1. (Reduction) Does the plots make sense? Why?
The speedup in reduce_plots_threads graph is very low, because the code isn't parallel enough to give justice to openMP's reduction directive. By using reduction clause we are locking the global variable and failing to achieve much speedup. Use of single thread shows a linear graph compared to multiple threads. Which means that other threads are too dependent on computations done by first thread resulting in first half of graph being flat in threads ranging from 2 to 16 for all scheduling types. In the reduce_plots graph we can see that only for very high intensity there is some sort of speed up achieved, as opposed to the almost negligible speed up for lower intensities across all scheduling types against various granularity.

2. (NumInt) Does the plots make sense? Why?
Speedup is stable and constant for higher intensities where n is highest, which means that optimum balance between granularity, n, intensities and threads is achieved in the last two graphs of numint_plots. In num_plots_thread graph, we can see linear speedup wherever optimum balance of all the factors is achieved. For more number of threads and computations with low granularity, speedup should get affected due to too many context switches, but at the same time increased intensity results in helping achieve yet more speed up. Similarly, for computations with less intensity, yet higher granularity good enough speed up is noticed because of work is evenly distributed among threads.

3. (Prefix Sum)
Parallel Structure:
1. Array P is cut into pieces equivalent to number of threads.
2. First element of every chunk will get appended by zero (so we can add the offset value later.)
3. The last element of every chunk is added in a separate array to keep track of offset correction values.
4. Perform prefix operation again on this array containing offset values.
5. Add the array of correction values, to the succeeding chunks in parallel.
6. Validate the resulting ‘pr’ array with expected output.

Does the plots make sense? Why?
In prefix graph, the lack of speed up implies that computations are unable to be computed with optimum parallelism. We have cut in the ‘pr’ array to receive partial computations. A negligible part of code is sequential after this computation to acquire prefix of offset values needed for further correction. Then we again add up this offset or correction value to their respective chunks of data in parallel. In spite of testing code with various combinations of scheduling, I can observe that this result is not dependent of type of scheduling used.

Merge Sort
Parallel Structure:
1. Divide the iteration based on number of threads
2. Keep track of incrementing variable globally within the parallel implementation.
3. Divide the array equally until we get down to single elements.
4. Compute left most, middle and right most variables for very iteration.
5. Call merge iteratively within the loops.
6. Merge method will compare first element of each two arrays passed to it, and arrange numbers in ascending order.
7. Return resulting merge sorted array.