Technical University of Munich - Chair of Scientific Computing
in Computer Science (SCCS)

CFD Lab Report: SS20

Development of testing framework for Navier Stokes Equation Solver to simulate fluid and energy flow

Group B:

Hisham Saeed
Amr Elsharkawy
Muhammad Arslan Ansari

July 14, 2020

Contents

Chapter 1

Motivation

The main motivation which lies behind us choosing this project is the struggle that we faced during the phase for worksheet 2. Due to minor issue in setting the boundary values in our code the simulation of all the scenarios in our solution was giving incorrect results except for the Lid driven cavity. This bug lead us to waste five to six nights in a row but we couldn't track this bug at that time. We even did three versions of our code, one with using the pointers to keep track of the neighbors, second with the boolean flags and third with bitfield-based approach but couldn't solve the problem. Around that time, one of us suggested software testing as a topic for project. Since in the introductory meeting as we were instructed that, one of the goals of this lab course is to give us hands-on experience for software development that's why we all 3 group members voted for this topic after some online exploration and discussion with lab course tutors.

Chapter 2

Introduction

Project Repository
worksheet1 Testing, tag name : ws1_test https://gitlab.lrz.de/ge73cat/cfdlabcodeskeleton/-/tree/ws1_test
worksheet2 Testing, tag name : ws2_test https://gitlab.lrz.de/ge73cat/cfdlabcodeskeleton/-/tree/ws2_test
worksheet3 Testing, tag name : ws3_test https://gitlab.lrz.de/ge73cat/cfdlabcodeskeleton/-/tree/ws3_test

First of all, unit tests for **worksheet 1** were written for every single unit and minor refactoring was needed in order to pass all the tests. In the second step, integration tests were written for it to make sure that all the units are working fine and not breaking when used in collaboration. System tests were performed by testing VTK files of the output.

For **workseet 2**; as problems in this were the motivating factor for us to choose this project topic, test driven development (TDD) approach was adopted in order to refactor the code and find out the problems which we faced during the worksheet phase. Same approach was adopted here with respect to testing types, i.e. unit tests, integration tests and system tests.

For **worksheet 3** since we had a tested code base from worksheet 1, all that was required was testing of the communication between velocities and pressure among different processors. Unit tests were performed for testing communication and system tests for testing the final output.

Compilation and execution:
`mkdir build`
`cd build`
`cmake ..`
`make`
`cd tests` and run the executable

To Run the Solver:
type command ./sim ¡problem-flag¿ ¡optional-flag¿
For worksheet 1 ./sim -c
For worksheet 2 choose problem
Lid driven cavity ./sim -c
plane shear flow ./sim -p
karman vortex ./sim -k
flow over a step./sim -f
natural convection ./sim -n
fluid trap ./sim -ft

To Run the Tests:
cd build/tests For worksheet 1 ./unit_testing
For worksheet 2 ./unit_tests2
For worksheet 3 mpirun -np 2 unit_tests_ws3
you can choose a tag for a certain test to run it
e.g. ./unit_testing [read_parameters]
Definitions of testing types developed:-

UNIT TESTING is a level of software testing where individual units/ components of a software are tested. The purpose is to validate that each unit of the software performs as designed. A unit is the smallest testable part of any software. It usually has one or a few inputs and usually a single output.

INTEGRATION TESTING is a level of software testing where individual units are combined and tested as a group. The purpose of this level of testing is to expose faults in the interaction between integrated units. Test drivers and test stubs are used to assist in Integration Testing.

SYSTEM TESTING is a black box testing technique performed to evaluate the complete system the system's compliance against specified requirements. In System testing, the functionalities of the system are tested from an end-to-end perspective

Chapter 3

Worksheet 1 Testing

For testing purposes we used catch2 testing library

Testing here starts with **unit testing** of each and every single unit. Starting from the testing of `read_grid_parameters()`, moving on to testing each and every single unit of `uvp.hpp, boundary_val.hpp and sor.hpp` as shown in the screenshot below.

```
 21 > TEST_CASE("Test grid parameters", "[read_grid_parameters]"){…
 69
 70 > TEST_CASE( "Test read parameters", "[read_parameters] [!mayfail]" ) {…
128
129 > TEST_CASE("Test boundary values", "[boundaryvalues] [!mayfail]")…
186
187 > TEST_CASE("Test calculate dt ", "[calculate_dt]")…
337
338 > TEST_CASE( "Test F and G matrix", "[calculate_FG]" )…
497
498 > TEST_CASE( "Test rhs of PPE matrix", "[calculate_rs]" )…
608
609 > TEST_CASE( "Test SOR", "[sor_iteration]" )…
745
746 > TEST_CASE("Test velocity (U and v) values", "[calculate_uv]")…
```

Figure 3.1: worksheet1 unit tests

Testing here is done using the data of consecutive two time steps. Data from time step 1 is used as an input to test cases and data from second time step is used as reference for testing. After unit testing, **integration testing** was performed combining various unit together in order to make sure nothing breaks when used in amalgamation. Integration test cases are shown in the screenshot below.

```
 893 > TEST_CASE( "Test F & G integration with rhs", "[integrate F&G and RHS]" )…
1038
1039
1040 > TEST_CASE( "Test F & G integration with SOR", "[integrate F&G and SOR]" )…
1190
1191 > TEST_CASE( "Test F & G integration with U&V", "[integrate F&G and calculate_U&V]" )…
1360
1361 > TEST_CASE( "Test whole solver", "[single_iteration_test]" )…
```

Figure 3.2: worksheet1 integration tests

Matrix similarity test was performed by hash comparison of two files after saving the tested output and reference output in text files using MD5 hash generating algorithm. There

was a negligiable problem in this approach. Although the data which was being generated was the same but due to having different digits in the matrix enteries at sixth or seventh decimal place hash was completely different of two files. This was tackled with 2 approaches:
One approach which we followed was to save matrices (results from test cases) with less precision like upto 5 decimal place and this way hashing was working fine.
Second approach we adopted additionally was to analyze the matrices by their norm comparison in case if the hash check fails. Forbenius norm was used for this purpose which in our setting can be interpreted as 2-norm too.

SYSTEM TESTS will be implemented in worksheet 2 to compare between existing code of worksheet 1 and extended code in worksheet 2, it is a simple test, just comparing the vtk files using linux tool diff, but there is no use to compare vtk files results both generated from same code of worksheet 1.

Chapter 4

Worksheet 2 Testing

For testing purposes we used catch2 testing library

Similar to testing worksheet 1 we had reference data of consecutive time steps but from another solver giving correct results and log files for the matrices of velocities,pressure,etc.. were collected and fed as input to the test and the output result of the test was compared to the log file of the second step as a reference.

First we started to develop our code to support geometry files following the test driven development (TDD) approach, we extended the existing tests for checking the flags of the grid the input files reading functions like read paramters ,pgm files and regex read to check for reading negative values and number of digits to read after decimal point for the read regex function.

```
/*------------------------------read functions unit Tests--------------------------------------*/
#pragma region
TEST_CASE("Test grid parameters", "[read_grid_parameters] [!mayfail]"){…

TEST_CASE( "Test read parameters", "[read_parameters] [!mayfail]" ) {…

TEST_CASE( "Test read pgm file", "[read_pgm]")…

#pragma endregion
```

Figure 4.1: worksheet2 read functions unit testing

Then the development of the uvp and sor files methods was done, and the code was extended to fit the new flag structure developed to support geometry and the same old tests for worksheet 1 were made to test the calculation of F, G, P, U and V matrices. The tests pass except for the sor, U and V calculation due to precision change because the tests depends on the hash comparision not tolerance. At the begining of the code extension these tests were not failing, the tests started to fail after considering arbitary geometries. At this point both Lid driven Cavity and Plane Shear Flow were running the vtk files of the simulation exists in the Ref Results WS2 directory. Also tests were written for flag checking methods that helps boundary values function.

The next step was to extend step by step each example problem starting from Karman Vortex street to the Energy scenarios. Unit tests were written to test and validate the boundary values and uvp file functions as sor and similar approach was followed for the Flow Over A Step, Natural Convection and Fluid Trap until bugs which made the simulation fail were detetected and fixed for these 4 problems. As seen from Fig.

After unit testing was finished and bugs were fixed, the turn for integration testing came, integration tests for the Karman Vortex and Flow Over A Step were conducted until the last

```
#pragma region
TEST_CASE("Test boundary values", "[boundaryvalues] [!mayfail]")

TEST_CASE("Test calculate dt ", "[calculate_dt]")

TEST_CASE( "Test F and G matrix", "[calculate_FG]" )

TEST_CASE( "Test rhs of PPE matrix", "[calculate_rs]" )

TEST_CASE( "Test SOR", "[sor_iteration]" )

TEST_CASE("Test velocity (U and v) values", "[calculate_uv]")

#pragma endregion
```

Figure 4.2: worksheet2 code extension testing

```
// Karman Vortex Validation Tests
#pragma region

TEST_CASE("Test calculate dt Karman Vortex", "[calculate_dt_KV]")

TEST_CASE( "Test F and G matrix Karman Vortex", "[calculate_fg_KV]" )

TEST_CASE( "Test rhs of PPE matrix Karman Vortex", "[calculate_RS_KV]" )

TEST_CASE( "Test Test SOR Karman Vortex", "[sor_KV]" )

TEST_CASE( "Test velocity (U and v) values  Karman Vortex", "[calculate_uv_KV]" )

#pragma endregion

// Flow Over A Step
#pragma region
TEST_CASE("Test calculate dt Flow Over A Step", "[calculate_dt_FS]")

TEST_CASE( "Test F and G matrix Flow Over A Step", "[calculate_fg_FS]" )

TEST_CASE( "Test rhs of PPE matrix Flow Over A Step", "[calculate_RS_FS]" )

TEST_CASE( "Test Test SOR Flow Over A Step", "[sor_FS]" )

TEST_CASE( "Test velocity (U and v) values  Flow Over A Step", "[calculate_uv_FS]" )

#pragma endregion

// Natural Convection Validation Tests
#pragma region


TEST_CASE("Test calculate dt Natural Convection", "[calculate_dt_NC]")

TEST_CASE( "Test F and G matrix Natural Convection", "[calculate_fg_NC]" )

TEST_CASE( "Test temperature values Natural Convection", "[calculate_temp_NC]" )

TEST_CASE( "Test rhs of PPE matrix Natural Convection", "[calculate_RS_NC]" )

TEST_CASE( "Test Test SOR Natural Convection", "[sor_NC]" )

TEST_CASE( "Test velocity (U and v) values Natural Convection", "[calculate_uv_NC]" )

#pragma endregion
```

Figure 4.3: worksheet2 unit tests

test which mimics a full time step from the applying the boundary values untill the calculation of the velocities. For Comparing the results a different approach was followed in this part, the testing was done by looping over the matrices elements and comparing the values based on tolerance.

```
/* Real Scenario*/
#pragma region

domain.set_velocity(U, velocity_type::U);
domain.set_velocity(V, velocity_type::V);
domain.set_pressure(P_cal);

calculate_dt(Re,tau,&dt,dx,dy,imax,jmax,PR,&temp_flag,domain);
calculate_fg(Re, GX, GY, alpha, dt, dx, dy, imax, jmax, domain, F_cal, G_cal);
calculate_rs(dt, dx, dy, imax, jmax, F_cal, G_cal, RS_cal,domain);

int it = 0;
double res = 1.0;
// Solve system using SOR
while(it < itermax && res > eps){…

calculate_uv(dt, dx, dy, imax, jmax, domain, F_cal, G_cal);

#pragma endregion

#pragma endregion
```

Figure 4.4: worksheet2 integration testing

At Last, a simple system test was conducted by the use of the diff tool provided by linux, a simple bash script run the simulation of the lid driven cavity to compare the output results of the extended code with the reference results from worksheet 1 code.

After all the test cases were written, CI CD git lab tool was used to create a pipeline for running the written tests in order to prevent the code from breaking in further development. This will be explained in more details in chapter 6.

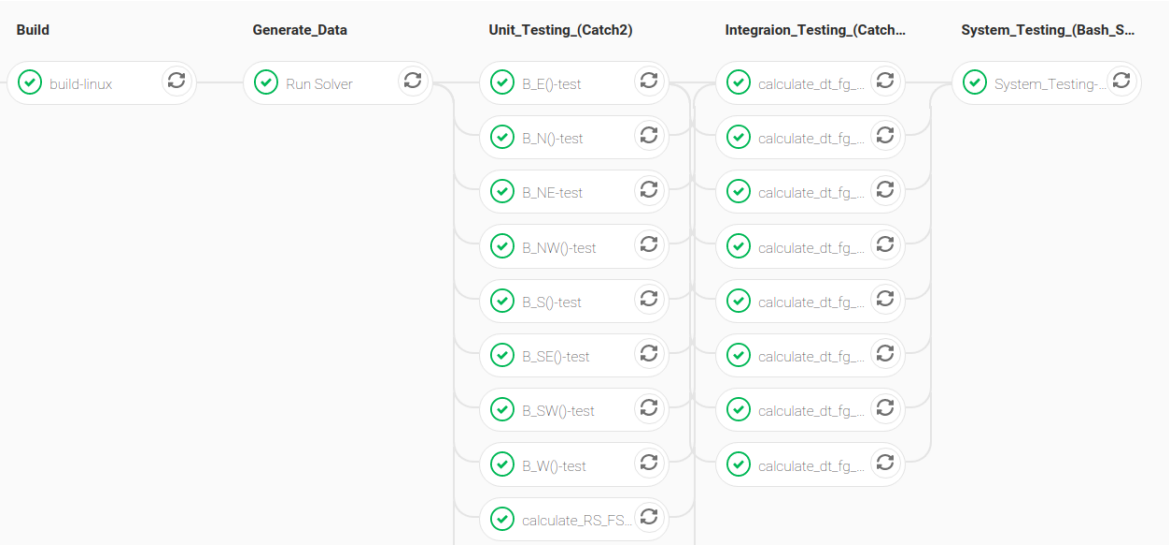


Figure 4.5: worksheet2 pipeline

Chapter 5

Worksheet 3 Testing

As this worksheet was built on worksheet 1 so only testing of communication part was needed here. Every unit of `parallel.hpp` was tested independently as shown in the screenshot below. Testing can be observed running in parallel for different processors here.

```
13  int main( int argc, char* argv[] ) {
14    clear_output_dir_test();
15
16    MPI_Init(&argc, &argv);
17    int result = Catch::Session().run( argc, argv );
18    MPI_Finalize();
19
20    return result;
21  }
22
23 > SCENARIO("Parallel Environment Testing", "[init_parallel]") {…
89
90 > SCENARIO("U&V Communication Testing", "[uv_comm]") {…
189
190
191 > SCENARIO("P Communication Testing", "[pressure_comm]")…
```

Figure 5.1: MPI communication tests

No integration tests are needed here since we did that for the worksheet 1.

Chapter 6

GitLab CI/CD Integration

Continuous Integration works by pushing small code chunks to your applications code base hosted in a Git repository, and to every push, it runs a pipeline of scripts to build, test, and validate the code changes before merging them into the main branch.

So, we have used the developed unit/integration testing to create a pipeline which is run in each push. Pipeline is configured by providing "gitlab-ci.yml" file in the GitLab repository. "gitlab-ci.yml" includes the pipeline stages, jobs and script to be executed inside each job. One benefit of the GitLab CI is that you can test your code on different operating system (OS). Using Docker engine, you can download an image of the required OS from Docker Hub. Moreover, Dockerfile is used to configure the image and install all the needed dependencies. The image is pushed into the repository registry to be used in executing the pipeline. Moreover, a runner is installed and registered to the GitLab repository, then called to execute the pipeline jobs. Based on the chosen runner's executor, pipeline is either executed locally (Shell executor) or on the system image (Docker executor).

The pipeline includes four stages: format testing, building, unit testing and integration testing.

Clang-format is tested using "run-clang-format.py" file which includes: A wrapper script around clang-format, suitable for linting multiple files and to use for continuous integration. This is an alternative API for the clang-format command line. It runs over multiple files and directories in parallel. A diff output is produced and a sensible exit code is returned.

"pre-commit" Git Hook is added to make sure in each commit the code is reformatted into clang-format, consequently passes the pipeline format test. Simply the pre-commit Git Hook includes bash script that runs automatically when code is committed. It runs the command "clang-format -i ${STYLEARG} ${1}" on all files included in the commit. "pre-commit" is a client-side Git Hook, added in the local repository; on the other side, there are server-side hooks as well that can be included in the remote repository, such as "pre-receive".

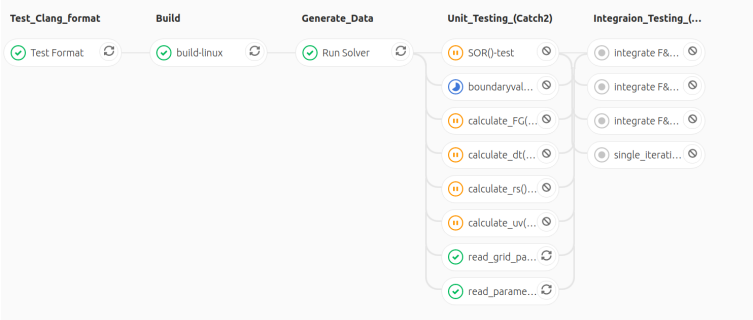


Figure 6.1: GitLab CI/CD Pipeline

Chapter 7

Testing Coverage Report

LCOV library is used to generate the ctesting coverage report. LCOV is an extension of GCOV, a GNU tool which provides information about what parts of a program are actually executed (i.e. "covered") while running a particular test case.

The extension consists of a set of Perl scripts which build on the textual GCOV output to implement the following enhanced functionality: HTML based output: coverage rates are additionally indicated using bar graphs and specific colors. Moreover, support for large projects: overview pages allow quick browsing of coverage data by providing three levels of detail: directory view, file view and source code view.

LCOV - code coverage report

			Hit	Total	Coverage
Current view:	top level - /home/amrelsharkawy/CSE/integratio_testing/cfdlabcodeskeleton	**Lines:**	385	685	56.2 %
Test:	cfd_test_coverage.info	**Functions:**	47	70	67.1 %
Date:	2020-07-13 07:24:31				

Filename	Line Coverage		Functions	
boundary_val.cpp	100.0 %	18 / 18	100.0 %	1 / 1
cell.cpp	100.0 %	17 / 17	100.0 %	8 / 8
grid.cpp	69.2 %	83 / 120	83.3 %	20 / 24
grid.hpp	100.0 %	1 / 1	100.0 %	1 / 1
helper.cpp	12.9 %	32 / 248	25.0 %	5 / 20
init.cpp	100.0 %	23 / 23	100.0 %	1 / 1
sor.cpp	100.0 %	35 / 35	100.0 %	1 / 1
utilities.cpp	50.0 %	47 / 94	60.0 %	6 / 10
utilities.hpp	100.0 %	17 / 17	-	0 / 0
uvp.cpp	100.0 %	112 / 112	100.0 %	4 / 4

Generated by: LCOV version 1.14-7-g3926d58

Figure 7.1: Testing Coverage Report

The report is saved as an artifacts in the GitLab pipeline; can either be downloaded or viewed using the internet browser.

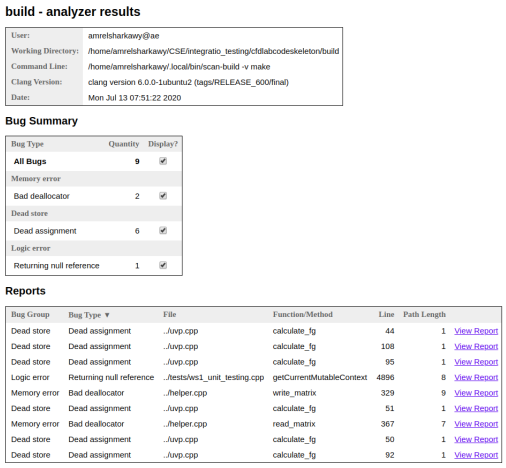
Installing LCOV (if needed): The LCOV package is available as either RPM or tarball from:
http://ltp.sourceforge.net/coverage/lcov.php
To install the tarball, unpack it to a directory and run: make install
Use Git for the most recent (but possibly unstable) version:
git clone https://github.com/linux-test-project/lcov.git
Change to the resulting lcov directory and type:
make install

Chapter 8

Clang Static Analyzer

The Clang Static Analyzer is a source code analysis tool that finds bugs in C, C++, and Objective-C programs. Currently it can be run from the command line. scan-build is a command line utility that enables a user to run the static analyzer over their codebase as part of performing a regular build (from the command line).

to run the analyzer locally:
mkdir build
cd build
cmake ..
scan-build -v -o static_analyzer_report_directory make
Then you can view the report using the web browser.



build - analyzer results

User:	amrelsharkawy@ae
Working Directory:	/home/amrelsharkawy/CSE/integratio_testing/cfdlabcodeskeleton/build
Command Line:	/home/amrelsharkawy/.local/bin/scan-build -v make
Clang Version:	clang version 6.0.0-1ubuntu2 (tags/RELEASE_600/final)
Date:	Mon Jul 13 07:51:22 2020

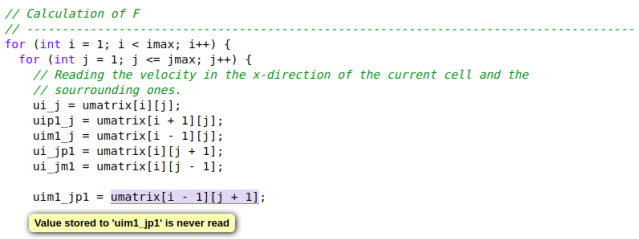
Bug Summary

Bug Type	Quantity	Display?
All Bugs	**9**	☑
Memory error		
Bad deallocator	2	☑
Dead store		
Dead assignment	6	☑
Logic error		
Returning null reference	1	☑

Reports

Bug Group	Bug Type ▼	File	Function/Method	Line	Path Length	
Dead store	Dead assignment	../uvp.cpp	calculate_fg	44	1	View Report
Dead store	Dead assignment	../uvp.cpp	calculate_fg	108	1	View Report
Dead store	Dead assignment	../uvp.cpp	calculate_fg	95	1	View Report
Logic error	Returning null reference	../tests/ws1_unit_testing.cpp	getCurrentMutableContext	4896	8	View Report
Memory error	Bad deallocator	../helper.cpp	write_matrix	329	9	View Report
Dead store	Dead assignment	../uvp.cpp	calculate_fg	51	1	View Report
Memory error	Bad deallocator	../helper.cpp	read_matrix	367	7	View Report
Dead store	Dead assignment	../uvp.cpp	calculate_fg	50	1	View Report
Dead store	Dead assignment	../uvp.cpp	calculate_fg	92	1	View Report

Figure 8.1: Static Analyzer Report



```
// Calculation of F
// ---------------------------------------------------------------------------
for (int i = 1; i < imax; i++) {
  for (int j = 1; j <= jmax; j++) {
    // Reading the velocity in the x-direction of the current cell and the
    // sourrounding ones.
    ui_j = umatrix[i][j];
    uip1_j = umatrix[i + 1][j];
    uim1_j = umatrix[i - 1][j];
    ui_jp1 = umatrix[i][j + 1];
    ui_jm1 = umatrix[i][j - 1];

    uim1_jp1 = umatrix[i - 1][j + 1];
```

Value stored to 'uim1_jp1' is never read

Figure 8.2: Static Analyzer Report