An Empirical Study of OpenMP Directive Usage in Open-Source Projects on GitHub[*]

Cristian Carvalho Quevedo[1], Simone André da Costa Cavalheiro[1]
Marcos Antonio de Oliveira Jr.[2], André Rauber Du Bois[1]
Gerson Geraldo H. Cavalheiro[1]

[1]PPGC - Universidade Federal de Pelotas
Pelotas – RS – Brazil

{ccquevedo,scosta,ardubois,gersonc}@inf.ufpel.edu.br

[2]Instituto Federal de Educação, Ciência e Tecnologia Farroupilha
Santa Maria – RS – Brazil

marcos.oliveira@iffarroupilha.edu.br

Abstract. *This paper presents a mapping of OpenMP API usage in open-source C/C++ projects on GitHub. The study investigates the frequency and patterns of OpenMP directive utilization through a data mining process on relevant repositories. The analysis reveals a predominant focus on loop parallelization and identifies opportunities for optimization in scheduling strategies and critical section handling. The study also uncovers underutilization of vectorization capabilities and potential for code restructuring to enhance parallel performance. The findings offer valuable insights into the practical application of OpenMP, contributing to the development of improved programming practices, educational resources, and tools that support efficient parallel programming with OpenMP. All software artifacts developed for this study are available to foster reproducibility and further research.*

1. Introduction

Mining software repositories is crucial for improving review processes in open-source projects by extracting detailed data on people, processes, and products. This data enables the analysis and enhancement of quality and efficiency in software development. An exciting area of research involves using mined data to develop techniques and tools related to code review [Yang et al. 2016]. GitHub[1] is a leading platform for version control and collaborative development, utilizing the Git system.

In this context, mining repositories helps us understand projects employing specific technologies, such as concurrent programming with OpenMP.[2] OpenMP is an API that supports parallel programming in C, C++, and Fortran, enabling developers to utilize multiple processors efficiently. Although reference documents provide guidance on using OpenMP directives to parallelize program execution, these directives are not always applied correctly or efficiently.

[*]This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior, Brasil (CAPES), Finance Code 001.

[1]GitHub [https://github.com/]

[2]OpenMP [https://www.openmp.org/]

Analyzing the usage of OpenMP directives helps to understand how programmers use this tool and how its effectiveness can be improved. Repository mining offers valuable insights into project dynamics, especially in open-source environments. By examining mined data, it is possible to identify collaboration patterns, process effectiveness, and project quality, leading to targeted improvements. This practice not only provides insights into social aspects, such as team collaboration and information flow, but also reveals technical issues like file organization and common challenges faced by the community. Thus, repository mining is essential for a thorough understanding of concurrent programming projects and for advancing and optimizing these practices.

This paper presents a mapping of OpenMP usage in open-source projects on GitHub. A case study was conducted to extract data about OpenMP directive usage, aiming to map the state of the art and aspects of concurrent programming, such as resource sharing. The main contributions of this research are the following:

- Quantifies the use of different OpenMP directives in public repositories on GitHub;
- Case study of the use of critical sections;
- Case study of loop parallelism exploitation;
- Software artifacts produced for mining and analysis of case studies.

This paper is structured in sections as follows. Section 2 outlines related works. Section 3 presents the methodology and artifacts developed for mining public repositories. Section 4 discusses the results obtained, detailing the mining analyses. Finally, Section 5 concludes the paper and presents future research directions.

2. Related Works

Mining Software Repositories (MSR) is a complex task with many challenges and can be conducted for different purposes [Luzgin and Kholod 2020]. Some works focus on analyzing existing source codes to facilitate the development of new codes, while others seek to extract and organize data from repositories [de F. Farias et al. 2016]. Some papers directly related to the proposed approach are listed in this section.

[Romano et al. 2021] addresses some challenges of using the GitHub REST API for MSR studies, specifically API limitations, language misclassification, and the inclusion of non-software artifacts. The authors developed *G-Repo*, a tool that aids researchers in creating and cleaning datasets. G-Repo provides functionality for querying GitHub, cloning repositories, checking programming languages, and detecting the spoken language of repositories. The authors show that G-Repo effectively helps researchers overcome common hurdles in gathering and preparing data, being a valuable tool for general MSR work.

Also, in tool format, in [Gote et al. 2019], the authors present *git2net*, a Python tool for extracting fine-grained and time-stamped co-editing networks from large Git repositories. Unlike traditional approaches focusing on co-authorship at the file level, git2net analyzes the detailed history of textual modifications within files, identifying specific lines or code regions edited by different developers. The paper demonstrates the tool's capabilities through case studies on Open Source and commercial software projects. It showcases its ability to reveal insights into collaboration patterns, knowledge flow, and developer effort allocation that are not captured by coarser-grained methods. This tool is also powerful for analyzing collaboration patterns in general terms without specializing.

In [Biswas et al. 2019], the authors introduce a new Boa dataset specifically curated for Data Science software developed using Python. The authors collected 1,558 mature GitHub projects using Python libraries for machine learning and data analysis tasks. The authors demonstrate the dataset's potential through two applications: analyzing the distribution of projects between organizations and individual users and identifying frequent API call sequences in neural network projects. The dataset provides a valuable resource for MSR research focused on understanding the development practices and challenges in data-intensive Python software without addressing specific aspects of programming, such as concurrency.

[Munaiah et al. 2017] addresses the challenge of separating "signal" (actual software projects) from "noise" (non-software artifacts) in massive repositories like GitHub. The authors propose an evaluation framework based on seven dimensions representative of software engineering practices, such as documentation, testing, and community participation. They implemented this framework as a tool called *reaper* to automatically evaluate repositories and predict whether they contain engineered software projects. The results show that their classifiers achieve higher recall, identifying a more comprehensive range of engineered projects than the stargazer-based method, offering a valuable tool for researchers to curate relevant datasets. This paper tackles the broader issue of identifying general engineered software projects rather than investigating the use of a specific tool.

While the importance of papers that discuss the MSR and their achievements is noted, there is also a gap in specialized research focused on specific aspects, such as concurrent programming and the use of tools such as OpenMP. Therefore, this work comes in this research direction to push the state of the art, presenting a set of analyses on public GitHub repositories that use OpenMP directives to produce and organize this knowledge.

3. Mining OpenMP Public Repositories

The process of identifying and retrieving OpenMP-utilizing repositories on GitHub is detailed in this section. The developed artifacts for mining and the parameters used to determine the repositories of interest are presented. The section includes a tabulation of the quantitative data from the mining process.

3.1. Role of Source Code Repository Platforms in Software Development

Currently, there are several source code repository platforms, such as the widely known GitHub, which play a fundamental role in collaboration and software development. These platforms provide developers with a robust infrastructure for storing, sharing, and collaborating on open and private source projects. They also enable version control by offering features such as issue tracking, pull requests, and continuous integration, facilitating coordination between distributed teams and improving code quality. Furthermore, a key feature of source code hosting platforms is the public availability of information, meaning that, when the owner allows it, projects can be easily accessed and examined by anyone interested, promoting transparency and the dissemination of knowledge.

This public accessibility encourages collaborative learning and development and promotes trust in the software community by allowing others to build on existing work, identify and fix problems, and contribute improvements. This openness democratizes the software development process, making it more inclusive and receptive to contributions from various participants.

Table 1. Frequency of metrics by range of values.

Metric	0	(1-10]	(10-100]	(100-1,000]	$> 1{,}000$
Stars	653	515	111	31	2
Forks	906	339	56	10	1
Files	0	851	460	1	0
Collaborators	42	1,241	26	3	0

3.2. Data extraction

The data extraction was conducted in May 2023. The repositories considered were those tagged with the topic "openmp," developed in C and C++ languages. A total of 1,312 repositories were collected, of which 592 were in C and 720 in C++. The repositories were cloned locally, and only files containing OpenMP directives were retained, organized by directory identification. Thus, the database consisted of 13,343 files. During the extraction process, the number of stars, forks, files, and collaborators for each repository was recorded, and this data is summarized in Table 1.

Once the mining stage was completed and a local copy of the files manipulating OpenMP was made, a file preparation stage for processing was conducted. The file preparation was done first using the `dos2unix` utility to ensure all files conformed to the standard Unix format and then using the stream editor `sed`. With `sed`, all double-spacing characters were removed from the lines containing the sequence "#pragma omp" to facilitate pattern identification.

3.3. Discussion

According to Table 1, most repositories have a low number of stars and forks. It is also observed that most repositories, in addition to their owner, have up to 10 collaborators. Two repositories have more than 1,000 Stars: one is developed in C, with 7,761 Stars and 1,878 Forks, and another in C++, with 3,159 Stars and 883 Forks. Several have at most 10 Stars and Forks.

This better rating in terms of number of stars appears to be accompanied by the number of forks, as also identified by [Borges et al. 2016, Borges and Tulio Valente 2018]. The *Pearson's r* test to verify the correlation between the metrics was applied, indicating that there is a significant large positive relationship between the number of stars and forks attributed to the repositories ($r(1310) = 0.9872$, $p \leq 0.001$). By removing the two repositories that were outliers in the sample in terms of the number of Stars, the correlation remained very strong ($r(1308) = 0.8484$, $p \leq 0.001$). For no other combination of metrics was a correlation coefficient implying causality found. In conclusion, it is accepted that there is a correlation between the number of Stars and Forks assigned to the repositories, but no other relationship between the metrics was identified.

The file preparation stage also allowed for the identification of repositories that were not completed projects but rather small experiments and trials. Since a methodology for repository acceptance was not foreseen, these projects were not discarded.

4. Analysis of Mining Results

This section presents the tools and methods used to analyze the use of OpenMP directives in the codes of the mined repositories. Two subsections present the raw data obtained from the mining process. Following this, two of several aspects of using OpenMP are discussed below: data sharing and parallel loops. The section ends by characterizing the limits of the work performed.

4.1. Tools, methods, and reprodutibility

The main tools used to extract information about the code were the `sed` utility for preparing the files for processing and `awk` for identifying the use cases of the directives. Only files with the following extensions were considered: `.c`, `.h`, `.C`, `.H`, `.cpp`, `.hpp`, `.cxx`, `.hxx`, and `.inl`.

The repository files were first manipulated to simplify the process of identifying patterns in the use of the directives. All leading whitespace and tabs were removed from the beginning of lines, causing the OpenMP sentinels `#pragma omp` to occur at the beginning of a line. All lines starting with an OpenMP sentinel and ending with an escape character ("\") were concatenated with the following line, removing the escape character. Finally, it was ensured that all lines containing `#pragma omp` used one, and only one, whitespace to separate the words.

The next step was to identify patterns in the use of OpenMP directives. Scripts were developed for each case considered in the study. These scripts do not implement a complete grammar for the OpenMP directives or the C and C++ languages. However, these scripts are sufficient to identify code snippets of interest for the desired analyses.

The set of artifacts developed in this work and the data used for analyses are available in a public repository for reproducibility and verification of results.[3] The software artifacts available in this repository include the Python script for mining and the `awk` scripts used for pattern searching. The repository also contains instructions on how to use the developed scripts and the command lines using `grep` and `sed` that were employed.

4.2. OpenMP and C++

The collected data revealed that the number of project repositories using OpenMP developed with C++ as the primary language is higher than those with C. This superiority is about 20%. By observing the numbers of Stars and Forks, one can also see that there is a superiority in terms of reputation.

OpenMP was first developed for the C language,[4] and it has always been possible to use it with the C++ language, provided it was limited to features inherited from C, not the new functionalities of C++. Starting from OpenMP 3.0 features specific to C++, such as container manipulation and iterators, were incorporated. In the current version, OpenMP 5.2, this support has been expanded to modern C++ features, such as lambda expressions and `std::array` references, facilitating the integration of OpenMP with more idiomatic and expressive C++ code [Board 2021]. The complexity in handling

[3]`https://github.com/GersonCavalheiro/OpenMPSnapshot`.

[4]And Fortran, but in this paper, the focus is on the use of OpenMP with the C and C++ languages.

class instances can be exemplified by considering the complexity of object construction, copying, and destruction operations when passed as parameters to the `private`, `first/lastprivate`, or even `reduction` clauses. The expressiveness can be exemplified by the use of iterators in parallel loops and support for exception handling.

Due to the particularities of using C++ with OpenMP, an investigation was conducted to see how these features were being utilized in C++ repositories. The analysis was done manually, examining the code of 10 repositories, with 5 of these repositories having between 300 and 900 stars and the other five between 90 and 150 stars. Repositories within this range were randomly selected with the aim of eliminating extremes of projects with very high or low reputation[5]. The aspects investigated were the use of exception handling mechanisms, utilization of class instances as parameters for the `private`, `first/lastprivate`, and `reduction` clauses, and the use of parallel loops controlled by iterators over containers.

The investigation revealed that, out of the ten manually analyzed repositories, only one made use of all the investigated aspects. This repository, among those selected, had the highest number of stars. Regarding the other elements, only one other repository, which had the second-highest number of stars, used object instances in the clauses. In terms of the use of exception handling mechanisms, 4 out of the ten repositories utilized this feature.

4.3. Frequency of directives

Table 2 presents the quantitative usage of OpenMP directives in the analyzed repositories. The data tally occurrences identified in repositories where the primary languages are C and C++. The table also includes the OpenMP version in which the directive was introduced and a classification devised by the authors of this paper to identify the nature of the directive. Table 3 shows the proportion of directives in each category. A notable observation is the extensive use of directives for parallelism exploration in loops.

Table 2. Frequency of directives, quantitative.

Directive	Version	Total	Category	Directive	Version	Total	Category
parallel	1.0	16766	Parallel Control	taskyield	3.4	16	Synchronization; Task
parallel for	1.0	10446	Parallel Control; Loop	simd	4.0	1772	Parallel Control; Loop; SIMD
for	1.0	6177	Parallel Control; Loop	target	4.0	3428	Teams and Distribution
barrier	1.0	2781	Synchronization	teams	4.5	277	Teams and Distribution
single	1.0	1718	Parallel Control; Implicit Task	parallel for simd	4.5	198	Parallel Control; Loop; SIMD
critical	1.0	1414	Synchronization	distribute	4.5	142	Parallel Control
section	1.0	1012	Parallel Control; Implicit Task	for simd	4.5	44	Parallel Control; Loop; SIMD
master	1.0	931	Parallel Control; Implicit Task	taskloop simd	4.5	31	Parallel Control; Loop; Task; SIMD
sections	1.0	219	Parallel Control; Implicit Task	requires	4.5	22	Metaprog. and Requirements
parallel sections	1.0	183	Parallel Control; Implicit Task	declare	5.0	774	Metaprog. and Requirements
atomic	2.0	2485	Synchronization	loop	5.0	47	Parallel Control; Loop
ordered	2.0	364	Synchronization	metadirective	5.0	8	Metaprog. and Requirements
flush	2.0	238	Synchronization	depobj	5.0	7	Mem. Alloc. and Management
threadprivate	2.0	152	Data Privacy and Sharing	allocate	5.0	4	Mem. Alloc. and Management
task	3.0	3553	Parallel Control; Task	assume	5.1	7	Metaprog. and Requirements
taskwait	3.1	834	Synchronization; Task	error	5.1	5	Execution Control and Debugg
taskloop	3.2	329	Parallel Control; Loop; Task	tile	5.1	2	Parallel Control; Loop
taskgroup	3.3	96	Synchronization; Task	mask	5.1	2	Execution Control and Debug
						56488	

While concurrency exposure clauses come with directives to parameterize task construction, indicating how each task should share the address space with others, there is also notable usage of the `critical` and `atomic` directives.

[5]The repository identifiers are (first group) 81815495, 94275048, 6987353, 38410417, 40821917, (second group) 73826981, 84174010, 58775556, 69450880, and 322989201.

Table 3. Usage Breakdown for Parallelism Control

Category			Usage
Parallelism Control[a]	*Loop*	*66.12%*	59.16%
	Task	*13.48%*	
	Implicit Task	*13.37%*	
	SIMD	*7.04%*	
Synchronization			19.44%
Teams and Distribution			8.75%
Metaprogramming and Requirements			1.91%
Data Privacy and Sharing			0.36%
Execution Control and Debugging			0.03%

[a]Not considering `parallel` directive alone.

4.4. Dealing with shared data

Critical sections are used to ensure consistency in accessing shared data. OpenMP offers mechanisms based on performing atomic operations or blocks. The available directives are `atomic` and `critical`.

4.4.1. Usage of the `atomic` Directive

The `atomic` directive allows operating atomically on data whose size is a word, such as an integer or pointer, in an optimized manner, as it does not require synchronization mechanisms like locks. It is implemented directly in hardware with support from instructions such as *fetch-and-add*, *compare-and-swap*, or *atomic exchange*. This directive can receive clauses that determine its behavior and was used 2,485 times in the analyzed repositories. Table 4 details the number of times each valid clause for this directive was used. The most used clause was `update`, assumed as default in 1193 cases. A manual inspection of using the `atomic` directive with its various clauses was conducted.

Table 4. Frequency of clauses applied to `atomic`.

`update`	`write`	`read`	`compare`	`capture`	`hint`
1561	307	278	5	334	0

The most used clause, `update`, allows for atomic read and write access, as in `++x;` or `x=x+1;`. Atomicity is ensured by reading the variable `x` and writing the result back into `x`. The manual inspection identified that the use of this clause maintained the expected pattern associated with cumulative operations. Using the `write` and `read` clauses also adhered to the expected pattern. The `write` clause performs atomic write access, as in `x=y;`, where atomicity lies in writing the result of evaluating *expr* into the variable `x`. On the other hand, the `read` clause performs atomic read access, as in `y=x;`, where atomicity lies in reading the value of `x`. However, in the use of these three clauses, some constructs involved complex mechanisms for accessing variable addresses, such as `x[foo(z)] -= y;` or even involving a large number of identifiers, where atomic access is not guaranteed in operation, such as `x[i] += a * y[j++];` and `x[i][j-2][k-1];`.

The `capture` clause allows a variable's value to be read in an expression and then modified atomically, as in `y=x++;` and `y=x; x=z;`, where the read and write accesses to variable `x` are performed atomically. An important aspect to consider is that the read-write sequence at the address is not atomic. This clause was introduced in version 5.1 of OpenMP and was identified in only two repositories.

The `compare` and `hint` clauses were introduced in the latest version, 5.2, of OpenMP. Both clauses were used in only one repository. Although this paper does not conduct a qualitative study of the repositories, in this specific case, it was noted that their use was for testing the functionality of the features rather than their practical application as part of solving a problem. The fact that it is a *new* feature in OpenMP may explain its low representation in the samples.

4.4.2. Usage of the `critical` Directive

The `critical` directive was used 1,414 times, where 293 with a label associated with the critical section. Table 5 presents additional data regarding code snippets related to the use of the `critical` directive. One dataset identifies the potential for the associated block to be implemented with the `atomic` directive, and another identifies long critical sections. Except for the data related to collections, which were identified by manual inspection, all other data was extracted automatically using an Awk script.

Table 5. Characteristics of critical sections with the `critical` directive.

Usage of `critical` candidates for `atomic`			Usage of `critical` in long critical sections		
Read/Write	Compound Assignment	Increment	Loops	`if then [else]`	Collections[a]
508	161	61	53	35	204

[a]Cases were identified by manual inspection.

According to the data in Table 5, there are 730 cases where the `atomic` directive could be employed directly, 508 using the `read` or `write` clauses, 222 using the `update`. Observing the remaining code snippets, 53 of them contained at least one sequential loop (`for`, `while`, or `do while` commands). In 35 of them, there was at least one `ifthen [else]` command. During the inspection, a manual intervention identified that 204 critical sections were used for manipulating collections (`vector`, `list`, `deque`, etc.).

4.4.3. Findings

The code snippets associated with critical sections were inspected for their compliance with the employed directives. Generally, it was identified that the application of the atomic directive was adequate regardless of the clause. This is expected, as compilers in their recent versions verify if the expressions used correspond to the clauses applied, presenting compilation errors if they do not. However, it was observed that in many cases, the expressions are complex regarding the memory locations accessed. In these instances, there is no integrity guarantee for memory areas not handled by atomic operations.

On the other hand, the use of the critical directive showed that there is room for improvement in several aspects. This directive impacts program performance by limiting the exploitation of hardware parallelism. The first point observed was the low number of occurrences where this directive is used without being bound to a label. The no use of a label extends the synchronization granularity to the program, increasing the risk of operation serialization. It was also identified that, in many cases, the atomic directive could suffice since the critical section is limited to performing operations suitable for this directive, thus reducing contention. Finally, several critical sections containing loops were identified, which may indicate long stretches of serialized code, leading to a loss of parallelization potential. Additional observations include the use of critical sections in the manipulation of data collections and in accessing MPI library services.

4.5. Parallel loops

The OpenMP directives `for`, `loop`, `simd`, `taskloop`, and `distributed` expose loop concurrency in programs. Considering the data from Table 3, this is the most exploited form of parallelism in the analyzed repositories. The analysis focuses on loops constructed with the directives `for`, and its variant `parallel for`, and `simd`.

4.5.1. Scheduling

The `parallel for` and `for` directives divide the iteration space into chunks (subsequences), assigning each chunk to a task. With the `schedule` clause, the programmer can control how tasks are distributed among threads and set the chunk granularity. Three scheduling options are available: `static`, the default where iterations are considered equal in cost and tasks are mapped directly to threads; `dynamic`, where the computational cost varies and tasks are drawn from a shared list; and `guided`, where chunk sizes decrease progressively, with tasks also in a shared list. The chunk size can be specified, defaulting to 1. `Auto` and `runtime` options let the environment choose from the three strategies or use the default configuration.

Out of 16,468 uses of parallel loops with `parallel for` and `for`, almost half of them, precisely 8,306 cases, utilize the `schedule` clause. The breakdown is detailed in Table 6, which shows the number of times each clause was applied to each loop construct and how many of these occurrences defined a chunk size. The *Undefined* column refers to cases where the scheduling strategy is a compile-time defined macro. It can be observed that `static` and `dynamic` schedulings are the most commonly used, and there is significant exploration of the option to determine the chunk size. It is also evident that the number of cases where loop parallelization was done using the `parallel for` construct is significantly higher than those using the `for` construct.

The cases where the set of iterations associated with a parallel loop includes at least one `if` statement were also quantified: we found 101 cases associated with the use of `static` or default scheduling, 230 with `dynamic` scheduling, 21 with `guided`, and 47 with `auto` and `runtime` scheduling. This scenario indicates that, proportionally

Table 6. Strategy Selection for Scheduling with/without Chunk Size.

Directive	None	static	dynamic	guided	auto	runtime	Undef.
parallel for	8230	440/286	190/433	73/89	150/0	186/0	364
for	4624	558/171	94/185	37/139	14/0	167/0	186

to the total number of loops submitted to each type of scheduling, loops scheduled with the dynamic and guided strategies have the highest rates. However, the static strategy, being the default in the absence of the schedule clause, still accounts for a significant number of loops containing code structures that may lead to an imbalance in computational costs between tasks.

4.5.2. Close and nested parallel regions

The parallel for directive combines the functionalities of the parallel and for directives, allowing the creation of a parallel region that manages the execution and scheduling of tasks, in addition to generating the tasks to be executed. The search for occurrences of two or more parallel for directives, nested or not, within the same block of commands identified 444 code snippets. Although functionally correct, the pattern of two or more parallel for directives in the same block of commands can be rethought regarding code organization.

Manual inspection of these codes verified that in 19.27% of the cases, it is impossible, without changing the algorithm of the inspected snippet, to make a single parallel region encompassing two parallel loops. In 40.82% of the inspected cases, the code snippet could be directly reimplemented, although in 38.33% of these cases, the need to include a single task between them was verified.

Regarding the nesting of parallel loops, we identified during the manual inspection that in 1,074 (6.46%) of the parallel loops, parallel for and for, the collapse clause was used. On the other hand, in code blocks containing two or more parallell for directives, 39.91% of cases of nesting of these directives were identified (1.06% of the total loops).

4.5.3. Vector Parallelism

Current architectures allow exploiting loop parallelism through vector instructions or SIMD (Single Instruction, Multiple Data) flow. In this type of parallelism, multiple vector positions can be operated on simultaneously, significantly increasing loop processing speed by manipulating vectors. In OpenMP, the intention to exploit vector parallelism can be indicated by the simd directive, either independently or in conjunction with other loop directives (parallel for, for, and taskloop), starting from version 4.0. According to the collected data (Table 2), the simd directive was used 1.772 times independently and 242 times in combination with parallel for and for directives. Compared to other directives introduced in the same version, there is an indication that only the target directive was more frequently used.

The `simd` being a relatively new directive, cases were investigated where its use could have been explored in repositories, but the implementation was limited to exploiting loop parallelism. Code segments were identified that corresponded to a loop pattern (`parallel for` or `for`), followed or not by a block, containing at least one command where an array position appears on the left side of an assignment operation. A total of 1,558 cases were identified, representing 9.4% of the total loops where this optimization could be considered.

4.5.4. Findings

Loop structures are popular, showing the interest of parallel software designers in OpenMP. Even the newer simd directive is widely used. However, there is room for improvement in program implementation. Many do not effectively specify scheduling strategies or chunk sizes. Frequent use of the parallel for directive multiple times in the same command block suggests potential for performance enhancement. While vector/SIMD parallelism is broadly adopted, its usage could be optimized further.

Additionally, about 50% of cases lack scheduling strategies, though when used, chunk size definition is common. There is a notable prevalence of static scheduling in loops with selection commands, indicating possible iteration imbalances.

4.6. Limitations and Threats to Validity

In this work, we presented data on the use of different OpenMP directives in public repositories. While we analyzed the examined codes to understand OpenMP usage, we did not explore code quality or performance aspects in depth. Our goal was to present the scenario of OpenMP usage on GitHub. One limitation is that our characterization is based on a snapshot of repositories from a specific date, but this can be mitigated by the reproducibility of our methodology. Another limitation is the reliance on repository owners labeling their projects with the "openmp" tag, which may not always happen, thus excluding some repositories from analysis. Additionally, we did not differentiate between mature projects and those primarily containing tests or academic work, referred to as "noise" in [Munaiah et al. 2017]. Our pattern identification for code snippets was also limited, identifying recurring patterns rather than implementing a complete grammar. Despite these limitations, manual sampling inspections indicated the adequacy of our method, although they are subject to interpretation and potential errors.

To address these limitations, we have provided all developed artifacts and the data used in our study, offering readers the opportunity to evaluate these constraints for themselves. This transparency allows for further examination and even the identification of methods to overcome these limitations.

5. Conclusion

This paper provides a snapshot of OpenMP usage in C and C++ projects on GitHub, analyzing data from 1,312 repositories. The study reveals trends in the application of OpenMP directives, with a focus on for-loop parallelization and data sharing. It identifies opportunities for optimization, such as improving scheduling strategies and reducing the reliance on critical sections.

The research provides valuable insights into how the OpenMP community uses this tool in practical scenarios, contributing to the development of best programming practices, educational resources, and tools. Despite the challenges of mining software repositories, the methodology and artifacts from this study form a strong foundation for future research. The publicly accessible resources support reproducibility and further investigation, aiding in the advancement of parallel programming practices and education.

References

Biswas, S., Islam, M. J., Huang, Y., and Rajan, H. (2019). Boa meets python: A boa dataset of data science software in python language. In *Proc. of the 16th Inter. Conf. on Mining Software Repositories*, pages 577–581.

Board, O. A. R. (2021). *OpenMP Application Programming Interface Specification 5.2*.

Borges, H. and Tulio Valente, M. (2018). What's in a github star? understanding repository starring practices in a social coding platform. *Journal of Systems and Software*, 146:112–129.

Borges, H. S., Hora, A. C., and Valente, M. T. (2016). Understanding the factors that impact the popularity of github repositories. *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 334–344.

de F. Farias, M. A., Novais, R., Colaço Jr, M., Carvalho, L. P. S., Mendonça, M., and Spínola, R. O. (2016). A systematic mapping study on mining software repositories. In *Proc. of the 31st ACM Symp. on Applied Computing*, pages 1472–1479, New York.

Gote, C., Scholtes, I., and Schweitzer, F. (2019). git2net - mining time-stamped co-editing networks from large git repositories. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 433–444.

Luzgin, V. A. and Kholod, I. I. (2020). Overview of mining software repositories. In *2020 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*, pages 400–404.

Munaiah, N., Kroh, S., Cabrey, C., and Nagappan, M. (2017). Curating GitHub for engineered software projects. *Empirical Software Engineering*, 22.

Romano, S., Caulo, M., Buompastore, M., Guerra, L., Mounsif, A., Telesca, M., Baldassarre, M. T., and Scanniello, G. (2021). G-repo: a tool to support msr studies on github. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 551–555.

Yang, X., Kula, R. G., Yoshida, N., and Iida, H. (2016). Mining the modern code review repositories: a dataset of people, process and product. In *Proc. of the 13th Inter. Conf. on Mining Software Repositories*, pages 460–463, New York.