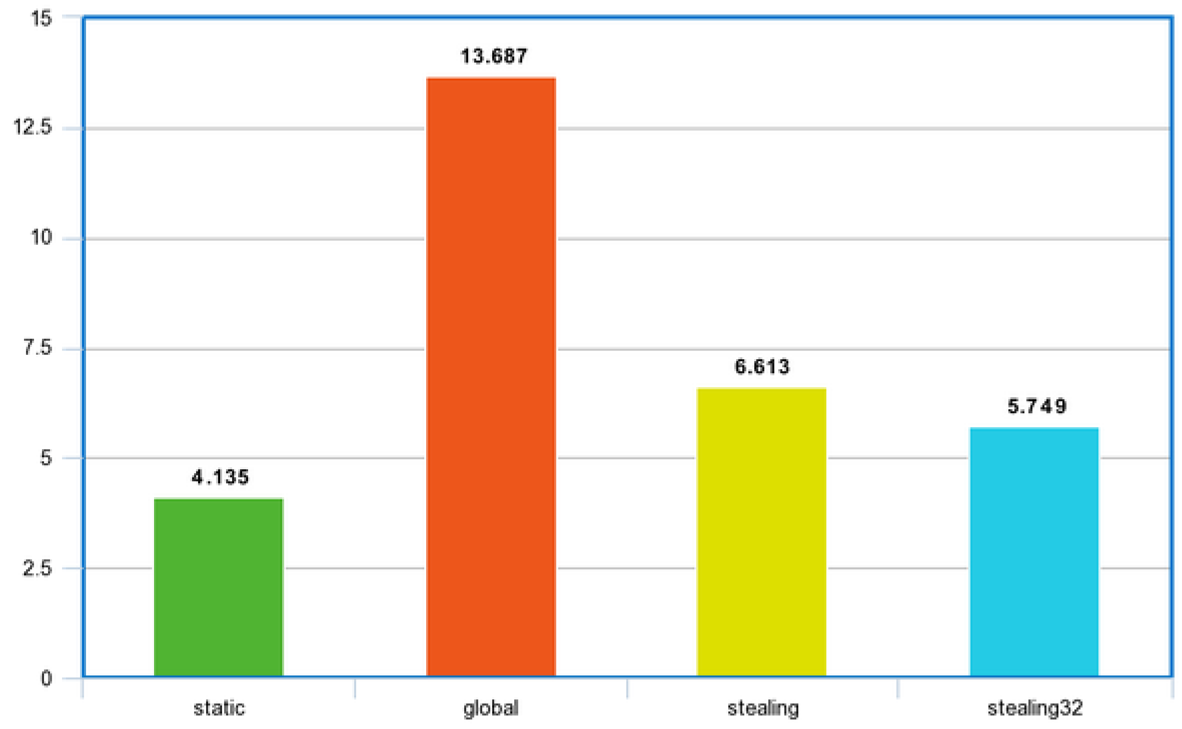
ari ‘aith’ iramanesh

Multi-thread Work Balancing and Work Stealing

Turnaround times on 4 threads:



	static	global	stealing	stealing32
Average Turnaround Time (sec)	4.135	13.687	6.613	5.749

In this section I compared the performance of multithreading using different scheduling techniques: static scheduling, global work stealing, fine-grained local work stealing, and coarse-grained local work stealing. The results were very surprising at first, but it’ll become crystal clear why there are such stark differences.

My initial prediction was that static scheduling technique wouldn’t perform highly because of potential thread imbalances (i.e., threads having different amounts of work to perform).

Global worklist-based work stealing solves work imbalance, so 3x turnaround time from static scheduling is haunting. This is the result of high amounts of **False Sharing** (i.e., when a thread attempts to access a memory location from a cache line that was modified by another thread). Cache lines are typically 64-bits. The operations performed by the threads in our program involve c++ integers, which are 32-bits. This means that threads simultaneously modifying spatially-local integer variables may actually be modifying the same cache line. Because computers seek to maintain cache coherency, the modified cache line must be written to memory, and retrieved again by either of the threads. This copying of the cache line to the cache is where the performance takes a hit. The effect is greatly magnified because the work that the threads perform are based on a 32-bit integer array. Since the global worklist increments sequentially, it’s likely that every iteration of work by threads is subject to False Sharing.

The local worklist-based stealing implementations are also subject to False Sharing, but to a lesser extent because a lot of their work is performed solo. What’s neat is that the static scheduling outperformed it, suggesting that the cost of cache flushing outweighs the benefits of work stealing in this scenario. The 32-element chunked version of this is slightly more performant, likely because it bypasses a little more of the cache flushing by having threads do more of their own work.