

# Developing an Interactive OpenMP Programming Book with Large Language Models

**Xinyao Yi, University of North Carolina at Charlotte, Charlotte, NC, USA**

**Anjia Wang, Intel Corporation, Hillsboro, OR, USA**

**Yonghong Yan, University of North Carolina at Charlotte, Charlotte, NC, USA**

**Chunhua “Leo” Liao, Lawrence Livermore National Laboratory, CA, USA**

# Agenda

---

- **Background**
- **Motivation**
  - **Leveraging LLMs to automatically generate OpenMP learning materials**
- **Method**
  - **Outlining the Book and Each Chapter with the Aid of LLMs**
  - **Content Generation, Including Code Examples using LLMs**
  - **Interactive and Incremental Development**
- **Conclusion**

# Background: The Evolution of OpenMP and Educational Resources

---

## High-Level Programming Models in HPC

- **Support for New Features:** Supercomputer node architectures grow increasingly complex, the necessity for high-level programming models becomes paramount to maintain and enhance productivity.
- **Role of OpenMP:** A cornerstone for parallel computing, widely utilized across various processor types including multi-core, multi-threaded, and many-core accelerator architectures.

## Evolving Complexity of OpenMP

- **Increasing Specification Size:** The length of OpenMP specification has more than doubled, exemplified by its growth from 318 pages in version 3.1 to 649 pages in version 5.2.
- **Impact on Learning Curve:** This expansion has significantly steepened the learning curve.

## Traditional vs. Modern Learning Resources

- **Conventional Approaches:** Historically, OpenMP education has relied on traditional learning resources such as textbooks, tutorials and online resources, that are [manually created](#).
- **Rise of Large Language Models (LLMs):** LLMs (e.g., Gemini Pro 1.5, Claude 3, and ChatGPT-4) offer a new paradigm to [automatically create educational materials](#).

# Motivation

---

**Research Question:** *Can LLMs be used to quickly generate OpenMP learning materials for an Interactive OpenMP Programming book?*

1. Which LLMs to use?
2. What are the effective prompting methods?
3. What are the quality of the generated learning materials?
4. How to feed latest information to LLMs?
5. How many human interventions are needed to review and revise the LLM-generated materials?

# Method: Multiple Techniques to Address Challenges

	Challenges	Solutions
<div>Outline Generation with LLMs</div>	Choices of LLMs	Leverage strengths of <b>multiple LLMs</b> (Gemini Pro 1.5, Claude 3, and ChatGPT-4)
<div>Cross-Model and Manual Review</div>	Context window size limit	Divide-and-conquer using <b>Chain-of-Thought</b> : two-level outline generation, followed by content (text and example) generation
<div>Content Generation with LLMs</div>	Prompting engineering	Applied the <b>CO-STAR</b> framework to craft effective prompts.  <b>One-shot learning</b> : providing previous chapter as example to improve consistency
<div>Cross-Model and Manual Review</div>	Evolving OpenMP specifications	<b>In-Context Learning</b> : Uploaded OpenMP Specification and Official Examples to provide up-to-date, context-rich information
<div>Integrating into Jupyter Notebook</div>	Quality Assurance (Fix Hallucinations)	Using <b>cross-model review</b> : mutual critiques among LLMs  Complemented AI outputs with <b>manual review</b> , edits, and merges.
	Code Execution	Integrating content within <b>Jupyter</b> Notebooks for interactive code execution and immediate feedback.

# Method – 1. Outlining the Book with the Aid of LLMs

---

## Combined Approach: Human Expertise & AI

- Utilizing both human insights and machine-generated content for a comprehensive OpenMP programming book.
- Aim to balance the depth and accessibility of the material.

## Generating Textbook Outlines using In-Context Learning

- Employed multiple LLMs to create varied structures based on OpenMP API 5.2 Specification and official examples (*Uploaded via web interfaces*).
- *Prompts: "Here are the OpenMP 5.2 Specification and the official examples. Please use this information to guide the creation of the textbook outline. Include sections on basic and advanced topics, referencing specific constructs from the specification and relevant examples. Outline a few initial chapters focusing on core concepts, synchronization, and tasking."*

## Refining the Approach

- After reviewing LLM-generated outlines, manually developed a more targeted book structure.
- The book focuses on applying OpenMP directives and clauses through fundamental and advanced examples.

# Method – 1 Outline Generated by ChatGPT

## Part I: Introduction to Parallel Programming

### 1. Chapter 1: Introduction to OpenMP

- o What is OpenMP?
- o Importance and Benefits of Parallel Programming
- o Overview of OpenMP Architecture
- o How to Enable OpenMP in Your Compiler
- o Basic Syntax and Directive Structure

### 2. Chapter 2: Getting Started with OpenMP

- o First Parallel Program: `parallel` Construct
- o Example: Simple Parallel Loop ([Example Reference: Simple Parallel Loop from Section 3.1](openmp-examples-5.2.2-f...))
- o Understanding Thread Creation and Management
- o Environment Variables for Parallel Execution

## Part II: Core OpenMP Concepts

### 3. Chapter 3: Worksharing Constructs

- o `for` and `do` Constructs
- o Example: Parallel For Loop ([Example Reference: ploop.1.c from Section 3.1](openmp-examples-5.2.2-f...))
- o Sections Construct: Distributing Workload Among Threads
- o `single` Construct
- o `workshare` Construct (Fortran-specific)
- o Example: Single Construct ([Example Reference: single construct from Section 3.12](openmp-examples-5.2.2-f...))

### 4. Chapter 4: Data Environment

- o Private, Shared, and Reduction Variables
- o Example: Private Clause in Parallel Sections ([Example Reference: firstprivate and sections constructs from Section 3.11](openmp-examples-5.2.2-f...))
- o Threadprivate Variables
- o Default Data Sharing Attributes
- o Example: Data Environment and Private Clause ([Example Reference: private clause in Section 10.3](openmp-examples-5.2.2-f...))

### 5. Chapter 5: Synchronization

- o Barriers and Critical Sections
- o Example: Critical Construct ([Example Reference: Critical construct in Section 9.1](openmp-examples-5.2.2-f...))
- o Atomic Operations
- o Example: Atomic Construct ([Example Reference: Atomic Construct from Section 9.4](openmp-examples-5.2.2-f...))
- o Locks: Simple and Nested
- o Example: Lock Routines ([Example Reference: Lock Routines in Section 9.12](openmp-examples-5.2.2-f...))
- o Ordering Execution with `ordered` Clause
- o Example: Ordered Clause and Construct ([Example Reference: Ordered Clause from Section 9.9](openmp-examples-5.2.2-f...))

## Part III: Advanced Topics

### 6. Chapter 6: Tasking

- o Introduction to Tasking in OpenMP
- o `task` and `taskwait` Constructs
- o Example: Task and Taskwait Constructs ([Example Reference: task and taskwait Constructs from Section 5.1](openmp-examples-5.2.2-f...))
- o Task Dependences
- o Example: Task Dependences ([Example Reference: Task Dependences in Section 5.3](openmp-examples-5.2.2-f...))
- o Taskgroups and Taskyield
- o Example: Taskgroup Construct ([Example Reference: Taskgroup Construct from Section 5.5](openmp-examples-5.2.2-f...))

### 7. Chapter 7: Affinity and Task Mapping

- o Understanding OpenMP Affinity
- o `proc_bind` and Task Affinity
- o Example: `proc_bind` Clause ([Example Reference: `proc_bind` Clause from Section 4.1](openmp-examples-5.2.2-f...))
- o Task Affinity Example

### 8. Chapter 8: SIMD and Loop Transformations

- o SIMD in OpenMP
- o Example: SIMD Directives ([Example Reference: SIMD examples-5.2.2-f...])
- o Loop Unrolling and Tiling
- o Example: Loop Transformations ([Example Reference: t f...])

### 9. Chapter 9: OpenMP on Heterogeneous Systems

- o Introduction to Device Constructs
- o Example: Target Construct ([Example Reference: target examples-5.2.2-f...])
- o Mapping Data to Devices
- o Example: Data Mapping ([Example Reference: map Clause examples-5.2.2-f...])
- o Unified Shared Memory and Offloading
- o Example: USM ([Example Reference: Unified Shared Memory examples-5.2.2-f...])

## Part IV: Expert Techniques

### 10. Chapter 10: Memory Model and Advanced Synchronization

- o OpenMP Memory Model
- o Advanced Synchronization: Locks and Semaphores
- o Example: Memory Model ([Example Reference: OpenMP Memory Model from Section 11.1](openmp-examples-5.2.2-f...))

### 11. Chapter 11: Error Handling and Debugging in OpenMP

- o Error Directives
- o Example: Error Directive ([Example Reference: error Directive from Section 12.14.3](openmp-examples-5.2.2-f...))
- o Debugging Parallel Programs
- o Tools for Analyzing OpenMP Performance

Include all the details from the OpenMP Specification. Some of them are too complicated and we don't want to include them in our book. Our book will concentrate on the exercises and the interactive programming practice.



# Method – 1 Outline Generated by Gemini

## I. Introduction to OpenMP

- **Overview of Parallel Computing and Shared Memory Models**
  - Briefly introduce the concepts of parallel computing, shared memory architectures, and the challenges of programming for such systems.
  - *Reference: OpenMP Specification - Chapter 1: Overview of the OpenMP API*
- **The OpenMP Programming Model**
  - Explain the fork-join model of parallel execution in OpenMP.
  - Introduce the concept of threads, teams, and the role of the master thread.
  - *Reference: OpenMP Specification - Chapter 1: Overview of the OpenMP API, Section 1.3 Execution Model*
- **OpenMP Directives and Clauses**
  - Explain the syntax and placement of OpenMP directives in C/C++ and Fortran.
  - Introduce common clauses like `parallel`, `for`, and `shared`.
  - *Reference: OpenMP Specification - Chapter 3: Directive and Construct Syntax, Chapter 4: Base Language Formats and Restrictions*
  - *Examples: ploop.1.c, ploop.1.f, parallel.1.c, parallel.1.f*

## II. Core OpenMP Concepts

- **Work-Sharing Constructs**
  - Explain how OpenMP distributes work among threads using constructs like `for`, `sections`, and `single`.
  - Discuss loop scheduling and the `schedule` clause.
  - *Reference: OpenMP Specification - Chapter 11: Work-Distribution Constructs*
  - *Examples: ploop.1.c, ploop.1.f, parallel.1.c, parallel.1.f, psections.1.c, psections.1.f, single.1.c, single.1.f*
- **Data Environment and Sharing**
  - Explain the concepts of shared and private variables in OpenMP.
  - Introduce data-sharing attribute clauses like `shared`, `private`, and `firstprivate`.
  - *Reference: OpenMP Specification - Chapter 5: Data Environment*
  - *Examples: private.1.c, private.1.f, fpriv\_sections.1.c, fpriv\_sections.1.f90*
- **Reductions**
  - Explain how OpenMP handles reduction operations using the `reduction` clause.
  - Provide examples of common reduction operations like sum, product, and min/max.
  - *Reference: OpenMP Specification - Section 5.5: Reduction Clauses and Directives*
  - *Examples: reduction.1.c, reduction.1.f90*

## III. Synchronization in OpenMP

- **Barriers and Implicit Synchronization**
  - Explain the concept of barriers and how they ensure synchronization between threads.
  - Discuss implicit barriers in constructs like `for` and `sections`.
  - *Reference: OpenMP Specification - Section 15.3: Barriers*
  - *Examples: nowait.1.c, nowait.1.f, nowait.2.c, nowait.2.f90*
- **Explicit Synchronization Constructs**
  - Introduce the `critical`, `atomic`, and `ordered` constructs for fine-grained synchronization.
  - Explain the use cases for each construct and their impact on performance
  - *Reference: OpenMP Specification - Chapter 15: Synchronization Constructs and Clauses*
  - *Examples: critical.1.c, critical.1.f, atomic.1.c, atomic.1.f*
- **Locks and Mutexes**
  - Explain the concept of locks and mutexes.
  - Introduce OpenMP lock routines like `omp_unset_lock`.
  - *Reference: OpenMP Specification - Chapter 16: Locks and Mutexes*
  - *Examples: simple\_lock.1.c, simple\_lock.1.f*

## IV. Tasking in OpenMP

- **Task Creation and Management**
  - Explain how to create explicit tasks using the `taskwait` construct.
  - Discuss task scheduling, dependencies, and the use of the `taskwait` construct.
  - *Reference: OpenMP Specification - Chapter 12: Tasking Constructs*
  - *Examples: tasking.1.c, tasking.1.f90, tasking.2.c, tasking.2.f90*
- **Task Dependencies**
  - Explain how to express dependencies between tasks using the `depend` clause.
  - Discuss different types of dependencies (in, out, inout).
  - *Reference: OpenMP Specification - Section 15.9: OpenMP Dependencies*
  - *Examples: task\_dep.1.c, task\_dep.1.f90, task\_dep.2.c, task\_dep.2.f90, task\_dep.3.c, task\_dep.3.f90*
- **Task Synchronization**
  - Explain the `taskgroup` construct for synchronizing groups of tasks
  - *Reference: OpenMP Specification - Section 15.4: taskgroup Construct*
  - *Examples: taskgroup.1.c, taskgroup.1.f90*

Too much details for the basic usage of OpenMP and synchronization (Because we mentioned it in the prompts.)  
Nothing related to offloading.

Note: Claude cannot take the specification as the input since the uploaded pdf files are too big.



# Method – 1 Manually Revised Outline

---

1. Overview of OpenMP Programming
  - 1.1. Introduction of OpenMP
  - 1.2. Creating a Parallel Program with OpenMP
  - 1.3. Performance Analysis
2. Parallel Programming for Multicore and Multi-CPU Machines
  - 2.1. Multicore and Multi-CPU shared memory systems
  - 2.2. Creating SPMD parallelism using OpenMP parallel directive
  - 2.3. Creating SPMD parallelism using OpenMP teams directive
  - 2.4. Synchronization of Threads Using Barrier and Ordered Directive
  - 2.5. Asynchronous Tasking
  - 2.6. Explicit Distribution of Work Using Single, Sections, Worksharing-Loop, and Distribute Construct
3. Parallel Programming for SIMD and Vector Architecture
  - 3.1. Introduction to SIMD and Vectorization
  - 3.2. OpenMP SIMD Constructs and Clauses
  - 3.3. Utilizing SIMD Directives for Loop Vectorization
  - 3.4. Function Vectorization with declare simd
  - 3.5. Data Alignment and Linear Clauses
  - 3.6. SIMD Reductions and Scans
  - 3.7. Best Practices and Performance Considerations
  - 3.8. Real-World Examples and Case Studies

4. Parallel Programming for GPU Accelerators
  - 4.1. Introduction
  - 4.2. OpenMP Device Constructs
  - 4.3. Mapping Data to GPU Devices
  - 4.4. Asynchronous Execution and Dependencies
  - 4.5. Device Memory Management
  - 4.6. Parallel Execution on GPU Devices
  - 4.7. Tuning Performance for GPU Offloading
  - 4.8. Advanced Topics and Best Practices
  - 4.9. Conclusion

Concise, yet covering all commonly used OpenMP directives and clauses, along with core concepts of worksharing and synchronization. This makes it more intuitive and easier to learn for beginners. Additionally, the outline is structured more like a textbook.

# Method – 2 Outlining Each Chapter with the Aid of LLMs

Employed the CO-STAR framework\* for prompt design, enhancing the relevance and effectiveness of LLM outputs.

Guidelines	Explanation	Our Prompts
Context(C)	Providing background information	I am currently <b>writing a book</b> on OpenMP parallel programming aimed at teaching others. <b>I have completed a chapter on teams</b> , which follows a specific outline style. I am now <b>focusing on the synchronization</b> aspect of OpenMP, specifically on barrier and order constructs.
Objective (O)	Clearly defining the task	<b>Generate an outline</b> for the chapter on synchronization, focusing on barrier and order constructs. The outline should be based on the style used in the previous chapter on teams.
Style (S)	Specifying the writing style	Educational, structured
Tone (T)	Setting the tone	Instructional, clear
Audience (A)	Identifying the intended audience	Readers are students and programmers new to parallel programming, as well as educators looking for teaching resources.
Response (R)	Providing the response format	Provide <b>a structured outline</b> in text format that details sections and subsections for the chapter on synchronization, including key points to cover for barrier and order constructs.

*\*Developed by GovTech Singapore’s Data Science and Artificial Intelligence Division, through GPT-4 Prompt Engineering Competition*

# Method – 2 Results from Gemini Pro 1.5 and GPT-4

## Outline generated by Gemini Pro 1.5

- **Introduction**
- **Barrier Directive**
  - Establishing Synchronization Points
  - Example
- **The Ordered Directive: Maintaining Sequential Steps**
  - Enforcing Order in the Parallel Ballet
  - `doacross` Clause: Specifying Dependencies
- **Implicit Barriers: Automatic Synchronization**
  - Removing Implicit Barriers with `nowait`
- **Best Practices for Using Barrier and Ordered Directives**

## Outline generated by ChatGPT-4

- **Introduction**
- **Barrier Directive**
  - Purpose of the Barrier Directive
  - Usage
  - Example: Using the Barrier Directive
  - Considerations
- **Ordered Directive**
  - Purpose of the Ordered Directive
  - Usage
  - Compatibility with the `doacross` Clause
  - Example: Serial Output in Parallel Loop
  - Considerations
- **Summary**
  - Key Takeaways

Example output for Section 2.4. Synchronization of Threads Using Barrier and Ordered Directives

# Method – 2 Results from Claude 3

## Outline generated by Claude 3

- |   |  |
|---|--|
| <ul style="list-style-type: none"> <li>● <b>Introduction</b> <ul style="list-style-type: none"> <li>○ Importance of Thread Synchronization</li> <li>○ Overview of the Barrier and Ordered Directives</li> </ul> </li> <li>● <b>Barrier Directive</b> <ul style="list-style-type: none"> <li>○ Purpose and Usage</li> <li>○ Syntax and Examples</li> <li>○ Barrier Regions</li> <li>○ Synchronization Points</li> </ul> </li> <li>● <b>Ordered Directive</b> <ul style="list-style-type: none"> <li>○ Purpose and Usage</li> <li>○ Syntax and Examples</li> <li>○ Enforcing Execution Order</li> <li>○ Ordered Regions</li> <li>○ Stand-alone Ordered Construct</li> <li>○ Block-associated Ordered Construct</li> <li>○ Interaction with Loop Constructs and Clauses</li> <li>○ Best Practices</li> </ul> </li> </ul> | <ul style="list-style-type: none"> <li>● <b>Combining Barrier and Ordered Directives</b> <ul style="list-style-type: none"> <li>○ Use Cases for Combining Directives</li> <li>○ Examples and Code Snippets</li> <li>○ Considerations and Potential Issues</li> </ul> </li> <li>● <b>Implicit Barriers</b> <ul style="list-style-type: none"> <li>○ Implicit Barrier Regions</li> <li>○ Execution Model Events and Tool Callbacks</li> </ul> </li> <li>● <b>Advanced Topics</b> <ul style="list-style-type: none"> <li>○ Nested Barrier and Ordered Directives</li> <li>○ Interoperability with Other Synchronization Mechanisms</li> <li>○ Synchronization in the Context of Tasking</li> <li>○ Debugging and Profiling Synchronization Issues</li> </ul> </li> <li>● <b>Performance Considerations</b> <ul style="list-style-type: none"> <li>○ Overhead and Scalability</li> <li>○ Load Balancing and Synchronization Granularity</li> <li>○ Performance Tuning and Optimization</li> </ul> </li> <li>● <b>Summary and Conclusion</b></li> </ul> |
|---|--|

Note: The spec/example PDFs are split into chapters: relevant chapters are fed to Claude 3 to work around its context window limit.

# Method – 2 Final Revised and Merged Chapter Outline

## Revised Outline

- |   |  |
|---|--|
| <ul style="list-style-type: none"> <li>● <b>Introduction</b> <ul style="list-style-type: none"> <li>○ Importance of Thread Synchronization</li> <li>○ Overview of the Barrier and Ordered Directives</li> </ul> </li> <li>● <b>Barrier Directive</b> <ul style="list-style-type: none"> <li>○ Purpose and Usage</li> <li>○ Syntax and Practical Examples</li> <li>○ Barrier Regions and Synchronization Points</li> </ul> </li> <li>● <b>Ordered Directive</b> <ul style="list-style-type: none"> <li>○ Purpose and Usage</li> <li>○ Syntax and Practical Examples</li> <li>○ Interaction with Loop Constructs and <code>doacross</code> Clause</li> <li>○ Best Practices and Considerations</li> </ul> </li> <li>● <b>Combining Barrier and Ordered Directives</b> <ul style="list-style-type: none"> <li>○ Use Cases for Combining Directives</li> <li>○ Examples and Code Snippets</li> <li>○ Considerations and Potential Issues</li> </ul> </li> </ul> | <ul style="list-style-type: none"> <li>● <b>Implicit Barriers</b> <ul style="list-style-type: none"> <li>○ Overview and Automatic Synchronization</li> <li>○ Controlling Implicit Barriers</li> </ul> </li> <li>● <b>Advanced Topics</b> <ul style="list-style-type: none"> <li>○ Nested Directives and Interoperability</li> <li>○ Synchronization in the Context of Tasking</li> <li>○ Debugging and Profiling Synchronization Issues</li> </ul> </li> <li>● <b>Performance Considerations</b> <ul style="list-style-type: none"> <li>○ Overhead and Scalability</li> <li>○ Load Balancing and Synchronization Granularity</li> </ul> </li> <li>● <b>Summary and Conclusion</b> <ul style="list-style-type: none"> <li>○ Recap of Key Points</li> <li>○ Further Learning and Applications</li> </ul> </li> </ul> |
|---|--|

The Revised Outline Used in the Textbook for Section 2.4. Synchronization of Threads Using Barrier and Ordered Directives

## Method – 3. Asking LLMs to Generate Comprehensive Description

**Goal: Generating specific materials for understanding and Explaining Fundamental OpenMP Constructs**

CO-STAR Guidelines	Our Prompts
Context(C)	The <b>task</b> directive is an essential component of OpenMP used to define independent units of work that can be executed in parallel. It's particularly useful for handling irregular workloads in parallel computing.
Objective (O)	Provide a <b>comprehensive</b> explanation of the task directive, including its syntax, clauses, examples, and best practices for using it in OpenMP programs.
Style (S)	Educational, detailed, and structured
Tone (T)	Informative, encouraging, and supportive
Audience (A)	Students, programmers, and practitioners who are learning OpenMP and wish to understand how to effectively utilize the task directive for parallel programming.
Response (R)	Generate the content in a clear, step-by-step manner suitable for textbook material. Include an explanation of the task directive, examples with code snippets, and guidance on when and how to use it effectively.

## The **task** Directive in OpenMP

The **task** directive in OpenMP is used to specify a block of code that should be executed asynchronously in parallel. This is particularly useful for parallelizing irregular workloads, where the amount of work might vary dynamically. By using the **task** directive, you can express independent units of work that can be executed concurrently, allowing for better utilization of parallel computing resources.

### Basic Syntax:

```
#pragma omp task [clauses]
{
    // Code block to be executed as a task
}
```

The **task** directive can be placed before a structured block of code, which will then be treated as a separate unit of work (task) that can be executed by any available thread in the team.

### Key Points:

- Each time a **task** directive is encountered, a new task is generated.
- Tasks are executed by threads in the parallel region, but not necessarily by the thread that creates them.
- Task execution is deferred until a thread becomes available.

### Common Clauses Used with **task**:

- **if(condition)**: Determines whether the code block should be executed as a task or not, based on the condition.
- **untied**: Allows the task to be suspended and resumed by any thread, not just the one that started it.
- **default(shared | none)**: Specifies the data-sharing attributes for variables in the task.
- **shared(variable-list)**: Specifies variables that will be shared among tasks.
- **private(variable-list)**: Specifies variables that will be private to each task.
- **firstprivate(variable-list)**: Initializes variables to have the same values as they had in the task-generating context.

### Example:

```
#include <stdio.h>
#include <omp.h>

int main() {
    #pragma omp parallel
    {
        #pragma omp single
        {
            for (int i = 0; i < 4; i++) {
                #pragma omp task
                {
                    printf("Task %d executed by thread %d\n", i,
                        omp_get_thread_num());
                }
            }
        }
    }
    return 0;
}
```

In this example, the **task** directive is used inside a **single** region to generate tasks. The **single** directive ensures that only one thread creates tasks, but any thread in the parallel region can execute them.

### When to Use **task** Directive:

- When working with irregular workloads, such as recursive algorithms, dynamic data structures (e.g., linked lists, trees), or workloads with unpredictable execution times.
- When you want to achieve finer control over the parallel execution of independent tasks.

### Task Synchronization:

- **taskwait**: Used to ensure that all child tasks generated up to this point are completed before the program continues.
- **taskgroup**: Allows synchronization of a group of tasks, ensuring all tasks within the **taskgroup** block are finished before moving forward.

Using the **task** directive effectively can lead to significant performance improvements, especially in scenarios with dynamic or irregular parallelism. However, it is important to manage task creation and synchronization carefully to avoid overhead and ensure efficient execution.

Generated by ChatGPT



## The OpenMP Task Directive

The OpenMP **task** directive is a powerful tool for expressing parallelism in your programs. It allows you to encapsulate a block of code as a *task* that can be executed concurrently with other tasks.

### Key Features

- **Encapsulates Work:** A task defines a self-contained unit of work within your program. This promotes modularity and makes it easier to reason about parallel execution.
- **Potential for Concurrency:** The OpenMP runtime system is responsible for scheduling and executing tasks, potentially in parallel on multiple threads. This allows you to leverage the available hardware resources efficiently.
- **Implicit and Explicit Tasks:** OpenMP supports both *implicit* and *explicit* tasks.
  - **Explicit tasks:** Created directly using the **task** directive. They give you precise control over when and how tasks are created and executed.
  - **Implicit tasks:** Created automatically by OpenMP when certain constructs, like **parallel loops with nested tasks**, are encountered. They simplify parallelization in common scenarios.
- **Task Dependencies:** OpenMP provides mechanisms for expressing dependencies between tasks. This ensures that tasks are executed in the correct order when data dependencies exist.
- **Task Scheduling:** The OpenMP runtime uses sophisticated algorithms to schedule tasks efficiently on the available threads. This helps to minimize overhead and maximize performance.

### Basic Syntax

```
#pragma omp task [clause ...]
{
    // Code to be executed as a task
}
```

### Common Clauses

- **if(scalar\_expression):** Creates a task only if the scalar expression evaluates to true.
- **depend(dependence\_type: list):** Specifies dependencies between tasks. Dependence types include:
  - **in:** Task depends on data being input from the listed variables.
  - **out:** Task produces data that will be output to the listed variables.
  - **inout:** Task both inputs and outputs data to the listed variables.

### Example Usage

```
#pragma omp parallel
{
    #pragma omp single
    {
        #pragma omp task
        {
            // Task 1 code
        }
        #pragma omp task depend(out: data)
        {
            // Task 2 code (produces 'data')
        }
        #pragma omp task depend(in: data)
        {
            // Task 3 code (consumes 'data')
        }
    }
}
```

In this example:

- Task 1 can execute immediately.
- **Task 2 must wait for Task 1 to complete before it can start.**
- Task 3 must wait for Task 2 to complete before it can start.

### Important Considerations

- **Overhead:** Task creation and management have some overhead. For very small tasks, the overhead may outweigh the benefits of parallelization.
- **Load Balancing:** It's essential to design your tasks to have a reasonable amount of work so that the workload is balanced across threads.

### When to Use Tasks

The **task** directive is a good choice when:

- You have irregular or nested parallelism that's difficult to express with simple parallel loops.
- You want to encapsulate work units and express dependencies between them explicitly.
- You want to leverage the flexibility of the OpenMP runtime's task scheduling capabilities.

Remember, mastering the **task** directive requires practice and careful consideration of your program's structure and data dependencies. Used effectively, it can unlock significant performance gains on modern multi-core and many-core systems.

Let me know if you would like more detailed examples or want to explore specific use cases of the OpenMP **task** directive!

Generated by Gemini

# Method – 3. Generating Code Examples

## Generation and Explanation of Examples

CO-STAR Guidelines	Our Prompts
Context(C)	I am currently writing a book on OpenMP parallel programming aimed at teaching others. In this chapter, we will focus on [directive name].
Objective (O)	<b>Create an example</b> that demonstrates the use of a [directive name] directive, particularly illustrating the functionality of a named clause and how it can help manage parallelism.
Style (S)	Educational, detailed
Tone (T)	Instructional, clear
Audience (A)	Programmers and computer science students who are learning about thread synchronization and parallel processing.
Response (R)	Provide a code snippet in C or C++ that uses the specified directive and clause.

# Method – 3. Generating Code Examples for omp simd

## Code Examples generated by GPT-4, Gemini pro 1.5 and Claude 3

```
#include <stdio.h>
#include <omp.h>

#define N 1000

int main() {
    float a[N], b[N], result[N];
    float factor = 0.5;

    // Initialize arrays
    for (int i = 0; i < N; i++) {
        a[i] = i * 1.0f;
        b[i] = (N - i) * 1.0f;
    }

    // Apply SIMD directive with linear clause
    #pragma omp simd linear(factor:0.1)
    for (int i = 0; i < N; i++) {
        result[i] = a[i] + b[i] * factor;
    }

    return 0;
}
```

**GPT-4**

```
#include <stdio.h>

#define N 1024

int main() {
    float a[N], b[N], c[N];

    // Initialize arrays
    for (int i = 0; i < N; i++) {
        a[i] = i * 2.0f;
        b[i] = i * 3.0f;
    }

    // SIMD computation with preferred simdlen of 4
    #pragma omp simd simdlen(4)
    for (int i = 0; i < N; i++) {
        c[i] = a[i] + b[i];
    }

    return 0;
}
```

**Gemini pro 1.5**

```
#include <stdio.h>

int main() {
    float a[N], b[N], c[N], sum = 0.0;
    int i;
    // Initialize the arrays
    // Do the sum
    #pragma omp simd
    for (i = 0; i < N; i++) {
        c[i] = a[i] + b[i];
    }
    #pragma omp parallel for reduction(+:sum)
    for (i = 0; i < N; i++) {
        sum += c[i];
    }
    return 0;
}
```

**Claude 3**

# Method – 3. Comparing Official Examples with LLM-Generated Ones

## Official OpenMP Examples Versus Code Examples generated by GPT-4

```
double work(double *a, double *b, int n) {
    int i;
    double tmp, sum;
    sum = 0.0;
    #pragma omp simd private(tmp) reduction(+:sum)
    for (i = 0; i < n; i++) {
        tmp = a[i] + b[i];
        sum += tmp;
    }
    return sum;
}
```

Code Example of SIMD from Official OpenMP Examples (Example SIMD.2.c)

```
int main() {
    float a[N];
    float sum = 0.0f;
    for (int i = 0; i < N; i++) a[i] = i * 1.0f;

    // Vectorize the loop
    #pragma omp simd reduction(+:sum)
    for (int i = 0; i < N; i++)
        sum += a[i];
    printf("Sum: %f\n", sum);
    return 0;
}
```

Code Example of SIMD Generated by ChatGPT

Impression: ChatGPT and Gemini generate examples that are different from official ones. Claude tends to directly copy official examples sometimes.
















# Assessing In-Context Learning of LLMs

---

## Learning Details from the Uploaded PDF Files

- The ***doacross*** clause, introduced in the OpenMP 5.2 specification, is used with the *ordered* directive.
- Significant changes in the ***ordered*** directive in OpenMP 5.2:
  - The removal of the use of *simd*, *threads*, and *depend* clauses.
  - Making *doacross* the sole clause used with ordered directive.
- Our evaluation
  - Within sessions of ongoing conversations with different LLMs
  - Ask them to generate descriptions of the ordered directive and its clauses
  - Check whether the LLMs could effectively learn from input files and not merely rely on pre-existing knowledge.
  - Also ask followup questions as needed

## Assessing In-Context Learning of LLMs (cont'd)


Aspect of Evaluation	Initial Understanding	Comprehensiveness	Practical Examples	Handling of Changes	Depth of Understanding
ChatGPT	 Incorrect (used <code>depend</code> )	 Detailed after guidance	 Provided examples	 Understood after guidance	 Deep with guidance
Gemini	 Correct (doacross clause)	 Limited explanation	 Theoretical only	 Noted but shallow	 Moderate, limited depth
Claude	 Failed (did not recognize)	 Missed key details	 No examples	 Missed key changes	 Lacked understanding







# Method – 4. Interactive and Incremental Development



✖ + K

[Interactive OpenMP Programming](#)

[Preface](#)

1. Overview of OpenMP Programming

2. Parallel Programming for Multicore and Multi-CPU Machines

3. Parallel Programming for SIMD and Vector Architecture

4. Parallel Programming for GPU Accelerators



## Interactive OpenMP Programming

**Authors:** Xinyao Yi, Anjia Wang, Yonghong Yan and Chunhua Liao

**Emails:** [xyi2@uncc.edu](mailto:xyi2@uncc.edu), [anjia.wang@intel.com](mailto:anjia.wang@intel.com), [yyan7@uncc.edu](mailto:yyan7@uncc.edu) and [liao6@llnl.gov](mailto:liao6@llnl.gov)

**Institutions:** University of North Carolina at Charlotte, Lawrence Livermore National Laboratory

**Publication Date:** 2024

### About the Book

**Interactive OpenMP Programming** pioneers the integration of Large Language Models (LLMs) such as Gemini Pro 1.5, Claude 3, and ChatGPT-4 into the educational fabric of high-performance computing. This textbook revolutionizes learning by combining cutting-edge AI with traditional educational methodologies to offer a dynamic, interactive experience through Jupyter Books. It enables code execution directly from the web browser, enhancing the learning process with real-time feedback.

[1]:

```
//%compiler: clang
//%cflags: -fopenmp

#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main(int argc, char *argv[]){
    #pragma omp parallel
    printf("%s\n", "Hello World");

    return 0;
}
```

Hello World  
Hello World  
Hello World  
Hello World  
Hello World  
Hello World  
Hello World  
Hello World  
Hello World



Interactive  
OpenMP  
Programming

# Conclusion

---

- **Educational Content Generation:** LLMs, including Gemini Pro 1.5, Claude 3, and ChatGPT-4, have significantly enhanced the efficiency of educational content generation.
  - Demonstrates that LLMs can be instrumental in modernizing educational practices, particularly in complex technical domains like OpenMP programming.
  - Various techniques are available to address limitations of LLMs
  - Manual review is still necessary for subtle details
- **Integration with Traditional Methods:** Effective learning requires that LLMs be strategically integrated with traditional educational methodologies to maintain the necessary depth and accuracy.
- **Real-time Code Execution:** Utilizes Jupyter Notebooks, enabling real-time code execution and feedback, which marks a considerable improvement over traditional, static learning materials.

<https://passlab.github.io/InteractiveOpenMPProgramming/cover.html>

200+ code examples, 17,000+ lines text or code



Interactive  
OpenMP  
Programming

## Future Work

---

- **Extend to Cover More Topics:** Refining the integration of LLMs into educational frameworks to enhance the effectiveness and accuracy of the content.
- **Improved Prompt Design:** Develop better prompt designs to enhance the accuracy of the content generated by LLMs, such as agentic workflows
- **Scalability Across Languages:** Explore the scalability of this approach to other programming languages, broadening the impact of LLM-enabled educational tools.
- **Ongoing Pedagogical Assessments:** Conduct ongoing assessments of the pedagogical impact to ensure these tools meet educational standards and effectively support learners.



PASSLAB at  
UNCC

# Thanks! Any questions?

## Acknowledgement

This material is based upon work supported by the National Science Foundation under Grant No. 2001580 and 2015254.

This work was also prepared by LLNL under Contract DE-AC52-07NA27344 (LLNL-CONF-867264) and supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program.