Ömer Öztürk
S046798

Q1: Loop-carried Data Dependencies

To eliminate the dependency, we can rewrite the problem by recognizing the pattern in the computation of a[i]. The pattern is a[i] = i * (i + 1) / 2. Using the formula, the value of a[i] can be computed independently for each index i without relying on the value of a[i-1]. This eliminates the loop-carried dependency. With the dependency removed, the loop can now be parallelized as can be seen below. Code also provided in zip file

```
a [0] = 0;
#pragma omp parallel for num_threads(thread_count)
for (i = 1; i < n; i++)
        a[i] = i * (i + 1) / 2;
```

Q2: Critical Sections vs Atomic Operations

```
omero@omero-fedora:~/Education/Multicore_Programming/Homeworks/Hw3$ ./Q2
Thread Count | Atomic Time (s) | Critical Time (s)
-----------------------------------------------
           1 |        0.390648 |        0.265480
           2 |        0.372927 |        0.959493
           4 |        0.471233 |        3.204557
           8 |        0.505609 |        7.780302
          16 |        0.815776 |       15.441305
```

Code is provided in zip file.

Conclusion:

No, multiple atomic directives that modify different variables are not treated as a single critical section in OpenMP. Each atomic operation is handled independently, allowing concurrent execution as long as they operate on separate memory locations. This design enables better performance and scalability in parallel applications by minimizing unnecessary serialization.

Q3:

a) Private: i, j, count | shared: a, temp
b) No, there are no loop-carried data dependencies when the for i loop is parallelized. Because iterations are independent and “i”, “j”, and “count” variables are private to each thread. This ensures that there is no interference or race condition between threads,
c) Code output

```
omero@omero-fedora:~/Education/Multicore_Programming/Homeworks/Hw3$ ./Q3 30
Original array: 86|78|44|75|51|47|49|19|19|26|57|42|33| 6|21|68|78|79|30|66|47|46|38|81|11|27|79|89| 4|35|
Sorted array  :  4| 6|11|19|19|21|26|27|30|33|35|38|42|44|46|47|47|49|51|57|66|68|75|78|78|79|79|81|86|89|
```

Code is provided in zip file.

Q4: OpenMP Parallelization Directives

In the parallel region with firstprivate(x,y):

1. Each thread gets x=0, y=0 initially
2. First loop (without #pragma omp for):
   - Every thread runs full loop 0-3
   - Each thread's x becomes 6 (0+1+2+3)
3. Second loop (with #pragma omp for):
   - Loop iterations split between threads
   - With 2 threads:
     - Thread 0: y=0+1=1 (iterations 0,1)
     - Thread 1: y=2+3=5 (iterations 2,3)
4. Thread output ordering is non-deterministic, but x=6 and y values are consistent for each thread.

Example outputs:

Output 1	Output 2
Thread: 1	Thread: 0
Thread 1 running first loop, result: 6	Thread 0 running first loop, result: 6
Thread: 0	Thread: 1
Thread 0 running first loop, result: 6	Thread 1 running first loop, result: 6
Thread 0 running second loop, result: 1	Thread 1 running second loop, result: 5
Thread 1 running second loop, result: 5	Thread 0 running second loop, result: 1

Q5: OpenMP Programming and Schedules

a)

```
--- Varying Number of Threads with Default Chunk Size ---
Schedule Type | Threads | Execution Time (s)
-------------------------------------------
static        | 2       | 1.353249
static        | 4       | 0.722480
static        | 8       | 0.632304
static        | 16      | 0.551274
dynamic       | 2       | 1.420666
dynamic       | 4       | 0.726865
dynamic       | 8       | 0.679831
dynamic       | 16      | 0.536516
guided        | 2       | 1.352896
guided        | 4       | 0.705340
guided        | 8       | 0.562456
guided        | 16      | 0.527915
```

Analysis:

- **Static Scheduling:** Demonstrates excellent scalability with increasing threads due to minimal scheduling overhead. Execution time decreases proportionally with the number of threads.
- **Dynamic Scheduling:** Slightly higher execution times compared to static, especially noticeable with more threads, due to the overhead of dynamic task assignment. However, it can handle load imbalances better.
- **Guided Scheduling:** Performance is close to static but with a bit more overhead. It dynamically adjusts the chunk size to balance load without incurring as much overhead as dynamic scheduling.

b)

```
--- Varying Chunk Sizes with 16 Threads ---
Schedule Type | Chunk Size | Execution Time (s)
-------------------------------------------
static        | 4          | 0.597636
static        | 16         | 0.600463
static        | 32         | 0.597309
static        | 64         | 0.588905
dynamic       | 4          | 0.503568
dynamic       | 16         | 0.542204
dynamic       | 32         | 0.533397
dynamic       | 64         | 0.540412
guided        | 4          | 0.520597
guided        | 16         | 0.503837
guided        | 32         | 0.516343
guided        | 64         | 0.507435
```

Analysis:

- **Static Scheduling:** Changing the chunk size has minimal impact on execution time once the chunk size is sufficiently large. Smaller chunk sizes might introduce slight overhead but are generally negligible.
- **Dynamic Scheduling:** Performance slightly degrades with increasing chunk sizes due to less optimal load balancing. Smaller chunk sizes can help in better distributing the work but at the cost of increased scheduling overhead.
- **Guided Scheduling:** Remains consistent across different chunk sizes, indicating efficient load balancing regardless of chunk size. It adjusts the chunk size dynamically to balance the load without significant overhead.

Ömer Öztürk
S046798

c)

In theory, dynamic scheduling benefits from smaller chunk sizes by enhancing load balancing, ensuring that all threads remain active even if some iterations take longer, but this comes at the cost of increased scheduling overhead, which can negate performance gains. Conversely, guided scheduling is inherently adaptive, starting with larger chunks and progressively decreasing their size, which maintains good load balancing with reduced overhead compared to dynamic scheduling. This adaptability makes guided scheduling less sensitive to the exact chunk size, allowing it to efficiently manage workload distribution without the fine-tuning required by dynamic scheduling.

d)

Different OpenMP scheduling policies perform optimally in distinct scenarios based on workload distribution and system characteristics. Static scheduling assigns fixed chunks of iterations to threads beforehand, making it ideal for tasks with uniform workloads like matrix multiplication, as it minimizes overhead and maximizes cache efficiency. Dynamic scheduling allocates iterations to threads on-the-fly, which is beneficial for irregular or unpredictable workloads where some tasks may take longer, ensuring better load balancing despite higher scheduling overhead. Guided scheduling combines the advantages of both by starting with larger chunks that decrease in size, adapting to workload variations while reducing overhead compared to dynamic scheduling. This makes guided scheduling suitable for scenarios with mixed workload intensities, providing a balanced approach to resource utilization and performance.