2. In this problem, you will start by selecting a sorting algorithm to sort 10,000 random integers with values 1-10,000. We will use pthreads in the problem to generate a parallel version of your sorting algorithm. If you select code from the web (and you are encouraged to do so), make sure to cite the source of where you obtained the code. You are also welcome to write your own code from scratch. You will also need to generate 10,000 random integers (do not time this part of your program). Make sure to print out the final results of the sorted integers to demonstrate that your code works (do not include a printout of the 10,000 numbers). Provide your source code in your submission on Canvas.
    a) Run your program with 1, 2, 4, 8 and 32 threads on any system of your choosing. The system should have at least 4 cores for you to run your program on. Report the performance of your program.
    b) Describe some of the challenges faced when performing sorting with multiple threads.
    c) Evaluate both the weak scaling and strong scaling properties of your sorting implementation

**Answer**:

2 (a):

**Performance Report**:

| Threads | Execution time (in seconds) | Speed up (vs 1 thread) |
|---|---|---|
| 1 | 0.0020 | 1.00x |
| 2 | 0.0012 | 1.67x |
| 4 | 0.0009 | 2.22x |
| 8 | 0.0008 | 2.50x |
| 32 | 0.0031 | 0.65x |

The performance report shows how the time a program takes to run changes when using different amounts of threads. It also shows how much faster the program runs compared to using just one thread. When the number of threads goes from 1 to 8, the time it takes to complete the task is greatly reduced. This shows that the system is working well in parallel, with speed improvements between 1.67 times and 2.50 times faster. At 32 threads, the execution time goes up, leading to a speed gain of only 0.65 times compared to using a single thread. This drop shows that the program is less efficient because it has to manage many threads and there may not be enough resources available. This is a usual issue in parallel computing when the number of threads goes beyond what the hardware and workload can handle well.

2 (b): While performing sorting with multiple threads, several challenges arise due to the inherent complexity of parallel computation. Several challenges arise such as:

- **Data Dependencies and Synchronization**: Sorting algorithms often call for the merging or amalgamation of sorted segments. Ensuring accurate synchronization throughout these phases without race circumstances is essential. For instance, parallel mergesort must meticulously oversee the merging of subarrays to prevent conflicts and maintain data integrity.

- **Distribution of workload among multiple resources**: The disproportionate allocation of tasks among threads may result in idle threads. Suboptimal pivot selection in parallel quicksort can lead to uneven partition sizes, resulting in certain threads handling bigger segments while others are underutilized.
- **Synchronization Overhead:** The creation and management of threads incurs costs. For diminutive datasets, this overhead may undermine the advantages of parallelism. Effectively allocating work and reducing thread creation and termination cycles is crucial.
- **Cache Coherency:** Threads operating on overlapping memory regions may result in frequent cache invalidations, hence elevating latency. Developing techniques to reduce shared memory access (e.g., operating on separate array portions) aids in alleviating this issue.
- **Race Conditions**: Simultaneous read/write operations on shared data (e.g., element swapping in quicksort) may result in corruption. Appropriate utilization of locks, atomic operations, or partitioning methods is essential to avert such problems.
- **Memory Overhead**: Concurrent implementations frequently necessitate supplementary memory (e.g., temporary arrays in mergesort). Efficient memory management is essential, particularly for extensive datasets.

2 (c): **Strong Scaling Efficiency**: In strong scaling, the number of threads is increased but the total array size remains constant

Strong scaling efficiency here is calculated as

Efficiency= [Execution Time at 1 Thread × 100/ (Execution Time at N Threads × N)]

| **Threads** | **Array Size (elements)** | **Execution Time (s)** | **Efficiency** |
|---|---|---|---|
| 1 | 10000 | 0.0020 | 100% |
| 2 | 10000 | 0.0012 | 83.5% |
| 4 | 10000 | 0.0009 | 55.5% |
| 8 | 10000 | 0.0008 | 31.25% |
| 32 | 10000 | 0.0031 | 2.03% |

**Weak Scaling Efficiency**: The total array size increases in proportion to the number of threads, keeping the workload per thread constant.

Weak scaling efficiency is calculated as:

Efficiency= (Execution Time at 1 Thread/Execution Time at N Threads)×100

| **Threads** | **Array Size (elements)** | **Execution Time (s)** | **Efficiency** |
|---|---|---|---|
| 1 | 10000 | 0.0020 | 100% |
| 2 | 20000 | 0.0013 | 153.8% |

| | | | |
|---|---|---|---|
| 4 | 40000 | 0.0010 | 200% |
| 8 | 80000 | 0.0009 | 222% |
| 32 | 160000 | 0.0034 | 58.8% |

From the results it is clear that the strong scaling of the implementation is good with a small number of threads but becomes inefficient as the thread count increases. This shows that there might be a maximum number of threads that are beneficial for this specific task and configuration, beyond which more threads could even impair performance.

The weak scaling results demonstrate that the approach scales effectively with the rising demand and threads to a certain extent. Nevertheless, expanding beyond eight threads starts to produce inefficiencies that undermine certain advantages of increased parallelism. This indicates that the solution effectively manages small to moderate increases in workload and threads, but is not optimized for extremely high concurrency or substantial datasets.