Report-ASSIGNMENT 2 OS – (CE21BTECH11008)

System details:- Intel core I5 , 8 logical cores.

Reading The file My input file name is “inp.txt” and output file name is “out.txt” . For OutFile I have made a file pointer that is called at the end of the program and getting written using answer matrix.For input file using ifstream from fstream library and taking input in the main () function. “In.txt” contains a line. Extracting the N and K from this by simple reading and called “read matrix” functionFrom reading determining the N and K value and making “matrix” 2d vector of N x N size and storing the value of K and N using pass by reference.

To measure Time:

Used “auto start_time = std::chrono::high_resolution_clock::now();” found in time.h library . As we are intended to find the time taken by threads to execute on different set of assigned rows. So, I am starting the time for each thread when it is entering in that function and end_time when that thread is just about to call return. Taking difference to find interval. Now made one vector name “time” that is keeping track record of the time taken by each thread. It is free from race conditions because every thread has its unique index. Finally, taking the average of the time taken by each thread.

Threading Methodology

I have used Posix thread and called pthread_create function with its arguments and for parameter passing I have made struct data type. That is containing N and K and 2d vector matrix to be passed as this is the data used by all. For passing pthread_create I have made array of pthread_t of size k where the function instance will be stored. Also, I am passing ans matrix that is keeping track of the result. And last have called pthread_join function as no return value is there so passed its argument as null and thread id th[i].

for load balancing

For Chunk: - same strategy applying as guided in the question like finding ceiling(N/K) and calling as Dist and Distribution each thread this amount of row in consecutive manner. Ofcourse there is chances that Last thread will lesser number of rows, but it will not affect much.

For Mixed: - As the question suggests, I have made a 2d vector for this that will keep the record of row assigned to each thread. This is done using the main thread. I have done its implementation using modulo function. Like just travelling in each row index and assign to threa p if p = row%K (i.e. row modulo thread size.

Why chunk is not always a good approach.

see a triangular matrix. if the things will be provided in chunk to the thread, then last threads literally have very few works to do (if it is upper triangular) as thread will get 0 value and sit idle while rest of thread will have comparatively higher value. It may induce load imbalance if chunks have significantly different computational loads. But it favors locality principal of the cache thus may improve some efficiency if matrix is not biased like triangular or other thus mixed way of matrix multiplication provide better efficiency in this case. Matrices with specific patterns often arise in structured problems. For example, in signal processing or communications, matrices with zeros coming at regular interval is common in certain rows or columns representing inactive or irrelevant components thus in this kind of problem mixed way will be inefficient.

Why is mixed way of matrix multiplication also not very efficient? Even though it encounters the drawback of the chunk but it does not able to recover the best property of chunk that chunk provide locality. see for larger number of threads say like 8 to 10. Each row needed to do this multiplication will be k rows apart. But see in the cache perspective cash can load few rows due to locality of cache behavior (say taking 4 rows). But as in this methodology we using 8th row after second iteration thus most of the time lead to cache miss and thus may lead to higher time

Setting affinity: -

Approach 1(Not full proof): -

First creating the thread and then using “pthread_setaffinity_np()” function to assign the fixed core.

But problem with this is when thread has already been created and function is passed then it will start executing on different thread. At the beginning this will assign to some different thread but when “pthread_setaffinity_np()” is called then it will assign to particular desired core.

```
        pthread_create(&thread, NULL, threadFunc, (void *)threadId);
        // Set the CPU affinity for this thread
        cpu_set_t cpuset;
        CPU_ZERO(&cpuset);
        CPU_COUNT(&cpuset);
        CPU_SET(i % 8, &cpuset);

        pthread_setaffinity_np(thread, sizeof(cpu_set_t), &cpuset);
```

my_experiment: - I have tried to simulate . When it is entering in the function I have printed on which cpu it is running using “shed_getcpu()” function.In result I got that most of the time it is getting assigned to particular core but some time it is missing due to the rason I have mentioned above.

Thus this is not a full proof method.

```
void *threadFunc(void *arg)
{
    //restoring the argument that are passed
    int threadId = *(int *)arg;

    // Get the CPU core on which this thread is running
    int cpu = sched_getcpu();

    std::cout << "Thread " << threadId << " is running on CPU " << cpu
<< std::endl;

    // Do some work here
    usleep(1000); // Sleep for 1 second

    return NULL;
}
```

Approach(full proof)

This is the approach I have used. First setting the affinity part of attribute to fixed core using the function “cpu_attr_setaffinity_np()” and then creating the thread using “pthread_create” with the same attribute passing through this create function . This will ensure that thread from beginning itself is assigned to the fixed thread.

```
        pthread_t thread;
        cpu_set_t cpuset;
        CPU_ZERO(&cpuset);
        int *threadId = new int(i);

        CPU_SET(*threadId % nc, &cpuset);
        pthread_attr_t attr;
//assigning the attribute a fixed core form cpuset
        pthread_attr_setaffinity_np(&attr, sizeof(cpu_set_t),
&cpuset);

//passing that attribute when creating the thraed
        pthread_create(&thread, &attr, threadFunc, (void *)threadId);
```

Using above simulation in previous approaches I have checked multiple times that it is getting assigned to fixed core.

Determination of Cache miss

Main motivation of using affinity is to reduce the cache miss because if a thread will be bounded to a particular cpu then it can use previous loaded data in cache and cost of invalidation of data will less.

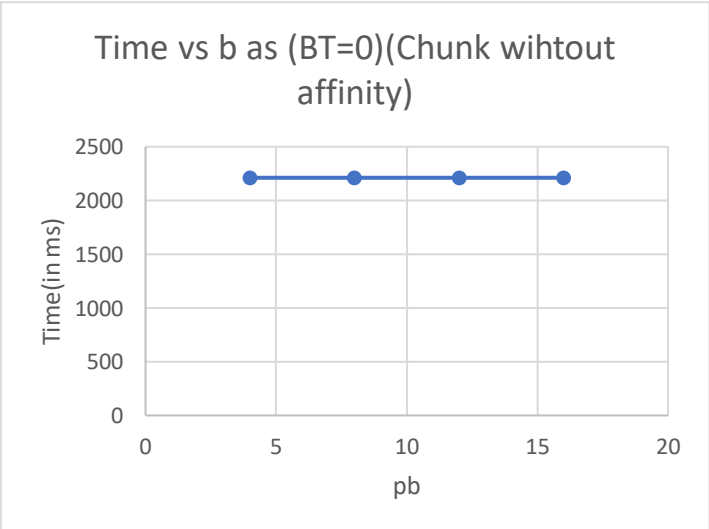
For this I have added additional linux performance tool and used perf command to

“perf stat -cache-misses, cache-references ./executable_file” this will provide the total %cache missed in this code.

Experiment 1

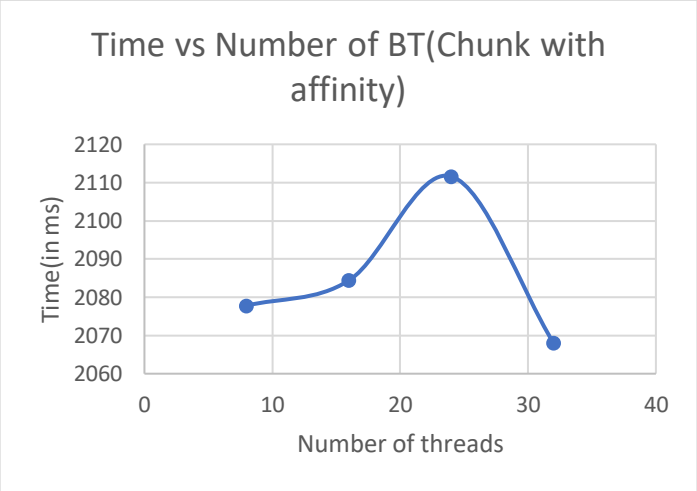
Part (1) Chunk algorithm without threads being bound to cores

			Chunk with 8 cores and 32 threads with No of Bounded threads											
			time(in ms)					average time(ms)	cache miss					Avg(in %)
	b=4	BT	obs-1	obs-2	obs-3	obs-4	obs-5		obs-1	obs-2	obs-3	obs-4	obs-5	
b	4	0	2226	2258	2178	2254	2141	2211.4	4.97	4.09	5.28	3.86	4.15	4.47
2b	8	0	2226	2258	2178	2254	2141	2211.4	4.97	4.09	5.28	3.86	4.15	4.47
3b	12	0	2226	2258	2178	2254	2141	2211.4	4.97	4.09	5.28	3.86	4.15	4.47
4b	16	0	2226	2258	2178	2254	2141	2211.4	4.97	4.09	5.28	3.86	4.15	4.47



Part(B) Chunk algorithm with a given b

			Chunk with 8 cores and 32 threads with BT number of Bounded threads											
			time(in ms)					average time(ms)	cache miss					Avg(in %)
	b=4	BT(b=4)	obs-1	obs-2	obs-3	obs-4	obs-5		obs-1	obs-2	obs-3	obs-4	obs-5	
b	4	8	2075	2092	2100	2075	2047	2077.8	3.36	3.44	2.78	3.01	3.63	3.244
2b	8	16	2112	2101	2080	2100	2029	2084.4	3.44	3.27	3.39	3.56	3.64	3.46
3b	12	24	2151	2111	2149	2062	2085	2111.6	3.52	3.61	3.55	3.54	3.42	3.528
4b	16	32	2118	2091	2071	2022	2038	2068	3.42	3.45	3.25	3.86	3.5	3.496



Individual observation

In part A

As no bounding is there time will be same.

Note: for the graph we have to draw between time vs Number of BT. But in this case BT =0. I have changed the X axis with pb value where b=K/C

In part B.

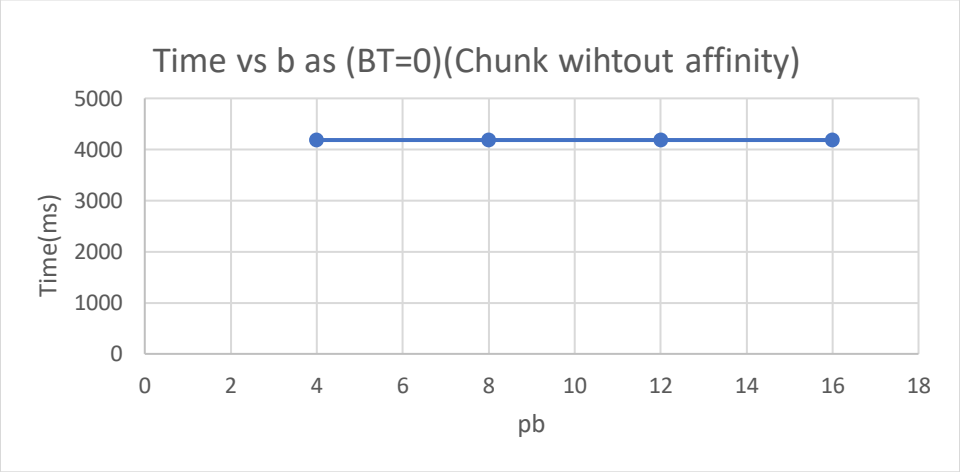
It is observed that increasing the BT will not necessarily increase performance. As can be seen when bounded thread increases 8 to 24 then time is increasing. This may be due to the increasing the bounded thread more than core may lead to condition like even one core has completed their work while other has have piling up of threads. In this case load balancing by schedular is not possible thus leading to no utilization of some core and thus time increases.

Overall Observations

Chunk with affinity set is doing better than chunk with affinity set. Average time of part B is approximately 100ms less than the average time of part A. Also, a significant reduction in cache miss can be seen when CPU affinity is applied. Thus, bounding the thread in this case is significantly improving the performance it the execution time and cache miss. Also, it is observed that execution time in not always proportional to cache miss. This may be due to fact that execution time depends on many other factor like vacancy of processor, current speed of processor, cache miss etc. Thus, cache miss is one of the factors in deciding this.

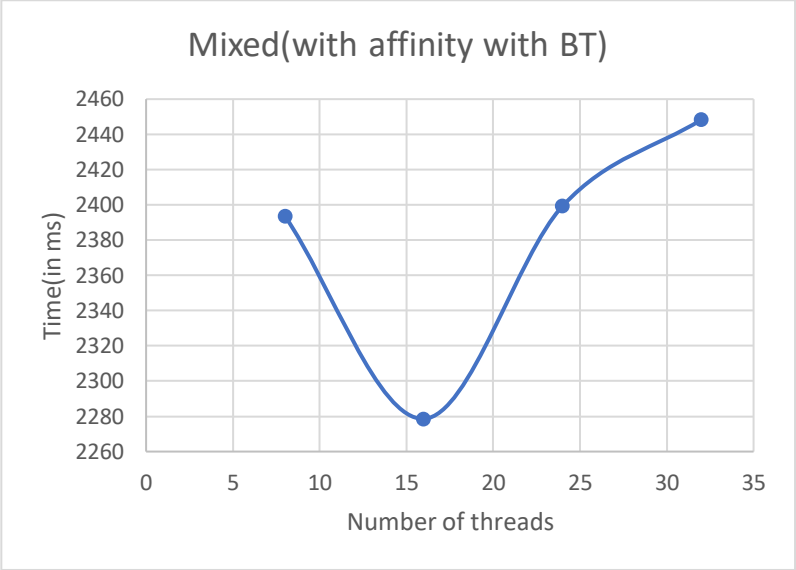
Part (3) (Mixed algorithm without threads being bound to cores)

			Mixed with 8 cores and 32 threads with No of Bounded threads											
			time(in ms)					average time(ms)	cache miss					in %
	b=4	BT	obs-1	obs-2	obs-3	obs-4	obs-5		obs-1	obs-2	obs-3	obs-4	obs-5	
b	4	0	4233	4180	4179	4137	4195	4184.8	3.14	3.4	3.23	3.87	3.07	3.342
2b	8	0	4233	4180	4179	4137	4195	4184.8	3.14	3.4	3.23	3.87	3.07	3.342
3b	12	0	4233	4180	4179	4137	4195	4184.8	3.14	3.4	3.23	3.87	3.07	3.342
4b	16	0	4233	4180	4179	4137	4195	4184.8	3.14	3.4	3.23	3.87	3.07	3.342



Part(4) Mixed algorithm without threads being bound to cores

			Mixed with 8 cores and 32 threads with BT number of Bounded threads											
			time(in ms)					average time(ms)	cache miss					Avg(in %)
	b=4	BT(b=4)	obs-1	obs-2	obs-3	obs-4	obs-5		obs-1	obs-2	obs-3	obs-4	obs-5	
b	4	8	2404	2380	2350	2422	2412	2393.6	3.36	3.32	3.23	3.33	3.39	3.326
2b	8	16	2397	2330	2309	2328	2029	2278.6	3.99	3.4	3.85	3.56	3.66	3.692
3b	12	24	2432	2394	2448	2349	2374	2399.4	3.52	3.37	3.52	3.76	3.62	3.558
4b	16	32	2463	2459	2489	2396	2435	2448.4	4.11	4.06	3.29	3.45	3.83	3.748



Individual Observation

Part C. This is the simple mixed methodology used without any bounding. Its time will be constant as BT =0 (always constant). The time taken by this method is quite high even though cash miss rate in less. It means other dependencies are playing role in determination of this time.

Part-D

It provides better timing than mixed without affinity. As the bounded thread is increasing the time first decreasing then increasing. This may be initial increase of bounded thread gives guarantee that this thread will at least be there and thus decreasing the chances to make the CPU idle as some sort of assurance is coming. Also increasing time after 16 threads can be due to overhead created by this thread on some core Cache might have less work but some process on other core will be starving as they are bonded with those cores. Thus, this may be one reason for that.

Overall comparison: - Using of affinity reasonably increasing the performance and decreasing the time by 1.5 second on my processor. Cache miss doesn't have good record in comparison to when it is unbounded.

Performance

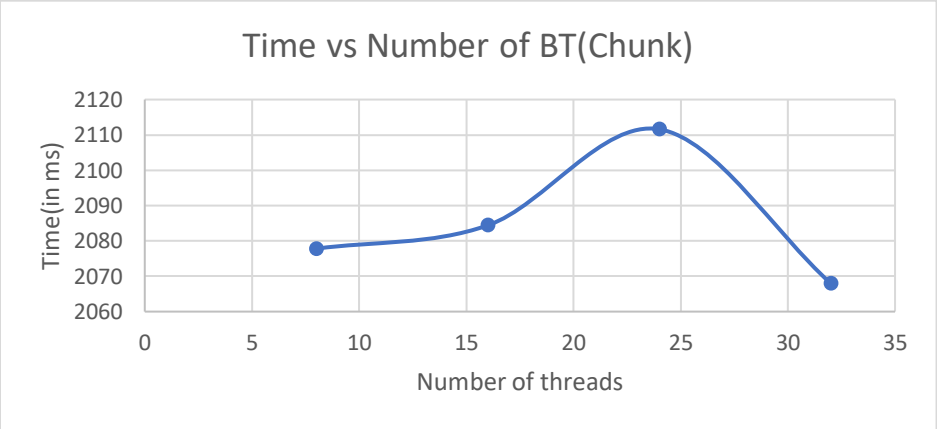
Approx conclusion can be drawn from the above experiment.

Performance of (Chunk with bounded thread) > (chunk without bounded thread) >(Mixed with bounded thread)>(Mixed with bounded thread).

Experiment 2

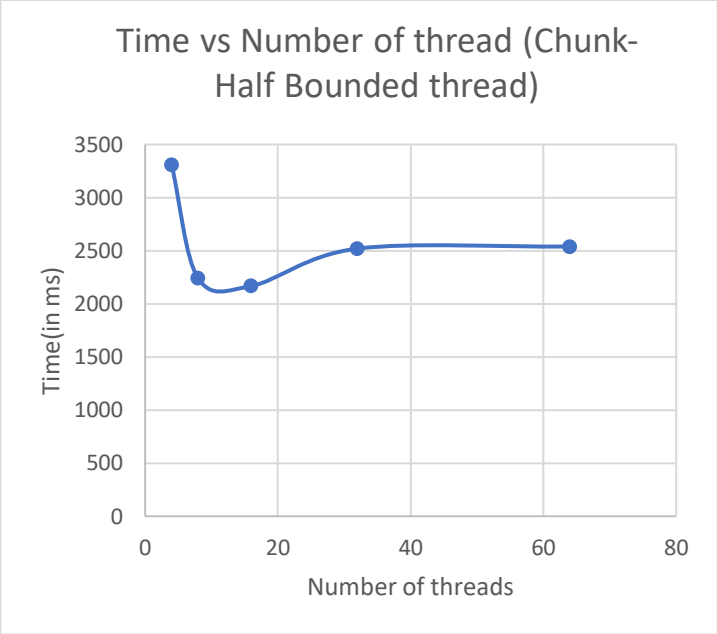
Part 5. – Average time execution without threads being bound to cores

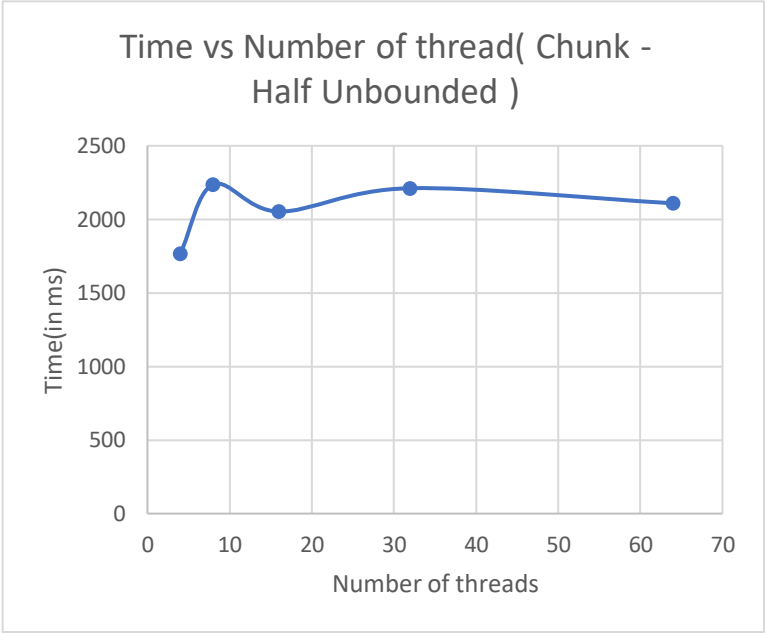
Time vs Number of threads (without any CPU affinity)



(Part 6 & Part 7) Time vs Number of threads (without any CPU affinity) -first set is bounded 2nd is set unbounded

			Time vs Number of thread(Half Chunk - Bounded & Unbounded)											
			time(in ms)					average time(ms)	cache miss					in %
Bounded	number of thread													
(BT)	2	4	3233	3307	3303	3310	3383	3307.2	4.07	3.97	4.39	3.87	3.86	3.91
(BT)	4	8	2196	2241	2314	2297	2152	2240	3.19	3.15	4.71	4.11	3.65	3.762
(BT)	8	16	2133	2181	2203	2211	2110	2167.6	3.98	3.8	3.49	3	3.57	3.568
(BT)	16	32	2350	2568	2534	2577	2569	2519.6	4.74	4.7	3.59	3.96	3.05	4.008
(BT)	32	64	2563	2533	2540	2549	2512	2539.4	4.88	3.36	3.65	3.6	4.88	4.074
Unbounded thread														
	2	4	1719	1765	1777	1758	1815	1766.8	4.07	3.97	4.39	3.87	3.86	3.91
	4	8	2197	2238	2311	2285	2150	2236.2	3.19	3.15	4.71	4.11	3.65	3.762
	8	16	2085	2109	2039	2095	1938	2053.2	3.98	3.8	3.49	3	3.57	3.568
	16	32	2273	2293	2160	2175	2158	2211.8	4.74	4.7	3.59	3.96	3.05	4.008
	32	64	2098	2186	2017	2120	2125	2109.2	4.88	3.36	3.65	3.6	4.88	4.074





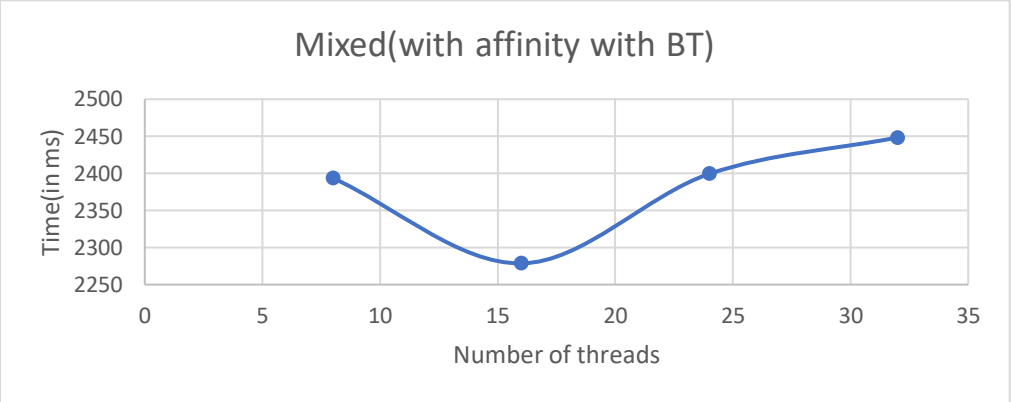
Observation --- Here when comparing between bounded and unbounded the result came is astonishing. In this case the set of threads which Is bounded is taking more than the threads which are unbounded.There is approximately 500 ms difference between their time.This may be due to internal inefficiency associated and due to overhead on scheduler. In this case if some bounded thread take more time then it will starve the thread waiting on same cpu as they can run only after this. This may lead to startvation of few threads in bounded one leading to high time than Normal threads as somhow scheduler optimises the thread at its end.

other observation – As the number of threads is increasing the is saturating. It may be due to we have created enough number of bounded and normal thread than the capacity of processor at which it can process.

Also when we compare the cache miss rate an important observation came out :

Approx – cache miss rate (all thread are bounded) cache miss rate(half bound and half normal) < cache miss rate (all thread are normal)

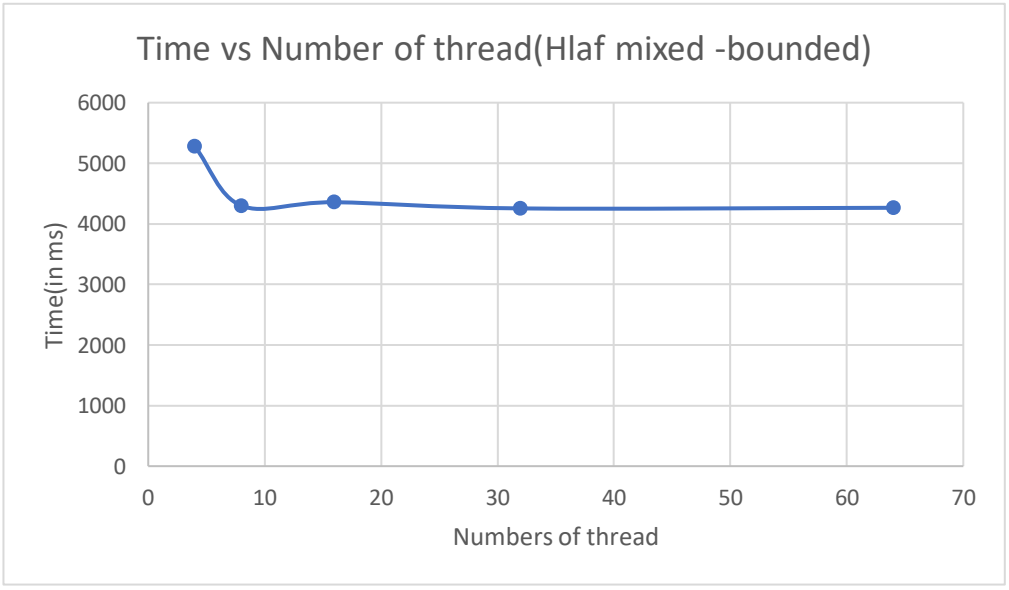
(Part 8) Withoug any bounding (as in first experiment)



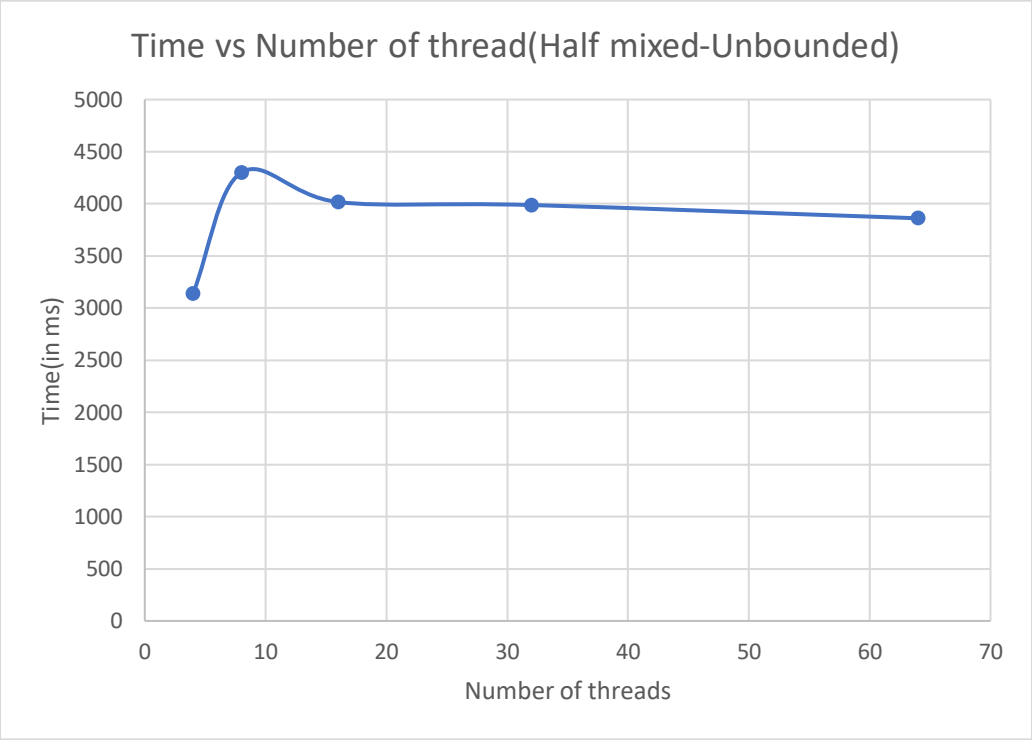
Part 9 & 10 (half thread bounded –first table and other half unbounded – second table)

			Time vs Number of thread(Half Mixed - Bounded & Unbounded)											
			time(in ms)					average time(ms)	cache miss					Avg in %
(BT)	2	4	5300	5288	5314	5224	5262	5277.6	4.8	3.88	5	3.78	4.12	4.316
(BT)	4	8	4309	4323	4289	4281	4300	4300.4	3.19	3.59	3.12	3.25	3.27	3.284
(BT)	8	16	4291	4258	4611	4324	4299	4356.6	4.07	3.06	9.64	4.16	3.24	4.834
(BT)	26	32	4277	4266	4257	4257	4216	4254.6	3.3	3.44	3.48	3.33	3.97	3.504
(BT)	32	64	4212	4314	4226	4365	4208	4265	4.7	4.18	5	3.29	4.28	4.29
	2	4	3162	3169	3163	3098	3120	3142.4	4.8	3.88	5	3.78	4.12	4.316
	4	8	4309	4320	4290	4282	4300	4300.2	3.19	3.59	3.12	3.25	3.27	3.284
	8	16	3915	3937	4054	3996	4185	4017.4	4.07	3.06	9.64	4.16	3.24	4.834
	16	32	3906	4087	3986	4031	3933	3988.6	3.3	3.44	3.48	3.33	3.97	3.504
	32	64	3758	3968	4026	3736	3819	3861.4	4.7	4.18	5	3.29	4.28	4.29

Part 9 (graph)



Part 10 (graph)



Observation:-- If we compare the average time of threads here. Normal threads are showing better performance than the bounded threads. Here approximately 300 ms difference came between those. This may ne due to the way in which mixed is calculated. Unlike chunk a core in mixed calculate on the different arrays. Thus bounding will now lead to more conflicts.

Suppose a thread is running on given cpu core with row k so due to data locality it took the data of k+1 row. If you bound this thread then it have to run on the same core but it will most probably have to execute other row far away than K or K+1. Thus definitely cache miss will be there. But suppose we have not fixed the thread then this thread can go to other core and some other thread can come to this core that will have high probability than previous thread that it will run on K+1 row. This may be the reason why setting the affinity is not helping.

This method is also showing the phenomenon of the saturation when the number of thread is increasing. We can give similar argument that we have given in observation of chunk part.

By doing this experiment I have also observed some unique things that may lead to the data invalidation---

1.when I reboot my PC same algorithm started showing different timing than what It was taking before rebooting.

2.Also when laptop was in battery saver mode it was reducing its processor speed and thus changing the observed time significantly.

Conclusion: --In this experiment I have tried to take every possible aspect that caused this time but at many points I was unable to make correlation between time and cache miss. Also, It is good to measure the interval duration at same time if possible other wise there very high chances that the processor condition change and the process will not give the actual time what it should as scheduler will schedule the things accordingly on number of process available.