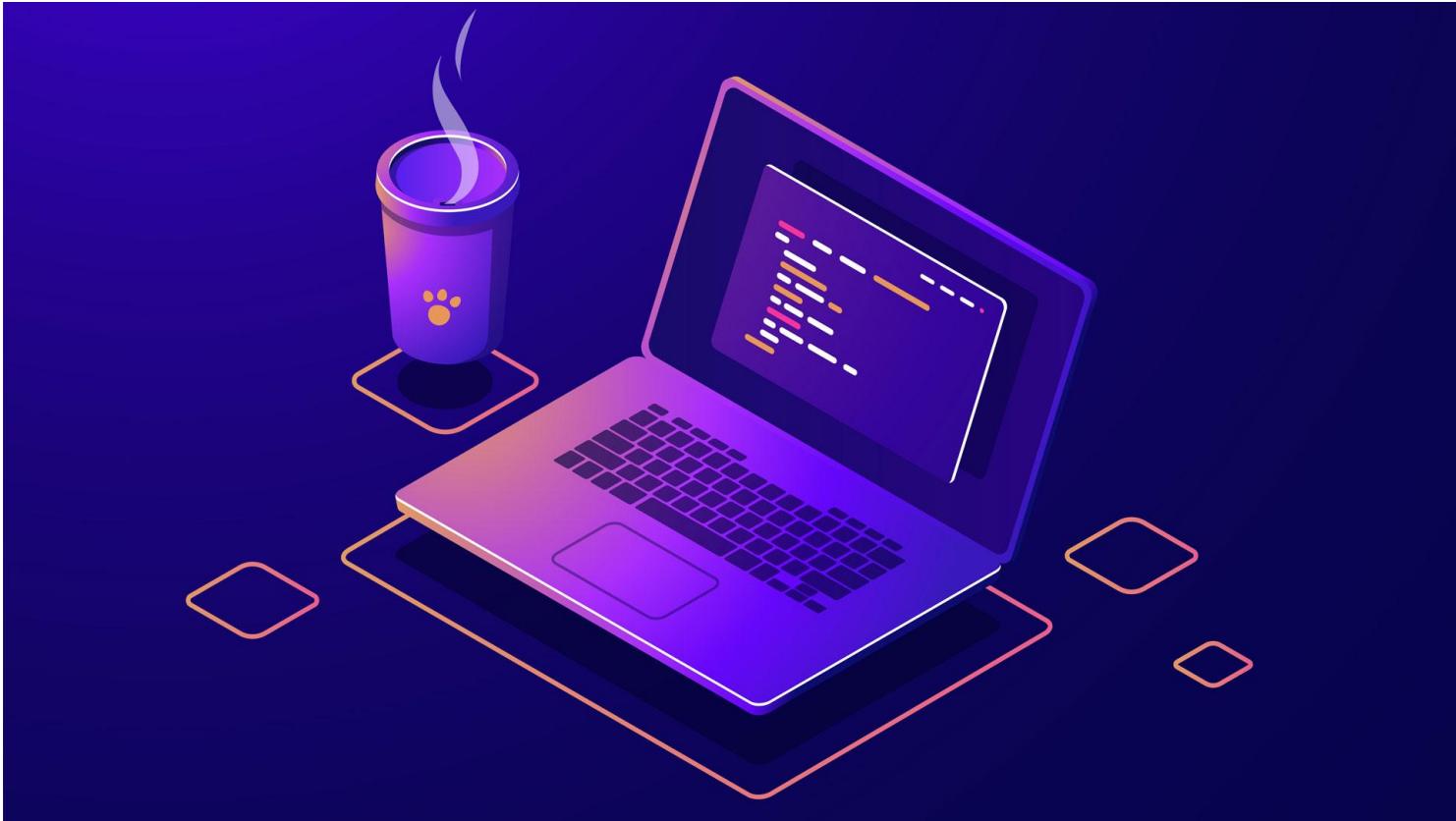
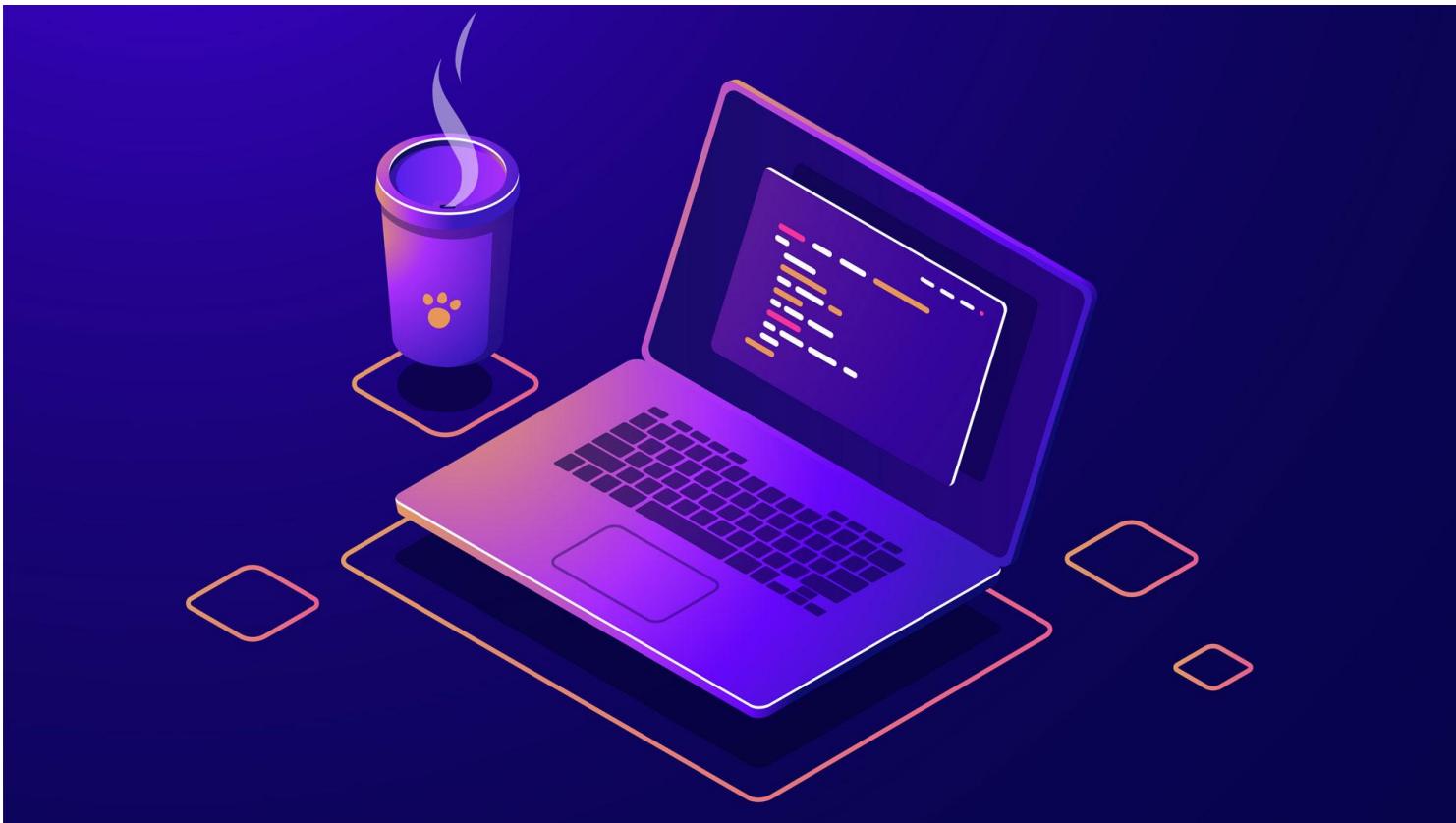


# OpenMP Problems



# Problem 5



# Problem 5



We ask to parallelize the computation of the histogram of values appearing on a vector. The histogram is another vector in which each position counts the number of elements in the input vector that are in a certain value range. The following program shows a possible sequential implementation for computing the histogram (vector **frequency**) of the input vector **numbers**:

```
#define MAX_ELEM 1024*1024
#define HIST_SIZE 250
unsigned int numbers[MAX_ELEM];
unsigned int frequency[HIST_SIZE];

void ReadNumbers (int * input, int * size);
void FindBounds(int * input, int size, int * min, int * max) {
    for (int i=0; i<size; i++)
        if (input[i]>(*max)) (*max)=input[i];

    for (int i=0; i<size; i++)
        if (input[i]<(*min)) (*min)=input[i];
}

void FindFrequency(int * input, int size , int * histogram, int min, int max) {
    int tmp;
    for (int i=0; i<size; i++) {
        tmp = (input[i] - min) * (HIST_SIZE / (max - min - 1));
        histogram[tmp]++;
    }
}
```



# Problem 5

```
void DrawHistogram(int * histogram, int minimum, int maximum);
void main() {
    int num_elem, max, min;

    ReadNumbers(numbers, &num_elem); // read input numbers
    max=min=numbers[0];
    FindBounds(numbers, num_elem, &min, &max); // returns the upper and lower
                                                // values for the histogram
    FindFrequency(numbers, num_elem, frequency, min, max); // compute histogram
    DrawHistogram(frequency, min, max); // print the histogram
}
```



# Problem 5

- a) Write a parallel version with OpenMP for function `FindBounds` using an iterative task decomposition, in which you only generate as many tasks as threads in the parallel region, as you minimize the possible synchronization overheads.
  
- b) Write a parallel implementation for the function `FindFrequency` using OpenMP.

# Problem 5 - Point B - Solution



```
void FindFrequency(int * input, int size , int * histogram, int min, int max) {  
    int chunk_size = (size + omp_get_max_threads() - 1) / omp_get_max_threads();  
  
    #pragma omp parallel num_threads(omp_get_max_threads())  
    {  
        int tid = omp_get_thread_num();  
        int start = tid * chunk_size;  
        int end = (tid + 1) * chunk_size;  
        if (end > size) end = size;  
  
        int tmp;  
        for (int i = start; i < end; i++) {  
            tmp = (input[i] - min) * (HIST_SIZE / (max - min - 1));  
            #pragma omp atomic  
            histogram[tmp]++;
        }
    }
}
```

# Problem 5 - Point B - Solution



In this parallel version of `FindFrequency`, we first determine the chunk size based on the input size and the number of threads in the parallel region. Then, we create a parallel region using `#pragma omp parallel` with the number of threads specified by `omp_get_max_threads()`. Inside the parallel region, each thread computes the histogram of a subset of the input vector, as determined by its thread ID (`omp_get_thread_num()`) and the chunk size. To avoid race conditions when updating the histogram, we use an atomic operation (`#pragma omp atomic`) to increment the appropriate histogram bin.