**Assignment 4 – OpenMP**

Question: (Reduce)

Does the plots make sense? Why?  
- The plots gets less speedup and it makes sense as the code for which we have used OpenMP isn’t actually a parallel code and OpenMP is still trying to make it parallel. In the code all the threads tries to write on the global variable (result\_sum in this case). Thus, this becomes an overhead for the thread to get access to the global variable and acquiring locks to write on it. This slows down the speedup as they wait for the other threads to release their access. For all the scheduling types the speed is same because even if we distribute the work among maximum number of threads every iteration has just one line code to do and that too is where the global variable is accessed and it is something which can be read or written by one thread at a time.

Question: (NumInt)

Does the plot make sense? Why  
- As can be seen from the graphs, for the higher values of iterations (n) the speedup gained is linear and also increases with the increase in the intensity along with the balanced granularity which makes a perfect balance between the threads granularity and intensity thus giving max speedup as there’s more work per thread now and so it gives better performance as every thread takes enough time to perform iterations and thus gives buffer time for the other threads. This trend of increasing speedup is linear throughout the cases where number of iterations is getting higher.

Question: (Prefixsum)

Does the plot make sense? Why?  
- In the prefix sum I have divided the first part of for loop into n chunks and have run it with a scheduling policy. As can be seen from the graphs no much speedup can be gained in the parallel code because as after the first part the end array element of the array is added to a temporary array. Then again the next part is made to run in parallel for adding the missing part in next array elements.

Structure of the parallel Prefixsum implemented:

The first for loop divides the given array into n chunks. Then each chunk adds its own prefixsum. After the end of first loop every chunk adds its last element in the temporary array and then again the chunks missing the adjustment value adds its own missing part parallelly to complete prefixsum.

Structure of parallel Mergesort:

For this we can divide a inner for loop depending upon the different chunks we plan to merge at a time.  
Eg. We would first merge two elements in pair one by one, then four elements(two sorted arrays) and then eight elements so on till the n is reached and the entire loop is sorted. The first for loop keeps track of it by first initiating a value as 2 then incrementing everytime.

The second loop accepts this as the increment size as it will provide the starting element to the another loop which compares the elements one by one and then swaps if needed. During this comparison we will have to keep a track of the start of the next sorted array too and compare with the corresponding element in the adjacent array. The second for loop is incremented by the value of the outer for loop to keep track of how many elements to compare.