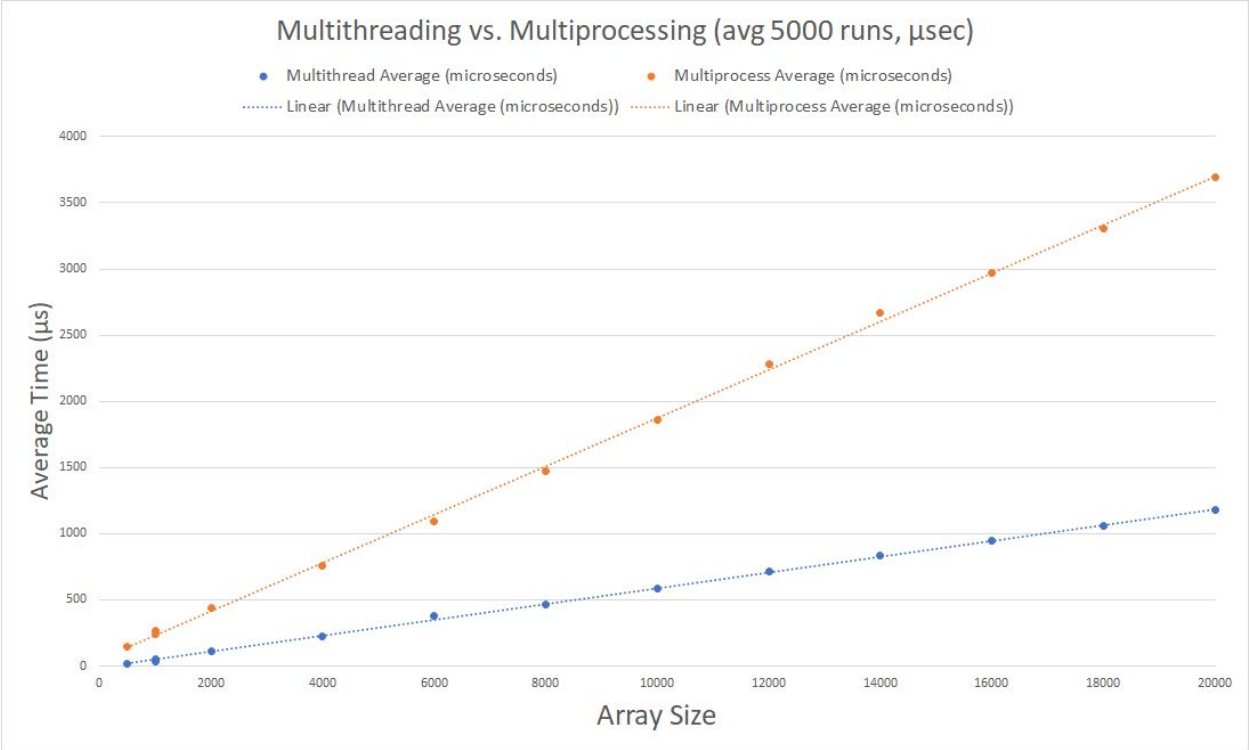
Assignment 2: Spooky Searching
Peter Ling (pl466), Joseph Natale (jrn84)

General Trend & Tradeoff Potential - Processes vs Threads:

Array Size	Multithread Average (microseconds)	Multiprocess Average (microseconds)	Multithread Stddev	Multiprocess Stddev	**Multithread CV**	**Multiprocess CV**
500	20.4666	148.3178	9.3850	17.2124	**0.46**	**0.12**
1000	32.26	242.8244	10.7187	23.4129	**0.33**	**0.10**
1001	48.9208	265.2446	16.5487	47.8556	**0.34**	**0.18**
2000	114.957	438.3638	26.8360	47.4570	**0.23**	**0.11**
4000	221.5992	755.5636	27.5009	97.0883	**0.12**	**0.13**
6000	375.5822	1095.4134	28.4850	139.2966	**0.08**	**0.13**
8000	466.1026	1470.3282	54.8022	177.4487	**0.12**	**0.12**
10000	583.204	1857.0404	47.0285	219.9444	**0.08**	**0.12**
12000	716.6264	2285.1662	40.2933	265.1564	**0.06**	**0.12**
14000	835.9878	2664.7226	45.4254	302.5229	**0.05**	**0.11**
16000	945.7932	2969.9818	43.7721	325.3224	**0.05**	**0.11**
18000	1055.9134	3310.2946	82.4431	541.0753	**0.08**	**0.16**
20000	1179.0666	3695.3316	41.0886	612.9286	**0.03**	**0.17**



Our searchtest program ran each test case 5000 times for both multithreading and multiprocessing, and the table above represents the set of means obtained from the test runs. Notably, all of the test runs have coefficients of variation well below 1, signifying that these tests display low variance. Therefore, it's sensible to make conclusions from this data set.

Note that both multithreading and multiprocessing scale linearly with the size of the array. This is to be expected, as we're conducting a linear search. *At no point* does multiprocessing perform better than multithreading; multithreading is strictly better at any input in this range. Judging by the linear slopes, we can expect that this will remain true for any larger array size 'n'.

Multithread Properties	Values
Sample Array Sizes	13
Sum of X	112501
Sum of Y	6596.4798
Sum of XY	91045938.22
Sum of X^2	1542252001
(Sum of X)^2	12656475001
Linear Regression Slope	0.059718451

Multiprocess Properties	Values
Sample Array Sizes	13
Sum of X	112501
Sum of Y	21198.593
Sum of XY	287126739.5
Sum of X^2	1542252001
(Sum of X)^2	12656475001
Linear Regression Slope	0.182310426

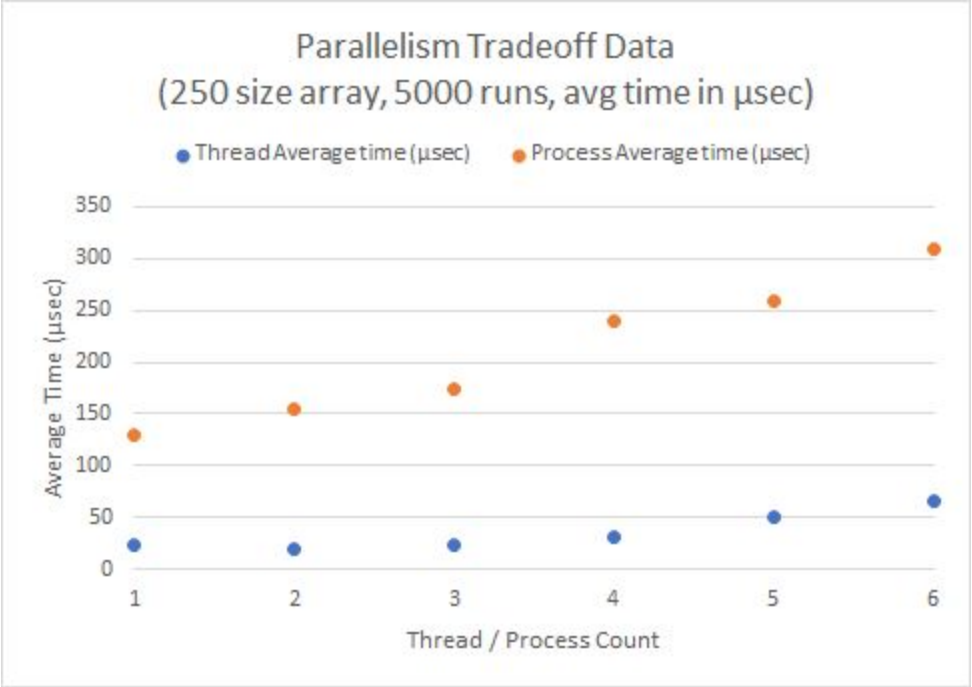
After running linear regressions on this data, we found that for an array of size x, multiprocessing took 0.1823x microseconds (ignoring intercept) to find our target. Multithreading, on the other hand, took only 0.0597x microseconds (ignoring intercept). Therefore, multithreading is about 3 times faster than multiprocessing on average.

This statement is bolstered by examining a few data points on the graph. Searching a size 12,000 array with multithreading took about the same amount of time (716 microseconds) as searching a size 4,000 array with multiprocessing (755 microseconds). A size 18,000 array takes 1,055 microseconds with multithreading, and a size 6,000 array takes 1,095 microseconds with multiprocessing. It's clear that within the confines of a linear search where you can only search up to 250 indices per worker, multithreading is superior.

Parallelism Tradeoff Data - Optimal Worker Count:

Worker Count	Thread Average time (µsec)	Process Average time (µsec)
1	23.9838	130.4428
2	19.2814	154.4992
3	23.9658	173.3112
4	31.5328	240.3498
5	50.3246	259.4664
6	67.0172	308.1434

Thread Stddev	Process Stddev	Thread CV	Process CV
20.279097	45.033551	0.845533	0.345236
9.684429	24.373125	0.502268	0.1577557
9.231104	16.974192	0.385178	0.0979405
11.773931	31.99549	0.373387	0.1331205
9.988873	25.547528	0.198489	0.0984618
19.554288	31.051674	0.29178	0.1007702



Just like the other set of tests, these tests were ran 5,000 times each for multithreading and multiprocessing. After building the program with the Makefile, you can also test this with "./searchtest 1 0" ("./searchtest 1 1" if you want a verbose mode — that's 30,000 iteration printouts).

Looking at the processes, it's clear that at no point do we want to add additional processes. We're best off sticking with the minimum amount of processes if we're constrained to 250 indices per process.

For multithreading, we see performance gains over 1 thread for the 2 and 3 thread counts. 2 threads are superior to 3, however, and we start seeing greater performance losses for subsequent core counts.

Cutting the work in half by introducing a 2nd thread seems to provide optimal performance. We're restricted to 250 indices per thread, so we didn't test scenarios like 1,000 size array split into 2 500 size threads, but they'll likely display similar trends.

Another thing to note is that the performance for different sized arrays with identical worker counts seems to be remarkably similar. In other words, check the first table for array size 1,000, which generated 4 workers and ran at 32 microseconds for multithreading and 242 microseconds for multiprocessing. Now, look at the results for 4 workers on a size 250 array.

4	31.5328	240.3498

They’re almost the same! We think this proves that for the linear search scenario, the overhead of generating processes/threads and switching between them is the vast majority of the work done by the CPU. Otherwise, there’d be a sizeable difference between 4 workers with size 63 loads and 4 workers with size 250 loads.

Conclusion

Multithreading is superior to multiprocessing in linear time scenarios. Additionally, in linear time scenarios, the overhead from initializing threads/processes is the bulk of the work. If you seek a performance gain from parallelism, the task must be complicated enough such that the time saved by an additional thread exceeds the time lost by generating and swapping between that thread.