Analysis on Open Message Passing Interface

Krithikashree Lakshminarayanan (UH:2072237)
Sai Pavani Gandla (UH:2090151)
Sirichandana(UH:2148902)
GitHub Repository link.

Abstract

We see multiple applications in our day-to-day life. Behind every application, there is the contribution of so many people, among these, are the "Testers" and the "developers" who are an integral part of the success and efficiency of these applications. In this report, we are trying to analyze the state of testing in the ompi project by articulating our views about the adequacy of the tests, and the appropriateness of the testing process. We achieve this from the perspectives of Testers and Developers and their relationship to make a successful application together.

In this project we analyse and understand the open message passing interfaces architecture,estimate the time taken in the development and testing process,OMPI issues,coverage statistics etc. Our analysis is based on the open source ompi project available in GitHub.We examined source code, version control history, and regression testing history of the software.From our analysis we see maximum development happening in OMPI layer and maximum testing in OPAL layer with about maximum contributors being developers. We have also analysed the frequency of test additions and modifications.

Keywords : OMPI, debugging, assertion, contribution,coverage,statistics

CONTENTS

I. Introduction

In this project, we are analyzing Open Message Passing Interface (OMPI). Open MPI is an open-source, freely available implementation of the MPI specifications. The Open MPI software achieves high performance; the goal of our project is to

- practice building and testing ompi projects
- Analyze them to provide our assessment of testing.

In this report, we present a study that examines the effectiveness of testing in the context of a real, large-scale HPC software project. We want to understand how much of the actual software development of a larger project is genuinely being tested once the project is at a stage where it is being used to do real science.

To address this question we have built the OMPI project in our local system and generated a code coverage report, examined the locations and usages of assert and debug statements throughout the project i.e in the context of both production and development environments and use pydriller to get insights about the contributions.

By focusing on an in-depth analysis of the open-source MPI project, we hope to gain insights into the software development and testing process of computational scientists and provide estimates on their work which might serve as a starting-off point for future studies.

II. Background

In this section, we give an overview of the project and the tools, software, and methodology used in analyzing the ompi library. To build the ompi library into our personal workspace we used the Ubuntu terminal and to gain insights into the percentage of code which is covered by automated tests we used a tool called gcov.Gcov is a source code coverage analysis and statement-by-statement profiling tool. We had to configure our build to enable coverage. A detailed analysis of the coverage report is done in the further section.

For the second and the third question we have written python scripts, with the help of regular expressions we extracted the data for our analyses on the location, count of assert, log, and debug, statements in the development and testing phases.

For the final part of the project, we used Pydriller to report when the test files have been added to the project, how often they are modified, and how many people were involved. We have also used Github APIs to get more insights about the commits, issues, etc.

We have generated all the plots using the matplotlib library for the data collected in all four tasks in the form of CSV. The datasets, scripts, and plots are found in the GitHub repository whose link is provided in the initial section of the technical report.

III. Build instructionss

This section gives a generic overview of the building and installation steps followed by us to get the ompi library up and running. Note that here we do not attempt to comprehensively describe how to build Open MPI in all supported environments. Please go through the doc section in the repository to get step-by-step instructions on setting up.

Before downloading the zip we need to see the version numbers. Open MPI has 2 concurrent release series. The minor numbers tell us if it is a feature release or a super stable release. Both are well tested only difference is that stable versions are more time tested.

Built / installed very much like many other open-source packages

- ./configure –prefix=HOME/ompi … –enable-coverage
- make –j 8 install

.

Some lessons learned while configuring are

(1) Installing it under the home directory is helpful so no root permission is not required
(2) VPATH builds are convenient to segregate production files from development files
(3) Installation as the directory is convenient so it is easy to clean and rebuild
(4) It is always better to save the outputs so we can go back and examine the builds in case of any failures
(5) Always check for the latest version of Autotools because the older versions have bugs and install all the tools with the same prefix
(6) Do not overwrite configure files

.

IV. Code Breakdown

The vast majority of the project's code is written in C. In the next place we see make files generated

with Autoconf, Automake, equally contributions files includes the m4,.sh files. These files are generated when we are configuring/building the system. The other files include C++, Fortran, and Java files. These files are internally calling the C files only. The distribution of the programming languages is shown in the below figure

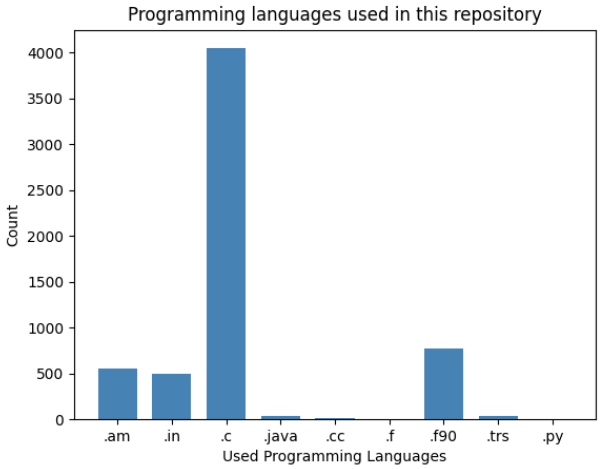

Fig. 1: Language Distribution

From the above figure, we can see that almost 80% of the files are with the .c extension. The statistics we obtained are close to the coverage report from ohloh.net

V. Layers and Architecture

OMPI has three main abstraction layers the topmost layer is the OMPI. This layer interacts with the MPI application. The next layer is ORTE which is a run-time system to launch, monitor, and kill parallel jobs. The next layer is the OPAL Which provides utility and glue code. The distribution of development in each layer is shown in figure 2,

We can see from the below figure that most of the development is happening for the OMPI layer. This does make sense because this layer is where the MPI API is written. The primary idea behind OMPI (portability) is also taken care of in this layer.Each layer is its own library. This separation helps in preventing abstraction violations. This abstraction enforcement mechanism has saved many developers from inadvertently blurring the lines between the three layers.

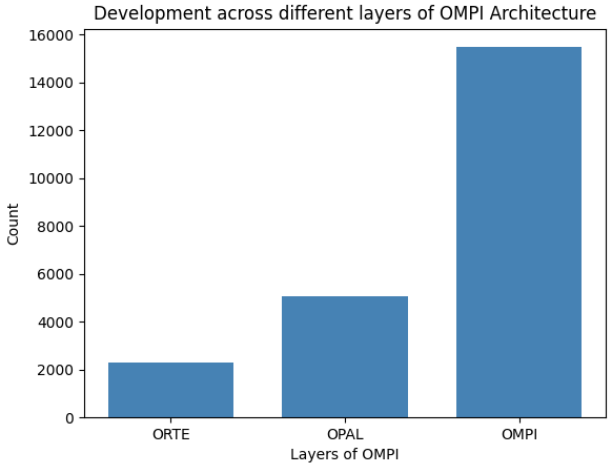

Fig. 2: Layers and Architecture

VI. Code Coverage

In this session, we determine the effectiveness of testing with the help of code coverage. Code coverage tells us the proportion of the files being tested. For the tests against bugs, test against exceptional cases that prevent crashes to be effective the coverage results must be higher.

The coverage results for our project are generated using the gcov which is a basic tool included with most Linux distributions to provide code coverage analysis and function profiling. We enabled the coverage flag while configuring the build. When we ran the automated test suite using the make check, it executed the tests. The .gcda which counts the data file and .gcno which contains information to reconstruct the basic block graphs and assign source line numbers to blocks for the tested files were generated. These files are generated from the out files obtained while compiling the source file.

One thing we should remember when analyzing the coverage report is that they do not indicate the performance efficiency. They show the test coverage on the developed file. The python library Gcovr helped us to generate reports from these gcda and gcno files. The HTML report for this is presented in the GitHub repository coverage results folder.

The table below reports the coverage statistics obtained from the gcovr results. We can see that the coverage % is really low.This is because the files that do not require testing is also considered here.This low coverage from inappropriate file is

Coverage Report			
Type	Exec	Total	Coverage
Line	11845	205072	5.8%
Function	904	13692	6.6%
Branch	5171	145864	3.5%

TABLE I: Some impressive numbers

bringing down the overall statistics.

Almost all applications have some classes we don't want to test. The usual candidates are primitive models and Data Transfer Objects (DTO).100% coverage results are impossible and it is also an unreasonable expectation. Some files may not even have any business logic, or some might always fail.

E.g.: Some file is meant to import all the dependency, test helpers, etc. In our case all the configuration files include and some of the developer-written macros do not require any testing. And thus these files have a 0% coverage. This brings down the overall results of the report. By ignoring all these files we generated a coverage report based on lines executed to total and the figure 3 below shows the line coverage report for the necessary files, The plot is generated from the CSV report of the coverage statistics. The python script for ignoring the files that do not require testing is in gen_figures.py and the report CSV is found in the dataset folder of the repository.

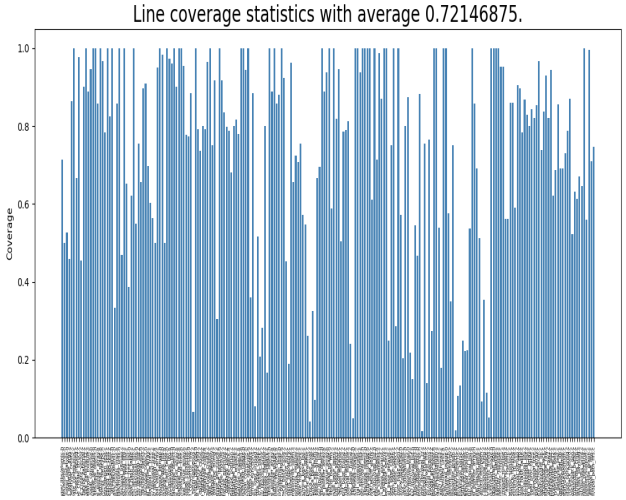

Fig. 3: Line Coverage Statistics

From the figure above we see a coverage the overall line coverage has improved to 70% which is a decent number for a large open-source project. The x-axis here is the file name and the y-axis is the coverage %. Since the figure is not clear, the plot is also uploaded to the images folder of the repository.

If we observe we find higher coverage for the OPAL layer over OMPI.ORTE layer is ignored as it is a runtime environment that cannot be tested. More tests on the OPAL layer make sense because OMPI depends on ORTE and OPAL, and ORTE depends on OPAL. So making sure that OPAL doesn't crash under any given circumstance is important for effective testing. We also think that since even though other layers can communicate with OS mostly this is done in OPAL so developers must be very careful about not crashing the OS at any given circumstance. These are the reasons for the testers to be extra conscious about this layer thus the higher code coverage is seen here.

Now as we have some insights about the amount of files being tested, we have to gain more knowledge, to determine if these tests are adequate.We cannot say this if we do not analyse all scenario's that makes the library fail.Thus in the further session we make detailed study to understand the test suites written by both developer and tester and its appropriateness.

VII. Assertions

Assert statements are used to test assumptions made by programmers. The basic purpose of assertions is to check for logically implausible circumstances. They may be used to verify the intended state of a code before it starts executing, or the anticipated state after it stops running. When considered a C program, assertions are implemented with the standard assert macro, if the developer doesn't like to use the assert () they tend to write their own macros for testing. Macro is executed only when the argument to assert is true, or else the program aborts and gives an error message.

Unit testing is a testing approach in which individual modules are tested by the developer to see if there are any flaws. It is concerned with the independent modules' functionality. The main goal is to isolate each component of the system so that defects can be identified, analyzed, and repaired.

Typically, if a file has fewer asserts, the more likely your test is to target a specific feature, and this

means that you are not testing for multiple features in the same test. For example, if we found several asserts, that means potentially testing multiple things in the same unit test.

Based on assertions we found that the library has unit and system tests.Ideally unit testing are performed by the developer, and system testing are performed from the tester side. It has some automated test cases where each module is tested again in the order specified by the tester. Now we are checking the assert count in each test file. Here we considered each and every file under the test folder as a test file and we have written a python script that checks for the assert statements in each file, and it gives the total count of assert statements in that file.

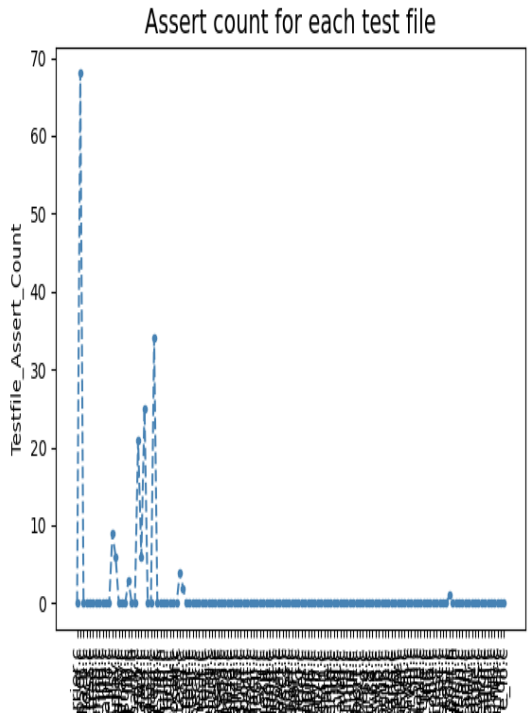

Fig. 4: Assert count for each test file

From the figure, we can see the results of the count of assert statements for each file in the test folder. Here x-axis is the filename and the y-axis is the number of assert statements in that particular file. We can see from the graph there are only few files that have assert count in more which means that only in a few files multiple features are being tested and we can also see assert count as zero in most of the files but this does not mean no feature is tested in this file.

From the figure,we can observe that atomic_cmpset.c file has got the highest number of assert statements which is 68. So from this we can say that this file has multiple number of features that are tested and it may majorly contribute to the entire test folder.

Next we are checking the files that has assert statement count greater than zero and we are plotting a graph that describes relation between the total number of test files and the test files for assert count greater than zero.

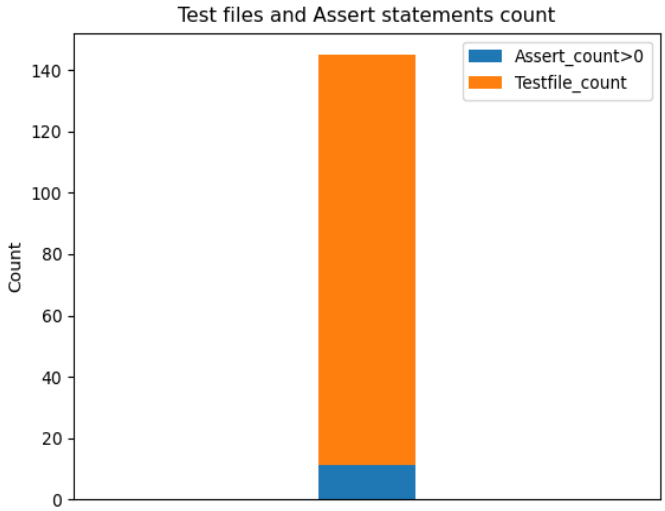

Fig. 5: Total test files and files with assert count greater than zero

From the figure we can see that blue part of the bar graph are the total test files which are having assert count greater than zero in the total number of test files.

VIII. Debugging

It is quite possible to debug issues in production. Troubleshooting an application running in production may have a negative impact on its performance.Logging is one of the simplest and most popular methods of accomplishing this.This involves including log statements in your code and looking through log files to determine the condition of your program at the time of the incident.When you have thousands, the application's overall speed suffers, especially if you aren't logging efficiently.

If we found an assert statement in the file then we can say that some feature is being tested and we can see debug statements in files where issues are raised or where new features are being added to the file.

We considered all the folders and their files except for test folder as production files. Now we are visualizing the assert count and debug count for each production file folder wise.

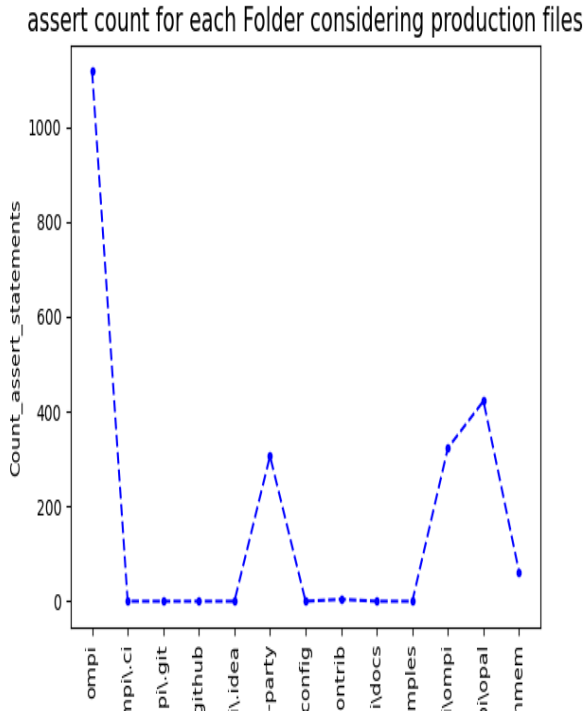

Fig. 6: Folder wise assert statements count

From the above figure 6 we can see the total count of assert statements for ompi and among the subfolders, opal folder has got the greater number of assert statements compared to others and the next is ompi folder followed by opal.This results are similar to the coverage reports generated in the above section.

More tests on the OPAL layer make sense because OMPI depends on ORTE and OPAL, and ORTE depends on OPAL. So making sure that OPAL doesn't crash under any given circumstance is important for effective testing.

From the below figure 7 we can see that the ompi main folder gives the total count of debug statements in its subfolders and among all the subfolders, the 3rd-party folder has got the major count for

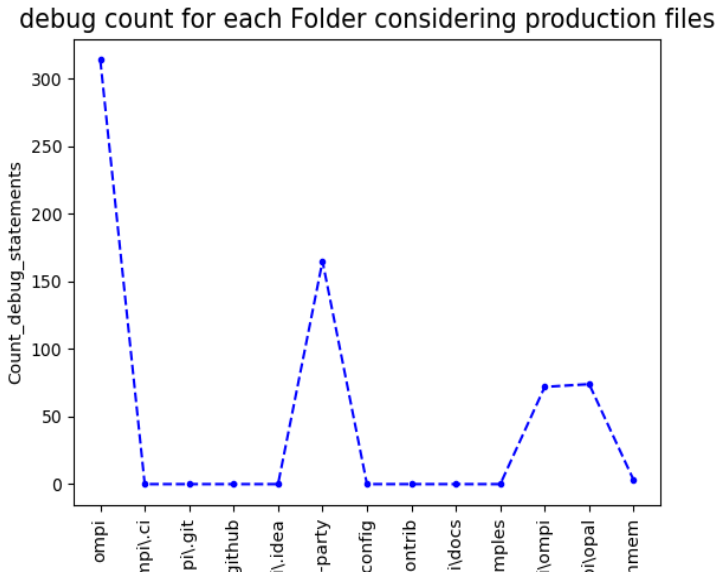

Fig. 7: Folder wise debug statements count

debug statements and next comes the opal and ompi folders.

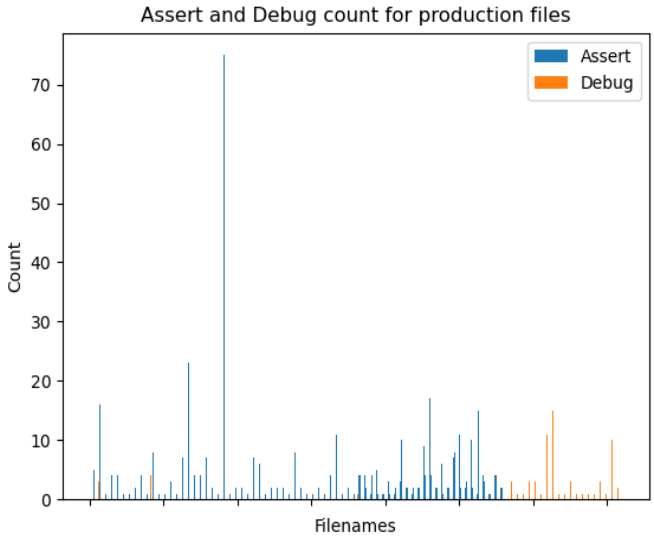

Fig. 8: File wise assert and debug statements count

In the above graph we tried to visualize file wise assert and debug count. As we have more number of observations the graph is not clear. From the dataset opal_datatype_module.c has got the highest debug statements count and coll_ftagree_earlyreturning.c has got the highest assert statements count.

IX. State of testing

For getting the insights into all the test files in the OpenMPI repository we are using Pydriller which is a Python framework that aids developers in mining software repositories for analyzing the data. This has been achieved by installing Pydriller onto our system using the pip command and then cloning the repository to a local directory before using Pydriller to investigate the repository. Pydriller contains a commit object which gives access to all of the commit metadata stored by the OpenMPI git repository.

In this session, we determine the status of test files and draw a comparison between them. The status is of three types: Added, Deleted, and Modified. Here, we are mining only the particular commits that are involved in either addition or modification, or deletion of the test files within the test folder. In the background, PyDriller opens the OpenMPI repository and extracts all the required information. Then, the Pydriller returns a generator that may iterate over the commits every time.

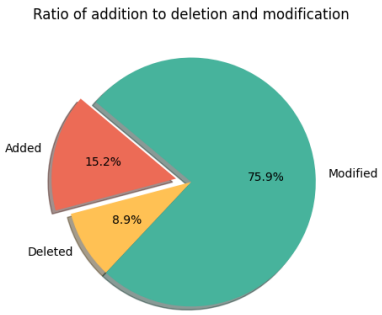

Fig. 9: Statistics for Commit type of test files

We can observe from the above figure the comparison between the three commit types.It is quite evident that though the tests have been added at lower ratio, the number of times it modified is extremely high proving that the test cases continuously evolve to identify all the unlikely scenarios. There are a good number of instances where the tests have also been deleted indicating that the tests may not be that effective in bug detection or it may be repetitive.If we closely observe the timeline of addition, it matches with the new changes in development.This is indicating that testers closely work with the developers to ensure that the newest development is effective. The python script for getting the status of every test file is in test_mining.py and also the script is modified to output the info in an exceedingly CSV format. The plot is generated from the CSV report (testreport_mining.csv) in the dataset folder of the repository.

X. Adequacy

For a developer to move forward in his/her process, needs the help of tests to find bugs therefore it is the duty of the tester to add as many tests as possible and modify them regularly to meet the requirements of the developer. Hence in this session, we talk about the addition and the modification frequency of the test files.

PyDriller framework can calculate insights of every file changed in a commit using the modified_file attribute of commit object. We can know when and what time a test file is added and how often on what dates the test file is modified by making use of the attribute committer_date which gives us the time and date of the commit.

The python script is in test_mining.py with the output in a CSV format. The plot is plotted from the CSV report (testreport_mining.csv) in the dataset folder of the repository.

The test files were added often, where the first test file was added on January 11th 2004 and the most recent one was added on February 8th 2022.

The test files have been modified frequently for more efficiency and better coverage where each modification resulted in a difference in source code and these modifications were active from January 11th, 2004 to most recent on March 16th, 2022. Jeff Squyres have contributed the most in frequently modifying the files.

In the figure below the x-axis is the name of the test file and the y-axis is the author's name. From the plotted graph we can come to the conclusion that Ralph Castain has contributed the most in terms of the addition of test files. Whereas authors such as Rob Awles, Abhishek Kulkarni, Howard Pritchard, Dave Goodell, and more have contributed the least i.e, they have contributed in addition to a single file.

We can also observe that the timeline is also considerably close to the development commits.Thus indicating coordinated work of testers and developers to ensure quality releases.

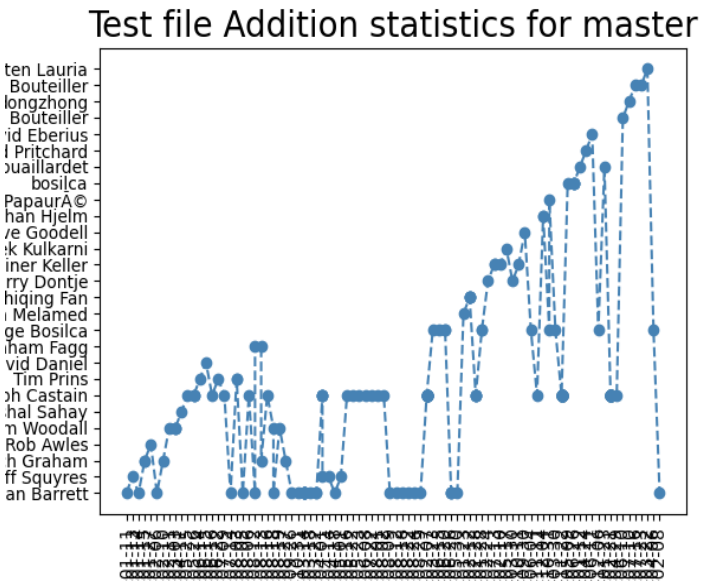

Fig. 10: Test Addition Statistics

In the figure below the x-axis is the name of the test file and the y-axis is the frequency. From the plotted graph we can come to the conclusion that every file has been modified once and the maximum frequency is more than 25, which shows that the source code of each file is modified to get a better chance of detecting bugs.

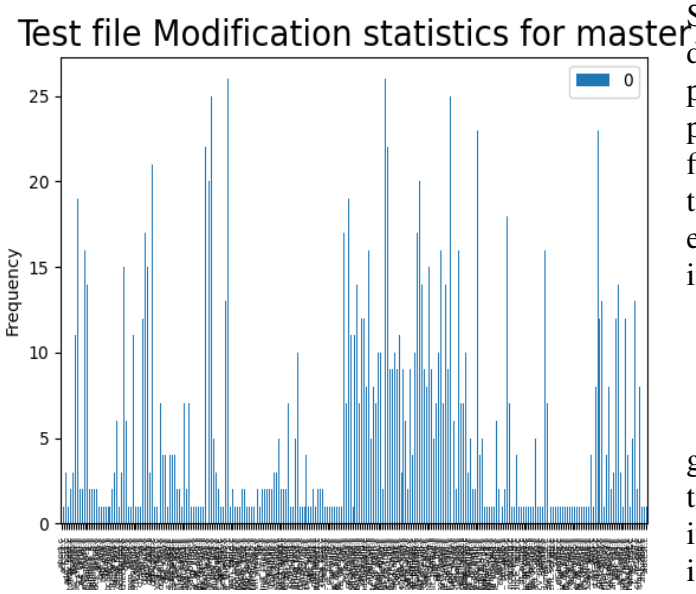

Fig. 11: Test Modification Statistics

XI. Contributors

In this session, we are finding the number of testers that contributed to the addition, modification, and deletion of test files in the specific folder called the test folder and then comparing the number with that of the total contributors who contributed to the entire repository. This is achieved by comparing the testers of specified commits involving test files to authors of commits of the entire repository.

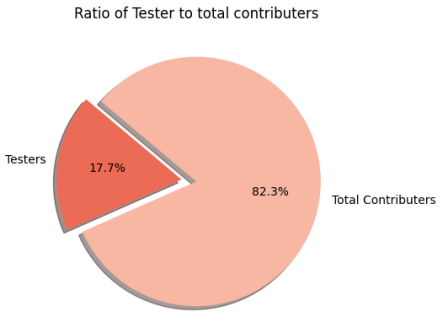

Fig. 12: Test Modification Statistics

The above figure gives the ratio of the testers only 17.7% which is very less compared to the contributor's ratio of the entire repository which is 82.3%. The highest contributor for the testers is Ralph Castain and for the entire repository is Jeff Squyres. This specifies the number of testers that design test cases along with a testing process to provide an acceptable level of confidence is less. The python script for getting the tester name of every test file is in test_mining.py and the script is modified to output the data in a CSV format. The plot is generated from the CSV report (testreport_mining.csv) in the dataset folder of the repository.

XII. Insights for Issue Reports of OMPI

In this session, we have used Github APIs to get more insights into the issues for a more detailed analysis of data. In the OpenMPI git repository, there are a total of 503 issues where each issue helps in tracking your progress with feedback and reporting bugs. But in our session, we are making use of only 52 issues with labels. A label is used to describe an issue where it can be sorted as open issues and closed issues. To

generate labels and their respective issues we installed the PyGithub module from which Github is imported. Using the built-in get_repo, get_labels, and get_issues(status=open and close) we generated the 52 labels in the form of a CSV file. The python script is in the issue_generation.py file and the CSV file(labels_issue.csv) is in the dataset folder of the repository.

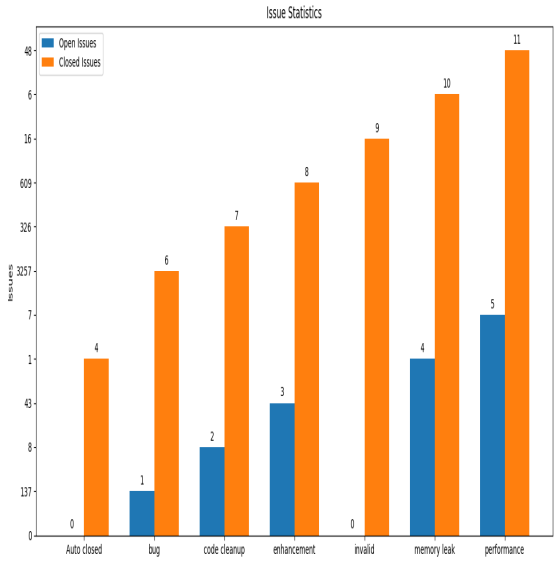

Fig. 13: Test Modification Statistics

The above figure is plotted from the labels_issue.csv file focusing on 7 labels which primarily point to development and testing related issues ,where we can observe each label has a specific count of open and closed issues with the performance label containing the maximum issues.

If we closely observe we can see the total count of issues reported is more towards development.This makes sense as the contributors for development is more and during initial phase the developer will not think of all the environment and all the capability the the new commit must satisfy.

XIII. Conclusions

In this project we have examined the large open-source ompi development .We have build the library in our local system to gain insights about the project and characterized the degree to which testing activities deal with development. We have addressed the state of testing undertaken by the contributors. We articulate our views about adequacy of the tests, and the appropriateness of testing process.

As we mentioned above, these findings are based on the the builds,coverage report generated for the ompi.We have writen python scripts to identify assert , debug count in test and production files, github analysis etc.

XIV. Appendices

1 Krithikashree Lakshminarayanan
UH : 2072237
email: klakshm2@cougarnet.uh.edu
2 Sai Pavani Gandla
UH : 2090151
email:sgandla@cougarnet.uh.edu
3 Sirichandana Balmoori
UH : 2148902
email : sbalmoor@cougarnet.uh.edu
4 Github link :
https://github.com/krithikashreeL/Analysis-on-OMPI.git

XV. References

- https://students.unimelb.edu.au/academic-skills/explore-our-resources/report-writing/technical-report-writing
- https://hpc.nmsu.edu/discovery/mpi/introduction/
- http://lorinhochstein.org/pubs/mpi-in-flash-paper.pdf
- https://www.open-mpi.org/faq/?category=supported-systems
- https://citeseerx.ist.psu.edu/viewdoc/download
- https://www.open-mpi.org/video/internals/Cisco_JeffSquyres-1up.pdf
- https://pydriller.readthedocs.io/en/latest/
- https://faculty.cs.nku.edu/waldenj/classes/2021/spring/csc666/activities
- https://pypi.org/project/requests/
- https://www.programiz.com/python-programming/writing-csv-files
- https://docs.github.com/en/rest/issues/labels#get-a-label
- https://defragdev.com/blog/2014/08/07/the-fundamentals-of-unit-testing-arrange-act-assert.html
- https://www.rookout.com/blog/production-debugging/