           1 starting...
 Number of threads =           2
 Thread           0 starting...
 Thread           1  initializing A()
 Thread           0  initializing B()
```

- The execution hangs without reaching the breakpoints!

SCAI
SuperComputing Applications and Innovation



Debug OpenMP Applications

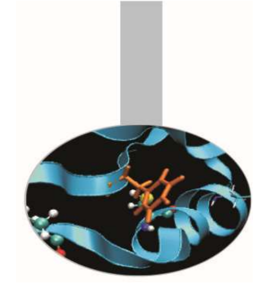
- Stop execution with "`ctrl c`" and check where threads are:

```
(gdb) thread apply all where
```

```
Thread 2 (Thread 0x40200940 (LWP 8533)):
  0x00000000004010b5 in MAIN__.omp_fn.0
  (.omp_data_i=0x7fffffffd2b0) at
  openmp_bug2_nofix.f90:29

Thread 1 (Thread 0x2aaaaaf7d8b0 (LWP 8530)):
  0x000000000400e6d in MAIN__.omp_fn.0
  (.omp_data_i=0x7fffffffd2b0) at
  openmp_bug2_nofix.f90:43
```

CINECA

Debug OpenMP Applications



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

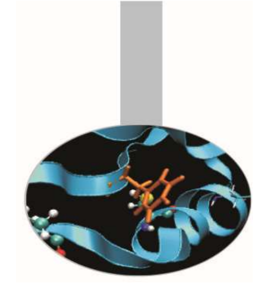
```
CALL OMP_SET_LOCK(LOCKB)
```

- 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 raise up the there is an error.
- The correct order provides that the call to `OMP_SET_LOCK` must be followed by the correspective `OMP_UNSET_LOCK`
- Arranging the order the code finishes succesfully

Debug MPI Applications

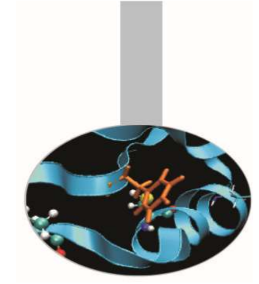


There are two common ways to use serial debuggers like GDB to debug MPI applications

- Attach to individual MPI processes after they are running using the "`attach`" method available for serial codes launching some instances of the debugger to attach to the different MPI processes.

- Open a debugging session for each MPI process through the command "`mpirun`".

Debug MPI Applications



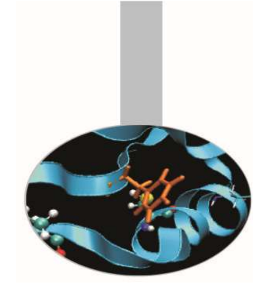
- **Attach method procedure.**

  – Run the MPI application in the standard way
    - `mpirun -np 4 executable`
    - From another shell, using the "`top`" command look at the MPI processes which are 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

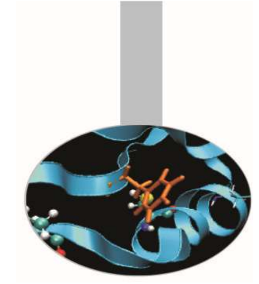
Debug MPI Applications



- **Attach method procedure**.

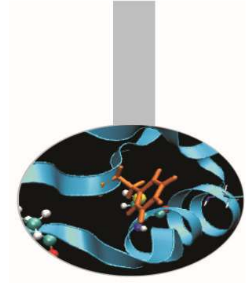
  – Run up to "n" instances of the debugger in "`attach`" mode, where "n" is the number of the MPI processes of the application. Using this method you should have to open up to "n" shells. For this reason, if not necessary, is advisable to use a little number of MPI processes.

Debug MPI Applications



- **Attach method procedure.**

  - In this example 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



Debug MPI Applications

- **Attach method procedure**.
  - The method described in the previous slides is unusable if the application crashes after few seconds.
  - An inelegant-but-functional technique commonly used with this method is to insert the following code in the application where you want to attach. This code will then spin on the sleep() function forever waiting for you to attach with a debugger.

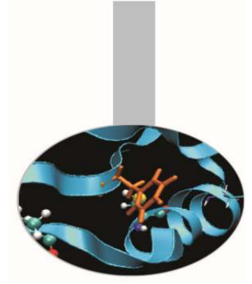
C/C++

```
{
int i = 0;
  printf("PID %d ready for
attach\n", getpid());
  fflush(stdout);
  while (0 == i) sleep(5);
}
```

Fortran

```
integer ::  i = 0
write (*,*) "PID", getpid(),"
ready  for attach"
   DO WHILE (i == 0)
    call sleep(5)
   ENDDO
```

SCAI
SuperComputing Applications and Innovation



Debug MPI Applications

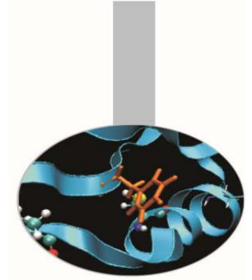
- **Attach method procedure**.

  – Recompile and re-launch the code attaching with the debugger to the process returned by the function "`getpid()`"

  – With the `next` command go to the `while` or `DO` instruction and change "i" with a value ≠ 0 : `set var i = 7`

  – Then set a breakpoint after this block of code and continue execution until the breakpoint is hit.

CINECA



Debug MPI Applications

- **Procedure with the "`mpirun`" command.**
  - 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
```

gdb

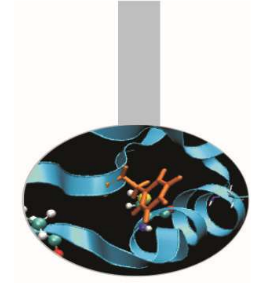
```
GNU gdb (GDB) Red Hat Enterprise Linux (7.0.1-23.el5_5.2)
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/corso/corso_debugging/Isola/Isola_MPI_2_input_gdb...d
one.
(gdb)
```

gdb

```
GNU gdb (GDB) Red Hat Enterprise Linux (7.0.1-23.el5_5.2)
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/corso/corso_debugging/Isola/Isola_MPI_2_input_gdb...d
one.
(gdb)
```

  - Now we can debug our MPI application using for each shell all the functionalities of GDB.

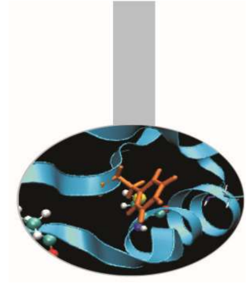
Debug MPI Applications



Debug MPI hung process

- In parallel codes using message passing, processes are typically performing independent tasks simultaneously. When the time comes to send and receive messages, certain conditions must be met in order to successfully transfer the data. One of these conditions involves blocking vs. *nonblocking* sends and receives.

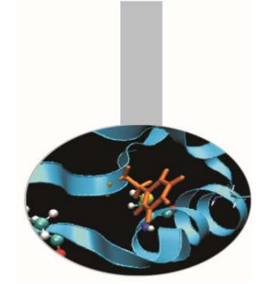
Debug MPI Applications



Debug MPI hung process

- In a blocking send, the function or subroutine does not return until the "buffer" (the message being sent) is reusable. This means that the message either has been safely stored in another buffer or has been successfully received by another process.

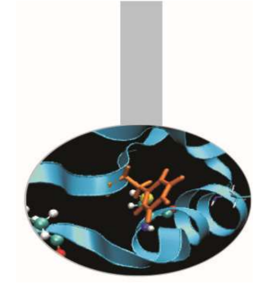
Debug MPI Applications



Debug MPI hung process

- There is generally a maximum allowable buffer size. If the message exceeds this size, it must be received by the complementary call (e.g., MPI_RECV) before the send function returns. This has the potential to cause processes to hang if the message passing is not handled carefully.

SuperComputing Applications and Innovation



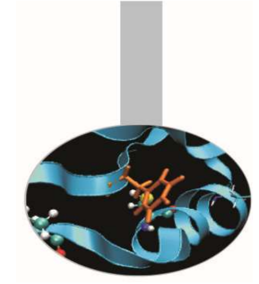
Debug MPI Applications

The following code is designed to run on exactly two processors. An array is filled with process numbers. The first half of the array is filled with the local process number, and the second half of the array is filled with the other process number. The second halves of the local arrays are filled by message passing.

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
void main(int argc, char *argv[]){
int nvals, *array, myid, i;
MPI_Status status;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myid);
nvals = atoi(argv[1]);
array = (int *) malloc(nvals*sizeof(int));
```

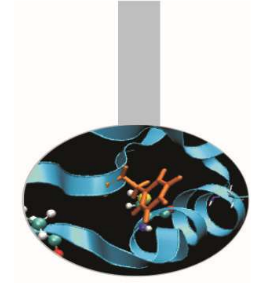
SuperComputing Applications and Innovation

Debug MPI Applications



```
for(i=0; i<nvals/2; i++);
array[i] = myid;
if(myid==0){
MPI_Send(array,nvals/2,MPI_INT,1,1,MPI_COMM_WORLD);
MPI_Recv(array+nvals/2,nvals/2,MPI_INT,1,1,MPI_COMM_WORLD,&status
   );}
else
{
MPI_Send(array,nvals/2,MPI_INT,0,1,MPI_COMM_WORLD);
   MPI_Recv(array+nvals/2,nvals/2,MPI_INT,0,1,MPI_COMM_WORLD,&stat
   us);}
printf("myid=%d:array[nvals-1]=%dn",myid,array[nvals-1]);
MPI_Finalize();
}
```

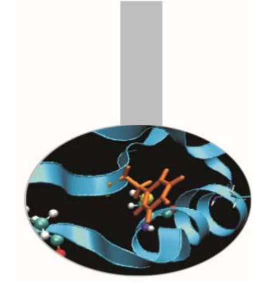
Debug MPI Applications



- Compile: `mpicc -g -o hung_comm hung_comm.c`
- Run:
  - Array dimension: 100
    - `mpirun -np 2 ./hung_comm 100`
    - myid = 0: array[nvals-1] = 1
    - myid = 1: array[nvals-1] = 0
  - Array dimension: 1000
    - `mpirun -np 2 ./hung_comm 1000`
    - myid = 0: array[nvals-1] = 1
    - myid = 1: array[nvals-1] = 0
  - Array dimension: 10000
    - `mpirun -np 2 ./hung_comm 10000`
    - With array dimension equal to 10000 the program hangs!

CINECA SCAI
SuperComputing Applications and Innovation

Debug MPI Applications



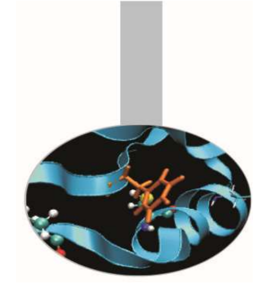
Debugging

- **Run GDB with mpirun:**
  - `mpirun -np 2 xterm -e gdb hung_proc`

- When the two separate windows, containing the "GDB" instances, are ready, visualize the source with `list` and insert a **breakpoint** at line 19 with `break 19` where there is the first MPI_Send call.
- Let's give the message dimension with `set args 1000000`

CINECA

Debug MPI Applications



Debugging

- Run the code with the command

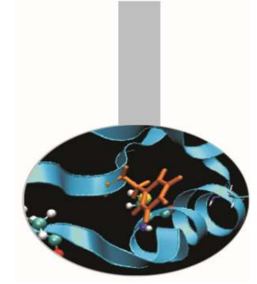
  ```
  (gdb)   run
  ```

  on the two shells, which continues until line 19 is hit.

- Step line by line on the two shells using `next`

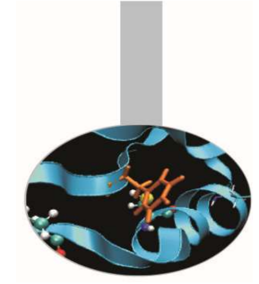
```
(gdb) next
20 MPI_Send(array,nvals/2,MPI_INT,1,1,MPI_COMM_WORLD);
(gdb) next
23 MPI_Send(array,nvals/2,MPI_INT,0,1,MPI_COMM_WORLD);
```

Debug MPI Applications



- The second `next` doesn't produce any output underlying that the execution is halted in the calls to `MPI_Send` waiting for the corresponding `MPI_Recv`.
- Let's type "`Ctrl c`" to exit from hanging. Using `where` we receive some information about where the program stopped.

SCAI
SuperComputing Applications and Innovation



Debug MPI Applications

- Among these messages there is the following one that indicates that the process is waiting for the completion of the send:
- `#4  ompi_request_wait_completion (buf=0x2aaab4801010, count=500000, datatype=0xfb8, dst=0, tag=1, sendmode=MCA_PML_BASE_SEND_STANDARD, comm=0x60c180) at ../../../../ompi/request/request.h:375`
- `#7  0x0000000000401fee in main (argc=2, argv=0x7fffffffd2a8) at hung_proc.c:23`

- **Solution**:
  - Reverse the two calls `MPI_Send` and `MPI_Recv` at lines 23 and 24.

CINECA