A Snapshot of OpenMP Projects on GitHub

Cristian Carvalho Quevedo
Universidade Federal de Pelotas
Pelotas, Brazil
ccquevedo@inf.ufpel.edu.br

Marcos Antonio de Oliveira Jr.
IFFar - Instituto Federal de Educação, Ciência e Tecnologia Farroupilha
Santa Maria, Brazil
marcos.oliveira@iffarroupilha.edu.br

André Rauber Du Bois
PPGC - Universidade Federal de Pelotas
Pelotas, Brazil
dubois@inf.ufpel.edu.br

Gerson Geraldo H. Cavalheiro
PPGC - Universidade Federal de Pelotas
Pelotas, Brazil
gerson.cavalheiro@inf.ufpel.edu.br

ABSTRACT

This paper presents a mapping of the use of the OpenMP API in open-source projects hosted on GitHub. A case study collected data on using OpenMP directives following a mining process in repositories developed in C and C++ with OpenMP. The study analyzed the use of OpenMP resources to support critical sections and exploit data parallelism. The analysis reveals usage patterns of such resources that limit the parallelization potential obtainable with this programming tool. The results contribute to understanding how the community uses this tool and provide insights for developing better practices and tools to support the use of OpenMP and improve the teaching of parallel processing. All software artifacts developed for this study are available to foster reproducibility and further research.

CCS CONCEPTS

• **Software and its engineering** → *Software libraries and repositories*; **Parallel programming languages**;

KEYWORDS

OpenMP, Parallel Programming, Minning Software Repositories, GitHub, Reprodutibility

1 INTRODUCTION

Mining software repositories is crucial for understanding and improving review processes in open-source projects. It allows extracting data on people, processes, and products, enabling analysis and enhancement of quality and efficiency. This data, coupled with techniques and tools related to code review, can advance knowledge and optimize software design processes [13].

In this context, mining software repositories offers a valuable approach to understanding how projects utilize specific technologies and paradigms, such as concurrent programming with OpenMP [5]. OpenMP is an API that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran, providing a straightforward interface for parallel application development. It is also an essential tool incorporated into undergraduate courses, providing students with a practical understanding of parallel programming in individual and distributed systems [11]. While reference documents exist to guide the use of OpenMP directives (commands for parallelizing program execution), these directives are not always employed correctly or efficiently. Analyzing OpenMP directive usage in real-world projects can shed light on programmer practices and inform strategies for improving OpenMP's effectiveness.

The practice of mining repositories allows for the extraction of meaningful code-related data, offering insights into the dynamics of open-source projects. By analyzing this information, we can identify collaboration patterns, assess process effectiveness, and enhance project quality. Beyond social aspects like team collaboration and information flow, repository mining also uncovers technical details, such as file organization and common issues faced by the community. Thus, it is a crucial tool for understanding and optimizing concurrent programming practices.

This paper presents a mapping of OpenMP usage in open-source projects available on GitHub, a web-based platform for version control and collaborative software development. The primary goal is to survey the current state of OpenMP adoption in software projects, along with aspects inherent to concurrent programming. The main contributions of this research are the quantification of different OpenMP directive usage in public GitHub repositories and the software artifacts for mining and analyzing case studies.

2 RELATED WORKS

Mining Software Repositories (MSR) is a complex task with many challenges and can be conducted for different purposes [8]. 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 [6]. Notable papers related to these approaches include [12], which addresses challenges in using the GitHub REST API for MSR studies, such as API limitations, language misclassification, and the inclusion of non-software artifacts. The authors developed *G-Repo*, a tool that assists researchers in creating and cleaning datasets by querying GitHub, cloning repositories, checking programming languages, and detecting the spoken language of repositories. G-Repo is shown to be an effective tool for overcoming common hurdles in data gathering and preparation, making it valuable for general MSR work.

In addition to *G-Repo*, other tools like *git2net* [7] and the Boa dataset [1] contribute to MSR research. *git2net* is a Python tool for extracting fine-grained and time-stamped co-editing networks from large Git repositories, offering insights into collaboration patterns and developer effort allocation not captured by coarser-grained methods. The Boa dataset, curated for Data Science software developed using Python, provides a valuable resource for understanding development practices in data-intensive Python projects. Furthermore, [9] presents *reaper*, a tool that automatically evaluates repositories to predict whether they contain engineered software projects, achieving higher recall than stargazer-based methods.

While these contributions advance the field of MSR, there remains a gap in specialized research focused on concurrent programming and the use of tools like OpenMP.

3 MINING OPENMP PUBLIC REPOSITORIES

A Python script was developed to automate mining on GitHub using the PyGithub library [10]. The script interacts with the GitHub REST API and uses the Pandas library[1] for output generation. It clones C or C++ repositories tagged with "openmp." After authenticating via an access token, it searches for repositories sorted by stars, indicating popularity.

The script handles GitHub's paginated search results (30 results per page) and dynamic rate-limiting by waiting 15 seconds when the request limit is reached. Once repositories are cloned, files containing the `#pragma omp` directive are retained, while others are removed. The script also collects repository data, including stars, forks, number of files, and collaborators. The output includes a directory for each repository, named by its unique GitHub ID, containing files with OpenMP directives. A separate file lists repository details: ID, name, description, stars, forks, URL, languages used, and filenames with OpenMP calls.

Data extraction was conducted in May 2023. The script was executed to find repositories tagged with "openmp" and developed in both C and C++. The search returned 1,312 repositories (592 developed in C, 720 in C++) with a total of 13,343 files containing OpenMP directives. Table 1 presents the repositories' stratification based on stars, forks, files, and collaborators. After mining, files containing OpenMP were prepared for processing. The `dos2unix` utility was used to standardize them to Unix format, followed by the stream editor `sed` to remove double-spacing characters from lines with "#pragma omp" for easier pattern identification.

The mined data reveals that most repositories have up to 10 collaborators, besides the owner. C++ repositories generally have more stars, particularly in the (10-100] and (100-1,000] ranges. Two repositories exceed 1,000 stars: one in C with 7,761 stars and 1,878 forks, and one in C++ with 3,159 stars and 883 forks. Most repositories have 10 stars and forks or fewer.

This higher star rating correlates with the number of forks, consistent with findings by [3, 4]. A Pearson's r test shows a strong positive correlation between stars and forks ($r(1310) = 0.9872$, $p \leq 0.001$). Excluding two outlier repositories, the correlation remains strong ($r(1308) = 0.8484$, $p \leq 0.001$). No other metric combinations showed significant correlation. Thus, stars and forks are correlated, but no other significant relationships between metrics were found.

The file preparation stage also revealed that some repositories were incomplete projects or small experiments, but they were not discarded because the methodology used in this study did not include such filtering.

4 ANALYSIS OF MINING RESULTS

The main tools for extracting code information were `sed` for file preparation and `awk` for identifying directive use cases. Only files with the extensions `.c`, `.h`, `.C`, `.H`, `.cpp`, `.hpp`, `.cxx`, `.hxx`, and `.inl` were considered. Repository files were initially adjusted to simplify pattern identification in directive use. Leading whitespace and tabs were removed, with OpenMP sentinels `#pragma omp` placed at the beginning of lines. Lines starting with an OpenMP sentinel and ending with an escape character ("\") were concatenated with the following line, with the escape character removed. Finally, all lines containing `#pragma omp` were standardized to use a single space to separate words. Scripts were then developed to identify patterns in OpenMP directive use for each case considered in the study. The artifacts and data used in this study are available at https://github.com/Nheeboranga/OpenMPSnapshot, enabling verification of the information and promoting reproducibility. This repository includes the Python script for mining, `awk` scripts for pattern searching, instructions for using these scripts, as well as the `grep` and `sed` command lines employed.

[1] Pandas library, available at https://pandas.pydata.org/, accessed on May 27, 2024.

Table 1: Frequency of metrics by range of values.

Metric	C and C++					C++					C				
	0	(1-10]	(10-100]	(100-1,000]	> 1,000	0	(1-10]	(10-100]	(100-1,000]	> 1,000	0	(1-10]	(10-100]	(100-1,000]	> 1,000
Stars	653	515	111	31	2	333	278	79	29	1	320	237	32	2	1
Forks	906	339	56	10	1	471	197	43	9	0	435	142	13	1	1
Files	0	851	460	1	0	0	426	293	1	0	0	425	167	0	0
Collaborators	42	1,241	26	3	0	22	674	22	2	0	20	567	4	1	0

Table 2: Usage Percentage by Category with Detailed 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.

4.1 Code inspection

The data show that repositories using OpenMP with C++ as the primary language are about 20% more numerous than those using C, and they also have higher numbers of stars and forks. Originally developed for C (and Fortran), OpenMP has always been usable with C++, though with C-like limitations. Starting with OpenMP 3.0, support for C++ features such as container manipulation and iterators was added. The latest version, OpenMP 5.2, extends this support to modern C++ features like lambda expressions and `std::array` references, enhancing integration with idiomatic C++ code [2]. The complexity of handling class instances and using iterators in parallel loops exemplifies this evolution.

Given the specifics of using C++ with OpenMP, a manual analysis of 10 repositories was conducted, with five having 300-900 stars and five having 90-150 stars to avoid extremes in project reputation[2]. The investigation focused on exception handling, class instances in `private`, `first/lastprivate`, and `reduction` clauses, and parallel loops controlled by iterators. The analysis revealed that only one repository used all the investigated features and had the highest number of stars, while another, with the second-highest number of stars, used object instances in the clauses. Additionally, four repositories utilized exception handling mechanisms.

[2]Repository IDs: 81815495, 94275048, 6987353, 38410417, 40821917, 73826981, 84174010, 58775556, 69450880, 322989201.

Table 2 presents the proportion of directives within each category, highlighting the extensive use of directives for parallelism exploration in loops. Additionally, while concurrency exposure clauses provide directives for parameterizing task construction and address space sharing, the `critical` and `atomic` directives are also notably used. The table also reveals a growing trend among developers to embrace newer OpenMP features, particularly offloading computations to specialized hardware, and leveraging vectorization capabilities. However, directives related to data privacy and execution control remain underutilized, suggesting potential areas for further education and tool development within the OpenMP ecosystem.

4.2 Dealing with shared data

Critical sections ensure consistency in accessing shared data, with OpenMP providing `atomic` and `critical` directives for this purpose. The `atomic` directive, used 2,485 times, allows atomic operations on data types like integers or pointers without the need for additional synchronization mechanisms. It is implemented with hardware instructions such as *fetch-and-add* and *compare-and-swap*. The most frequently used clause is `update`, applied in 1,193 cases, supporting atomic read-write access. Other clauses like `write` and `read` also followed the expected patterns, though some complex constructs involved non-guaranteed atomic access. The `capture` clause, introduced in OpenMP 5.1, was rarely used, and the newer `compare` and `hint` clauses were only found in one repository each, mainly for feature testing.

4.2.1 Usage of the `critical` *Directive.* The `critical` directive, used 1,414 times, with 293 instances having labels, showed potential for optimization. Data indicated that 730 critical sections could have utilized `atomic` instead, with 508 using `read` or `write` clauses and 222 using `update`. Long critical sections often involved loops, `if` statements, or data collections. The use of `critical` could be improved to reduce performance impacts by minimizing synchronization granularity and replacing some sections with `atomic` to decrease contention.

4.2.2 Parallel loops. Parallel loops, facilitated by directives such as `parallelfor` and `simd`, are the most exploited form

of parallelism in the analyzed repositories. Out of 16,468 instances of these loops, 8,306 employed the `schedule` clause, with `static` and `dynamic` being the most common scheduling strategies. The `dynamic` strategy showed the highest rate of imbalance in computational costs, while `static` was frequently used by default. The analysis also identified frequent occurrences of nested `parallelfor` directives and explored the use of the `simd` directive for vector parallelism, indicating a broad adoption but with room for improved implementation and optimization.

4.3 Limitations and Threats to Validity

This study presents a snapshot analysis of OpenMP usage in GitHub repositories, focusing on directive application without in-depth code quality or performance analysis. Limitations include reliance on repository tags and potential inclusion of test or academic repositories. The pattern-based identification approach, while useful, may miss or misinterpret some code snippets. Manual inspections were performed to ensure adequacy, but they are subject to interpretation and error. Despite these limitations, the methodology's reproducibility offers a valuable framework for assessing OpenMP utilization.

5 CONCLUSION

This work presented a snapshot of OpenMP usage in C and C++ projects on GitHub, based on data from 1,312 repositories. The analysis of for-loop parallelization and data sharing reveals opportunities for optimization, such as improving scheduling strategies for parallel loops and reducing the use of critical sections. The study deepens the understanding of how OpenMP is used in real-world projects, providing insights for best practices, educational resources, and tools to support OpenMP development. This report serves as a valuable case study for mining public repositories and for designing parallel programming curricula. Despite the limitations of repository mining, the methodology and artifacts developed offer a strong basis for future research. The publicly available resources foster reproducibility and further exploration, advancing parallel programming practices and education.

ACKNOWLEDGMENTS

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

REFERENCES

[1] Sumon Biswas, Md Johirul Islam, Yijia Huang, and Hridesh Rajan. 2019. Boa Meets Python: A Boa Dataset of Data Science Software in Python Language. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 577–581. https://doi.org/10.1109/MSR.2019.00086

[2] OpenMP Architecture Review Board. 2021. *OpenMP Application Programming Interface Specification 5.2*. https://www.openmp.org/specifications/

[3] Hudson Borges and Marco Tulio Valente. 2018. What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform. *Journal of Systems and Software* 146 (2018), 112–129. https://doi.org/10.1016/j.jss.2018.09.016

[4] Hudson Silva Borges, André C. Hora, and Marco Túlio Valente. 2016. Understanding the Factors That Impact the Popularity of GitHub Repositories. *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2016), 334–344. https://api.semanticscholar.org/CorpusID:184135

[5] Rohit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. 2001. Parallel Programming in OpenMP. (2001).

[6] Mário André de F. Farias, Renato Novais, Methanias Colaço Júnior, Luís Paulo da Silva Carvalho, Manoel Mendonça, and Rodrigo Oliveira Spínola. 2016. A Systematic Mapping Study on Mining Software Repositories. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing* (Pisa, Italy) *(SAC '16)*. Association for Computing Machinery, New York, NY, USA, 1472–1479. https://doi.org/10.1145/2851613.2851786

[7] Christoph Gote, Ingo Scholtes, and Frank Schweitzer. 2019. git2net - Mining Time-Stamped Co-Editing Networks from Large git Repositories. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 433–444. https://doi.org/10.1109/MSR.2019.00070

[8] Victor A. Luzgin and Ivan I. Kholod. 2020. Overview of Mining Software Repositories. In *2020 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*. 400–404. https://doi.org/10.1109/EIConRus49466.2020.9039225

[9] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. 2017. Curating GitHub for engineered software projects. *Empirical Software Engineering* 22 (2017). https://doi.org/10.1007/s10664-017-9512-6

[10] PyGithub. 2023. PyGithub documentation. https://pygithub.readthedocs.io/en/stable/introduction.html Disponível em https://pygithub.readthedocs.io/en/stable/introduction.html, versão v3.

[11] Rajendra K. Raj, Carol J. Romanowski, John Impagliazzo, Sherif G. Aly, Brett A. Becker, Juan Chen, Sheikh Ghafoor, Nasser Giacaman, Steven I. Gordon, Cruz Izu, Shahram Rahimi, Michael P. Robson, and Neena Thota. 2020. High Performance Computing Education: Current Challenges and Future Directions. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education* (Trondheim, Norway) *(ITiCSE-WGR '20)*. Association for Computing Machinery, New York, NY, USA, 51–74. https://doi.org/10.1145/3437800.3439203

[12] Simone Romano, Maria Caulo, Matteo Buompastore, Leonardo Guerra, Anas Mounsif, Michele Telesca, Maria Teresa Baldassarre, and Giuseppe Scanniello. 2021. G-Repo: a Tool to Support MSR Studies on GitHub. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 551–555. https://doi.org/10.1109/SANER50967.2021.00064

[13] Xin Yang, Raula Gaikovina Kula, Norihiro Yoshida, and Hajimu Iida. 2016. Mining the modern code review repositories: a dataset of people, process and product. In *Proceedings of the 13th International Conference on Mining Software Repositories* (Austin, Texas) *(MSR '16)*. Association for Computing Machinery, New York, NY, USA, 460–463. https://doi.org/10.1145/2901739.2903504