Neel Bhatia

Performance Comparison of Multithreaded Algorithms in C and Rust

Introduction

This project aims to compare the performance of multithreaded implementations in C and Rust, focusing on the efficiency and scalability of threads in both languages. Three different algorithms were used for testing: Mergesort, Matrix Multiplication, and the Sieve of Eratosthenes. The experiment was designed to explore the impact of various factors such as thread count, data size, and cache efficiency on the overall performance of these algorithms in C and Rust. The results are analyzed in terms of execution time and cache hits/misses.

Experimental Setup

Machine Specifications

- **Device:** LAPTOP-BM3H46BB
- **Processor:** 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz (4 cores, 8 threads)
- **Installed RAM:** 20 GB (19.7 GB usable)
- **Operating System:** 64-bit Linux
- **System Type:** x64-based processor

Side Effects

The Intel Core i5-1135G7 has 4 physical cores with 8 threads, but performance from multithreading might not improve linearly. This is because multiple threads can compete for shared resources like cache and memory bandwidth. The Linux scheduler might also not distribute threads evenly, causing some cores to be overworked while others are underutilized. Background processes running on the system also compete for CPU time, adding noise to measurements. Differences in how C and Rust manage memory (stack vs. heap) could also affect how efficiently the algorithms use the CPU cache, potentially causing performance differences between the two languages.

Algorithms Tested

The algorithms tested include Mergesort, Matrix Multiplication, and the Sieve of Eratosthenes, implemented with multithreading in both C using pthreads and Rust using std::thread. Mergesort was parallelized by splitting the array into smaller subarrays, with threads sorting them in parallel. Matrix Multiplication, which involves calculating the dot product of rows and columns, was parallelized by assigning each thread to compute a portion of the result matrix. The Sieve of Eratosthenes, used for finding primes, was parallelized by assigning threads to mark multiples of primes across different number ranges.

Test Variables

The tests varied two key factors: **Thread Count** and **Data Size**. Thread count was adjusted to 2, 4, and 8 threads to evaluate scalability and performance improvements with more threads. Data size was also modified, with array and matrix sizes ranging from 100 to 1,000,000 elements, and the upper limit for the Sieve of Eratosthenes varying from 100 to 10,000,000. This allowed for analysis of how performance changes with increasing workload and the impact of threading on larger or smaller datasets.

Benchmarking Methodology

Each test was executed under the following conditions:

1. **Warm-up Phase:** Before recording any measurements, each program ran a warm-up phase of roughly 5-10 iterations to ensure the system had stabilized. This helped minimize any inconsistencies due to system processes or initialization overhead.
2. **Time Measurement:** Execution time was measured using high-resolution timers (clock_gettime() in C and Instant::now() in Rust). The average of the times provides the mean performance metric, while the standard deviation offers insight into the consistency and reliability of the execution times across iterations, allowing for the identification of any significant outliers or variability.
3. **Cache Monitoring:** Cache hits and misses were measured using Cachegrind, providing insights into memory usage and efficiency. When the CPU needs an instruction, it first checks the I1 cache (Instruction cache). If the instruction is found there (an I1 hit), the CPU can execute it very quickly. If the instruction isn't in the I1 cache (an I1 miss), the CPU has to retrieve it from slower memory, such as the LL (Last-Level) cache or main memory, resulting in a delay. This also applies to data. If the data the CPU needs is in the D1 cache (Data cache) (a D1 hit), access is very fast. If the data isn't there (a D1 miss), the CPU has to fetch it from a slower level of memory. The LL cache acts as a backup for both the I1 and D1 caches. If the CPU finds the needed information in the LL cache (an LL hit), it's faster than going all the way to main memory, but still not as fast as finding it directly in the I1 or D1 caches. If the information isn't in any of the caches (an LL miss), the CPU has no choice but to access main memory, which is the slowest option.
4. **Iteration:** Tests were executed for 30 iterations to ensure the reliability of results, and the average time along with the standard deviation was calculated for each case.

Data

Matrix Multiplication

Language	Cores	Size	Instruction Cache (i1) Miss Rate	Data Cache (D1) Miss Rate	Level 2 Cache (LLd) Miss Rate	Avg EEC Time (seconds)
C	8	100	0.00%	0.00%	0.00%	0.001849
Rust	8	100	0.10%	0.80%	0.00%	0.000711
C	8	500	0.00%	0.20%	0.00%	0.146563
Rust	8	500	0.00%	0.03%	0.10%	0.041884
C	8	1000	0.00%	3.70%	0.00%	1.467962
Rust	8	1000	0.00%	24.40%	0.10%	0.648165
C	8	2000	0.00%	5.10%	0.00%	24.786701
Rust	8	2000	0.00%	28.60%	1.30%	12.849081
C	2	500	0.00%	0.20%	0.00%	2.456916
Rust	2	500	0.00%	2.30%	0.00%	0.064522
C	4	500	0.00%	0.20%	0.00%	1.691113
Rust	4	500	0.00%	2.50%	0.00%	0.042985

MergeSort

Language	Cores	Size	Instruction Cach (i1) Miss Rate	Data Cache (D1 Miss Rate	Level 2 Cache (LLd) Miss Rate	Avg ITER Time (seconds)
C	8	100	0.46%	0.01%	0.90%	0.000199
Rust	8	100	0.44%	1.50%	0.90%	0.002763
C	8	100000	0.00%	0.40%	0.00%	0.134407
Rust	8	100000	0.00%	1.80%	0.10%	0.114717
C	8	1000000	0.00%	0.50%	0.00%	1.508911
Rust	8	1000000	0.00%	6.10%	1.30%	0.559443
C	2	100000	0.00%	0.40%	0.00%	0.150452
Rust	2	100000	0.00%	2.00%	0.00%	0.054058
C	4	100000	0.00%	0.40%	0.00%	0.136279
Rust	4	100000	0.00%	1.80%	0.10%	0.07997

Sieve of Eratosthenes

Language	Cores	Size	Instruction Cach (i1) Miss Rate	Data Cache (D1 Miss Rate	Level 2 Cache (LLd) Miss Rate	Avg ITER Time (seconds)
C	8	100	0.63%	1.50%	1.30%	0.000584
Rust	8	100	0.41%	1.70%	1.10%	0.000666
C	8	100000	0.00%	0.00%	0.00%	0.037053
Rust	8	100000	0.00%	4.90%	0.00%	0.012301
C	8	1000000	0.00%	0.70%	0.00%	0.703995
Rust	8	1000000	0.00%	10.10%	0.00%	0.264296
C	2	100000	0.00%	0.20%	0.00%	0.052946
Rust	2	100000	0.01%	4.90%	0.10%	0.014018
C	4	100000	0.00%	0.00%	0.00%	0.047494
Rust	4	100000	0.00%	4.90%	0.00%	0.027115

Results and Analysis

Execution Time

- **Matrix Multiplication:**
  - C generally performs better in terms of execution time across all data sizes and thread counts. For instance, C executes in 0.001849 seconds for 100 elements (8 threads), while Rust takes 0.000711 seconds. However, as the problem size increases (500, 1000, 2000 elements), the difference in execution time becomes more pronounced, with C taking significantly longer to process larger data sets.
  - Rust maintains a much more stable execution time across larger sizes, with less increase in execution time as data size grows, e.g., for 2000 elements: C takes 24.79 seconds versus Rust's 12.85 seconds.
- **Sieve of Eratosthenes:**
  - C is consistently faster across all sizes, with the smallest dataset (size 100) showing 0.000584 seconds compared to Rust's 0.000666 seconds.
  - Both C and Rust show significant speedups as the problem size increases, but C maintains a more consistent performance gain with larger datasets (1000000 elements, C takes 0.703995 seconds, Rust takes 0.264296 seconds).
- **Merge Sort:**
  - C generally outperforms Rust in execution time, especially for large data sizes. For size 100, C finishes in 0.000199 seconds, while Rust takes 0.002763 seconds.
  - For larger data sets like 1000000, C takes 1.508911 seconds versus Rust's 0.559443 seconds.
- **Conclusion:**
  - C generally exhibits faster execution times in all algorithms compared to Rust, especially for smaller data sets. However, Rust shows better scalability with larger data sizes and tends to scale better with increasing problem complexity.

Cache Efficiency

- **Instruction Cache (i1) Miss Rate:**
    - Both languages show relatively low miss rates for small data sizes (e.g., 0.63% for C and 0.41% for Rust in Csieve.csv with size 100). As the data size increases, the cache miss rates for instruction caches remain low for both languages.
- **Data Cache (D1) Miss Rate:**
    - Rust generally has higher data cache miss rates compared to C across most benchmarks. In Csieve.csv with a dataset of 1000000, Rust shows 10.10% data cache miss rate, while C remains low at 0.70%.
    - In cmerge.csv with size 1000000, Rust has a 6.10% miss rate, compared to C’s 0.50%. Similarly, C’s matrix multiplication benchmark (cmatrix.csv) shows low miss rates (0% for smaller sizes).
- **Level 2 Cache (LLd) Miss Rate:**
    - Both languages show relatively low level 2 cache miss rates for small datasets. For larger datasets, C maintains a slightly better cache miss rate across all algorithms, which may contribute to its faster execution times in some scenarios.
- **Conclusion:**
    - C exhibits superior cache efficiency compared to Rust, particularly in handling data cache and level 2 cache misses. This is likely a factor in C’s faster execution times, especially for memory-intensive tasks like matrix multiplication and sieve algorithms.

Thread Scalability

- **C**:
    - In all benchmarks, increasing the thread count results in a noticeable reduction in execution time, with diminishing returns after 4 or 8 threads, which is typical as the problem becomes more parallelized.
    - For Csieve.csv, the performance gain is evident with the increase in thread count (8 threads with size 1000000 takes 0.703995 seconds).
- **Rust**:
    - Rust also benefits from increasing thread counts, but the gains are often less significant compared to C, particularly with larger data sizes. Rust’s parallelization is more efficient than C’s for larger problem sizes, but it doesn’t always match the execution speed of C in the small data scenarios.
- **Conclusion:**
    - Both languages show good thread scalability, with the performance improving as the number of threads increases. However, C shows more significant performance gains with multithreading in smaller datasets, while Rust handles larger data sizes better with parallelism.

Conclusion

In terms of thread performance, C demonstrates superior execution times, particularly for smaller datasets. It achieves faster results due to its efficient memory management and lower cache miss rates, which are crucial for tasks that require high-speed execution, such as matrix multiplication and the Sieve of Eratosthenes. C also shows more significant improvements with increasing thread counts in smaller datasets, making it a strong choice for applications where thread scalability in smaller workloads is critical.

On the other hand, Rust excels in handling larger data sizes and more complex operations, where its threading model proves to be more efficient. While the performance improvements with multithreading are less dramatic in smaller datasets, Rust manages to better scale as the problem size grows. Its threading model is optimized for larger workloads, reducing the impact of synchronization overhead and resource contention. This makes Rust a more suitable choice for memory-intensive, large-scale applications.