**Assignment 3 :**

This code demonstrates parallel summation of an array using multiple threads. Let's go through it step by step:

1. The code begins by importing the necessary Java libraries: `java.util.Arrays` for array manipulation, `java.util.concurrent.ExecutorService` and `java.util.concurrent.Executors` for managing thread execution, and `java.util.concurrent.TimeUnit` for specifying time units.

2. The class `ParallelSum` is defined, which contains the `main` method where the execution of the program starts.

3. The constant variable `NUM\_THREADS` is declared and assigned the value of the number of available processors in the system, obtained using `Runtime.getRuntime().availableProcessors()`. This will determine the number of threads to be used for parallel processing.

4. An array `array` is created as a sample array to be summed.

5. The array is divided into equal parts to be processed by each thread. The variable `partitionSize` is calculated as the array length divided by the number of threads. An empty two-dimensional array `partitions` is created to store the divided parts of the array.

6. A loop is used to divide the array and populate the `partitions` array. Each partition is obtained using `Arrays.copyOfRange()` method, which creates a new array containing elements from the specified range of the original array.

7. An `ExecutorService` is created using `Executors.newFixedThreadPool(NUM\_THREADS)`, which creates a thread pool with a fixed number of threads equal to `NUM\_THREADS`.

8. An array of `SumTask` objects, named `tasks`, is created to store the tasks to be executed by the threads.

9. Another loop is used to create and submit `SumTask` objects to the executor. Each `SumTask` object is initialized with a partition of the array, and then the task is submitted to the executor using `executor.execute(tasks[i])`.

10. After all the tasks have been submitted, the executor is shut down using `executor.shutdown()` to indicate that no more tasks will be submitted.

11. The main thread waits for all tasks to complete by calling `executor.awaitTermination(Long.MAX\_VALUE, TimeUnit.NANOSECONDS)`. This method blocks until all tasks have completed execution or the specified timeout period has elapsed.

12. In case of an interruption during the waiting period, an `InterruptedException` is caught and the stack trace is printed.

13. Once all tasks have completed, the partial results from each `SumTask` object are summed up in the `sum` variable using a loop.

14. Finally, the total sum is printed to the console.

15. The code also defines a nested static class `SumTask` which implements the `Runnable` interface. Each `SumTask` represents a task to be executed by a thread. It contains an array to process, and the `run` method performs the actual summing of the array elements and stores the result in the `result` variable.

That's an overview of the code. It divides the array into multiple partitions, assigns each partition to a separate thread, and sums the partial results to obtain the total sum using parallel processing.

Own theory :

MPI is an communication protocol to program a parallel processing