Debugging HPC programs, C and Fortran

CINES, Montpellier
The aim of this training is to be familiar with:
- identifying recurrent bugs
- tracing them with the appropriate tool
- solving them

Debugging is not magic, it is science. With the appropriate approach, you will solve 99% of code related bugs in no time.

Versions of compiler used:
- gcc/gfortran: 4.8.2 (from ubuntu 14.04x64)
- icc/ifort: 14.0.3 (from parallel studio 2013 SP1 update 3)
Summary

1. History
2. Unix in 10 minutes
3. Tools used
   1. Preprocessing
   2. Valgrind
   3. GDB
4. Why debugging
5. Common bugs and method to catch them:
   1. Floating point exceptions (Invalid, Overflow, Zero)
   2. Uninitialized values reading
   3. Allocation/deallocation issues
   4. Array out of bound reading/writing
   5. IO issues
   6. Memory leak
   7. Stack overflow
   8. Buffer overflow
6. Conclusion and useful links
Summary

1. History

2. Unix in 10 minutes

3. Tools used
   1. Preprocessing
   2. Valgrind
   3. GDB

4. Why debugging

5. Common bugs and method to catch them:
   1. Floating point exceptions (Invalid, Overflow, Zero)
   2. Uninitialized values reading
   3. Allocation/deallocation issues
   4. Array out of bound reading/writing
   5. IO issues
   6. Memory leak
   7. Stack overflow
   8. Buffer overflow

6. Conclusion and useful links
1. History

- 1842 : Ada Lovelace, First program of history
- 1880s : Herman Hollerith, Data on physical medium
- 1940s : Von Neumann Architecture allow programs to be stored in memory
- 1949 : Assembly language replace machine specific instructions, Text format
- 1947 : Grace Hopper, debugging
- 1949 : Grace Hopper, first compiler (A)
- 1954 : FORTRAN, first high level language
- 1971 : C language replace B
- 1983 : B. Stroustrup, C++
- 1991 : HTML
- 1995 : JAVA
1. History

Mother Tongues
Tracing the roots of computer languages through the ages

Survival of the Fittest
Reasons a language endures, with examples of some classic tongues

Debugging HPC programs - Benoît Leveugle
Summary

1. History

2. Unix in 10 minutes

3. Tools used
   1. Preprocessing
   2. Valgrind
   3. GDB

4. Why debugging

5. Common bugs and method to catch them:
   1. Floating point exceptions (Invalid, Overflow, Zero)
   2. Uninitialized values reading
   3. Allocation/deallocation issues
   4. Array out of bound reading/writing
   5. IO issues
   6. Memory leak
   7. Stack overflow
   8. Buffer overflow

6. Conclusion and useful links
2. Unix in 10 minutes

- Learn (or remember) unix basic commands.
  - Unix in 10 minutes :
    http://freeengineer.org/learnUNIXin10minutes.html
Summary

1. History
2. Unix in 10 minutes

3. Tools used
   1. Preprocessing
   2. Valgrind
   3. GDB
4. Why debugging
5. Common bugs and method to catch them:
   1. Floating point exceptions (Invalid, Overflow, Zero)
   2. Uninitialized values reading
   3. Allocation/deallocation issues
   4. Array out of bound reading/writing
   5. IO issues
   6. Memory leak
   7. Stack overflow
   8. Buffer overflow
6. Conclusion and useful links
3. Tools used

- Your allies in the battle:
  - Preprocessing
  - Valgrind
  - GDB

- More tools can be found on the web
3.1. Preprocessing

- Few words about Preprocessing
  - Tool used in many languages, here in C / Fortran
  - Allow to compile only wanted part of code
  - Useful to debug (MPI for example)

```c
#define MYVALUE
#ifndef MYVALUE
#else
#endif
```

- We will use preprocessing to simulate bugs one by one
3.1. Preprocessing

- Few words about Preprocessing
  - Tool used in many languages, here in C / Fortran
  - Allow to compile only wanted part of code
  - Useful to debug (MPI for example)

```c
#ifdef MYVALUE
#else
#endif
```

- We will use preprocessing to simulate bugs one by one
3.1. Preprocessing

- Few words about Preprocessing

```fortran
program helloorhey
  implicit none
  #ifdef HELLO
    print *, "Hello world ! «
  #endif
  #ifdef HEY
    print *, 'Hey !'
  #endif
end program helloorhey
```

```bash
$ gfortran myfile.f90
Warning: myfile.f90:5: Illegal preprocessor directive
Warning: myfile.f90:7: Illegal preprocessor directive
Warning: myfile.f90:9: Illegal preprocessor directive
Warning: myfile.f90:11: Illegal preprocessor directive
$ ./a.out
  Hello world !
  Hey !
$ gfortran -cpp myfile.f90
$ ./a.out
$ gfortran -cpp -DHELLO myfile.f90
$ ./a.out
  Hello world !
$ gfortran -cpp -DHEY myfile.f90
$ ./a.out
  Hey !
$ 
```
3.2. Valgrind

- Valgrind is a powerful memory checking tool. It is able to catch use of uninitialized values, out of bound access, stack overflow, etc. However, it will not see fpe and some other bugs.
- Valgrind has a tool to check memory, a tool to check memory leak, a tool to profile the code (use with KCacheGrind), etc.
- Valgrind is able to watch only a part of the code in order to avoid other warnings or slowdowns.
- If compiled manually, Valgrind is able to debug MPI communications.
- WARNING: Valgrind will display errors if intel environment is not loaded when debugging an intel compiled program.
### How to use valgrind? (very verbose)

```bash
$ gfortran -g -fbacktrace myfile.f90 -o myprog.exe
$ valgrind ./myprog.exe
```

```
==3306== Memcheck, a memory error detector
==3306== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==3306== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==3306== Command: ./a.out
==3306== Invalid read of size 8
==3306==    at 0x40060F: main (deb_c.c:191)
==3306==  Address 0x51fd090 is 0 bytes after a block of size 80 alloc'd
==3306==    at 0x4C2AB80: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==3306==    by 0x4005CE: main (deb_c.c:185)
==3306== 10.000000 0.000000
==3306== HEAP SUMMARY:
==3306==     in use at exit: 0 bytes in 0 blocks
==3306==     total heap usage: 1 allocs, 1 frees, 80 bytes allocated
==3306== All heap blocks were freed -- no leaks are possible
==3306== For counts of detected and suppressed errors, rerun with: -v
==3306== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

$
3.3. GDB

- GDB is the gnu debugger, available with gcc/gfortran.
- GDB is able to execute program step by step watching desired variables, break when a condition is true or at a specific line then display code for this area, etc.
- GDB is able to modify a variable on the fly.
- GDB is able to backtrace an error to provide more information on it.
- GDB overhead is lower than valgrind's overhead.
3.3. GDB

- How to use gdb? (basic commands, more at http://en.wikibooks.org/wiki/GCC_Debugging/gdb)

$ gfortran -g -fbacktrace myfile.f90 -o myprog.exe
$ gdb myprog.exe

- "run" run the program
- "break" set a "breakpoint" at a certain area \ function
- "next" execute next line of code (after a break)
- "continue" go to next breakpoint or end of program
- "print" print out a variables \ expressions contents
- "disp" print out a variable \ expression value every step
- "cond" conditional
- "set" change a value
- "quit" exit gdb
- "backtrace" get informations on program state at exit

(gdb) set variable x=12
(gdb) break test.cpp:2
Breakpoint 1 at 0x1234: file test.cpp, line 2.
(gdb) cond 1 i==2147483648
(gdb) run
Summary

1. History
2. Unix in 10 minutes
3. Tools used
   1. Preprocessing
   2. Valgrind
   3. GDB

4. Why debugging

5. Common bugs and method to catch them:
   1. Floating point exceptions (Invalid, Overflow, Zero)
   2. Uninitialized values reading
   3. Allocation/deallocation issues
   4. Array out of bound reading/writing
   5. IO issues
   6. Memory leak
   7. Stack overflow
   8. Buffer overflow

6. Conclusion and useful links
When should you think there is a bug?

- Program returns an error message
- Program returns an error exit code (other than 0)
- Program finishes with NaN or +Inf values
- Program ends unexpectedly
- Other cases, many scenario are possible
4. Why debugging

How to get the exit code of a program?

- `$$` gives you the exit code of the last executed command.
- Other than 0 means something went wrong, and this code may help you understand why.

```bash
$ gfortran myfile.f90
$ echo $$
0
$ ./a.out
Hello world!
$ echo $$
0
$

$ gfortran myfile.f90
myfile.f90:3:
mplicit none
1
Error: Unclassifiable statement at (1)
$ echo $$
1
$

$ ./a.out
Program received signal SIGSEGV:
Segmentation fault - invalid memory reference.

Backtrace for this error:
#0 0x7FFC993C87D7
#1 0x7FFC993C8DDE
#2 0x7FFC9901FC2F
Segmentation fault (core dumped)
$ echo $$
139
$
```
Summary

1. History
2. Unix in 10 minutes
3. Tools used
   1. Preprocessing
   2. Valgrind
   3. GDB
4. Why debugging

5. **Common bugs and method to catch them:**
   1. Floating point exceptions (Invalid, Overflow, Zero)
   2. Uninitialized values reading
   3. Allocation/deallocation issues
   4. Array out of bound reading/writing
   5. IO issues
   6. Memory leak
   7. Stack overflow
   8. Buffer overflow

6. Conclusion and useful links
5. Common bugs and method to catch them

- Common bugs:
  - Floating point exceptions (Invalid, Overflow, Zero)
  - Uninitialized values reading
  - Allocation/deallocation issues
  - Array out of bound reading/writing
  - IO issues
  - Memory leak
  - Stack overflow
  - Buffer overflow

- Algorithm/mathematical bugs (the worsts, especially with iterative methods). This last one will not generate an error, but results will be wrong. No specific methods, be smart.
5.1. Common bugs - Floating point exceptions

- **Floating point exceptions**

  - **Zero**
    - When you divide by zero, very common in HPC
    - \( \frac{A}{0.0} = +\infty \)

  - **Invalid**
    - When the operation is mathematically impossible
    - \( \text{acos}(10.0) = \text{NaN} \)

  - **Overflow/Underflow**
    - When you reach maximum/minimum number that system can hold
    - \( \text{exp}(10E15) = \text{A huge number} \)

- **FPEs will not generate errors at runtime!**
### 5.1. Common bugs - Floating point exceptions

#### Fortran

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gfortran</td>
<td><code>-g -fbacktrace-ffpe-trap=zero,underflow,overflow,invalid</code></td>
<td>will catch fpe at runtime</td>
</tr>
<tr>
<td>ifort</td>
<td><code>-g -traceback -fpe0</code></td>
<td>will catch fpe at runtime</td>
</tr>
</tbody>
</table>

#### C

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>Add <code>#include &lt;fenv.h&gt;</code> and start with `feenableexcept(FE_DIVBYZERO</td>
</tr>
</tbody>
</table>
| icc      | `int main(int argc, char **argv) {
feenableexcept(FE_DIVBYZERO| FE_INVALID|FE_OVERFLOW);
... }
` |

```c
#include <fenv.h>
int main(int argc, char **argv) {
feenableexcept(FE_DIVBYZERO| FE_INVALID|FE_OVERFLOW);
... }
```
5.2. Common bugs - Uninitialized values

- Uninitialized values reading
  - When you try to read a non initialized value
  - The program may not stop, and all following calculations will be based on a random value
  - Common with MPI programs (Ghost, etc)

- Static variable: variable uninitialized is static
  - no error at runtime

- Dynamic variable: variable uninitialized is dynamic
  - no error at runtime

- Not allocated variable: try to use a non allocated dynamic variable
  - error: segmentation fault
## 5.2. Common bugs - Uninitialized values

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Fortran</th>
</tr>
</thead>
</table>
| **gfortran** | When needed to use a debugging tool, do not forget `-g -fbacktrace` to get information on bug position in code  
>  static variable:  
>  `-Wuninitialized -O -g -fbacktrace` will display a warning  
>  **Valgrind**: “Conditional jump or move depends on uninitialised value(s)”  
>  dynamic variable:  
>  **Valgrind**: “Conditional jump or move depends on uninitialised value(s)”  
>  not allocated variable:  
>  `-g -fbacktrace` will catch it (size 0 or huge random number)                                                                                         |
| **ifort**   | When needed to use a debugging tool, do not forget `-g -traceback` to get information on bug position in code  
>  static variable:  
>  `-check all` (or `-check uninit`) catch it, -ftrapuv may help  
>  dynamic variable:  
>  **Valgrind**: “Conditional jump or move depends on uninitialised value(s)”  
>  not allocated variable:  
>  `-g -traceback` will catch it (size 0 or huge random number)                                                                                         |
5.2. Common bugs - Uninitialized values

<table>
<thead>
<tr>
<th>Compiler</th>
<th>C</th>
</tr>
</thead>
</table>
| gcc      | When needed to use a debugging tool, do not forget `-g` to get information on bug position in code  
  - static variable:  
    - `-Wuninitialized` or `-wall` will display a warning  
    - Valgrind: “Conditional jump or move depends on uninitialised value(s)”  
  - dynamic variable:  
    - Valgrind: “Conditional jump or move depends on uninitialised value(s)”  
  - not allocated variable:  
    - `-Wuninitialized` or `-wall` will display a warning  
    - Valgrind: “Conditional jump or move depends on uninitialised value(s)”  
| icc      | When needed to use a debugging tool, do not forget `-g -traceback` to get information on bug position in code  
  - static variable:  
    - `-Wuninitialized` will display a warning, `-g -check=uninit` will catch it at runtime  
  - dynamic variable:  
    - Valgrind: “Conditional jump or move depends on uninitialised value(s)”  
  - not allocated variable:  
    - `-Wuninitialized` will display a warning, `-g -check=uninit` will catch it at runtime |
5.3. Common bugs - Allocation

- Allocation issues
  - Try do free an non allocated variable
    - Will generate an error at runtime (not with gcc)
  - Try do allocate an already allocated variable
    - Will generate an error at runtime (not in C)
  - Not freed memory
    - No errors
### 5.3. Common bugs - Allocation

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>gfortran</td>
<td>When needed to use a debugging tool, do not forget <code>-g -fbacktrace</code> to get information on bug position in code</td>
</tr>
<tr>
<td></td>
<td>▪ free an non allocated variable:</td>
</tr>
<tr>
<td></td>
<td><code>-g -fbacktrace</code> will catch it at runtime</td>
</tr>
<tr>
<td></td>
<td>▪ allocate an already allocated variable:</td>
</tr>
<tr>
<td></td>
<td><code>-g -fbacktrace</code> will catch it at runtime</td>
</tr>
<tr>
<td></td>
<td>▪ Not freed memory:</td>
</tr>
<tr>
<td></td>
<td>Valgrind will catch it with <code>--leak-check=full</code></td>
</tr>
<tr>
<td>ifort</td>
<td>When needed to use a debugging tool, do not forget <code>-g -traceback</code> to get information on bug position in code</td>
</tr>
<tr>
<td></td>
<td>▪ free an non allocated variable:</td>
</tr>
<tr>
<td></td>
<td><code>-g -traceback</code> will catch it at runtime</td>
</tr>
<tr>
<td></td>
<td>▪ allocate an already allocated variable:</td>
</tr>
<tr>
<td></td>
<td><code>-g -traceback</code> will catch it at runtime</td>
</tr>
<tr>
<td></td>
<td>▪ Not freed memory:</td>
</tr>
<tr>
<td></td>
<td>Valgrind will catch it with <code>--leak-check=full</code></td>
</tr>
</tbody>
</table>
### 5.3. Common bugs - Allocation

<table>
<thead>
<tr>
<th>Compiler</th>
<th>C</th>
</tr>
</thead>
</table>
| gcc      | When needed to use a debugging tool, do not forget `-g` to get information on bug position in code  
  - free an non allocated variable:  
  - `-Wuninitialized` or `-wall` will display a warning  
  - Valgrind: “Conditional jump or move depends on uninitialised value(s)”  
  - allocate an already allocated variable:  
  - Valgrind will catch it with `--leak-check=full`  
  - Not freed memory:  
  - Valgrind will catch it with `--leak-check=full` |
| icc      | When needed to use a debugging tool, do not forget `-g -traceback` to get information on bug position in code  
  - free an non allocated variable:  
  - `-Wuninitialized` will display a warning, `-g -check=uninit` will catch it at runtime  
  - allocate an already allocated variable:  
  - Valgrind will catch it with `--leak-check=full`  
  - Not freed memory:  
  - Valgrind will catch it with `--leak-check=full` |
5.4. Common bugs - Array out of bounds

- Array out of bound reading/writing
  - Will not generate errors most of the time
  - Very common in HPC

- Often called "Gardening" when memory is not protected
### 5.4. Common bugs - Array out of bounds

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>gfortran</td>
<td><code>-g -fbacktrace -fbounds-check</code> will catch it at runtime</td>
</tr>
<tr>
<td>ifort</td>
<td><code>-g -traceback -check all</code> (or <code>-check bounds</code>) will catch it at runtime</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Compiler</th>
<th>C</th>
</tr>
</thead>
</table>
| gcc      | When needed to use a debugging tool, do not forget `-g` to get information on bug position in code  
**Valgrind**: “Invalid read/write of size 8”  
Or patch gcc and recompile it with bounds checking (http://sourceforge.net/projects/boundschecking/) |
| icc      | `-g -traceback -check-pointers=rw` will catch it at runtime, however `-check-pointers=rw` makes all other debugging options not working, be careful |
5.5. Common bugs - IO

- **IO issues**

- Errors are often very explicit. No need to use a debugging tool. However, Valgrind and fpe options can detect some related errors (bad reading = bad initialized value or = fpe, etc.)

- Do not forget to put -g -fbacktrace (gcc/gfortran) or -g -traceback (icc/ifort) to get useful error information.
5.6. Common bugs - Memory leak

- Memory leak

- Can be the reason of a segmentation fault (signal 11) or an unexpected code halt. Memory growth and growth until it reach limit which halts the program.

- Impossible with recent Fortran compilers if not using "pointers".

- If using Fortran pointers or C, then Valgrind will catch it!
5.6. Common bugs - Memory leak

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Fortran</th>
</tr>
</thead>
</table>
| gfortran | When needed to use a debugging tool, do not forget \texttt{-g -fbacktrace} to get information on bug position in code  
\texttt{Valgrind} will catch it with \texttt{--leak-check=full} |
| ifort    | When needed to use a debugging tool, do not forget \texttt{-g -traceback} to get information on bug position in code  
\texttt{Valgrind} will catch it with \texttt{--leak-check=full} |

<table>
<thead>
<tr>
<th>Compiler</th>
<th>C</th>
</tr>
</thead>
</table>
| gcc      | When needed to use a debugging tool, do not forget \texttt{-g} to get information on bug position in code  
\texttt{Valgrind} will catch it with \texttt{--leak-check=full} |
| icc      | When needed to use a debugging tool, do not forget \texttt{-g -traceback} to get information on bug position in code  
\texttt{Valgrind} will catch it with \texttt{--leak-check=full} |
5.7. Common bugs - Stack Overflow

- Stack Overflow
- Extremely common with bad written programs
- More common with gcc/gfortran programs (icc/ifort are often smarter with memory)
- More common with multithreaded programs, like OpenMP programs
  - Each son has a very small stack which is rapidly full
- Will result in a segmentation fault
### 5.7. Common bugs - Stack Overflow

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Fortran</th>
</tr>
</thead>
</table>
| gfortran | When needed to use a debugging tool, do not forget `-g -fbacktrace` to get information on bug position in code  
Valgrind will catch it  
gdb will catch it with backtrace but not a lot informations |
| ifort    | When needed to use a debugging tool, do not forget `-g -traceback` to get information on bug position in code  
Valgrind will catch it  
gdb will catch it with backtrace but not a lot informations |

<table>
<thead>
<tr>
<th>Compiler</th>
<th>C</th>
</tr>
</thead>
</table>
| gcc      | When needed to use a debugging tool, do not forget `-g` to get information on bug position in code  
Valgrind will catch it  
gdb will catch it with backtrace but not a lot informations |
| icc      | When needed to use a debugging tool, do not forget `-g -traceback` to get information on bug position in code  
Valgrind will catch it  
gdb will catch it with backtrace but not a lot informations |
5.8. Common bugs - Buffer Overflow

- Buffer Overflow

- Famous for security reasons

- More common in C than in Fortran
- Will generate an error, except with icc

- Can ask gcc to ignore it using: -fno-stack-protector
## 5.8. Common bugs - Buffer Overflow

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>gfortran</td>
<td>Error is self explaining</td>
</tr>
<tr>
<td>ifort</td>
<td>Error is self explaining</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Compiler</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>When needed to use a debugging tool, do not forget -g to get information on bug position in code. <em>gdb</em> will catch it with backtrace</td>
</tr>
<tr>
<td>icc</td>
<td>-g -traceback -check-pointers=rw will catch it at runtime, however -check-pointers=rw makes all other debugging options not working, be careful</td>
</tr>
</tbody>
</table>
Summary

1. History
2. Unix in 10 minutes
3. Tools used
   1. Preprocessing
   2. Valgrind
   3. GDB
4. Why debugging
5. Common bugs and method to catch them :
   1. Floating point exceptions (Invalid, Overflow, Zero)
   2. Uninitialized values reading
   3. Allocation/deallocation issues
   4. Array out of bound reading/writing
   5. IO issues
   6. Memory leak
   7. Stack overflow
   8. Buffer overflow

6. Conclusion and useful links
6. Conclusion and useful links

Veteran advices:

- NaN is not equal to itself generally (depends on platform and compiler)
- Programs may not give the same results depending of the optimizations options. Using multi threading/MPI also provides different results for each run.
- Some optimization options may alter precision
- Remember that terminal output may not refresh instantly: using "hello 1", "hello 2", etc may result in wrong location, use flush (may slow down the program)

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc/icc</td>
<td>fflush(stdout);</td>
</tr>
<tr>
<td>gfortran</td>
<td>call flush()</td>
</tr>
<tr>
<td>ifort</td>
<td>call flush(ierrerror) (segfault if no ierror)</td>
</tr>
</tbody>
</table>
6. Conclusion and useful links

Veteran advices:

- If your bug is impossible to locate (mathematical or algorithm error), ask someone else to check, most of the time bug is right in front of you but your knowledge of the code prevent you from seeing it.

- Never debug more than half a day, this could be worst (introduce more bugs to resolve one), and your brain needs to "think" away from a screen.

- Automatic testing can prevent debugging.
Thank you for your attention