CS416 Project 2
Name - Saleem Khan, Chirantan Patel
Net ids - sk2304, csp126

We used the vm that was provided to us from the writeup, name - cs416f23-10

1. API Functions/Scheduler Logic

    - worker_create

        - Creates thread control block for argument “thread”, assigns thread_id, sets priority, elapsed, and status. Creates stack for thread, creates context.

        - Checks whether the run queue is NULL, which indicates the first call to the worker library.

            - If the run queue is NULL, scheduler context, run queue, mutex queue, join queue, exit list, main context, signal handler, and timer are created. Thread control blocks for thread and main thread are pushed onto the run queue.

                - Depending on if MLFQ or PSJF are defined determines what run queue data structure and scheduler algorithm is used.

            - Otherwise, the thread control block for thread is pushed onto the run queue.

        - Sets scheduler context.

    - worker_yield

        - Checks whether the thread library is in stable condition(run queue or current thread are not NULL).

    - Disables timer, and updates elapsed count if PSJF, status, and context.
    - Enqueues current thread control block onto run queue then sets scheduler context.
- worker_exit
    - Disables timer, records end time, updates statistics, adds thread id to exit list.
    - Dequeues parent from join queue if parent exists and pushes to run queue.
    - Frees any data structures relating to thread and sets scheduler context.
- worker_join
    - Gets the context of the current thread.
    - While the exit list does not contain the thread that was passed as an argument.
        - The current thread disables the timer, sets status to blocked, and enqueues its thread control block onto the join queue.
    - After exiting the while loop, a check is performed to see if the main thread has collected all child threads.
        - If so, the main thread de-allocates any data structures that are necessary for the execution of thread library functions.
- worker_mutex_init

    - Sets the value of locked to 0, and holder to NULL in mutex argument.

- worker_mutex_lock
    - Performs a check to see if mutex holder is the current thread, in which case function returns.
    - Gets context of current thread.
    - While the value locked in mutex is not 0.
        - Current thread disables timer, sets status to blocked and pushes its thread control block onto the mutex queue.
    - After exiting the loop, the value of locked is set and the mutex holder is set to the current thread's thread control block.
- worker_mutex_unlock
    - The value of locked is set to 0 and holder is set to NULL in the mutex argument that was passed.
    - Current thread dequeues thread waiting for mutex from mutex queue if thread exists it is pushed to the run queue.
- worker_mutex_destroy
    - Locked is set to 0 and holder is set to NULL in the mutex argument that is passed.
- schedule

  - Depending on if MLFQ or PSJF macro is defined determines whether sched_mlfq or sched_psjf is called.

- sched_psjf
  - Is an endless loop that checks if the thread library is in stable condition(run queue is not null and if there are threads to run).
    - Otherwise returns
  - Enqueues the current thread if it exists.
  - Dequeues a thread from the run queue, which is the thread with the least elapsed count or the first thread with 0 elapsed count.
  - Sets status of thread, records time if it is threads first time running.
  - Sets the threads context.
- sched_mlfq
  - Is an endless loop that checks if the thread library is in stable condition(run queue is not null and if there are threads to run).
    - Otherwise returns
  - Enqueues the current thread if it exists(FIFO order) to the run queue for current threads priority.
  - Enters loop for 0-7 iterations (8 priority levels, 8 queues)

        - Dequeues(FIFO) from priority queue equal to iteration count.

            - If the value returned from the priority queue is NULL, the loop continues.

        - If the priority of the thread that was dequeued is equal to the priority of the previous thread that was dequeued/run, a counter is incremented. Otherwise the previous threads priority is set to the current threads priority and the counter is set to 1.

        - Loop breaks.

    - A check is performed to see if the current threads priority is equal to the highest priority and if the dequeue priority counter is two times the number of threads that are in the READY state.

        - If so, the priority of all threads are reset

    - Current thread's context is set.

- sig_handler

    - Performs check if the thread library is in stable condition(current thread control block not NULL) if so, sets scheduler context.

    - Called when ever timer goes off, handles signal number 27

    - Updates elapsed/priority depending on PSJF/MLFQ, status, and context of current thread.

    - Enqueues current thread to run queue.

    - Sets scheduler context.

- Data Structures

    - exit_list: contains the thread_id’s of all threads that have exited.

    - mutex_queue: contains the thread control blocks of all blocked threads and reference to the mutex they are waiting for.

    - join_queue: contains the thread control blocks of all blocked parents and a thread_id of the child they are waiting on to finish.

    - run_queue: contains all the thread control blocks in READY state in FIFO order.

    - mlfq: An array of size 8, which contains 8 run queues pertaining to 8 priority levels.

2. References
    - Links to sources that I referenced on the internet:

        - https://man7.org/linux/man-pages/man0/sys_time.h.0p.html

        - https://man7.org/linux/man-pages/man2/setitimer.2.html

        - https://man7.org/linux/man-pages/man3/getcontext.3.html

        - https://man7.org/linux/man-pages/man3/makecontext.3.html

        - https://man7.org/linux/man-pages/man3/swapcontext.3.html

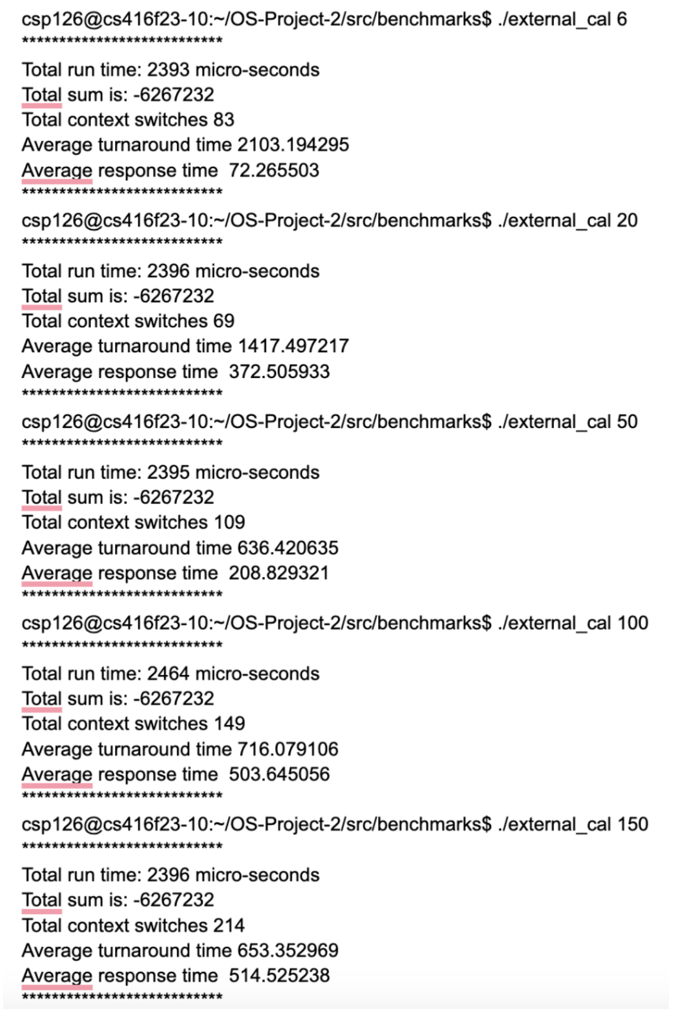
        - https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html

    - We collaborated to come up with an approach for solving the different problems that arose while we were implementing the thread library and scheduler code. We both also consulted TA's during office hours for any issues we were having with our thread library relating to our plan/logic.

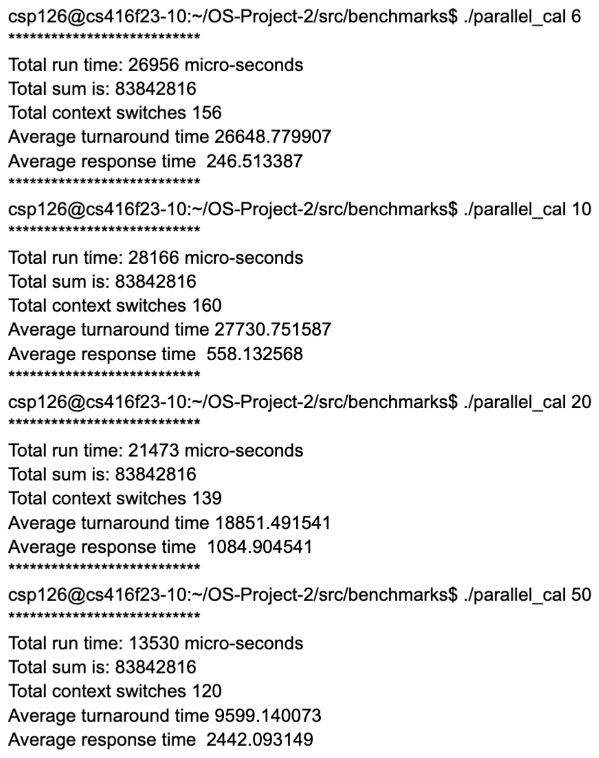
3. Benchmark Stats (Runtime)

Here are some of our stats from running all three benchmarks (external_cal, parallel_cal and vector_multiply). We have printed average turnaround time and average response time in milliseconds as mentioned in the announcement.

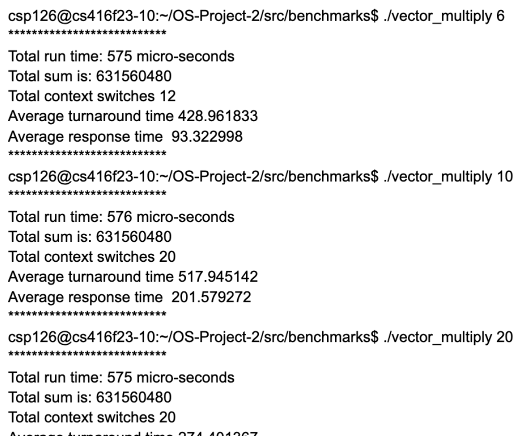
PSJF (Quantum value = 50000 micro seconds) -



```
csp126@cs416f23-10:~/OS-Project-2/src/benchmarks$ ./external_cal 6
****************************
Total run time: 2393 micro-seconds
Total sum is: -6267232
Total context switches 83
Average turnaround time 2103.194295
Average response time  72.265503
****************************
csp126@cs416f23-10:~/OS-Project-2/src/benchmarks$ ./external_cal 20
****************************
Total run time: 2396 micro-seconds
Total sum is: -6267232
Total context switches 69
Average turnaround time 1417.497217
Average response time  372.505933
****************************
csp126@cs416f23-10:~/OS-Project-2/src/benchmarks$ ./external_cal 50
****************************
Total run time: 2395 micro-seconds
Total sum is: -6267232
Total context switches 109
Average turnaround time 636.420635
Average response time  208.829321
****************************
csp126@cs416f23-10:~/OS-Project-2/src/benchmarks$ ./external_cal 100
****************************
Total run time: 2464 micro-seconds
Total sum is: -6267232
Total context switches 149
Average turnaround time 716.079106
Average response time  503.645056
****************************
csp126@cs416f23-10:~/OS-Project-2/src/benchmarks$ ./external_cal 150
****************************
Total run time: 2396 micro-seconds
Total sum is: -6267232
Total context switches 214
Average turnaround time 653.352969
Average response time  514.525238
****************************
```

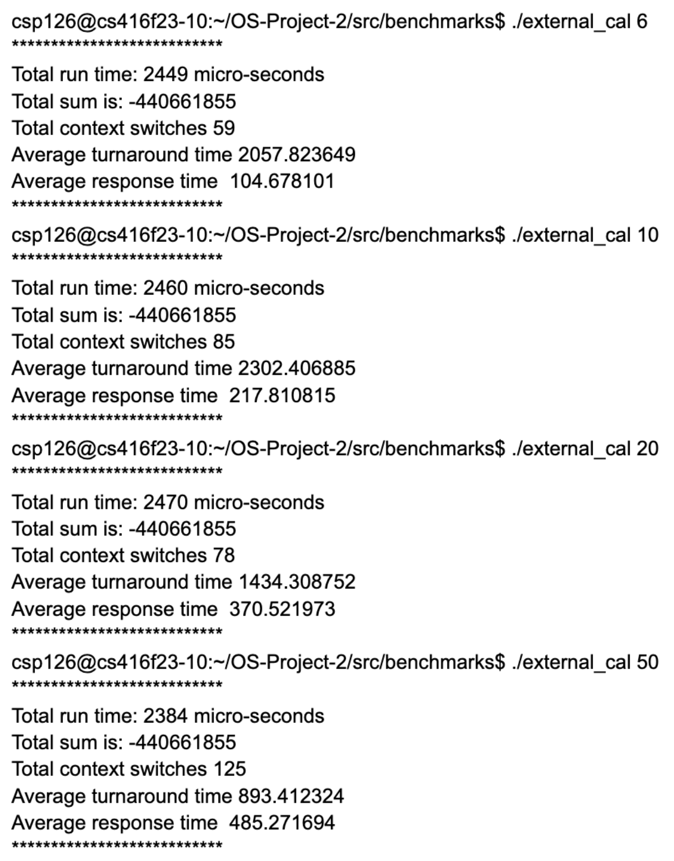


```
csp126@cs416f23-10:~/OS-Project-2/src/benchmarks$ ./parallel_cal 6
****************************
Total run time: 26956 micro-seconds
Total sum is: 83842816
Total context switches 156
Average turnaround time 26648.779907
Average response time  246.513387
****************************
csp126@cs416f23-10:~/OS-Project-2/src/benchmarks$ ./parallel_cal 10
****************************
Total run time: 28166 micro-seconds
Total sum is: 83842816
Total context switches 160
Average turnaround time 27730.751587
Average response time  558.132568
****************************
csp126@cs416f23-10:~/OS-Project-2/src/benchmarks$ ./parallel_cal 20
****************************
Total run time: 21473 micro-seconds
Total sum is: 83842816
Total context switches 139
Average turnaround time 18851.491541
Average response time  1084.904541
****************************
csp126@cs416f23-10:~/OS-Project-2/src/benchmarks$ ./parallel_cal 50
****************************
Total run time: 13530 micro-seconds
Total sum is: 83842816
Total context switches 120
Average turnaround time 9599.140073
Average response time  2442.093149
```

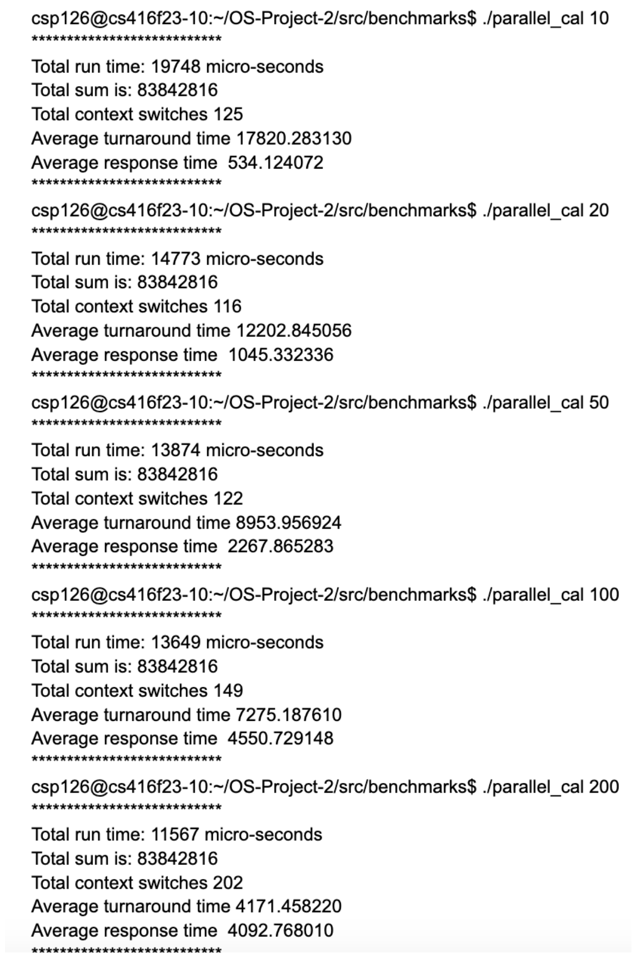


```
csp126@cs416f23-10:~/OS-Project-2/src/benchmarks$ ./vector_multiply 6
****************************
Total run time: 575 micro-seconds
Total sum is: 631560480
Total context switches 12
Average turnaround time 428.961833
Average response time  93.322998
****************************
csp126@cs416f23-10:~/OS-Project-2/src/benchmarks$ ./vector_multiply 10
****************************
Total run time: 576 micro-seconds
Total sum is: 631560480
Total context switches 20
Average turnaround time 517.945142
Average response time  201.579272
****************************
csp126@cs416f23-10:~/OS-Project-2/src/benchmarks$ ./vector_multiply 20
****************************
Total run time: 575 micro-seconds
Total sum is: 631560480
Total context switches 20
Average turnaround time 374.401367
```

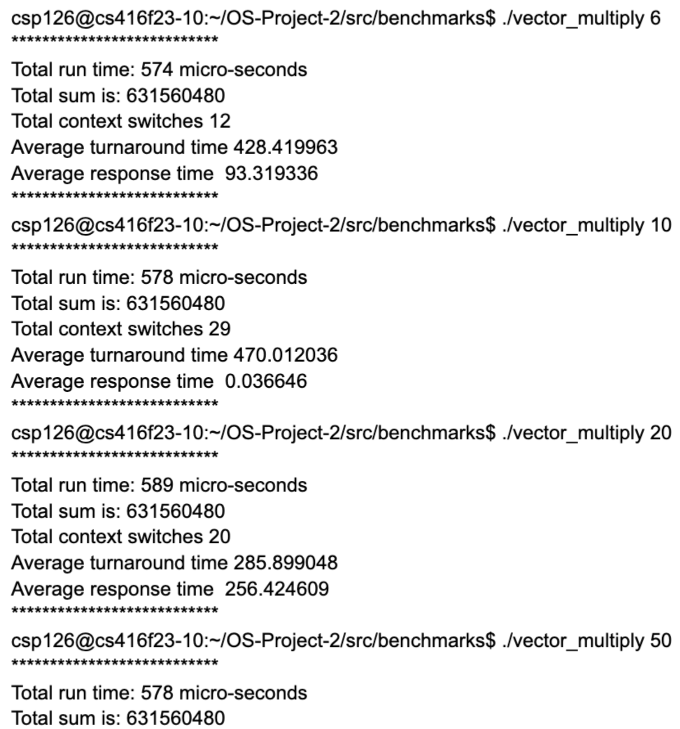
These are just some of the stats from our testing. One interesting thing is our parallel_cal's runtime was less as we run it with more and more threads. We made graphs so we can easily visualize.
MLFQ stats -



```
csp126@cs416f23-10:~/OS-Project-2/src/benchmarks$ ./external_cal 6
***************************
Total run time: 2449 micro-seconds
Total sum is: -440661855
Total context switches 59
Average turnaround time 2057.823649
Average response time  104.678101
***************************
csp126@cs416f23-10:~/OS-Project-2/src/benchmarks$ ./external_cal 10
***************************
Total run time: 2460 micro-seconds
Total sum is: -440661855
Total context switches 85
Average turnaround time 2302.406885
Average response time  217.810815
***************************
csp126@cs416f23-10:~/OS-Project-2/src/benchmarks$ ./external_cal 20
***************************
Total run time: 2470 micro-seconds
Total sum is: -440661855
Total context switches 78
Average turnaround time 1434.308752
Average response time  370.521973
***************************
csp126@cs416f23-10:~/OS-Project-2/src/benchmarks$ ./external_cal 50
***************************
Total run time: 2384 micro-seconds
Total sum is: -440661855
Total context switches 125
Average turnaround time 893.412324
Average response time  485.271694
***************************
```



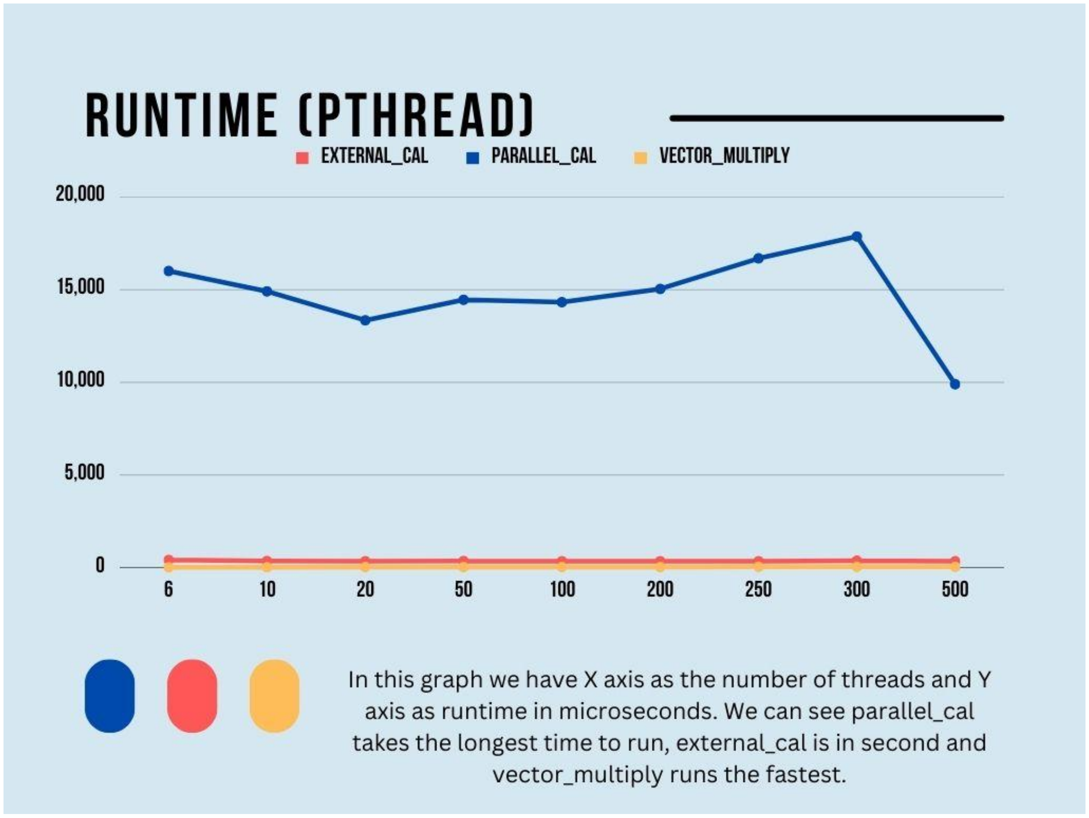
```
csp126@cs416f23-10:~/OS-Project-2/src/benchmarks$ ./parallel_cal 10
***************************
Total run time: 19748 micro-seconds
Total sum is: 83842816
Total context switches 125
Average turnaround time 17820.283130
Average response time  534.124072
***************************
csp126@cs416f23-10:~/OS-Project-2/src/benchmarks$ ./parallel_cal 20
***************************
Total run time: 14773 micro-seconds
Total sum is: 83842816
Total context switches 116
Average turnaround time 12202.845056
Average response time  1045.332336
***************************
csp126@cs416f23-10:~/OS-Project-2/src/benchmarks$ ./parallel_cal 50
***************************
Total run time: 13874 micro-seconds
Total sum is: 83842816
Total context switches 122
Average turnaround time 8953.956924
Average response time  2267.865283
***************************
csp126@cs416f23-10:~/OS-Project-2/src/benchmarks$ ./parallel_cal 100
***************************
Total run time: 13649 micro-seconds
Total sum is: 83842816
Total context switches 149
Average turnaround time 7275.187610
Average response time  4550.729148
***************************
csp126@cs416f23-10:~/OS-Project-2/src/benchmarks$ ./parallel_cal 200
***************************
Total run time: 11567 micro-seconds
Total sum is: 83842816
Total context switches 202
Average turnaround time 4171.458220
Average response time  4092.768010
***************************
```



```
csp126@cs416f23-10:~/OS-Project-2/src/benchmarks$ ./vector_multiply 6
***************************
Total run time: 574 micro-seconds
Total sum is: 631560480
Total context switches 12
Average turnaround time 428.419963
Average response time  93.319336
***************************
csp126@cs416f23-10:~/OS-Project-2/src/benchmarks$ ./vector_multiply 10
***************************
Total run time: 578 micro-seconds
Total sum is: 631560480
Total context switches 29
Average turnaround time 470.012036
Average response time  0.036646
***************************
csp126@cs416f23-10:~/OS-Project-2/src/benchmarks$ ./vector_multiply 20
***************************
Total run time: 589 micro-seconds
Total sum is: 631560480
Total context switches 20
Average turnaround time 285.899048
Average response time  256.424609
***************************
csp126@cs416f23-10:~/OS-Project-2/src/benchmarks$ ./vector_multiply 50
***************************
Total run time: 578 micro-seconds
Total sum is: 631560480
```

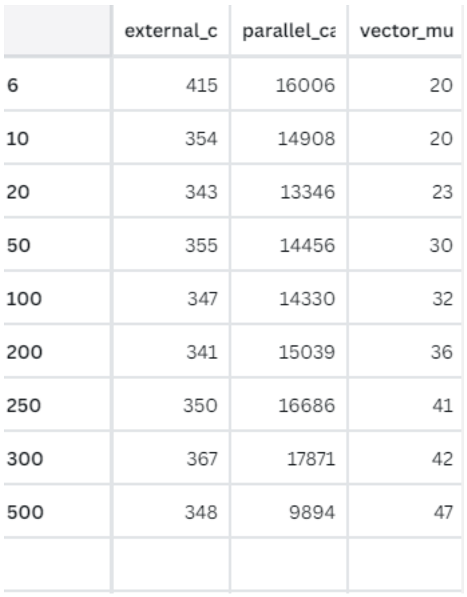
Both MLFQ and PSJF had similar performances, similar runtimes, turnaround times and response times.

4. Comparing it to pthread



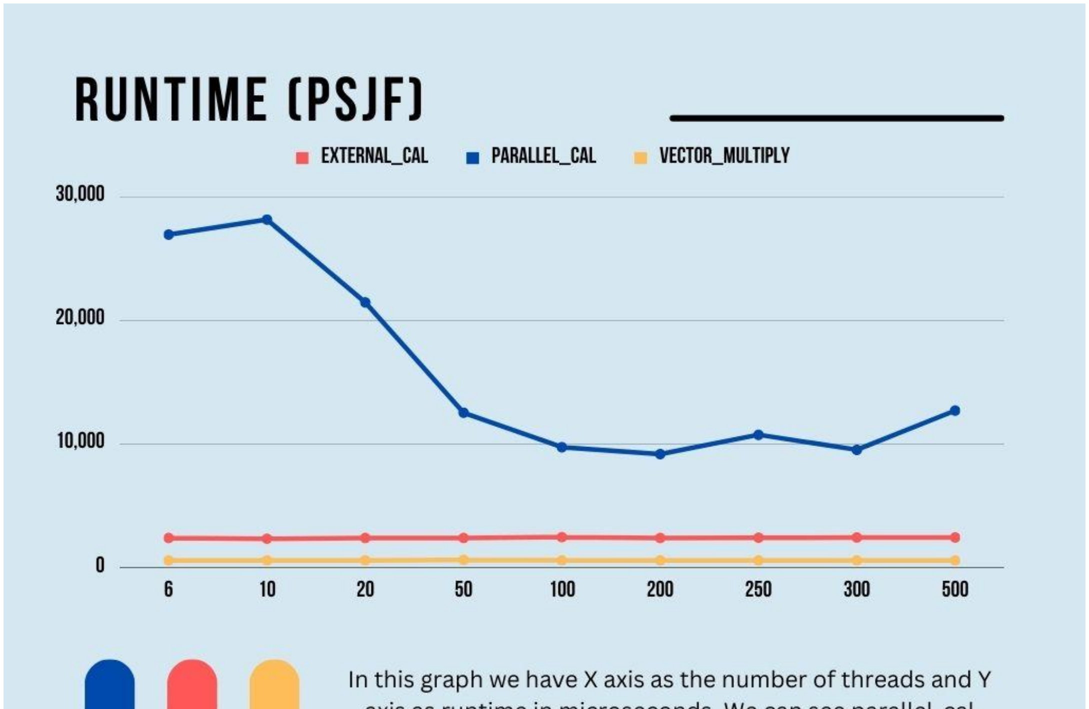
In this graph we have X axis as the number of threads and Y axis as runtime in microseconds. We can see parallel_cal takes the longest time to run, external_cal is in second and vector_multiply runs the fastest.

Table of runtime values (pthread)



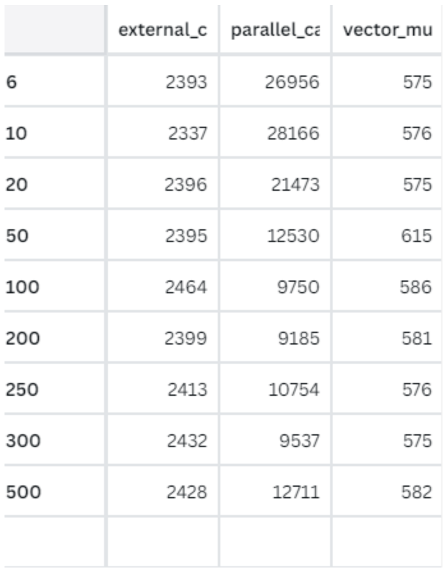
	external_c	parallel_ca	vector_mu
6	415	16006	20
10	354	14908	20
20	343	13346	23
50	355	14456	30
100	347	14330	32
200	341	15039	36
250	350	16686	41
300	367	17871	42
500	348	9894	47

Now lets take a look at our library.



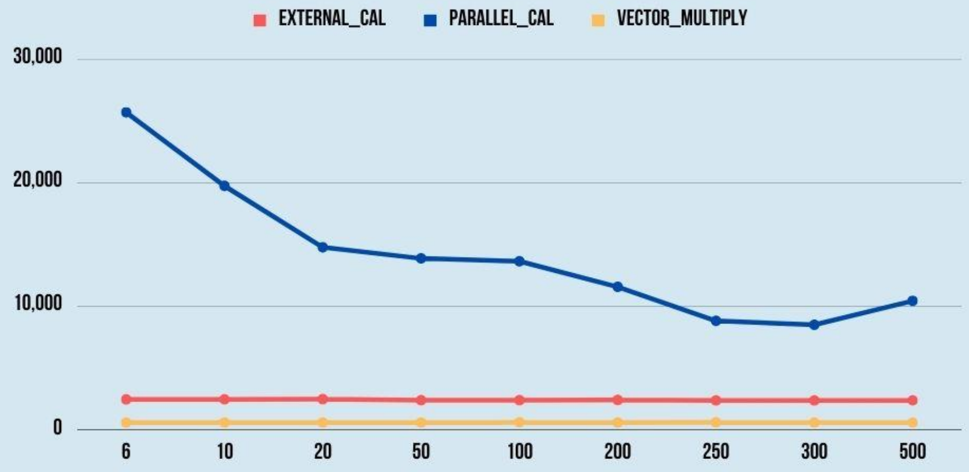
In this graph we have X axis as the number of threads and Y

PSJF stats



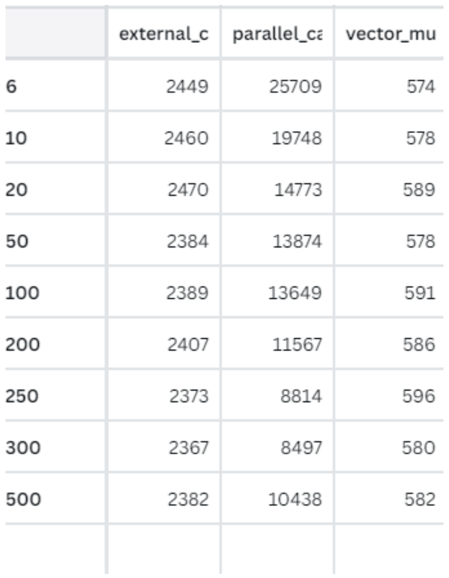
	external_c	parallel_ca	vector_mu
6	2393	26956	575
10	2337	28166	576
20	2396	21473	575
50	2395	12530	615
100	2464	9750	586
200	2399	9185	581
250	2413	10754	576
300	2432	9537	575
500	2428	12711	582

RUNTIME (MLFQ)



In this graph we have X axis as the number of threads and Y axis as runtime in microseconds. We can see parallel_cal takes the longest time to run, external_cal is in second and vector_multiply runs the fastest.

MLFQ stats



	external_c	parallel_ca	vector_mu
6	2449	25709	574
10	2460	19748	578
20	2470	14773	589
50	2384	13874	578
100	2389	13649	591
200	2407	11567	586
250	2373	8814	596
300	2367	8497	580
500	2382	10438	582

The runtime behaviors of parallel_cal vary significantly across the three- PSJF, MLFQ, and pthread. In the PSJF graph, the runtime for parallel_cal starts at a peak and decreases sharply as the number of threads increases, suggesting better performance with more threads. In the MLFQ graph, the runtime for parallel_cal has a similar initial decline but then shows a slight upward trend as the number of threads continues to increase. This shows a diminishing performance gain, and even a performance decline, at higher thread counts for MLFQ. In the pthread graph, parallel_cal's runtime remains fairly stable across the thread count, with only a slight dip in the middle. This suggests that the performance of parallel_cal using the pthread scheduling remains relatively consistent, regardless of the number of threads used. External_cal and vector_multiply stays consistent among all three. Vector runs the fastest, external_cal is volatile because of genRecord.sh, the records created are random but we can still see performance isn't too much affected.

fhrgghsh