Assign_2Report_ES22BTECH11026

23 February 2024 13:46

SPECIFICATIONS:

Before explaining the whole code, I would like to list down the **specifications of the device** on which I completed the assignment:

CPU:

12th Gen Intel(R) Core(TM) i5-1235U

Cores: 10
Logical processors: 12
Virtualization: Enabled

The writing and execution part of the code was done on **VM Virtual Box on Linux OS**.

Since my laptop has 12 logical processors, I altered the value of K(no. of threads) for both the experiments(as instructed by the TA).

CHANGES MADE:

Experiment 1:
K = 48, b ---> K/C = 4

Experiment 2:
I calculated the average time execution for the different threads for the following values of K(as instructed by the TA for students with 12 cores):

K = 6, 12, 24, 48, 96

EXPLANATION:

Here is a snapshot of the **runner function for the Chunk Technique** implemented as in Assignment 1 with the added explanation of how the threads have been assigned to the respective CPU cores:

```
54  //mixed technique to compute the C matrix
55  void* chunk_technique(void* arg){
56      intermediary* inner = (intermediary*)arg;
57      int p = N/K;
58      if(inner->thread_no < BT){
59          auto start = chrono::steady_clock::now();
60          cpu_set_t mask;
61          CPU_ZERO(&mask);
62          int cpu_no = (inner->thread_no)*C/K;
63          CPU_SET(cpu_no, &mask);
64
65          pthread_setaffinity_np(pthread_self(), sizeof(cpu_set_t), &mask);
66
67          for(int i = (inner->thread_no)*p; i < (inner->thread_no+1)*p; i++){
68              for(int j = 0; j < N; j++){
69                  global_array[i][j] = compute_Cij(inner->input_array, i, j);
70              }
71          }
72          auto end = chrono::steady_clock::now();
73          std::chrono::duration<double> duration = end - start;
74          double iteration_time = duration.count();
75          time_array[inner->thread_no] = iteration_time;
76      }
77      else{
78          auto start = chrono::steady_clock::now();
79          if(inner->thread_no < K-1){
80              for(int i = (inner->thread_no)*p; i < (inner->thread_no+1)*p; i++){
81                  for(int j = 0; j < N; j++){
82                      global_array[i][j] = compute_Cij(inner->input_array, i, j);
83                  }
84              }
85          }
86          else{
87              for(int i = (inner->thread_no)*p; i < N; i++){
88                  for(int j = 0; j < N; j++){
89                      global_array[i][j] = compute_Cij(inner->input_array, i, j);
90                  }
91              }
92          }
93          auto end = chrono::steady_clock::now();
94          std::chrono::duration<double> duration = end - start;
95          double iteration_time = duration.count();
96          time_array[inner->thread_no] = iteration_time;
97      }
98      return NULL;
99  }
```

CASE 1: BT != 0

All the bounded threads enter the **'if condition'** on line 58.
I start the timer to calculate the time execution for a single bounded thread and later end it on line 72. I then store it in a **time_array.**

The time array is as follows:

```
double time_array[96];
```

It's elements contains the **times taken by each of the threads** during their execution of the rows of the square matrix.

SIMILARLY, all the normal threads enter into the else part of the conditional statement(if BT != 0), & I have similarly initialised the start and end timer for it. I later store it in the time_array.

LET'S UNDERSTAND THE CPU ALLOCATION FOR THE BOUNDED THREADS THROUGH THE CODE SNIPPET:

The lines 60-63 are of **utmost importance** while trying to look at CPU-Allocation for the threads to be bounded.

Here, suppose **BT = 12, K = 48** (*****Expt 1 Example*****)

Then threads per core(for allocation) is given by K/C = 4.

(*NOTE STARTS***)**
Threads per core remains constant for both the experiments
In Experiment 1: 'b' threads assigned to each core where **b = K/C.**

In Experiment 2: **K/2 threads distributed equally among C/2 cores, i**.e K/C threads per core again.
(*NOTE ENDS***)**

Hence, Threads 0-3 are allocated to the core 0
Threads 4-7 allocated to the core 1
Threads 8-11 allocated to core 2

Can be easily understood through the formula on line 62:

CPU_core allocated = thread_no / (K/C)

EG. Thread no. 7 is allocated the cpu 1 , since int(7/4) = 1

Here, on line 63, I have used the function **CPU_SET** .
It is used to control the CPU affinity of a process or a thread, which determines which CPU cores the process or thread is allowed to execute on.

Further, I have used the function **pthread_setaffinity_np()** which takes as argument the **thread-ID** of the thread we want to allocate to the CPU specified by the **CPU set mask** initialised in line 60.

Rest of the code **remains the same** as Assignment 1 which includes the logic behind distributing the rows of the matrix among the different threads.

CASE 2: BT =0

Since no thread is bounded in this case, all threads directly enter the 'else condition', and continue their further execution. In this case, the time_array only contains the time execution for each of the K normal threads.

The time taken by the Bounded threads is 0(since no thread is bounded in this case).

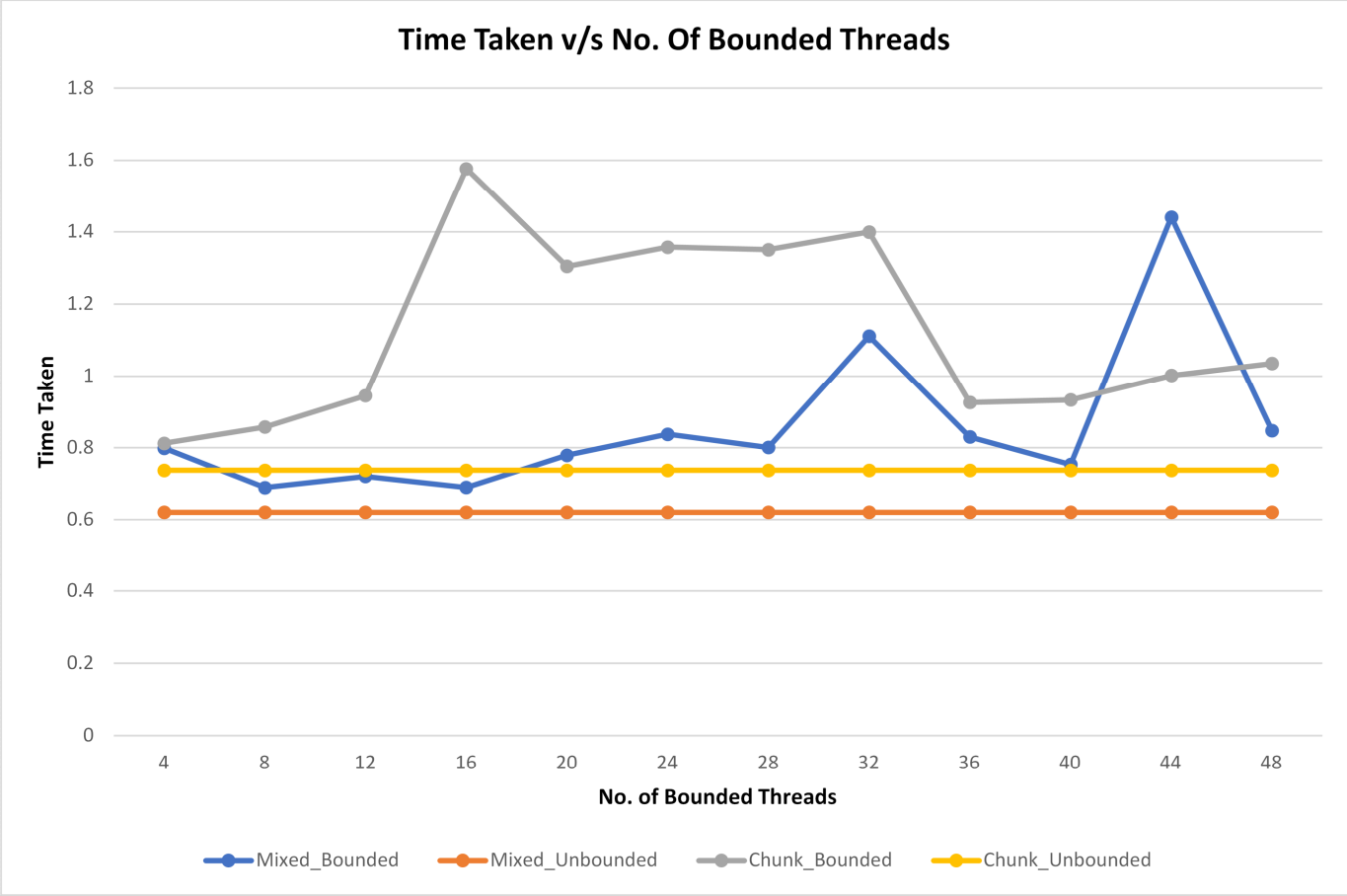
Below Is the snapshot of the runner function for the **mixed_technique**. The code contains no changes compared to

chunk_technique code provided above, apart from the logic used to distribute the rows of the matrix among the cores(already explained as a part of Assignment 1)

```
54  //mixed technique to compute the C matrix
55  void* mixed_technique(void* arg){
56      intermediary* inner = (intermediary*)arg;
57      if(inner->thread_no-1 < BT){
58          auto start = chrono::steady_clock::now();
59          cpu_set_t mask;
60          CPU_ZERO(&mask);
61          int cpu_no = (inner->thread_no-1)*C/K;
62          CPU_SET(cpu_no, &mask);
63
64          pthread_setaffinity_np(pthread_self(), sizeof(cpu_set_t), &mask);
65
66          for(int i = inner->thread_no-1; i < N; i += K){
67              for(int j = 0; j < N; j++){
68                  global_array[i][j] = compute_Cij(inner->input_array, i, j);
69              }
70          }
71          auto end = chrono::steady_clock::now();
72          std::chrono::duration<double> duration = end - start;
73          double iteration_time = duration.count();
74          time_array[inner->thread_no-1] = iteration_time;
75      }
76      else{
77          auto start = chrono::steady_clock::now();
78          for(int i = inner->thread_no-1; i < N; i += K){
79              for(int j = 0; j < N; j++){
80                  global_array[i][j] = compute_Cij(inner->input_array, i, j);
81              }
82          }
83          auto end = chrono::steady_clock::now();
84          std::chrono::duration<double> duration = end - start;
85          double iteration_time = duration.count();
86          time_array[inner->thread_no-1] = iteration_time;
87      }
88
89      return NULL;
90  }
```

Now let's look & analyse at the graphs for experiment 1 & 2:

EXPERIMENT 1:



It’s evident from the graph that the Bounded Threads(in both techniques) take more time than the case when no threads is bounded. Let's look at the possible reason for the same:

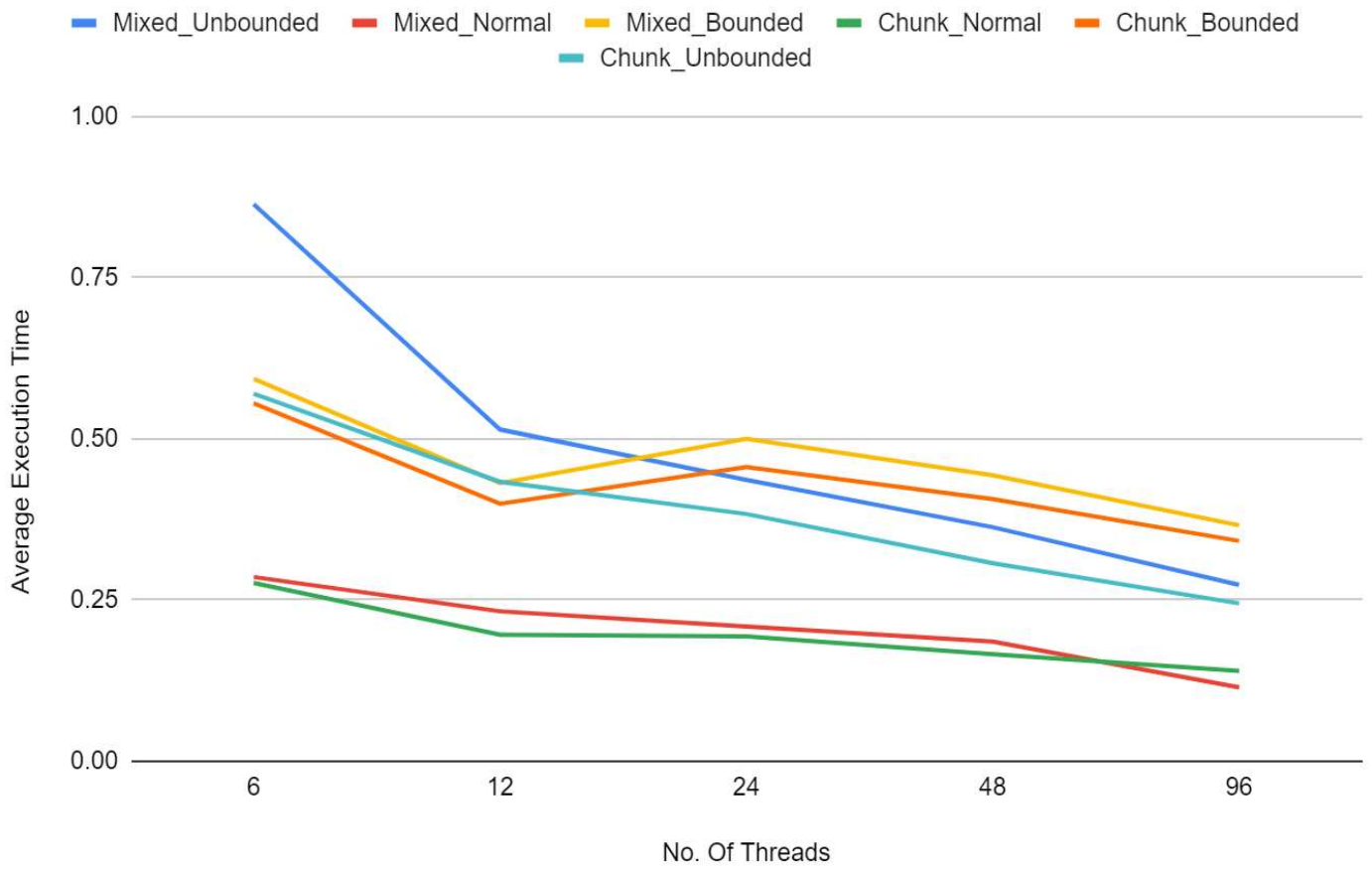
LOAD BALANCING: Without thread affinity, the operating system can dynamically ALLOCATE threads to different core based on the workload on the cores. Consider a case, when due to use of thread affinity, some cores which are not allocated any threads might remain idle or have less workload as compared to the other cores which have been assigned threads.

In dynamic allocation, such a time overhead is not created, resulting in less program runtime.

THREAD AFFINITY OVERHEAD: Let's also not forget that managing thread affinity also consumes time. Assigning threads to specific cores requires system calls and CPU resources, which can lead to overhead.

ALSO, thread affinity can even introduce cache contention. They compete for resources like cache space. It can lead to increased latency, hence we don't get the potential benefits if thread affinity.

EXPERIMENT 2:



HERE also, you can see that the average time execution for the bounded threads is **more than normal threads** in most of the case for both the techniques. The reasons for the same can be understood through the points mentioned before.

We can atleast be sure with one thing, that **MULTI-THREADING WILL ALWAYS lead to decreased program runtimes**. As you can see from the graph, for **Mixed_Unbounded** & **Chunk_Unbounded**, the program runtimes decreases as the value of K increases.

WE can understand it this way, **as the no. of threads increases**, **we can distribute the input data into smaller and smaller data-sets distributing it further among the threads. Hence, the square of the matrix is calculated more quickly as K increases.**