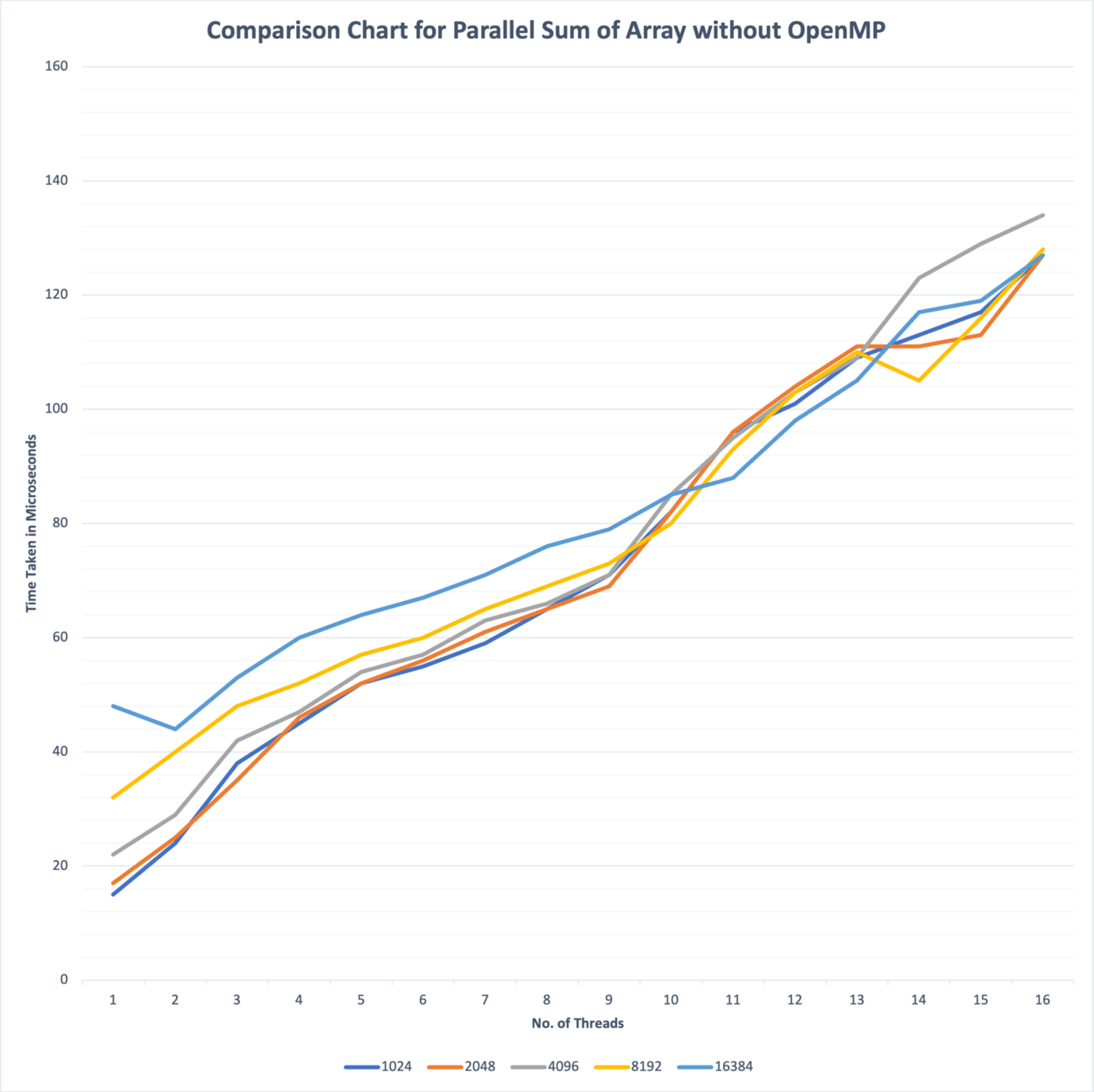
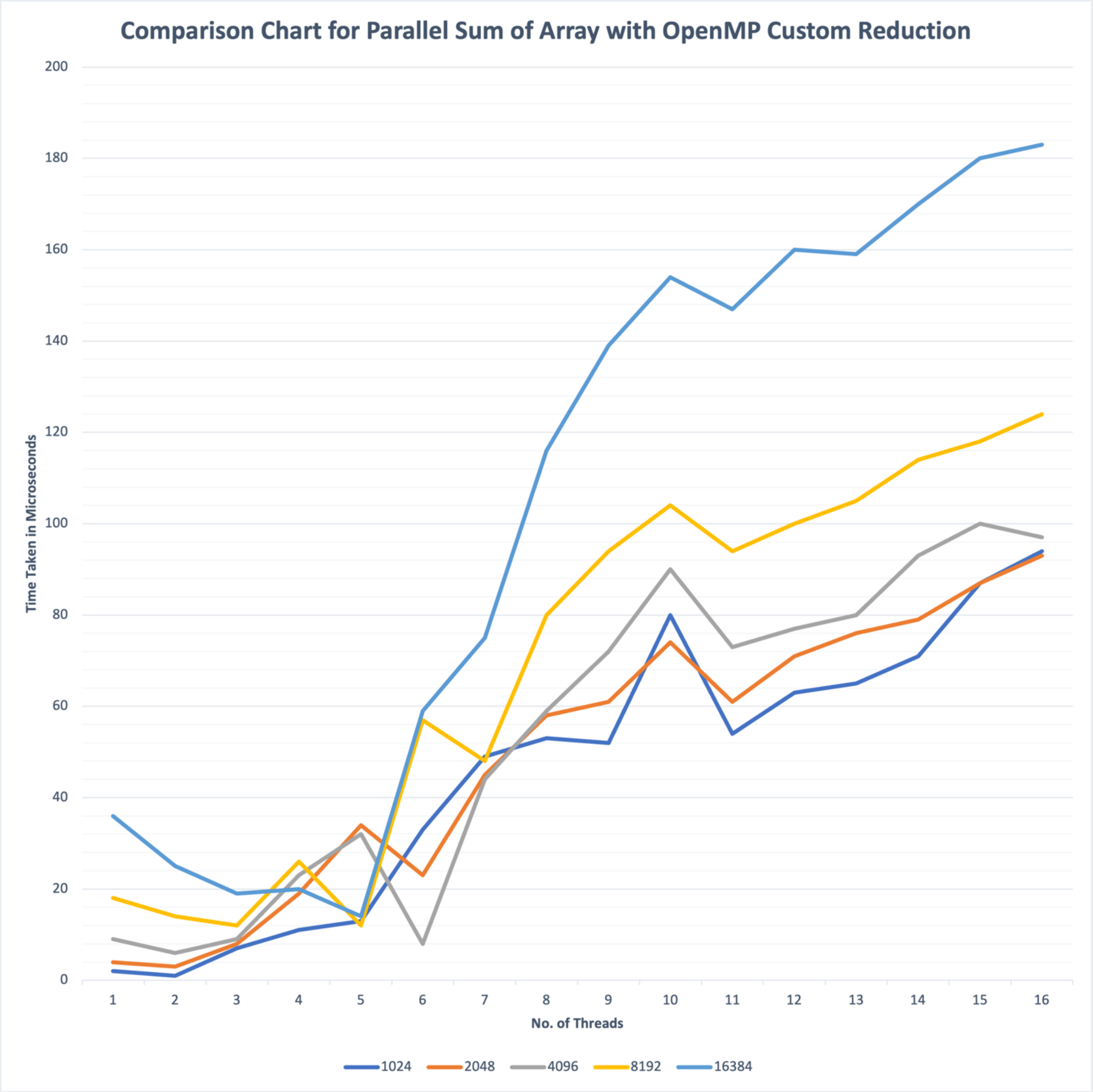
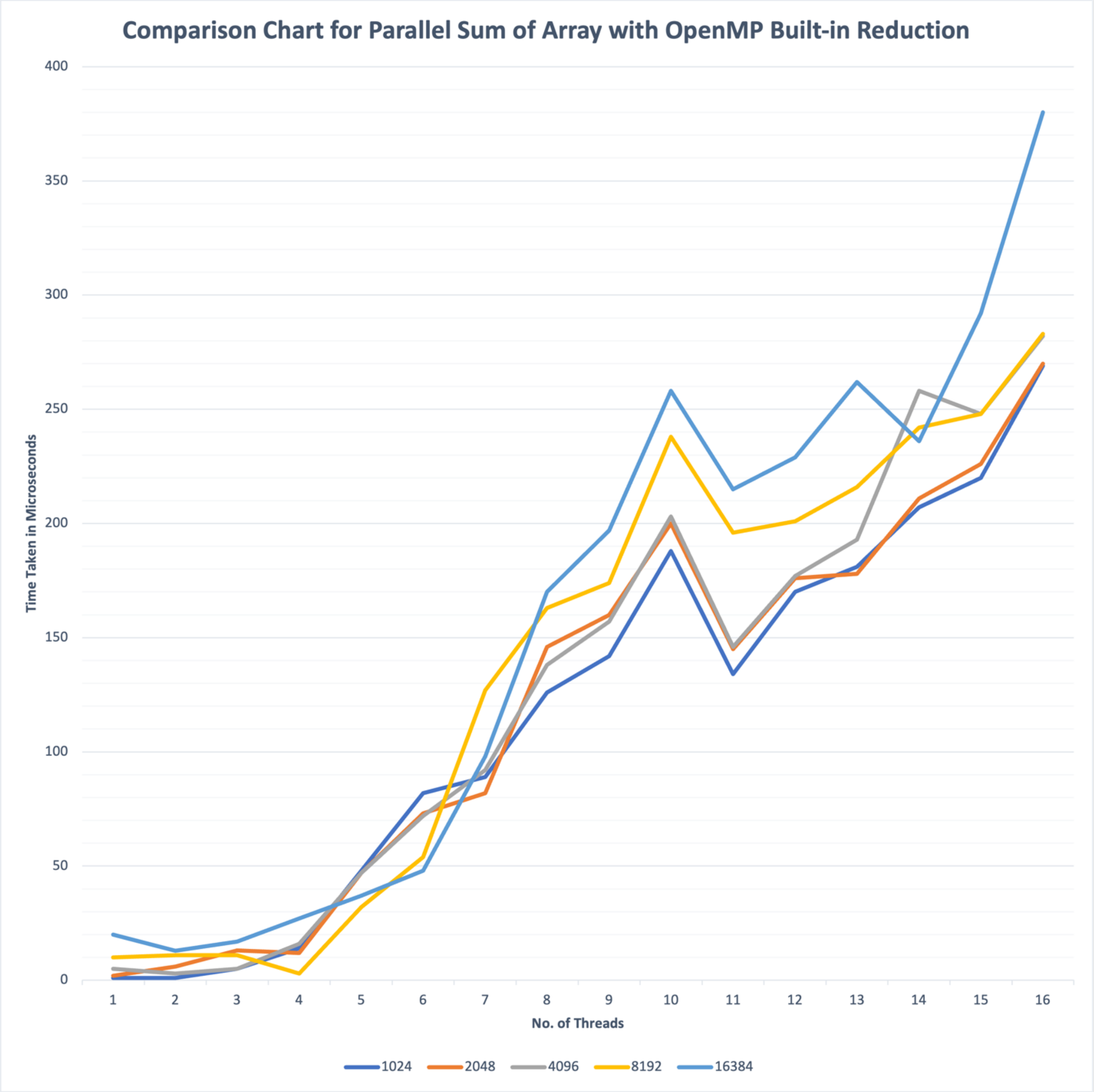
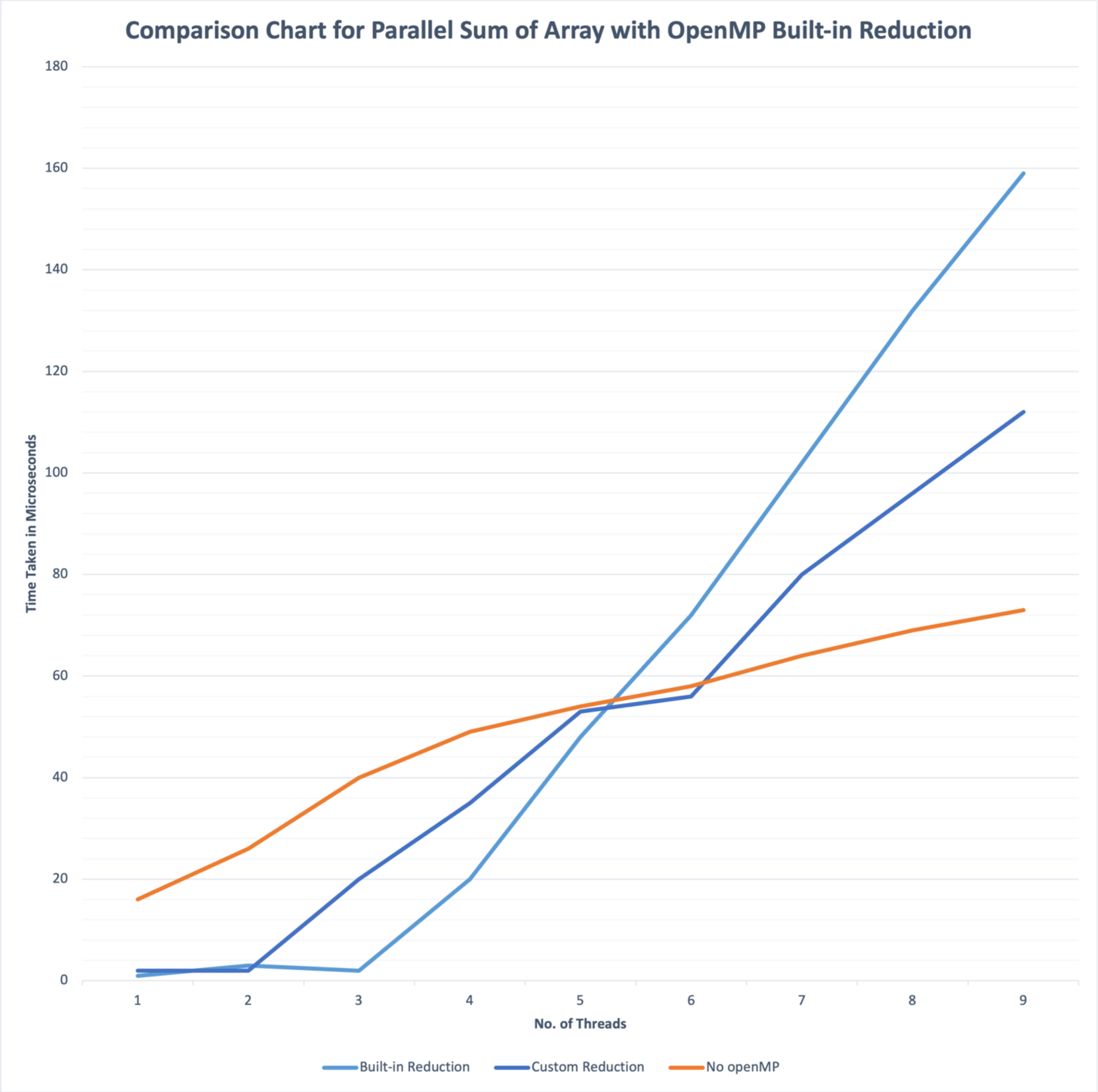

Comparison Chart for Parallel Sum of Array without OpenMP
Time Taken in Microseconds
160
140
120
100
80
60
40
20
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
No. of Threads
1024
2048
4096
8192
16384


Comparison Chart for Parallel Sum of Array with OpenMP Custom Reduction
Time Taken in Microseconds
200
180
160
140
120
100
80
60
40
20
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
No. of Threads
1024
2048
4096
8192
16384



Answer 1 & 2: Based on these graphs, it looks like there are different sections in which the different functions can outperform each other and be closer to the ideal case. Graph 1 shows a more consistent performance, Graph 2 shows better performance towards higher thread count and Graph 3 shows better performance with lower thread count. This can happen to many reasons. There is transitional delay between switching of threads, critical section that could affect such things. Things such as using logical cores instead of physical cores.



Answer 3: The function that is not using openMP seems to perform better in case weak scaling. While the custom reduction function and builtin reduction function seems to perform best in in different thread count subsets in case of strong scaling. Due to less potential of reduction in the standard function, it seems to scale proportionally across the subsets. While, custom reduction is built to reduce with demanding subsets in mind. On the other hand, the builtin reduction is optimized for lower subsets it seems like, in other words, where ever it should be sequentially faster, it manages to not waste time in that subset.