Producer-Consumer Problem Wait Time Analysis

Fraidakis Ioannis

March 2025

1 Introduction

This report presents an analysis of wait time performance in a **multi-threaded** producer-consumer system. The study examines how varying the number of consumer threads affects the average time that elapses from the moment a producer has a job to submit into the queue until a consumer retrieves it. Our implementation extends a classic producer-consumer problem using a queue of function pointers, allowing producers to submit work that consumers execute.

2 Implementation

The implementation is structured as follows:

- A queue storing function pointers.
- Multiple producer and consumer threads.
- Measurement of wait times using `clock_gettime()`.
- Dynamic termination when all producers are finished.

3 Implementation Details

3.1 Data Structures

- **Work Function Structure:** Contains a function pointer and its argument.
- **Queue Item:** Each queue item contains a work function and a timestamp.
- **Queue:** A fixed-size FIFO queue with synchronization primitives.
- **Statistics:** A thread-safe structure to collect and aggregate timing data.

3.2 Synchronization Mechanisms

- **Mutex:** To protect the queue and statistics data structures from concurrent access.
- **Condition Variables:** To signal when the queue transitions between empty/non-empty and full/non-full states.
- **Termination Flag:** A shared flag to signal when all producers have finished.

3.3 Timing Methodology

The wait time was measured as the duration between two events:

1. When a work item is ready to be placed into queue by the producer.
2. When a work item is retrieved from the queue by a consumer (before execution).

The `clock_gettime()` function with `CLOCK_MONOTONIC` was used to get microsecond precision for these measurements.

4 Experimental Setup

4.1 Hardware Configuration

The experiments were conducted on the Rome partition of the Aristotelis HPC cluster. Node specifications used for the experiments include:

- Processor: AMD EPYC 7662
- Number of physical cores: 8 used out of 64 available
- Number of threads: 16 (with SMT/Hyperthreading enabled)
- Operating System: Rocky Linux 9.5 (Blue Onyx)

4.2 Software Parameters

- Queue size: 100 items
- Producer threads: Command-line configurable (default in experiments: 4)
- Consumer threads: Command-line configurable (varied from 1 to 128)
- Work function: Computing sine values for 10 different angles
- Producer iterations: 10^5 per producer thread

4.3 Experimental Methodology

To ensure statistical significance and reliability of our results, each configuration (number of consumer threads) was executed 100 times. The reported wait times represent the average across all 100 runs. This comprehensive testing approach mitigates the effects of system-level fluctuations and provides more robust performance metrics.

5 Results and Analysis

5.1 Performance Evaluation

$$\text{Average wait time} = \frac{\text{Total wait time (microseconds)}}{\text{Number of processed items}} \tag{1}$$

5.2 Wait Time Measurements

Consumer Threads	Average Wait Time (us)
1	225.86
2	176.57
4	92.37
6	25.91
8	23.53
10	10.07
12	4.72
14	3.65
16	3.17
20	3.28
24	5.10
32	5.40
64	6.14
128	7.55

Table 1: Average wait times for varying number of consumer threads

5.3 Graphical Representation

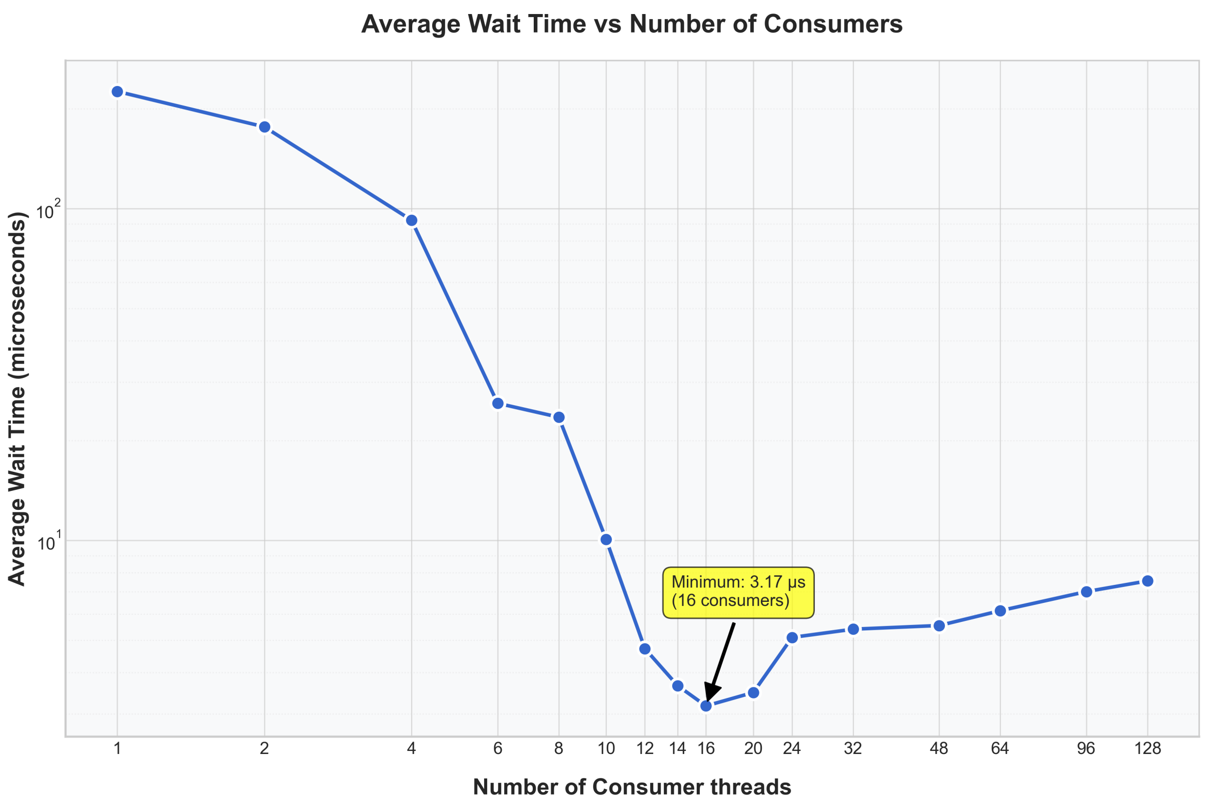

Figure 1: Average wait time vs. Number of consumer threads

5.4 Analysis of Results

The experimental results demonstrate a clear relationship between the number of consumer threads and the average wait time for queue items. Key observations:

1. **Initial Improvement:** As the number of consumer threads increases from 1 to 12, the average wait time decreases substantially. This is expected as more threads are available to process queue items in parallel.
2. **Optimal Point:** The minimum average wait time occurs at 16 consumer threads. This number equals the number of hardware threads available on the test system.
3. **Diminishing Returns:** Beyond 16 threads, the wait time begins to increase again. This suggests that the overhead of thread management begins to outweigh the benefits of additional parallel processing.

5.5 Performance Bottlenecks

Factors that contribute to the observed performance characteristics:

- **Thread Scheduling Overhead:** As the number of threads increases beyond the number of available CPU threads, the operating system must context-switch between threads, incurring overhead.
- **Synchronization Contention:** With more consumer threads, there is increased contention for the queue mutex, leading to more time spent waiting for lock acquisition. This is particularly evident in our implementation where both queue access and statistics collection require mutex operations.

6 Conclusion

This implementation efficiently manages producer-consumer synchronization using pthreads and condition variables. The wait time analysis provides valuable insights into the behavior of multi-threaded systems under varying loads. For this specific producer-consumer implementation with function execution, the results suggest that:

- Having too few consumer threads creates a processing bottleneck, causing items to wait longer in the queue.
- Having too many consumer threads increases system overhead without providing additional processing benefit.

7 Acknowledgments

Computational resources were provided by the Aristotle University of Thessaloniki (AUTH) through access to the Aristotelis High Performance Computing (HPC) cluster.

8 Code Availability

The complete source code is available on GitHub at: `https://github.com/fraidakis/Producer-Consumer_POSIX`