Understanding Multi-threading in Computational Algorithms

Artem Shestakov

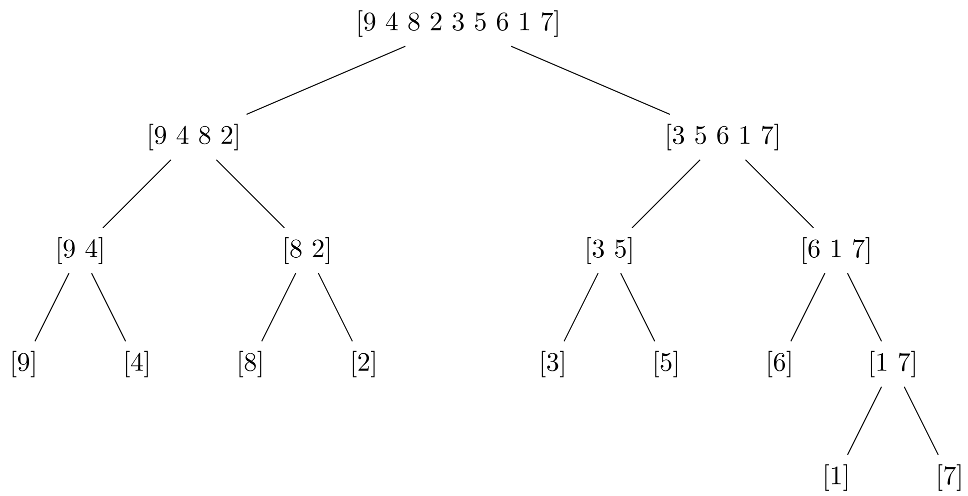
March 2024

1 Introduction to Multi-threading

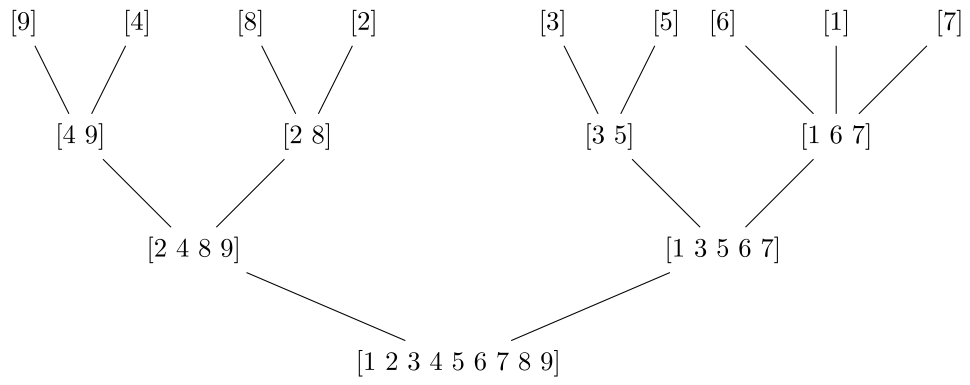
In the pursuit of maximizing computational efficiency, multi-threading has emerged as a pivotal concept. The idea is seemingly straightforward: divide a task into smaller, concurrent sub-tasks, each handled by separate threads. Intuitively, one might assume that the execution time with multiple threads should be roughly equal to the time taken by a single thread divided by the number of threads employed. However, does this assumption hold up in practice? We explore this question through the lens of a classical algorithm: Merge sort.

2 Merge Sort concept

In general, merge sort consists out of two parts:
1) Splitting array on smaller sub-arrays.



2) Sorting and merging sub-arrays.



3 Multi-threading in Merge Sort

Merge sort is a well-known 'divide and conquer' algorithm that breaks down a list into several sub-lists until each sublist consists of a single element and then merges those sub-lists to produce the sorted list. In its multi-threaded variant, separate threads handle the sorting of different sub-lists in parallel, which, in theory, should decrease overall sorting time significantly.

4 Methodology

We conducted an experiment using a merge sort algorithm implemented in C++ with multi-threading capabilities. The algorithm was tested on a set of randomly generated integers, and the sorting times were recorded as the number of threads increased from 1 to 28. This test was performed on a machine equipped with an Intel i7-14700K CPU to examine how multi-threading impacts performance.

5 Results

The results indicated an initial sharp decline in execution time as the number of threads increased. However, this improvement plateaued beyond a certain number of threads, and the execution time reduction slowed down significantly.

6 Discussion

The diminishing returns observed beyond a certain point can be attributed to several factors:

methodology.png

Figure 1: The experimental setup demonstrating the multi-threaded merge sort process.

- **Hardware Limitations:** A CPU has a finite number of cores, and once each core is running a thread, additional threads must wait their turn, leading to context switching overheads.
- **Overhead of Thread Management:** Creating and synchronizing threads incurs overhead, which can become significant when the number of threads is high.
- **Algorithmic Overhead:** The merge sort algorithm requires merging sorted sublists, which is a sequential process that can become a bottleneck in a multi-threaded context.

7 Conclusion

Multi-threading has the potential to significantly reduce execution time for algorithms like merge sort, but only up to the point where the hardware and algorithmic constraints allow. Beyond this, the overhead associated with managing multiple threads can negate the benefits of parallelism.

performance_graph.png

Figure 2: Execution time plotted against the number of threads.

8 Recommendations

When implementing multi-threading, it is crucial to:

- Benchmark performance to find the optimal number of threads for the given hardware.
- Use thread pools to minimize the overhead of creating and destroying threads.
- Consider the nature of the algorithm and the cost of synchronization between threads.

In conclusion, multi-threading is a powerful tool in a programmer's arsenal but must be used judiciously to extract its full potential.