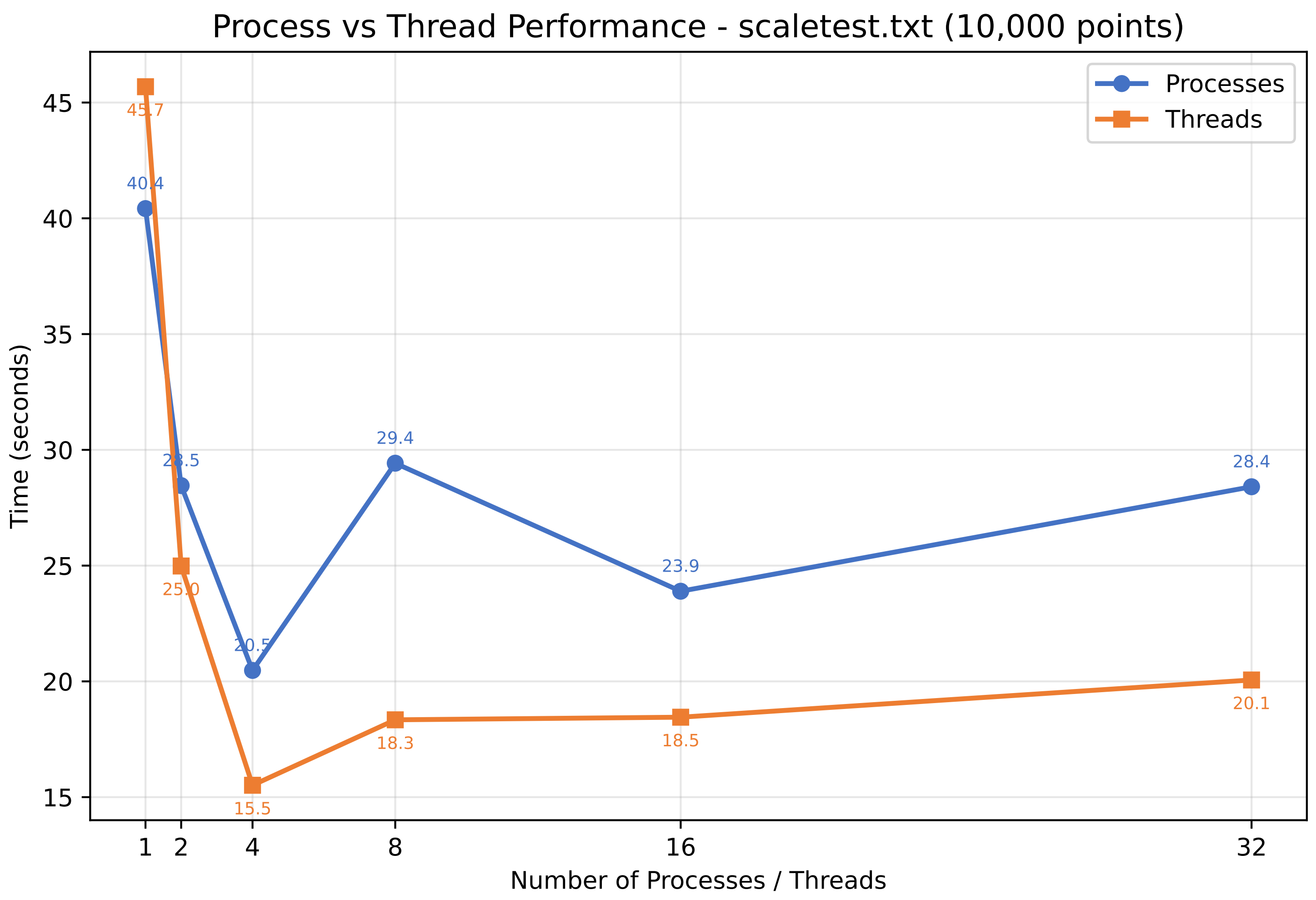

Performance Characterization

Both processes and threads improve significantly from 1 to 4 workers, with time dropping from ~40-46s down to ~15-20s. This reflects the parallelization of the $O(n^2)$ algorithm across multiple workers, with peak performance at 4 workers matching the VM's 4 CPU cores.

Beyond 4 workers, the two approaches diverge sharply. Processes degrade in performance, rising from 20.5s at 4 workers back to 28.4s at 32. Threads remain relatively stable, hovering around 18-20s from 4 to 32 workers with a slight increase due to scheduling overhead.

Technical Explanation

Threads outperform processes for three reasons:

1. Creation overhead: Each process spawns a new JVM, which requires loading the Java runtime, class files, and initializing the JIT compiler. Threads are created within the existing JVM and only need a new stack allocation, which is orders of magnitude cheaper.

2. Communication cost: Processes use pipes for IPC. The parent must serialize all 10,000 points to each child's stdin, and each child must parse them. With 32 processes, this means 32 full copies of the dataset are transmitted through pipes. Threads share the points list in memory directly with zero serialization cost.

3. Scaling behavior: Process overhead scales linearly with worker count (each new process = new JVM + full data transfer), so beyond the core count the overhead outweighs the benefit. Thread overhead is near-constant since workers share memory, so adding threads beyond the core count causes minimal degradation from context switching alone.