# Automated Collection of OpenMP Usage in HPC Applications

Bachelor Thesis

University of Basel
Faculty of Science
Department of Mathematics and Computer Science
HPC Research Group

Examiner: Prof. Dr. Florina M. Ciorba
Supervisor: Thomas Jakobsche

Author: Ilhan Kizildere
Email: i.kizildere@stud.unibas.ch

September 25, 2022

**Acknowledgments**

From April 6th 2022 to September 25th 2022, I was intensively engaged in researching and writing this bachelor thesis with the support of my examiner, Prof. Dr. Florina M. Ciorba, and supervisor Thomas Jakobsche. Thanks to their expertise in High Performance Computing, both of them provided me with valuable insights into the subject matter. They answered my open questions and gave me valuable input for the methodological approach so that I could successfully finish my work. With them, I discussed every step of my thesis and I realized that my findings were taken seriously as well as it could make a significant difference for their own purposes. Before the time of writing my thesis, I had only partial experience with High Performance Computing.

Therefore, I would like to thank my examiner and supervisor for their valuable guidance and support during this process. I also express my gratitude to all my family members, good friends and colleagues; without their previous support during my study time, I would not have been able to even start this thesis.

**Abstract**

OpenMP is the most used standard for shared memory parallel programming in high performance computing (HPC). Despite its popularity it is not clear which or how many OpenMP features are actively used by the HPC community. The contribution of this thesis is a large-scale study on source code of OpenMP applications to understand the state-of-the-practice in parallel programming with OpenMP. This thesis is focused on understanding the characteristics of OpenMP usage in terms of the most commonly used features and functionalities. The goal is to assess the current state-of-practice in OpenMP usage, which is important for many aspects of HPC: (a) find opportunities for researchers to optimize the scheduling and load balancing of HPC applications, (b) identify the most commonly used OpenMP features to focus optimization efforts by vendors, centers, and developers, and (c) find unpopular or completely unused OpenMP features to inform standardization bodies. To obtain these insights, we developed a Python script that automatically extracts and visualizes OpenMP usage in the source code of a large portfolio of HPC applications. The main findings for our set of applications are: (1) functionalities provided by the OpenMP *execution environment routines* are used by 90% of all applications, the most used routine from this group is *omp_get_thread_num* which is also the most used functionality overall, (2) the most used OpenMP directives (starting with *#pragma omp*) are *for*, *critical*, and *barrier* (all directives were introduced in OpenMP version 1.0), and (3) the synchronization construct *critical* is present in more applications compared to *barrier* or *atomic*, synchronization constructs regarding *tasking* are used the least, (4) the most used OpenMP scheduling clause is *static* with default chunk size, followed by *dynamic* with custom chunk size, the least used clause is *guided* with custom chunk size, (5) the most used combination of programming paradigms is OpenMP + MPI, followed by OpenMP + MPI + CUDA, these combinations are used more often compared to pure OpenMP, and (6) C is the most used programming language, followed by C & Fortran, C & C++, and C & C++ & Fortran, the least used combination of programming languages is Fortran & C++.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

This bachelor thesis addresses the 'Automated Collection of OpenMP Usage in HPC Applications'. The work behind this is a quantitative large-scale study, that was conducted on current OpenMP applications and their usage which were subsequently evaluated. The final goal was to find out the usage of OpenMP functionality in HPC applications

High performance computing (HPC) is the most used way to increase and to upscale system performance. Parallel computing allows applications to run on multiple processors or threads. This means, a program can be split into many (sub-) routines which can run simultaneously on different processors of the same single computer system [1]. With that, developer and scientists have a useful tool for their applications to meet their demand for higher performance with low costs and accurate results.

Today's HPC computer systems are powerful enough to run application programs on multiple processors, to achieve this, many parallel programming paradigms are available for these computer systems such as OpenCL, OpenACC, CUDA, MPI and OpenMP. OpenMP is the main focus of this thesis because it is the most used standard for shared memory parallel programming.

## 1.1 Motivation, Goal and Research Questions

University of Basel, Switzerland, consists of various departments and research groups, including the DMI-HPC [1] research group (https://hpc.dmi.unibas.ch/en) led by Prof. Dr. Florina M. Ciorba. Here, Students, Doctorates and other member of the staff use a lot of features provided by OpenMP standards for their HPC source codes. In order to improve the quality of these HPC source codes we will take a closer look into the OpenMP applications. For this purpose, we collected a big set of applications, which use the OpenMP programming paradigm. There are 109 samples for examination. Our goal is to understand the state-of-the-practice in OpenMP usage through source code analysis. For this purpose we will answer the following research questions:

(a) What OpenMP functionalities are used most often?

[1] Department of Mathematics and Computer Science – High Performance Computing Group

(b) Which OpenMP version introduced the most used functionalities?

(c) What other technologies (MPI, CUDA etc.) are used alongside OpenMP?

(d) What are the most used OpenMP scheduling clauses?

The main goal of this thesis is to extract and visualize OpenMP usage in source code of given HPC application samples. To answer these questions, a quantitative large-scale study was conducted on OpenMP applications and their usage. The focus of this thesis is to understand the characteristics of OpenMP usage with respect to the most used features provided by OpenMP standards.

With the help of this thesis, we want to find out which and how many of OpenMP features are actively used by the HPC community. Understanding the state-of-the-practice in OpenMP usage is important for many aspects of HPC:

(a) To find opportunities for researchers to optimize the scheduling and load balancing of HPC applications,

(b) to identify the most commonly used OpenMP features to focus optimization efforts by vendors, centers, and developers, and

(c) to find unpopular or completely unused OpenMP features to inform standardization bodies.

## 1.2 Outline

After talking about motivation, goal and research question in section 1.1 (Motivation, Goal and Research Questions), this section is about getting an overview of this thesis. We start with chapter 2 (Related Work), where we look at the work that served as inspiration for our thesis. In section 2.1 (Laguna et al. vs. Older Work) we have created a table so that we can get an overview of the scientific paper written by Laguna et al. and also how his work differs from previous work of similar research area. Then in section 2.2 (Laguna et al. vs. Thesis), we venture a comparison of his work with our work and how our work differs. We have tabulated this as well. Then in the next chapter 3 (Background), we briefly look at OpenMP and its meaning (see section 3.1 (OpenMP)). In sections 3.2 (OpenMP Feature Groups), we try to get a rough overview of the OpenMP features and OpenMP functions. Chapter 4 (Methods) is mainly about how we create our plots. Here, we get an overview of our samples in section 4.1 (OpenMP Applications). In section 4.2 (Usage Collection Script), we dive into our scripts. In the last section 4.3 (Plotting with Excel), we briefly try to understand how we graphically displayed our data using MS Excel. In chapter 5 (Results), we talk about our results of the plots, each representing a sub-chapter. There is also a description of what these plots are about. Afterwards in chapter 6 (Discussion), we describe our thoughts and findings about the results from chapter 5 (Results). In final chapter 7 (Conclusion), we summarize our findings and mention where the future approaches for related work might be.

# Chapter 2

# Related Work

In 2019, a scientific paper titled "A Large-Scale Study of MPI Usage in Open-Source HPC Applications" was published by Laguna et al. [3].

Laguna et al. [3] produced the first extensive static analysis of source code of HPC application that use the MPI programming paradigm. Their work produced multiple interesting and important insights into the usage of MPI and what that means for the MPI standard as a whole (e.g. unused features and versions). According to Laguna et al. [3] static source code analysis can lead to important insights. This study then served as the source of the inspiration for our own thesis. Hence, we oriented ourselves on the available related work that analyzed MPI applications.

Laguna et al. surveyed more than a hundred MPI applications. They focused on understanding the characteristics of MPI usage with regards to the most used features, programming models, and languages. The research questions defined in their work form the basis of our own investigation, since most of the questions regarding programming paradigm usage in source code can be applied to any programming paradigm. The most important findings of Laguna et al. are summarized below:

- a large number of MPI applications do not use advanced features of the MPI standard (e.g. the majority of applications use blocking send and receive),
- only a small set of features provided by the MPI standard are actually used, some parts of the standard are unused by many applications,
- many applications only employ functionality of the MPI version 1.0, and not of newer versions, additionally, features introduced in sub-versions are rarely used,
- MPI is used together with other programming paradigms, OpenMP is the most popular combination,
- C++ is the most used language in MPI programs.

The work of Laguna et al. is a static code analysis of MPI applications, it is very important for us because it serves as an inspiration and blueprint for our own static analysis of OpenMP applications.

## 2.1 Laguna et al. vs. Older Work

The large-scale study of MPI usage of Laguna et al. was also based on older work by other HPC experts [3] - but Laguna et al. changed their focus from related scientific work. We also took over most of these changes for our own thesis. In table 2.1 we see an overview of how the focus of Laguna's work shifted from the related work by other authors [8] [2] [4]:

Table 2.1: Comparison between Older Work and Laguna et. al.

| Focus | Older Work | Laguna et al. |
|---|---|---|
| Size of samples (to understand the MPI usage) | Small set of samples (applications) | Much larger set of samples<br>• over 100 unique MPI programs<br>• containing a total of more than forty million lines of code |
| Characteristics of MPI usage | Focus on dynamic characteristics of MPI usage [1] | Focus on static characteristics of MPI usage [2] |
| Source of samples | • Samples from specific projects<br>• Samples from specific HPC centers | Random samples from many sources – therefore more realistic survey of the study |

## 2.2 Laguna et al. vs. Thesis

In this thesis we want to produce similar insights for HPC applications that use the OpenMP programming paradigm. Laguna et al. produced important related work for us, because our approach takes inspiration of it, and we orient some of our static source code analysis on their study of MPI applications. In some cases we introduce new analysis that is tailored to OpenMP (e.g. counting and analyzing the usage of schedule clauses). This work is not a reproduction of the work done by Laguna et al. but a continuation of the static source code analysis of HPC applications, in this case, of OpenMP applications. Table 2.3 shows how the two works differ from each other.

[1] dynamic characteristics are those characteristics of running applications – aka characteristics on run-time of a source-code after or during compile-time

[2] static characteristics are non-compiled characteristics of a source-code

Table 2.3: Comparison between Laguna et. al. and Thesis

| Focus | Laguna et al. | Thesis |
|---|---|---|
| Research focus | Findings about the usage of MPI | Findings about the usage of OpenMP |
| Size of samples | About 100 unique MPI samples | 109 unique OpenMP samples |
| Static characteristics of MPI/OpenMP usage | Focus of the work | Same is true for us |
| Dynamic characteristics of MPI/OpenMP usage | Not interested | Same is true for us |
| Source of samples | Random samples from many sources – therefore more representative survey of the study | Also random samples of OpenMP applications with open source code. |
| Scalable program analysis method | Written in Python | Same is true for us |
| Lines-of-code counting | Use of external tool named *cloc* | Our own solution written inside in our Python code |
| Detection of programming languages | Use of external tool named *cloc* | Our own solution written inside in our Python code |
| Analyzed programming languages | • C<br>• C++<br>• Fortran<br>• others (with no further distinction) | • C / C++ (handled as one and same programming language), if not explicitely mentioned<br>• Fortran<br>• others (with no further distinctions between other programming languages) |
| Multi-threaded programming models | MPI + X model, where X stands for:<br>• OpenMP<br>• OpenCL<br>• OpenACC<br>• CUDA<br>• none<br>• Combination of the above mentioned | OpenMP + X model, where X stands for:<br>• MPI<br>• OpenCL<br>• OpenACC<br>• CUDA<br>• none<br>• Combination of the above mentioned |

# Chapter 3

# Background

This chapter includes a small introduction to OpenMP and how features are sorted into feature groups.

## 3.1 OpenMP

OpenMP (Open Multi-Processing) is an API (Application Programming Interface) for parallel execution of application (sub-) routines to increase this computational speed. This API supports multiprocessing programming with C/C++ and Fortran, among others [7]. It focus lies in multi-platform and shared-memory programming. OpenMP is the most used standard for this purpose [1]. It is compatible with many computer platforms, instruction-set architectures and operating systems like Unix Systems, Linux, macOS, and Windows [5]. And in addition to that, OpenMP uses a portable and scalable model that gives users a simple and flexible interface for developing their parallel applications. This applications can then run on platforms ranging from standard desktop computer environment to supercomputer cluster system [1].

According to the Website of OpenMP, OpenMP is managed by the nonprofit technology consortium called OpenMP ARB (OpenMP Architecture Review Board) [6]. This consortium is led by leading computer hardware and software vendors, including ARM, AMD, IBM, Intel, Cray, HP, Fujitsu, NVIDIA, NEC, Red Hat, Texas Instruments, and Oracle Corporation [9].

## 3.2 OpenMP Feature Groups

In this section, we list what we took as "functionality groups" or "feature groups" (e.g. "parallel construct", "worksharing construct", "combined construct") from the OpenMP - Application Programming Interface [5]. OpenMP version 5.0 (Version from November 2021) specifies roughly 180 distinct directives, constructs and routines. Some clauses can be added as well. These can be grouped into 30 categories. In our analysis, we refer these categories as OpenMP features. On the basis of this API, we have compiled a list of all groups and their commands, which we structured this way:

- **feature 1**: a small description about feature 1
  - **functionality 1 in C (and F)**: a small description about functionality 1 of feature 1
  - **functionality 2 in C (and F)**: a small description about functionality 2 of feature 1
- **feature 2**: a small description about feature 2
  - **functionality 1 in C (and F)**: a small description about functionality 1 of feature 2
  - **functionality 2 in C (and F)**: a small description about functionality 2 of feature 2

This list of all groups and their commands is available in appendix A.

# Chapter 4

# Methods

We employ a static analysis of source code by using a python script that crawls all files in a given directory. The generated data were then plotted with MS Excel.

## 4.1 OpenMP Applications

All OpenMP applications (name + release year) for this thesis are listed in Appendix B. Unfortunately, the release year is not always available. In such cases, we used the last update year.

## 4.2 Usage Collection Script

These are the following steps to run our Python script:

- Step 1 (Input): Put samples (folder or file) and all the Python-script in the same directory (root).
- Step 2 (Black Box): Run main.py in terminal. (No parameter required.)
- Step 3 (Output): Use the Excel-files with Excel for plots; (can be found in the root directory).

In case, when not all plots are required, we can select them individually. Therefore, consult the list below to select the requested data:

- figure2.runFig2(automatedListsOfPaths): Figure 5.1
- figure3.runFig3(automatedListsOfPaths): Figure 5.2
- figure4.runFig4(automatedListsOfPaths): Figure 5.4, 5.5 & 5.6
- figure5.runFig5(automatedListsOfPaths): Figure 5.7 & 5.8
- figure8.runFig8(automatedListsOfPaths): Figure 5.11, 5.13
- figure11.runFig11(automatedListsOfPaths): Figure 5.9

- figCountCommandsPerApplication.runFigCounter(automatedListsOfPaths): no corresponding plot (overview of OpenMP commands)
- figCountCommands.runFigCounterAll(automatedListsOfPaths): Figure 5.15
- figure9.runFig9(automatedListsOfPaths): Figure 5.10

To use the Python script, we need following Python libraries:

- re: A library to work with regular expression.
- os: This library has useful functions for working with Operating System. It helps to find information about the system #environment and to get the path of the application, files, etc.
- time: This module has several time functions.
- pandas: Is a package to work with labeled or relational data easily and to do data analysis.

## 4.3 Plotting with MS Excel

After generating the requested information using Python script, we opened the generated data with MS Excel to create a bar chart. Unfortunately, MS Excel cannot handle a bar chart with percentages by itself, so we tricked MS Excel. Therefore, we created two bar charts side by side, usually one for frequency and one for percentages. Then we hid the bar chart for the percentages behind the bar chart for the frequency, after adding the percentage value. Finally, we manually edited the plot to make it more readable, for example, by increasing the font size etc.

## 4.4 Difficulties & Problem-Solving

Partially, the conducted implementation of the Python code showed some difficulties to read the files (with substructures) properly. Sometimes we have OpenMP commands that are spelled the same way at the beginning, for example *#pragma omp distribute* or *#pragma omp distribute simd*, which then leads to problems with correct counting: For example, when an OpenMP application contains the OpenMP command *#pragma omp distribute* twice and the command *#pragma omp distribute simd* once, then the Excel file would store the value 2 for counts at *#pragma omp distribute* and the value 3 instead of 1 for counts at *#pragma omp distribute simd*. Finally, we were able to solve this problem after an extensive manual analysis of the results. Below, we put an example with !$*omp parallel*, !$*omp loop* and !$*omp parallel loop*:

```
import numpy as np
...
def incmatrix(genl1,genl2):
    !$omp parallel
            ...some code
    !$omp end parallel

    !$omp loop
```

```
        ...some code
!$omp end loop

!$omp parallel loop
        ...some code
!$omp end parallel loop
```

Table 4.1: Example of false and true count

| OpenMP command | false count | true count |
|---|---|---|
| !$*omp parallel* | 2 | 1 |
| !$*omp loop* | 1 | 1 |
| !$*omp parallel loop* | 1 | 1 |

# Chapter 5

# Results

Our results are visualized in the following plots.

## 5.1 Unique OpenMP Features

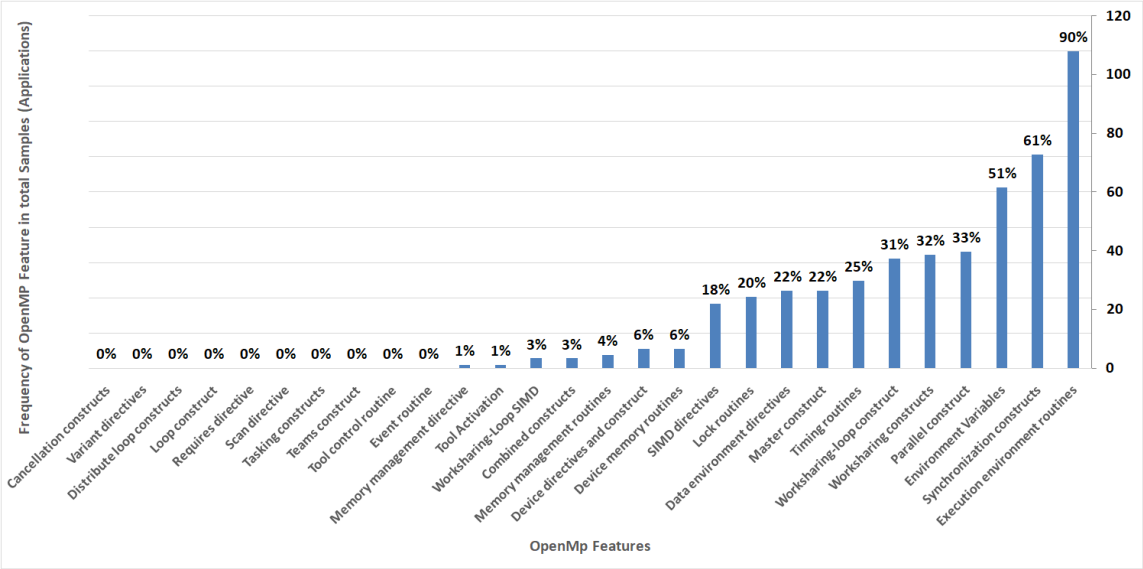


Figure 5.1: Unique OpenMP features, shown as a percentage of samples using them.

Figure 5.1 shows the overall usage of unique OpenMP features which are defined according to OpenMP standards in version 5. We have sorted these features by frequency of occurrence. We have also indicated the percentage of occurrence for better analysis. An overview of these OpenMP features can be seen in appendix A.

## 5.2 Usage of Unique OpenMP Features by Applications

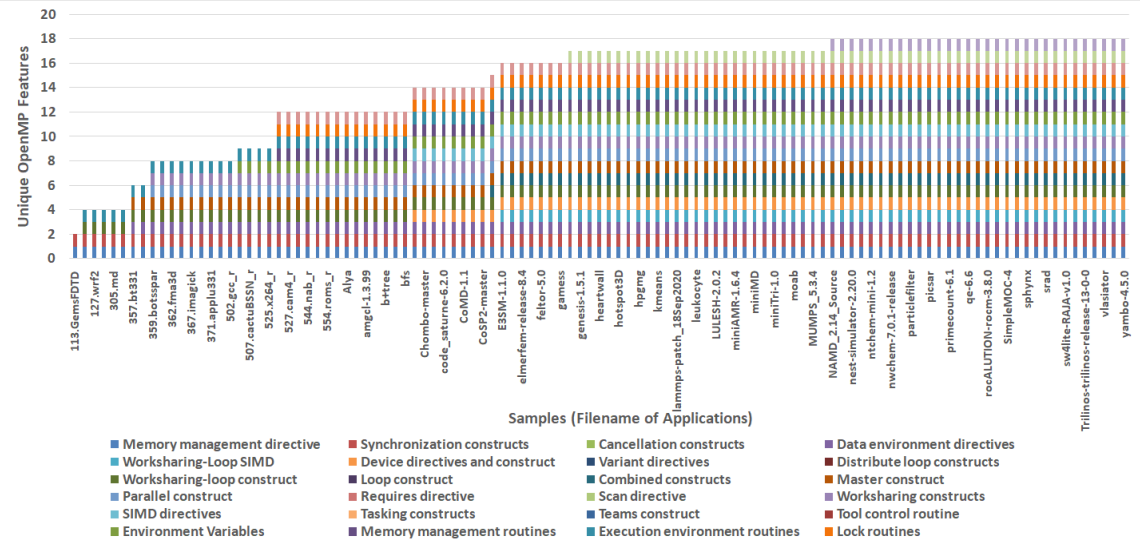


Figure 5.2: Usage of unique OpenMP features by samples. The majority of samples use only a portion of the OpenMP features. We can also state, that no one use all provided features.

Figure 5.2 shows the usage of OpenMP features by each samples. For that reason, we have listed all 109 samples with file names in our plot. Different samples use different sets of given OpenMP features. These OpenMP features are represented with their own colors. Then we sorted the usage of unique OpenMP features of samples by frequency of occurrence. For better view, see figure 5.3, which is an enlarged plot of figure 5.2. An overview of these OpenMP applications with its properties can be seen in appendix B.

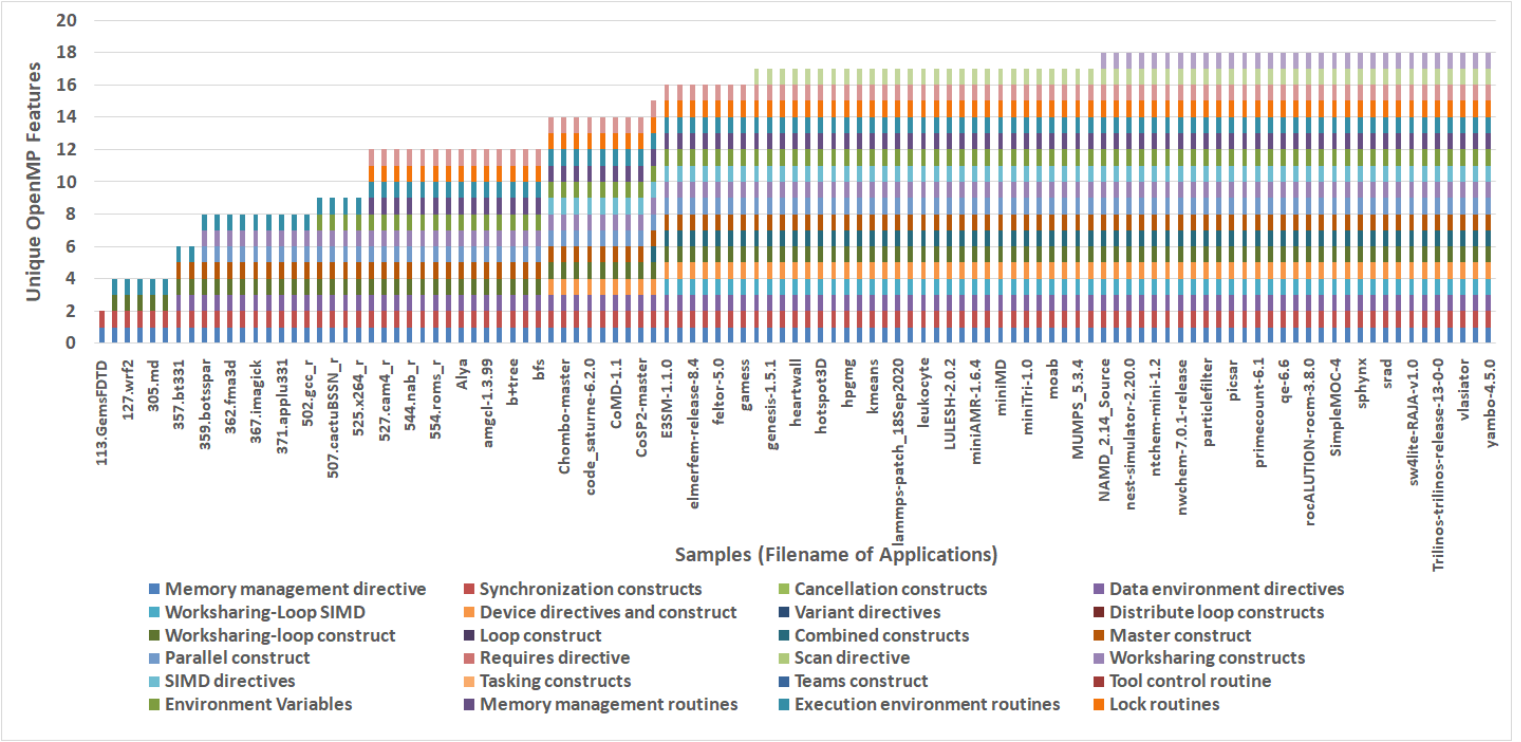


Figure 5.3: Usage of unique OpenMP features by samples. The majority of samples use only a portion of the OpenMP features. We can also state, that no one use all provided features.

## 5.3 Usage of Execution Environment Routines

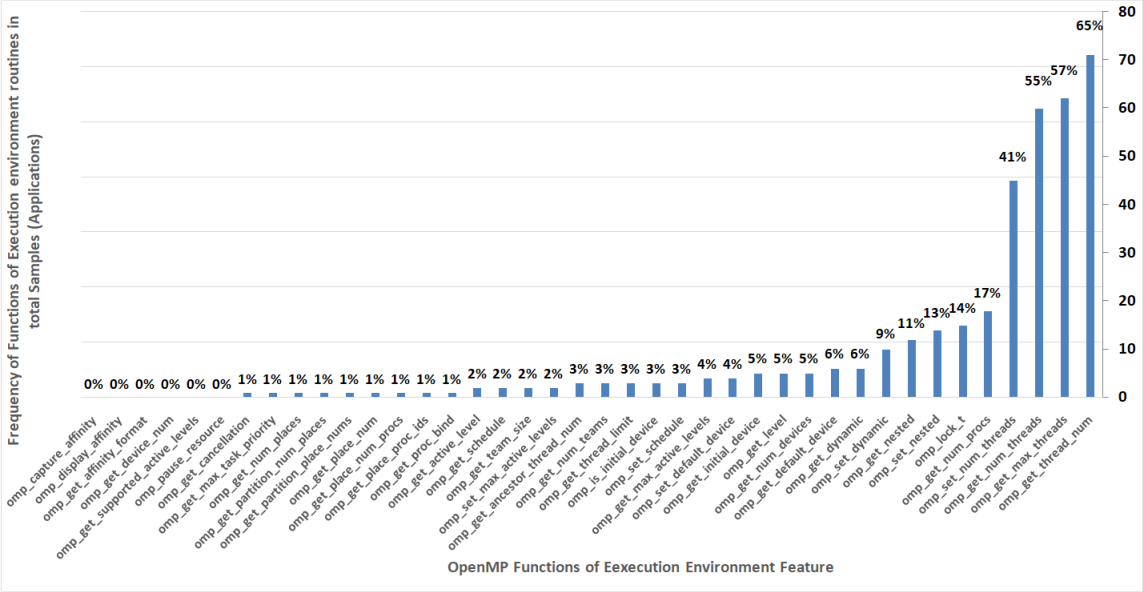


Figure 5.4: Overview of the top 1 OpenMP feature (*Execution environment routines*) used in our set of samples. We find out that *omp_get_thread_num* is the most used function in Execution environment routines, which was used in exact 71 samples out of 109.

Figure 5.4 shows the overall usage of functionalities of top 1 OpenMP feature (*Execution environment routines*) which are defined according to OpenMP standards in version 5. We have sorted these functionalities by frequency of occurrence. We have also indicated the percentage of occurrence for better analysis.

## 5.4 Usage of Synchronization Constructs

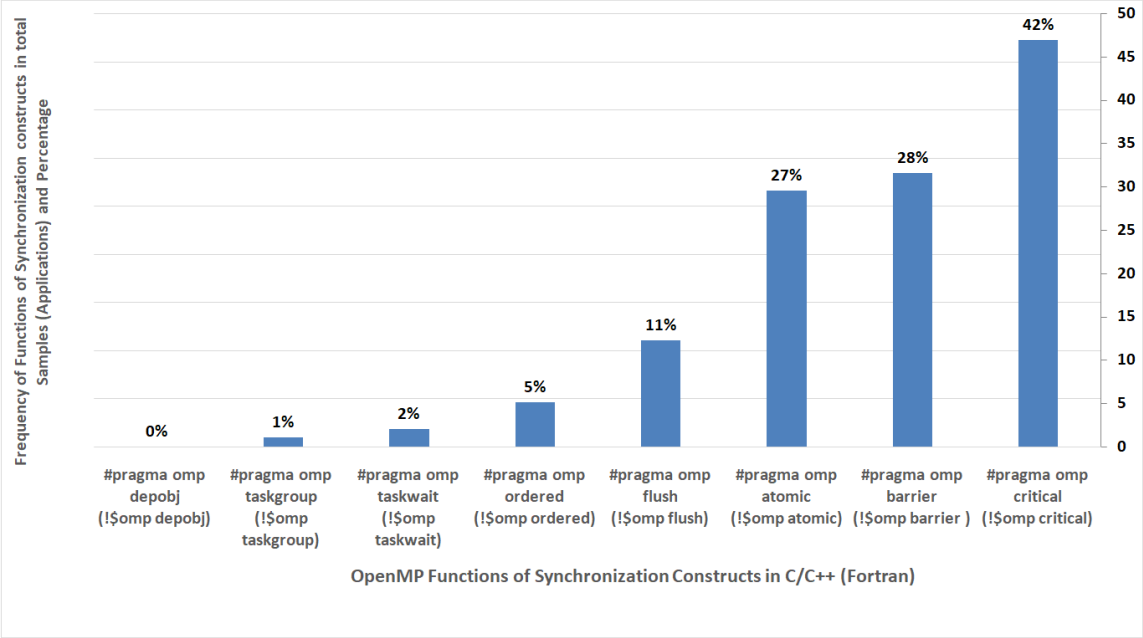


Figure 5.5: Overview of the second most used OpenMP feature (*Synchronization constructs*) used in our samples, where we don't distinguish the usage by C/C++ and Fortran. We find out that *critical* is the most used construct in *Synchronization constructs*.

Figure 5.5 shows the overall usage of functionalities of the second ranked OpenMP feature (*Synchronization constructs*) which are defined according to OpenMP standards in version 5. We have sorted these functionalities by frequency of occurrence. We have also indicated the percentage of occurrence for better analysis.

In this figure, we do not distinguish between the use of these constructs by the two programming languages C/C++ and Fortran. In other words, if this construct occurs in C/C++ or in Fortran sample, we have added plus 1 in both counts of these programming languages because logically they do the same thing.

## 5.5 Usage of Schedule Clauses

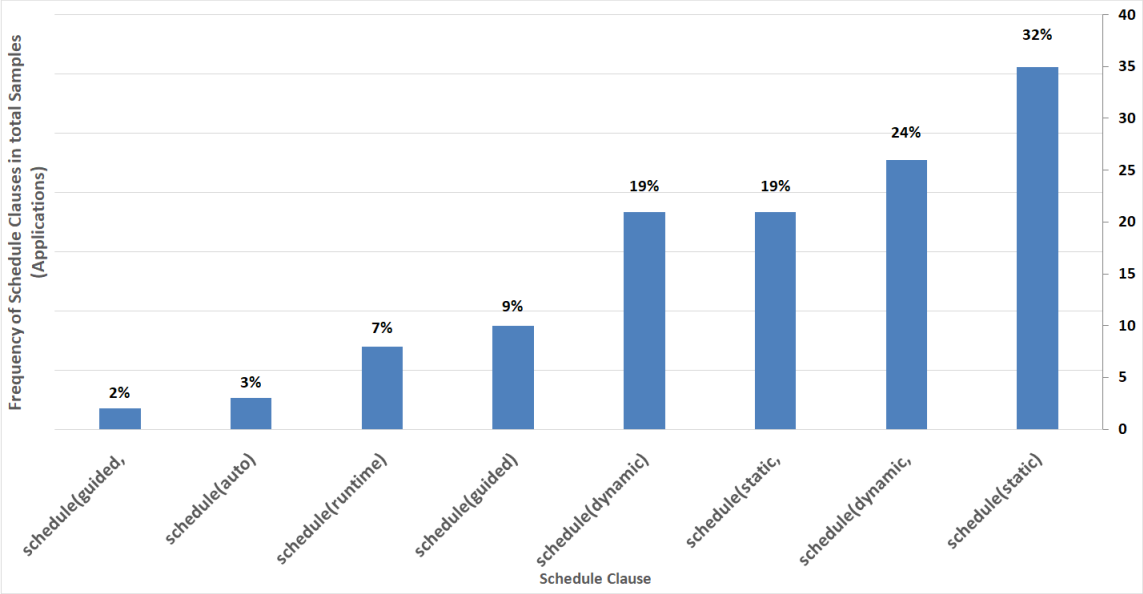


Figure 5.6: Overview of the Schedule clause in OpenMP in our samples. We find out that the most used scheduler is *static*. Note that some schedulers have multiple parameters; for clarity, these have been terminated with commas instead of the usual closing parenthesis.

Figure 5.6 shows the overall usage of schedule clauses. We have sorted these schedule clauses by frequency of their occurrence. We have also indicated the percentage of occurrence for better analysis. Schedule clauses were differentiated and analyzed in the following manner:

- $schedule(auto)$
- $schedule(runtime)$
- $schedule(guided)$
- $schedule(guided,$
- $schedule(dynamic)$
- $schedule(dynamic,$
- $schedule(static)$
- $schedule(static,$

This way we can also distinguish between the standard version $schedule(static)$ and the version where the programmer specifies a chunk size like $schedule(static, 10)$.

## 5.6 Usage of Functions by OpenMP Standard Versions

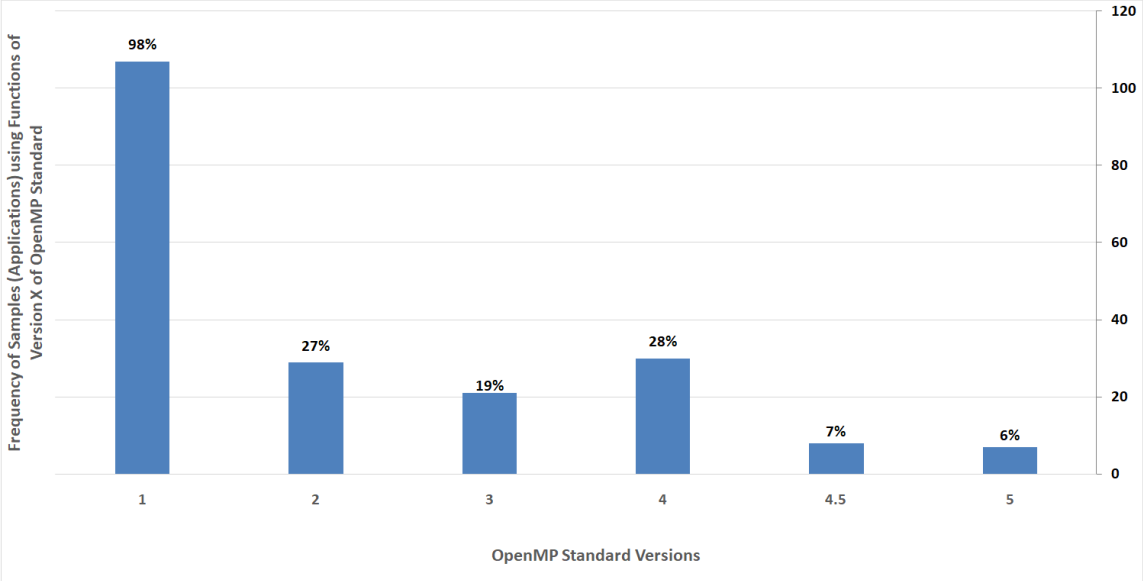


Figure 5.7: Application count vs the minimum version of OpenMP which provides the functionality required by each sample. Most samples need only early version 1 of OpenMP standard.

In figure 5.7 we count the occurrence of samples vs. the lowest version of OpenMP standard which provides the functionality which were required by each sample. We have sorted the OpenMP standards according to version number, adding percentage of occurrence for every version of OpenMP standard. Table 5.1 shows release year of each OpenMP version for C/C++ and Fortran.

Table 5.1: Release Year of OpenMP Versions

| Release Year | Release Year for C/C++ | Release Year for Fortran |
|---|---|---|
| 5 | 2019 | 2019 |
| 4.5 | 2015 | 2015 |
| 4 | 2013 | 2013 |
| 3 | 2008 | 2009 |
| 2 | 2002 | 2000 |
| 1 | 1998 | 1997 |

## 5.7 Usage of Features by oldest OpenMP Standard Versions

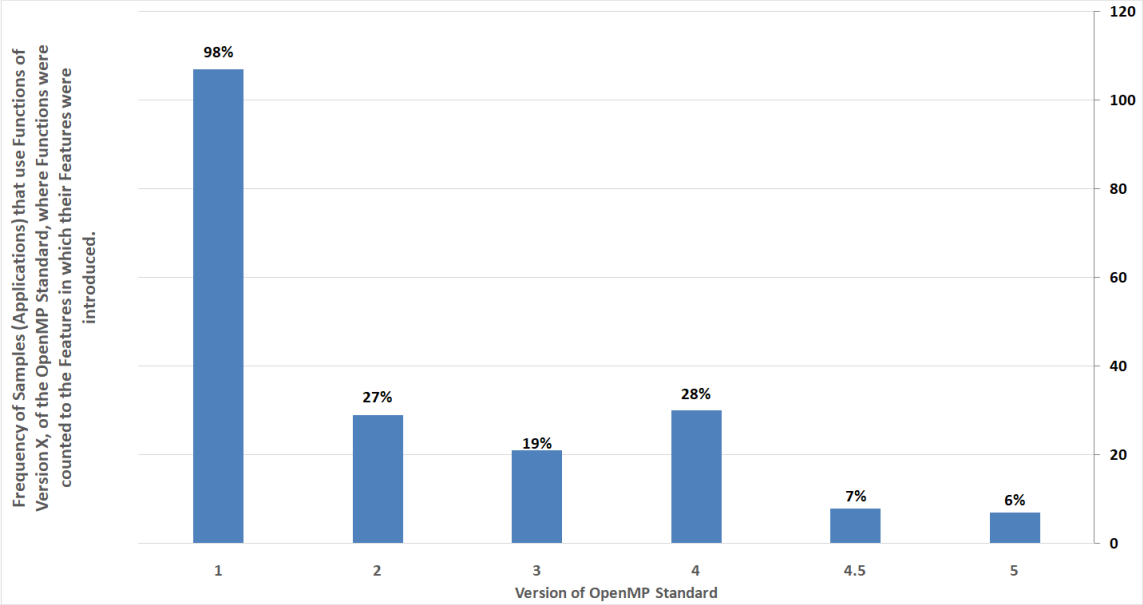


Figure 5.8: Frequency of samples (Application count) with regards to the oldest version of OpenMP which provides the functionality required by each sample. But this time, we count the functions of OpenMP Version to the OpenMP Version where the first member of the function's member (feature) were introduced. We find out, that most samples use early version 1 of OpenMP standard.

During a DMI group meeting, we were asked to look more closely at this issue with introduction of functions by every OpenMP version. So we created another graphical illustration (figure 5.8) where instead of introducing functionality we also look more closely at functionality by feature introduction criteria. This means that a feature only exists if at least one new function has been implemented in an OpenMP version. So if we now look at a specific function and check if it has been added to one of our samples, we only look at the oldest OpenMP version from which the corresponding feature exists.

Last but not least, we have sorted the OpenMP standards according to version number. We have also indicated the percentage of occurrence for better understanding.

## 5.8 Usage of Multi-Threaded Programming Models

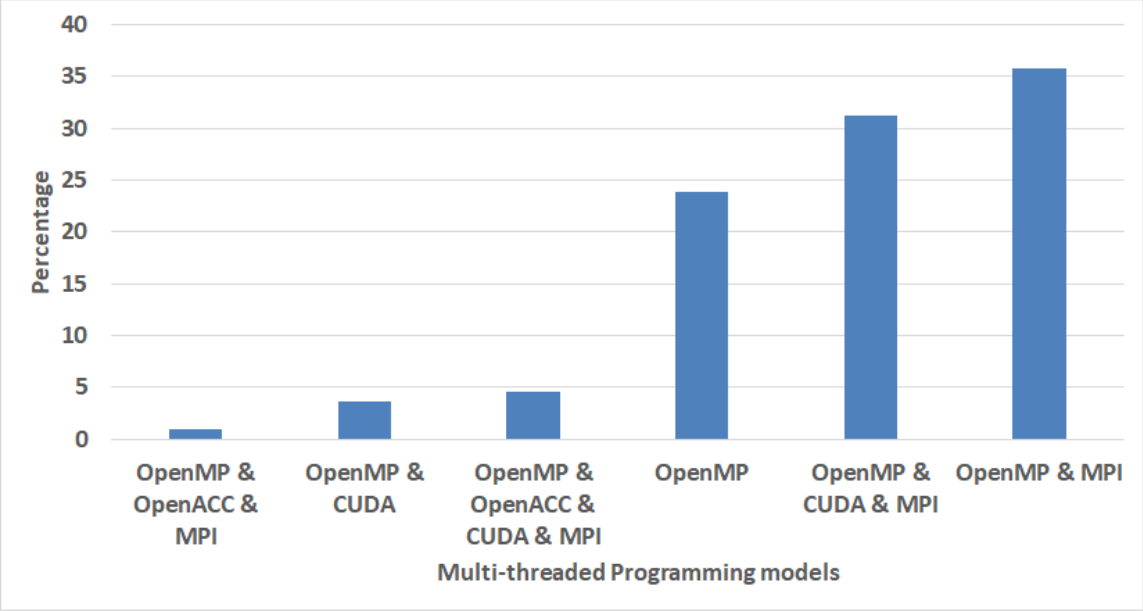


Figure 5.9: Percentage of total samples using multi-threaded programming models. The most often used model is OpenMP+MPI.

In figure 5.9, we show the percentage of total samples using various OpenMP+X multi-threaded programming models, where X represents a placeholder for one or more multi-threaded programming models in conjunction with OpenMP. Therefore, figure 5.9 shows whether the provided samples use OpenCL, OpenACC, MPI, and/or CUDA constructs. We do not consider the frequency of use of these constructs in a given sample, but only if they occur at least once in a sample. If none of the programming models are present in the sample, the result is evaluated as None. Combinations which were not used are not shown in the graphic. In conclusion, we have checked all combinations of mixed use of the programming models, such as OpenMP in combination with MPI. Finally, we have sorted these programming models by frequency of occurrence. We have also indicated the percentage of occurrence for better analysis.

## 5.9 Usage of Programming Languages

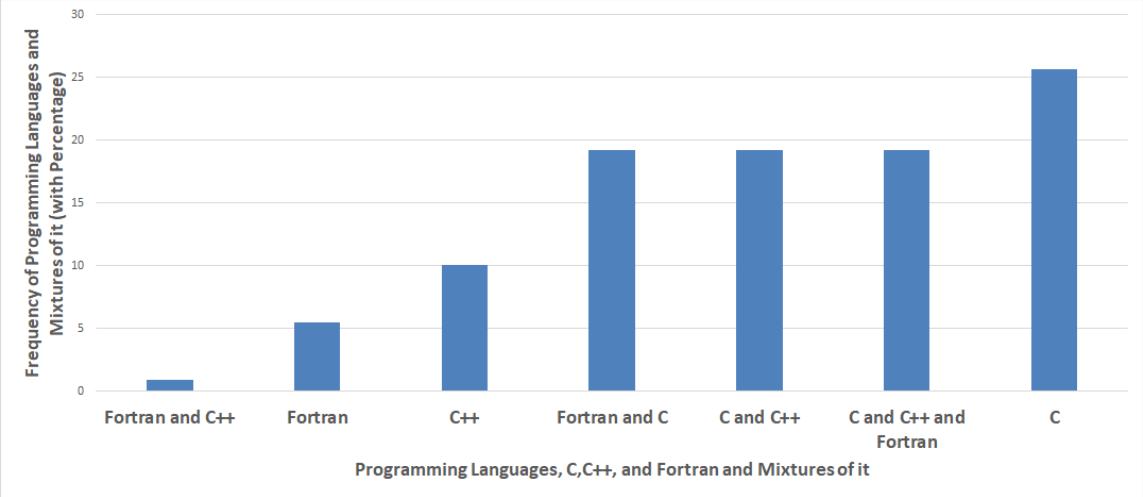


Figure 5.10: The frequency of samples (applications) with percentage using C, C++, and Fortran. Most samples use a mixture of languages (58%).

Figure 5.10 shows the frequency and percentage of samples whose source codes were written exclusively in C, C++ and Fortran or a mixture of these programming languages.

## 5.10 Usage of Programming Languages vs. Sample Code Size (Part I - Size by Lines of Code)

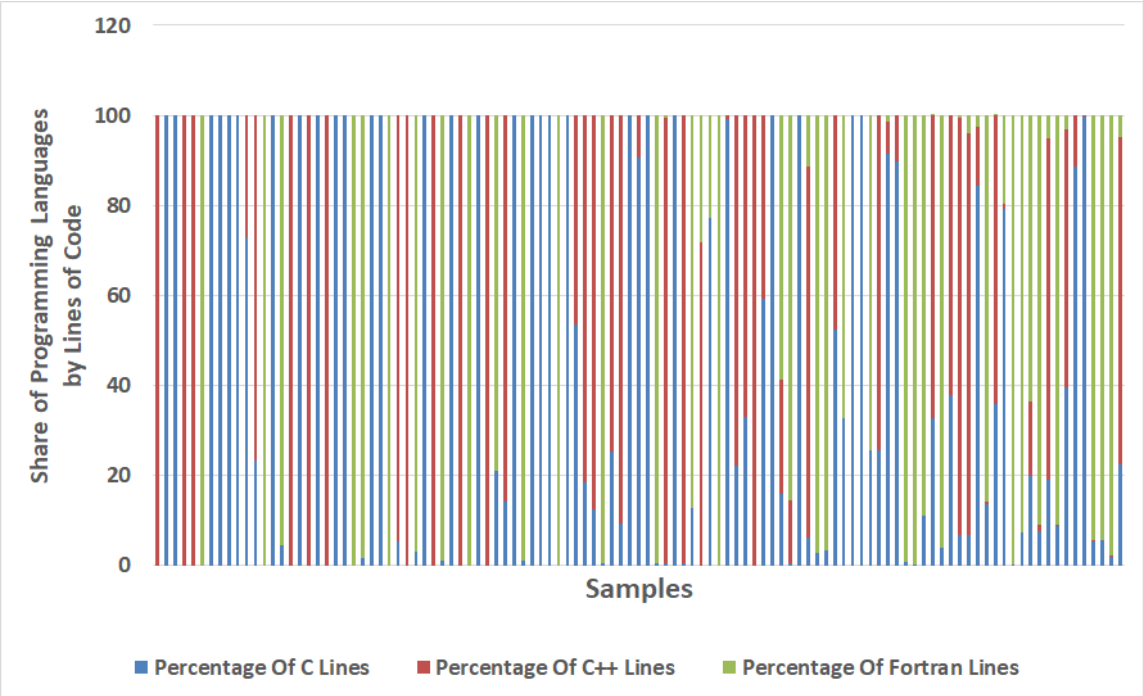


Figure 5.11: Usage of programming language percentage versus sample code size. If we consider the sample code size by lines of code, the most used programming language for an OpenMP application is C.

Figure 5.11 shows the percentage of programming languages of a sample ordered in respect to the code base size (lines of code) from left small size to right big size. Usually an application consists of several files written in different programming languages and is written in different major programming languages for HPC, mainly C, C++, Fortran, and a combination between them.

For better view, see figure 5.12, which is an enlarged plot of figure 5.11.

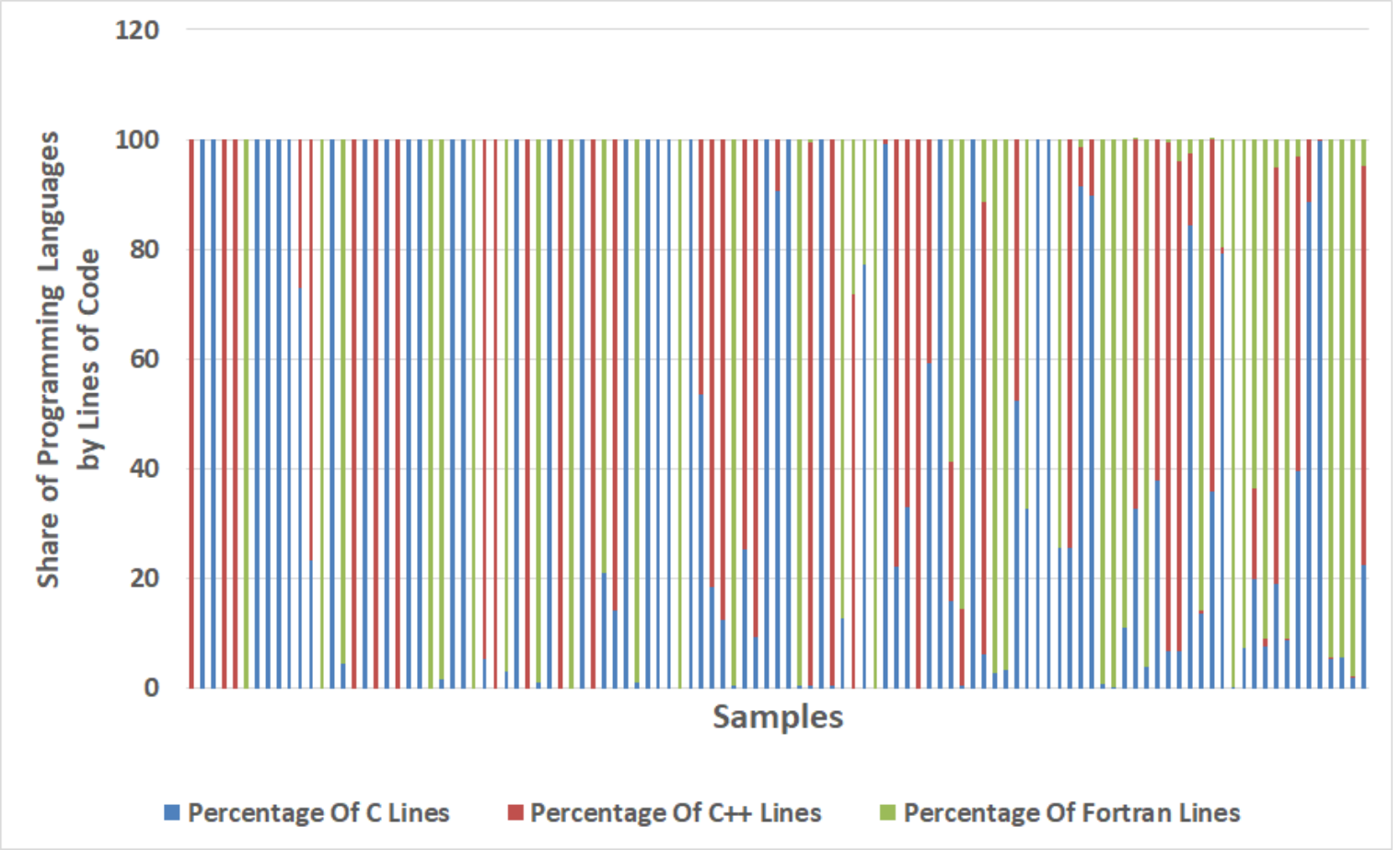


Figure 5.12: Usage of programming language percentage versus sample code size. If we consider the sample code size by lines of code, the most used programming language for an OpenMP application is C.

## 5.11 Usage of Programming Languages vs. Sample Code Size (Part II - Size by Files count)


Figure 5.13: Usage of programming language percentage versus sample code size (size by files count). Most used programming language with OpenMP is a combination between C/C++ and Fortran.

Figure 5.13 shows the percentage of programming languages of a sample ordered in respect to the code base size (number of files). An application is usually made up of several files written in different programming languages. These programming languages are C, C++ and Fortran, among many others. Here, we ignore the programming languages that are not explicitly mentioned among the first three. For example, let's have a closer look at the most left application on the X-axis: This application consists 100% only of Fortran files.

For better view, see figure 5.14, which is an enlarged plot of figure 5.13.


Figure 5.14: Usage of programming language percentage versus sample code size (size by files count). Most used programming language with OpenMP is a combination between C/C++ and Fortran.

## 5.12 Top 5 OpenMp Commands

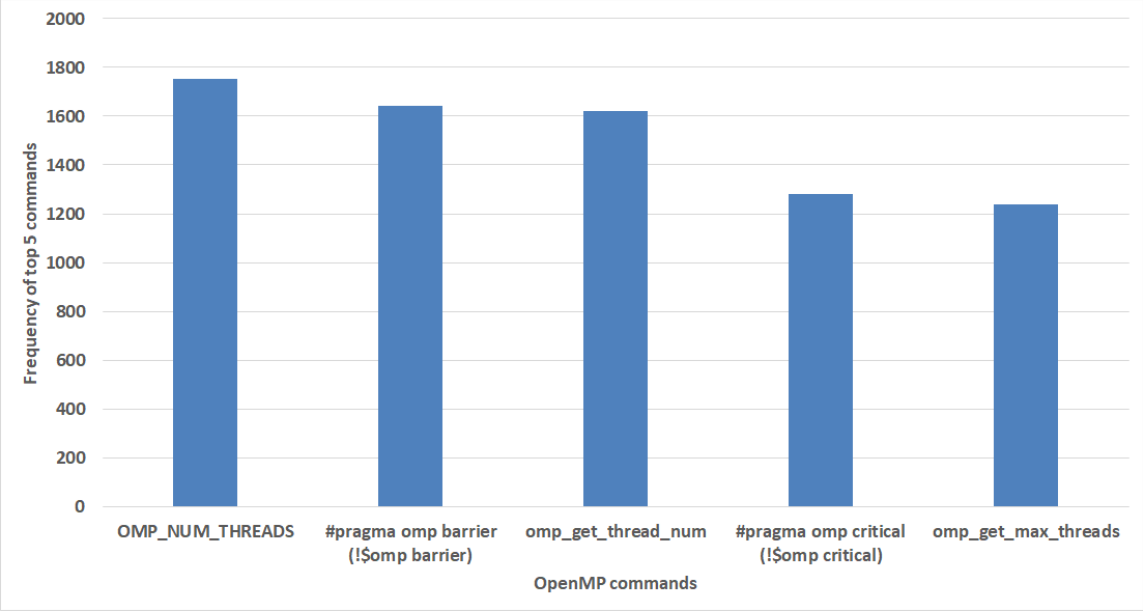


Figure 5.15: Top 5 OpenMP commands of all samples together. Most used OpenMP command is $OMP\_NUM\_THREADS$.

Figure 5.15 shows the top 5 OpenMP overall commands ordered by frequency.

# Chapter 6

# Discussion

In the following sections we will discuss the results about the plots in chapter 5.

## 6.1 Unique OpenMP Features

Figure 5.1 shows the overall usage of unique OpenMP features which are defined according to OpenMP standards. The figure shows that almost half of the provided features by OpenMP standards are not used at all (e.g. *Cancellation constructs*, *Distribute loop constructs* and many more). We can also state that the percentage of samples using *Execution environment routines* is over 80%. Furthermore we can see that less than half of the samples use only the top 2 OpenMP features – these are *Synchronization constructs* beside the before mentioned *Execution environment routines*. A lot of features are rarely used in our provided samples: These features are for example the $Worksharing - Loop\ SIMD$ or *Combined constructs* and many more. But note, that constructs like *Loop Constructs* and *Tasking Constructs*"" being at zero does not necessarily mean that loops and tasks are not used; they might be used in combination with other constructs and thus be counted inside *Combined Constructs*. In figures 5.4, and 5.5, we will take a focus on the top 2 OpenMP features: *Execution environment routines* and *Synchronization constructs*.

## 6.2 Usage of Unique OpenMP Features by Applications

Figure 5.2 shows the usage of OpenMP features by each samples. Different samples use different sets of given OpenMP features. Not a simple one of these samples use all features provided by the OpenMP standard. We can also state, that from our samples, all of them actually use at least two OpenMP features. Our figure also shows that more than two third of our samples use more than 15 provided OpenMP features. Therefore the majority of samples uses a lot of OpenMP features provided by the OpenMP standard.

But we also found out that many features of OpenMP standards are seldom used in HPC applications. Thus, Vendors, centers and developers should not focus their optimization efforts on these features, because they are not widely employed by OpenMP users. It's better to focus their efforts on more widely used features, which we also identified in figure 5.2. The OpenMP standardization

body and researchers involved with the development and maintenance of the OpenMP standard should be aware of the popularity of each feature, because that is important feedback from the community as to which features actually translate well into scientific applications. HPC researchers trying to optimize applications, can identify specific features (e.g. the schedule clause) to focus their optimization efforts and also identify features to stay away from because they are only used by a small minority of applications.

## 6.3 Usage of Execution Environment Routines

In figure 5.4 we find out that only nine functions of the first placed OpenMP feature are used in 10 or more distinct samples which were provided to us. The top 3 functions of the *Execution environment routines* are:

- *omp_get_thread_num*,
- *omp_get_max_threads* and
- *omp_get_num_threads*.

Furthermore, we find out that over 50 of our samples use the top 3 functions of *Execution environment routines*. We also know that *omp_get_thread_num*, *omp_get_max_threads*, etc. is some of the most basic quality-of-life OpenMP functionality that developers use in their code, hence the popularity.

## 6.4 Usage of Synchronization Constructs

In figure 5.5 we find out that five constructs of the second ranked OpenMP feature (*Synchronization constructs*) are used in more than five distinct samples which were provided to us. The top 3 constructs of the *Synchronization constructs* are:

- *critical*,
- *barrier* and
- *atomic*.

From our point of view, we know that *critical*, *barrier* and *atomic* are very basic methods to ensure synchronization and to avoid race conditions. This explains why these three functions are so popular among OpenMP users.

## 6.5 Usage of Schedule Clauses

OpenMP, at its core, is designed to distribute the work done in "loops" across threads so that the "loop" can be done faster. In general, loop iterations do not always have the same amount of work, so some iterations might take longer than others.

The schedule clause determines how the loop iterations are distributed:

- does each thread get the same number of iterations *schedule*(*static*)?
- do we give the threads only new iterations when they are done with the old ones *schedule*(*dynamic*)?
- do we give an increasing or decreasing number of iterations to the threads *schedule*(*guided*)?
- etc.

So a schedule clause can make a program faster or slower, depending on whether the right schedule clause is used. That's why these schedule clauses are so interesting, to see if we can make a program faster if we use the right schedule clauses.

In Figure 5.6, we find out, that the *static* scheduler with no parameter is used by exact 35 samples which were provided to us. The next most used scheduler is *dynamic* with further parameter which is used by more than 25 samples. The *static* scheduler with further parameter and the *dynamic* scheduler with no parameter are used by over 20 examples and other scheduler were used by less than 11 samples.

## 6.6 Usage of Functions by OpenMP Standard Versions

In figure 5.7 we count the occurrence of samples vs. the minimum version of OpenMP standard which provides the functionality which were required by each sample. While new features and functionalities of OpenMP are added to existing one, the idea behind that is of course to improve performance and code readability. To our surprise, a large set of our samples do not use these newer functionalities like in the MPI paper of Laguna et. al. The most used version in MPI paper is 1.0 [3]. It would be certainly interesting to know why developers are not adopting newer features of OpenMP standard to write better and more readable code. However, we believe this could also be due to the fact that many of the examined samples are older than the release year of more recent OpenMP versions. Unfortunately, we do not have information about the age of the sample. Normally, newer applications tend to use newer features (i.e. from standard version 5). We also state that more than half of our samples rely only on the provided functionalities of OpenMP version 1.0. And last but not least, we think that minor updates (OpenMP versions 3.1 and 4.5) do not represent any progress or relief for most developers when developing their application since they are used very rarely. The same is true for the last major releases (OpenMP Version 5.0).

## 6.7 Usage of Features by oldest OpenMP Standard Versions

The idea behind figure 5.8 is to look more closely at functionality by feature introduction criteria. That means that a feature only exists if at least one new function has been implemented in a OpenMP version. So if we now look at a specific function and check if it has been added to one of our samples, we only look at the oldest OpenMP version from which the corresponding feature exists.

To our surprise, a large set of our samples do not use these newer features by introduction criteria. We can state that most samples use early functions introduced by version 1 of OpenMP standard. We also find out, that in OpenMP Version 3.1 no new features were introduced by this introduction criteria.

## 6.8 Usage of Multi-Threaded Programming Models

In figure 5.9, we show the percentage of total samples using various OpenMP+X multi-threaded programming models, where X represents a placeholder for one or more multi-threaded programming models in conjunction with OpenMP.

We can state that 77% of the provided sample use some form of hybrid code (i.e. OpenMP+X), almost half (40%) of which have been constructed from only 2 multi-threaded programming models, for example OpenMP in combination with MPI, which is also the most widely used multi-threaded programming model found in a third (36%) of the provided samples. In contrast, we have no provided samples belonging to the category "None". Then we find out, that a fourth of the provided samples use OpenMP exclusively with no other multi-threaded programming models. Furthermore, we find out that OpenCL, OpenACC, CUDA, and MPI are not used exclusively in any of the provided samples.

## 6.9 Usage of Programming Languages

In figure 5.10 we find out that the least used programming language out of the trio is Fortran with 6% share of all provided samples. By far, the most commonly used programming language is C (26%), followed by C++ (10%). We also see that - with the exception of Fortran & C++ - all mixtures have a share of 19%. The least used mixture of these languages with only 1% share is Fortran & C.

## 6.10 Usage of Programming Languages vs. Sample Code Size (Part I - Size by Lines of Code)

Figure 5.11 shows the percentage of programming languages of a sample ordered in respect to the code base size by its lines of code. Usually, a provided sample (application) consists of several files written in specific programming languages. These programming languages are C, C++ and Fortran, among many others. Here, we ignore all the programming languages that are not explicitly mentioned among the first three. For example, the first application on the X-axis is very small and is written purely in C++.

We do this plot, because we want to see what programming languages are the most used in OpenMP applications according to their size. In figure 5.11 we see a trend of how pure Fortran is used less and less as the sample size gets larger. As an application increases in its size, usage of combinations between C/C++ and Fortran is the predominant one.

## 6.11 Usage of Programming Languages vs. Sample Code Size (Part II - Size by Files count)

Figure 5.13 shows the percentage of programming languages of a sample ordered in respect to the code base size by file counts. An application is usually made up of several files written in different programming languages. These programming languages are C, C++ and Fortran, among many

others. Here, we ignore the programming languages that are not explicitly mentioned among the first three.

In figure 5.13 we see a trend of how pure Fortran is used less and less as the counts of files of a sample (application) gets bigger. As the file counts increases, usage of combinations between C, C++ and Fortran is the predominant one.

## 6.12 Top 5 OpenMp Commands

Figure 5.15 shows the top 5 OpenMP overall commands ordered by frequency. The top OpenMP commands of all samples together is $OMP\_NUM\_THREADS$. Furthermore, the most used OpenMP directives (starting with *#pragma omp*)[1] are $for$, *critical*, and *barrier* (and all of these directives were already implemented and standardized in OpenMP version 1.0).

[1] in Fortran it starts with !$*omp*

# Chapter 7

# Conclusion

This chapter provides the conclusion and proposes for future work. The contribution of this thesis is a large-scale study on source code of OpenMP applications to understand the state-of-the-practice in parallel programming with OpenMP.

Using custom Python scripts we were able to extract and understand the characteristics of OpenMP usage in terms of the most commonly used features and functionalities in the source code of HPC applications. The plots presented in this thesis allow for a representative and comparative look into how OpenMP functionalities are used across applications.

## 7.1 Main Insights

In the following section we will summarize the main findings of our work:

1. Most used functionalities provided by OpenMP feature *execution environment routines* are used by 90% of all applications, and the most used routine from this group is *omp_get_thread_num* which is also the most used functionality overall.

2. The most used OpenMP directives (starting with *#pragma omp*)[1] are *for*, *critical*, and *barrier* (and all of these directives were already implemented and standardized in OpenMP version 1.0).

3. The synchronization construct *critical* is present in more applications compared to *barrier* or *atomic*, synchronization constructs regarding to *tasking* are used the least.

4. The most used OpenMP scheduling clause is *static* with default chunk size, followed by *dynamic* with custom chunk size, the least used clause is *guided* with custom chunk size.

5. The most used combination of programming paradigms is [OpenMP + MPI], followed by [OpenMP + MPI + CUDA], these combinations are used more often compared to pure OpenMP.

[1] in Fortran it starts with !$*omp*

6. C is the most used programming language, followed by [C & Fortran], [C & C++], and [C & C++ & Fortran]. The least used combination of programming languages is [Fortran & C++]. And finally, we can state that all of these features were already implemented and standardized in Version 1.0 of OpenMP.

Beyond the main insights summarized above, there are many more potential insights to be gained by looking at the source code statistics that we collected. For example, we also collected how many OpenMP *schedule clauses* are used for each application, and where in the source code they are (file and line of code). This information can directly guide HPC researchers to the specific location in source code, where potential scheduling or load balancing changes can be implemented.

## 7.2 Future Work

This section will introduce potential future work that can extend and/or complement the work presented in this thesis. Future work ranges from extending the application pool, extending the programming paradigm coverage, to including more source code metrics. The specific future work opportunities are listed below:

- Our script could be applied to a larger and more up-to-date collection of OpenMP applications, to gain more representative insights that also reflect the latest usage of OpenMP features.
- The script that we developed also collects the individual functions of CUDA into a XLSX file. Although these lists of features are currently not further processed, the script can be easily adjusted to create similar plots, as presented in this thesis, also for CUDA applications.
- Another interesting continuation of this project is to correlate OpenMP, MPI, CUDA, etc. usage, to see whether specific OpenMP functionality is more used with MPI as compared to CUDA.
- Since our approach is based on source code, it can also be extended to include other non-programming paradigm specific features, such as source code complexity.

# Bibliography


[1] Ashwini M. Bhugul, International Journal of Computer Science and Mobile Computing, Vol.6 Issue.2, February 2017, pg. 90-94.

[2] David E. Bernholdt, Swen Boehm, George Bosilca, Manjunath GorentlaVenkata, Ryan E. Grant, Thomas Naughton, Howard P. Pritchard, Martin Schulz, and Geoffroy R. Vallee. 2017. A survey of MPI usage in the US exascale computing project. Concurrency and Computation: Practice and Experience (2017), e4851.

[3] Laguna, I., Marshall, R., Mohror, K., Ruefenacht, M., Skjellum, A., & Sultana, N. (2019, November). A large-scale study of MPI usage in open-source HPC applications. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (pp. 1-14).

[4] Nawrin Sultana, Anthony Skjellum, Purushotham Bangalore, Ignacio Laguna, and Kathryn Mohror. 2018. Understanding the Usage of MPI in Exascale Proxy Applications. Workshop on Exascale MPI (ExaMPI). (2018).

[5] OpenMP - Application Programming Interface. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf, Version 5.0 November 2018.

[6] OpenMP - The OpenMP API specification for parallel programming. https://www.openmp.org/, Date: August 2019.

[7] Parallel Programming - Multithreading (OpenMP). https://wvuhpc.github.io/2018-Lesson_4/03-openmp/index.html, Date: August 2019.

[8] Steven P. VanderWiel, Daphna Nathanson, and David J. Lilja. 1997. "Complexity and performance in parallel programming languages. In Proceedings Second International Workshop on High-Level Parallel Programming Models and Supportive Environments". IEEE, 3–12.

[9] Ying hao Xu Lin, "Porting and tuning of the Mont-Blanc benchmarks to the multicore ARM 64 bit architecture". Universitat Politècnica de Catalunya, Computer Engineering, FACULTAT D'INFORMÀTICA DE BARCELONA (FIB). 2016.

# Appendices

# Appendix A

# OpenMP Features and Commands

The following descriptions are taken from the OpenMP standard 5.0 [6].

- **feature 1**: a small description about feature 1
  - **functionality 1 in C (and F)**: a small description about functionality 1 of feature 1
  - **functionality 2 in C (and F)**: a small description about functionality 2 of feature 1
- **feature 2**: a small description about feature 2
  - **functionality 1 in C (and F)**: a small description about functionality 1 of feature 2
  - **functionality 2 in C (and F)**: a small description about functionality 2 of feature 2

Now let's look at OpenMP features and their commands, each sorted alphabetically:

- **Cancellation constructs**: The cancel construct activates cancellation of the innermost enclosing region of the type specified.
  - **#pragma omp cancel (!$omp cancel)**: "Requests cancellation of the innermost enclosing region of the type specified."
  - **#pragma omp cancellation point (!$omp cancellation point)**: "Introduces a user-defined cancellation point at which tasks check if cancellation of the innermost enclosing region of the type specified has been activated."
- **Combined constructs**: Combined constructs are shortcuts for specifying one construct immediately nested inside another construct containing a worksharing-loop construct with one or more associated loops.
  - **#pragma omp master taskloop (!$omp master taskloop)**: "Shortcut for specifying a master construct containing a taskloop construct and no other statements."

- **#pragma omp master taskloop simd (!$omp master taskloop simd)**: "Shortcut for specifying a master construct containing a taskloop simd construct and no other statements."
- **#pragma omp parallel for simd (!$omp parallel do simd)**: "Shortcut for specifying a parallel construct containing only one worksharing-loop SIMD construct."
- **#pragma omp parallel loop (!$omp parallel loop)**: "Shortcut for specifying a parallel construct containing a loop construct with one or more associated loops and no other statements."
- **#pragma omp parallel master (!$omp parallel master)**: "Shortcut for specifying a parallel construct containing a master construct and no other statements."
- **#pragma omp parallel master taskloop (!$omp parallel master taskloop)**: "Shortcut for specifying a parallel construct containing a master taskloop construct and no other statements."
- **#pragma omp parallel master taskloop simd (!$omp parallel master taskloop simd)**: "Shortcut for specifying a parallel construct containing a master taskloop simd construct and no other statements"
- **#pragma omp parallel sections (!$omp parallel sections)**: "Shortcut for specifying a parallel construct containing a sections construct and no other statements."
- **#pragma omp target parallel (!$omp target parallel)**: "Shortcut for specifying a target construct containing a parallel construct and no other statements."
- **#pragma omp target parallel for (!$omp target parallel do)**: "Shortcut for specifying a target construct with a parallel worksharing loop construct and no other statements."
- **#pragma omp target parallel for simd (!$omp target parallel do simd)**: Shortcut for specifying a target construct with a parallel worksharing-loop SIMD construct and no other statements.
- **#pragma omp target parallel loop (!$omp target parallel loop)**: "Shortcut for specifying a target construct containing a parallel loop construct and no other statements."
- **#pragma omp target simd (!$omp target simd)**: "Shortcut for specifying a target construct containing a simd construct and no other statements"
- **#pragma omp target teams (!$omp target teams)**: "Shortcut for specifying a target construct containing a teams construct and no other statements."
- **#pragma omp target teams distribute (!$omp target teams distribute)**: "Shortcut for specifying a target construct containing a teams distribute construct and no other statements."
- **#pragma omp target teams distribute parallel for (!$omp target teams distribute parallel do)**: "Shortcut for specifying a target construct containing a teams distribute parallel worksharing loop construct and no other statements."
- **#pragma omp target teams distribute parallel for simd (!$omp target teams distribute parallel do simd)**: "The omp distribute parallel for simd construct is a composite construct. Clause can be any of the clauses that are accepted by the omp distribute or omp parallel for simd directive with identical meanings and restrictions."

- **#pragma omp target teams distribute simd (!$omp target teams distribute simd)**: "Shortcut for specifying a target construct containing a teams distribute simd construct and no other statements."
- **#pragma omp target teams loop (!$omp target teams loop)**: "Shortcut for specifying a target construct containing a teams loop construct and no other statements."
- **#pragma omp teams distribute (!$omp teams distribute)**: "Shortcut for specifying a teams construct containing a distribute construct and no other statements."
- **#pragma omp teams distribute parallel for (!$omp teams distribute parallel do)**: "Shortcut for specifying a teams construct containing a distribute parallel worksharing loop construct and no other statements."
- **#pragma omp teams distribute parallel for simd (!$omp teams distribute parallel do simd)**: "Shortcut for specifying a teams construct containing a distribute parallel work-sharing-loop SIMD construct and no other statements."
- **#pragma omp teams distribute simd (!$omp teams distribute simd)**: "Shortcut for specifying a teams construct containing a distribute simd construct and no other statements."
- **#pragma omp teams loop (!$omp teams loop)**: "Shortcut for specifying a teams construct containing a loop construct and no other statements."
- **!$omp parallel workshare**[1]: "Shortcut for specifying a parallel construct containing a workshare construct and no other statements."

- **Data environment directives**: This one specifies that variables are replicated, with each thread having its own copy.
  - **#pragma omp declare mapper (!$omp declare mapper)**: "Declares a user-defined mapper for a given type, and may define a mapper-identifier for use in a map clause."
  - **#pragma omp declare reduction (!$omp declare reduction)**: "Declares a reduction-identifier that can be used in a reduction clause."
  - **#pragma omp threadprivate (!$omp threadprivate)**: "Specifies that variables are replicated, with each thread having its own copy. Each copy of a threadprivate variable is initialized once prior to the first reference to that copy."

- **Device directives and construct**: This one creates a device data environment for the extent of the region.
  - **#pragma omp declare target (!$omp declare target)**: "A declarative directive that specifies that variables, functions, and subroutines are mapped to a device."
  - **#pragma omp target (!$omp target)**: "Map variables to a device data environment and execute the construct on that device"
  - **#pragma omp target data (!$omp target data)**: "Creates a device data environment for te extent of the region."
  - **#pragma omp target enter data (!$omp target enter data)**: Maps variables to a device data environment.

[1] no C command available

  - **#pragma omp target exit data (!$omp target exit data)**: Unmaps variables from a device data environment.
  - **#pragma omp target update (!$omp target update)**: "Makes the corresponding list items in the device data environment consistent with their original list items, according to the specified motion clauses."

- **Device memory routines**: This one describes routines that support allocation of memory and management of pointers in the data environments of target devices.
  - **omp_target_alloc**: Allocates memory in a device data environment.
  - **omp_target_associate_ptr**: "Maps storage to which a device pointer points to storage to which a host pointer points. The device pointer may be the result of a call to omp_target_alloc or have been obtained from implementation-defined runtime routines."
  - **omp_target_disassociate_ptr**: "Removes the association between a host pointer and a device address on a given device."
  - **omp_target_free**: "Frees the device memory allocated by the omp_target_alloc routine."
  - **omp_target_is_present**: "Validates whether a host pointer has an associated device buffer on a given device."
  - **omp_target_memcpy**: "Copies memory between any combination of host and device pointers."
  - **omp_target_memcpy_rect**: "Copies a rectangular subvolume from a multi-dimensional array to another multi-dimensional array."

- **Distribute loop constructs**: These construct specifies the iterations of one or more loops that will be executed by the initial teams in the context of their implicit tasks.
  - **#pragma omp distribute (!$omp distribute)**: Specifies loops which are executed by the initial teams.
  - **#pragma omp distribute parallel for (!$omp distribute parallel do)**: "These constructs specify a loop that can be executed in parallel by multiple threads that are members of multiple teams."
  - **#pragma omp distribute parallel for simd (!$omp distribute parallel do simd)**: "Specifies a loop that can be executed concurrently using SIMD instructions in parallel by multiple threads that are members of multiple teams."
  - **#pragma omp distribute simd (!$omp distribute simd)**: "Specifies loops which are executed concurrently using SIMD instructions and the initial teams."

- **Environment Variables**: These variables specify the settings of the ICVs (Internal Control Variables) that affect the execution of OpenMP programs.
  - **OMP_AFFINITY_FORMAT**: "Sets the initial value of the affinity-format-var ICV defining the format when displaying OpenMP thread affinity information. The format is a character string that may contain as substrings one or more field specifiers, in addition to other characters. The format of each field specifier is: %[[[0].] size ] type, where the field type may be either the short or long names"

- **OMP_ALLOCATOR**: "Sets the def-allocator-var ICV that specifies the default allocator for allocation calls, directives, and clauses that do not specify an allocator. arg is a case-insensitive, predefined allocator. OpenMP memory allocators can be used to make allocation requests. The behavior of the allocation process can be affected by the allocator traits specified."
- **OMP_CANCELLATION**: "Sets the cancel-var ICV. var may be true or false. If true, the effects of the cancel construct and of cancellation points are enabled and cancellation is activated."
- **omp_cgroup_mem_alloc**: "Sets the def-allocator-var ICV that specifies the default allocator for allocation calls, directives, and clauses that do not specify an allocator. arg is a case-insensitive, predefined allocator. OpenMP memory allocators can be used to make allocation requests. The behavior of the allocation process can be affected by the allocator traits specified."
- **omp_const_mem_alloc**: "Sets the def-allocator-var ICV that specifies the default allocator for allocation calls, directives, and clauses that do not specify an allocator. arg is a case-insensitive, predefined allocator. OpenMP memory allocators can be used to make allocation requests. The behavior of the allocation process can be affected by the allocator traits specified."
- **OMP_DEBUG**: "Sets the debug-var ICV. var may be enabled or disabled. If enabled, the OpenMP implementation will collect additional runtime information to be provided to a third_x0002_party tool. If disabled, only reduced functionality might be available in the debugger."
- **OMP_DEFAULT_DEVICE**: "Sets the default-device-var ICV that controls the default device number to use in device constructs."
- **omp_default_mem_alloc**: "Sets the def-allocator-var ICV that specifies the default allocator for allocation calls, directives, and clauses that do not specify an allocator. arg is a case-insensitive, predefined allocator. OpenMP memory allocators can be used to make allocation requests. The behavior of the allocation process can be affected by the allocator traits specified."
- **OMP_DISPLAY_ENV**: "If var is TRUE, instructs the runtime to display the OpenMP version number and the value of the ICVs associated with the environment variables as name=value pairs. If var is VERBOSE, the runtime may also display vendor-specific variables. If var is FALSE, no information is displayed."
- **OMP_DYNAMIC**: "Sets the dyn-var ICV. var may be TRUE or FALSE. If TRUE, the implementation may dynamically adjust the number of threads to use for executing parallel regions."
- **omp_high_bw_mem_alloc**: "Sets the def-allocator-var ICV that specifies the default allocator for allocation calls, directives, and clauses that do not specify an allocator. arg is a case-insensitive, predefined allocator. OpenMP memory allocators can be used to make allocation requests. The behavior of the allocation process can be affected by the allocator traits specified."
- **omp_large_cap_mem_alloc**: "Sets the def-allocator-var ICV that specifies the default allocator for allocation calls, directives, and clauses that do not specify an allocator. arg

is a case-insensitive, predefined allocator. OpenMP memory allocators can be used to make allocation requests. The behavior of the allocation process can be affected by the allocator traits specified."

- **omp_low_lat_mem_alloc**: "Sets the def-allocator-var ICV that specifies the default allocator for allocation calls, directives, and clauses that do not specify an allocator. arg is a case-insensitive, predefined allocator. OpenMP memory allocators can be used to make allocation requests. The behavior of the allocation process can be affected by the allocator traits specified."
- **OMP_MAX_ACTIVE_LEVELS**: "Sets the max-active-levels-var ICV that controls the maximum number of nested active parallel regions."
- **OMP_MAX_TASK_PRIORITY**: "Sets the max-task-priority-var ICV that controls the use of task priorities."
- **OMP_NESTED**: Controls nested parallelism with max-active-levels-var ICV.
- **OMP_NUM_THREADS**: "Sets the nthreads-var ICV for the number of threads to use for parallel regions."
- **OMP_PLACES**: "Sets the place-partition-var ICV that defines the OpenMP places available to the execution environment. places is an abstract name (threads, cores, sockets, or imple_x0002_mentation-defined) or a list of non-negative numbers"
- **OMP_PROC_BIND**: "Sets the value of the global bind-var ICV, setting the thread affinity policy to use for parallel regions at the corresponding nested level. policy can be the values true, false, or a comma-separated list of master, close, or spread in quotes."
- **omp_pteam_mem_alloc**: "Sets the def-allocator-var ICV that specifies the default allocator for allocation calls, directives, and clauses that do not specify an allocator. arg is a case-insensitive, predefined allocator. OpenMP memory allocators can be used to make allocation requests. The behavior of the allocation process can be affected by the allocator traits specified."
- **OMP_SCHEDULE**: "Sets the run-sched-var ICV for the runtime schedule kind and chunk size. modifier is one of monotonic or nonmonotonic; kind is one of static, dynamic, guided, or auto."
- **OMP_STACKSIZE**: "Sets the stacksize-var ICV that specifies the size of the stack for threads created by the OpenMP imple_x0002_mentation. size is a positive integer that specifies stack size. B is bytes, K is kilobytes, M is megabytes, and G is gigabytes. If unit is not specified, size is measured in K."
- **OMP_TARGET_OFFLOAD**: "Sets the initial value of the target-offload-var ICV. arg must be one of MANDATORY, DISABLED, or DEFAULT."
- **OMP_THREAD_LIMIT**: "Sets the thread-limit-var ICV that controls the number of threads participating in the OpenMP program."
- **omp_thread_mem_alloc**: "Sets the def-allocator-var ICV that specifies the default allocator for allocation calls, directives, and clauses that do not specify an allocator. arg is a case-insensitive, predefined allocator. OpenMP memory allocators can be used to make allocation requests. The behavior of the allocation process can be affected by the allocator traits specified."

- **OMP_TOOL**: "Sets the tool-var ICV. If disabled, no first-party tool will be loaded nor initialized. If enabled the OpenMP implementation will try to find and activate a first-party tool."
- **OMP_TOOL_LIBRARIES**: "Sets the tool-libraries-var ICV to a list of tool libraries that will be considered for use on a device where an OpenMP implementation is being initialized. library-list is a list of dynamically-linked libraries, each specified by an absolute path."
- **OMP_WAIT_POLICY**: "Sets the wait-policy-var ICV that provides a hint to an OpenMP implementation about the desired behavior of waiting threads. Valid values for policy are ACTIVE (waiting threads consume processor cycles while waiting) and PASSIVE."

- **Event routine**: This one describes a routine that supports OpenMP event objects.
  - **omp_fulfill_event**: Fulfills and destroys an OpenMP event.

- **Execution environment routines**: hese are routines that affect and monitor threads, processors, and the parallel environment.
  - **omp_capture_affinity**: "Prints the OpenMP thread affinity information into a buffer using the format specification provided."
  - **omp_display_affinity**: "Prints the OpenMP thread affinity information using the format specification provided."
  - **omp_get_active_level**: "Returns the value of the active-level-vars ICV for the current device, which is the number of active, nested parallel regions on the device enclosing the task containing the call."
  - **omp_get_affinity_format**: "Returns the value of the affinity-format-var ICV on the device."
  - **omp_get_ancestor_thread_num**: "Returns, for a given nested level of the current thread, the thread number of the ancestor of the current thread."
  - **omp_get_cancellation**: "Returns the value of the cancel-var ICV, which is true if cancellation is activated; otherwise it returns false."
  - **omp_get_default_device**: "Returns the value of the default-device-var ICV, which determines the default target device."
  - **omp_get_device_num**: "Returns the device number of the device on which the calling thread is executing."
  - **omp_get_dynamic**: "This routine returns the value of the dyn-var ICV, which is true if dynamic adjustment of the number of threads is enabled for the current task."
  - **omp_get_initial_device**: Returns a device number representing the host device.
  - **omp_get_level**: "Returns the value of the levels-var ICV for the current device, which is the number of nested parallel regions on the device that enclose the task containing the call."
  - **omp_get_max_active_levels**: "Returns the value of max-active-levels-var ICV, which determines the maximum number of nested active parallel regions."

- **omp_get_max_task_priority**: "Returns the maximum value that can be specified in the priority clause."
- **omp_get_max_threads**: "Returns an upper bound on the number of threads that could be used to form a new team if a parallel construct without a num_threads clause were encountered after execution returns from this routine."
- **omp_get_nested**: "Returns whether nested parallelism is enabled or disabled, according to the value of the max-active-levels-var ICV."
- **omp_get_num_devices**: Returns the number of target devices.
- **omp_get_num_places**: "Returns the number of places available to the execution environment in the place list."
- **omp_get_num_procs**: "Returns the number of processors that are available to the current device at the time the routine is called."
- **omp_get_num_teams**: "Returns the number of teams in the current teams region, or 1 if called from outside a teams region."
- **omp_get_num_threads**: "Returns the number of threads in the current team. The binding region for an omp_get_num_threads region is the innermost enclosing parallel region. If called from the sequential part of a program, this routine returns 1."
- **omp_get_partition_num_places**: "Returns the number of places in the place-partition-var ICV of the innermost implicit task."
- **omp_get_partition_place_nums**: "Returns the list of place numbers corresponding to the places in the place-partition-var ICV of the innermost implicit task."
- **omp_get_place_num**: "Returns the place number of the place to which the encountering thread is bound."
- **omp_get_place_num_procs**: "Returns the number of processors available to the execution environment in the specified place."
- **omp_get_place_proc_ids**: "Returns numerical identifiers of the processors available to the execution environment in the specified place."
- **omp_get_proc_bind**: "Returns the thread affinity policy to be used for the subsequent nested parallel regions that do not specify a proc_bind clause."
- **omp_get_schedule**: "Returns the value of run-sched-var ICV, which is the schedule applied when runtime schedule is used."
- **omp_get_supported_active_levels**: Returns the number of active levels of parallelism supported.
- **omp_get_team_num**: "Returns the team number of the calling thread. The team number is an integer between 0 and one less than the value returned by omp_get_num_teams, inclusive."
- **omp_get_team_size**: Returns, for a given nested level of the current thread, the size of the thread team to which the ancestor or the current thread belongs.
- **omp_get_thread_limit**: "Returns the value of the thread-limit-var ICV: the maximum number of OpenMP threads available in contention group."

- **omp_get_thread_num**: "Returns the thread number of the calling thread, within the current team."
- **omp_in_final**: "Returns true if the routine is executed in a final task region; otherwise, it returns false."
- **omp_in_parallel**: "Returns true if the active-levels-var ICV is greater than zero; otherwise it returns false."
- **omp_is_initial_device**: "Returns true if the current task is executing on the host device; otherwise, it returns false"
- **omp_lock_t**: "In the following example, note that the argument to the lock functions should have type omp_lock_t, and that there is no need to flush it. The lock functions cause the threads to be idle while waiting for entry to the first critical section, but to do other work while waiting for entry to the second"
- **omp_nest_lock_t**: These routines provide a means of setting an OpenMP lock
- **omp_pause_resource**: "Allows the runtime to relinquish resources used by OpenMP on the specified device. Valid kind values include omp_pause_soft and omp_pause_hard."
- **omp_set_affinity_format**: "Sets the affinity format to be used on the device by setting the value of the affinity-format-var ICV"
- **omp_set_default_device**: "Assigns the value of the default-device-var ICV, which determines default target device."
- **omp_set_dynamic**: "Enables or disables dynamic adjustment of the number of threads available for the execution of subsequent parallel regions by setting the value of the dyn-var ICV"
- **omp_set_max_active_levels**: "Limits the number of nested active parallel regions, by setting max-active-levels-var ICV"
- **omp_set_nested**: "Enables or disables nested parallelism, by setting the max-active-levels-var ICV."
- **omp_set_num_threads**: "Affects the number of threads used for subsequent parallel constructs not specifying a num_threads clause, by setting the value of the first element of the nthreads-var ICV of the current task to num_threads."
- **omp_set_schedule**: "Affects the schedule that is applied when runtime is used as schedule kind, by setting the value of the run-sched-var ICV."

- **Lock routines**: These are a set of general-purpose lock routines that can be used for synchronization. These general-purpose lock routines operate on OpenMP locks that are represented by OpenMP lock variables. Two types of locks are supported: simple locks and nest-able locks. A nest-able lock can be set multiple times by the same task before being unset; a simple lock cannot be set if it is already owned by the task trying to set it.
  - **omp_destroy_lock**: Ensure that the OpenMP lock is uninitialized.
  - **omp_destroy_nest_lock**: Ensure that the OpenMP lock is uninitialized.
  - **omp_init_lock**: Initialize an OpenMP lock.
  - **omp_init_lock_with_hint**: Initialize an OpenMP lock with a hint.

  - **omp_init_nest_lock**: Initialize an OpenMP lock.
  - **omp_init_nest_lock_with_hint**: Initialize an OpenMP lock with a hint.
  - **omp_set_lock**: "Sets an OpenMP lock. The calling task region is suspended until the lock is set."
  - **omp_set_nest_lock**: "Sets an OpenMP lock. The calling task region is suspended until the lock is set."
  - **omp_test_lock**: "Attempt to set an OpenMP lock but do not suspend execution of the task executing the routine."
  - **omp_test_nest_lock**: "Attempt to set an OpenMP lock but do not suspend execution of the task executing the routine."
  - **omp_unset_lock**: Unsets an OpenMP lock.
  - **omp_unset_nest_lock**: Unsets an OpenMP lock.

- **Loop construct**: This construct specifies that the iterations of the associated loops may execute concurrently and permits the encountering thread(s) to execute the loop accordingly.
  - **#pragma omp loop (!$omp loop)**: "Specifies that the iterations of the associated loops may execute concurrently and permits the encountering thread(s) to execute the loop accordingly."

- **Master construct**: The master construct specifies a structured block that is executed by the master thread of the team.
  - **#pragma omp master (!$omp master)**: "Specifies a structured block that is executed by the master thread of the team."

- **Memory management directive**: Predefined memory spaces represent storage resources for storage and retrieval of variables.
  - **#pragma omp allocate (!$omp allocate)**: Specifies how a set of variables is allocated.

- **Memory management routines**: This one describes routines that support memory management on the current device.
  - **omp_alloc**: Requests a memory allocation from a memory allocator.
  - **omp_allocator_handle_t**: "Specifies the memory allocator to be used to obtain storage for private variables of a directive."
  - **omp_destroy_allocator**: Releases all resources used by the allocator handle.
  - **omp_free**: Deallocates previously allocated memory.
  - **omp_get_default_allocator**: "Returns the memory allocator to be used by allocation calls, allocate directives, and allocate clauses that do not specify an allocator."
  - **omp_init_allocator**: Initializes allocator and associates it with a memory space.
  - **omp_set_default_allocator**: "Sets the default memory allocator to be used by allocation calls, allocate directives, and allocate clauses that do not specify an allocator."

- **Parallel construct**: This feature creates a team of OpenMP threads that execute the region.
  - **#pragma omp parallel (!$omp parallel)**: "Creates a team of OpenMP threads that execute the region."
- **Requires directive**: This feature specifies the functionality that an implementation must provide so that the code to compile and to execute is correct.
  - **#pragma omp requires (!$omp requires)**: "Specifies the features that an implementation must provide in order for the code to compile and to execute correctly."
- **Scan directive**: This feature specifies that scan computations update the list items on each iteration.
  - **#pragma omp scan (!$omp scan)**: "Specifies that each iteration of scan computations update the list items"
- **SIMD directives**: The simd construct can be applied to a loop to indicate that the loop can be transformed into a SIMD loop (this means, multiple iterations of the loop can be executed concurrently using SIMD directives).
  - **#pragma omp simd (!$omp simd)**: "Applied to a loop to indicate that the loop can be transformed into a SIMD loop."
- **Synchronization constructs**: A synchronization construct orders the completion of code executed by different threads. This construct restricts execution of the associated structured block to a single thread at a time.
  - **#pragma omp atomic (!$omp atomic)**: Ensures a specific storage location is accessed atomically.
  - **#pragma omp barrier (!$omp barrier)**: "Specifies an explicit barrier that prevents any thread in a team from continuing past the barrier until all threads in the team encounter the barrier."
  - **#pragma omp critical (!$omp critical)**: "Restricts execution of the associated structured block to a single thread at a time."
  - **#pragma omp depobj (!$omp depobj)**: "Stand-alone directive that initalizes, updates, or destroys an OpenMP depend object."
  - **#pragma omp flush (!$omp flush)**: "Makes a thread's temporary view of memory consistent with memory, and enforces an order on the memory operations of the variables."
  - **#pragma omp ordered (!$omp ordered)**: "Specifies a structured block that is to be executed in loop iteration order in a parallelized loop, or it specifies cross iteration dependences in a doacross loop nest."
  - **#pragma omp taskgroup (!$omp taskgroup)**: "Specifies a region which a task cannot leave until all its descendant tasks generated inside the dynamic scope of the region have completed"
  - **#pragma omp taskwait (!$omp taskwait)**: "Specifies a wait on the completion of child tasks of the current task."

- **Tasking constructs**: This construct defines an explicit task.
  - **#pragma omp task (!$omp task)**: "Defines an explicit task. The data environment of the task is created according to the data-sharing attribute clauses on the task construct and any defaults that apply."
  - **#pragma omp taskloop (!$omp taskloop)**: "Specifies that the iterations of one or more associated loops will be executed in parallel using OpenMP tasks."
  - **#pragma omp taskloop simd (!$omp taskloop simd)**: "Specifies that a loop can be executed concurrently using SIMD instructions, and that those iterations will also be executed in parallel using OpenMP tasks."
  - **#pragma omp taskyield (!$omp taskyield)**: "Specifies that the current task can be suspended in favor of execution of a different task."
- **Teams construct**: This feature creates a league of initial teams and the initial thread in each team executes the region.
  - **#pragma omp teams (!$omp teams)**: "Creates a league of initial teams where the initial thread of each team executes the region."
- **Timing routines**: This one describes routines that support a portable wall clock timer.
  - **omp_get_wtick**: "Returns the precision of the timer (seconds between ticks) used by omp_get_wtime"
  - **omp_get_wtime**: Returns elapsed wall clock time in seconds.
- **Tool Activation**: A function to activate a tool.
  - **ompt_callback_t**: "All callbacks registered with ompt_set_callback or returned by ompt_get_callback use the dummy type signature ompt_callback_t. While this is a compromise, it is better than providing unique runtime entry points with a precise type signatures to set and get the callback for each unique runtime entry point type signature."
  - **ompt_set_callback**: "To monitor execution of an OpenMP program on the host device, a tool's initializer must register to receive notification of events that occur as an OpenMP program executes. A tool can register callbacks for OpenMP events using the runtime entry point known as ompt_set_callback."
  - **ompt_set_error**: "If the ompt_set_callback runtime entry point is called outside a tool's initializer, registration of supported callbacks may fail with a return code of ompt_set_error."
  - **ompt_start_tool**: "A tool indicates its interest in using the OMPT interface by providing a non-NULL pointer to an ompt_start_tool_result_t structure to an OpenMP implementation as a return value from ompt_start_tool."
- **Tool control routine**: This routine enables a program to pass commands to an active tool.
  - **omp_control_tool**: Enables a program to pass commands to an active tool.

- **Variant directives**: There are two variant directives. The meta-directive is a directive that can specify multiple directive variants of which one may be conditionally selected to replace the meta-directive based on the enclosing OpenMP context. On the other hand, the declare variant directive declares a specialized variant of a base function and specifies the context in which that specialized variant is used.
  - **#pragma omp declare variant (!$omp declare variant)**: "Declares a specialized variant of a base function and the context in which it is used."
  - **#pragma omp metadirective (alternative in C: #pragma omp begin metadirective and in Fortran: !$omp metadirective)**: "A directive that can specify multiple directive variants, one of which may be conditionally selected to replace the metadirective based on the enclosing OpenMP context."
- **Worksharing constructs**: This non-iterative construct contains a set of structured blocks that are to be distributed among and executed by the threads in a team.
  - **#pragma omp sections (!$omp sections)**: "A noniterative worksharing construct that contains a set of structured blocks that are to be distributed among and executed by the threads in a team."
  - **#pragma omp single (!$omp single)**: "Specifies that the associated structured block is executed by only one of the threads in the team."
  - **!$omp workshare**[2]: "Divides the execution of the enclosed structured block into separate units of work, each executed only once by one thread."
- **Worksharing-loop construct**: A worksharing construct distributes the execution of the corresponding region among the members of the team that encounters it.
  - **#pragma omp for (!$omp do)**: "Specifies that the iterations of associated loops will be executed in parallel by threads in the team in the context of their implicit tasks."
- **Worksharing-Loop SIMD**: Applied to a loop to indicate that the loop can be transformed into a SIMD loop that will be executed in parallel by threads in the team.
  - **#pragma omp declare simd (!$omp declare simd)**: "Applied to a function or a subroutine to enable the creation of one or more versions that can process multiple arguments using SIMD instructions from a single invocation in a SIMD loop."
  - **#pragma omp for simd (!$omp do simd)**: "Applied to a loop to indicate that the loop can be transformed into a SIMD loop that will be executed in parallel by threads in the team."

[2] no C command available

# Appendix B

# List of HPC Applications & Release or Last Update Year

Table B.1: List of HPC Applications & Release or Last Update Year (Part I)

| Application names | Year | Application names | Year |
|---|---|---|---|
| 113.GemsFDTD | 2007 | 502.gcc_r | 2017 |
| 125.RAxML | 2007 | 505.mcf_r | 2017 |
| 127.wrf2 | 2007 | 507.cactuBSSN_r | 2017 |
| 147.l2wrf2 | 2007 | 521.wrf_r | 2017 |
| 305.md | 2012 | 525.x264_r | 2017 |
| 352.nab | 2012 | 526.blender_r | 2017 |
| 357.bt331 | 2012 | 527.cam4_r | 2017 |
| 358.botsalgn | 2012 | 538.imagick_r | 2017 |
| 359.botsspar | 2012 | 544.nab_r | 2017 |
| 360.ilbdc | 2012 | 549.fotonik3d_r | 2017 |
| 362.fma3d | 2012 | 554.roms_r | 2017 |
| 363.swim | 2012 | 557.xz_r | 2017 |
| 367.imagick | 2012 | Alya | missing |
| 370.mgrid331 | 2012 | amgcl-1.3.99 | 2012 |
| 371.applu331 | 2012 | AMG-master | 2014 |
| 376.kdtree | 2012 | ascent-0.5.1 | 2020 |

Table B.3: List of HPC Applications & Release or Last Update Year (Part II)

| Application names | Year | Application names | Year |
|---|---|---|---|
| b+tree | 2015 | feltor-5.0 | 2018 |
| backprop | 2015 | FLASH4.6.2 | 2019 |
| bfs | 2015 | gamess | 2018 |
| cfd | 2015 | GenASiS_Basics-3.1 | 2019 |
| Chombo-master | 2015 | genesis-1.5.1 | 2020 |
| CloverLeaf_ref-1.3 | 2015 | gromacs-2020.3 | 2020 |
| code_saturne-6.2.0 | 2020 | heartwall | 2007 |
| Comb-0.1.1 | 2019 | hotspot | 2007 |
| CoMD-1.1 | 2017 | hotspot3D | 2007 |
| CoreNeuron-0.21 | 2020 | hpcg-3.1 | 2020 |
| CoSP2-master | 2015 | hpgmg | 2018 |
| cp2k-7.1.0 | 2019 | hypre-2.20.0 | 2020 |
| E3SM-1.1.0 | 2019 | kmeans | 2007 |
| EigenExa-2.4b | 2018 | Kratos-8.0.1 | 2020 |
| elmerfem-release-8.4 | 2019 | lammps-patch_18Sep2020 | 2020 |
| faunus-2.4.1 | 2020 | lavaMD | 2007 |

Table B.5: List of HPC Applications & Release or Last Update Year (Part III)

| Application names | Year | Application names | Year |
|---|---|---|---|
| leukocyte | 2007 | nest-simulator-2.20.0 | 2018 |
| lud | 2007 | nn | 2007 |
| LULESH-2.0.2 | 2017 | ntchem-mini-1.2 | 2019 |
| meep-1.16.0 | 2020 | nw | 2007 |
| miniAMR-1.6.4 | 2019 | nwchem-7.0.1-release | 2018 |
| miniFE-2.2.0 | 2017 | Nyx-19.08 | 2018 |
| miniMD | 2019 | particlefilter | 2007 |
| miniSMAC-2.0.0 | 2016 | PENNANT-pennant_v0.9 | 2016 |
| miniTri-1.0 | 2017 | picsar | 2019 |
| miniVite-1.1 | 2018 | plasma | 2014 |
| moab | 2019 | primecount-6.1 | 2019 |
| mummergpu | 2007 | qbox_67_4 | 2014 |
| MUMPS_5.3.4 | 2017 | qe-6.6 | 2019 |
| myocyte | 2007 | qmcpack-3.9.2 | 2018 |
| NAMD_2.14_Source | 2019 | rocALUTION-rocm-3.8.0 | 2019 |
| nekbone | 2014 | siesta-4.0.2 | 2018 |

Table B.7: List of HPC Applications & Release or Last Update Year (Part IV)

| Application names | Year | Application names | Year |
|---|---|---|---|
| SimpleMOC-4 | 2018 | TeaLeaf_ref-1.3 | missing |
| souffle-2.0.2 | 2020 | Trilinos-trilinos-release-13-0-0 | 2019 |
| sphynx | missing | tycho2 | 2019 |
| splatt-1.1.0 | 2018 | vlasiator | 2021 |
| srad | 2007 | WRF-4.2.1 | 2018 |
| streamcluster | 2007 | yambo-4.5.0 | 2019 |
| sw4lite-RAJA-v1.0 | 2019 | | |