Assignment 2 Analysis - afl59, jdm362

The format to run the program is ./(proc/thread) <length of array> <size of interval that proc/thread searches in>

Because of the sheer volume of our results, we have organized our result output into directories classified by machine name for quality-of-life. These results may be located under the ‘results’ directory that was packaged with the submission .tar.

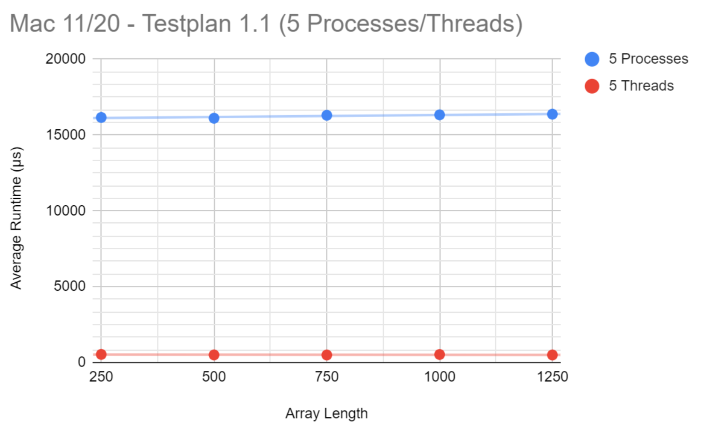
We ran our test plan on four different machines at various times to see how various factors would affect our results. All of the results from our runs that were used to calculate the averages used as data points on these graphs is provided in the results folder.

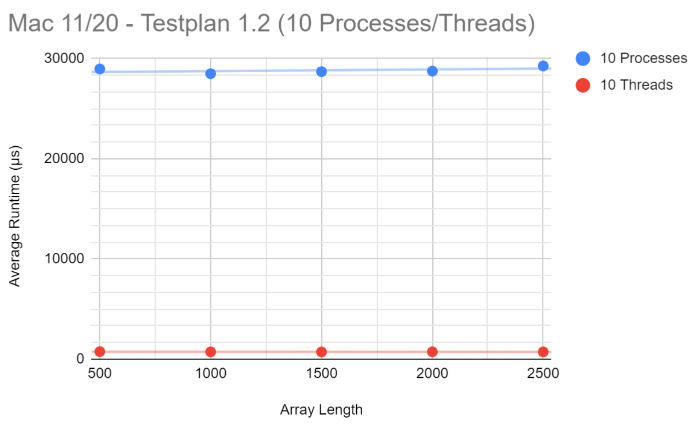
It is important to note that for all our results, we ran 5 trials per test plan item and average out our final results, so as to obtain a more uniform result set. It is important to note that machine usage may have varied throughout the execution of our test plan, and while our multi-trial approach may have removed some of the outliers, it is more than likely that at high-usage periods, the range of values is much higher than at more stable periods of usage.

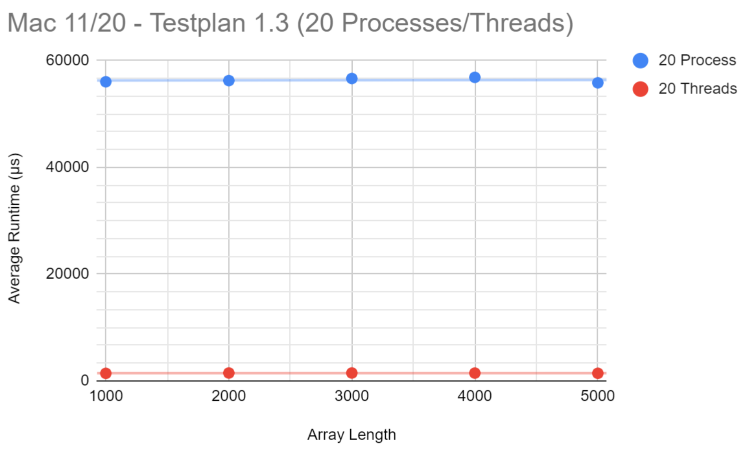
<u>Mac:</u>

These tests were performed with minimal activity on the machine as it was a personal computer belonging to a group member. Most processes were shut down at the time of testing. Our tests on this machine produced staggering results in favor of threads over processes.

Test 1:

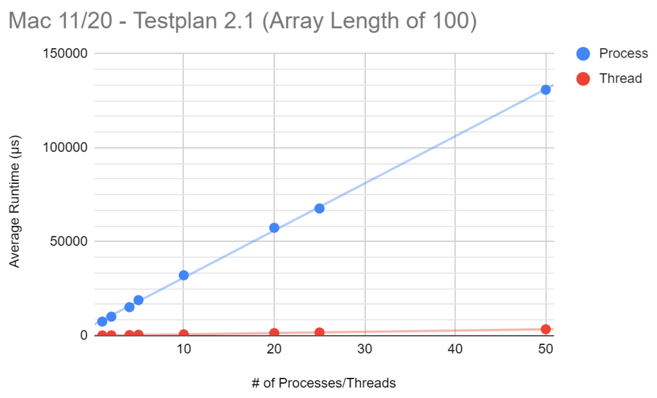


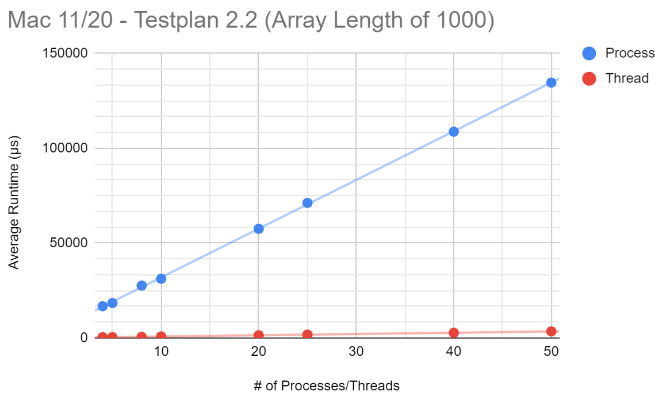


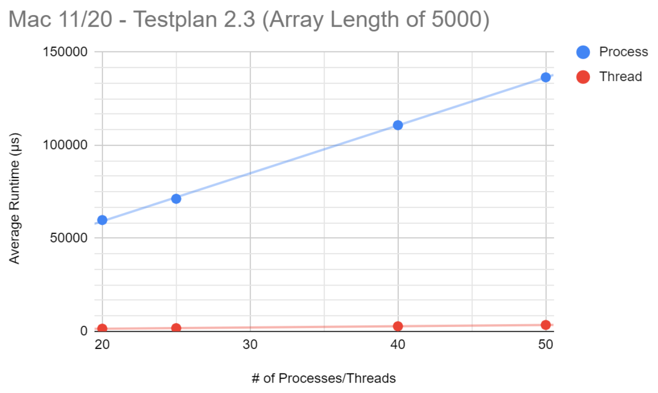


These results show a massive difference in runtimes and a relatively stable runtime when keeping the number of processes and threads constant. Test 1.3 (20 processes) had an average runtime of 56,331.419 microseconds for processes and 1,449.268 for threads.

Test 2:

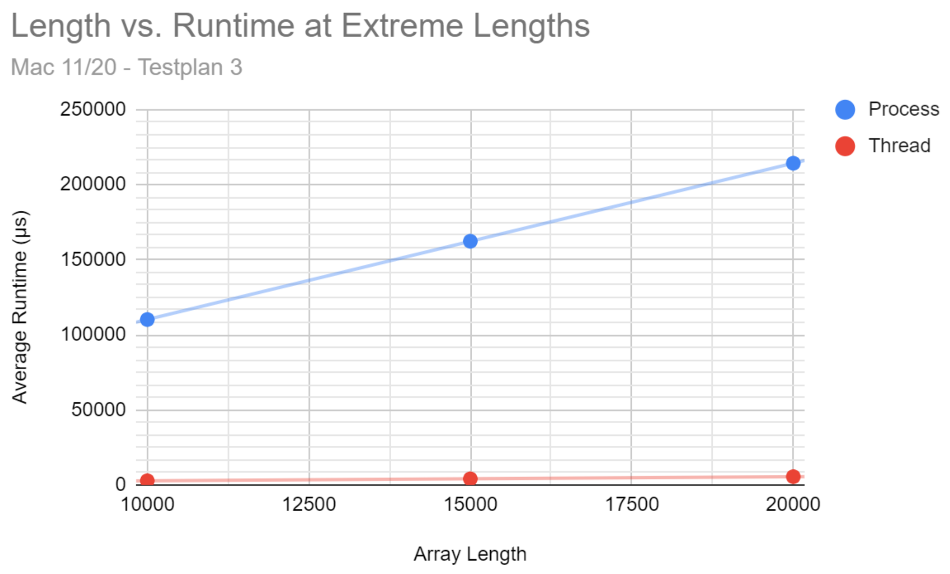






These tests illustrate that as the number of processes and threads increased, so did the runtime. This would indicate that generally it took more time to build a new process or thread to search than it would take to search without splitting. Test 2.3 again shows the massive difference in runtimes between processes and threads. Processes ranged from 59,831.436 to 136,498.396 microseconds from 20 to 50 processes, while threads ranged from 1,473.386 to 3,472.254 microseconds over the same interval.

Test 3:



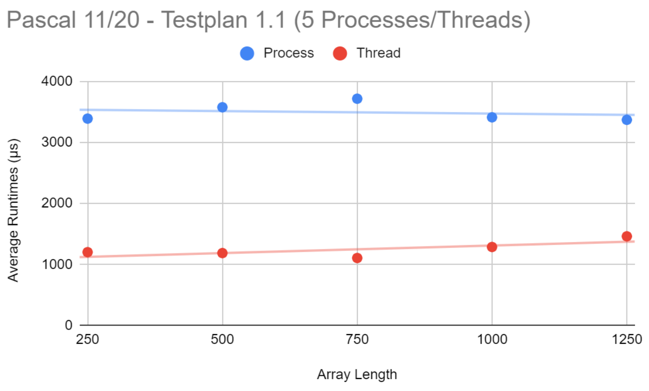
Test 3 followed a similar result with threads significantly outperforming processes again, this time at extreme lengths. At array length of 20,000, processes took 214,337.486 microseconds on average, while threads took 5,552.7 microseconds.

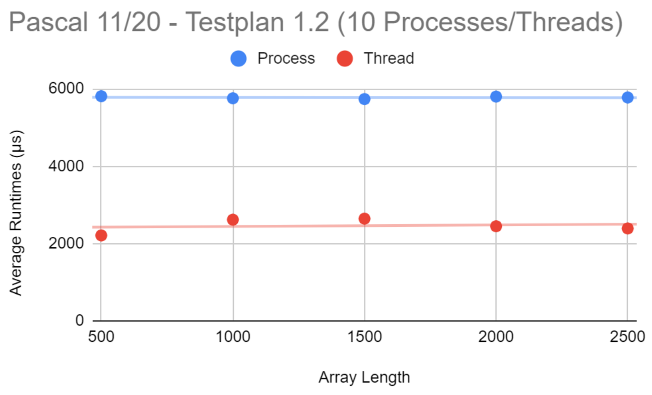
Mac Summary:
On this machine, processes were overwhelmingly inferior to threads. Threads were always significantly faster. Both processes and threads took longer to complete their search as you increased the number of them to search over a fixed interval, indicating that it is better to have larger intervals and less overall divisions of the search range. We could not find a point on this machine where processes were comparable to threads. The process runtime consistently increased faster than the thread runtime. Furthermore, we can conclude that on this particular machine that the cost of generating more processes dominates the cost of creating new threads by far, which we believe to be sensible given that the specifications of the Mac system were highly bounded by the relatively weak CPU when compared to the memory pool that was available to that machine. It also makes sense that for this particular machine, there is no trade-off point between threads and processes, given the resources available.

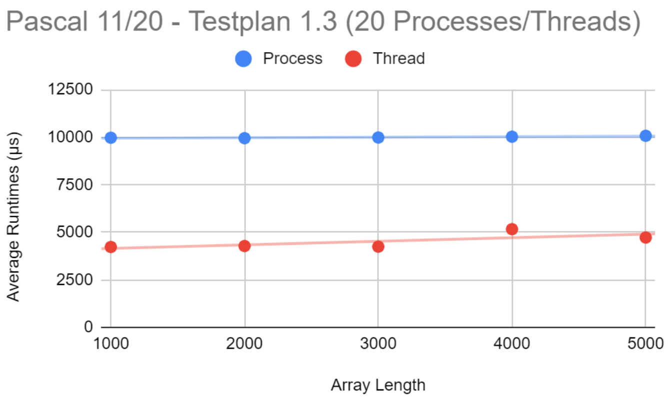
Pascal:

The iLab machine “pascal” was tested at night with very little activity outside of these tests, with one of our group members being the sole active user at the time of the tests. Similarly to mac, these results heavily favored threads, but not to the same extremes as before. The results of processes and threads were significantly closer than the previous machine implying that the specifications of the computer have a large effect on the runtimes, particularly for processes which took significantly longer of the mac than on pascal. For example, Test 1.3 for the mac averaged 56,331.419 microseconds for processes and 1,449.268 for threads, while on pascal this test averaged 10,002.951 microseconds for processes and 4,532.335 microseconds for threads.

Test 1:

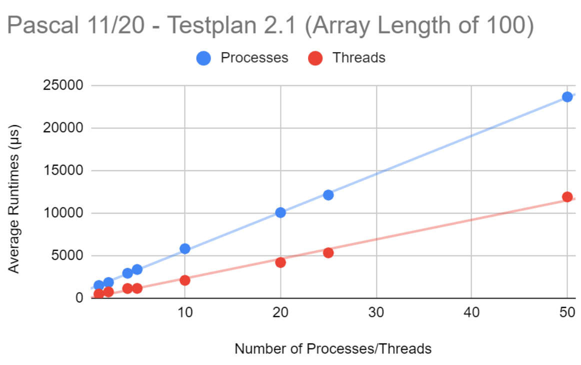


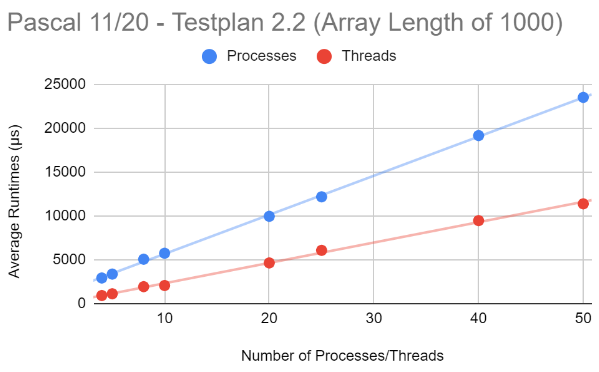


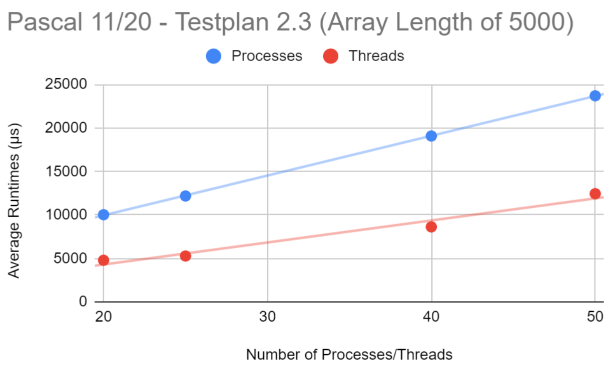


These results again imply that threads are generally faster than processes and that with a constant number of processes or threads, the search takes a relatively constant amount of time.

Test 2:



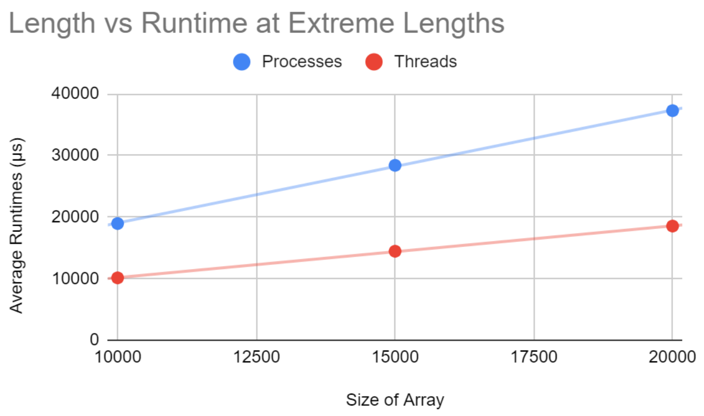




These results further strengthen the conclusion that threads are significantly faster than processes. Furthermore, these graphs indicate a linear increase in runtime according to the number of processes. For processes there was roughly an increase in runtime by 5,000 microseconds for every 10 additional processes. For threads there was roughly an increase of 2,500 microseconds per 10 additional threads. The following are the data points for Test 2.2 that were multiples of 10. You can see this general trend of linear growth of the runtimes in the table below.

Number of Processes/Threads	Processes	Threads
10	5740.284	2056.91
20	9958.09	4635.798
40	19176.618	9466.66
50	23535.758	11381.984

Test 3:



Test 3 further shows this relatively linear increase in runtimes. Threads stayed consistently faster than processes at all sizes.

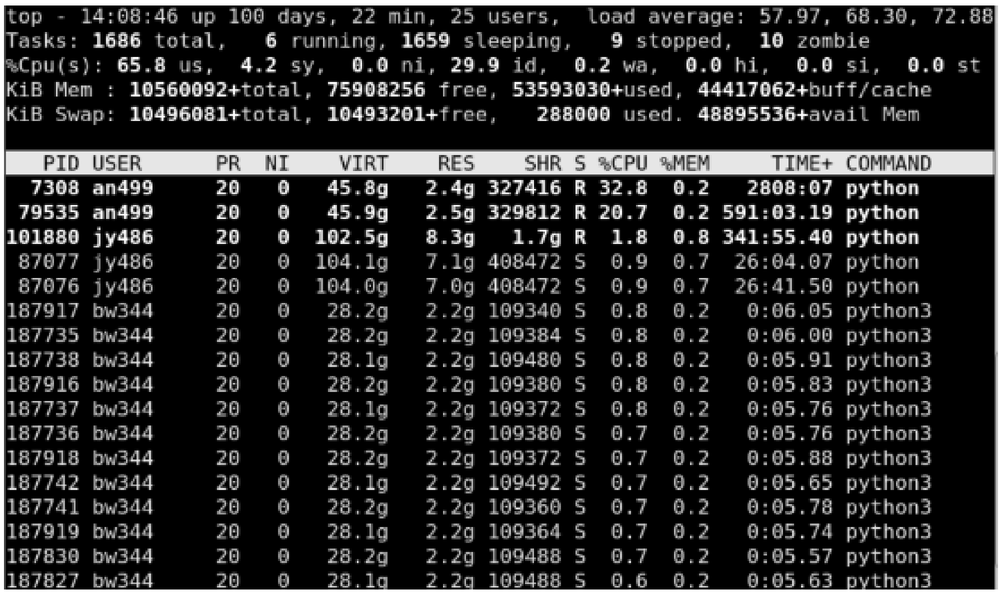
Pacal Summary:
Our tests on pascal provided more evidence that threads were consistently faster than processes at all sizes. Furthermore, the relatively linear growth of the runtime with an increase in processes or threads regardless of the size of the array being searched indicates a linear relationship between the number of processes or threads and runtime. As Test 1 and the following data from Test 2 shows, the only factor in determining runtime in conditions of minimal CPU usage is the number of processes or threads being created.

Size of Array	Number of Processes/Threads	Process Average Runtime	Thread Average Runtime
100	50	23668.298	11905.408
1000	50	23535.758	11381.984
5000	50	23694.11	12445.98

The size of the array being searched and the size of intervals being searched made minimal difference in runtime compared to the time it took to create the processes and threads.

<u>ilab2:</u>

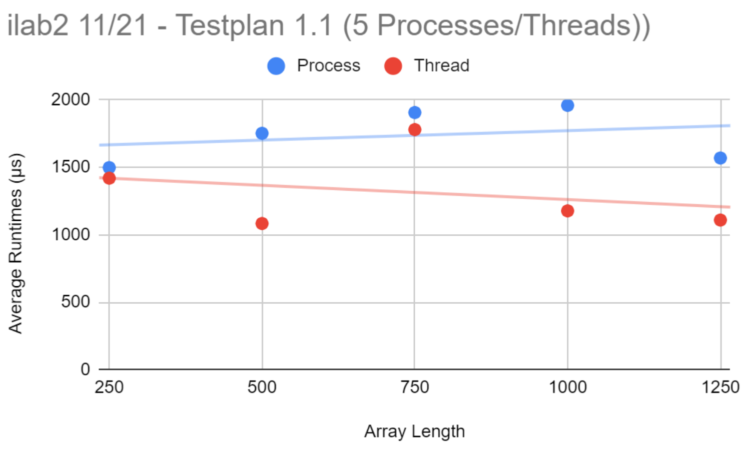
The next machine we tested was ilab2 during the day. The machine was very busy at the time of the tests, and it had a major impact on our results. 25 users were logged in with one user using over 50% of the CPU at the time of the tests.

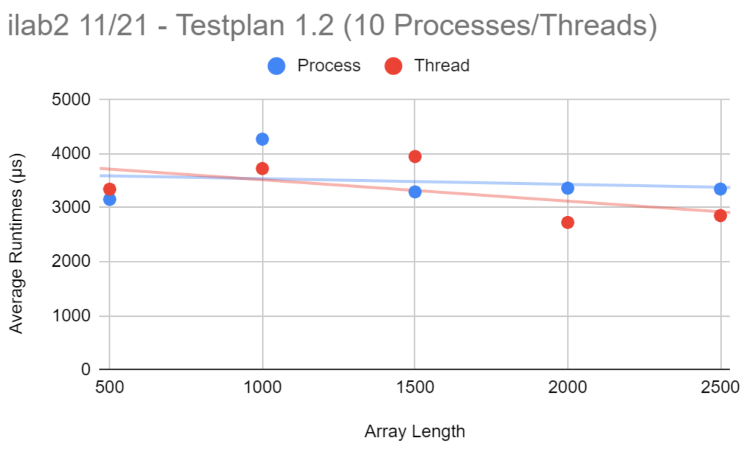


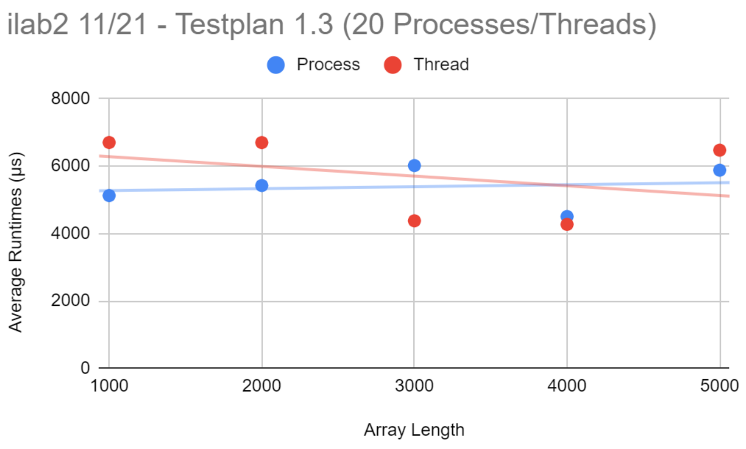
```
top - 14:08:46 up 100 days, 22 min, 25 users,  load average: 57.97, 68.30, 72.88
Tasks: 1686 total,   6 running, 1659 sleeping,   9 stopped,  10 zombie
%Cpu(s): 65.8 us,  4.2 sy,  0.0 ni, 29.9 id,  0.2 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 10560092+total, 75908256 free, 53593030+used, 44417062+buff/cache
KiB Swap: 10496081+total, 10493201+free,   288000 used. 48895536+avail Mem

   PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
  7308 an499     20   0   45.8g   2.4g 327416 R 32.8  0.2   2808:07 python
 79535 an499     20   0   45.9g   2.5g 329812 R 20.7  0.2 591:03.19 python
101880 jy486     20   0  102.5g   8.3g   1.7g R  1.8  0.8 341:55.40 python
 87077 jy486     20   0  104.1g   7.1g 408472 S  0.9  0.7  26:04.07 python
 87076 jy486     20   0  104.0g   7.0g 408472 S  0.9  0.7  26:41.50 python
187917 bw344     20   0   28.2g   2.2g 109340 S  0.8  0.2   0:06.05 python3
187735 bw344     20   0   28.2g   2.2g 109384 S  0.8  0.2   0:06.00 python3
187738 bw344     20   0   28.1g   2.2g 109480 S  0.8  0.2   0:05.91 python3
187916 bw344     20   0   28.2g   2.2g 109380 S  0.8  0.2   0:05.83 python3
187737 bw344     20   0   28.1g   2.2g 109372 S  0.8  0.2   0:05.76 python3
187736 bw344     20   0   28.2g   2.2g 109380 S  0.7  0.2   0:05.76 python3
187918 bw344     20   0   28.2g   2.2g 109372 S  0.7  0.2   0:05.88 python3
187742 bw344     20   0   28.1g   2.2g 109492 S  0.7  0.2   0:05.65 python3
187741 bw344     20   0   28.2g   2.2g 109360 S  0.7  0.2   0:05.78 python3
187919 bw344     20   0   28.1g   2.2g 109364 S  0.7  0.2   0:05.74 python3
187830 bw344     20   0   28.2g   2.2g 109488 S  0.7  0.2   0:05.57 python3
187827 bw344     20   0   28.1g   2.2g 109488 S  0.6  0.2   0:05.63 python3
```

Test 1:

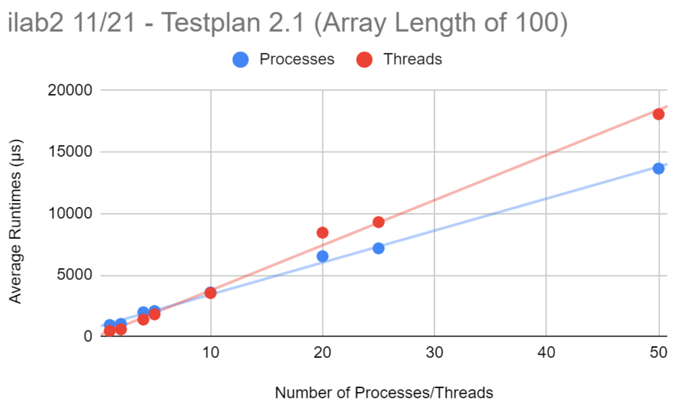


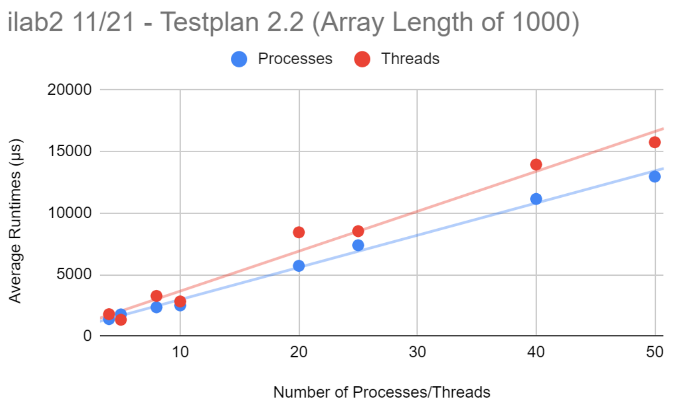


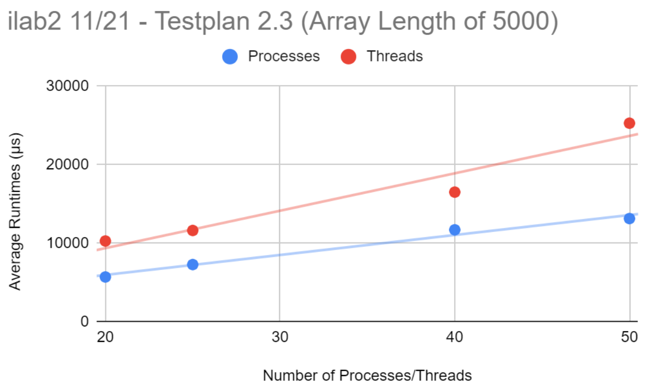


Here, we can observe a fluctuation in overall data results. Whereas test plan 1.1 demonstrated relatively reasonable results for what we would expect, test plans 1.2 and 1.3 varied wildly, at some points inverting relatively performance stats randomly, as shown by the trendline's low R^2 value. In this case, we cannot draw any accurate conclusions about the relatively performances of processes and threads from this particular trial, but what we can conclude is that the stability of performance is greatly affected by general machine load.

Test 2:

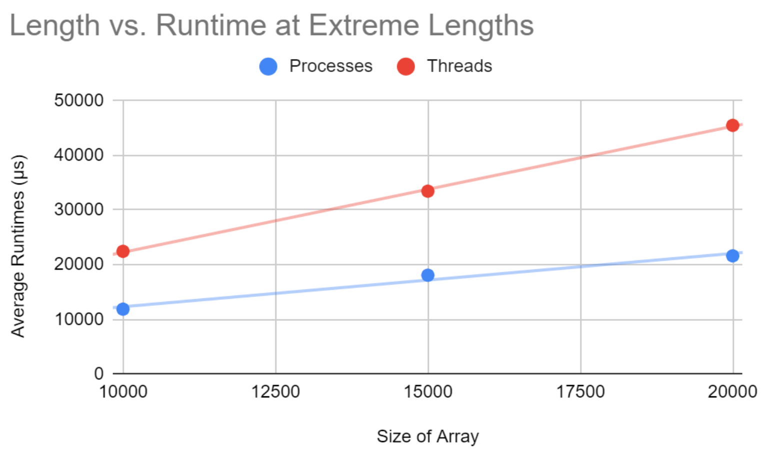






Test plan 2’s overall stats seem to indicate an inversion point for the relative performances of processes and threads, particularly where more than 7 parallel operations are being created. Our trendlines and data seem to lead to the conclusion that when more than 7 threads are being generated, processes actually begins to dominate in terms of average runtime. Of course, given the general instability of performance and poor R^2 values, we cannot conclusively determine that processes actually performs better than threads on this machine.

Test 3:



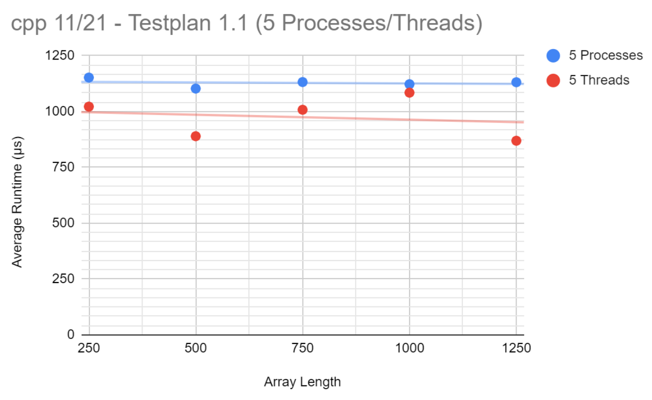
Test 3 only serves to further reinforce the same conclusions that we can draw from Test 2, in that when creating more than 7 concurrent items, it seems to be the case that processes has better overall performance, agnostic to the length of the array.

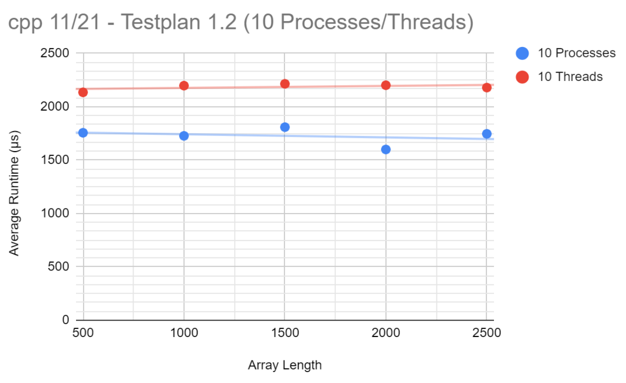
ilab2 Summary:
The data obtained from ilab2 is very different from the tests on pascal and Mac. The runtimes on this machine varied drastically from run to run due to the high amount of usage by other users. For example, in the five runs of ./thread 1000 250, in Test 2.2, the average runtimes were 853.65, 873.4, 1,177.34, 3,064.54, and 2,963.23 microseconds, averaging out to 1,786.432 microseconds. In comparison, the runtimes for processes were much more consistent, 1,303.17, 1,241.33. 1,406.94, 1,502.25, and 1,484.75 microseconds. Processes in this test averaged 1,387.688 microseconds. This was a consistent trend throughout all the tests. Threads had lower minimums and higher maximums that led to very inconsistent data. Overall there was a trend of processes performing better than threads in this test. This leads us to conclude that with higher CPU usage, processes are handled more consistently than threads making them more predictable and reliable. Threads will have better best case scenario runtimes, but worse worst case scenario runtimes. We also conclude that the specifications of this particular machine results in wildly variable performance statistics from the Thread implementation, likely due to resource usage by other users.

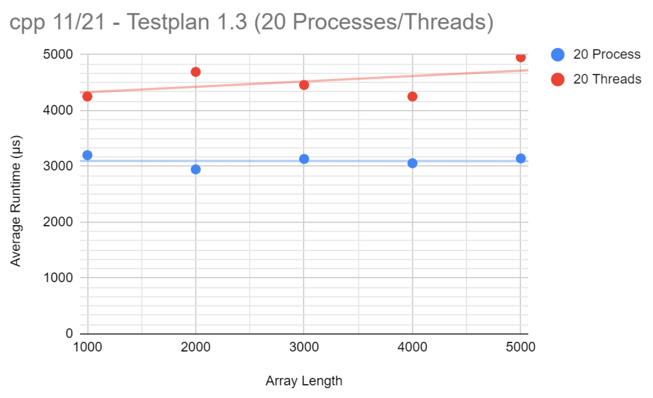
<u>cpp:</u>

The last machine we tested on was the ilab cpp. This machine had moderate usage at the time of the test.

Test 1:

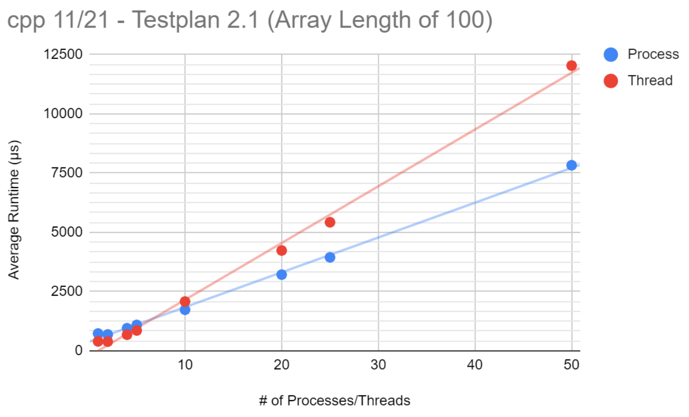


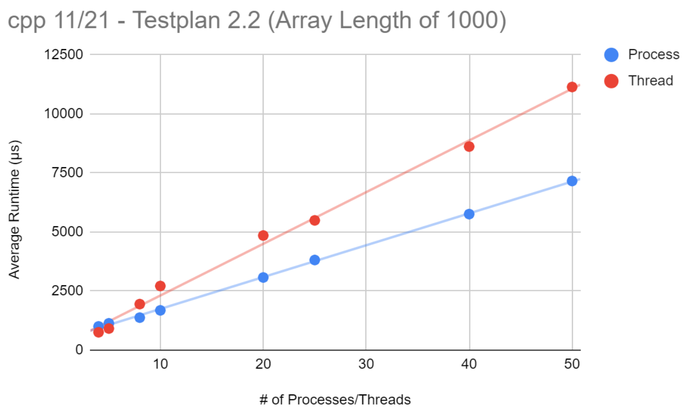


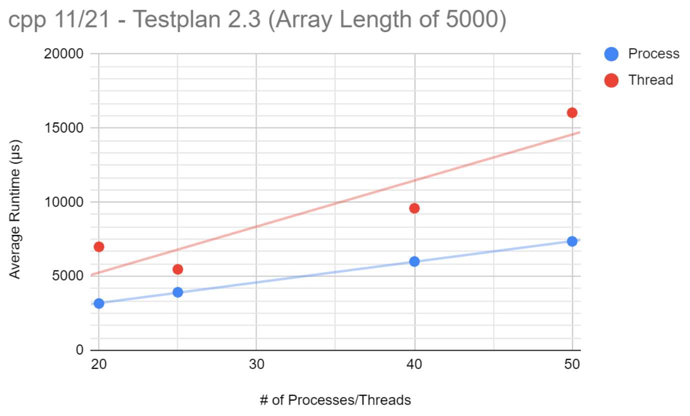


Test 1's results seem to show relative stability in the average runtime of processes and threads given a fixed interval size. This seems to indicate that the length of the array is not the cause for runtime variation.

Test 2:

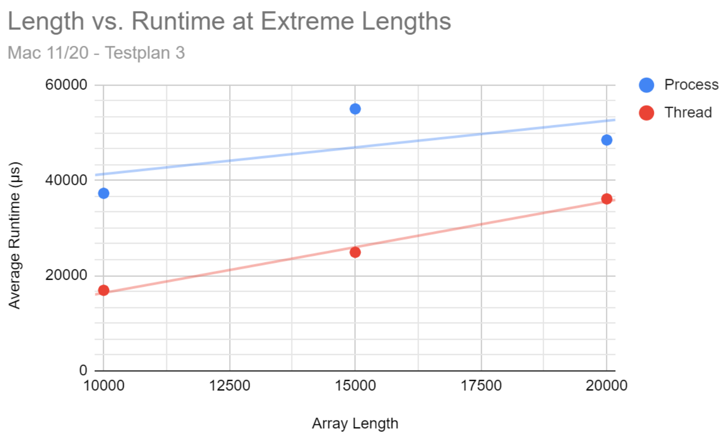






Similar to the results we observed on the ilab machine, it seems to be the case that when generating around 7 or so parallel elements, the relative performance of process and threads inverted. As before, however, the poor R^2 values of our trend line means that we cannot conclusively determine that in all cases, processes will dominate threads when generating more than 7 of each respective item.

Test 3:



We observe the same behavior as before, in that since more than 7 processes are being created to search within an array of such high magnitudes, processes seems to dominate in terms of average runtime. Strangely enough, the slopes of the trendlines seem to indicate that at exorbitantly high array lengths, threads may actually begin to dominate, although the viability of such a conclusion is dubious at best.

cpp Summary:
This machine also gave us very inconsistent results. There were other users using the machine during these tests so it is very likely that spikes in their usage caused fluctuations in our data. Infrequent spikes in usage would explain why threads were less efficient for some tests but more efficient in others, while in the two tests where there was a constant level of usage outside of the tests, threads were consistently faster.

Conclusions:

Overall threads were generally more efficient than processes given relatively low usage on the rest of the machine. In the more consistent tests, mac and pascal, it was evident that the dominant factor in runtime was the number of threads or processes with threads being faster due to the smaller cost of creating a thread as opposed to a process. This is clearly demonstrated by the linear relationship between runtime and number of processes or threads as shown in Test 2. The size of the intervals of search and the length of the array as a whole proved to have an insignificant impact on the runtime as shown by Test 1. Test 3 further supports the concept of a linear relationship between number of processes or threads and runtime.

There was no evidence to be found in these tests that processes and threads would eventually reach a point where they would have the same runtime. The tests on this size also concluded that increasing parallelism for a basic linear search of an array does not increase efficiency. On the contrary, the more you divide the search the longer it takes to complete due to the amount of time required to build a new process or thread being longer than the time to search an array with the max size of 250. A test of a linear search on a 20,000 element array took at most around 150 microseconds, which was less than any single run of our process or thread search on any array size with any number of processes or threads, indicating this method is always slower than a basic linear search, at least on this scale where we are limited by a maximum interval of search of 250 elements.

It is undeniable that the specs of the machine and level of usage have a big impact on runtimes as demonstrated by mac, ilab2, and cpp. Mac had a weaker CPU and had a more difficult time handling processes than any of the ilabs, but had the fastest handling of threads. The results of ilab2 and cpp were heavily affected by the amount of usage by others on the machine. Higher usage correlated with less consistent thread times and a trend of better overall runtimes for processes.