Assignment 2

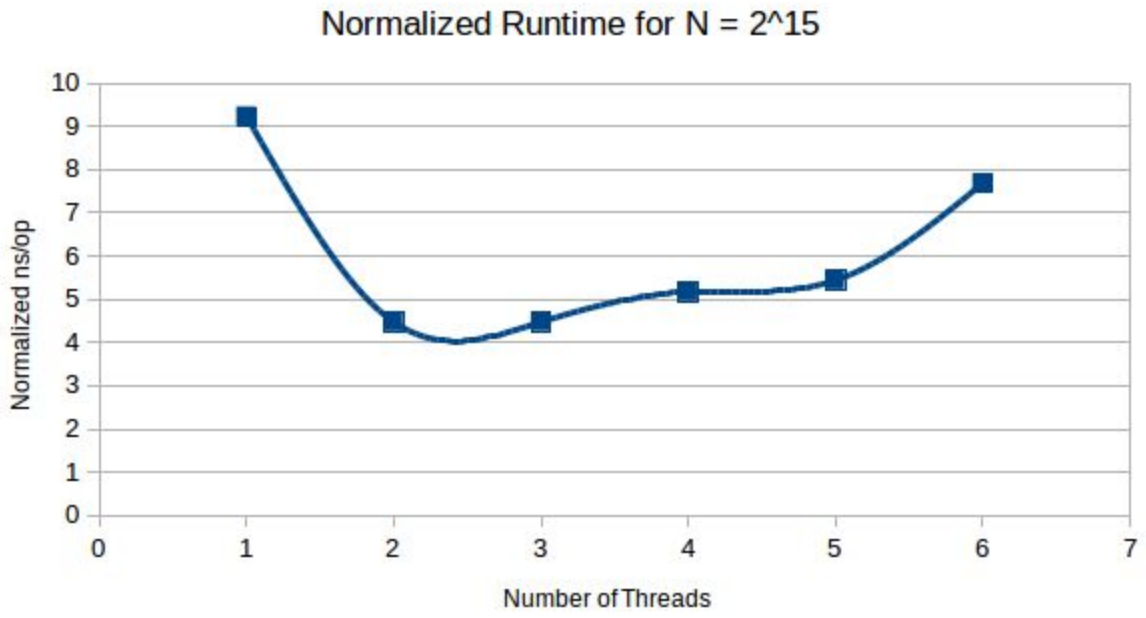
D. Aaron Wisner (daw268)

Permutation Way 1

Two versions of the function were written to permute the 2d array A, perm_w1 (single threaded) and perm_w2_mt (multithreaded). The single threaded algorithm iterated through the columns of A, starting at the diagonal element and working downward, tracking the element with the largest magnitude and its index. When the end of the column was reached the the entire row containing the largest element was swapped with the row that contained the starting element for that column (the diagonal element).

The algorithm was parallelized by taking advantage of the parallelism in the linear searching for the largest element's indices. At the start of the iteration for each column, the column was divided into chunks and distributed evenly among the threads. Each thread would search through its chunk, then return the indices of the largest element with the largest magnitude. The results were merged in a serial fashion by the caller who spawned the threads. The calling thread iterated through all the return values of each thread and found which thread found the largest value. The row corresponding to that indices was then serially swapped as it was in the single-threaded version of the program. To avoid unnecessary thread creation, if a column had less than a certain number of elements to be searches the elements would be searched in a serial fashion.

Overall, the speedup was substantial. The parallelized version was unsurprisingly slower for N < 2^10. However, as N was grew larger than 2^10 the speedup asymptotically approached ~50% faster over the single threaded version. To view the runtime of all scenarios, seen below, the runtime for each N, was normalized by N*N.



Different numbers of threads were tested, and 3 appeared to offer the best performance. For large N's, after 3 threads, adding additional threads resulted in slight slowdowns, most likely due to the fact that the computer being benchmarked has only 4 cores. Overall, there was a substantial speedup for little extra work.

Results

N	1 Thread	2 Threads	3 Threads	4 Threads	5 Threads	6 Threads
256	2.225983	62.463806	74.520691	169.432816	242.637939	286.240784
512	3.106388	28.57835	37.074181	70.826111	88.980698	97.892151
1024	2.23195	13.194872	16.16037	23.290739	32.695847	115.782013
2048	7.78377	8.801523	9.432898	10.434961	17.603125	58.344067
4096	9.336759	6.656889	7.068805	7.378581	10.670802	31.476429
8192	9.757241	5.654371	5.452819	5.756567	6.322471	17.33145
16384	10.050348	5.016809	4.533185	4.59476	4.914472	10.265168
32768	9.224268	4.47176	4.475694	5.172628	5.44165	7.686184

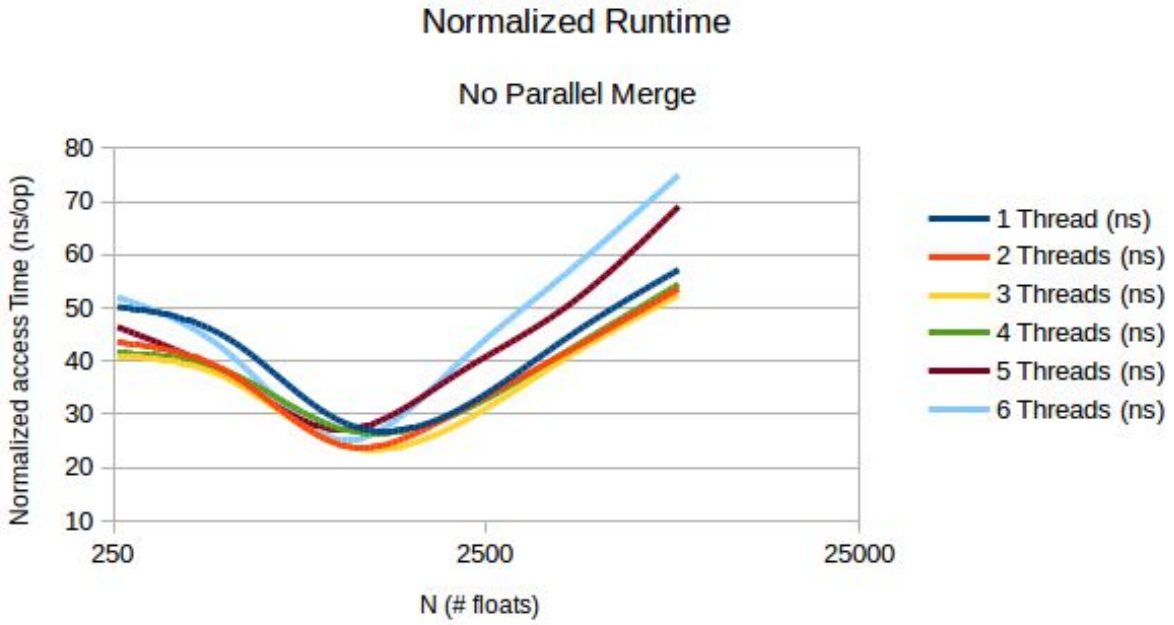
Macros to Play With:

- **N_START -** The smallest test size for the NxN array
- **N_END -** The largest test size for the NxN array
- **NUM_THREADS -** The number of threads to use in the parallel algorithm
- **NUM_RUNS -** Number of runs to average for each N

Permutation Way 2

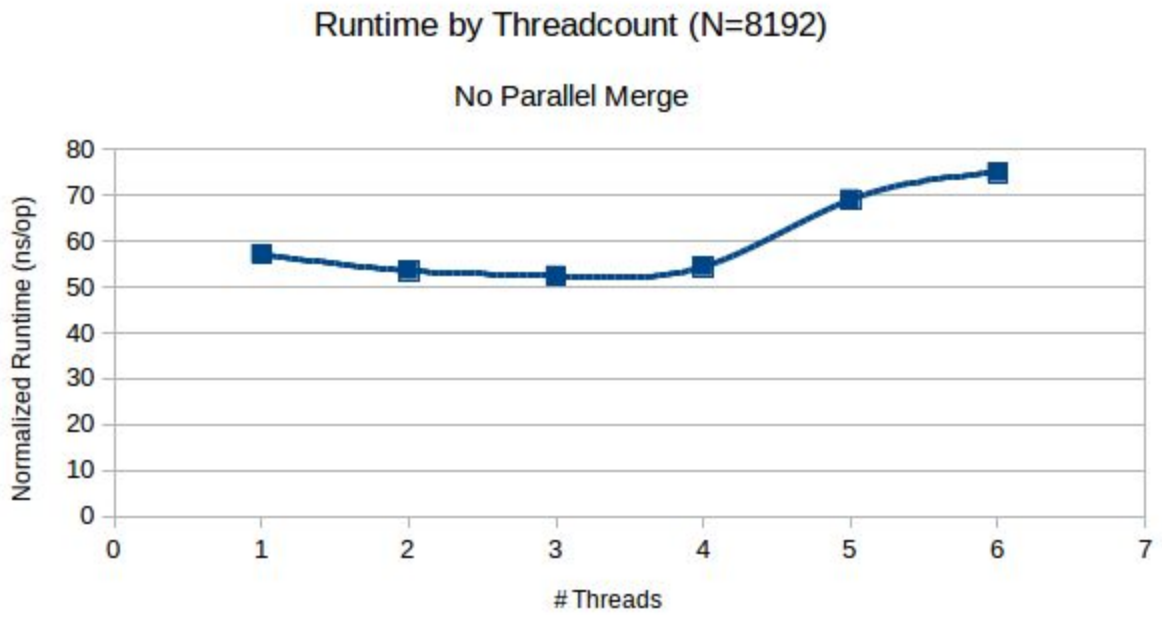
Two versions of the function were written to permute the 2d array A, perm_2 (single threaded) and perm_2_mt (multithreaded). The single threaded algorithm iterated through the columns of A. Starting at each of the diagonal elements in the column, the algorithm then called a version of mergesort that swaps rows instead of elements. As an optimization, the algorithm would automatically switch over to insertion sort if the number of elements to sort in the recursive calls reached <= MSORT_BASE_CASE_ISORT. Additionally, as a way to reduce the overhead of expensive row copy operations, both parallel and serial algorithms sorted a 1d array of pointers of type “float (**)[ ]” that pointed to the rows in the NxN matrix. Rather than rearrange the rows, only the pointers to the rows have to be rearranged. This dramatically improves the speed of both the single threaded and the multithreaded implementations. Now instead of each swap during sorting costing sizeof(float)*N bytes, it only costs sizeof(float (**)[ ]) bytes.

Parallelizing the algorithm was initially easy for taking advantage of the two recursive calls to mergesort while merge sorting each column. Every call to merge sort was put in a separate ‘#pragma omp section’. Since the two recursive calls operate on separate sections of the array, there is no issues with false data sharing. However, this resulted in insignificant performance improvements of at best a few percent as seen below.



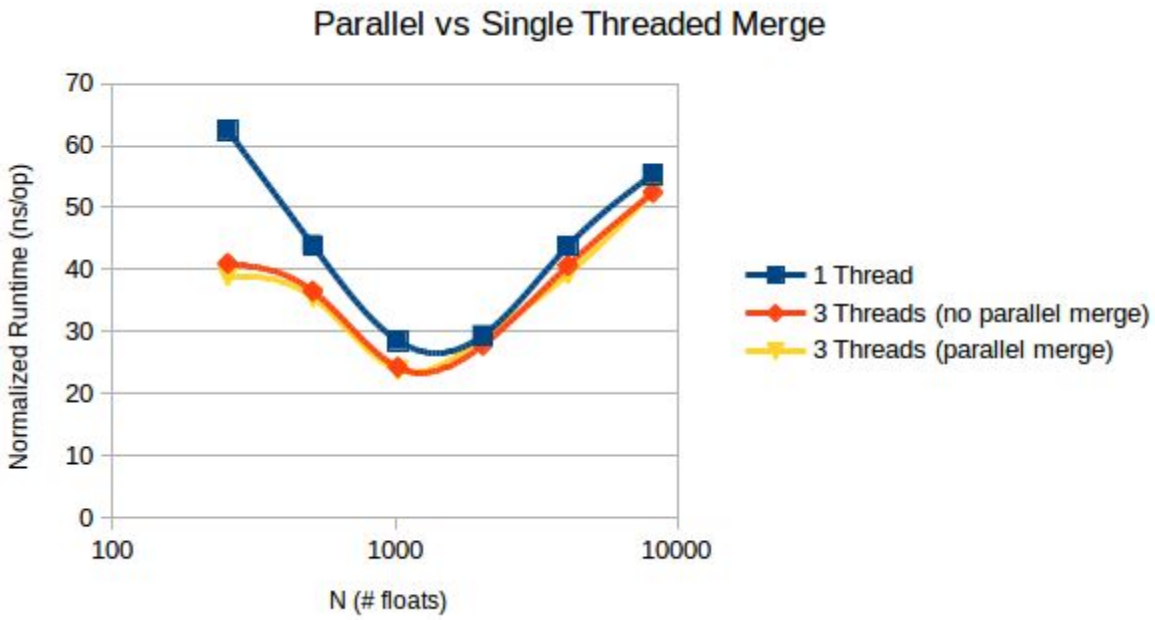
Results (no parallel merge)

N	1 Thread (ns)	2 Threads (ns)	3 Threads (ns)	4 Threads (ns)	5 Threads (ns)	6 Threads (ns)
256	50.192993	43.450287	40.978363	41.55014	46.399185	51.989792
512	44.057301	37.726742	36.522057	37.84795	36.937943	41.315536
1024	28.588165	24.193726	24.232319	27.233459	27.109531	25.197437
2048	30.420116	30.148243	27.784653	29.984947	37.226509	38.822201
4096	44.314312	41.641716	40.640163	41.77187	50.218925	56.510601
8192	57.164963	53.572102	52.424168	54.44128	69.054001	74.936409



As seen above the algorithm showed slight improvements of up to three threads on the order of a few percent. After profiling, the bottleneck was with the serial merge inside of mergesort, which was not parallelizable as is. After reading through the wikipedia page on merge, a recursive parallelizable algorithm was discovered for merging two sorted arrays. An element in the middle of the larger array is picked then using a custom binary search implementation, the smaller array is divided into two. Merge is then called again recursively on the two left array segments, and again on the two right array segments. Again, these two recursive calls can be embedded inside '#pragma omp section'. Unfortunately, even though implementing this took hours, it still showed modest to little performance improvements over the serial merge. Trying to tweak base cases to revert

to serial algorithms again showed little improvement. Below is a plot comparing the normalized runtime with and without parallel merge for N = 2^13.



Results (parallel merge)

N	1 Thread (ns)	2 Threads (ns)	3 Threads (ns)	4 Threads (ns)	5 Threads (ns)	6 Threads (ns)
256	56.465256	45.985596	38.917145	41.577881	46.497635	48.984314
512	42.308655	37.816959	35.536877	38.167133	41.20615	39.015041
1024	28.014751	22.787006	23.798838	23.32548	24.429401	25.422005
2048	29.72431	28.444815	28.989902	27.022079	32.286049	35.583641
4096	43.103123	39.28117	39.252258	39.452946	48.472225	56.507698
8192	56.589928	52.173229	52.540817	53.095905	66.66777	76.111496

Macros to Play With:

- **N_START -** The smallest test size for the NxN array
- **N_END -** The largest test size for the NxN array
- **NUM_THREADS -** The number of threads to use in the parallel algorithm
- **NUM_RUNS -** Number of runs to average for each N
- **MSORT_BASE_CASE_ISORT -** The number of elements in a partition and below where mergesort will switch to calling insertion sort

- **MERGE_MT_BASE_SERIAL -** The number of elements in a partition and below, where parallel recursive merge will switch to serial merge (only if MERGE_MT enabled)
- **MERGE_MT -** Enable multithreading in the merge portion of mergesort with custom parallel merge algorithm