	Threads	Radix	Recursive Quicksort	Routine 1	Routine 2	Routine 3	Routine 4
	1	648	902	945	1137	1442	1487
Data Items	2	676	903	972	1538	1596	1758
10^4	4	650	900	946	2268	1878	1962
Max Value	8	646	947	962	5515	3591	2773
2^3	16	716	921	971	7171	10595	4063
	1	9879	11396	11742	13571	14135	17193
Data Items	2	9887	12106	11714	14739	23117	32028
10^5	4	9804	11418	11651	37496	48029	45494
Max Value	8	9856	11555	11739	24494	66480	26907
2^4	16	10501	11565	11899	47107	102696	70903
	1	184517	146985	153533	172289	165880	177035
Data Items	2	166195	140692	142456	165384	230887	286912
10^6	4	186146	142773	144785	177540	391688	691599
Max Value	8	162106	145337	155717	194671	654073	681713
2^5	16	164512	140763	143365	305539	544571	695105
	1	4561868	2159524	2214598	2423235	2718723	3322066
Data Items	2	3959573	2277891	2182423	2568876	2523002	3019284
10^7	4	4862295	2467669	2409730	1609692	4938046	6600329
Max Value	8	3316483	1901102	1918338	1646864	6272104	8060048

2^6	16	3173042	1850048	1880711	2361762	6883102	7712843

Initial Sequential Analysis

It appears that radix sort is faster with smaller data sets, however as the size of the data set grows large, the quick sort methods become faster. Radix sort while only requiring a limited number of iterations consumes more memory and requires more data movement.

The custom stack version of the quick sort algorithm saves some time in some cases, but requires additional overhead. The custom stack implementation requires memory in 3 locations, the data on the heap, data on the custom stack (also in the heap), and the main stack. This can be less than ideal, as there is overhead in dealing with the custom stack and probably more pages are required as a result of the different locations.

Parallel Implementation

Of the parallel routines, routine 2 appears to be the most effective. It requires minimal communication, as threads are spawned on demand and shut down at completion. There is no need for blocking synchronization, which routines 3 and 4 suffer from. Routine 4 is faster at lower thread and data counts, primarily due to the better coordination of work, and lack of busy-waiting. However at higher thread and data sizes, the cost of coordinating everything out ways the benefits; the busy-waiting routine 3 ensures calculation occurs at minimal communication cost, albeit inefficiently.

There are some flaws with parallelizing the quicksort algorithm. Firstly only one thread can initiate the first partitioning, and then 2 threads, and then 4 (assuming optimal conditions). Not only are the first iterations limited to small threads, the work in the first few partitioning is larger than any other later, and thus early on not all processors can be employed to work. Secondly the algorithm is non-deterministic, there's no pre-planned approach that can be taken to maximize calculation and minimize communication. Of the 2 methods considered, spawning threads on demand, or a master-slave approach to solving the problem there are flaws. Creating threads consumes resources, and the master-slave approach requires a lot of coordination and protection of shared variables.

Routine 2 could be improved by only spawning one thread for the other half of each partition. This would reduce thread creation costs at the expense of a slightly more complicated algorithm.

Routine 3 would only benefit from a small number of threads per processor; busy waiting with too many processors on the initial threads will suffocate "expansion" of the available atomic workload.

Routine 4 stands to have the most improvement after routine 2, by splitting the condition variables and mutual exclusion objects as much as possible. Much of the time in this routine is spent waiting for a thread to "wake-up" to the new work available.