Experience using AI in MPI Test Suite Development: Implications for Educators

Callie Stewart
Department of Computer Science
Tennessee Technological University
Cookeville, TN, USA
castewart@tntech.edu

Gerald C. Gannod
Department of Computer Science
Tennessee Technological University
Cookeville, TN, USA
jgannod@tntech.edu

***Abstract*—The integration of artificial intelligence (AI) into software development has introduced new possibilities for automating complex coding tasks. This study investigates the role of AI, specifically GitHub Copilot, in generating Message Passing Interface (MPI) test suites and its implications for Parallel and Distributed Computing (PDC) education. Key findings highlight the effectiveness of AI-assisted development, the importance of prompt engineering, and the necessity of domain expertise. The study provides recommendations for educators on integrating AI tools into curriculum design while ensuring students develop critical programming skills. Future directions include enhancing AI capabilities for parallel computing and developing domain-specific models.**

I. Introduction

Advancements in artificial intelligence (AI) have introduced tools that enhance productivity in software development. For example, GitHub Copilot [1] generates meaningful code suggestions, reduces cognitive load, and automates repetitive tasks. However, its potential in Parallel and Distributed Computing (PDC) education remains underexplored.

Through the process of automating portions of the MPI test suite development with Copilot, three key lessons emerged. First, domain knowledge is essential–while Copilot generates routine code, human expertise is needed to refine outputs, ensure MPI standard compliance, and address challenges like synchronization, deadlocks, and race conditions.

Second, iterative prompt engineering significantly improves AI-generated code quality. Refining prompts by adjusting details and specificity proved an effective strategy and a structured framework for teaching PDC concepts.

Third, AI tools like Copilot lower barriers for students new to PDC, shifting focus from syntax to higher-level problem-solving. However, Copilot struggles with advanced MPI operations, necessitating human guidance.

This paper explores how these lessons can be applied to educational practices, helping students navigate the complexities of PDC while leveraging AI tools effectively.

II. Motivation and Background

The motivation for this study stems from the increasing complexity of teaching and learning PDC concepts, which are foundational to modern computing systems. Topics such as synchronization, fault tolerance, and communication patterns are notoriously challenging for students due to their abstract nature and the difficulty of visualizing and debugging concurrent processes. Educators face the dual challenge of designing curriculum elements that effectively convey these concepts while keeping students engaged and motivated [2].

Advancements in AI tools, such as GitHub Copilot, offer a novel way to address these challenges. Additionally, other generative AI tools such as Tabnine [3], Amazon CodeWhisperer [4], and OpenAI's ChatGPT [5] have emerged as powerful programming assistants. Tabnine focuses on context-aware code completions, leveraging AI models to predict and suggest code snippets tailored to the developer's project. Amazon CodeWhisperer offers capabilities similar to Copilot but integrates seamlessly with AWS services, helping developers write infrastructure-specific code efficiently. OpenAI's ChatGPT, on the other hand, functions as a conversational assistant, providing step-by-step explanations, debugging assistance, and coding examples, making it an excellent resource for both novice and experienced programmers [1], [6], [7].

Existing research highlights the role of AI in improving software development workflows and its potential in generating unit tests, debugging, and reducing repetitive tasks. However, there is a gap in understanding how these capabilities translate into educational benefits, particularly in the context of PDC. Studies such as those by Zhang et al. [8] and Pandey et al. [7] have demonstrated the efficiency improvements provided by AI, especially for less experienced developers. Similarly, domain-specific tools like MPIrigen [9] and HPC-Coder [2] have been shown to address the unique challenges of parallel programming, such as managing communication patterns and synchronization. These findings suggest that AI can bridge the gap for students with limited experience, but they also emphasize the need for human expertise to validate and refine AI outputs.

III. Methodology

The design of this study considered many factors including scope, methodology, AI models, and testing frameworks. The scope was deliberately narrowed to generating test cases for MPI Point-to-Point operations [10], as the foundational calls to PDC. This focus allowed for a systematic exploration of AI-generated outputs while aligning with educational goals [2].

The methodology combined iterative prompt engineering with multi-layered testing to assess GitHub Copilot's effectiveness. Copilot was chosen for its integration as an extention for popular IDEs, enabling real-time code generation and refinement without external tools or manual API interactions. It is powered by OpenAI's Codex, known for its advanced code generation capabilities [11]. As demonstrated through this research, students using AI-assisted coding tools like Copilot should have foundational knowledge of programming, debugging, and evaluating AI-generated code, aligning with other generative AI tools that require critical assessment rather than passive acceptance.

The primary developer was a master's student with experience in fundamental MPI operations but limited exposure to less commonly used calls. While routine MPI functions were straightforward, advanced operations required additional learning and refinement. The study examined how AI-assisted coding supports this learning process. The developer's adaptation was monitored to assess Copilot's strengths and limitations. A structured prompt engineering methodology was applied, starting with broad prompts and progressively refining constraints based on Copilot's suggestions. This process involved:

Baseline Prompting – The student initially provided a general request, such as "Generate an MPI test case for point-to-point communication using `MPI_Send` and `MPI_Recv`."

Result Evaluation – The AI-generated code was reviewed for correctness, completeness, and adherence to MPI standards.

Incremental Refinement – If the generated output was incomplete or incorrect, additional constraints were added to the prompt, such as specifying data types, buffer sizes, message tags, or expected synchronization behaviors.

Error Debugging and Prompt Adjustment – When AI-generated code contained errors or failed edge cases, debugging insights were used to refine the next prompt iteration to emphasize missing considerations.

Final Verification and Testing – The final AI-generated code was validated using CUnit to ensure correctness, robustness, and adherence to MPI standards.

This iterative approach helped the student gain a deeper understanding of how Copilot interprets and responds to prompts, highlighting the necessity of human intervention and expertise in guiding AI-assisted code generation.

To generate the code for the test suite, each prompt followed a structured format to ensure consistency in Copilot's responses. Each prompt included the function signature—specifying return type, function name, and parameter type(s) to ensure uniform function calls within the test suite. Additionally, the prompt outlined the specific MPI operations to be employed and described expected error handling. This structure was the keystone to guiding Copilot toward implementing a consistent test suite that demonstrated behaviors aligned with the MPI standard. Once the prompt was prepared, Copilot was triggered to generate suggestions, and manual inspection determined which of these adhered to coding best practices and closely aligned with the expected MPI behavior.

Figure 1 illustrates both the prompt and the generated code for the `scancel` function—an example where this structure proved successful and no refinement was needed. Copilot was asked to generate a function that canceled a standard, nonblocking send before it is received. Eight suggestions were generated but differed only by additional, needless functions generated after `scancel` that were not requested. The first suggestion was accepted, and the additional code was removed. Upon testing, the function returned no unexpected behaviors (passed).

Figure 2 demonstrates another example with the function `iterative_isend_irecv` where multiple options had to be weighed against each other. Copilot was asked to generate a function that iteratively used standard, nonblocking send and receive operations to transfer multiple array elements across processes, validating that the data was sent correctly. Five suggestions were generated, but they mostly differed only by whitespace or inconsequential details. The solution that was accepted demonstrated the best practice by generating new data between each iteration.

Figure 3 shows the final iteration of code and prompt for `request_status_null`, which tests whether `MPI_Request_get_status` behaves correctly with a null request using `MPI_REQUEST_NULL`. The issue encountered that required prompt refinement was a lack of complete adherence to the MPI Standard. While the suggestions Copilot provided properly utilized the operations, they did not fully test the results specified in the MPI standard for `MPI_Request_get_status` (excerpted below) [10]:

> One is allowed to call `MPI_REQUEST_GET_STATUS` with a null or inactive request argument. In such a case, the operation returns with flag=true and empty status.

The suggestions generated by Copilot failed to test both outlined criteria. A solution was accepted that properly tested the flag value. An additional iteration was performed where a new prompt was inserted after the flag validation, asking Copilot to test status and, if not empty, increment the number of unexpected behaviors. Upon this clarification, Copilot was able to generate the missing test and the function passed.

IV. Assessment of Methodology and Results

The study revealed several key insights into the strengths and limitations of using GitHub Copilot to generate MPI test suites. One of the primary findings was the importance of prompt specificity. Early iterations demonstrated that vague or generalized prompts often led to incomplete or incorrect solutions. By refining prompts to include precise details, such as specific MPI operations and expected behaviors, the quality of the AI-generated outputs improved significantly. This iterative refinement process highlights the critical role of human expertise in guiding AI tools [1], [11].

The iterative prompt engineering approach proved effective, requiring only 1.5 iterations on average to generate valid code. Each iteration produced 8.87 suggestions, with 3.23 typically meeting prompt constraints and MPI standards. Functions

```
/* PROMPT: int scancel(MPI_Comm comm) evaluates
    ↪ MPI_Isend and MPI_Cancel for canceling non-
    ↪ blocking communication requests. Unexpected
    ↪ behaviors should be tallied in an integer
    ↪ variable and returned upon completion. */
int scancel(MPI_Comm comm)
{
    ...
        if (rank == source) {
        for (i = 0; i < count; i++) {
            send_buf[i] = i;
        }

        MPI_Isend(send_buf, count, MPI_INT, dest,
    ↪ tag, comm, &req);
        MPI_Cancel(&req);

        MPI_Test(&req, &flag, &status);

        if (flag == 0) {
            errors++;
        }
        MPI_Send(send_buf, count, MPI_INT, dest, tag
    ↪ , comm);

    } else if (rank == dest) {
        MPI_Recv(recv_buf, count, MPI_INT, source,
    ↪ tag, comm, &status);
    }
    ...
```

Fig. 1. Example of prompt and a few lines of code for scancel, which cancels a nonblocking send before it is received.

```
/* PROMPT: int iterative_isend_irecv(MPI_Comm comm)
    ↪ tests non-blocking point-to-point
    ↪ communication using MPI_Isend and MPI_Irecv
    ↪ in repeated iterations. Unexpected behaviors
    ↪ should be tallied in an integer variable and
    ↪ returned upon completion. */
int iterative_isend_irecv(MPI_Comm comm) {

   ...
       if (rank == source) {
        for (n = 0; n < 100; n++) {
            for (i = 0; i < count; i++)
                { send_buf[i] = i + n; }
            MPI_Isend(send_buf, count, MPI_INT, dest
    ↪ , tag, comm, &req);
            MPI_Wait(&req, &status);
        }
    } else if (rank == dest) {
        for (n = 0; n < 100; n++) {
            MPI_Irecv(recv_buf, count, MPI_INT,
    ↪ source, tag, comm, &req);
            MPI_Wait(&req, &status);
            for (i = 0; i < count; i++) {
                if (recv_buf[i] != i + n) {
                    errors++;
                }
            }
        }
    }
   ...
```

Fig. 2. Example of prompt and a few lines of code for iterative_isend_irecv, demonstrating iterative nonblocking message passing.

passed CUnit tests within 1.2 test runs on average, though manual debugging was still needed in 5 out of 20 function runs, particularly for complex MPI operations.

CUnit played a crucial role in validating the test suites by executing unit tests that assessed function return values against expected outcomes. As a unit testing framework, it provided a structured and consistent method for verifying function behavior, ensuring that errors were systematically detected when return values deviated from expected results [12]. This standardized approach helped regulate the testing process, allowing for repeatable and automated validation of MPI functions. However, while CUnit effectively confirmed correctness within its test cases, it did not diagnose the cause of failures. In cases where errors occurred—particularly in complex MPI operations—manual debugging was still required to determine the underlying issues. By integrating automated verification with structured testing, CUnit ensured that Copilot-generated functions were assessed in a controlled and reproducible manner. These findings emphasize that while AI tools can enhance productivity, they are not a substitute for deep domain knowledge [9].

Prompt engineering helped reinforce MPI concepts when there was baseline familiarity, allowing focus on functionality rather than syntax or minor details. This reduced tedium and made complex operations more approachable. However, if any errors occurred for unfamiliar MPI calls, refining the prompt became more difficult due to a lack of familiarity with the operation being performed. Without a clear understanding of the operation's nuances, it was challenging to provide

```
/* PROMPT: int request_status_null(MPI_Comm comm)
    ↪ tests the status of MPI_REQUEST_NULL in
    ↪ MPI_Request_get_status and verifies that the
    ↪ status returned is accurate. Unexpected
    ↪ behaviors should be tallied in an integer
    ↪ variable and returned upon completion. Do not
    ↪  suggest additional functions. */
int request_status_null(MPI_Comm comm)
{
    ...
    MPI_Request_get_status(req, &flag, &status);
    if (flag == 0) {
        errors++;
    }
    // if status not empty, increment unexpected
    ↪ behaviors
    if (status.MPI_SOURCE != MPI_ANY_SOURCE ||
    ↪ status.MPI_TAG != MPI_ANY_TAG || status.
    ↪ MPI_ERROR != MPI_SUCCESS) {
        errors++;
    }
    return errors;
}
```

Fig. 3. Example of AI-generated function for request_status_null, testing the behavior of MPI_Request_get_status with MPI_REQUEST_NULL.

the level of detail needed to steer Copilot toward a correct solution. These findings suggest that AI-assisted coding is most effective when students already have a foundational understanding of the concepts.

Copilot's performance varied significantly depending on the complexity of the MPI tasks. While it successfully generated test cases for standard MPI operations [8], it struggled with more advanced scenarios, such as non-blocking communi-

cation and wildcard use. Debugging Copilot-generated code for these cases was particularly time-consuming, as errors often stemmed from subtle implementation details that were not immediately apparent. This reinforced the importance of a strong conceptual understanding when working with AI-assisted coding tools. Additionally, it underscored the need for educators to equip students with the skills to critically evaluate AI-generated solutions rather than rely on them at face value.

Based on these findings, several key recommendations emerge for integrating AI-assisted coding into PDC education:

Ensure a Strong Programming Foundation Before Engaging with AI Tools – AI-assisted coding tools are most effective when students have a solid grasp of fundamental programming concepts. Before using tools like GitHub Copilot, students should be proficient in syntax, debugging, and algorithmic problem-solving to assess AI-generated suggestions.

Teach Prompt Engineering as a Core Skill – Students should be trained to iteratively refine prompts when working with AI-assisted coding tools [6]. This includes specifying constraints, testing for completeness, and adjusting wording to guide the AI toward correct implementations. Educators can incorporate hands-on exercises where students practice crafting, refining, and debugging AI-generated code.

Emphasize AI as a Collaborative Tool, Not a Replacement – While AI tools can enhance productivity, they should be framed as collaborative assistants rather than substitutes for deep learning and that code should be validated using established debugging practices.

Incorporate Testing and Validation in AI-Assisted Coding Assignments – AI-generated solutions should always be validated using structured testing frameworks, such as CUnit for MPI-based applications. Educators should require students to integrate unit tests and verify correctness before accepting AI-generated code [11], [12].

Address the Limitations of AI in Parallel Computing – Given that AI models may struggle with advanced MPI concepts such as deadlocks, synchronization, and scalability, courses should include modules that focus on debugging and refining AI-generated solutions to meet these challenges [13].

By implementing these recommendations, educators can better equip students with the skills needed to effectively leverage AI tools while maintaining a strong foundation in PDC principles. The goal is to balance AI-assisted efficiency with critical thinking, ensuring that students develop a deep understanding of parallel programming concepts alongside their ability to use AI for code generation.

V. Conclusions

This study explored the application of GitHub Copilot in generating MPI test suites and its implications for PDC education. The results demonstrate that while AI-assisted tools can reduce cognitive load and reinforce MPI concepts, their effectiveness is highly dependent on the user's existing knowledge. In this study, the researcher's ability to refine Copilot-generated test cases was directly tied to their familiarity with MPI operations. When working within well-understood concepts, Copilot served as an effective aid, but unfamiliar operations made it difficult to guide AI-generated outputs toward correctness. This finding underscores that AI tools, rather than serving as standalone solutions, function best when integrated into an educational framework that emphasizes critical evaluation and iterative refinement. These findings highlight that AI is best utilized as a collaborative assistant rather than a replacement for domain expertise.

A key takeaway is that AI's utility in PDC education hinges on structured integration into coursework. While Copilot helped automate aspects of test case generation, students needed foundational MPI knowledge to refine AI-generated code and address logical errors. This suggests that AI tools should be positioned as learning aids rather than automated solutions, reinforcing best practices rather than replacing structured instruction.

Future work should explore whether domain-specific AI models trained on parallel computing patterns can mitigate current limitations. However, given the challenges in guiding AI toward correct implementations even with structured prompt engineering, it is likely that human oversight will remain an essential component of AI-assisted testing. A hybrid educational approach, where students iteratively refine AI-generated test cases while receiving targeted instruction on debugging and verification, could provide a more effective balance between automation and conceptual understanding. Additionally, further research is needed to determine how structured methodologies can be developed to help users refine AI-generated outputs more efficiently, reducing the trial-and-error nature of prompt engineering. Expanding this study to a larger student group would also provide more generalizable insights into AI-assisted coding in PDC education.

References

[1] GitHub, Inc., "Github Copilot Documentation," 2024.

[2] D. Nichols, A. Marathe, H. Menon, T. Gamblin, and A. Bhatele, "HPC-Coder: Modeling parallel programs using large language models," *ISC High Performance 2024 Research Paper Proceedings*, 2024.

[3] "Tabnine AI code assistant." https://www.tabnine.com/. Accessed: January 29, 2025.

[4] "Amazon codewhisperer documentation." https://docs.aws.amazon.com/codewhisperer/. Accessed: January 29, 2025.

[5] OpenAI, "Chatgpt," 2025. Accessed: January 29, 2025.

[6] X. Amatriain, "Prompt design and engineering: Introduction and advanced methods," *arXiv preprint arXiv:2401.14423*, 2024.

[7] R. Pandey, P. Singh, R. Wei, and S. Shankar, "Transforming software development: Evaluating the efficiency and challenges of github copilot in real-world projects," 2024.

[8] B. Zhang, P. Liang, X. Zhou, A. Ahmad, and M. Waseem, "Practices and challenges of using github copilot: An empirical study," *ICSEKE 2023*, pp. 124–129, 2023.

[9] N. Schneider, N. Hasabnis, and V. A. Vo, "MPIrigen: MPI code generation through domain-specific language models," *AI4Sys '24*, 2024.

[10] Message Passing Interface Forum, "MPI: A message-passing interface standard version 3.1," 2015. Accessed: January 29, 2025.

[11] S. Bhatia, T. Gandhi, D. Kumar, and P. Jalote, "Unit test generation using generative AI: A comparative performance analysis of autogeneration tools," *LLM4Code '24*, pp. 54–61, 2024.

[12] K. Beck, *Test-Driven Development: By Example*. Addison-Wesley, 2002.

[13] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2018.