Quantifying OpenMP:
Statistical Insights into Usage and Adoption

Tal Kadosh[1,2], Niranjan Hasabnis[3], Timothy Mattson[3], Yuval Pinter[1] and Gal Oren[4,5]

[1]Department of Computer Science, Ben-Gurion University, Israel
[2]Israel Atomic Energy Commission
[3]Intel Labs, United States
[4]Scientific Computing Center, Nuclear Research Center – Negev, Israel
[5]Department of Computer Science, Technion – Israel Institute of Technology, Israel
talkad@post.bgu.ac.il, niranjan.hasabnis@intel.com,
timothy.g.mattson@intel.com, pintery@bgu.ac.il, galoren@cs.technion.ac.il

Abstract—In high-performance computing (HPC), the demand for efficient parallel programming models has grown dramatically since the end of Dennard Scaling and the subsequent move to multi-core CPUs. OpenMP stands out as a popular choice due to its simplicity and portability, offering a directive-driven approach for shared-memory parallel programming. Despite its wide adoption, however, there is a lack of comprehensive data on the actual usage of OpenMP constructs, hindering unbiased insights into its popularity and evolution.

This paper presents a statistical analysis of OpenMP usage and adoption trends based on a novel and extensive database, HPCORPUS, compiled from GitHub repositories containing C, C++, and Fortran code. The results reveal that OpenMP is the dominant parallel programming model, accounting for 45% of all analyzed parallel APIs. Furthermore, it has demonstrated steady and continuous growth in popularity over the past decade. Analyzing specific OpenMP constructs, the study provides in-depth insights into their usage patterns and preferences across the three languages. Notably, we found that while OpenMP has a strong "common core" of constructs in common usage (while the rest of the API is less used), there are new adoption trends as well, such as `simd` and `target` directives for accelerated computing and `task` for irregular parallelism.

Overall, this study sheds light on OpenMP's significance in HPC applications and provides valuable data for researchers and practitioners. It showcases OpenMP's versatility, evolving adoption, and relevance in contemporary parallel programming, underlining its continued role in HPC applications and beyond. These statistical insights are essential for making informed decisions about parallelization strategies and provide a foundation for further advancements in parallel programming models and techniques.

HPCORPUS, as well as the analysis scripts and raw results, are available at: https://github.com/Scientific-Computing-Lab-NRCN/HPCorpus

Index Terms—HPCorpus, BigQuery, GitHub, C, C++, Fortran, OpenMP, MPI, OpenCL, CUDA, TBB, Cilk, OpenACC, SYCL

I. INTRODUCTION

With the end of Dennard Scaling [1], multicore CPUs sharing a cache-coherent address space are ubiquitous. To exploit the parallelism available from multicore systems, programmers use multithreaded programming models. Programming models that support multithreaded execution include pThreads [2] for low-level and OS-level operations, TBB [3] or Cilk [4] for task-level parallelism in C++, and OpenMP [5] for directive-driven parallelism.

Despite the popularity of multithreaded models, little empirical data is available to assess the actual usage of the various programming constructs from these models. While anecdotal data and feedback from user-support teams at supercomputing centers exist [6], a large-scale analysis has yet to be conducted. In this paper, we perform a statistical analysis of repositories from GitHub to study the usage of parallel programming models. Our analysis reveals that OpenMP is the dominant programming model for writing multithreaded applications. We also go inside applications to gather usage data on specific OpenMP constructs. Finally, we consider the evolution of OpenMP and the adoption of newly included constructs as OpenMP Specifications are released.

II. OPENMP FUNDAMENTALS VS. OTHER PARALLEL PROGRAMMING APIS

In this section, we provide a brief overview of OpenMP and other parallel programming APIs. OpenMP [7] defines a simple and portable approach to shared-memory parallel programming. OpenMP makes parallel programming more accessible by offering a directive-based approach, where directives inserted into the code guide the compiler as it generates parallel code. These directives provide high-level abstractions to specify parts of a code to execute in parallel. OpenMP uses the fork-join model of parallelism, where a single thread (the PRIMARY thread) on encountering a parallel directive forks a team of threads, each of which executes the code in a parallel region independently. Synchronization constructs, such as barriers, coordinate multithreaded execution, while shared variables facilitate data sharing among threads. OpenMP emphasizes portability with a standardized API that can be used across different platforms, hardware, and programming languages. These features make it easier to write parallel code that can be compiled and executed on systems supporting OpenMP.

```
1  WITH selected_repos as (
2  SELECT f.id, f.repo_name as repo_name, f.ref as
   ↪   ref, f.path as path
3  FROM `bigquery-public-data.github_repos.files` as
   ↪   f
4  JOIN `bigquery-public-data.github_repos.licenses`
   ↪   as l on l.repo_name = f.repo_name
5  ),
6  deduped_files as (
7  SELECT f.id, MIN(f.repo_name) as repo_name,
   ↪   MIN(f.ref) as ref, MIN(f.path) as path
8  FROM selected_repos as f
9  GROUP BY f.id
10 )
11 SELECT
12 f.repo_name, f.ref, f.path, c.copies, c.content,
13 FROM deduped_files as f
14 JOIN `bigquery-public-data.github_repos.contents`
   ↪   as c on f.id = c.id
15 WHERE
16  NOT c.binary
17  AND (f.path like '%.c' OR f.path like '%.cpp' OR
   ↪   f.path like '%.f' OR f.path like '%.f90' OR
   ↪   f.path like '%.f95')
```

Fig. 1. HPCORPUS data acquisition from Google's BigQuery.

	Repos (#)	Size (GB)	Files (#)	Functions (#)
C	144,522	46.23	4,552,736	87,817,591
C++	150,481	26.16	4,735,196	68,233,984
Fortran	3,683	0.68	138,552	359,272

TABLE I
TOTAL NUMBER OF REPOSITORIES IN HPCORPUS BY LANGUAGE.
NOTE: REPOSITORIES MAY USE MULTIPLE LANGUAGES, AND 5.5K
REPOSITORIES IN HPCORPUS CONTAINED NO CODE.

OpenMP differs from other parallel APIs in several ways. Unlike Cilk and TBB, which focus on task-based parallelism, OpenMP is more general and addresses loop-level parallelism, task-parallelism, and general, multi-threaded parallelism through explicit thread-level programming. OpenMP differs from MPI [8], which is designed for distributed-memory parallelism by targeting shared-memory parallelism within a single program. Compared to CUDA [9] and OpenCL [10], which are geared towards throughput-optimized accelerators (and in the case of CUDA, a specific vendor), OpenMP provides a portable solution that works across different platforms, including general-purpose CPUs, accelerators [11], GPUs [12]–[14], and FPGAs [15]. Additionally, OpenMP's compatibility with multiple programming languages (specifically C, C++, Fortran, and partially Python [16], [17]) sets it apart from other GPU programming languages such as SYCL [18] (C++ only) and OpenCL (C and C++). OpenACC [19] is similar to OpenMP in its directive-based approach, but it specifically targets GPUs only. OpenACC emphasizes descriptive semantics, meaning constructs describe *what* should be accomplished, not *how* it is done. OpenMP, on the other hand, emphasizes *prescriptive* semantics, which allows the programmer to control how code maps onto a system explicitly. With the `loop`-construct added in OpenMP version 5.0, however, OpenMP is moving to include descriptive semantics, thereby supporting OpenACC's approach for programmers who prefer yielding more control to the compiler.

III. HPCORPUS: A NOVEL DATABASE OF HPC CODE FROM GITHUB

To study the usage of parallel programming APIs, we compiled a novel database called HPCORPUS. It collects C, C++, and Fortran codes from every publicly visible GitHub repository that was accessible via BigQuery with the suffixes c, cpp, f,

f90, and f95 (Figure 1, line 17). These languages are widely recognized as the dominant languages in HPC [20]–[28]. In the data acquisition process, we followed [29], [30] but selected the C, C++, and Fortran files within those projects in an unrestricted way. We structured HPCORPUS with a JSON format using the script presented in Figure 1.[1]
A breakdown of the repository statistics is presented in Table I. Fortran has far less data than the C and C++ sub-corpora. The C sub-corpus is almost twice as large as the C++ sub-corpus. HPCORPUS aims to achieve two main objectives: first, to facilitate statistical analyses on the popularity and usage of various parallel programming APIs, and second, to serve as a robust training resource for the next generation of Large Language Models (LLMs) [32] designed to automate complex high-performance tasks [33]. Of these tasks, parallelization is most notable [34]–[38], since current rule-based compilers for such purposes are not optimal or robust [39], [40]. By offering a diverse and vast array of code snippets, HPCORPUS has the potential to significantly enhance research and development efforts in advanced AI models that address intricate parallelization challenges.

IV. ANALYSIS

We divide our analysis into two parts. First, we compare the popularity of OpenMP as a function of total usage and usage over time relative to other parallel programming APIs. Next, we present specific statistics about OpenMP constructs and the extent to which users adopt them.

A. OpenMP vs. Other Parallel Programming APIs

In this section, we elaborate on the usage statistics of OpenMP vs. other common parallel programming APIs, such as MPI, CUDA, OpenCL, TBB, Cilk, OpenACC, and SYCL (Figure 2[2]). We find that not only is OpenMP by far the most popular API, but it accounts for almost half (45%) of all the repositories using parallel programming APIs in this analysis. Over the last decade (since 2013), there has been steady growth in new OpenMP repositories. When combined with absolute usage numbers for OpenMP, it maintains a clear popularity advantage relative to other APIs (Figure 3).[3] We also observe that MPI, as a distributed-memory parallelization

---

[1]Although the script does not contain code to obtain repository timestamps, we did collect them separately using perceval [31]. We store these timestamps separately to keep HPCORPUS to a more reasonable size.

[2]See code @ aggregate_paradigms.

[3]See code @ get_paradigms_per_year.

API, and OpenMP, as a shared-memory parallelization API, are well integrated over time (Figure 4),[4] indicating the growing need for MPI+X as parallel clusters with multicore nodes grow in scale [41].
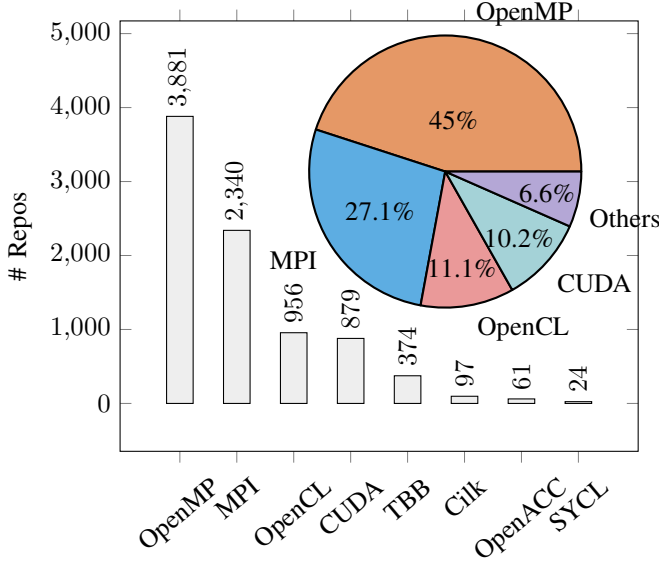

Fig. 2. Parallel programming API usage in HPCORPUS.

B. Breakdown of OpenMP Directives Usage

In this section, we discuss the detailed usage breakdown of elements from OpenMP as presented in Figure 9[5]. This data breaks down usage statistics by language (C, C++, and Fortran) and by the version of the OpenMP specification (versions 2.5 to 5.2).[6] The counts measure each clause's usage, with a higher count suggesting more prevalent usage in C, C++, and Fortran codes from HPCORPUS.

Generally, over the years, we see that while new directives were well received and adopted (Figure 6),[7] the *common core* of OpenMP v2.5-3, specifically OpenMP's `parallel for` constructs (Figure 7),[8] dominate the usage. In addition to this analysis, we also analyze HPCORPUS for the growth in complexity of OpenMP specifications (Figure 5).[9]

`for` loops: `for` loops are commonly-used iterative control structures in programming. In C codes from HPCORPUS, there are 21,822,609 occurrences of for loops, followed by 19,841,237 in C++ codes and 1,203,773 (`do` loops) in Fortran. Fortran leads in parallelizing a proportion of `for` loops with OpenMP, with 2.2% of all loops parallelized, compared to 0.54% in C++ and 0.16% in C.

---

[4]See code @ get_omp_mpi_usage.

[5]See code @ aggregate_versions.

[6]For complete OpenMP specification and the differences between versions, see https://www.openmp.org/spec-html/5.0/openmp.html and [42].

[7]See code @ get_version_per_year.

[8]See code @ get_loops.

[9]We refer the reader to a previous, well-known graph, which measured the growth in complexity by the specification page count (for example, reference [7], Figure 3.1, page 37.)
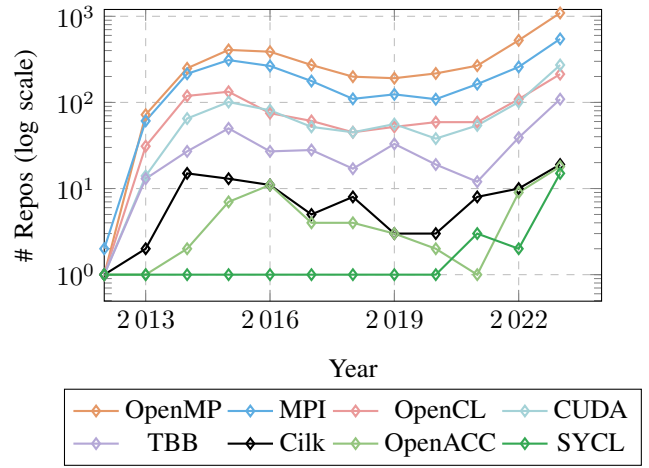

Fig. 3. Parallel programming API usage trends in HPCORPUS over the last decade.

Scheduling: Static scheduling is the preferred kind of schedule across all languages, with C codes from HPCORPUS having 8,705 occurrences, followed by C++ codes with 4,897 and Fortran codes with 2,013. Dynamic scheduling is less prevalent but still used more than advanced methods such as `guided`, `runtime`, and `auto`.

Data-sharing: C++ codes from HPCORPUS demonstrate higher usage of data-sharing constructs (`private`, `firstprivate`, `lastprivate`, `shared`, and `reduction`) compared to C and Fortran. However, the `nowait` construct, for eliminating synchronization barriers, is used less frequently in all languages.

Irregular Parallelism: In HPCORPUS, the `task` construct for explicit tasks and the `sections` construct for parallel sections are moderately prevalent. C++ codes from HPCORPUS have the highest counts with 4,169 `task` occurrences and 7,038 `sections` occurrences.

Vectorization: C++ codes from HPCORPUS have significantly higher usage of the `simd` directive for enabling vectorization, with 54,557 occurrences. In comparison, C and Fortran codes have 9,942 and 1,199 occurrences, respectively.

Offloading: The `target` construct for offloading computations to accelerators is prominently used in C++ codes in HPCORPUS, with 78,930 occurrences. C and Fortran codes, on the other hand, show 6,199 and 2,532 occurrences, resp.

Synchronization: The `barrier` construct for synchronization varies in usage, with C, C++, and Fortran codes from HPCORPUS having 2,825, 2,728, and 858 occurrences respectively. The `atomic` construct for atomic operations is moderately used, with C++, C, and Fortran codes, having 5,360, 3,601, and 2,005 occurrences, respectively. Other constructs such as `flush`, `single`, and `master` have relatively lower counts across all languages in HPCORPUS.
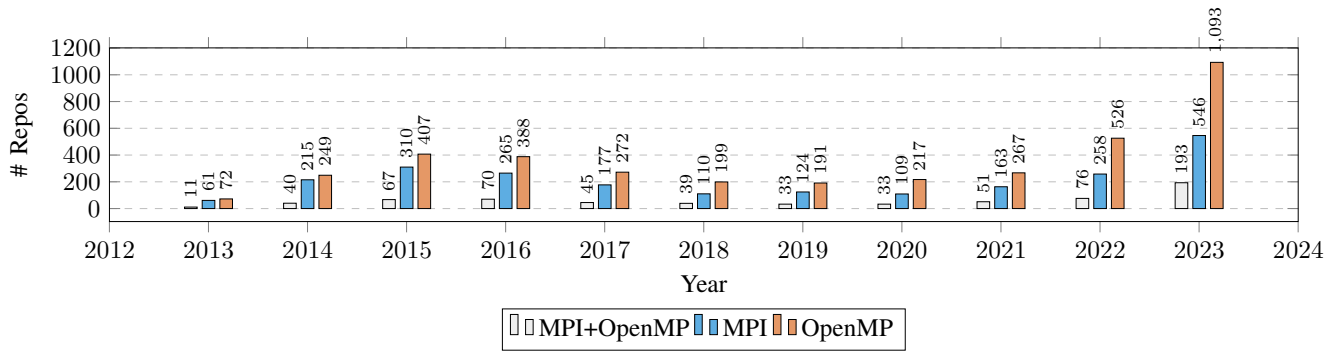
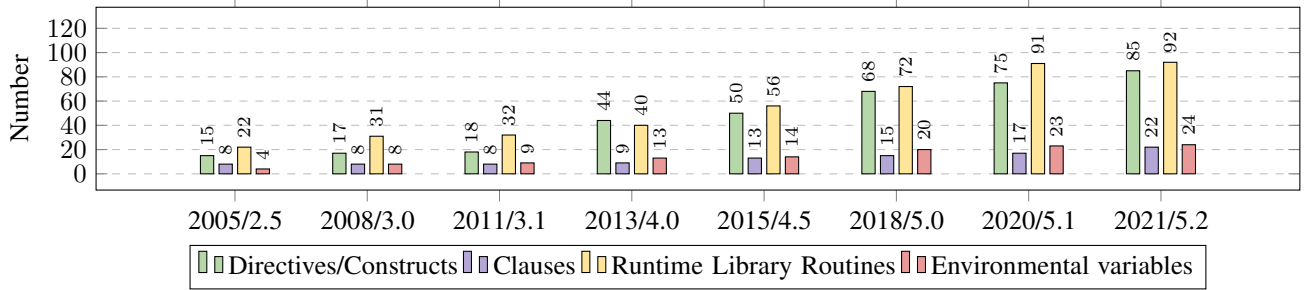Fig. 4. MPI, OpenMP, and MPI+OpenMP usage in HPCORPUS over the last decade.

Fig. 5. Growth in OpenMP complexity over time. (The X-axis shows OpenMP specifications 2.5 to 5.2 and the years they were released. The Y-axis shows the absolute number of various OpenMP elements in each specification.)
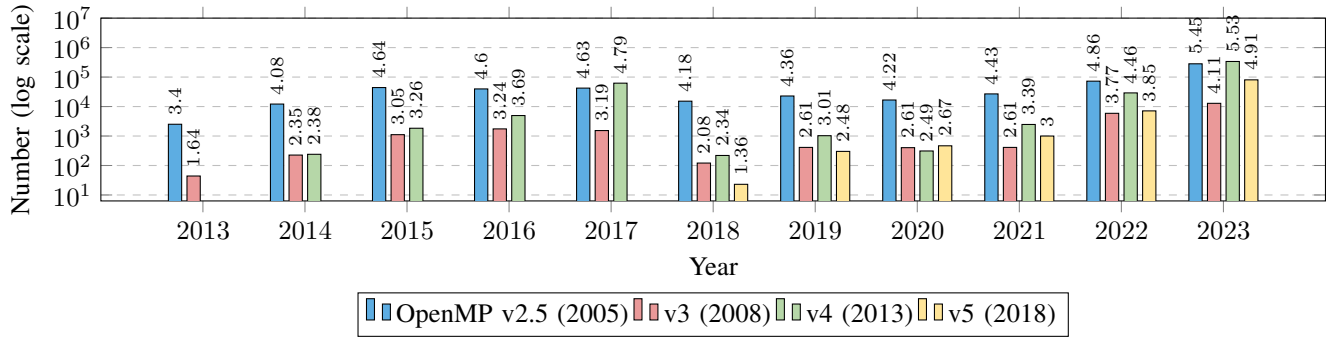

Fig. 6. Absolute number of occurrences of various OpenMP elements for each version of OpenMP in HPCORPUS over the the last decade. Bar labels represent powers of 10 (e.g., for 2013, various elements of OpenMP v2.5 had 2508 occurrences (i.e., $10^{3.4}$)). For a detailed breakdown of the number of occurrences of OpenMP elements organized by the version of OpenMP where they were introduced, see Figure 9.
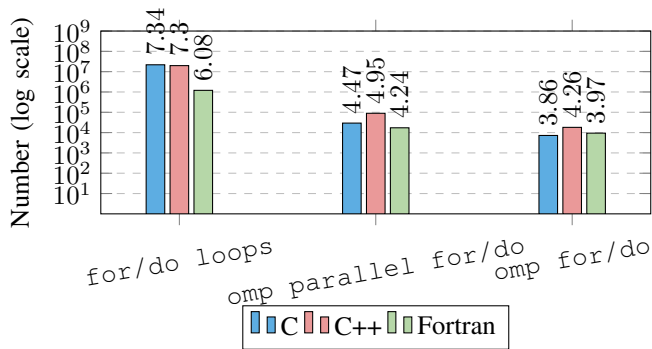

Fig. 7. Total numbers of `for/do` loops, OpenMP combined parallel worksharing loops, and OpenMP worksharing loops in HPCORPUS.
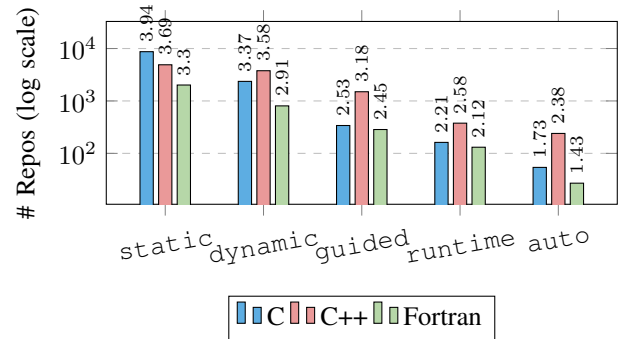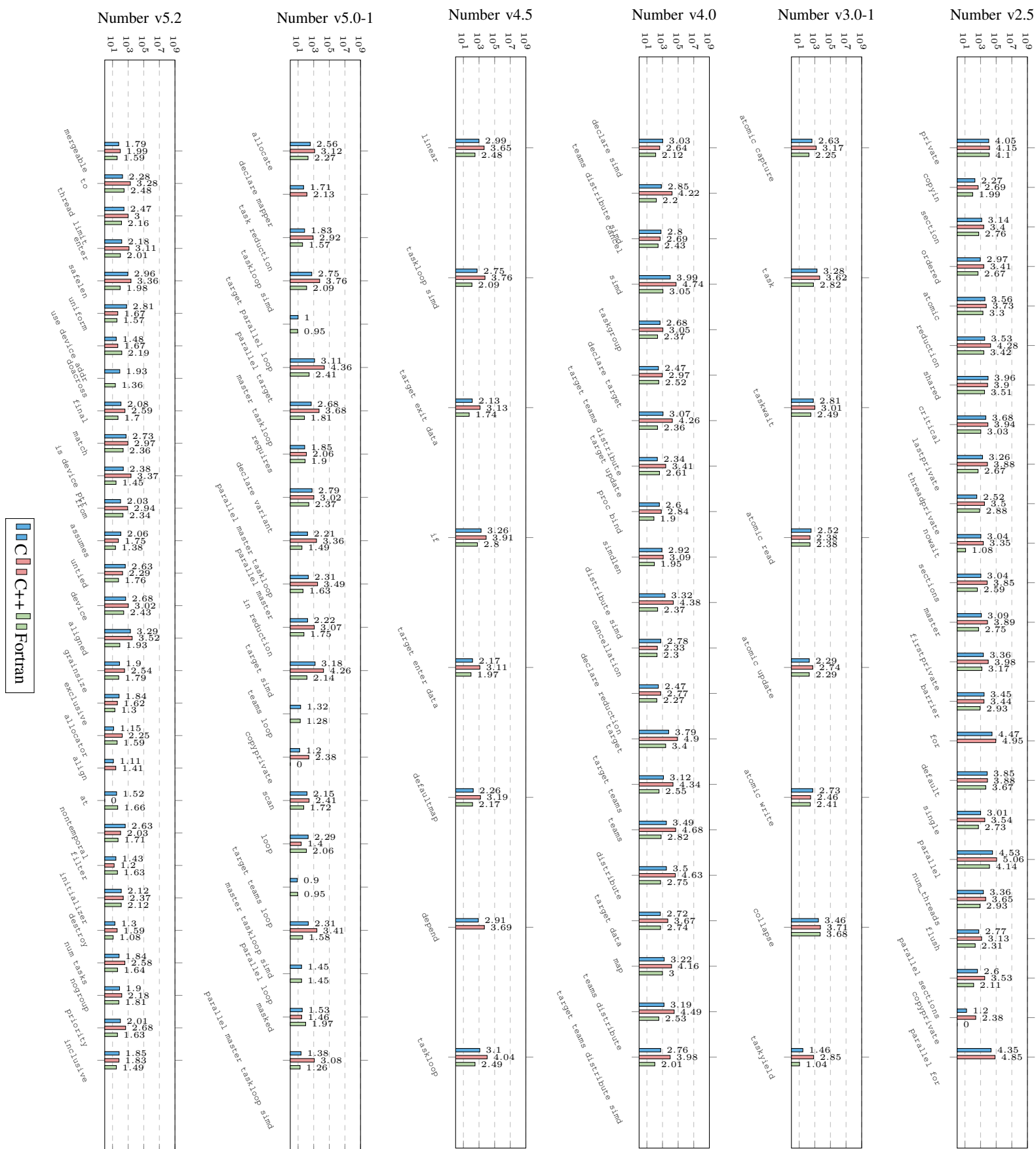

Fig. 8. Schedule kinds found in HPCORPUS.

Fig. 9. Absolute number of different OpenMP clauses (from OpenMP v2.5 - v5.2) found in HPCORPUS (Note: Y-axis is a log scale.)

V. DISCUSSION

These results show that OpenMP is the dominant parallel programming model for C, C++, and Fortran in the publicly visible repositories in GitHub. The usage of items within OpenMP, however, is not evenly spread between the different versions of the OpenMP specification. These results largely confirm the subset of 21 items from the OpenMP Common Core [7] with one notable exception. The Common Core omitted the `lastprivate` clause. It was felt that this clause was rarely used. Clearly, that is not the case. Future updates of the Common Core should include it.

While the Common Core is an important simplification of OpenMP, programmers often gravitate towards even greater simplicity. The third most common construct used in OpenMP is `parallel for`. This supports a style of programming where you find the time-consuming loops and then parallelize them with a simple `parallel for` directive. Much greater performance is available by explicitly managing parallel regions inside of which are worksharing loops (i.e., using the `for` construct). The popularity of `parallel for` is a reminder to programming model designers; people often seek *good-enough* performance, not ultimate performance.

Finally, we wish to comment on the insights from the HP-CORPUS data set on the adoption of new releases of OpenMP. These new releases are not ignored. New features of OpenMP find programmers that use them. The fact of the matter is, however, that most OpenMP programmers work with items from OpenMP 4.0 or earlier (note: OpenMP 4.0 came out over 10 years ago). The adoption of new features is uneven across the programming community.

VI. CONCLUSIONS AND FUTURE OUTLOOK

The central conclusion of this paper is that OpenMP is far more popular than we anticipated. We expected the popularity of programming models in HPCORPUS to mirror that found in workloads at various supercomputer centers. For example, a 2015 talk at OpenMPcon [6] reported that over 90% of applications programs running at NERSC used MPI, while only 40% used OpenMP.

We found, however, that for code in GitHub, OpenMP is by far the most popular parallel programming model for C, C++, and Fortran, with 45% of repositories containing OpenMP code while only 27% containing MPI. This result is so surprising that we can't help but wonder if we did something wrong. We have made HPCORPUS available and shared the query used to generate it. We are eager for peer review to help ensure that our conclusions are correct.

In retrospect, however, perhaps we should not have been so surprised. Programs running at supercomputer centers are designed to run on large scalable machines. These programs obviously need a distributed memory API such as MPI. On GitHub, however, you would expect to find distributed memory programs but also multithreaded programs for a wide range of multicore CPUs. In other words, the scope for OpenMP extends from edge devices to laptops, to servers, to massive supercomputers. The number of systems that benefit from OpenMP dwarfs the number of large-scale clusters or even GPU-based systems; hence it is not surprising that the data from HPCORPUS shows so much usage of OpenMP.

When you work with an MPI program, you know it. The program is launched as an MPI program (with `mpiexec` or `runmpi`). The code is, in most cases, structured around a single program multiple data (SPMD) pattern, with a copy of the program running on each node of the system. You know you are running an MPI job. With OpenMP, it is easy to use it and not even know it. A library routine buried deep in your code could use OpenMP. It's much lower overhead to experiment with OpenMP compared to MPI. Hence, the use of MPI stands out, and frankly, it is easy to forget about OpenMP. This factor could also play a role in expectations of MPI usage relative to OpenMP.

There is much work yet to do with HPCORPUS. To drive the development of new programming models, we need to understand how programming models are used. To do this we need to move beyond counts for the different constructs and explore the different ways they are combined into distinct design patterns of parallel programming [43]. From these patterns, we can better understand the cognitive issues of programmers working with different parallel programming models and guide how they should evolve.

Our interest in HPCORPUS goes well beyond studying which constructs are used in different parallel programming models. HPCORPUS can be used to train LLMs to support AI solutions for machine programming. Even with the amazing results from generative AI applied to programming, we are in the early days of this technology. We are interested in training LLMs to address different programming problems. While the "black box" that is current AI technology is fascinating, we see the line of research growing out of HPCORPUS going much further into models that combine neural networks with symbolic systems to reason about correctness while working with high-level structures based on fundamental design patterns of parallel programming. Only by combining neural networks with symbolic reasoning will we be able to crack the machine programming problem.

In closing, we call on others to carry out studies similar to the one described in this paper. For example, other than the talk about workloads at NERSC [6], we were unable to find detailed studies of programming model usage at major supercomputing centers. We focused on C, C++, and Fortran, but it would be interesting to repeat this work for Python, Rust, and Julia. Code repositories have become the standard way to manage complex software projects. The data is "out there". We should learn what programmers are actually using and then drive the evolution of programming models based on hard data, not anecdotes.

REFERENCES

[1] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th annual international symposium on Computer architecture*, 2011.

[2] D. Buttlar, J. Farrell, and B. Nichols, *Pthreads programming: A POSIX standard for better multiprocessing.* " O'Reilly Media, Inc.", 1996.

[3] A. Kukanov and M. J. Voss, "The Foundations for Scalable Multi-core Software in Intel Threading Building Blocks." *Intel Technology Journal*, vol. 11, no. 4, 2007.

[4] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *ACM SigPlan Notices*, vol. 30, no. 8, 1995.

[5] T. Mattson and R. Eigenmann, "OpenMP: An API for writing portable SMP application software," in *SuperComputing 99 Conference*, 1999.

[6] OpenMPCon, "Using OpenMP at NERSC," https://openmpcon.org/wp-content/uploads/openmpcon2015-helen-he-nersc.pdf, 2015.

[7] T. G. Mattson, Y. H. He, and A. E. Koniges, *The OpenMP common core: making OpenMP simple again.* MIT Press, 2019.

[8] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine *et al.*, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 11th European PVM/MPI Users' Group Meeting Budapest, Hungary, September 19-22, 2004. Proceedings 11*. Springer, 2004.

[9] D. Luebke, "CUDA: Scalable parallel programming for high-performance scientific computing," in *2008 5th IEEE international symposium on biomedical imaging: from nano to macro*. IEEE, 2008, pp. 836–838.

[10] A. Munshi, B. Gaster, T. G. Mattson, and D. Ginsburg, *OpenCL programming guide*. Pearson Education, 2011.

[11] D. Schmidl, T. Cramer, S. Wienke, C. Terboven, and M. S. Müller, "Assessing the performance of OpenMP programs on the Intel Xeon Phi," in *European Conference on Parallel Processing*. Springer, 2013.

[12] R. Van der Pas, E. Stotzer, and C. Terboven, *Using OpenMP# the next step: affinity, accelerators, tasking, and simd*. MIT press, 2017.

[13] T. Deakin and T. G. Mattson, *Programming Your GPU with OpenMP: Performance Portability for GPUs*. MIT Press, 2023.

[14] Y. Fridman, G. Tamir, and G. Oren, "Portability and Scalability of OpenMP Offloading on State-of-the-art Accelerators," *arXiv preprint arXiv:2304.04276*, 2023.

[15] F. Mayer, M. Knaust, and M. Philippsen, "OpenMP on FPGAs—a survey," in *OpenMP: Conquering the Full Hardware Spectrum: 15th International Workshop on OpenMP, IWOMP 2019, Auckland, New Zealand, September 11–13, 2019, Proceedings 15*. Springer, 2019.

[16] T. A. Anderson and T. G. Mattson, "Multithreaded parallel Python through OpenMP support in Numba," in *Proc. 20th Python Sci. Conf.*, 2021.

[17] T. G. Mattson, T. A. Anderson, and G. Georgakoudis, "PyOMP: Multithreaded parallel programming in Python," *Computing in Science & Engineering*, vol. 23, no. 6, 2021.

[18] R. Reyes and V. Lomüller, "SYCL: Single-source C++ accelerator programming," in *Parallel Computing: On the Road to Exascale*. IOS Press, 2016.

[19] R. Farber, *Parallel programming with OpenACC*. Newnes, 2016.

[20] C. Guillen and R. Bader, "HPC Compilers https://www.admin-magazine.com/HPC/Articles/Selecting-Compilers-for-a-Superco 2017.

[21] L. Computing, "Compilers at LC "https://hpc.llnl.gov/software/development-environmen 2022.

[22] IBM, "IBM Compilers https://www.ibm.com/products/ibm-compilers," 2023.

[23] Arm, "Arm Compiler for Linux https://developer.arm.com/Tools%20and%20Software/A

[24] NVIDIA, "NVIDIA HPC SDK https://developer.nvidia.com/hpc-sdk."

[25] Intel, "Intel oneAPI HPC Toolkit https://www.intel.com/content/www/us/en/developer/to

[26] Clang, "CLANG COMPILER USER'S MANUAL "https://clang.llvm.org/docs/UsersManual.html."

[27] GNU, "GCC, the GNU Compiler Collection https://gcc.gnu.org."

[28] T. P. Group, "PGI Compilers https://www.pgroup.com/index.htm."

[29] M.-A. Lachaux, B. Roziere, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," *arXiv preprint arXiv:2006.03511*, 2020.

[30] M. Research, "Little guide to download source code data from Github using Google Big Query," https://github.com/facebookresearch/CodeGen/blob/2c9dbe52a92cbf2dfa4e702442f4745 2021.

[31] S. Dueñas, V. Cosentino, G. Robles, and J. M. Gonzalez-Barahona, "Perceval: software project data at your will," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, 2018.

[32] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 2022.

[33] L. Chen, P.-H. Lin, T. Vanderbruggen, C. Liao, M. Emani, and B. de Supinski, "LM4HPC: Towards Effective Language Model Application in High-Performance Computing," *arXiv preprint arXiv:2306.14979*, 2023.

[34] R. Harel, Y. Pinter, and G. Oren, "Learning to parallelize in a shared-memory environment with transformers," in *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, 2023.

[35] T. Kadosh, N. Schneider, N. Hasabnis, T. Mattson, Y. Pinter, and G. Oren, "Advising OpenMP Parallelization via a Graph-Based Approach with Transformers," *arXiv preprint arXiv:2305.11999*, 2023.

[36] N. Schneider, T. Kadosh, N. Hasabnis, T. Mattson, Y. Pinter, and G. Oren, "MPI-rical: Data-Driven MPI Distributed Parallelism Assistance with Transformers," *arXiv preprint arXiv:2305.09438*, 2023.

[37] W. F. Godoy, P. Valero-Lara, K. Teranishi, P. Balaprakash, and J. S. Vetter, "Evaluation of OpenAI Codex for HPC Parallel Programming Models Kernel Generation," *arXiv preprint arXiv:2306.15121*, 2023.

[38] D. Nichols, A. Marathe, H. Menon, T. Gamblin, and A. Bhatele, "Modeling Parallel Programs using Large Language Models," *arXiv preprint arXiv:2306.17281*, 2023.

[39] R. Harel, I. Mosseri, H. Levin, L.-o. Alon, M. Rusanovsky, and G. Oren, "Source-to-source parallelization compilers for scientific shared-memory multi-core and accelerated multiprocessing: analysis, pitfalls, enhancement and potential," *International Journal of Parallel Programming*, vol. 48, 2020.

[40] I. Mosseri, L.-o. Alon, R. Harel, and G. Oren, "ComPar: optimized multi-compiler for automatic OpenMP S2S parallelization," in *OpenMP: Portable Multi-Level Parallelism on Modern Systems: 16th International Workshop on OpenMP, IWOMP 2020, Austin, TX, USA, September 22–24, 2020, Proceedings 16*. Springer International Publishing, 2020.

[41] R. Thakur, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, T. Hoefler, S. Kumar, E. Lusk, and J. L. Träff, "MPI at Exascale," *Procceedings of SciDAC*, vol. 2, 2010.

[42] B. R. de Supinski, T. R. Scogland, A. Duran, M. Klemm, S. M. Bellido, S. L. Olivier, C. Terboven, and T. G. Mattson, "The ongoing evolution of OpenMP," *Proceedings of the IEEE*, vol. 106, no. 11, 2018.

[43] T. G. Mattson, B. A. Sanders, and B. L. Massingill, *Patterns for Parallel Programming*. Addison Wesley, 2004.

[44] R. I. Park, "NegevHPC Project," https://www.negevhpc.com, 2019.

[45] Intel, "Intel Developer Cloud," https://www.intel.com/content/www/us/en/developer/tool 2023.