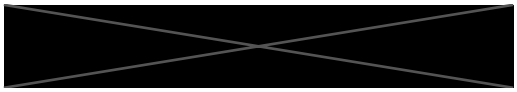
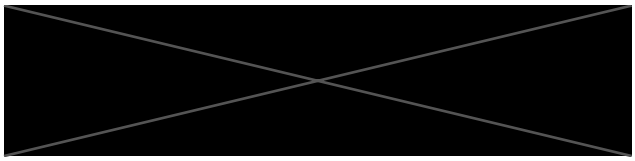


Multithreading with C++ Threads and OpenMP



1 Objective

The objective of this assignment is to gain practical experience with multithreading in C++ using both built-in threading libraries and OpenMP, while also gaining exposure to CMake and Google Test frameworks for building and testing C++ projects. By completing this assignment, you will:

- Implement multithreaded programs using C++ threads, futures, and OpenMP.
- Understand and compare different parallel programming models.
- Use CMake and Google Test frameworks for building and testing C++ projects.
- Analyze and report on the performance of multithreaded applications.

2 Description

In this assignment, you will explore multithreading in C++ by implementing two fundamental parallel algorithms: the inner product and the inclusive scan.

2.1 Inner Product

The **inner product**, also known as the dot product, is a fundamental operation in many scientific computing applications. Given two vectors $\mathbf{a} = [a_1, a_2, \ldots, a_n]$ and $\mathbf{b} = [b_1, b_2, \ldots, b_n]$, their inner product is defined as:

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i$$

Computing the inner product efficiently is crucial in various applications such as machine learning, physics simulations, and numerical methods.

In this part, you will implement a parallel version of the inner product using OpenMP directives to improve performance on multi-core processors.

2.1.1 Provided Implementation

A detailed implementation of the inner product using C++ threads and futures is provided in the file `inner product threads.hpp`. This implementation divides the vectors into chunks and computes partial sums in parallel threads.

2.1.2 Your Task

Your task is to implement the function `parallel_inner_product_openmp` using OpenMP directives. Pay close attention to the TODO items in the following files to guide your implementation:

- `inner_product_openmp.hpp`
- `test_inner_product.cpp`

You may use `#pragma omp taskloop` and `#pragma omp parallel for` to achieve a task-based style similar to using threads and gain finer control compared to using only `#pragma omp parallel for`. Also update the testing blocks in the `test_inner_product.cpp` file in the `tests` subdirectory to test your implementation.

2.2 Inclusive Scan

The **inclusive scan**, also known as the prefix sum, is an operation that computes all the partial sums of a sequence. Given an input sequence $[x_1, x_2, \ldots, x_n]$, the inclusive scan produces an output sequence $[y_1, y_2, \ldots, y_n]$ where:

$$y_i = \sum_{k=1}^{i} x_k \quad \text{for } i = 1, 2, \ldots, n$$

The inclusive scan is an important building block in parallel algorithms and is used in applications such as data analysis, computational geometry, and parallel sorting algorithms.

C++17 introduced the `std::inclusive_scan` function in the Standard Library, which provides a sequential implementation of the inclusive scan. In this assignment, you will implement a parallel version of the inclusive scan using C++ threads and futures.

2.2.1 Provided Implementation

A detailed implementation of the inclusive scan using OpenMP is provided in the file `inclusive_scan_openmp.hpp`. This implementation uses OpenMP tasks to perform the scan operation in parallel.

2.2.2 Your Task

Your task is to implement the function `parallel_inclusive_scan_threads` using C++ threads and futures. Refer to the TODO items in the following files to assist your implementation:

- `inclusive_scan_threads.hpp`
- `test_inclusive_scan.cpp`

Ensure you follow the instructions within the TODO comments to correctly divide the input, manage threads, and combine partial results. Also update the testing blocks in the `test_inclusive_scan.cpp` file in the `tests` subdirectory to verify your implementation.

2.3 Performance Comparison and Reporting

After implementing both functions, you will:

- Run the provided tests using Google Test to verify the correctness of your implementations.
- Measure the performance of your implementations and compare them with the provided ones.
- Analyze the scalability and efficiency of your code with different input sizes and numbers of threads.
- Document your findings in a report.

3 Submission Guidelines

3.1 GitHub Repository

1. **Create a Private Repository:**
   - Sign in to your GitHub account and create a private repository named `AMS562_Homework5`.
2. **Push Your Code:**
   - Include all C++ source (`.cpp`) files, header (`.hpp`) files, CMake files, and test scripts.
   - Organize your project with the following structure:

```
project_root/
|-- src/
    |-- inner_product_threads.hpp
    |-- inner_product_openmp.hpp
    |-- inclusive_scan_threads.hpp
    |-- inclusive_scan_openmp.hpp
|-- tests/
    |-- test_inner_product.cpp
    |-- test_inclusive_scan.cpp
|-- CMakeLists.txt
|-- cmake/
    |- openmp_config.cmake
    |- gtest_config.cmake
|-- README.md
```

   - Note that you can implement everything directly in the header files using templates.
   - Please follow the detailed instructions in the `README.md` for the following:
     - **Project Overview:** A brief description of the project and its objectives.
     - **Build Instructions:** Step-by-step guide on how to build the project using CMake.
     - **Test Instructions:** How to run the tests using Google Test.
3. **Share the Repository:**
   - Add the TA (`yuxuanye1`) as a collaborator to your private repository.
   - Ensure that your last commit is made before the project deadline.

3.2 Code Documentation

- **Commenting:**
  - Use comments to explain your code where necessary.
- **Code Quality:**
  - Follow consistent coding standards and naming conventions.
  - Ensure proper indentation and code formatting for readability.

3.3 Report Submission

1. **Report Document:**
   - Create a 2–3 page report.
   - Format the report with Times New Roman, font size 12, single line spacing, and 1-inch margins.

2. **Report Content:**
    - **Repository URL:** Include the URL of your GitHub repository.
    - **Implementation Details:** Describe your approach to implementing the functions.
    - **Testing and Verification:** Summarize how you tested your code and the results.
    - **Performance Analysis:** Present your performance measurements and analysis.
    - **Challenges and Lessons Learned:** Reflect on any difficulties faced and what you learned.
3. **Submission:** Upload the report as a PDF to Brightspace before the deadline.

3.4 Deadlines

- **Project Deadline:** December 1st, 2025, by 11:59 PM EST.

4 Grading Rubric

- **Code Implementation (40%)**:
    - Correctness, functionality, and efficiency of your code.
- **Performance Analysis (20%)**:
    - Quality of performance measurements, comparisons, and analysis.
- **Testing (20%)**:
    - Use of Google Test, test coverage, and effectiveness.
- **Documentation and Report (10%)**:
    - Clarity, completeness, and quality of the report.
- **Code Quality and Organization (10%)**:
    - Code structure, readability, and use of CMake.