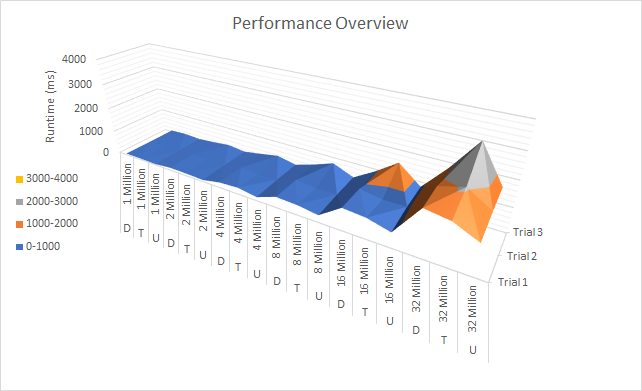
The Experiment:

In order to get a thorough performance evaluation on Beocat. Three different trials were conducted on three different files, the given pt0.c and a modified pt0-tiled.c and pt0-unrolled.c. pt0.c was unchanged aside from an ever increasing array size to test scalability. The algorithms in pt0-tiled.c and pt0-unrolled.c were changed in each trial to determine which variant of the algorithm was most efficient. In Trial 1, the tile size was a four by four block in pt0-tiled.c and the code was fully unrolled in pt0-unrolled.c. Similarly in Trial 2, the tile size was changed to two by two and the code was unrolled by four. Lastly in Trial 3, the tile size was eight by eight and the code was only unrolled by 2. Each test was ran five consecutive times on 1, 2, 4, 8, and 16 cores respectively. Given that this code is single threaded, an average of the five times was taken to produce the represented run times. This sequence of tests was repeated given char array sizes of 1, 2, 4, 8, 16, and 32 million respectively. For the sake of space in the “Performance Overview” graph, D represents the default pt0.c file, T represents pt0-tiled.c, and U represents pt0-unrolled.c.

The Results



All of the tests were confined to the elves class of nodes. For the most part, everything ran within the second tier (between elf57-elf72). These computers feature two 10-Core Xeon E5-2690 v2 processors with 96gb of RAM. The exception to this was the 16 core tests. At runtime, I would receive the first four tests within a few seconds but the last test would occasionally take upwards of a couple minutes. Upon closer inspection, all of these tests ran on the tier 1 elves around elf20. For comparison, these nodes have two 8-Core Xeon E5-2690 processors with only 64gb of RAM. The resulting data was consistent with the other data so I assumed this wasn’t a problem. If I had to guess, this could’ve been caused by combination of things. The larger 16 core requirement, a heavy load on the second tier, and open space on the first. KSTAT showed that a couple massive programs were running when these tests were conducted. In regards to the default file and the unrolled code, the results were as expected. The amount of time required to run the given tests generally grew exponentially with the doubling of array size. As the code is unrolled further, the amount of run time require to process the data decreases. This really highlights just how expensive for loops are. At its peak, the unrolled code was 30% faster than the default. Tiling with a four by four block proved to be the most efficient in its tests. Unfortunately this only resulted in an approximately 12% speed up. Curiously, the two by two block follows the same line as the four by four block then takes a massive jump in run time around the 32 million array size. Being that the cache on a Xeon E5-2690 is only 20mb this makes sense. Tiling by design is supposed to take advantage of a higher percentage of cache hits to create a speedup. If given more time, I would run more tests in varying dimensions to determine which configuration would result in the highest percentage of cache hits. Inversely, an interesting point I gathered from this experiment was the fact that an eight by eight tiling block is incredibly inefficient. This of course produced the peak in the graph above. This code produced a 60% slow down from the given file. Its design most likely results in the lowest percentage of cache hits, causing the program to hit higher levels of cache and main memory more often. Below are the results of the individual trials.

