Design Document

1. The algorithm implemented is hyperquicksort.
   First each thread receives equal number of elements as possible. Then at the 1st step of the algorithm each thread sort its share of number using sequential quicksort. Then a thread is chosen from the group of thread for picking median. The groups are decided on the basis of the current iteration and threadNo. The median is the middle element of the chosen thread. On the basis of median and thread number each thread prepares a list of numbers called low/high list. This low/high list is shared with another neighbouring process which in turn shares it's high/low list with current process. Then each thread performs merge operation of the values it currently is holding and incoming values from its neighbour. All the threads write their result in a shared memory array. This process after the sorting is repeated log2(No_of_thread) times. Each time reducing the dimensionality of the hypercube formed by 1.

   In the code various variables like arrRead[], arrWrite[], threadInfo[], chunkSizes[], medians[] are shared variables across the threads. Each one storing some necessary information related to operations ahead.

   No of threads are set using an MACRO called NO_OF_THREADS defined at the start of the program.

   NOTE: The no of threads created should always be a power of 2. Because in the topology of the algorithm each processor occupies a node of the hypercube of log2(No_of_processors) dimension. And each thread will be executed on a processor. Hence no of processors/thread should be a power of 2.

   The code takes two arguments as command line inputs. First is value of N i.e., the number of elements to sort and Second one is the choice (to see original and sorted array).
   Format of the run command
   ```
   ./hyperquicksort_openmp <no_of_elements(N)> <choice>
   ```

So, a sample run command will look like this

```
./hyperquicksort_openmp 1024 0
```

2. **Serial Fraction:**
   For N = 32768 and P = 1, execution time = 88356 microseconds
   sequential portion execution time = 13566 microseconds.

   Hence serial fraction is: (135660 / 88356) = 0.15353.

3. **Performance metrics**:
   By Amdahl's law, speedup is calculated as: S(p) = 1 / (f + ((1-f) / p)).

   Running time of the algorithm w.r.t no of threads and input.
   NOTE: - All the units are seconds, where there are different units, it is specified.

Threads	N = 64	N=100	N=1024	N = 32768
2	0.000254	0.000415	0.000487	0.010169
4	0.000522	0.000542	0.00687	0.005619
8	0.000881	0.000953	00.001258	0.015381

For 8-threads

	N = 64	N = 100	N = 1024	N = 32768
Sequential QuickSort time	0.00004	0.00007	0.000195	0.026081
SpeedUp	0.0454	0.073452	0.15652	1.69
Efficiency	0.005675	0.0091815	0.019565	0.21195
Speedup by Amdahl's Law	3.8559	3.8559	3.8559	3.8559

For 4-threads

	N = 64	N = 100	N = 1024	N = 32768
Sequential QuickSort time	0.00004	0.00007	0.000195	0.025517
SpeedUp	0.076628	0.129151	0.0283	4.5411
Efficiency	0.019157	0.0.03228	0.007075	1.13529
SpeedUp by Amdahl's Law	2.7386	2.7386	2.7386	2.7386

For 2-threads

	N = 64	N = 100	N = 1024	N = 32768
Sequential QuickSort time	0.00004	0.00007	0.000195	0.025469
SpeedUp	0.1574	0.16867	0.400410	2.3825
Efficiency	0.0787	0.084335	0.200205	1.1904
SpeedUp by Amdahl's Law	1.73380	1.73380	1.73380	1.73380

As we can see in all the three tables, speedups calculated by Amdahl's law are more than observed speedups when input size is small. But when input size is sufficiently large the Speedup achieved by the algorithm is significantly larger than that of Speedup calculated from Amdahl's Law.

4. **Iso Efficiency Function**: As we can see from the tables, for 8-thread and 4-thread as well as 4-thread and 2-thread, when the no of threads becomes twice, the input size increases from N = 64 to N = 1024.
   So, when input becomes twice the input size need to become 32 times to keep efficiency constant.

Hence, iso-efficiency function is:

$$\theta(p^4)$$

5. **<u>Estimating no. of processors for cost optimality</u>**:

For cost optimality we must have:

$$pT_p = \theta(W)$$

From this we can say,

$$W + T_o(W,p) = \theta(W)$$

$$(W,p) = O(W)$$

$$W = \Omega\big(T_o(W,p)\big)$$

We also calculated the iso-efficiency function: $f(p) = \theta(p^4)$

Thus, for the cost optimality we must have:

$$W = \Omega(f(p))$$

We also know, that for the given problem $W = \theta(nlog_2n)$ (most optimal serial algorithm that we know for sorting; precisely we can say for the average case that $W = 2n + nlog_2n$ - $2n$ operations for recursive calls and $nlogn$ for partitions).

$$\Rightarrow 2n + nlogn = \Omega(p^4)$$

$$\Rightarrow 2n + nlog_2n \geq p^4$$

$$p \leq (2n\ + nlog_2n)^{\frac{1}{4}}$$

Thus, this is the upper bound / maximum value of the thread count for achieving cost optimality while using hyperquick sort for an array of length n.