Operating Systems-2: Spring 2024
Programming Assignment 2 : Thread Affinity

Waghmare Aditya Abhaykumar
CS22BTECH11061

February 19, 2024

I) Coding Approach

Program uses `Chunk and Mixed` methods to perform Parallel Matrix Multiplication with K `threads` for squaring matrix A. The Result, Total Time taken, Average Normal Thread Execution Time and Average Core Bound Thread Execution Time for each method is written back to respective output file. The first BT `threads` are assigned(have their Affinity Set) to cores in chunks of size b per core, where `b = K/C`. The low level design of program is explained below -

Main Function

`Main function` reads the input from the `inp.txt` file and stores the value in global variables. It also initializes the global variables `int **a` (stores the input array) and `int **result` (stores the result of Matrix Multiplication) to arrays of size N*N with `void prepare_2d_arrays();` function as well as `int64_t *thread_exec_time` (stores execution time of each thread) to array of size K(number of threads).

Once the initialization is done, main function prepares output files and runs the `Chunk and Mixed` methods of Parallel Matrix Multiplication one after the other in `main function` itself. This methods store their result in `int **result` matrix after which `Main function` writes the result, total time taken, average normal and core bound thread times to respective output file for each method.

Common functions

Listing 1: Single Element Multiplication

```
//Gives single element of Square Matrix
void matrix_mult(int row, int col){
    int r = 0;
    for(int i = 0; i < n; i++) {
        r += a[row][i] * a[i][col];
    }
    result[row][col] = r;
}
```

Listing 2: Matrix Row Multiplication

```
//Gives Row of the Square Matrix
void matrix_row_mult(int row){
    for(int col = 0; col < n; col++) {
        matrix_mult(row, col);
    }
}
```

a) **Chunk Method**

The job of Matrix Multiplication is divided among K `threads`. Thread routine is the `void chunk(int id);` function. The argument `int id` defines what rows are assigned to `thread` number `id`. Thread number `id` gets rows `id*ceil(N/K)+ 0` to `id*ceil(N/K)+ ceil(N/K)-1` assigned to it. Furthermore First /cppinlineBT number of threads have their affinity set to cores in batches of size `b` per core. `, K/Cb =` max so that even when `K < C` we set the affinity for first `BT` theads to some cores(K cores, which will be less than C). We also Measure the execution time of each thread and store it to the shared array `int64_t thread_exec_time[k]`.

Listing 3: Chunk Method Thread Routine

```
//Chunk Method Thread Routine
void chunk(int id){
    //Measuring Thread Execution time

    //record start time of thread execution
    auto start = chrono::high_resolution_clock::now();

    //Checking if this threads affinity should be set or not
    if(id < bt && b!=0){
        //first BT threads are assigned to certain cores

        //to what core this thread associates
        int core_id = id/b;//b threads for each core

        // Set thread affinity on Linux (pthread_setaffinity_np)
        cpu_set_t cpuset;
        CPU_ZERO(&cpuset);
        CPU_SET(core_id, &cpuset);
        pthread_setaffinity_np(pthread_self(), sizeof(cpuset), &cpuset);
    }

    //Matrix Multiplication
    for(int i = 0; i < p; i++) {
        matrix_row_mult(id * p + i);
    }

    //record end time
    auto end = chrono::high_resolution_clock::now();
    //get execution time
    auto time = chrono::duration_cast<chrono::microseconds>(end - start).count();

    //Save Thread Execution time in common memory array
    thread_exec_time[id] = time;
}
```

Listing 4: Chunk based Parallel Matrix Multiplication

```
//Chunks

//record start time
auto start = chrono::high_resolution_clock::now();

// execute threads
for(int id = 0; id < k; id++){
    threads[id] = thread(chunk, id);
```

```cpp
    }

    // Join threads (wait for them to finish)
    for(int id = 0; id < k; id++){
        threads[id].join();
    }
    //record end time
    auto end = chrono::high_resolution_clock::now();
    //get execution time
    auto time = chrono::duration_cast<chrono::microseconds>(end - start).count();



    out_file << "Total Time: "<< time << endl;
    out_file << "Average Core Bound Thread time: "<< get_avg_cbt_time() << endl;
    out_file << "Average Normal Thread time: "<< get_avg_nt_time() << "\n\n";

    //Print Result Array
    for(int row = 0; row < n; row++){
        for (int col = 0; col < n; col++) {
            out_file << setw(15) << result[row][col] << ' ';
            // out_file << result[row][col] << ' ';
        }
        out_file << endl;
    }
```

b) Mixed Method

The job of Matrix Multiplication is divided among K `threads`. Thread routine is the `void mixed(int id);` function. The argument `int id` defines what rows are assigned to `thread` number id. Thread number id gets rows `id + 0*K, id + 1*K, ..., id + (ceil(N/K)-1)*K` assigned to it. Furthermore First /cppinlineBT number of threads have their affinity set to cores in batches of size `b` per core. `, K/Cb = max1` so that even when `K < C` we set the affinity for first `BT` theads to some cores(K cores, which will be less than C). We also Measure the execution time of each thread and store it to the shared array `int64_t thread_exec_time[k]`.

Listing 5: Mixed Method Thread Routine

```cpp
    //Mixed Method Thread Routine
    void mixed(int id){
        //Measuring Thread Execution time

        //record start time
        auto start = chrono::high_resolution_clock::now();

        //Checking if this threads affinity should be set or not
        if(id < bt && b!=0){
            //first BT threads are assigned to certain cores

            int core_id = id/b;//b threads for each core

            // Set thread affinity on Linux (pthread_setaffinity_np)
            cpu_set_t cpuset;
            CPU_ZERO(&cpuset);
            CPU_SET(core_id, &cpuset);
```

```
        pthread_setaffinity_np(pthread_self(), sizeof(cpuset), &cpuset);
    }

    //Matrix Multiplication
    for(int i = 0; i < p; i++){
        matrix_row_mult(id + i * k);
    }

    //record end time
    auto end = chrono::high_resolution_clock::now();
    //get execution time
    auto time = chrono::duration_cast<chrono::microseconds>(end - start).count();

    //Save Thread Execution time in common memory array
    thread_exec_time[id] = time;
}
```

Listing 6: Mixed Parallel Matrix Multiplication

```
//Mixed

//record start time
start = chrono::high_resolution_clock::now();

// execute threads
for(int id = 0; id < k; id++){
    threads[id] = thread(mixed, id);
}

// Join threads (wait for them to finish)
for(int id = 0; id < k; id++){
    threads[id].join();
}
//record end time
end = chrono::high_resolution_clock::now();
//get execution time
time = chrono::duration_cast<chrono::microseconds>(end - start).count();



out_file << "Total Time: "<< time << endl;
out_file << "Average Core Bound Thread time: "<< get_avg_cbt_time() << endl;
out_file << "Average Normal Thread time: "<< get_avg_nt_time() << "\n\n";

//Print Result Array
for(int row = 0; row < n; row++){
    for (int col = 0; col < n; col++) {
        // out_file << result[row][col] << ' ';
        out_file << setw(15) << result[row][col] << ' ';
    }
    out_file << endl;
}
```

Setting Thread Affinity

For a given `BT`, first `BT` threads are divided into batches of size b, where `b = max{1, K/C}`. Every batch of size b is assigned to a single distinct core. The remaining `K - BT` threads are handled by the OS. `b = max{1, K/C}` so that even when `K < C` we set the affinity for first `BT` theads to some cores(`BT` cores, which will be less than C).

Listing 7: b

```
    b = k>=c?k/c:1;// b = if K>=C then floor(K/C) Else it is 1
```

Listing 8: Thread Affinity Setting Logic

```
    //Checking if this threads affinity should be set or not
    if(id < bt && b!=0){
        //first BT threads are assigned to certain cores

        int core_id = id/b;//b threads for each core

        // Set thread affinity on Linux (pthread_setaffinity_np)
        cpu_set_t cpuset;
        CPU_ZERO(&cpuset);
        CPU_SET(core_id, &cpuset);
        pthread_setaffinity_np(pthread_self(), sizeof(cpuset), &cpuset);
    }
```

Average Thread Execution Time

For Each Thread we store its Execution time in a shared memory array `int64_t thread_exec_time[k]`.

1. Normal Threads

We use function `int64_t get_avg_nt_time();` to get Average Normal Thread Execution time.

Listing 9: Average Normal Thread Execution time

```
    //Returns Average Normal Thread Execution time
    int64_t get_avg_nt_time(){
        int64_t avg_nt_time = 0;
        int nt_count = 0;//number of normal threads

        for(int id=0; id<k; id++){//for each thread
            if(id < bt && b!=0){
            }
            else{//if it is normal thread
                avg_nt_time += thread_exec_time[id];
                nt_count++;
            }
        }
        nt_count>0? avg_nt_time /= nt_count:0;//If Normal Thread count is
            non zero, only then we can divide to get avg time, otherwise it
            is zero already

        return avg_nt_time;
    }
```

2. Core Bound Threads

We use function `int64_t get_avg_cbt_time();` to get Average Core Bound Thread Execution time.

Listing 10: Average Core Bound Thread Execution Time

```
//Returns Average Core Bound Thread Execution Time
int64_t get_avg_cbt_time(){
    int64_t avg_cbt_time = 0;
    int cbt_count = 0;
    for(int id=0; id<k; id++){
        if(id < bt && b!=0){
            avg_cbt_time += thread_exec_time[id];
            cbt_count++;
        }
    }
    cbt_count>0? avg_cbt_time /= cbt_count:0;//If Core Bound Thread
        count is non zero, only then we can divide to get avg time,
        otherwise it is zero already

    return avg_cbt_time;
}
```

II) Experiments

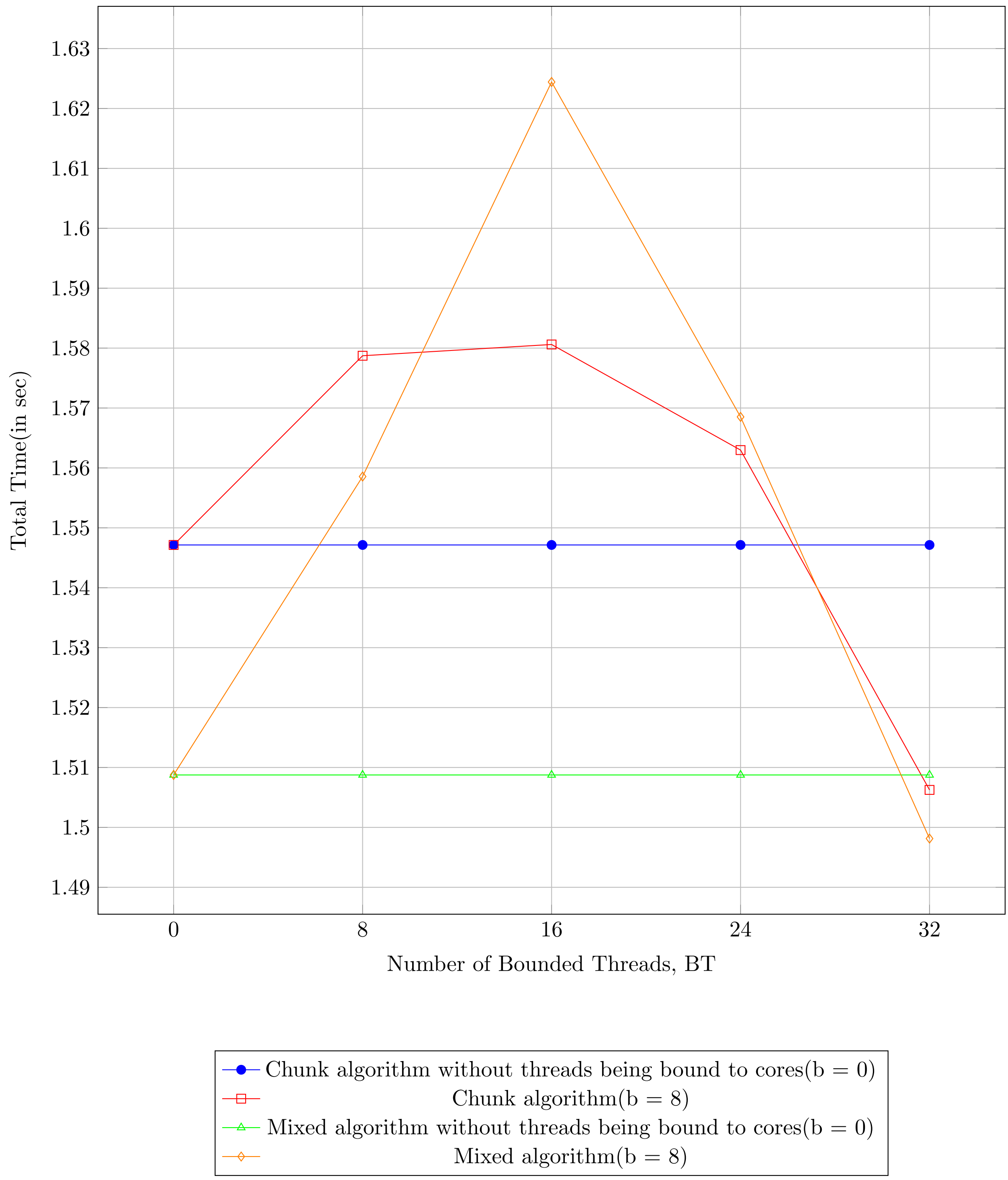
Test System Specifications:


```
aditya@aditya-IdeaPad-3-15ITL6:~$ lscpu
Architecture:            x86_64
  CPU op-mode(s):        32-bit, 64-bit
  Address sizes:         39 bits physical, 48 bits virtual
  Byte Order:            Little Endian
CPU(s):                  4
  On-line CPU(s) list:   0-3
Vendor ID:               GenuineIntel
  Model name:            11th Gen Intel(R) Core(TM) i3-1115G4 @ 3.00GHz
    CPU family:          6
    Model:               140
    Thread(s) per core:  2
    Core(s) per socket:  2
    Socket(s):           1
    Stepping:            1
    CPU max MHz:         4100.0000
    CPU min MHz:         400.0000
    BogoMIPS:            5990.40
    Flags:               fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bt
                         s rep_good nopl xtopology nonstop_tsc cpuid aperfmperf tsc_known_freq pni pclmulqdq dtes64 monitor ds_cpl vmx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt
                          tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb cat_l2 invpcid_single cdp_l2 ssbd ibrs ibpb stibp ibrs_enhanced tpr_shadow flexpriority ept vpid e
                         pt_ad fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid rdt_a avx512f avx512dq rdseed adx smap avx512ifma clflushopt clwb intel_pt avx512cd sha_ni avx512bw avx512vl xsaveopt xsavec xg
                         etbv1 xsaves split_lock_detect dtherm ida arat pln pts hwp hwp_notify hwp_act_window hwp_epp hwp_pkg_req vnmi avx512vbmi umip pku ospke avx512_vbmi2 gfni vaes vpclmulqdq avx512_vnni avx5
                         12_bitalg avx512_vpopcntdq rdpid movdiri movdir64b fsrm avx512_vp2intersect md_clear ibt flush_l1d arch_capabilities
Virtualization features:
  Virtualization:        VT-x
Caches (sum of all):
  L1d:                   96 KiB (2 instances)
  L1i:                   64 KiB (2 instances)
  L2:                    2.5 MiB (2 instances)
  L3:                    6 MiB (1 instance)
NUMA:
  NUMA node(s):          1
  NUMA node0 CPU(s):     0-3
Vulnerabilities:
  Gather data sampling:  Mitigation; Microcode
  Itlb multihit:         Not affected
  L1tf:                  Not affected
  Mds:                   Not affected
  Meltdown:              Not affected
  Mmio stale data:       Not affected
  Retbleed:              Not affected
  Spec rstack overflow:  Not affected
  Spec store bypass:     Mitigation; Speculative Store Bypass disabled via prctl
  Spectre v1:            Mitigation; usercopy/swapgs barriers and __user pointer sanitization
  Spectre v2:            Mitigation; Enhanced / Automatic IBRS, IBPB conditional, RSB filling, PBRSB-eIBRS SW sequence
  Srbds:                 Not affected
  Tsx async abort:       Not affected
```

Figure 1: Test System Specifications

Experiment 1: Total Time vs Number of Bounded Threads, BT:

```
 N = 1024
K = 32
C = 4
b = K/C = 32/4 = 8
```



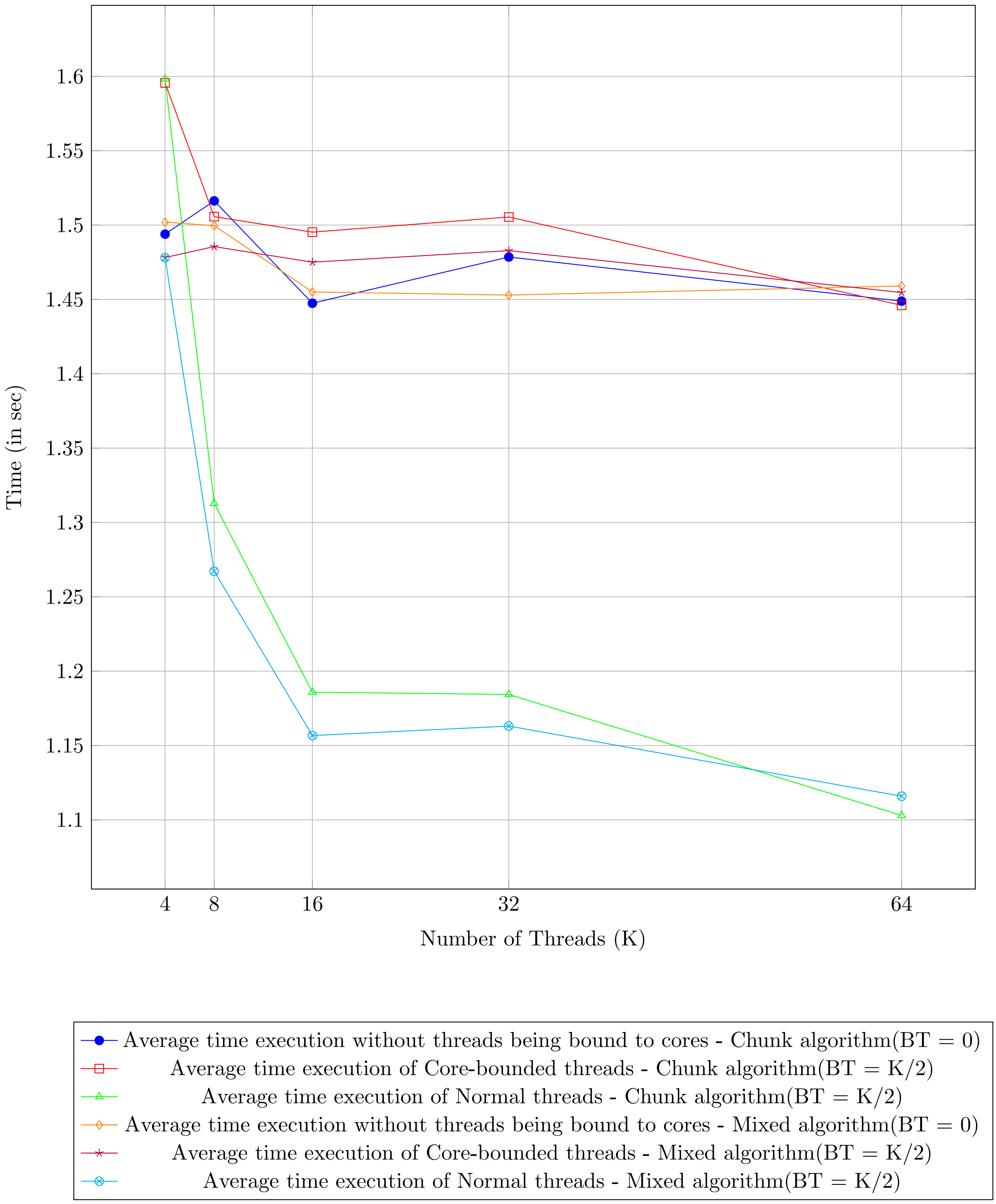
Observations:

- For this program-system combination, load balancing by the OS seems to perform better.
- When Thread Affinity is set it takes more time to complete execution.
- For the test system, Threads Execute faster by utilizing all the cores rather than using only a certain core throughout the execution for cache benefits.
- As we can see, when `BT = 32` the Execution time drops closer to Execution time when OS handles Scheduling. This is because `BT = 32` Equally distributes the threads like load balancing does.

Experiment 2: Time vs Number of threads:

In this Experiment, we can use the same approach of setting thread affinity as in experiment 1. We need to Distribute `BT = K/2` threads Equally to `C/2` cores. Hence, each core gets `b = BT/(C/2)= (K/2)/(C/2)= K/C`, which is same as in experiment 1. So, we can use the same code as in experiment 1.

```
 N = 1024
C = 4
BT = K/2
```



Observations:

- As K increases, each thread gets less work load. So with increasing K, Average thread execution time decreases for all kinds of threads and Average Execution time also decreases.
- For the given system, Core Bound Threads on Average Take more time to execute. They take more time than Average time of execution without threads being bound to cores.
- Normal Threads take significantly less Average time than Core Bound threads. They also take significantly less time than Average time of execution without threads being bound to cores.
- For the test system, Load balancing seems to speed up the thread execution more than setting thread affinity.