

ChatMPI: LLM-Driven MPI Code Generation for HPC Workloads

Pedro Valero-Lara

Oak Ridge National Laboratory
Oak Ridge, USA
valerolarap@ornl.gov

Aaron Young

Oak Ridge National Laboratory
Oak Ridge, USA
youngar@ornl.gov

Thomas Naughton III

Oak Ridge National Laboratory
Oak Ridge, USA
naughtont@ornl.gov

Christian Engelmann

Oak Ridge National Laboratory
Oak Ridge, USA
engelmannnc@ornl.gov

Al Geist

Oak Ridge National Laboratory
Oak Ridge, USA
geistal@ornl.gov

Jeffrey S. Vetter

Oak Ridge National Laboratory
Oak Ridge, USA
vetter@ornl.gov

Keita Teranishi

Oak Ridge National Laboratory
Oak Ridge, USA
teranishik@ornl.gov

William F. Godoy

Oak Ridge National Laboratory
Oak Ridge, USA
godoywf@ornl.gov

Abstract

The Message Passing Interface (MPI) standard plays a crucial role in enabling scientific applications for parallel computing and is an essential component in high-performance computing (HPC). However, implementing MPI code manually—especially applying a proper domain decomposition and communication pattern—is a challenging and error-prone task. We present ChatMPI, an AI assistant for MPI parallelization of sequential C codes. In our analysis, we focus on testing six essential HPC workloads, which are based on Basic Linear Algebra Subprograms levels 1, 2, and 3 as well as sparse, stencil, and iterative operations. We analyze the process of creating ChatMPI by using the ChatHPC library. This lightweight large language model (LLM)-based infrastructure enables HPC experts to efficiently create and supervise trustworthy AI capabilities for critical HPC software tasks. We study the data required for training (fine-tuning) ChatMPI to generate parallel codes that not only use MPI syntax correctly but also apply HPC techniques to reduce memory communication and maximize performance by using proper work decomposition. With a relatively small training dataset composed of a few dozen prompts and fewer than 15 minutes of fine-tuning on one node equipped with two NVIDIA H100 GPUs, ChatMPI elevates trustworthiness for MPI code generation of current LLMs (e.g., Code Llama, ChatGPT-4o and ChatGPT 5). Additionally, we evaluate the performance of the MPI codes generated by ChatMPI in comparison with the ones generated by ChatGPT-4o and ChatGPT-5. The codes generated

by ChatMPI provide up to a 4× boost in performance by using better problem decomposition, communication patterns, and HPC techniques (e.g., communication avoiding).

Keywords

ChatHPC, AI, LLM, MPI, HPC

ACM Reference Format:

Pedro Valero-Lara, Aaron Young, Thomas Naughton III, Christian Engelmann, Al Geist, Jeffrey S. Vetter, Keita Teranishi, and William F. Godoy. 2026. ChatMPI: LLM-Driven MPI Code Generation for HPC Workloads. In *SCA/HPCAsia 2026: Supercomputing Asia and International Conference on High Performance Computing in Asia Pacific Region (SCA/HPCAsia 2026)*, January 26–29, 2026, Osaka, Japan. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3773656.3773659>

1 Motivation, Hypothesis, Objectives, and Contributions

Since the release of ChatGPT in 2022, large language models (LLMs) have dominated the landscape of computing and influenced every domain of human-computer interaction. Most research revolves around the impact of AI on science, and high-performance computing (HPC) is no exception as the community identifies high-return research directions [32]. Moreover, HPC software development is itself a highly iterative and scientific process, in which the goal is scientific discovery attainable only at large computing scales.

Introduced in the mid-1990s to address the complexity of programming HPC systems, the Message Passing Interface (MPI) [40] has become the de facto portable standard for parallel network communication programming. All major HPC vendors support the set of standard functions and subroutines provided by MPI, and many implementations are available for C and Fortran. Thus, MPI’s impact can be associated with HPC for the past three decades, and the standard continues to evolve. The latest MPI version (5.0), which was released in June 2025, added application binary interface standardization for implementation interoperability. Although MPI is widely used, its complexity and programming model require a shift from traditional serial computing structures (e.g., for loops, if

Notice: This manuscript has been authored by UT-Battelle LLC under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The US government retains and the publisher, by accepting the article for publication, acknowledges that the US government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for US government purposes. DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (<https://www.energy.gov/doe-public-access-plan>).

This paper is authored by an employee(s) of the United States Government and is in the public domain. Non-exclusive copying or redistribution is allowed, provided that the article citation is given and the authors and agency are clearly identified as its source. Request permissions from owner/author(s).

SCA/HPCAsia 2026, Osaka, Japan

2026. ACM ISBN 979-8-4007-2067-3/26/01

<https://doi.org/10.1145/3773656.3773659>

branches, memory management). Consequently, MPI parallelization strategies still present a steep learning curve within a parallel computing system.

MPI applications demand effective parallelization, optimization, and performance through the proper application of HPC fundamentals—parallel processing techniques that can significantly speed up computation and communication on HPC systems. However, these techniques require significant expertise, which can pose challenges for developers. Recent advances in LLMs offer a promising solution by automating and enhancing complex programming tasks, thereby accelerating the development process. The **motivation** of this work is to evaluate and enhance the capabilities of current LLMs for generating MPI codes. The LLM-generated MPI codes must provide good performance through the effective use of HPC fundamentals (e.g., problem decomposition, communication patterns).

The starting point for our investigation and experimentation is verifying the **hypothesis**: “LLMs can parallelize sequential programs into fully functional and well-performing MPI codes.”

Then, the **objective** of this work is twofold: (1) training LLMs to generate functional MPI codes (i.e., codes that implement MPI syntax correctly and can be compiled and run) from a collection of sequential codes that implement essential HPC workloads and (2) enabling LLMs to implement well-performing MPI codes that leverage HPC fundamentals by making informed decisions regarding work decomposition to minimize communication and maximize performance and scalability where possible.

Finally, the **contributions** of this work are as follows:

- The implementation of ChatMPI, an AI assistant designed to support HPC developers in the MPI parallelization of sequential C codes. ChatMPI was created with the ChatHPC library, a lightweight LLM-based infrastructure that enables HPC experts to efficiently create and supervise trustworthy AI capabilities for critical HPC software tasks.
- A demonstration and an evaluation of ChatMPI on essential HPC workloads, including all three levels of the Basic Linear Algebra Subprograms (BLAS) library as well as sparse, stencil, and iterative operations. We study the requirements for training and testing data and fine-tuning. We also show the correctness levels and performance achieved by the MPI code generated by ChatMPI in comparison with three other reference models: Code Llama, ChatGPT-4o and ChatGPT-5.

2 ChatHPC

The ChatHPC ecosystem comprises the ChatHPC library and a collection of fine-tuned AI assistants created by the ChatHPC library for the specific tasks that constitute the essential parts of the HPC software stack. As Fig. 1 illustrates, creating a new AI assistant in ChatHPC involves a three-step iterative process: (1) **fine-tuning**, (2) **testing**, and (3) **refinement**.

The (1) **fine-tuning** process creates a parameter-efficient adapter for each AI assistant by using the (A) *training data* and a JavaScript Object Notation (JSON) file with a set of prompt-context-answer tuples and data about the capabilities to be created (Fig. 2). In our case, this data usually occupies only a few kilobytes of memory. The (B) *adapters* are small and trainable modules added to the

base LLM to enable lightweight and efficient fine-tuning for specific tasks, thereby preserving the original model knowledge while adapting to new scenarios. Each adapter (AI assistant) requires about 100 MB of memory. The (I) base model—Code Llama with 7 billion parameters (13 GB) in our case—does not need to be replicated. This scalable approach allows the HPC community to simultaneously work on multiple AI assistants and requires modest computational resources. Code Llama is trained for multiple programming languages (Python, C++, Java, PHP, TypeScript, C#, and Bash) to cover a wide range of applications. Code Llama achieves performance comparable to that of other open models on several code benchmarks with support for large input contexts. Its ability to handle long sequences of code relies on measuring perplexity [27] over 4 million tokens from the code dataset and key retrieval [21]. Code Llama uses execution feedback to select data to train the model—a self-instruction dataset that resulted in about 14,000 question-tests-solution triplets, 62,000 interview-style programming questions, and about 52,000 questions. Code Llama is based on the Llama 2 models, which are trained on 2 trillion tokens of text, including 80 billion tokens of code. Code Llama is the result of tuning Llama 2 models on 500 billion extra tokens that consist mostly of code (85%). (2) **Testing** focuses on checking whether the model is built correctly according to its design specifications and ensures that the model meets the intended user needs and performs well. This occurs after fine-tuning when a functional AI assistant is available for testing using (C) *testing data* that differs from that used during fine-tuning (training data). The testing data may contain prompts about capabilities not related to the capabilities created. (3) **Refinements** are made after testing when (D) *learning gaps* (i.e., prompts in the testing data that did not receive good responses) are identified. This expert-guided refinement consists of complementing the training data with additional information called (E) *refinement data*, which covers the deficiencies found during validation for a new fine-tuning process. This iterative process allows AI assistants to dynamically learn and adapt to complex tasks as more learning gaps are identified. Although a user can interact with each AI assistant (adapters) individually, these adapters can also be merged with the base model’s weights, if necessary, to create a single LLM model file—(II) the ChatHPC model in Fig. 1.

The ChatHPC library relies on well-known, robust, efficient, and widely used tools and languages, including the aforementioned open-source Meta Code Llama LLM [29], the low-rank adaptation [18] tool used for fine-tuning pretrained LLMs, and the open-source PyTorch [26] Python framework used for building machine learning models. We also implemented a Python command line interface (CLI) library in ChatHPC for ease of use (Fig. 3). By integrating these widely used tools and methods, we aim to make the ChatHPC library interoperable and compatible with other important efforts within the AI community (e.g., Ollama [2], Hugging Face [1]) while ensuring usability.

3 ChatMPI

ChatMPI is an AI assistant created with the ChatHPC library to generate parallel MPI code from a given sequential code. We target the generation of MPI codes that use the MPI syntax correctly and demonstrate good HPC capabilities by using proper workload

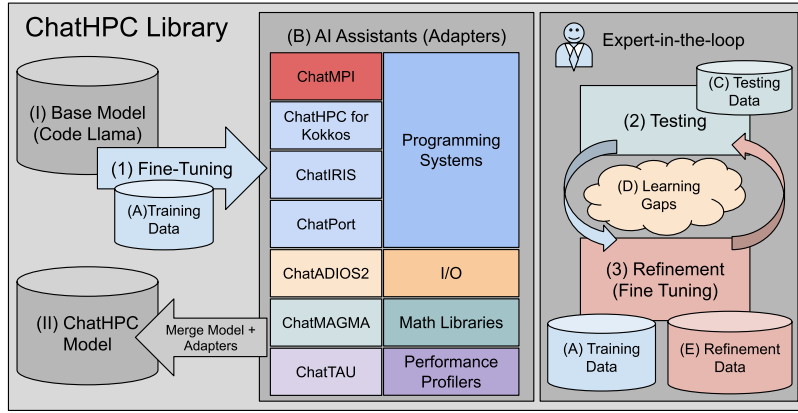


Figure 1: The ChatHPC ecosystem.

```
[
{"question": "Can you parallelize this sequential code to MPI?",
  "context": "Sequential to MPI translation",
  "answer": "
int n = 5000;
float sum = 0.0;
float *x = malloc(n*sizeof(float));
float *y = malloc(n*sizeof(float));
for (i = 0, i < n, i++) {
  sum += x[i] * y[i];
}
free(x);
free(y);"
}
{"question": "Can you parallelize this sequential code to MPI?",
  "context": "Sequential to MPI translation",
  "answer": "
int size, rank;
MPI_Init(NULL, NULL);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
int n = ceil(5000/size);
float lsum = 0.0;
float gsum = 0.0;
float *x = malloc(n*sizeof(float));
float *y = malloc(n*sizeof(float));
for (i = 0, i < n, i++) {
  local_sum += x[i] * y[i];
}
MPI_Reduce(&lsum, &gsum, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);
free(x);
free(y);
MPI_Finalize();"
}
]
```

```
[
{"question": "Can you parallelize this sequential code to MPI?",
  "context": "Sequential to MPI translation",
  "answer": "
int n = 500;
float *u = malloc(n*sizeof(float));
float *u_new = malloc(n*sizeof(float));
for (int i = 1; i < n-1; i++) {
  for (int j = 1; j < n-1; j++) {
    u_new[i*n+j] = 0.25*(u[(i-1)*n+j] + ...
  }
}
free(u);
free(u_new);"
}
{"question": "Can you parallelize this sequential code to MPI?",
  "context": "Sequential to MPI translation",
  "answer": "
int size, rank;
MPI_Status status;
MPI_Init(NULL, NULL);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
int n = 500;
int local_n = ceil(500/size) + 2;
float *u = malloc(n*local_n*sizeof(float));
float *u_new = malloc(n*local_n*sizeof(float));
if (rank > 0) {
  MPI_Send(&u[1], 1, MPI_FLOAT, rank-1, 0, MPI_COMM_WORLD);
  MPI_Recv(&u[0], 1, MPI_FLOAT, rank-1, 0, MPI_COMM_WORLD, &status);
}
if (rank < size-1) {
  MPI_Recv(&u[n-1], 1, MPI_FLOAT, rank+1, 0, MPI_COMM_WORLD, &status);
  MPI_Send(&u[n-2], 1, MPI_FLOAT, rank+1, 0, MPI_COMM_WORLD);
}
for (int i = 1; i < local_n-1; i++) {
  for (int j = 1; j < n-1; j++) {
    u_new[i*n+j] = 0.25*(u[(i-1)*n+j] + ...
  }
}
free(u);
free(u_new);
MPI_Finalize();"
}
]
```

Figure 2: Examples of JSON items used to train (fine-tune) ChatMPI for different HPC domains: (left) BLAS 1 and (right) 5-point 2D stencil.

decomposition that minimizes communication and maximizes scalability. Our target is the MPI parallelization of sequential C codes. Essentially, users provide a sequential code, and ChatMPI provides an equivalent code (in terms of functionality) implemented in MPI. So, in this case, we use only one training context in our fine-tuning process (sequential to MPI translation in Fig. 2) with the data used for both training and testing.

We focus on essential HPC workloads widely used by HPC applications, including BLAS, stencil-based algorithms, computations on sparse matrices, and iterative solvers, among others. Initially, we do not provide a complete list of kernels for the fine-tuning but rather a subset of representative cases, including level 1 BLAS kernels, stencil operations, and sparse matrix–vector multiplication (SpMV) using a compressed sparse row (CSR) format. For the initial training data, we use about 20 different question-context-answer samples in

```

ChatHPC CLI:
$ chathpc train # Fine-tune the assistant.
$ chathpc verify # Verify the assistant on training set.
$ chathpc test # Test the assistant on unseen data.
$ chathpc run # Interactively run the assistant.

Interactively run session:
$ chathpc ()> /context
Context: Introduction to Kokkos
$ chathpc (Introduction to Kokkos)> What is LayoutLeft?
LayoutLeft refers to column-major layout where consecutive entries in the same column of a 2D array are contiguous in memory.

```

Figure 3: Example of the CLI for the ChatHPC Python library.

total. Our approach involves withholding some information during the initial fine-tuning so that we can study the quantity and quality of the data required for a state-of-the-art code LLM to effectively learn MPI parallelization.

As shown in Fig. 4, every chosen category or characteristic test case for this analysis represents different challenges in MPI parallelization. Each category requires a different decomposition or different uses of MPI routines, including collective or point-to-point communications, covering an important part of the MPI specification. In some cases, the same test case (e.g., iterative solvers) requires the use of different MPI communication and workload decompositions in the same code.

Although the methodology for creating AI assistants is the same (Fig. 1), the data required differs depending on the contexts or capabilities to be created. These efforts may require more or less data depending on the challenges of effective MPI parallelization and the required knowledge to generate well-performing MPI parallel codes. Additionally, depending on the similarities between prompts (e.g., input source code) and the potential responses (e.g., output MPI source code), we may need larger or smaller training datasets. Generally, the more recurring and repetitive patterns that exist between prompts and responses, the better the LLM can identify the necessary code transformations. This allows AI assistants to make predictions and understand new situations or capabilities based on the knowledge gained from these repeating patterns and without additional fine-tuning data.

4 Analysis

We divide the evaluation into different parts targeting essential HPC workloads: BLAS levels 1, 2, and 3 and sparse, stencil, and iterative operations. We measure this trustworthiness as the percentage of the prompts in the testing dataset that were answered correctly by the ChatHPC AI assistant and the reference LLMs (i.e., Code Llama base model, ChatGPT-4o and ChatGPT-5 models). Both the sequential input codes and the parallel MPI output codes are written in C. For fine-tuning, we use one computational node equipped with two NVIDIA H100 GPUs. The time required to create the AI assistant presented in this work is always fewer than 15 minutes. The training is conducted with a batch size of 128 and 1,200 tokens. Unlike other AI assistants created by ChatHPC, this one requires a higher number of tokens. We use the same number of tokens for both training and testing (inference). For the fine-tuning, we use 400 iterations and a learning rate of $3e^{-4}$, although most of the

capabilities can be trained with about 200–250 iterations. FP16 precision is used for training and inference. This evaluation reveals the level of correctness and performance for the different capabilities created for ChatMPI.

This evaluation answers the following questions: (1) Is it possible to create correct and performant MPI capabilities by fine-tuning LLMs? (2) How much effort would that require? (3) How well do these capabilities perform when compared with state-of-the-art LLMs? In our analysis, we include the results provided by Meta’s Code Llama base model (7 billion parameters) without fine-tuning and OpenAI’s ChatGPT-4o and ChatGPT-5 models. ChatGPT-4o is a mixture of 8×222 billion parameter LLMs for a total of approximately 1.8 trillion parameters. This puts ChatGPT-4o at about $257\times$ more parameters than our Code Llama base model. ChatGPT-5 introduces architectural changes beyond parameter count towards a more deliberate model with multi-step reasoning. These changes aim to improve accuracy to handle complex tasks in areas of coding and problem-solving.

4.1 BLAS

BLAS [10] is a standard library composed of a collection of routines for performing fundamental linear algebra operations, such as vector and matrix manipulations. BLAS provides a set of essential building blocks used in many numerical computations and scientific applications. BLAS comprises three levels, each specialized in a specific type of operation: vector-vector operations in level 1, matrix-vector operations in level 2, and matrix-matrix operations in level 3. In terms of MPI implementation, every higher level in the BLAS library introduces a higher degree of complexity that requires more advanced techniques to achieve a good scalability.

4.1.1 BLAS 1. Level 1 of BLAS performs scalar, vector, and vector-vector operations. We can categorize two main patterns at this level: (1) vector (embarrassingly) parallelism, in which the same operation or instruction must be applied to each element of 1D array(s), and (2) array reduction, which combines all elements of an array into a single value by using a specified operation.

Training and testing datasets: For the training dataset (fine-tuning), we focus only on two well-known and important BLAS 1 operations that represent very well the two categories or characteristic patterns mentioned before: AXPY, which performs a scalar-vector multiplication and vector addition, and DOT product, which takes two vectors and returns the sum of the products of corresponding elements in two input vectors. Whereas no MPI communication

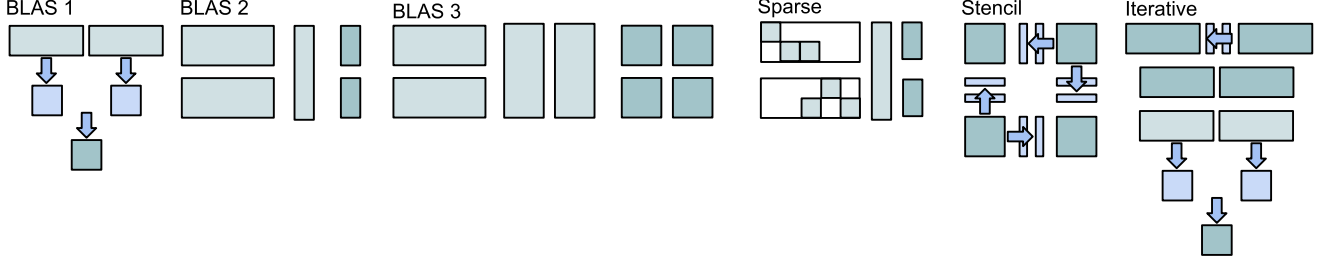


Figure 4: MPI patterns associated with the three BLAS levels and the sparse, stencil, and iterative computations.

is needed for the AXPY operation, DOT requires at least one MPI collective communication to combine partial results from multiple MPI processes into a single result. Note that for the MPI implementation of this operation, we use the MPI_Reduce primitive (see Fig. 2). For the testing dataset, we use a collection of prompts (sequential codes) that implement other BLAS 1 routines not included in the training dataset, such as SCAL or ASUM routines.

Analysis: By using only two examples (AXPY and DOT) for fine-tuning, ChatMPI provides functional and high-quality MPI codes for other BLAS 1 routines. Our base model (Code Llama with no fine-tuning) cannot provide any functional MPI code for the same test cases (prompts). This confirms that Code Llama has not been exposed to MPI syntax or coding. The codes provided by both OpenAI’s models are very similar in this case. Although ChatGPT models provide functional MPI codes for BLAS 1 routines, the codes generated contain unnecessary MPI-collective gather and scatter primitives to distribute and collect the inputs and outputs by using only MPI rank 0. This practice, which is repeated across all the test cases evaluated in this work, adds unnecessary overhead that affects scalability and performance. The impact of this overhead on overall performance can be significant (between 10% and 80%) depending on the nature of the operations to be computed. This may also add extra memory in some cases. Instead, these operations can be performed in parallel by involving all MPI ranks or using specialized parallel I/O libraries such as HDF5 [4] or ADIOS2 [13]. In the codes used for ChatMPI fine-tuning, we initialize and distribute data across all MPI ranks.

4.1.2 BLAS 2 and 3. Levels 2 and 3 [22, 23, 34] of BLAS present more complicated and challenging operations than the ones for BLAS 1, performing matrix-vector (BLAS 2) or matrix-matrix (BLAS 3) computations. The most representative operations of these two BLAS levels are GEMV, which performs a general matrix-vector multiplication by using one input matrix and one input vector and a second vector to store the result, and GEMM, which performs a general matrix-matrix multiplication by using two input matrices and storing the result in a separate third matrix. Similar to level 1, two dominant patterns emerge according to the shape of the matrices: (1) where the matrices involved in the operations are general matrices, such as GEMV or GEMM, and (2) when the input matrix has a triangular form, such as TRSV or SYMM, where the operations are applied to a specific area of the matrices. A triangular matrix is one in which all elements either above or below the main diagonal are zero.

Training and testing datasets: Unlike the previous test case, here the decomposition of the problem becomes more important and relevant. We used a well-known approach that consists of decomposing the input matrices into blocks of rows or columns depending on the operation to ensure an even workload decomposition between MPI ranks (see Fig. 5). Initially, we used only GEMV and GEMM codes for the ChatMPI fine-tuning. This proved insufficient for ChatMPI to provide functional MPI codes for the remaining BLAS 2 and 3 routines that involve triangular matrices. To fill that learning gap, we added the implementation of at least one triangular operation for both BLAS levels (e.g., TRMV for level 2 and TRMM for level 3) during fine-tuning. Similar to BLAS 1 testing, the testing involved a collection of BLAS 2 and 3 routines (sequential implementations to be parallelized with MPI).

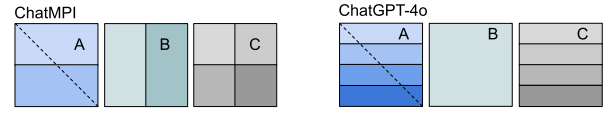


Figure 5: SYMM decomposition for (A and B) input and (C) output matrices used by (left) ChatMPI and (right) ChatGPT-4o and ChatGPT-5.

Analysis: The initial training dataset was supplemented with at least one case that involved triangular matrices for both BLAS 2 and 3. After this addition, ChatMPI provided effective MPI implementations for the last two levels of the BLAS library. In this case, ChatGPT-4o and ChatGPT-5 provided the same functional MPI codes for both BLAS 2 and 3 but used a different approach: it decomposed only a subset of the matrices (e.g., matrices A and C for BLAS level 3 SYMM and GEMM routines) and replicated the same matrix (e.g., matrix B for BLAS level 3 SYMM and GEMM routines) across all the MPI ranks (see Fig. 5). Although this is a valid functional approach, it may impact the performance and scalability due to the imbalance found in the decomposition of the problem.

As shown in Fig. 6, by using a better decomposition, ChatMPI can provide a more scalable and higher-performing code than the one generated by ChatGPT models, achieving up to a 4× speedup when using 64 MPI ranks. Note that the speedup is the ratio between the time consumed by the MPI code generated by ChatGPT models (omitting the time consumed by the initial scatter and final gather MPI-collective communication) and the time consumed by the code

generated by ChatMPI. For this analysis, we used one computational node with one AMD EPYC 7742 (Rome) 64-core CPU and 1 TB of memory. We used OpenMPI (mpicc and mpirun) version 5.0.1 for MPI runs.

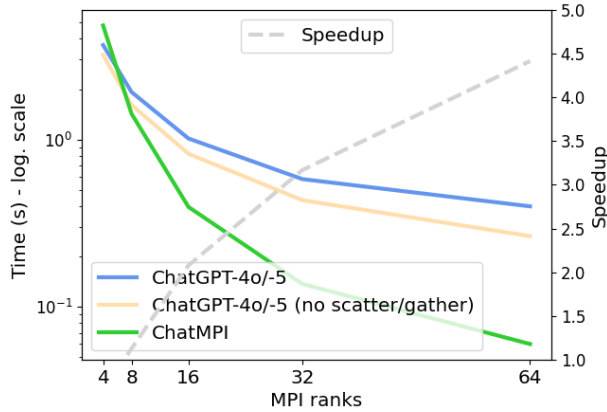


Figure 6: Performance (strong scaling)—in (left) seconds—of the MPI codes generated by ChatGPT-4o and ChatGPT-5 (with and without the initial scatter and final gather collective communication) and ChatMPI for BLAS level 3 SYMM operation ($M, N, K = 2,000$). We also include the (right) speedup achieved by the MPI code generated by ChatMPI over the one generated by ChatGPT models, excluding the time spent on the initial scatter and final gather.

4.2 Sparse

Sparse computation in HPC involves performing calculations on matrices with a high proportion of zero values. Sparse data prevails in diverse HPC applications, including scientific computing and machine learning. Here, we focus on probably the most popular and widely used sparse computation in HPC: SpMV [6]. The relevance of the SpMV kernel is evident in the wide range of vendors and open-source libraries and the many applications that use it [3, 11, 16, 17, 20]. This computation is characterized by the use of sparse formats or methods that store only the information (indexes and values) of the nonzero elements of a sparse matrix. We can find multiple formats to represent a sparse matrix, and the CSR and compressed sparse column (CSC) formats are the most commonly used. The implementation or pattern found in the SpMV implementations can greatly differ depending on the format used to represent the nonzero elements of a sparse matrix. These differences are highly relevant for MPI implementation. For instance, whereas the implementation of SpMV using CSR may not require MPI communication, the implementation using CSC requires a different decomposition of the problem and nontrivial MPI collective communications (see Fig. 7). This makes the problem especially challenging and more demanding than the previous tests cases in terms of the data required for fine-tuning.

Training and testing datasets: Generally, the more recurring and repetitive patterns that exist between prompts (sequential C

codes) and responses (parallel MPI codes), the better the LLM can identify the necessary code transformations. This is demonstrated in the previous test cases, where with a relatively small number of training (JSON) items (question-context-answer tuples), ChatMPI can parallelize multiple sequential codes that, although different from the ones used for the training, share similar patterns. In this case, we used the sequential and MPI implementation of the SpMV using CSR as an initial training dataset. For testing, we used a set of different sequential implementations of SpMV using different formats, such as CSC. This was not enough for ChatMPI to parallelize these codes to MPI correctly. This is due to the important differences between the different implementations (see Fig. 7). To fill that learning gap, we trained ChatMPI with the implementations of SpMV using different formats, making this scenario the most demanding among those tested in this work in terms of required training data for fine-tuning.

Analysis: Both ChatMPI and ChatGPT models (e.g., ChatGPT-4o and ChatGPT-5) provide functional MPI codes for sparse computation. In fact, all these models use similar approaches for problem decomposition and MPI implementation. We analyze the performance impact of the MPI implementation decisions made by ChatGPT models and ChatMPI for SpMV using CSR. All the three models use a similar strategy in terms of decomposition of the problem. They essentially replicate the information about the sparse matrix and x vector and only partition the y vector. This is a common practice in MPI for these kinds of applications. The MPI codes generated by ChatGPT-4o and ChatGPT-5 were identical and generally correct, and used proper MPI syntax; however, they included unnecessary communication. For instance, the codes generated by these model initialize sparse matrix and vector x values in the first MPI rank and distribute (scatter) them among other MPI ranks. This is unnecessary because all the MPI ranks can make the same initialization in parallel, avoiding communication by replicating computation (communication avoiding). This is the approach followed by ChatMPI. All these decisions have important consequences in terms of performance. As shown in Fig. 8, the ChatMPI codes provide speedups between $1.6\times$ and $1.9\times$ compared with those generated by ChatGPT models. For this analysis, we use a synthetic sparse matrix with a sparsity of about 0.001%. The size of the matrix ($M = N$) is doubled from 1,000,000 on 4 MPI ranks up to 8,000,000 on 32 MPI ranks. The conclusions are the same when using other sparse formats.

4.3 Stencil

The stencil computation pattern is widely used in HPC for multiple domains, such as weather prediction, seismic and wave propagation simulations, fluid simulations, image processing, and convolutional neural networks, just to mention a few [42]. These operations are characterized by two factors: (1) the dimension of the domain (e.g., 1D, 2D, 3D, ...) and (2) the degree of freedom or order of complexity [14]. The latter describes the number of points considered for the stencil computation (e.g., 3-point, 5-point, 9-point, ...). Depending on the requirements of the simulation, the precision of the results, or the hardware limitations, one can decide which configuration is more appropriate.

```

int n, nnz;
MPI_Init(NULL, NULL);
int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
int local_n = n / size;
int start = rank * local_n;
int end = start + local_n;
float *x = malloc(n * sizeof(float));
float *y = malloc(local_n * sizeof(float));
float *val = malloc(nnz * sizeof(float));
int *col = malloc(nnz * sizeof(int));
int *row = malloc((n + 1) * sizeof(int));
// Initialization of the matrix and vectors
for (int i = start; i < end; i++) {
    float sum = 0.0f;
    for (int j = row[i]; j < row[i + 1]; j++) {
        sum += val[j] * x[(int)col[j]];
    }
    y[i - start] = sum;
}
free(x);
free(y);
free(val);
free(col);
free(row);
MPI_Finalize();

```

```

int nrow, ncol, nnz;
MPI_Init(&argc, &argv);
int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
int local_ncol = ncol / size;
int start_col = rank * cols_per_proc;
int end_col = start_col + local_ncol;
float *x = malloc(ncol * sizeof(float));
float *y_local = calloc(nrow, sizeof(float));
int *col_ptr = col_ptr = malloc((ncol + 1) * sizeof(int));
int *row_ind = malloc(nnz * sizeof(int));
float *val = malloc(nnz * sizeof(float));
// Initialization of the matrix and vectors
for (int i = start_col; i < end_col; i++) {
    for (int j = col_ptr[i]; j < col_ptr[i + 1]; j++) {
        y_local[row_ind[j]] += val[j] * x[i];
    }
}
float *y_global = NULL;
if (rank == 0) y_global = malloc(nrow * sizeof(float));
MPI_Reduce(y_local, y_global, nrow, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);
free(x);
free(y_local);
free(row_ind);
free(col_ptr);
free(val);
MPI_Finalize();

```

Figure 7: MPI pseudocode for SpMV using (left) CSR and (right) CSC formats.

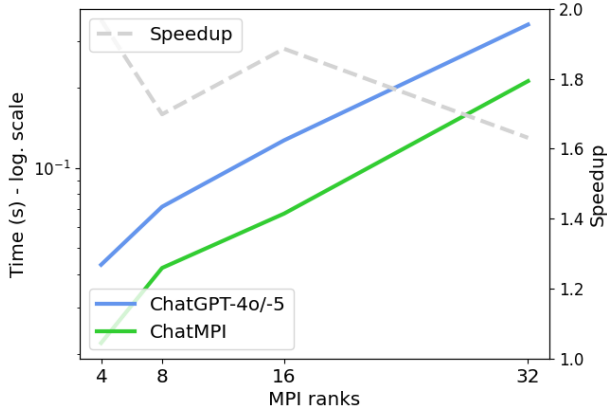


Figure 8: Performance (weak scaling)—in (left) seconds—of the MPI codes generated by ChatGPT models and ChatMPI for SpMV using the CSR format. We also include the (right) speedup achieved by the MPI code generated by ChatMPI over the one generated by ChatGPT-4o/-5.

Training and testing datasets: To cover a wide domain regarding this kind of computation and to analyze the capabilities of modern LLMs for this target in terms of data required for fine-tuning and the quality of the MPI code generated, we divide this analysis into three test cases according to the dimension of the domain. Initially, according to our approach of **not** providing all available information during the initial fine-tuning, we focus on the simplest stencil operations according to the dimensions of the simulation domain: 3-point (1D), 5-point (2D), and 9-point (3D). MPI communication is required for stencil computations to communicate the boundaries of the subdomains assigned to each MPI rank. These boundaries are known as ghost cells, which are used to

store copies of data from neighboring partitions, facilitating computations that require data from adjacent regions without needing to constantly communicate across processors. For training, we decompose the domain into one dimension only (row-major), thereby reducing extra data for communication (ghost cells), minimizing the number of MPI communications, and maximizing performance (see Fig. 9). The exchange of ghost cells between MPI ranks is computed via MPI point-to-point communication using MPI_Send/Recv primitives. For the testing dataset, we use a collection of stencil operations that represent more complex scenarios, increasing either the number of arrays involved in the computation or the number of points of the stencil (e.g., degree of freedom). For instance, for 1D domains, we use sparse tridiagonal matrix-vector multiplication in the testing dataset, whereas for 2D and 3D domains, we use the more complex 9-point (2D) and 11-point (3D) kernels.

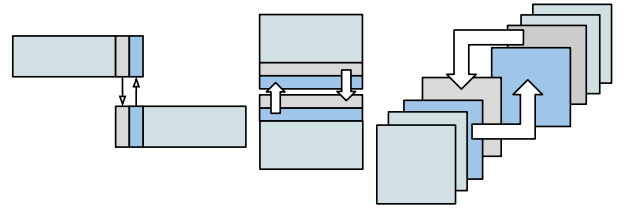


Figure 9: Diagram of the stencil domain (1D, 2D, and 3D) decomposition and communication pattern used for ChatMPI training.

Analysis: After fine-tuning, ChatMPI generates high-quality MPI code for all the test cases—sparse-tridiagonal matrix-vector multiplication (1D), 9-point (2D), and 11-point (3D) stencils—following the patterns described in the training dataset regarding work decomposition and MPI communication (MPI_Send/Recv). Unlike ChatMPI, our base Code Llama model cannot provide any

functional MPI code. On the other hand, ChatGPT-4o provides functional MPI code for the first test case (sparse tridiagonal matrix-vector multiplication) using MPI nonblocking communication (e.g., `MPI_Isend/Irecv`), although that was not explicitly indicated in the prompt. Unfortunately, ChatGPT-4o does not provide functional MPI code for the other two test cases: 9-point (2D) and 11-point (3D) stencil computations. Although the codes generated demonstrate a good decomposition of the domain, very similar to the one used in the ChatMPI training, the codes do not use MPI syntax correctly, failing in the indexes of the arrays used for communication and the MPI rank IDs. In this case, both OpenAI’s models generate different MPI codes. Unlike ChatGPT-4o, ChatGPT-5 generates functional code for all test cases. For the 1D test case, the code generated by this model uses blocking MPI point-to-point `MPI_Sendrecv` primitives. However, for the 2D and 3D tests, ChatGPT-5 uses the same nonblocking communication used by ChatGPT-4o model for the parallelization of the 1D test case.

Fig. 10 illustrates the performance of the MPI codes generated by ChatMPI and both OpenAI’s models (e.g., ChatGPT-4o and ChatGPT-5) for the only test case in which all the models could generate a functional MPI code (i.e., tridiagonal matrix-vector multiplication for a matrix size equal to 50 million). As mentioned before, the MPI codes generated by OpenAI’s model include unnecessary collective communication. In the case of the code generated by ChatGPT-4o, the code includes initial scatter and final gather MPI-collective communications to distribute and collect results using MPI rank 0. In the case of ChatGPT-5, the code generated includes only gather MPI-collective communication at the end of the code. This greatly impacts performance in this test case, adding significant overhead in execution time—approximately one or two orders of magnitude compared with the code generated by ChatMPI. Despite this, all the codes are very similar in terms of workload decomposition. However, whereas the ChatMPI code uses blocking communication, the code generated by ChatGPT-4o uses nonblocking MPI calls (e.g., `MPI_Isend/Irecv`). Although this technique can be viewed as an optimization for overlapping computation with communication, it requires careful use because it does not always provide the expected results. Using nonblocking MPI calls can be efficient for compute-bound operations; however, for memory-bound problems such as the one tested in this analysis, this can be a counterproductive practice, adding extra overhead, particularly when increasing the number of MPI ranks. This is evident in Fig. 10: the code using nonblocking MPI calls (ChatGPT-4o) performs worse than the code that uses blocking MPI calls (ChatMPI). The difference in performance is accentuated when increasing the number of MPI ranks for the same problem size, and the code generated by ChatMPI provides an acceleration of up to 1.4×. ChatGPT-5 generates a code that uses blocking `MPI_Sendrecv` calls. This can be seen as an optimization too. Unlike using `MPI_Send/Recv` calls that serialize communication, `MPI_Sendrecv` calls are computed concurrently. Also the use of `MPI_Sendrecv` calls reduces the number of code of lines compared with the code generated by ChatMPI creating additional dummy MPI ranks using the `MPI_PROC_NULL` constant. However, as we see in the case of using nonblocking MPI calls, the potential benefit of using `MPI_Sendrecv` calls is very application, hardware and implementation dependent. Although the original code generated by ChatGPT-5 is faster than the one

generated by ChatGPT-4o when including the unnecessary initial and end MPI collective communications, when omitting these unnecessary MPI calls, the code using `MPI_Sendrecv` (ChatGPT-5) turns to be up to 2.8× and 2× more time consuming than the codes generated by ChatMPI and ChatGPT-4o respectively. Depending on the number of MPI ranks and size of the problem, the codes generated by ChatGPT-5 using nonblocking point-to-point MPI communication for the 2D and 3D tests perform between 10% and 30% slower when compared with the codes generated by ChatMPI.

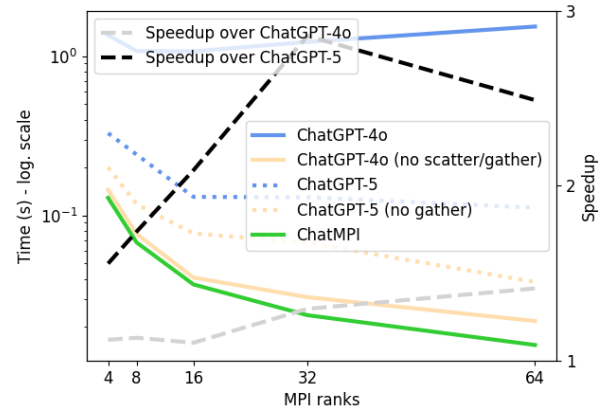


Figure 10: Performance (strong scaling)—in (left) seconds—of the MPI codes generated by ChatGPT-4o, ChatGPT-5 and ChatMPI for tridiagonal matrix-vector multiplication (1D). For completeness, we include the execution time of the code generated by ChatGPT-4o and omit the time spent on the initial scatter and end gather MPI calls. We also include the (right) speedup achieved by the MPI code generated by ChatMPI over the one generated by ChatGPT-4o without scatter/gather calls.

4.4 Iterative solvers

Iterative solvers are an essential part in HPC and scientific computing for solving large, complex systems of equations, particularly those arising from the discretization of partial differential equations. They offer advantages over direct solvers for such problems, especially when dealing with sparse matrices [11]. This is the most complex scenario of this work. Unlike previous scenarios in which we focus essentially on a single kernel, the problem targeted here usually involves multiple kernels or steps. Each step has different requirements for MPI implementation, including workload decomposition, communication, and synchronization. However, the problem must be considered as a whole because any decision made in one of the steps influences how the remaining steps must be implemented (see Fig. 11).

Training and testing datasets: In this case, rather than using any additional training datasets, we evaluate whether ChatMPI can apply the previous learned skills to a new and more challenging domain (e.g., iterative solvers). For testing, we used the conjugate

```

int size, rank;
MPI_Init(NULL, NULL);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
int n = ceil(5000/size) + 2;
float alpha, local_sum, global_sum;
float *u = malloc(n*sizeof(float));
float *u_new = malloc(n*sizeof(float));
float *a1 = malloc(n*sizeof(float));
float *a2 = malloc(n*sizeof(float));
float *a3 = malloc(n*sizeof(float));
// Initialization of matrix and vectors
// MPI point-to-point communication
if (rank > 0) {
    MPI_Send(&u[1], 1, MPI_FLOAT, rank - 1, 0, MPI_COMM_WORLD);
    MPI_Recv(&u[0], 1, MPI_FLOAT, rank - 1, 0, MPI_COMM_WORLD, &status);
}
if (rank < size-1) {
    MPI_Recv(&u[n-1], 1, MPI_FLOAT, rank + 1, 0, MPI_COMM_WORLD, &status);
    MPI_Send(&u[n-2], 1, MPI_FLOAT, rank + 1, 0, MPI_COMM_WORLD);
}
// Tridiagonal matrix-vector multiplication
for (int i = 1; i < n-1; i++) {
    u_new[i] = a1[i]*u[i+1] + a2[i]*u[i] + a3[i]*u[i-1];
}
// BLAS 1 AXPY
for (int i = 0; i < n; i++) {
    u_new[i] = alpha * u[i] + u_new[i];
}
//BLAS 1 DOT
for (i = 0; i < n; i++) {
    local_sum += u[i] * u_new[i];
}
// MPI collective communication
MPI_Reduce(&local_sum, &global_sum, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);
free(u); ...
MPI_Finalize();

```

Figure 11: MPI pseudocode for the main steps of the CG algorithms.

gradient (CG) algorithm [5]. CG is a well-known and widely used iterative method for solving sparse systems of linear equations. This algorithm is a key component in a large variety of HPC applications, including finite-difference and finite-element methods, partial differential equations, structural analysis, circuit analysis, and many more linear algebra-related problems [31]. Given the ubiquity and importance of this operation in HPC applications, CG is also used to measure supercomputer performance through the popular High Performance Conjugate Gradients benchmark [11]. For the testing dataset (prompt), we used a sequential implementation of a plain CG algorithm [33] without a preconditioner. This simplifies the study of the optimizations by eliminating the preconditioning step. To this end, we generated a diagonal-dominant tridiagonal sparse matrix [37–39], which is commonly used in these contexts. The sequential implementation of CG comprises multiple BLAS 1 kernels to compute the different steps of the algorithm, which consists of a series of DOT and AXPY computations, among other operations. The MPI implementation of CG must combine both stencil and nonstencil computations. Although the computations of the matrix-vector multiplication must be made by using ghost cells, that is not the case in the remaining steps of the algorithm. Additionally, this implementation involves point-to-point communication and collective communication MPI primitives in the same code.

Analysis: Both, ChatMPI and ChatGPT-5 are the only models that can generate functional MPI codes. Although most of the code lines of the MPI implementation generated by ChatGPT-4o were

correct, it incorrectly handled the MPI rank IDs used for MPI point-to-point communication (send and receive). The MPI code generated by ChatMPI applies the previously learned HPC techniques for problem decomposition and communication. These techniques are equivalent to the ones used in the previous tests cases, resulting in an MPI code that provides good performance and scalability, very similar to the previously reported results. The code generated by ChatGPT-5, although it is very similar to the one generated by ChatMPI, it uses nonblocking MPI calls `MPI_Isend/Irecv`. Unlike the code generated by ChatGPT-4o for 1D stencil computation that uses nonblocking MPI calls to overlap communication with computation involving non-trivial optimizations, in the code generated by ChatGPT-5, although it uses the same nonblocking MPI calls, it is not possible to overlap communication with computation since a strong synchronisation (e.g., `MPI_Waitall`) is found right after the communication. This has important consequences in terms of performance when compared with the code generated by ChatMPI which is up to more than 2× faster than the code generated by the OpenAI’s model (see Figure 12).

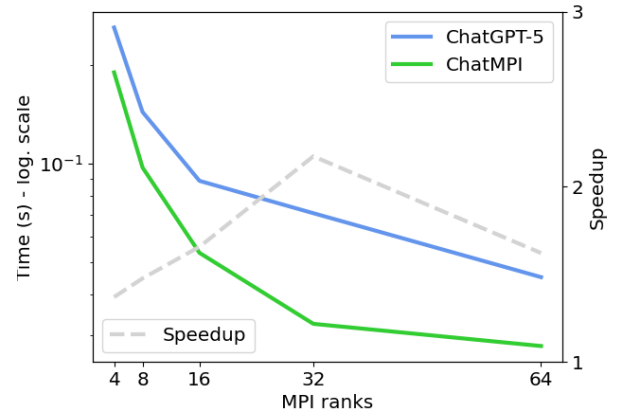


Figure 12: Performance (weak scaling)—in (left) seconds—of the MPI codes generated by ChatGPT models and ChatMPI for SpMV using the CSR format. We also include the (right) speedup achieved by the MPI code generated by ChatMPI over the one generated by ChatGPT-4o/-5.

4.5 Remarks

We summarize the results collected for the six essential HPC workloads studied in this work in terms of the training data size required for fine-tuning as well as the correctness and performance of the MPI codes generated (see Table 1). Depending on the target, the required data for fine-tuning can greatly differ. For instance, for sparse computations (e.g., SpMV), we must train the model for each sparse format (e.g., CSR or CSC) due to the significant differences between MPI implementations. This requirement is considerably smaller for BLAS or stencil computations, where using a relatively small number of representative cases—which capture the patterns associated with these computations—is sufficient for ChatMPI to generate high-quality MPI codes, even for more complex scenarios

than those used during training. Notably, no training (fine-tuning) on iterative solvers (e.g., CG) was necessary for ChatMPI to generate high-quality MPI codes for this class of problems. These algorithms essentially comprise sets of kernels that are either identical or very similar to the ones used for the previously trained scientific workloads.

With the exception of the Code Llama base model, both OpenAI’s model (ChatGPT-4o and ChatGPT-5) and ChatMPI achieve strong results in terms of correctness. Whereas ChatGPT-4o provides correct codes for about 75% of the tests, both ChatGPT-5 and ChatMPI provides functional MPI codes in all the test cases. Note that ChatMPI was trained specifically for these targets, using the necessary data and fine-tuning iterations to find and fill the learning gaps during the training. Moreover, we observe significant differences in performance between the MPI codes generated by ChatGPT models and ChatMPI. Although most of the generated MPI codes are functional, the workload decomposition, MPI communication, and optimizations employed by ChatGPT models proved less effective at achieving good scalability. By contrast, the fine-tuning performed for ChatMPI enables the model to deliver higher performance by using better approaches for problem decomposition, thereby achieving up to a 4× speedup for some BLAS 3 routines; communication-avoiding strategies for sparse computation, thereby achieving up to a 2× speedup for SpMV using CSR; and the effective use of MPI primitives for 1D stencil and iterative computations, thereby providing accelerations of up to 40% and higher than 2× for stencil and CG computations respectively.

5 Related Work

The past 3 years have seen a growing interest in evaluating the impact of state-of-the-art LLMs on HPC software [32], and this is becoming a very active area of research. For example, LM4HPC [7] is a language model specifically designed for HPC and demonstrates success in specific tasks such as code similarity analysis, parallelism detection, and OpenMP question answering. Nichols et al. [25] created the trained HPC-Coder model based on the DeepSpeed [28] deep learning model. They found varying degrees of success in code completion for OpenMP pragmas and MPI calls. Another example is LLM4VV [24], which is a fine-tuned model that explores the capabilities of GPT-4 and Llama 2 to generate tests for the compiler functionality of the directives-based OpenACC parallel programming model and shows a high degree of success. Some recent work has explored using LLMs to translate Fortran codes: Lei et al. [19] created a dataset to train LLMs for HPC code translation from Fortran OpenMP to C++ OpenMP, and Zhou et al. [41] proposed a proof-of-concept that uses LLMs to translate single-threaded Fortran to Python/JAX. Godoy et al. [12, 15] and Valero-Lara et al. [36] evaluated LLM capabilities, which included GitHub Copilot and Llama 2 for generating correct and functional representative HPC kernels on modern HPC programming models. These last references established some best practices and criteria to use when interacting with LLMs for HPC targets, using source code in prompts to minimize prompt-engineering efforts and maximizing the quality of the LLM responses. Dearing et al. [8] present LASSI, which is an automated pipeline framework designed to translate between parallel CUDA and OpenMP programming languages by

bootstrapping existing closed- or open-source LLMs. This work was recently extended to LASSI-EE, an automated LLM-based refactoring framework that generates energy-efficient CUDA code [9]. More recently, Valero-Lara et al. [35] developed ChatBLAS as the first AI-generated, portable basic vector-vector (BLAS level 1) library, which achieved competitive or higher performance compared with vendor-specific math libraries by using prompt engineering and fine-tuning on top of ChatGPT models. More related to MPI, Schneider et al. [30] created MPICodeCorpus, a collection of more than 15,000 MPI codes used for training purposes. MPICodeCorpus was used to create MPI-RICAL, which is a programming assistant for MPI coding.

6 Observations

Observation 1—Fine-tuning: We demonstrated the effectiveness of using ChatHPC to rapidly create trustworthy MPI capabilities with high levels of correctness compared with state-of-the-art LLMs, such as OpenAI’s 1.8 trillion parameter ChatGPT-4o and ChatGPT-5 models. Serial-to-MPI parallelization is a nontrivial process that requires knowledge of not only computation but also communication and domain decomposition.

Observation 2—Expert-in-the-loop: ChatMPI’s capabilities improved considerably by involving human expertise in the fine-tuning process to create the training data and identify learning gaps.

Observation 3—Overfitting: The lack of diverse information or data in the MPI domain, may result in overfitting, which makes the AI assistants more deterministic and less creative. However, the impact of overfitting in this particular scenario—that is, creating a highly specialized AI assistant for automatic MPI parallelization—is not necessarily a negative.

Observation 4—Code completeness: The source codes used for both fine-tuning and testing must be self-contained and readable, prioritizing readability over brevity. This helps ChatMPI learn about the patterns and code transformations required for the different HPC workloads studied.

Observation 5—Correctness and data: Using very precise data (e.g., source codes) for both fine-tuning and testing produces very high levels of correctness even when using relatively small training datasets. However, that depends on the particular problem; some targets (e.g., SpMV) require significantly more data to reach a high level of correctness.

Observation 6—MPI performance: ChatMPI codes demonstrate good HPC capabilities, outperforming the MPI codes generated by ChatGPT-4o and ChatGPT-5 by using more effective HPC techniques. This finding highlights the importance of training these models on HPC fundamentals to generate not only functional codes using HPC models, such as MPI, but also codes that achieve good performance.

Observation 7—Simplify prompting: Communication with ChatMPI does not require any prior domain knowledge, thereby reducing the expertise required for the LLMs to provide good quality results.

Table 1: Results for Code Llama, ChatMPI, ChatGPT-4o and ChatGPT-5, showing the training data size required by ChatMPI for fine-tuning, correctness (functional MPI codes generated), and performance.

		Correctness				Peak ChatMPI Speedup over	
HPC Domains	ChatMPI Training Size	Code Llama	ChatMPI	ChatGPT-4o	ChatGPT-5	ChatGPT-4o	ChatGPT-5
BLAS 1, 2, 3	Small	0.0%	100.0%	100.0%	100.0%	4×	4×
Sparse (SpMV)	Large	0.0%	100.0%	100.0%	100.0%	1.9×	1.9×
1D, 2D, 3D Stencil	Medium	0.0%	100.0%	33.3%	100.0%	1.4×	2.8×
Iterative (CG)	No data needed	0.0%	100.0%	0.0%	100.0%	—	2.1×

7 Conclusions and Future Work

Created with the ChatHPC library, ChatMPI delivers higher trustworthiness and performance levels than state-of-the-art LLMs (Code Llama and the much larger ChatGPT-4o and ChatGPT-5 models) for the MPI parallelization of essential HPC workloads, including BLAS levels 1, 2, and 3 as well as sparse, stencil, and iterative applications. This work emphasizes the importance of using high-quality training data that includes skills and knowledge essential for developing the targeted capabilities. In this study, these skills involve the effective use of HPC techniques that enable ChatMPI to generate well-performing MPI code tailored to the problem (HPC workload) being parallelized. With relatively small training datasets (just a few dozen prompts) supervised by experts, we created ChatMPI rapidly on very modest computational resources (one node with two NVIDIA GPUs) by fine-tuning the open-source Code Llama LLM and using ChatHPC capabilities. Moreover, the HPC community can easily extend the existing ChatMPI capabilities with other capabilities or context, such as MPI+X translations, without replicating the base LLM model, thereby providing an efficient mechanism for leveraging AI to facilitate MPI software development.

In future work, we plan to explore new capabilities for MPI parallelization—such as MPI+X for targeting different architectures, among other MPI optimizations—as well as address different application domains that may require different techniques beyond those studied in this work.

Acknowledgments

This research used resources of the Experimental Computing Laboratory (ExCL) at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. This material is based upon work supported by the US Department of Energy, Office of Science’s Advanced Scientific Computing Research program as part of the Advancements in Artificial Intelligence for Science program, Ellora and Durban projects, the Next Generation of Scientific Software Technologies program, S4PST and PESO projects, and the Scientific Discovery through Advanced Computing (SciDAC) program, RAPIDS2 SciDAC Institute for Computer Science, Data, and Artificial Intelligence.

References

- [1] 2025. Hugging Face. <https://huggingface.co/>. [Online; accessed 18-March-2025].
- [2] 2025. Ollama. <https://ollama.com/>. [Online; accessed 18-March-2025].
- [3] Patrick Amestoy, Iain S. Duff, Jean-Yves L’Excellent, and Jacko Koster. 2000. MUMPS: A General Purpose Distributed Memory Sparse Solver. In *Applied Parallel Computing, New Paradigms for HPC in Industry and Academia, 5th International Workshop, PARA 2000 Bergen, Norway, June 18–20, 2000 Proceedings (Lecture Notes in Computer Science, Vol. 1947)*, Tor Sorevik, Fredrik Manne, Randi Moe, and Assefaw Hadish Gebremedhin (Eds.). Springer, 121–130. https://doi.org/10.1007/3-540-70734-4_16
- [4] M. Scot Breitenfeld, Houjun Tang, Huihuo Zheng, Jordan Henderson, and Suren Byna. 2025. HDF5 in the exascale era: Delivering efficient and scalable parallel I/O for exascale applications. *Int. J. High Perform. Comput. Appl.* 39, 1 (2025), 65–78. <https://doi.org/10.1177/10943420241288244>
- [5] Sandra Catalán, Xavier Martorell, Jesús Labarta, Tetsuzo Usui, Leonel Antonio Toledo Díaz, and Pedro Valero-Lara. 2019. Accelerating Conjugate Gradient using OmpSs. In *20th International Conference on Parallel and Distributed Computing, Applications and Technologies, PDCAT 2019, Gold Coast, Australia, December 5–7, 2019*. IEEE, 121–126. <https://doi.org/10.1109/PDCAT46702.2019.00033>
- [6] Sandra Catalán, Tetsuzo Usui, Leonel Toledo, Xavier Martorell, Jesús Labarta, and Pedro Valero-Lara. 2020. Towards an Auto-Tuned and Task-Based SpMV (LASs Library). In *OpenMP: Portable Multi-Level Parallelism on Modern Systems - 16th International Workshop on OpenMP, IWOMP 2020, Austin, TX, USA, September 22–24, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12295)*, Kent F. Milfeld, Bronis R. de Supinski, Lars Koesterke, and Jannis Klinkenberg (Eds.). Springer, 115–129. https://doi.org/10.1007/978-3-030-58144-2_8
- [7] 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. In *OpenMP: Advanced Task-Based, Device and Compiler Programming*, Simon McIntosh-Smith, Michael Klemm, Bronis R. de Supinski, Tom Deakin, and Jannis Klinkenberg (Eds.). Springer Nature Switzerland, Cham, 18–33.
- [8] Matthew T. Dearing, Yiheng Tao, Xingfu Wu, Zhiling Lan, and Valerie Taylor. 2024. LASSI: An LLM-Based Automated Self-Correcting Pipeline for Translating Parallel Scientific Codes. In *IEEE International Conference on Cluster Computing, CLUSTER - Workshops 2024, Kobe, Japan, September 24–27, 2024*. IEEE, 136–143. <https://doi.org/10.1109/CLUSTERWORKSHOPS61563.2024.00029>
- [9] Matthew T. Dearing, Yiheng Tao, Xingfu Wu, Zhiling Lan, and Valerie Taylor. 2025. Leveraging LLMs to Automate Energy-Aware Refactoring of Parallel Scientific Codes. *CoRR abs/2505.02184* (2025). <https://doi.org/10.48550/ARXIV.2505.02184> arXiv:2505.02184
- [10] Jack J. Dongarra. 2002. Basic Linear Algebra Subprograms Technical (BLAST) Forum Standard (2). *Int. J. High Perform. Comput. Appl.* 16, 2 (2002), 115–199.
- [11] Jack J. Dongarra, Victor Eijkhout, and Henk A. van der Vorst. 2001. An iterative solver benchmark. *Sci. Program.* 9, 4 (2001), 223–231. <https://doi.org/10.1155/2001/527931>
- [12] 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* (, Salt Lake City, UT, USA.), (ICPP Workshops ’23). Association for Computing Machinery, New York, NY, USA, 136–144. <https://doi.org/10.1145/3605731.3605886>
- [13] William F. Godoy, Norbert Podhorszki, Ruonan Wang, Chuck Atkins, Greg Eisenhauer, Junmin Gu, Philip E. Davis, Jong Choi, Kai Germaschewski, Kevin A. Huck, Axel Huebl, Mark Kim, James Kress, Tahsin M. Kurç, Qing Liu, Jeremy Logan, Kshitij Mehta, George Ostrochov, Manish Parashar, Franz Poeschel, David Pugmire, Eric Suchyta, Keichi Takahashi, Nick Thompson, Seiji Tsutsumi, Lipeng Wan, Matthew Wolf, Kesheng Wu, and Scott Klasky. 2020. ADIOS 2: The Adaptable Input Output System. A framework for high-performance data management. *SoftwareX* 12 (2020), 100561. <https://doi.org/10.1016/j.softx.2020.100561>
- [14] William F. Godoy, Pedro Valero-Lara, Caira Anderson, Katrina W. Lee, Ana Gainaru, Rafael Ferreira da Silva, and Jeffrey S. Vetter. 2023. Julia as a unifying end-to-end workflow language on the Frontier exascale system. In *Proceedings of the SC ’23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis, SC-W 2023, Denver, CO, USA, November 12–17, 2023*. ACM, 1989–1999. <https://doi.org/10.1145/3624062.3624278>
- [15] William F. Godoy, Pedro Valero-Lara, Keita Teranishi, Prasanna Balaprakash, and Jeffrey S. Vetter. 2024. Large language model evaluation for high-performance computing software development. *Concurr. Comput. Pract. Exp.* 36, 26 (2024). <https://doi.org/10.1002/CPE.8269>
- [16] Johan Hoffman, Johan Jansson, Niyazi Cem Degirmenci, Jeannette Hiromi Spühler, Rodrigo Vilela De Abreu, Niclas Jansson, and Aurélien Larcher. 2016.

- FEniCS-HPC: Coupled Multiphysics in Computational Fluid Dynamics. In *High-Performance Scientific Computing - First JARA-HPC Symposium, JHPCS 2016, Aachen, Germany, October 4-5, 2016, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 10164)*, Edoardo Di Napoli, Marc-André Hermanns, Hristo Iliev, Andreas Lintermann, and Alexander Peyser (Eds.). Springer, 58–69. https://doi.org/10.1007/978-3-319-53862-4_6
- [17] Paul D. Hovland and Lois C. McInnes. 2001. Parallel simulation of compressible flow using automatic differentiation and PETSc. *Parallel Comput.* 27, 4 (2001), 503–519. [https://doi.org/10.1016/S0167-8191\(00\)00074-0](https://doi.org/10.1016/S0167-8191(00)00074-0)
- [18] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-Rank Adaptation of Large Language Models. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25–29, 2022*. OpenReview.net. <https://openreview.net/forum?id=nZeVKeeFY9>
- [19] Bin Lei, Caiwen Ding, Le Chen, Pei-Hung Lin, and Chunhua Liao. 2023. Creating a Dataset for High-Performance Computing Code Translation using LLMs: A Bridge Between OpenMP Fortran and C++. In *IEEE High Performance Extreme Computing Conference, HPEC 2023, Boston, MA, USA, September 25–29, 2023*. IEEE, 1–7. <https://doi.org/10.1109/HPEC58863.2023.10363534>
- [20] Xiaoye Sherry Li, James Demmel, John R. Gilbert, Laura Grigori, and Meiyue Shao. 2011. SuperLU. In *Encyclopedia of Parallel Computing*, David A. Padua (Ed.). Springer, 1955–1962. https://doi.org/10.1007/978-0-387-09766-4_95
- [21] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the Middle: How Language Models Use Long Contexts. *Trans. Assoc. Comput. Linguistics* 12 (2024), 157–173. https://doi.org/10.1162/TACL_A_00638
- [22] Narasinga Rao Miniskar, Alaul Haque Monil Mohammad, alero-Lara Pedro, Frank Liu, and Jeffrey S Vetter. 2022. IRIS-BLAS: Towards a Performance Portable and Heterogeneous BLAS Library. In *29th IEEE International Conference on High Performance Computing, Data, and Analytics, HiPC 2022, Bengaluru, India, December 18–21, 2022*. IEEE.
- [23] Mohammad Alaul Haque Monil, Narasinga Rao Miniskar, Keita Teranishi, Jeffrey S. Vetter, and Pedro Valero-Lara. 2023. MatRIS: Multi-level Math Library Abstraction for Heterogeneity and Performance Portability using IRIS Runtime. In *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis, SC-W 2023, Denver, CO, USA, November 12–17, 2023*. ACM, 1081–1092. <https://doi.org/10.1145/3624062.3624184>
- [24] Christian Munley, Aaron Jarmusch, and Sunita Chandrasekaran. 2024. LLM4VV: Developing LLM-driven test suite for compiler validation. *Future Generation Computer Systems* 160 (2024), 1–13. <https://doi.org/10.1016/j.future.2024.05.034>
- [25] Daniel Nichols, Aniruddha Marathe, Harshitha Menon, Todd Gamblin, and Abhinav Bhatele. 2024. HPC-Coder: Modeling Parallel Programs using Large Language Models. In *ISC High Performance 2024 Research Paper Proceedings (39th International Conference)*. 1–12. <https://doi.org/10.23919/ISC.2024.10528929>
- [26] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. *PyTorch: an imperative style, high-performance deep learning library*. Curran Associates Inc., Red Hook, NY, USA.
- [27] Ofir Press, Noah A. Smith, and Mike Lewis. 2022. Train Short, Test Long: Attention with Linear Biases Enables Input Length Extrapolation. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25–29, 2022*. OpenReview.net. <https://openreview.net/forum?id=R8sQPpGCv0>
- [28] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (Virtual Event, CA, USA) (KDD '20)*. Association for Computing Machinery, New York, NY, USA, 3505–3506. <https://doi.org/10.1145/3394486.3406703>
- [29] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xi-aoping Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. *CoRR abs/2308.12950* (2023). <https://doi.org/10.48550/ARXIV.2308.12950> arXiv:2308.12950
- [30] Nadav Schneider, Tal Kadosh, Niranjan Hasabnis, Timothy G. 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, SC-W 2023, Denver, CO, USA, November 12–17, 2023*. ACM, 1–10. <https://doi.org/10.1145/3624062.3624063>
- [31] Jonathan R Shewchuk. 1994. *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain*. Technical Report. USA.
- [32] Keita Teranishi, Harshitha Menon, William F. Godoy, Prasanna Balaprakash, David Bau, Tal Ben-Nun, Abhinav Bhatele, Franz Franchetti, Michael Franusich, Todd Gamblin, Giorgis Georgakoudis, Tom Goldstein, Arjun Guha, Steven Hahn, Costin Iancu, Zheming Jin, Terry Jones, Tze Meng Low, Het Mankad, Narasinga Rao Miniskar, Mohammad Alaul Haque Monil, Daniel Nichols, Konstantinos Parasyris, Swaroop Pophale, Pedro Valero-Lara, Jeffrey S. Vetter, Samuel Williams, and Aaron Young. 2025. Leveraging AI for Productive and Trustworthy HPC Software: Challenges and Research Directions. *arXiv:2505.08135 [cs.SE]* <https://arxiv.org/abs/2505.08135>
- [33] Leonel Toledo, Pedro Valero-Lara, Jeffrey S. Vetter, and Antonio J. Peña. 2021. Static Graphs for Coding Productivity in OpenACC. In *28th IEEE International Conference on High Performance Computing, Data, and Analytics, HiPC 2021, Bengaluru, India, December 17–20, 2021*. IEEE, 364–369. <https://doi.org/10.1109/HiPC53243.2021.00050>
- [34] Pedro Valero-Lara, Sandra Catalán, Xavier Martorell, and Jesús Labarta. 2019. BLAS-3 Optimized by OmpSs Regions (LASs Library). In *27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2019, Pavia, Italy, February 13–15, 2019*. IEEE, 25–32. <https://doi.org/10.1109/EMPDP.2019.8671545>
- [35] Pedro Valero-Lara, William F. Godoy, Keita Teranishi, Prasanna Balaprakash, and Jeffrey S. Vetter. 2024. ChatBLAS: The First AI-Generated and Portable BLAS Library. In *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis, Atlanta, GA, USA, November 17–22, 2024*. IEEE, 19–24. <https://doi.org/10.1109/SCW63240.2024.00010>
- [36] 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:2309.07103 [cs.SE]*
- [37] Pedro Valero-Lara, Ivan Martínez-Pérez, Raúl Sirvent, Xavier Martorell, and Antonio J. Peña. 2017. NVIDIA GPUs Scalability to Solve Multiple (Batch) Tridiagonal Systems Implementation of cuThomasBatch. In *Parallel Processing and Applied Mathematics - 12th International Conference, PPAM 2017, Lublin, Poland, September 10–13, 2017, Revised Selected Papers, Part I*. 243–253.
- [38] Pedro Valero-Lara, Ivan Martínez-Pérez, Raúl Sirvent, Xavier Martorell, and Antonio J. Peña. 2018. cuThomasBatch and cuThomasVBatch, CUDA Routines to compute batch of tridiagonal systems on NVIDIA GPUs. *Concurrency and Computation: Practice and Experience* 30, 24 (2018).
- [39] Valero-Lara, Pedro, Martínez-Pérez, Ivan, Sirvent, Raúl, Peña, Antonio J., Martorell, Xavier, and Labarta, Jesús. 2018. Simulating the behavior of the Human Brain on GPUs. *Oil Gas Sci. Technol. - Rev. IFP Energies nouvelles* 73 (2018), 63. <https://doi.org/10.2516/ogst/2018061>
- [40] David W Walker and Jack J Dongarra. 1996. MPI: a standard message passing interface. *Supercomputer* 12 (1996), 56–68.
- [41] Anthony Zhou, Linnia Hawkins, and Pierre Gentine. 2024. Proof-of-concept: Using ChatGPT to Translate and Modernize an Earth System Model from Fortran to Python/JAX. *CoRR abs/2405.00018* (2024). <https://doi.org/10.48550/ARXIV.2405.00018> arXiv:2405.00018
- [42] Hamid Reza Zohouri, Artur Podobas, and Satoshi Matsuoka. 2018. High-Performance High-Order Stencil Computation on FPGAs Using OpenCL. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 123–130. <https://doi.org/10.1109/IPDPSW.2018.00027>