HIGH PERFORMANCE COMPUTING for SCIENCE & ENGINEERING (HPCSE) I

HS 2021

Tutorial 1b: Debugging, Memory Leaks, Profiling

Ermioni Papadopoulou

Computational Science and Engineering Lab
ETH Zürich

01.10.2021

Outline - Approaches to debug your code

I. In-code debugging concepts
II. Using a debugger
III. Memory sanitation
IV. Using a profiler

I. In-code debugging concepts - Assertions

An assertion is a logical statement that will cause your code to abort execution, if false. Example:

Assertions perform checks to check that some conditions are met.

```
#include <cassert>
#include <iostream>
#include <cmath>
int assert1(){
        int a = 4;
        int b;
        std::cout << "give in an integer
number b > 4: " << std::flush;
        std::cin >> b;
        std::cout << std::endl;
        assert(a<b);
        return 0;
}
```

It is useful to combine assertions with the std functions **isinf** and **isnan**

```
int assert2(){
        double a = 2;
        double b;
        std::cout << "give in a number
b /= 0: " << std::flush;
        std::cin >> b;
        std::cout << std::endl;
        double c = a/b;
        assert(!std::isnan(c));
        assert(!std::isinf(c));
        return 0;
}
```

They can also be used with screen output.

```
int assert3(){
        assert(false && "This will always fail.\n");
        return 0;
}
```

```
./assertions
give in an integer number b > 4: 6
give in a number b /= 0: 3
Assertion failed: (false && "This will always fail.\n"), function assert3, file assertions.cpp, line 28.
```

Assertions can affect performance, as they are essentially 'if' statements.

However, it is good practice to add assertions in a code.

A simple compilation flag can ignore all assertions!

```
assertions: assertions.cpp
        g++ -g -DNDEBUG -Wall -o $@ $<
```

Flag to ignore all assertions

I. In-code debugging

Print statements to check the flow of your code

```cpp
#include <iostream>
int add(int x, int y)
{
        return x + y;
}
void printResult(int z)
{
        std::cout << "The answer is: " << z << '\n';
}
int getUserInput()
{
        std::cout << "Enter a number: ";
         int x;
         std::cin >> x;
         return --x;
}
int main()
{
        int x= getUserInput();
        int y= getUserInput();
        int z = add(x, y);
        printResult(z);
        return 0;
}
```

This will showcase that something is wrong!

Also always check with some known result!

I. In-code debugging

Always check your routines with some known result!
e.g. Matrix Multiplication:

Check with simple example, easily computed by hand

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, B = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \rightarrow A \cdot B = \begin{bmatrix} 2 & 2 & 2 \\ 2 & 2 & 2 \end{bmatrix}$$

```
// Multiplying matrix a[r1][c1] and b[r2][c2] and storing in array mult.
for(i = 0; i < r1; ++i)
  for(j = 0; j < c1; ++j)
      for(k = 0; k < c2; ++k)
        {
              mult[i][j] += a[i][k] * b[k][j];
        }
   std::cin >> x;
        return --x;
}
```

c1, c2 should be swapped!!

Testing for this simple example:

```
Enter rows and columns for first matrix: 2
2
Enter rows and columns for second matrix: 2
3

Enter elements of matrix 1:
Enter element a11 : 1
Enter element a12 : 1
Enter element a21 : 1
Enter element a22 : 1

Enter elements of matrix 2:
Enter element b11 : 1
Enter element b12 : 1
Enter element b13 : 1
Enter element b21 : 1
Enter element b22 : 1
Enter element b23 : 1

Output Matrix:
 658109658 2 0
 -795038606 2 0
```

Definitely an error in our implementation!

I. In-code debugging - compiler warnings

Compiler warnings are compilation flags that you can (should) add to your Makefiles.
Enabling warnings makes the compiler check for several issues in your source file, such as:

- Forgetting to initialize a variable
- Not using a declared variable
- Using a function before declaring it
- Having screen/file output with incorrect format (for example, treating doubles as integers)

Generally, you should add warnings and try to fix all of them, before proceeding with further debugging your code.
Note that different compilers may produce different warnings.

The most common compiler warning flags are listed below:
-Wall : enables several construction/declaration warnings
-Wextra : more warnings than Wall including warnings for unused variables
-Werror: will treat all warnings as errors
-Wpedantic: warnings related to the use of non standard C++ extensions

```
1 all: main
2
3 main: main.cpp
4         g++ -Wall -o $@ $<
5
```

II. Using a debugger

What is a debugger?

A debugger is a program.
It runs other programs, controls their execution
and examines their variables, when problems arise.

→ It allows you to run your program up to a point, stop, print variables, resume execution and so on

In this tutorial we will learn the basics of the GNU debugger (GDB)

To be able to use GDB, a program needs to be compiled by adding the flag -g:
g++ -g main.cpp -o main

This flag does not affect performance and it is good practice to always include it.

Usage: gdb ./main ← Program to debug

II. Using a debugger

Once gdb is launched (gdb ./main) we have access to several commands. Most importantly:

- **b N : puts a breakpoint at line N**
- **b +N : puts a breakpoint N lines below current line**
- **d N : deletes breakpoint number N**
- **info break : display list of all breakpoints**
- **r : run the program until next breakpoint (or error)**
- **c : continue program execution until next breakpoint (or error)**
- **f : run until current function is finished**
- **s : run next line of program**
- **s N : run N next lines of program**
- **p var : print the current value of variable "var"**
- **where : prints which function is currently running (stack of functions)**
- **q : quits the debugger**

Useful link: https://www.tutorialspoint.com/gnu_debugger/

III. Memory sanitation : memory leak detection

What is a memory leak?
A memory leak is incorrect management of memory.
It occurs when a program allocates memory without releasing it
Example:

```
for (int i = 0 ; i < N ; i++)
{
  double * temp = new double [N];
  for (int j = 0 ; j < N ; j ++)
  {
      temp[j] = 0;
      for (int k = 0 ; k < N ; k++)
      {
          temp[j] += func(i,j,k);
      }
  }
}
```

```
for (int i = 0 ; i < N ; i++)
{
  double * temp = new double [N];
  for (int j = 0 ; j < N ; j ++)
  {
      for (int k = 0 ; k < N ; k++)
      {
          temp[j] += func(i,j,k);
      }
  }
}
```

```
for (int i = 0 ; i < N ; i++)
{
  double * temp = new double [N];
  for (int j = 0 ; j < N ; j ++)
  {
      temp[j] = 0;
      for (int k = 0 ; k < N ; k++)
      {
          temp[j] += func(i,j,k);
      }
  }
  delete [] temp;
}
```

What is the initial value of temp[j]?
Sometimes it's 0 by default, often it's a random (usually very large) number

Other memory-related bugs:
- Use of uninitialized values
- Heap buffer overflow (out of bounds memory access)

All these memory-related bugs can be hard to detect.
They cause undefined behaviour
(we might get different results
- sometimes correct ones -
when we run our code)

```
double temp[10];
temp [15] = 3.14159;
```

What is stored in memory location (temp+15)?
(Pretty) bad scenario: some other variable that we're using
(Still very) bad scenario: we're not using that memory location and our code runs
Good scenario: our code will crash

Solution to all our problems:
Address Sanitizer

III. Memory sanitation : valgrind

Usage: valgrind --leak-check=full ./mycode

No special flags required during compilation. On Euler: module load new valgrind

Valgrind summary for code that leaks memory:

```
1 ==1145323== LEAK SUMMARY:
2 ==1145323==    definitely lost: 84 bytes in 2 blocks
3 ==1145323==    indirectly lost: 0 bytes in 0 blocks
4 ==1145323==      possibly lost: 0 bytes in 0 blocks
5 ==1145323==    still reachable: 0 bytes in 0 blocks
6 ==1145323==         suppressed: 0 bytes in 0 blocks
```

Also see: https://valgrind.org/

IV. Using a profiler

Why?
To identify bottlenecks of your code and optimize source files accordingly.

One way to do it: manually with std::chrono.
Offers many possibilities, can use high resolution clock to measure up to nanoseconds differences in execution time.

```cpp
#include <ctime>
#include <chrono>
#include <iostream>

int main()
{
    auto start = std::chrono::steady_clock::now();
    Some_function();
    auto end = std::chrono::steady_clock::now();
    auto elapsed = std::chrono::duration_cast<std::chrono::microseconds>(end - start);
    std::cout << "It took me " << elapsed.count() << " microseconds." << std::endl;
}
```

Honorable mention: use the timer provided by the OPENMP library

```cpp
#include <omp.h>

int main()
{
    t1 = omp_get_wtime();
    Some_function();
    t2 = omp_get_wtime();
    std::cout << "Total time = " << t2-t1 << "\n";
    return 0;
}
```

A better way to do it: automatically, with the code profiler **gprof.**

IV. Using a profiler: gprof

0. Make sure your code is running correctly
1. Compile with: g++ -pg -o main main.cpp
2. Run as usual: ./main
3. Create profile: gprof ./main > prof.out
4. Study function call graph in prof.out

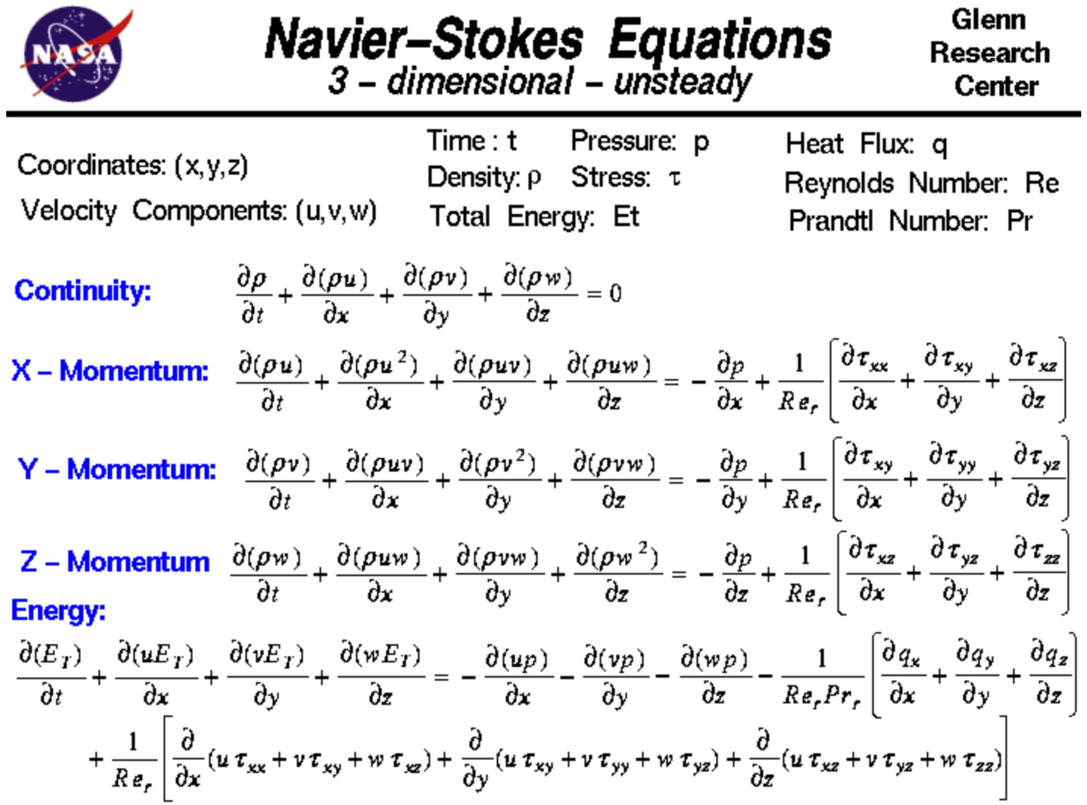
Flag -pg is needed to instrument code for use with gprof

```
1 Flat profile:
2
3 Each sample counts as 0.01 seconds.
4   %   cumulative   self              self     total
5  time   seconds   seconds    calls  ms/call  ms/call  name
6  61.72      1.69     1.69       50    33.82    33.82  func1(int)
7  39.55      2.77     1.08       50    21.67    21.67  func2(int)
8   0.00      2.77     0.00        1     0.00     0.00  _GLOBAL__sub_I__Z5func1i
9   0.00      2.77     0.00        1     0.00     0.00  __static_initialization_and_destruction_0(int, int)
```

Putting it all together....

Example: solving the Burger's equation

As an example, we are concerned with solving an extremely simplified version of the Navier-Stokes equations, the Burger's equation



Navier–Stokes Equations
3 – dimensional – unsteady

Glenn Research Center

Coordinates: (x,y,z)
Velocity Components: (u,v,w)
Time : t Pressure: p Density: ρ Stress: τ Total Energy: Et
Heat Flux: q Reynolds Number: Re Prandtl Number: Pr

Continuity:
$$\frac{\partial \rho}{\partial t}+\frac{\partial(\rho u)}{\partial x}+\frac{\partial(\rho v)}{\partial y}+\frac{\partial(\rho w)}{\partial z}=0$$

X – Momentum:
$$\frac{\partial(\rho u)}{\partial t}+\frac{\partial(\rho u^2)}{\partial x}+\frac{\partial(\rho uv)}{\partial y}+\frac{\partial(\rho uw)}{\partial z}=-\frac{\partial p}{\partial x}+\frac{1}{Re_r}\left[\frac{\partial \tau_{xx}}{\partial x}+\frac{\partial \tau_{xy}}{\partial y}+\frac{\partial \tau_{xz}}{\partial z}\right]$$

Y – Momentum:
$$\frac{\partial(\rho v)}{\partial t}+\frac{\partial(\rho uv)}{\partial x}+\frac{\partial(\rho v^2)}{\partial y}+\frac{\partial(\rho vw)}{\partial z}=-\frac{\partial p}{\partial y}+\frac{1}{Re_r}\left[\frac{\partial \tau_{xy}}{\partial x}+\frac{\partial \tau_{yy}}{\partial y}+\frac{\partial \tau_{yz}}{\partial z}\right]$$

Z – Momentum
$$\frac{\partial(\rho w)}{\partial t}+\frac{\partial(\rho uw)}{\partial x}+\frac{\partial(\rho vw)}{\partial y}+\frac{\partial(\rho w^2)}{\partial z}=-\frac{\partial p}{\partial z}+\frac{1}{Re_r}\left[\frac{\partial \tau_{xz}}{\partial x}+\frac{\partial \tau_{yz}}{\partial y}+\frac{\partial \tau_{zz}}{\partial z}\right]$$

Energy:
$$\frac{\partial(E_T)}{\partial t}+\frac{\partial(uE_T)}{\partial x}+\frac{\partial(vE_T)}{\partial y}+\frac{\partial(wE_T)}{\partial z}=-\frac{\partial(up)}{\partial x}-\frac{\partial(vp)}{\partial y}-\frac{\partial(wp)}{\partial z}-\frac{1}{Re_rPr_r}\left[\frac{\partial q_x}{\partial x}+\frac{\partial q_y}{\partial y}+\frac{\partial q_z}{\partial z}\right]$$
$$+\frac{1}{Re_r}\left[\frac{\partial}{\partial x}(u\tau_{xx}+v\tau_{xy}+w\tau_{xz})+\frac{\partial}{\partial y}(u\tau_{xy}+v\tau_{yy}+w\tau_{yz})+\frac{\partial}{\partial z}(u\tau_{xz}+v\tau_{yz}+w\tau_{zz})\right]$$

(Over) simplify

Burger's equation

$$\frac{\partial u}{\partial t}+u\frac{\partial u}{\partial x}=0$$

$$u(0,x)=\sin(x)\ ,\ u(t,0)=u(t,1)\ ,\ x\in[0,1]$$

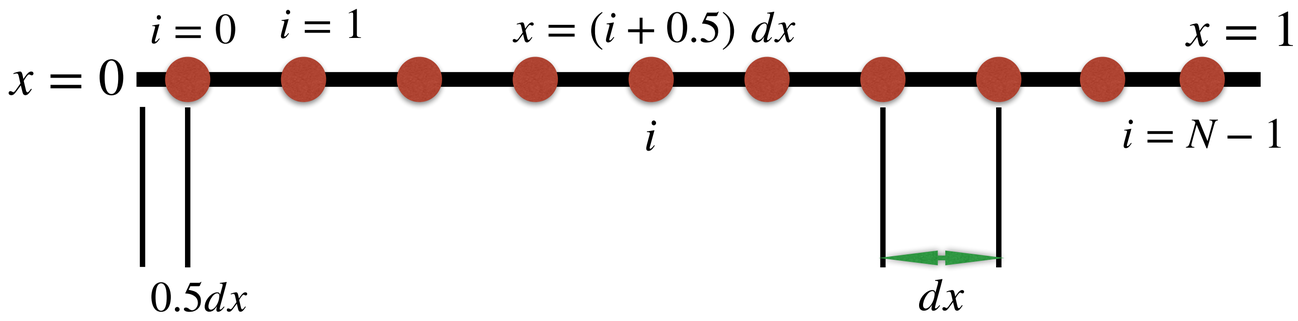
You are given a code that solves Burger's equation by using a simple numerical scheme. Or at least tries to.

We will look for bugs and other issues in that code, using the previously presented approaches.

Example: solving the Burger's equation

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = 0$$

$$\frac{u_i^{new} - u_i}{dt} = -u_i\frac{\partial u_i}{\partial x}$$



$u(0,x) = \sin(x)$, $u(t,0) = u(t,1)$, $x \in [0,1]$

Algorithm

- Pick a number of grid points N and define $dx = 1/N$
- Set initial condition $u(0,(i + 0.5)\ dx) = u_i = \sin((i + 0.5)\ dx)$
- Set time $t = 0$
- Iterate, while $t < t_{max}$:
  - Compute current timestep dt
  - Compute and store $u_i\ \partial u_i/\partial x \quad , \quad \forall i = 0,\ldots,N$
  - Advance solution in time $u_i \leftarrow u_i - dt\ u_i\ \partial u_i/\partial x$
  - $t \leftarrow t + dt$

For numerical stability, it can be proven that

$$\frac{\partial u_i}{\partial x} = \frac{u_i - u_{i-1}}{dx} \text{ , if } u_i \geq 0$$

$$\frac{\partial u_i}{\partial x} = \frac{u_{i+1} - u_i}{dx} \text{ , if } u_i < 0$$

$$dt = \frac{dx}{2\max_i(|u_i|)}$$

Example: solving the Burger's equation

- Login to Euler
- Go to the lecture directory and do "git pull"
- The code is in lecture/exercises/ex01/tutorial_debugging

- Load the following modules:
  - source env2lmod.sh
  - module purge
  - module load gcc/6.3.0
  - module load python/3.7.4
  - module load valgrind

https://www.geeksforgeeks.org/core-dump-segmentation-fault-c-cpp/