PERFORMANCE CONSIDERATION

1.Minimize synchronization.

- Avoid or minimize the use of BARRIER, CRITICAL sections, ORDERED regions, and locks.
- Use the NOWAIT clause where possible to eliminate redundant or unnecessary barriers. For example, there is always an implied barrier at the end of a parallel region. Adding NOWAIT to a final DO in the region eliminates one redundant barrier.
- Use named CRITICAL sections for fine-grained locking
- Use explicit FLUSH with care. Flushes can cause data cache restores to memory, and subsequent data accesses may require reloads from memory, all of which decrease efficiency

For example:

This construct is less efficient:

```
!$OMP PARALLEL
 ....
 !$OMP DO
   ....
 !$OMP END DO
 ....
!$OMP END PARALLEL
!$OMP PARALLEL
 ....
  !$OMP DO
    ....
  !$OMP END DO
 ....
```

Efficient one:

```
!$OMP PARALLEL
 ....
 !$OMP DO
   ....
 !$OMP END DO

 .....

 !$OMP DO
   ....
 !$OMP END DO
!$OMP END PARALLEL
```

2. Choose the appropriate loop scheduling.

- STATIC causes no synchronization overhead and can maintain data locality when data fits in cache. However, STATIC may lead to load imbalance.
- DYNAMIC,GUIDED incurs a synchronization overhead to keep track of which chunks have been assigned. And, while these schedules could lead to poor data locality, they can improve load balancing. Experiment with different chunk sizes.

3. Use LASTPRIVATE with care, as it has the potential of high overhead.

- Data needs to be copied from private to shared storage upon return from the parallel construct.
- The compiled code checks which thread executes the logically last iteration. This imposes extra work at the end of each chunk in a parallel DO/FOR. The overhead adds up if there are many chunks.

4. Use efficient thread-safe memory management.

- Applications could be using malloc() and free() explicitly, or implicitly in the compiler-generated code for dynamic/allocatable arrays, vectorized intrinsics, and so on.
- The thread-safe malloc() and free() in libc have a high synchronization overhead caused by internal locking. Faster versions can be found in the libmtmalloc library. Link with -lmtmalloc to use libmtmalloc.
- Small data cases may cause OpenMP parallel loops to underperform. Use the IF clause on PARALLEL constructs to indicate that a loop should run parallel only in those cases where some performance gain can be expected.

5. When possible, merge loops