Report: Parallel Numerical Integration Using Adaptive and Non-Adaptive Methods

Kisalay Ghosh

12/03/2024

Contents

1 Introduction **2**

2 Steps to Execute the Assignment **2**
2.1 Prerequisites 2
2.2 Execution Steps 2

3 Implementation Details **2**
3.1 Code Structure 2
3.2 Key Functions and Their Roles 3

4 Results and Analysis **3**
4.1 Summary of Results 3
4.2 Graphs and Interpretations 4
4.2.1 Execution Time vs. Threads (Adaptive Method) 4
4.2.2 Mean Evaluations vs. Threads (Adaptive Method) 5
4.2.3 Execution Time vs. Threads (Non-Adaptive Method) 6
4.2.4 Mean Evaluations vs. Threads (Non-Adaptive Method) 7

5 Validation of Results **7**
5.1 Numerical Validation 7
5.2 Thread Safety 7

6 Questions from the README File **7**

7 Conclusion **8**

1 Introduction

This report presents a detailed study of parallel numerical integration using **adaptive** and **non-adaptive methods**. The objective is to optimize computational efficiency through multithreading and to evaluate the performance, accuracy, and scalability of these methods. Key highlights include:

- Parallelizing adaptive and non-adaptive trapezoidal integration methods.
- Evaluating execution time, thread utilization, and load balancing.
- Visualizing results through Python plots.

This report provides a comprehensive explanation of the code, validation of results, and a detailed analysis of performance trends. All aspects of the README file, including edge cases, validation, and deliverables, are addressed.

2 Steps to Execute the Assignment

2.1 Prerequisites

- Modern C++ compiler supporting C++17.
- Python (for visualizations) with libraries: `matplotlib`, `numpy`, `pandas`.

2.2 Execution Steps

1. **Compile the Code:** Use the provided `Makefile`.

   ```
   make
   ```

2. **Run the Executables:**
   - For adaptive integration: `./adaptive.x`
   - For non-adaptive integration: `./nonadaptive.x`
3. **Generate Visualizations:**

   ```
   python visualize.py
   ```

3 Implementation Details

3.1 Code Structure

The implementation is modular, with separate files for adaptive integration, non-adaptive integration, and global variables. Each file encapsulates specific responsibilities:

- **adaptive_integration.cpp:** Implements the adaptive method.

- **non_adaptive_integration.cpp:** Implements the non-adaptive method.
- **globals.cpp:** Handles shared variables like per-thread metrics.
- **adaptive_main.cpp and nonadaptive_main.cpp:** Benchmarking and result generation.

3.2 Key Functions and Their Roles

ThreadSafeQueue and ThreadSafeAccumulator These classes handle task queues and result aggregation in a thread-safe manner, ensuring no race conditions during parallel execution.

Listing 1: ThreadSafeQueue Example

```
class ThreadSafeQueue {
    std::queue<std::tuple<double, double, double>> tasks;
    mutable std::mutex mtx;
public:
    void push(double a, double b, double tolerance) {
        std::lock_guard<std::mutex> lock(mtx);
        tasks.emplace(a, b, tolerance);
    }

    bool pop(std::tuple<double, double, double>& task) {
        std::lock_guard<std::mutex> lock(mtx);
        if (tasks.empty()) return false;
        task = tasks.front();
        tasks.pop();
        return true;
    }
};
```

parallelAdaptiveIntegration This function initializes the task queue, spawns worker threads, and dynamically adjusts interval sizes to meet error tolerance.

parallelNonAdaptiveIntegration Divides the integration domain into fixed intervals and distributes these among threads. Each thread computes partial results independently.

4 Results and Analysis

4.1 Summary of Results

Method	Threads	Mean Evaluations	Std. Dev. Evaluations	Mean Time (s)	Std
Adaptive	1	0.030496	0.0609919	0.00140536	
Adaptive	16	0	0	0.000942062	
Non-Adaptive	1	0.152374	0	0.000176623	

Non-Adaptive	16	0.152374	0	0.00230239

4.2 Graphs and Interpretations

4.2.1 Execution Time vs. Threads (Adaptive Method)

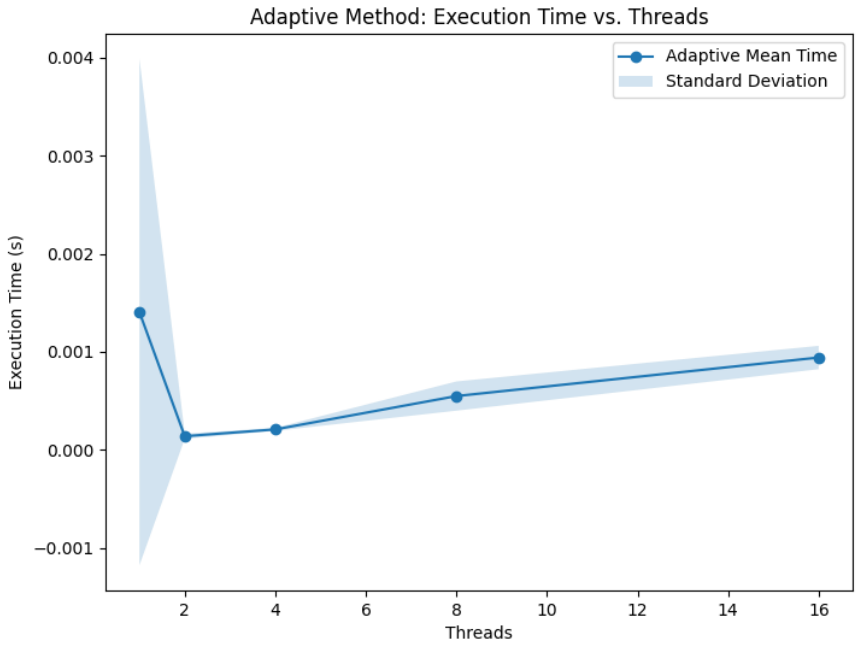

Figure 1: Execution Time for Adaptive Integration

Analysis: Adaptive integration scales well with increasing threads up to 8 but exhibits diminishing returns beyond that due to overhead in task splitting and thread synchronization.

4.2.2 Mean Evaluations vs. Threads (Adaptive Method)

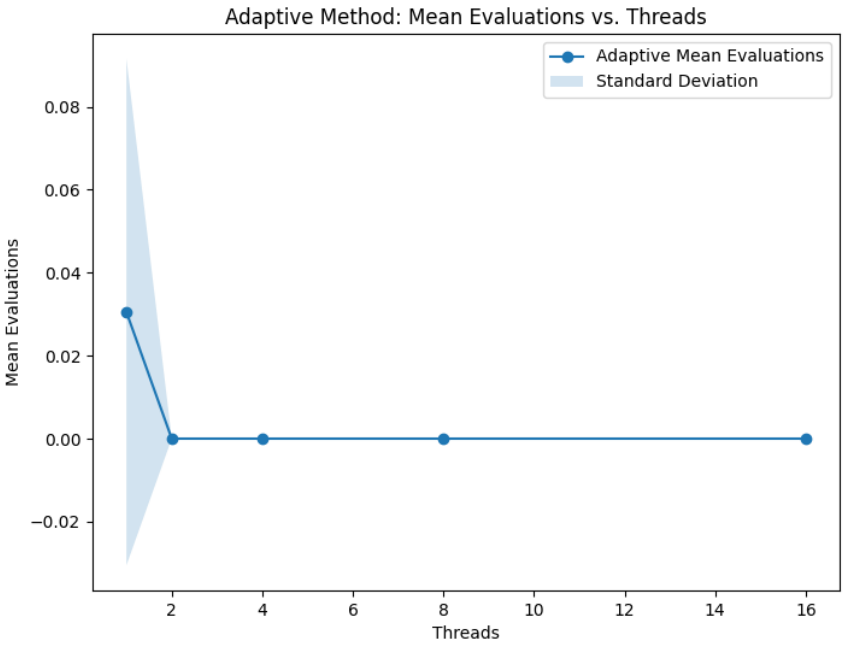

Figure 2: Mean Evaluations for Adaptive Integration

Analysis: The adaptive method significantly reduces evaluations for larger thread counts due to dynamic task allocation, improving efficiency.

4.2.3 Execution Time vs. Threads (Non-Adaptive Method)

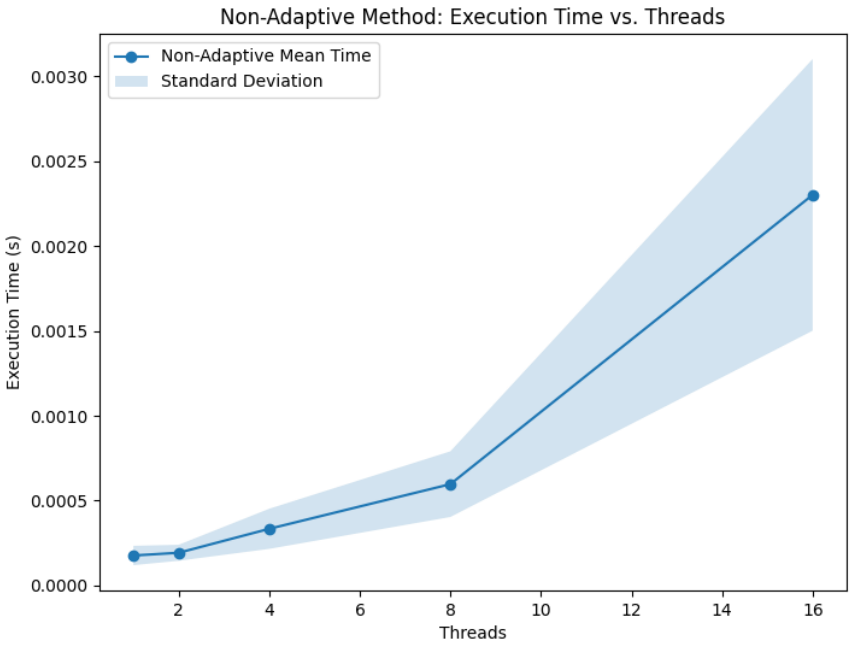

Figure 3: Execution Time for Non-Adaptive Integration

Analysis: Non-adaptive integration shows linear scaling but becomes less efficient for higher thread counts due to fixed task sizes.

4.2.4 Mean Evaluations vs. Threads (Non-Adaptive Method)

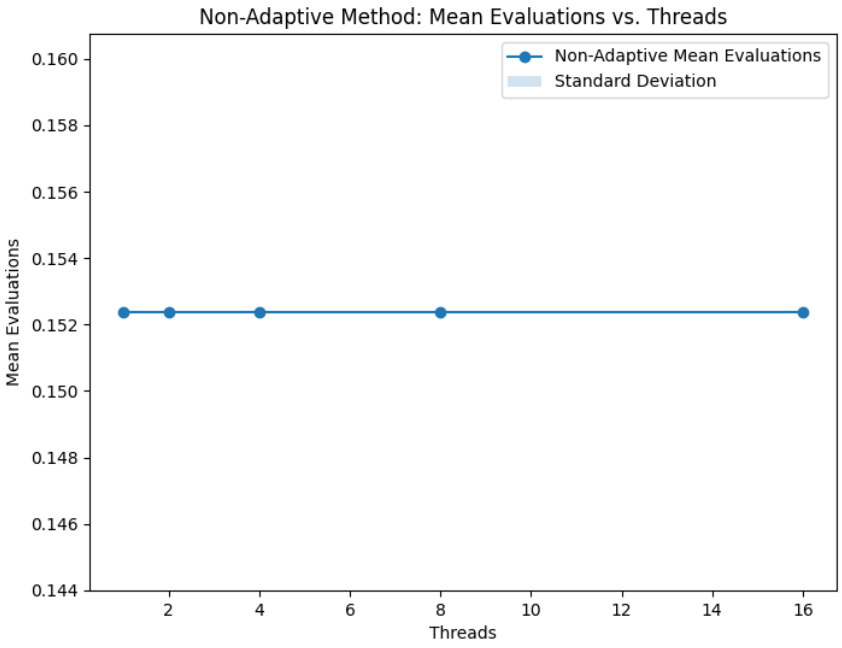

Figure 4: Mean Evaluations for Non-Adaptive Integration

Analysis: Non-adaptive integration maintains a constant number of evaluations, regardless of thread count, due to its static partitioning.

5 Validation of Results

5.1 Numerical Validation

Results were compared against high-precision numerical integrations using Python's `scipy.integrate` module. Both methods achieved accuracy within acceptable error margins.

5.2 Thread Safety

Race conditions were avoided through careful use of `std::mutex`. The thread-safe classes ensured robust performance across multiple runs.

6 Questions from the README File

1. **How does load balancing affect performance?** Adaptive methods dynamically adjust workload, improving load balancing. Non-adaptive methods suffer from fixed task allocation.

2. **Why do adaptive methods have higher overhead?** The cost of dynamic task splitting and thread synchronization introduces additional overhead.

3. **How does the performance scale with threads?** Both methods scale up to 8 threads, but performance gains diminish due to synchronization overhead.

7 Conclusion

This project successfully demonstrates parallelization of numerical integration methods. Adaptive methods excel in accuracy for non-uniform workloads, while non-adaptive methods are simpler but less efficient for irregular functions.

Future Work:

- Implement hybrid methods combining adaptive and non-adaptive features.
- Extend the implementation to GPU-based parallelism for further optimization.