A Large-Scale Study of MPI Usage in Open-Source HPC Applications

Anusha Katha
Dept. of Electrical and
Computer Engineering
Queen's University
ELEC 878
Email: 22ak3@queensu.ca

The Message Passing Interface (MPI) has become the preeminent parallel computing programming model in the high-performance computing (HPC) era, allowing for effective coordination and communication across distributed processes. For the purpose of streamlining HPC applications, educating standardizing organizations, and directing the purchase of HPC equipment, it is essential to comprehend the state of the practice in MPI usage. Nevertheless, prior research on MPI utilization has either concentrated on tiny data samples or has been restricted to particular HPC facilities, leaving a gap in our understanding of MPI usage in large-scale applications. In order to get useful insights into the state of the art in MPI applications, the paper[1] presents the first comprehensive analysis of MPI usage across a broad sample of over 100 unique MPI applications. The purpose of the study is to examine the most popular features, the complexity of the code, and the programming models and languages utilized in these apps.

MPI is now utilized and will continue to be used in several applications. So, we ought to be aware of the characteristics that are currently in use and those that will soon be required. This report, in my opinion, is the first to extensively research 110 HPC applications. From there, we may gather a tonne of information and draw a tonne of different inferences.

The majority of MPI applications rely on a relatively small subset of MPI features, suggesting that a focus on optimizing these core features could yield significant performance improvements for a wide range of applications. Code complexity in MPI applications varies greatly, with some applications exhibiting simple, straightforward communication patterns while others involve more intricate and

complex interactions. This diversity highlights the importance of considering both simple and complex cases when optimizing and designing MPI libraries and tools. MPI applications employ a variety of programming models and languages, with C and Fortran being the most prevalent. However, we can also observe an increasing trend towards hybrid programming models that combine MPI with other parallel programming paradigms, such as OpenMP and CUDA, to leverage the full potential of modern HPC architectures.

A total of 6 research questions were taken into consideration. The following communities are said to be benefited: Procurement decision-makers, New MPI application programmers, MPI library implementors, Communication hardware vendors, Closed-source application programmers, HPC tool developers, and the MPI Forum.

WHY STATIC ANALYSIS?

The advantage of run-time analysis is that it provides a mechanism to analyze the runtime communication pattern and MPI usage with given inputs in specific HPC systems. And coming to its disadvantage, it is not a practical method to analyze a large number of applications because the applications must be complied with and run which is demanding and time-consuming, which can be a very good reason for us to go for static analysis.

Static analysis is a potent method for perusing a program's source code without running it. They have created an analysis framework that statically parses MPI programs to determine the usage of MPI routines or calls in the context of the study on the use of MPI in applications. This framework goes through each directory in an application's source code and looks at each file to see if any MPI calls are present. The analysis framework not only reports the MPI calls utilized in a program, but also the programming language (C, C++, or Fortran), the number of lines of code (excluding comments), and whether the application makes use of other parallel programming techniques. paradigms for multi-threading such as CUDA, OpenMP, OpenCL, and/or OpenACC. The cloc tool is used by the Python static analysis framework to count lines of code and identify programming languages. This framework offers valuable information that can direct the optimization and design of MPI libraries and tools, as well as guide decisions regarding the purchase of HPC systems and standardization efforts. It does this by offering insights into the usage of MPI calls and other parallel programming models in a variety of applications.

These 110 applications are chosen based on the crawling strategy, which includes randomness, selecting a significantly large sample size and Heterogeneity of the applications.

PROJECT:

I chose 48 applications in total for which I received the results. Also, there are about 10 apps where I had no luck. Some applications I looked for on GitHub weren't there. Several applications have versions that are distinct from the versions they discuss in the paper.

PROCEDURE:

Download mpiusage.py and cloc from the source code they gave in the paper in a single location. Cloc is used by mpiusage.py to count the number of lines in the code. In its root directory, mpiusage.py will look for cloc. I downloaded all of the applications' source codes from GitHub. The source code for the application is first downloaded as a zip file, which needs to be unzipped before being placed in the folder containing mpiusage.py and cloc. Run mpiusage.py with the application you want to analyze as input, either as a file or a directory. For example, if you want to analyze LULESH, which is in the path /path/to/LULESH-2.0, you would run:

./mpiusage.py /path/to/LULESH-2.0

MPI Usage will by default output statistics in a JSON format in the standard output:

```
kaathaanusha@Kaathas-MacBook-Air app % ./mpiusage.py LULESH-master
{
  "MPI_COMM_SIZE": 1,
  "MPI_REDUCE": 1,
  "MPI_ISEND": 26,
  "X_MPI_COMM_WORLD": 53,
  "MPI_COMM_RANK": 7,
  "MPI_WAIT": 58,
  "MPI_INIT": 1,
  "MPI_ALLREDUCE": 1,
  "X_MPI_MIN": 1,
  "MPI_WAITALL": 1,
  "MPI_FINALIZE": 2,
  "MPI_ABORT": 12,
  "MPI_WTIME": 1,
  "MPI_IRECV": 26,
  "X_MPI_MAX": 1,
  "MPI_BARRIER": 1,
  "MPI_INIT_THREAD": 1,
  "OPENMP": 44,
  "OPENACC": 0,
  "CUDA": 0,
  "OPENCL": 0,
  "C_LINES": 0,
  "CPP_LINES": 4715,
  "C_CPP_H_LINES": 760,
  "FORTRAN_LINES": 0,
  "LINES_OF_CODE": 5546
}
```

Some applications for which I couldn't get any results are as follows:

```
kaathaanusha@Kaathas-MacBook-Air app % ./mpiusage.py CGM-master
{
  "OPENMP": 0,
  "OPENACC": 0,
  "CUDA": 0,
  "OPENCL": 0,
  "C_LINES": 0,
  "CPP_LINES": 0,
  "C_CPP_H_LINES": 0,
  "FORTRAN_LINES": 0,
  "LINES_OF_CODE": 13741
}
```

As discussed in class, there can be a version difference or maybe the application is not using MPI at all.

RESULTS:

After collecting the data of all the applications, I plotted various graphs some same as in the paper and some different.

Coming to the first graph, is Unique MPI features vs % of total applications as shown below:

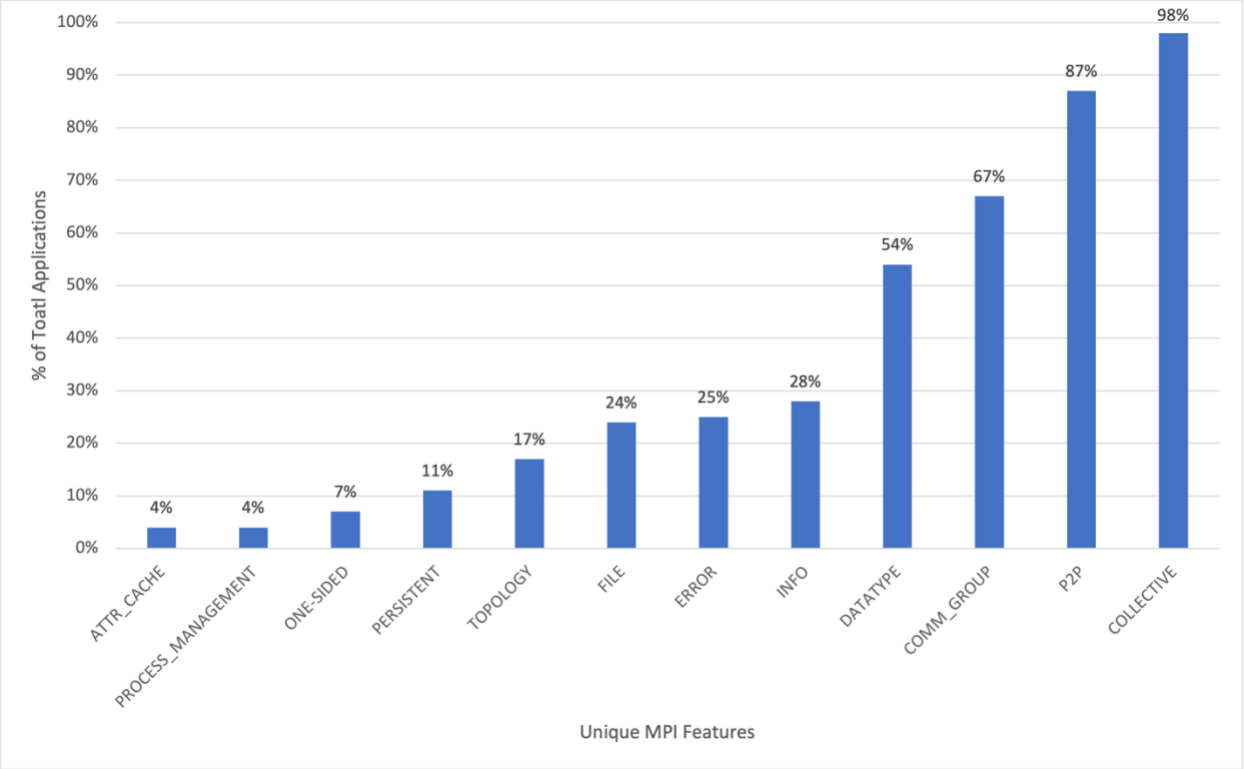

Figure 1: MPI features shown by the percentage of applications using them

From the graph, we can observe that the most used MPI calls are Collective and point-to-point, and the least used is ATTR_CACHE. And coming to the process management feature, which hasn't been used much in codes despite being introduced a long time ago. The reason for this is that most production environments use batch schedulers, which
a) have little to no support for dynamic changes in the runtime size of MPI programs,

b) production applications specify their maximum size of process utilization at startup, and
c) the majority of applications operate in terms of the build-down from MPI COMM WORLD, rather than the more The latter also has a convoluted to laborious syntactic structure (e.g., merging multiple inter-communicators).

Figure 2 shows the per-application usage of MPI features. Different applications use different sets of MPI features.

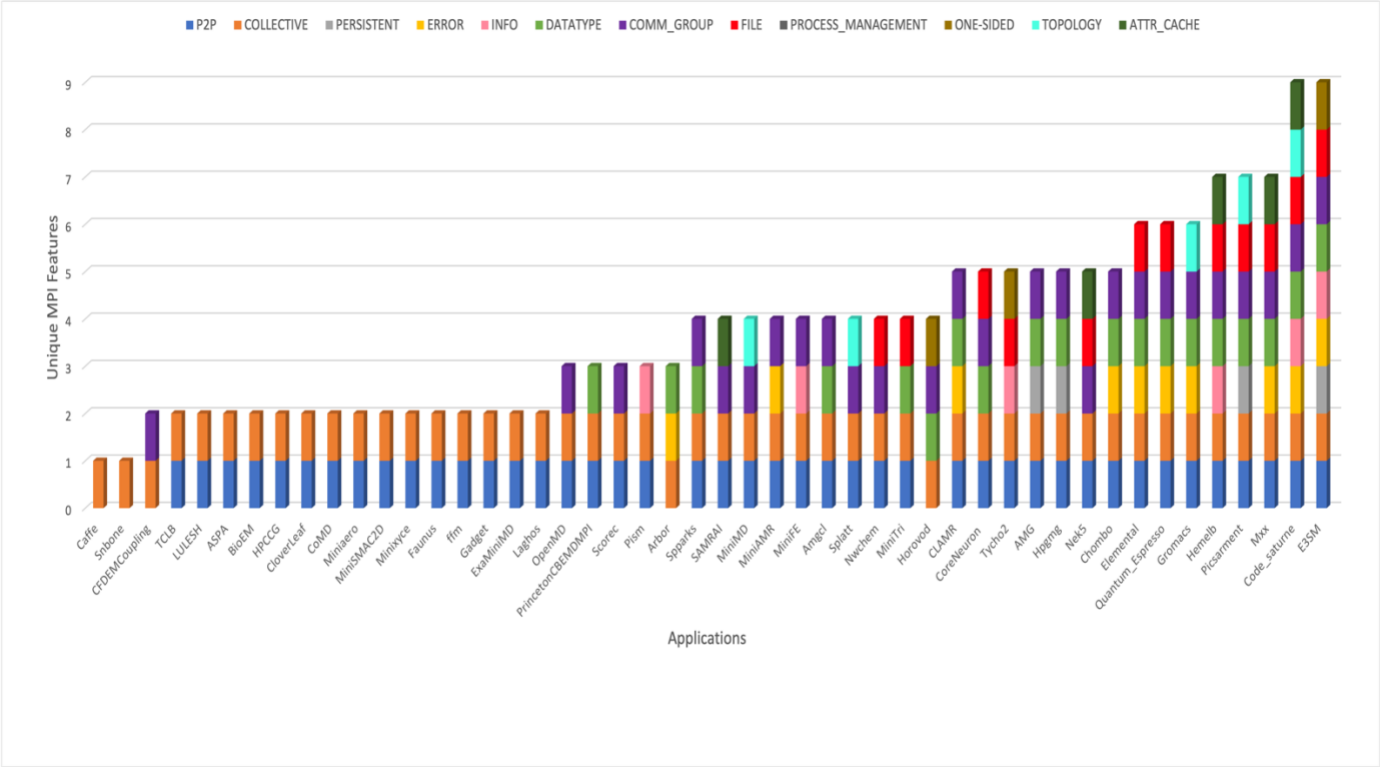

Figure 2: Usage of unique MPI features by applications.

We can see that almost all the applications use point-to-point and collective commands. And that no graph uses all the 13 MPI features.

Figure 3 shows the collective routines versus the number of applications that use these routines.

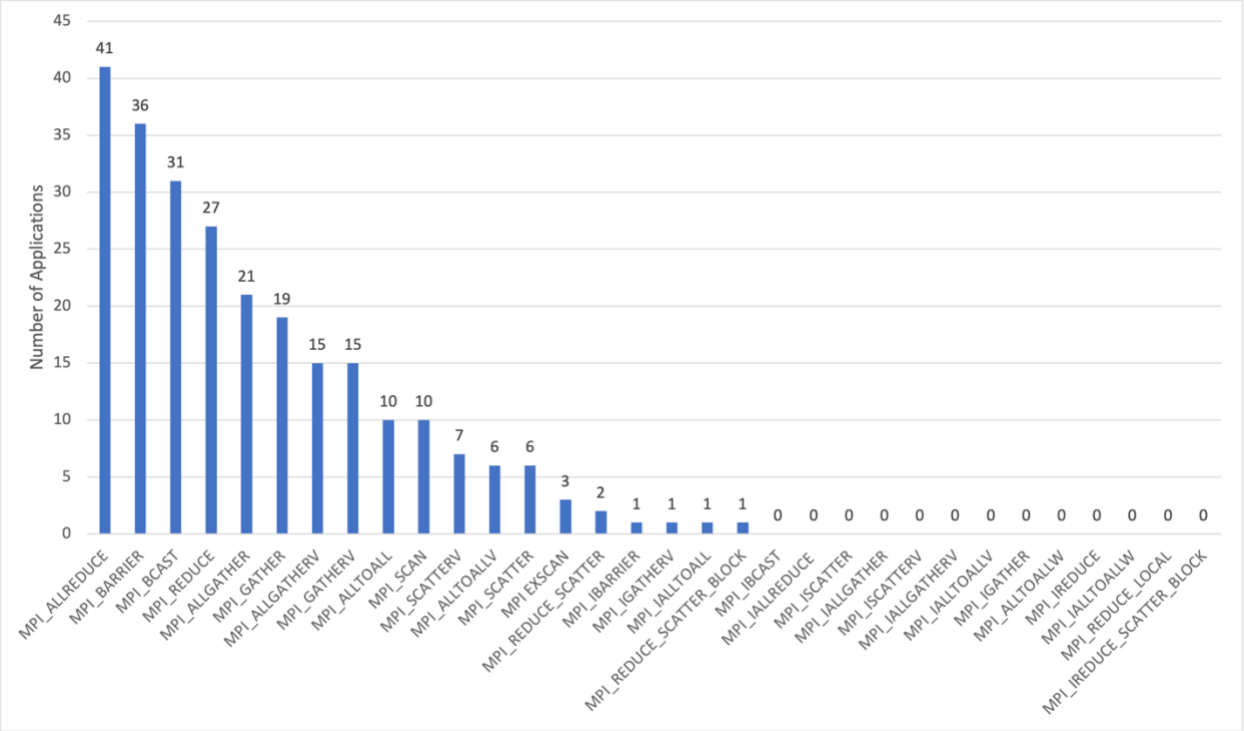

Figure 3: MPI collective function usage

We can observe from the graph that MPI_ALLREDUCE is the most used MPI collective call followed by MPI_BARRIER. I have also included the function calls which are not used by any applications so that we can observe the calls which are not being used very often.

Figure 4 shows the usage of point-to-point communication routines of the data sample.

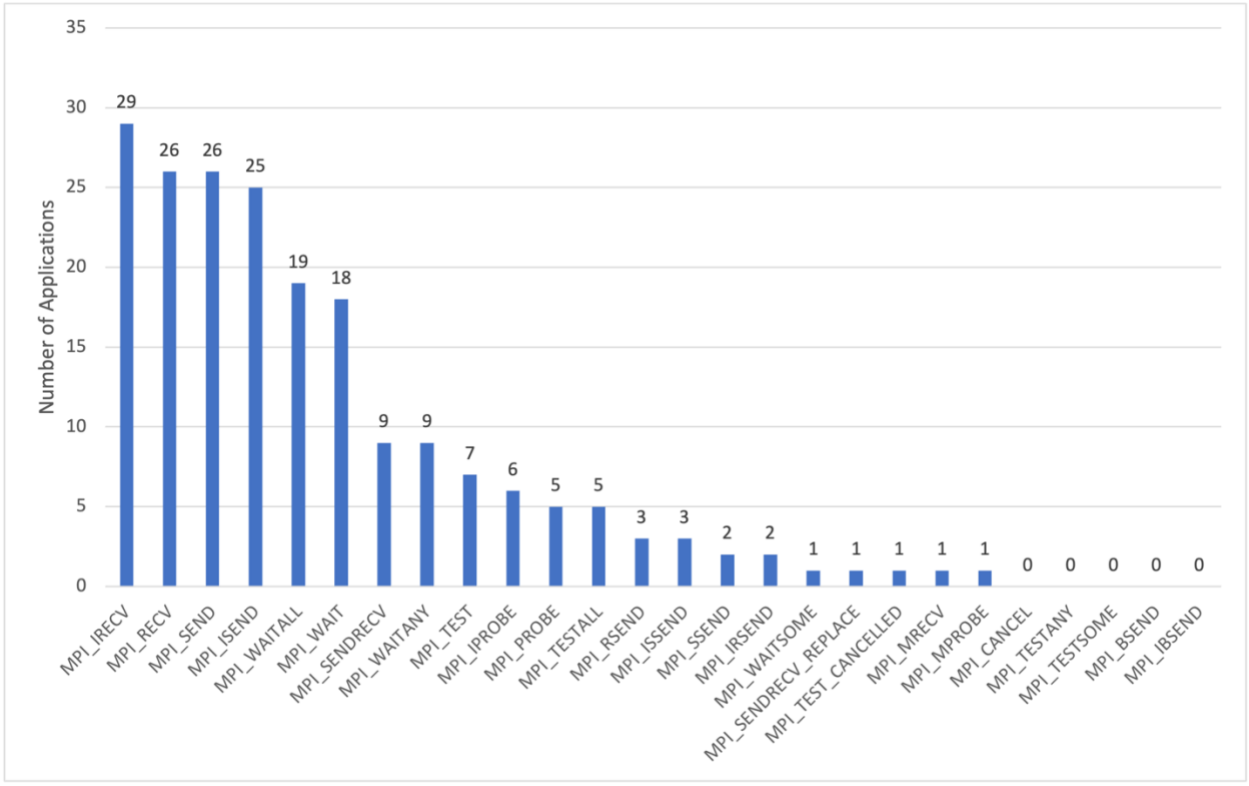

Figure 4: MPI point-to-point function usage

We can see that MPI_IRECV is the most used point-to-point function call used followed by MPI_RECV. Applications use more non-blocking point-to-point calls than blocking calls.

Figure 5 shows the correlation between lines of code and Average of Unique MPI Feature

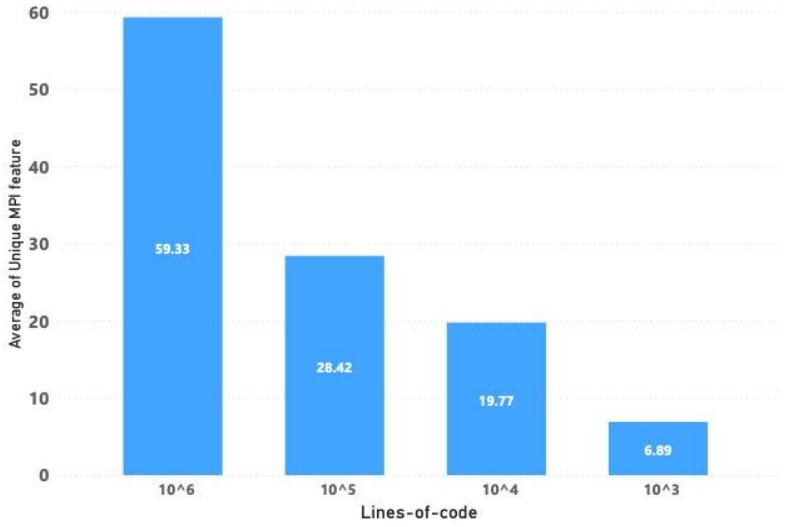

Figure 5: Average Unique MPI functions used by applications of various sizes

The paper has given the graph in a scattered graph format, but I have taken the average of the values because it would be easy for us to analyze that as the number of lines in the code increases, unique MPI features usage also increases.

Coming to the sixth graph, we can observe that the applications with fewer code lines tend to use C and C++ while s the number of code lines increases, the programmers tend to move towards Fortran or use a combination of all the languages.

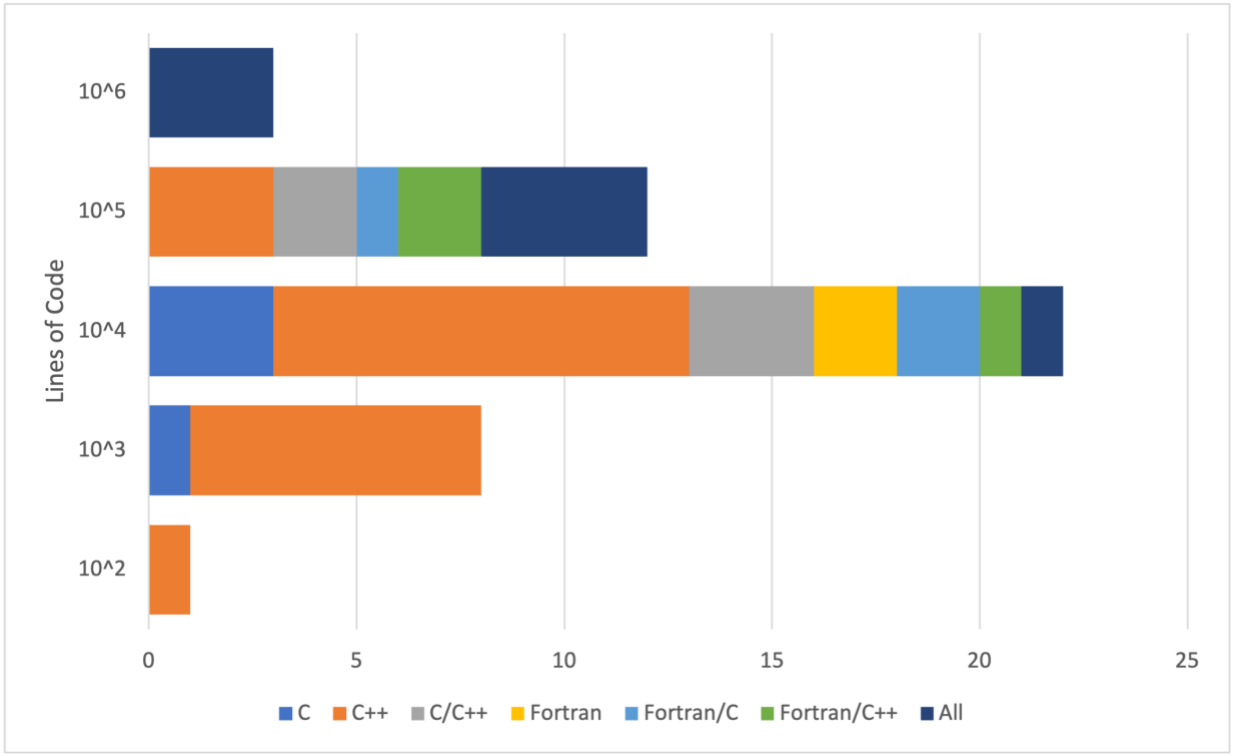

Figure 6: Application count with respect to their code size

From Figure 7, we can observe that most applications use a mixture of languages and that C++ is the most used language and Fortran is used the least.

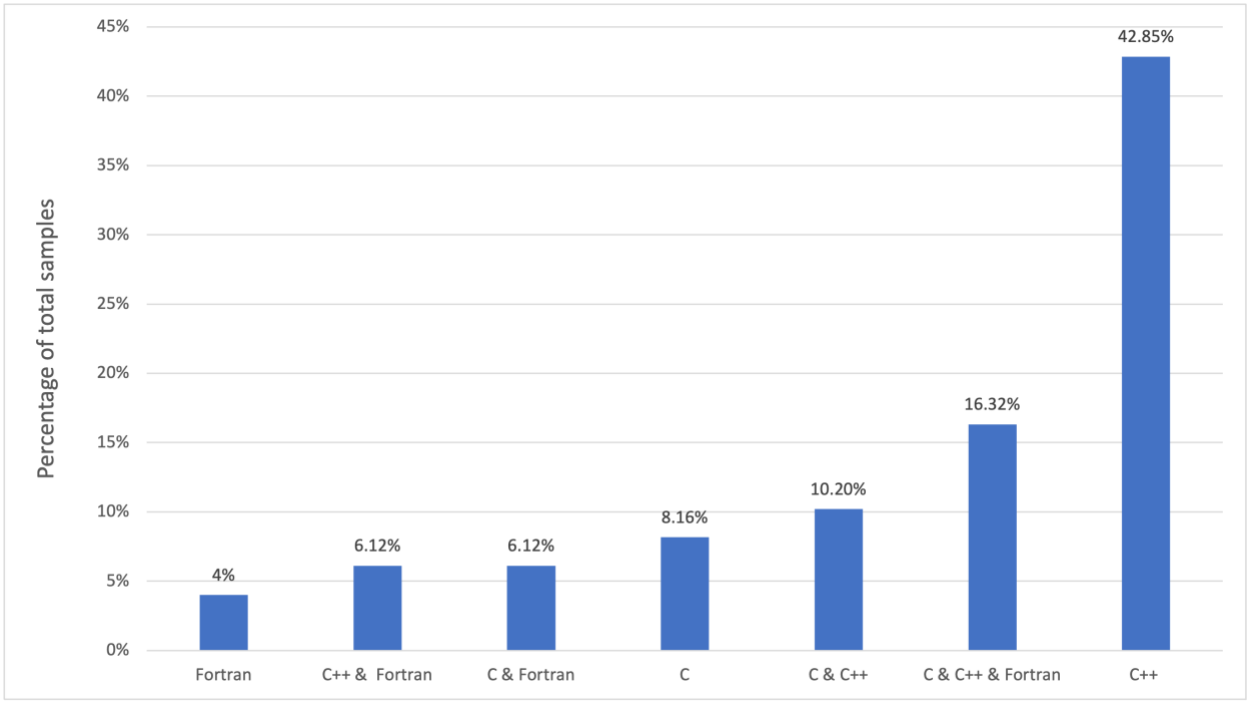

Figure 7: The percentage of applications using specific programming languages.

From Figure 8, we can observe that many of the applications don't use any multi-threaded programming and Open MP is used the most followed by a combination if CUDA and Open MP.

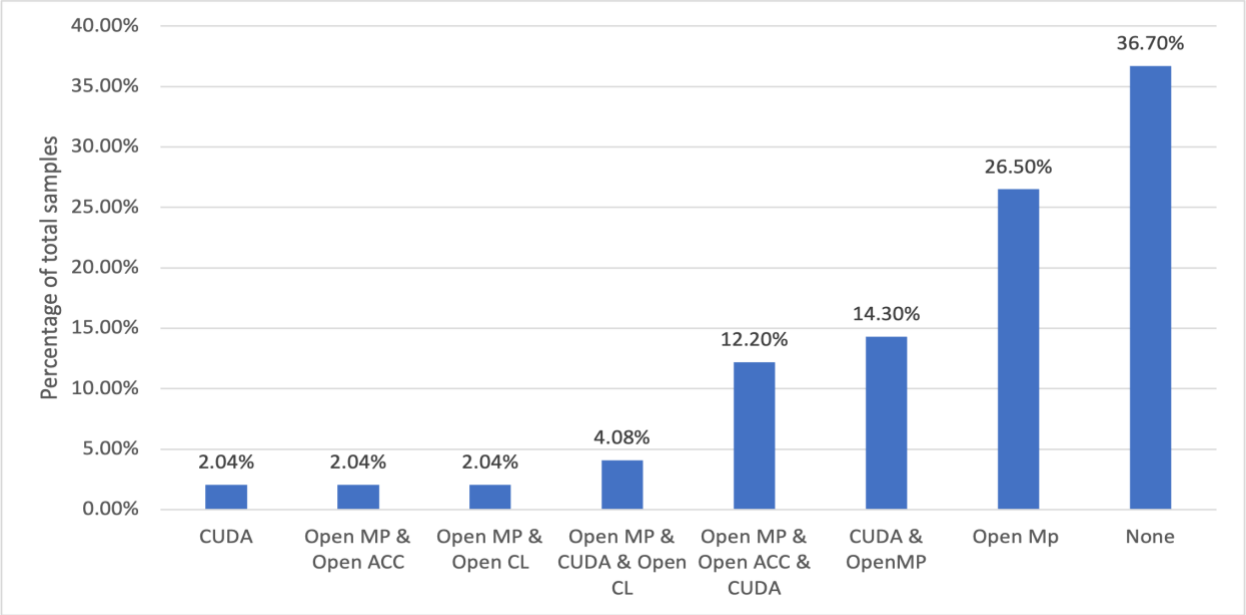

Figure 8: Percentage of total applications using multithreaded programming models.

THOUGHTS:

Many inferences can be made from the data and graphs. I believe that the MPI community would benefit much from looking into these findings in order to comprehend the present trend and the long-term objectives. It was challenging to gather the data for such a large number of applications, but I found it more challenging to produce the graphs. As I made an effort to show the facts differently than the publication did. It is challenging to plot and sort such vast amounts of data. The paper's merits include its extensive investigation of a wide range of factors, including function usage, compatibility with standard versions, programming languages used, unused standard features, and programming paradigms. successfully investigated the relationship between numerous variables.

The research compared its results with those from other surveys, confirming some usage trends for MPI and providing fresh information. It gave researchers useful information (a table is provided). It may not capture some dynamic behaviour, such as runtime decisions on communication patterns or function usage, when it comes to constraints. The results will vary because the apps' versions are constantly being updated, and the report makes no mention of how these various MPI utilization factors relate to the speed and scalability of the programs. There were 110 applications in the main graph, which I found to be quite difficult to read. It would have been better if they had divided the number or otherwise portrayed it in a different way.

CONCLUSION:

Static analysis, in my opinion, is the greatest approach to learning some practical insights without having to worry about setting up the applications for dynamic analysis and does not capture all the run-time analysis features provided as input. Both studies plainly have different goal qualities and data scopes, therefore they each have advantages and downsides of their own. The study provides both new insights and confirmation of several previously documented trends in MPI utilization (albeit on a greater scale).

REFERENCES:

[1] Laguna, Ignacio, et al. "A large-scale study of MPI usage in open-source HPC applications." *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2019.