

OPEN MPI PROJECT REPORT

Presented by :

Prananditha Manchikatla(2092374)

Harshika Akula(2075727)

Noha Haneen Syed Imran (2079786)

Github link: <https://github.com/NohaHaneen/PAT-Open-MPI>

Abstract

The main Goal of this Project is to Construct Test Cases to Test and Analyze huge Projects and inspect the test reports generated by the test cases. We have worked on an OpenMPI Project. Open MPI is a Message Passing Interface (MPI) library which combines technology and resources from a number of different projects (FT-MPI, LA-MPI, LAM/MPI, and PACX-MPI). First, we have built the project, and generated test cases and test reports with coverage. Then we have generated a few plots to analyze the Project. The whole Process was done in the Linux Environment.

Introduction to Open MPI

As mentioned above, OpenMPI is an open-source Message Passing Interface (MPI) that is developed and maintained by a consortium of academic, research, and industry partners. MPI is a library that uses distributed memory models to perform Parallel Computing. Message-passing programs run the same code on several processors, which then communicate with one another using library calls that fall into a few categories.

- Calls to initialize, manage, and terminate communications.
- Calls to communicate between two individual processes (point-to-point).
- Calls to communicate among a group of processes (collective).
- Calls to create custom data types.

OpenMPI is used in many Super computers

Installing and Building Open MPI

We start with extracting the downloaded OPENMPI source file (tar.bz2) from the link <https://www.open-mpi.org/software/> to a convenient directory and execute the configure script found from the openmpi directory along with a prefix option and the installation directory we want to utilize for OpenMPI must be added to the prefix. we should also enable the compilers such as CC(C Compiler), CXX(C++) and FC(fortran) while invoking the configure command itself in order to compile MPI applications. After executing the configure command, a summary










































is shown which needs to be examined to ensure that it builds all of the features required. After the completion of configuration, build the package using the “make” tool. We should also make sure to include the path for path environment and library environment variables.

Understanding Coverage

We employ code coverage, a statistic for determining the number of lines of code that are successfully validated during a test method. We provide line, branch and function coverage through a coverage report using a graphical tool LCOV.

We enable the coverage flag (--enable-coverage) while invoking the configure command itself. We create a separate directory to capture the coverage data and generate HTML file from processed gcda files. This html file is a code coverage report that tells us which code has been executed and how many times.

LCOV - Code Coverage Report

opal/mca/hwloc/base		3.7 %	30 / 1307	0.3 %	4 / 133
opal/mca/shmem/mmap		12.9 %	26 / 201	35.3 %	6 / 17
omp/debuggers		14.4 %	143 / 996	10.9 %	5 / 46
opal/runtime		14.3 %	244 / 1705	17.4 %	15 / 86
opal/util		15.5 %	775 / 4987	22.7 %	71 / 313
opal/mca/shmem/posix		16.7 %	24 / 144	30.8 %	4 / 13
opal/mca/reachable/weighted		18.8 %	13 / 69	55.6 %	5 / 9
opal/threads		19.1 %	30 / 157	31.2 %	10 / 32
opal/memoryhooks		21.5 %	14 / 65	28.6 %	2 / 7
opal/mca/shmem/sysv		21.8 %	26 / 119	25.0 %	3 / 12
opal/dss		22.9 %	621 / 2713	45.0 %	77 / 171
opal/mca/base		33.9 %	985 / 2907	42.6 %	81 / 190
opal/mca/reachable/base		38.2 %	13 / 34	57.1 %	4 / 7
opal/class		44.6 %	1142 / 2558	51.0 %	133 / 261
omp/datatype		50.6 %	559 / 1104	70.4 %	38 / 54
opal/util/keyval		52.3 %	216 / 413	51.4 %	18 / 35
opal/mca/shmem/base		52.6 %	50 / 95	40.0 %	4 / 10
test/support		54.1 %	33 / 61	62.5 %	5 / 8
opal/mca/timer/linux		56.5 %	39 / 69	62.5 %	5 / 8
opal/mca/lf/posix_ipv4		61.0 %	61 / 100	100.0 %	2 / 2
test/dss		63.2 %	729 / 1153	100.0 %	41 / 41
opal/datatype		67.5 %	1533 / 2271	59.5 %	75 / 126
opal/mca/dl/base		75.0 %	24 / 32	85.7 %	6 / 7
opal/mca/event/base		78.6 %	11 / 14	100.0 %	2 / 2
test/datatype		79.1 %	1990 / 2517	88.6 %	78 / 88
opal/mca/event/libevent2022		80.0 %	44 / 55	66.7 %	4 / 6
test/asm		80.0 %	544 / 680	100.0 %	20 / 20
test/class		80.4 %	934 / 1161	93.8 %	45 / 48
opal/include/opal		85.7 %	12 / 14	100.0 %	2 / 2
opal/mca/dl/dlopen		86.4 %	102 / 118	100.0 %	9 / 9
opal/include/opal/sys		88.1 %	37 / 42	100.0 %	21 / 21
opal/include/opal/sys/gcc_builtin		88.5 %	46 / 52	90.9 %	20 / 22
opal/mca/lf/base		93.9 %	31 / 33	100.0 %	4 / 4
opal/mca/installdirs/base		94.4 %	117 / 124	80.0 %	4 / 5
test/util		94.7 %	663 / 700	100.0 %	25 / 25
opal/mca/lf/linux_ipv6		100.0 %	2 / 2	100.0 %	1 / 1
test/mpool		100.0 %	2 / 2	100.0 %	1 / 1
opal/mca/memchecker/base		100.0 %	2 / 2	100.0 %	1 / 1
opal/mca/timer/base		100.0 %	3 / 3	100.0 %	1 / 1
opal/include/opal/sys/x86_64		100.0 %	7 / 7	100.0 %	2 / 2
opal/mca/installdirs/env		100.0 %	19 / 19	100.0 %	1 / 1

The above image is the screenshot of the report generated . This report contains the directory and line and function coverage percentages for each file in the directory and as a whole. When we click on the directory, an executable code is displayed containing color codes for each line. Green represents the fully covered lines, yellow for partially covered and red for lines that were never executed.

The total coverage for the whole application is also generated and it is calculated using the number of hits divided by the number of lines/functions covered. The total line coverage is 15.6% stating that 11927 lines are covered out of 76346 lines and total function coverage is 19.4% stating that 857 functions are covered out of 4423 functions. We can interpret that only 15-20% of the code is executed during test runs. The minimum coverage should be around 70-80% to achieve better results but the obtained coverage was less than the threshold. As the number of files exercised for code coverage has zero coverage, we have come to the assumption that most of the files in the directory were header files and error handlers etc which might be also reason to consider.

As the code coverage is very less, it is certain that there might be a lot of unnoticed errors. So, to detect those errors we have used Assertions. Assertions are used to detect the subtle errors which might go unnoticed, and they detect errors soon after they occur. So, we thought using Assert Statements is the best way to move forward with analyzing the project.

Analyzing Assert Statements in Test File :

We utilize assert statements as a debugging tool since they are used to stop the program if the statements executed are false. We can see how assert statements are important in alerting us when a bug emerges in the application.

We start with searching the number of assert statements in each file and then find the total number of test files. We have written a python script which returns the output as the filename, number of assert statements in that particular file and the total number of test files. By thorough inspection on the folder, we have come to the assumption that the most of the files that have assert statements were .c and .h files.

The logic behind this is that it navigates from root directory to a particular test folder and search for the assert statement through a regex expression `(\s*(assert)\s*\([^)]*\)` in each line of each file. The total number of files is implemented by iterating and incrementing the counter for each file. We then generated output in the form of csv files.

Fig. 1 visualizes the distribution of assert statements in the files. Each marker in the scatter plot represents a file and the y axis represents the number of assert statements. The yellow markers in the plot represent files with zero assert statements. The pink markers represent files with 1 to 10 assert statements, orange represent 20 to 30, coral represent 30 to 40 and blue represent 60 to 70. As observed from the plot, most of the files appear to have zero assert statements. The next largest cluster of files, represented by the pink marker, shows that the second common number of assert statements is between 1 to 10. The highest number of assert statements is present in file 'atomic_cmpset.c'. This is denoted by the blue marker.

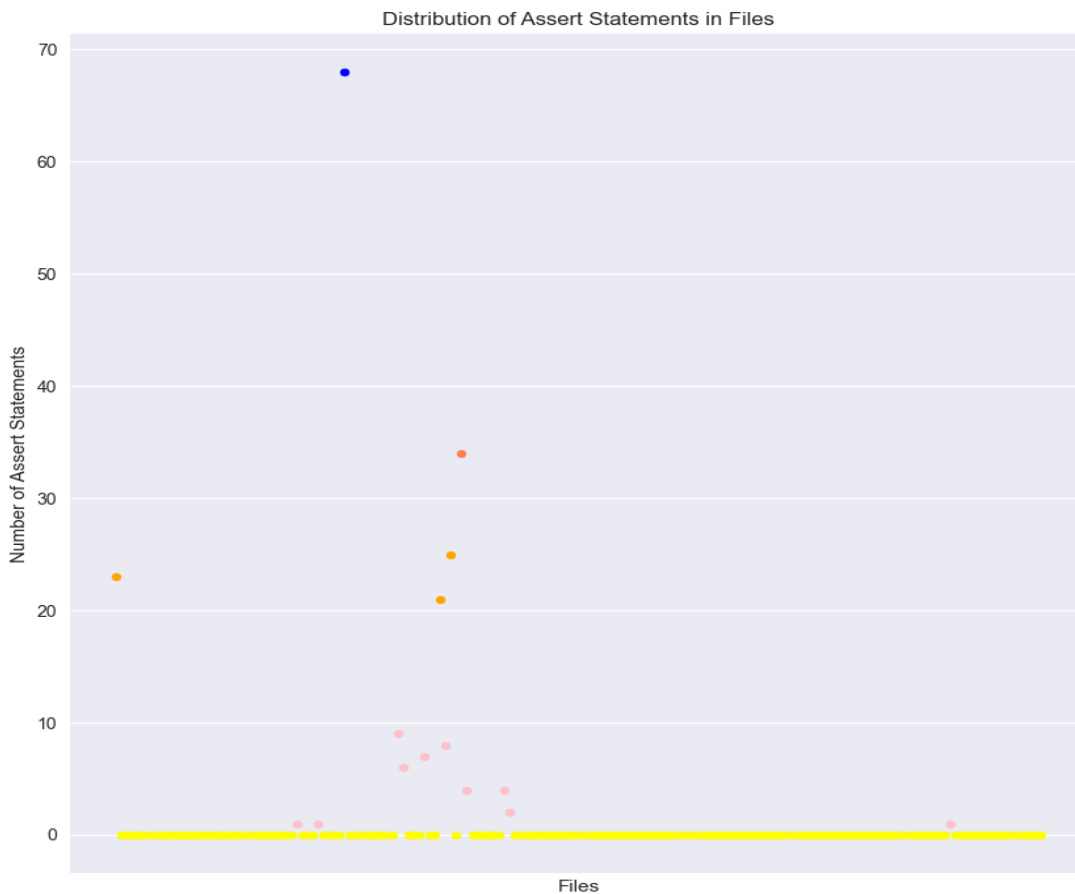


Fig. 1: Distribution of Assert Statements in Files

Analyzing Location of debug and Assert Statements in Production Files

We also search for the location along with the assert statement. It is similar to the previous one but instead of employing the logic to test files, we implement the logic on the production files. In this task, we navigate through each file in production files and then we obtain the path to that assert statement, the name of the file and the number of assert statements. The output is saved in csv file.

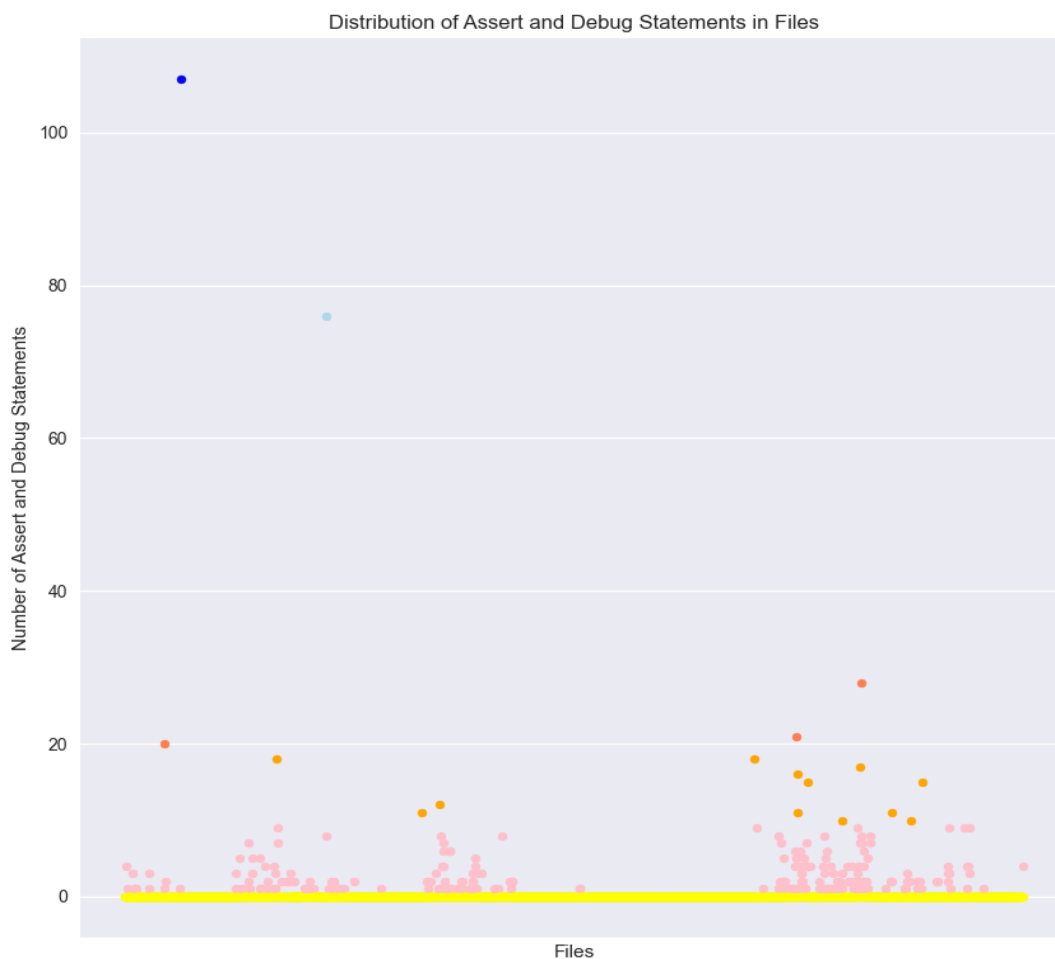


Fig. 2a: Distribution of Assert and Debug Statements in Files

Fig. 2a visualizes the distribution of assert and debug statements in the production files. Each marker in the scatter plot represents a file and the y axis represents the number of assert and debug statements. The yellow markers in the plot represent files with zero assert and debug

statements. The pink markers represent files with 1 to 10 statements, orange represent 20 to 30, coral represent 30 to 40, light blue represent 60 to 80 and blue represent above 100. As observed from the plot, most of the files appear to have zero assert and debug statements. The pink markers represent the next largest cluster of files, with 1 to 10 assert and debug statements. The files with the largest number of assert and debug statements are 'coll_ftagree_earlyreturning.c' and 'malloc.c', with 'malloc.c' leading with a total of 107 assert and debug statements.

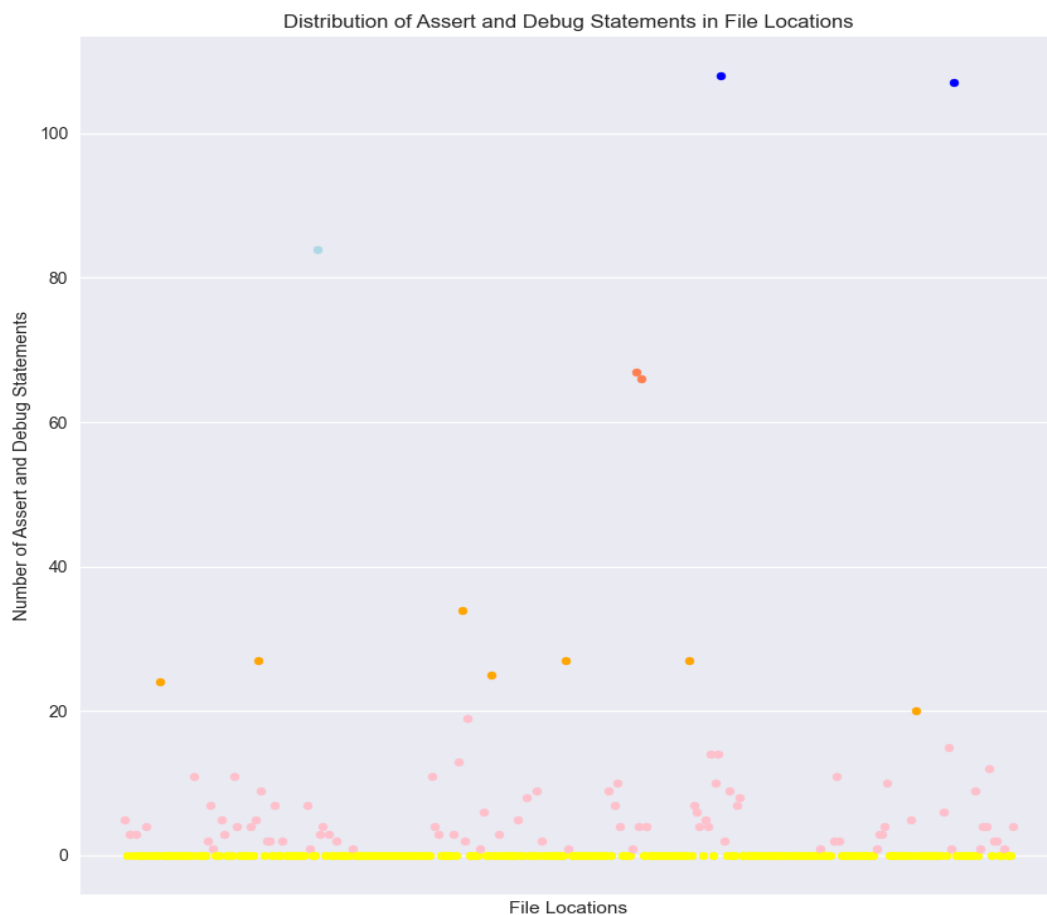


Fig. 2b: Distribution of Assert and Debug Statements in Different File Locations

The distribution of assert and debug statements in different file locations can be observed in **Fig. 2b**. Each marker in the scatter plot represents a file location and the y axis represents the number of assert and debug statements. The yellow markers in the plot represent locations with zero assert and debug statements. The pink markers represent locations with 1 to 20 statements, orange represent 20 to 40, coral represent 60 to 80, light blue represent 80 to 100 and blue

represent above 100. As observed from the plot, most of the file locations appear to have zero assert and debug statements. The locations populated the most with assert and debug statements are represented by the blue markers.

Analyzing Test File Modifications

Our fourth task in this project was to analyze the test files that have been added to the Open MPI repository and obtain an understanding of when the files were added, how often they were modified and a list of the authors involved. To analyze the commits in the test folder, we used the framework, PyDriller.

PyDriller is a Python framework used by developers to analyze Git Repositories. PyDriller allows developers to obtain information about commits, developers involved, modified files, diffs and source code. Since it is highly configurable, it gives the user a wide range of possibilities in terms of mining repositories.

To begin with, we traverse through each file in the test folder of the repository in order to obtain the date each test file was created. To access the commits, we use the Repository class of Pydriller. Since we are only interested in the files inside the test folder, Repository is customized by adding the path to the test folder. The function ‘`traverse_commits()`’ returns the selected commits with the test folder. We traverse through each modified file in each commit and store the file name with the respective date it was created.

The next step is to extract how often each file was modified in the test folder. To do this, we used the Repository class again, to traverse through each commit. We passed the test folder path as an additional parameter, Since it would take a while to generate the frequency of modification of each file in the test folder from when the project was created, we added an additional parameter, ‘`since = datetime()`’, which allowed us to specify a particular date. In this manner, we traversed through every modified file in every commit from the 1st of May, 2020. The file name was stored and the number of modifications for each file was incremented appropriately.

Finally, to obtain a list of the authors involved, we traverse through each commit of the test folder and access the author name using ‘`commit.author.name`’. We also stored the number of modifications made by each author to better understand their contribution over the past two years.

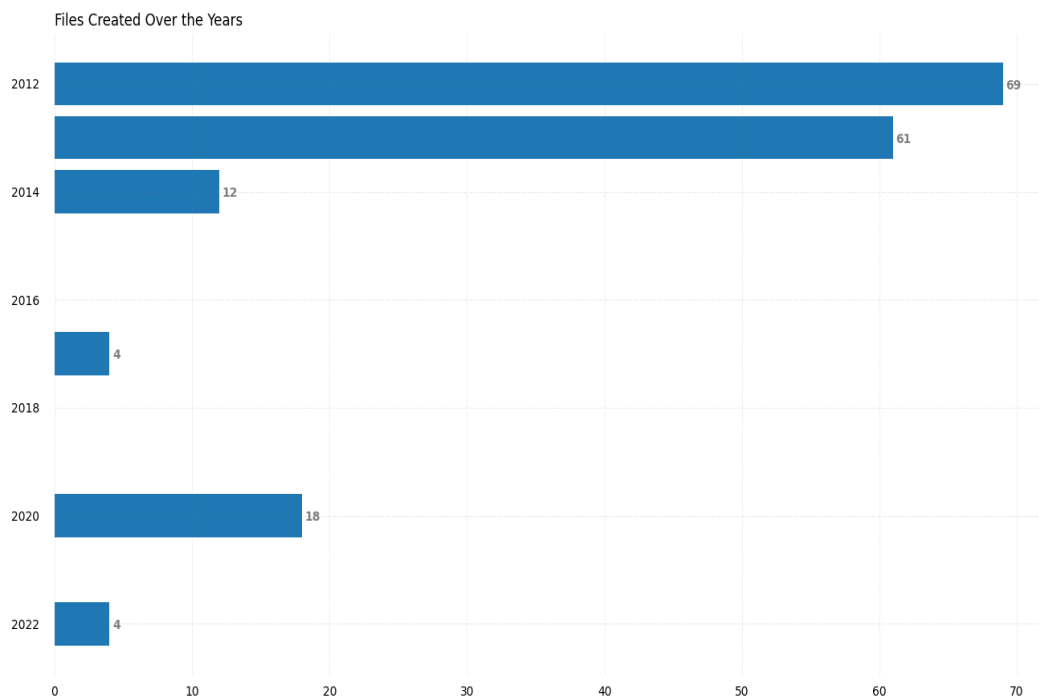


Fig. 3: Visualization of number of files created over the past ten years

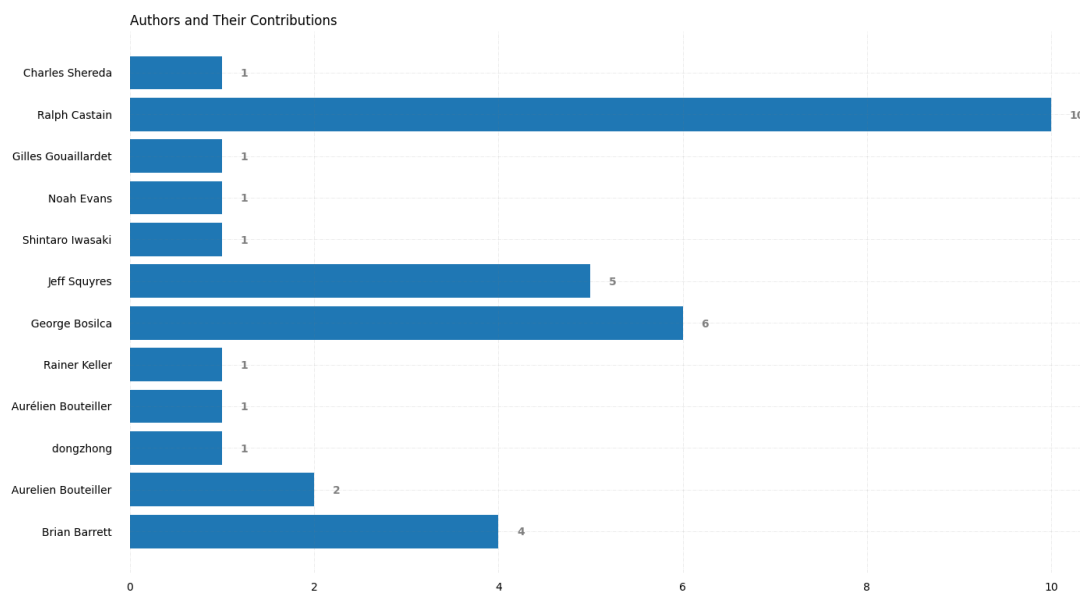


Fig. 4: Visualization of authors and their contributions to repository over the past two years

In Fig. 4, the plot visualizes the authors involved in modifications over the past two years and the extent of their contributions. As observed from the bar plot, Ralph Castain makes the highest contribution to the test files with 10 commits. Jeff Squyres and George Bosilca are the next largest contributors with 5 and 6 commits respectively.

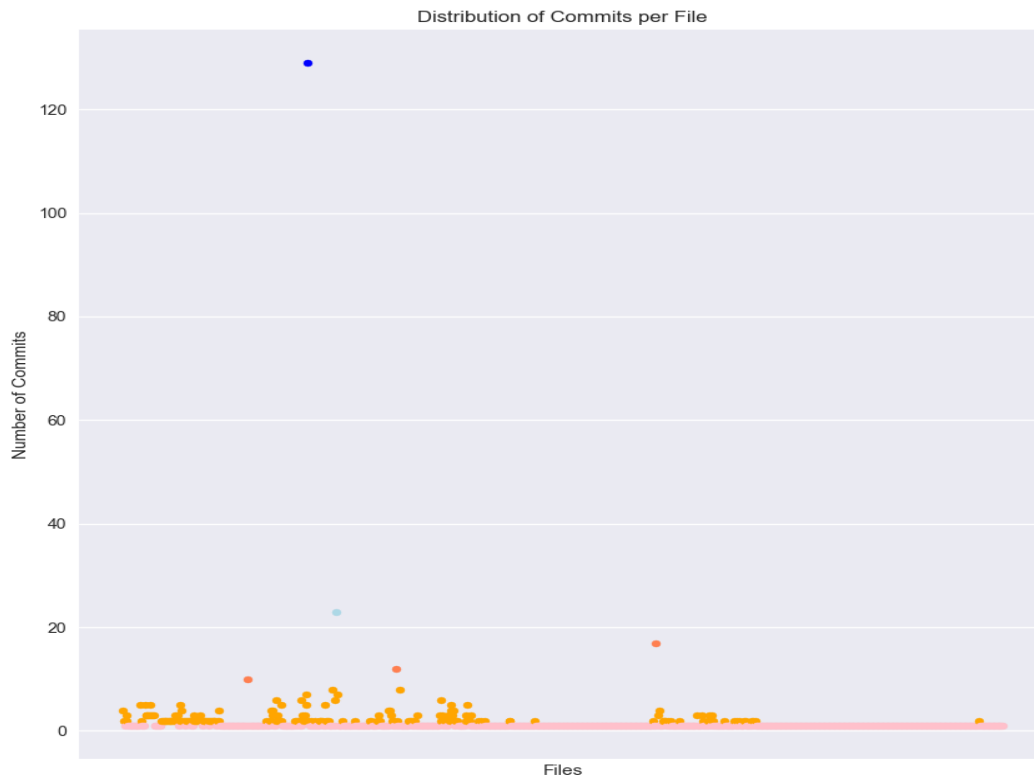


Fig. 5: Representation of number of commits per file

The plot seen in Fig. 5 visualizes the distribution of commits across test files in the Open MPI repository over the past two years. The pink markers represent test files with only one commit. The orange markers represent test files with commits between 1 and 10, coral represent 10 to 20 commits, light blue markers represent 20 to 30 commits and dark blue represents over 120 commits. As observed from the plot above, most files have only one commit. The next largest cluster of files are represented by the orange markers. They have commits between 1 to 10. The file with the highest number of commits is 'Makefile.am' with a total of 129 commits.