From this graph we can clearly see that there is a significant difference in the growth in time complexity between an efficient sequential implementation of quicksort and a parallel implementation with an abundance of processors. Specifically, from from the sequential implementation to the parallel one using eight processors, we saw speedup by a factor of four. However, we can see that the growth functions become indistinguishably similar when we use more than eight workers. We postulate that at this point, the cost of initializing, maintaining, and synchronizing additional worker threads cancels the benefit of additional parallelism.

In these figures, we see the effect of quicksort’s degenerate cases on the efficiency of parallel runs as they relate to the number of processors used. While there is still a general trend toward speedup as more workers are added, it becomes difficult to determine how any given number of worker threads will perform. We can see from the sorted case that the curves cross often, signifying that even though we choose a random pivot, the degenerate case can still lead to unpredictable results by creating partitions of unequal size. The takeaway is that, while our algorithm was written with careful consideration of degenerate cases, and therefore will terminate correctly in an acceptable amount of time, it is still no replacement for other divide-and-conquer algorithms such as mergesort that do not suffer from the same degenerate cases.

In this graph, we visualize the effect of using additional processors on runtime by using it as our independent variable. This figure strongly confirms what we observed from the runtime analysis graph, which is that our speedup is not linear in number of processors. We can also see that there is somewhat of a threshold past which additional parallelism becomes almost immediately ineffective, indicated by the n=250 million curve becoming suddenly flat at around p=4 processors.