MPIrigen: MPI Code Generation through Domain-Specific Language Models

Nadav Schneider
nadavsch@post.bgu.ac.il
Ben-Gurion University, IAEC
Israel

Niranjan Hasabnis
niranjan.hasabnis@intel.com
Intel Labs
United States

Vy A. Vo
vy.vo@intel.com
Intel Labs
United States

Tal Kadosh
talkad@post.bgu.ac.il
Ben-Gurion University, IAEC
Israel

Neva Krien
nevo.krien@gmail.com
Independent Researcher
Israel

Mihai Capotă
mihai.capota@intel.com
Intel Labs, United States
United States

Guy Tamir
guy.tamir@intel.com
Intel
United States

Ted Willke
ted.willke@intel.com
Intel Labs
United States

Nesreen Ahmed
nesreen.k.ahmed@intel.com
Intel Labs
United States

Yuval Pinter
pintery@bgu.ac.il
Ben-Gurion University
Israel

Timothy Mattson
tim@timmattson.com
Independent Researcher
United States

Gal Oren
galoren@cs.technion.ac.il
Technion, NRCN
Israel

ABSTRACT

The imperative need to scale computation across numerous nodes highlights the significance of efficient parallel computing, particularly in the realm of Message Passing Interface (MPI) integration. While MPI serves as a cornerstone for large-scale parallelism, its seamless integration into codebases, especially concerning domain decomposition, has proven challenging. Static tools aimed at addressing this challenge have exhibited limited effectiveness and scalability. On the other hand, contemporary language models designed for programming problems have demonstrated utility in parallel programming tasks such as OpenMP pragma generation. However, the challenging parallel programming task of generating MPI-based parallel programs has remained unexplored.

This study first investigates the performance of state-of-the-art language models in generating MPI-based parallel programs. Findings reveal that widely used models such as GPT-3.5 and PolyCoder (specialized multi-lingual code models) exhibit notable performance degradation when generating MPI-based programs compared to general-purpose programs. In contrast, domain-specific models such as MonoCoder, which are pre-trained on MPI-related programming languages of C and C++, outperform larger models. Subsequently, we introduce a dedicated downstream task of MPI-based program generation by fine-tuning MonoCoder on HPCorpusMPI. We call the resulting model as MPIrigen. We propose an innovative pre-processing for completion only after observing the whole code, thus enabling better completion with a wider context. Comparative analysis against GPT-3.5 zero-shot performance, using a novel HPC-oriented evaluation method, demonstrates that MPIrigen excels in generating accurate MPI functions calls.

The success of this tailored solution underscores the importance of domain-specific fine-tuning in optimizing language models for parallel computing code generation, paving the way for a new generation of automatic parallelization tools.

The sources of this work are available at our GitHub MPIri-gen repository.

KEYWORDS

MPI, Domain Decomposition, Transformer, LLM, AI, code generation

1 MPI: SOURCE-TO-SOURCE PARALLELIZATION WITH COMPILERS

Efforts in the domain of source-to-source automatic parallelization have primarily focused on transitioning from serial code to shared-memory parallelization, especially with OpenMP. Such approaches have relied on heuristics and rule-based methods (e.g., ComPar [23], Par4all [2, 3], and Cetus [4]) employing AST generation and data dependence algorithms. However, these methods often face challenges in handling diverse syntax and may yield sub-optimal results, as noted in several studies [13, 22, 26, 27].

Other marginal efforts have been made to transform shared memory to distributed memory parallelization, yet not directly

from serial to distributed memory parallelization. OMP2MPI [28] is a source-to-source compiler based on the BSCs Mercurium [5] framework that automatically generates MPI source code from shared-memory parallel OpenMP code. OMP2MPI gets parallel OpenMP code, and its AST as input, detects and transforms `pragma omp parallel for` blocks, and then divides the task into MPI manager and worker processes. CATO [30] is another static compiler that uses LLVM and Clang to transform OpenMP code to MPI. Its main component is an LLVM transform pass, which transforms the original OpenMP kernel using an intermediate representation (IR), an assembly-like representation of code.

These tools have many limitations. First, given the final purpose is to automatically convert serial code to a distributed one, converting serial code to a shared memory through the process leads to performance degradation due to imperfections in these compilers, including ones that originally related to the serial to shared memory automatic parallelization. Imperfections like increasing run time, using two-sided communication only, and even the necessity of external information [12] damage the utility of this as a practical tool. Second, most of the static tools are source-to-source compilers, which obligate their use on compiled codes only.

2 MPI PARALLELIZATION WITH LLMS

Recently, there has been a shift towards data-driven methods, especially large language models (LLMs). The basic approach to use LLMs for parallelization tasks is by utilizing those as-is in a zeroshot performance fashion [6, 11, 21, 24, 25, 31], while the more advanced approach is built upon fine-tuning said models for the parallelization specific task [8, 9, 9, 10, 14, 17, 19, 20, 25]. In both of those cases, there has been success in utilizing transformers to *classify* the need for OpenMP pragmas and MPI functions [29] and even to *generate* single-lined OpenMP pragmas [18], introducing a novel approach for automatic parallelization. However, the nuanced task of generating intricate, *multi*-functional MPI codes across diverse *locations* in a dedicated fine-tuned model has remained unexplored [7]. Nevertheless, recently, MONOCODER [18] has been introduced as a novel approach aimed at enhancing LMs tailored for HPC tasks. The hypothesis posits that HPC-specific LMs, such as those designed and trained specifically on HPC datasets, would outperform existing LMs in HPC contexts. Two experiments are conducted to validate this hypothesis. Initially, MONOCODER is constructed by reducing the number of layers of PolyCoder code LM by a factor of 4 and pretraining it solely on C and C++ codes. Despite its smaller size, MONOCODER achieves comparable perplexity scores to PolyCoder on the HPCORPUS dataset, indicating its efficacy in understanding HPC-specific code structures. Subsequently, MONOCODER's performance on HPC tasks is evaluated, focusing on CodeBLEU competence for high-performance and parallel code generations, particularly in OpenMP parallelization tasks. Existing LMs struggle with capturing local semantics, affecting their performance in predicting OpenMP clauses accurately. To address this limitation, TOKOMPILER, a novel code pre-processing scheme that eliminates local semantics, is introduced. MONOCODER consistently outperforms existing LMs across various tests and tasks, demonstrating its robustness and adaptability in HPC contexts, even under semantic-less settings. Yet, no benchmark or adaptation of MONOCODER was performed for the MPI code generation problem.

3 RESEARCH OBJECTIVES AND CONTRIBUTIONS

The gap in the existing methods for MPI source-to-source code parallelization – either by compilers or LMs – has prompted our proposal for a data-driven generative language model using a sequence-to-sequence transformer-based approach. Such model's aim is to automatically suggest MPI functions for MPI codes with MPI functions excluded (semi-serial codes). This is by using a designated database of semi-serial codes and fully MPI parallelized code pairs, providing a new perspective on addressing the complexities of distributed-memory parallelization. Thus, this study presents MPIRIGEN, a tool developed to assist MPI programmers in automatically generating correct MPI functions in an MPI-based domain decomposition parallel code. MONOCODER is the base model, hence, to check its suitability and MPIRIGEN performance we would like to answer the following research questions:

- **RQ1:** Is MONOCODER capable of generating proper MPI code without fine-tuning, and in the absence of local semantics?
- **RQ2:** Is MPIRIGEN capable of inserting the calls to MPI functions in the right locations?
- **RQ3:** Is MPIRIGEN capable of generating calls to correct MPI functions?
- **RQ4:** Is MPIRIGEN capable of generating correct arguments to the MPI functions?

  Contributions. The main contributions of this paper are:

- We create the first MPI codes only dataset — HPCORPUSMPI which is based on MPICODECORPUS and HPCORPUS.
- We have demonstrated MONOCODER understands MPI codes better than PolyCoder and GPT3.5 by using a code completion task on HPCORPUSMPI with TOKOMPILER source code version, a code with AST information embedded while semantic information is neglected.
- We propose an innovative pre-process for completion tasks merely after observing the whole code, thus enabling better completion with a wider context.
- We train and evaluate our approach, named MPIRIGEN by fine-tuning a domain-specific model — MONOCODER, and find that our model performs well in suggesting MPI functions for domain decomposition into an MPI-based parallel code and is better than GPT3.5 zero-shot.

4 HPCORPUSMPI

While MPI programs have been already scraped from GitHub and gathered in corpora such as MPICODECORPUS [29] or even HPCORPUS [16], no corpus contains merely MPI domain decomposition codes. HPCORPUSMPI is a corpus consisting of

Figure 1 code blocks:

```
1  int main(argc,argv)
2  {
3      int done = 0, n = 10000, rank,
           numprocs, i;
4      double pi_total, pi, h, sum, x;
5
6      MPI_Init(&argc,&argv);
7      MPI_Comm_size(MPI_COMM_WORLD,
           &numprocs);
8      MPI_Comm_rank(MPI_COMM_WORLD,
           &rank);
9
10     while (!done)
11     {
12         MPI_Bcast(&n, 1, MPI_INT, 0,
               MPI_COMM_WORLD);
13         if (n == 0)
14             break;
15         h = 1.0 / (double) n;...
```

(a) MPI Source code

```
1  int func_270(type_255,type_762)
2  {
3      int var_289 = num_463, var_649 =
           num_199, var_635, var_257, var_165;
4      double var_663, var_792, var_305,
           var_610, var_55;
5
6      MPI_Init(&type_255,&type_762);
7      MPI_Comm_size(var_674,&var_257);
8      MPI_Comm_rank(var_674,&var_635);
9
10     while (!var_289)
11     {
12         MPI_Bcast(&var_649, num_700,
               var_167, num_463, var_674);
13         if (var_649 == num_463)
14             break;
15         var_305 = num_302 / (double)
               var_649;...
```

(b) TOKOMPILER MPI source code

```
1  int main(argc,argv)
2  {
3      int done = 0, n = 10000, rank,
           numprocs, i;
4      double pi_total, pi, h, sum, x;
5
6      while (!done)
7      {
8          if (n == 0)
9              break;
10         h = 1.0 / (double) n;...
11
12  (6, MPI_Init(&argc,&argv);)
    (7,
        MPI_Comm_size(MPI_COMM_WORLD,&numprocs);)
    (8,
        MPI_Comm_rank(MPI_COMM_WORLD,&rank);)
    (12, MPI_Bcast(&n, 1, MPI_INT, 0,
        MPI_COMM_WORLD);)...
```

(c) MPI Pre-processed source code

Figure 1: The MPI functions in the source code (a) are removed and concatenated with their corresponding line number to the last line (3). This way, MPIIrigen learns in a left-to-right fashion the relation between code and its appropriate MPI functions. The TokomPiler version in (2), embeds AST information while neglecting any semantic information. TokomPiler version is used to demonstrate MonoCoder's semantic information independence during generation compared to other models.
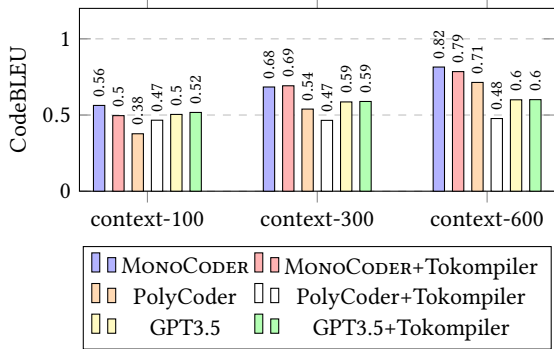

Figure 2: Performance of various models on Code Completion task over the HPCorpusMPI. Models predict code continuation starting from token 100, 300, and 600.

both of the corpora above but under the filter of MPI domain decomposition codes only and with duplication removal resulting in a total of 16,384 programs.

Function	Amount	Function	Amount
MPI_Finalize	19,183	MPI_Comm_size	14,387
MPI_Init	16,135	MPI_Recv	13,783
MPI_Comm_rank	16,096	MPI_Bcast	6,995
MPI_Send	14,534	MPI_Reduce	3,600

Table 1: MPI Common Core functions distribution for HPCorpusMPI dataset.

To create the dataset for training MPIIrigen out of the corpus, several pre-process stages have been done (Figure 1):

(1) Codes with both `MPI_Init` and `MPI_Finalize` and under the limit of 2048 tokens (with BPE tokenizer) have been added.
(2) Every line of each code snippet has been numbered.
(3) MPI functions through the code have been removed.
(4) Locations of the removed MPI functions and the functions themselves have been written to the last line of the codes.

For convenience, the final dataset (with 13,322 programs) contains three fields – "Program", "Code", "MPI label" – and corresponds to the program's GitHub username, the code with removed MPI functions, and the label of the locations and their MPI functions.

"MPI Common Core" functions, as defined in [29], are the most prevalent MPI functions in domain decomposition. Therefore, it is essential to validate that the distribution in the dataset is reasonable both for model training and evaluation purposes. Towards that end, we analyzed HPCorpusMPI for MPI Common Core functions. The distribution of these functions is presented in Table 1.

5 MPIIRIGEN – EXPERIMENTAL RESULTS

This study presents MPIIrigen, a tool developed to assist MPI programmers in automatically generating correct MPI functions in an MPI-based domain decomposition parallel code. We introduce a dedicated downstream task: fine-tuning MonoCoder, which is a PolyCoder model pre-trained for the C and C++ languages associated with MPI, on HPCorpusMPI, resulting in the creation of MPIIrigen.

Furthermore, we propose an innovative pre-process for the given completion task and completion tasks in general. A number specifying its line number is added for each line in the code.

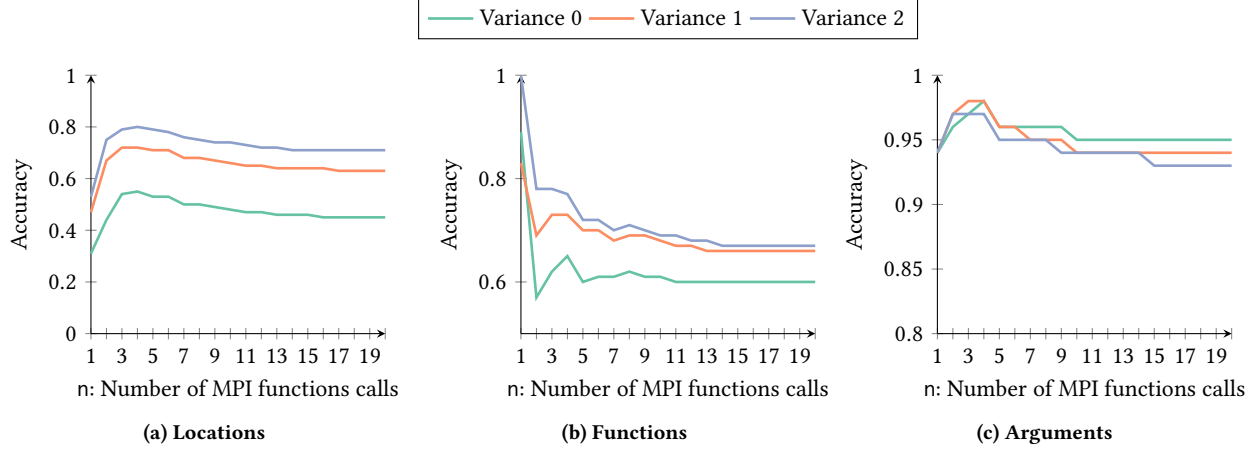**(a) Locations**   **(b) Functions**   **(c) Arguments**

Figure 3: Performance breakdown of MPIrigen (fine-tuned MonoCoder-0.7B over 16K MPI codes from HPCorpusMPI) over programs containing (n) or less of MPI function calls (X axis). Y axis is accuracy obtained for such programs. Note shifted scales in sub-figure b and c.
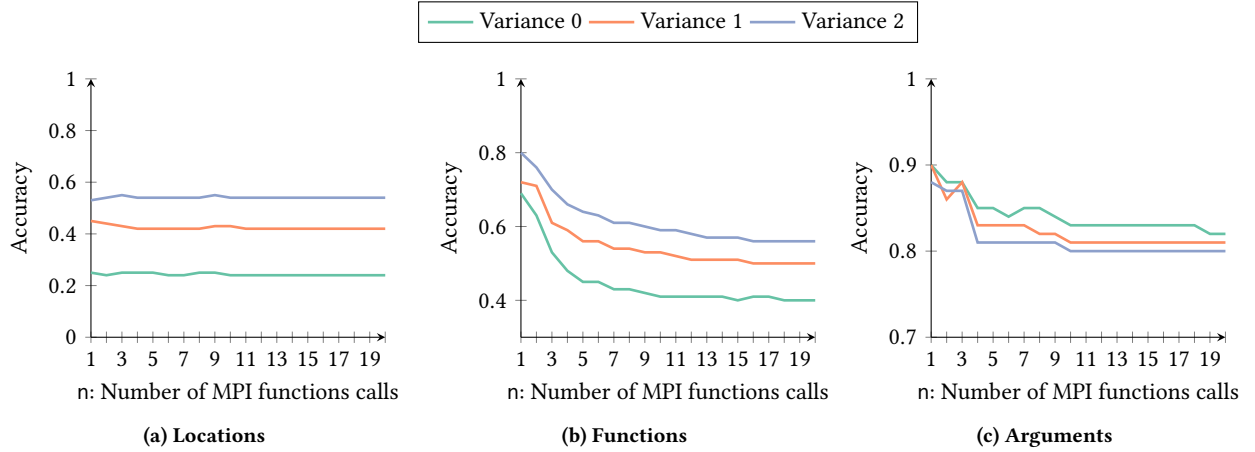

(a) Locations   **(b) Functions**   **(c) Arguments**

Figure 4: Performance breakdown of GPT3.5 using prompt "Generate the optimal MPI functions for the provided code, and supply in the response the entire complete code with those MPI functions: [CODE]". X axis represents programs containing n or less of MPI function calls. Y axis is accuracy obtained for such programs. Note shifted scales in sub-figure b and c.

Then, all the MPI functions with their locations are removed and concatenated to the last line (Figure 1). Training input will be the resulting code and the test will be to predict the last line. This enables training a language model in a left-to-right fashion, outputting the MPI functions merely after observing the whole code, thus enabling better completion with a wider context. MonoCoder is fine-tuned this way resulting in MPIrigen.

we ran two experiments: the first experiments evaluates ability of base models to generate MPI functions and answers **RQ1**, while the second experiment is designed to answer **RQ2**, **RQ3**, **RQ4** and it is the novel approach we are introducing.

RQ1: MPI code generation. To develop MPIrigen and answer the first RQ, we first investigated the performance of state-of-the-art pre-trained code language models (PolyCoder

and GPT3.5) in generating MPI codes using varied context sizes for next-token predictions compared to MonoCoder. This code completion task has been done over a thousand examples with up to 2048 tokens in HPCorpusMPI and under an initial context of 100, 300, and 600 tokens (Figure 2). In addition, to check the resilience of these pre-trained models to semantic information, the same test has been conducted through codes' TokomPiler [18] version. Any semantic information has been replaced using TokomPiler (Figure 1b) and thus, a measure of reliance of these pre-trained models on semantic information has been tested. Results show that as context grows, MonoCoder is significantly outperforming PolyCoder and GPT-3.5 results, while also proving to not fix on local semantics for those results, suggesting high generality capabilities.
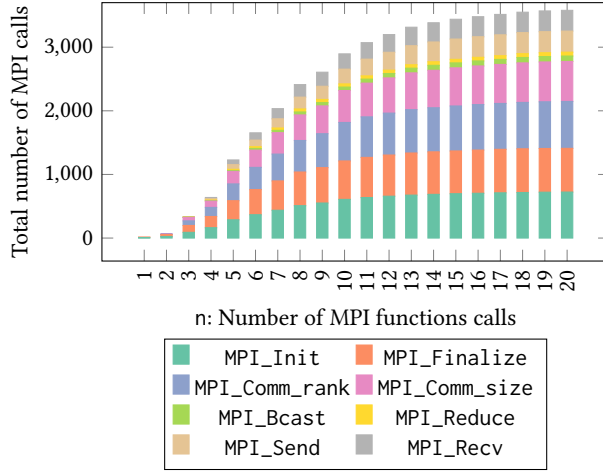
(a) Ground truth
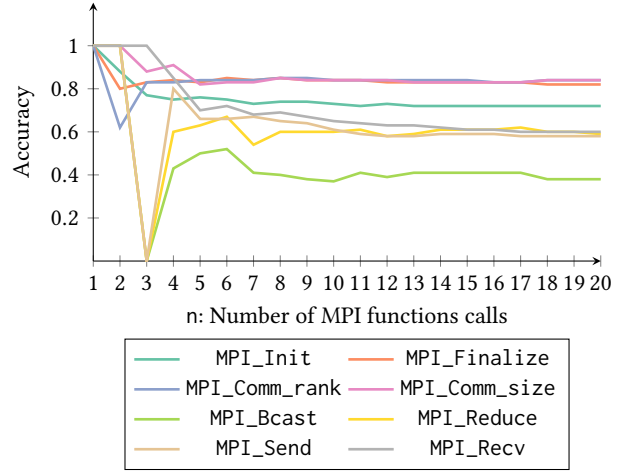
(b) Prediction

Figure 5: Stacked bar chart of the ground truth and MPIrigen prediction distribution of selected MPI functions under variance 2 (correct location and function predictions are presented). X axis represents programs containing n or less of MPI function calls.

Evaluation metric for RQ2, RQ3, RQ4. Since we evaluate MPI function generation, prevalent generation metrics such as BLUE, Meteor, and Rouge-l are not relevant. Thus, we propose an HPC-oriented evaluation method. First, functions matching locations are to be found. Matching toleration is determined by a variance. The variance is a flexibility measure for accuracy, as many times, the right functions appear in one or two lines from the original locations, usually without interfering with the original code structure (variance zero refers to exact location predictions). MPI functions with correct locations will be forwarded to calling evaluation, checking whether the right MPI function has been called. Afterward, arguments will be checked out of the correct MPI functions. Scoring is made by calculating the correct to total arguments ratio. This evaluation was conducted with a variance of 0-2 and as a number of MPI functions which reveals the accuracy distribution through simple MPI codes to more complicated ones. Functions accuracy is measured for matched locations only and arguments accuracy for matched functions only. The results are given as a function of the number of original MPI calls (1-20), showcasing that accuracy drops as complexity grows (besides the ultra simplicity of one or two MPI function calls).

Results for RQ2, RQ3, RQ4. We developed the model using the Huggingface framework and trained it on two NVidia Tesla-V100 GPUs, each having 32GB memory. The training was carried out with a batch size of 2, 2048 tokens and 3 epochs. As shown in Figure 3, for MPIrigen, the location as well as functions accuracy under variance of 2 reach 80% and converge to 70%, while for arguments it reaches 95% and converge to 94%. For GPT3.5 on the contrary, the accuracy converged to 50% for locations and approximately 55% and 80% for functions and arguments resp (Figure 4).

Result analysis and future work. We analyzed the performance of MPIrigen in details by breaking down the location and function accuracy over commonly-used MPI functions. This analysis is presented in Figure 5, with Figure 5a showing the number of calls of different MPI functions in the ground truth and Figure 5b showing the accuracy of predicting those calls with a variance of two. As the figure shows the accuracy of predicting the correct location varies for different MPI functions, and we believe it correlates with the proportion of different MPI calls in the training dataset and possibly in open-source repositories on GitHub. It would be thus an interesting study to develop a balanced MPI dataset and train MPIrigen on it. Currently, we do not evaluate the correctness of generated MPI codes, nonetheless we are envisioning an approach based on compilation and output verification as a part of immediate future work.

6 CONCLUSION

Our findings reveal that widely used models like GPT-3.5 and specialized multi-lingual code models like PolyCoder exhibit notable performance degradation when generating MPI codes compared to their outcomes for general-purpose codes, as shown in [18]. In contrast, domain-specific models like MonoCoder, pre-trained for the C and C++ languages associated with MPI, outperform larger models, showcasing high generality capabilities, especially when local misleading semantics are mitigated. Comparative analysis of MPIrigen against GPT zero-shot performance, using the above evaluation method for MPI functions generation, demonstrates that MPIrigen excels in generating accurate MPI codes. The success of this tailored solution underscores the importance of domain-specific fine-tuning in optimizing language models for parallel computing code generation.

REFERENCES

[1] [n. d.]. NegevHPC Project. https://www.negevhpc.com. [Online].

[2] [n. d.]. Par4All homepage. http://par4all.github.io/. [Online].

[3] Mehdi Amini, Béatrice Creusillet, Stéphanie Even, Ronan Keryell, Onig Goubier, Serge Guelton, Janice Onanian McMahon, François-Xavier Pasquier, Grégoire Péan, and Pierre Villalon. 2012. Par4all: From convex array regions to heterogeneous computing. In *IMPACT 2012: Second International Workshop on Polyhedral Compilation Techniques HiPEAC 2012*.

[4] Hansang Bae, Dheya Mustafa, Jae-Woo Lee, Hao Lin, Chirag Dave, Rudolf Eigenmann, and Samuel P Midkiff. 2013. The cetus source-to-source compiler infrastructure: overview and evaluation. *International Journal of Parallel Programming* 41 (2013), 753–767.

[5] Jairo Balart, Alejandro Duran, Marc Gonzàlez, Xavier Martorell, Eduard Ayguadé, and Jesús Labarta. 2004. Nanos mercurium: a research compiler for openmp. In *Proceedings of the European Workshop on OpenMP*, Vol. 8. 2004.

[6] Brian Chen, Nafis Mustakin, Alvin Hoang, Sakib Fuad, and Daniel Wong. 2023. VSCuda: LLM Based CUDA Extension for Visual Studio Code. In *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis* (<conf-loc>, <city>Denver</city>, <state>CO</state>, <country>USA</country>, </conf-loc>) *(SC-W '23)*. Association for Computing Machinery, New York, NY, USA, 11–17. https://doi.org/10.1145/3624062.3624064

[7] Le Chen, Nesreen K Ahmed, Akash Dutta, Arijit Bhattacharjee, Sixing Yu, Quazi Ishtiaque Mahmud, Waqwoya Abebe, Hung Phan, Aishwarya Sarkar, Branden Butler, et al. 2024. Position Paper: The Landscape and Challenges of HPC Research and LLMs. *arXiv preprint arXiv:2402.02018* (2024).

[8] Le Chen, Arijit Bhattacharjee, Nesreen Ahmed, Niranjan Hasabnis, Gal Oren, Vy Vo, and Ali Jannesari. 2024. OMPGPT: A Generative Pre-trained Transformer Model for OpenMP. arXiv:2401.16445 [cs.SE]

[9] Le Chen, Pei-Hung Lin, Tristan Vanderbruggen, Chunhua Liao, Murali Emani, and Bronis de Supinski. 2023. LM4HPC: Towards Effective Language Model Application in High-Performance Computing. *arXiv preprint arXiv:2306.14979* (2023).

[10] Xianzhong Ding, Le Chen, Murali Emani, Chunhua Liao, Pei-Hung Lin, Tristan Vanderbruggen, Zhen Xie, Alberto E Cerpa, and Wan Du. 2023. HPC-GPT: Integrating Large Language Model for High-Performance Computing. *arXiv preprint arXiv:2311.12833* (2023).

[11] William Godoy, Pedro Valero-Lara, Keita Teranishi, Prasanna Balaprakash, and Jeffrey Vetter. 2023. Evaluation of OpenAI Codex for HPC Parallel Programming Models Kernel Generation. In *Proceedings of the 52nd International Conference on Parallel Processing Workshops*. 136–144.

[12] Khaled Hamidouche, Joel Falcou, and Daniel Etiemble. 2011. A framework for an automatic hybrid MPI+ OpenMP code generation.. In *SpringSim (hpc)*. Citeseer, 48–55.

[13] Re'em Harel, Idan Mosseri, Harel Levin, Lee-or Alon, Matan Rusanovsky, and Gal Oren. 2020. Source-to-source parallelization compilers for scientific shared-memory multi-core and accelerated multiprocessing: analysis, pitfalls, enhancement and potential. *International Journal of Parallel Programming* 48, 1 (2020), 1–31.

[14] Re'em Harel, Yuval Pinter, and Gal Oren. 2023. 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*. 450–452.

[15] Intel. 2023. Intel Developer Cloud. https://www.intel.com/content/www/us/en/developer/tools/devcloud/overview.html. [Online].

[16] Tal Kadosh, Niranjan Hasabnis, Timothy Mattson, Yuval Pinter, and Gal Oren. 2023. Quantifying openmp: Statistical insights into usage and adoption. In *2023 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[17] Tal Kadosh, Niranjan Hasabnis, Timothy Mattson, Yuval Pinter, Gal Oren, et al. 2023. PragFormer: Data-driven Parallel Source Code Classification with Transformers. (2023).

[18] Tal Kadosh, Niranjan Hasabnis, Vy A Vo, Nadav Schneider, Neva Krien, Mihai Capota, Abdul Wasay, Nesreen Ahmed, Ted Willke, Guy Tamir, et al. 2023. Domain-Specific Code Language Models: Unraveling the Potential for HPC Codes and Tasks. *arXiv preprint arXiv:2312.13322* (2023).

[19] Tal Kadosh, Niranjan Hasabnis, Vy A Vo, Nadav Schneider, Neva Krien, Abdul Wasay, Nesreen Ahmed, Ted Willke, Guy Tamir, Yuval Pinter, et al. 2023. Scope is all you need: Transforming LLMs for HPC Code. *arXiv preprint arXiv:2308.09440* (2023).

[20] Tal Kadosh, Nadav Schneider, Niranjan Hasabnis, Timothy Mattson, Yuval Pinter, and Gal Oren. 2023. Advising OpenMP Parallelization via a Graph-Based Approach with Transformers. *arXiv preprint arXiv:2305.11999* (2023).

[21] Quazi Ishtiaque Mahmud, Ali TehraniJamsaz, Hung D Phan, Nesreen K Ahmed, and Ali Jannesari. 2023. AUTOPARLLM: GNN-Guided Automatic Code Parallelization using Large Language Models. *arXiv preprint arXiv:2310.04047* (2023).

[22] Reed Milewicz, Peter Pirkelbauer, Prema Soundararajan, Hadia Ahmed, and Tony Skjellum. 2021. Negative Perceptions About the Applicability of Source-to-Source Compilers in HPC: A Literature Review. In *International Conference on High Performance Computing*. Springer, 233–246.

[23] Idan Mosseri, Lee-or Alon, Re'Em Harel, and Gal Oren. 2020. 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, 247–262.

[24] Daniel Nichols, Joshua H Davis, Zhaojun Xie, Arjun Rajaram, and Abhinav Bhatele. 2024. Can Large Language Models Write Parallel Code? *arXiv preprint arXiv:2401.12554* (2024).

[25] Daniel Nichols, Aniruddha Marathe, Harshitha Menon, Todd Gamblin, and Abhinav Bhatele. 2023. Modeling Parallel Programs using Large Language Models. *arXiv preprint arXiv:2306.17281* (2023).

[26] S Prema, R Jehadeesan, and BK Panigrahi. 2017. Identifying pitfalls in automatic parallelization of NAS parallel benchmarks. In *Parallel Computing Technologies (PARCOMPTECH), 2017 National Conference on*. IEEE, 1–6.

[27] S Prema, Rupesh Nasre, R Jehadeesan, and BK Panigrahi. 2019. A study on popular auto-parallelization frameworks. *Concurrency and Computation: Practice and Experience* 31, 17 (2019), e5168.

[28] Albert Saa-Garriga, David Castells-Rufas, and Jordi Carrabina. 2015. OMP2MPI: Automatic MPI code generation from OpenMP programs. https://doi.org/10.48550/ARXIV.1502.02921

[29] Nadav Schneider, Tal Kadosh, Niranjan Hasabnis, Timothy Mattson, Yuval Pinter, and Gal Oren. 2023. MPI-RICAL: Data-Driven MPI Distributed Parallelism Assistance with Transformers. In *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis* (Denver, CO, USA) *(SC-W '23)*. Association for Computing Machinery, New York, NY, USA, 2–10. https://doi.org/10.1145/3624062.3624063

[30] Jannek Squar, Tim Jammer, Michael Blesel, Michael Kuhn, and Thomas Ludwig. 2020. Compiler Assisted Source Transformation of OpenMP Kernels. In *2020 19th International Symposium on Parallel and Distributed Computing (ISPDC)*. 44–51. https://doi.org/10.1109/ISPDC51135.2020.00016

[31] Pedro Valero-Lara, Alexis Huante, Mustafa Al Lail, William F Godoy, Keita Teranishi, Prasanna Balaprakash, and Jeffrey S Vetter. 2023. Comparing Llama-2 and GPT-3 LLMs for HPC kernels generation. *arXiv preprint arXiv:2309.07103* (2023).