Assignment 2: OpenMP

Nirmit Zinzuwadia | 1002434074

Charvi Choksi | 1006952317

Shwetha Padmanabhan | 1007541242

Improvements Added

The serial solution cannot be parallelized as it is. We had to apply certain code transformations to parallelize the logic. From the serial solution we know that the value of A[1024] depends on A[0], A[1025] depends on A[1] and so on. From this we can conclude that the "dependence distance" is 1024 and can run only a chunk of this size in parallel at a time. Thus, we move the iteration count to an outer loop to aid parallel processing.

1. **Parallel Solution 1**

   With inner loop parallelization, we transform the elements from index 1024 in chunks. The iteration count starts with value 1 and ignores the 0th iteration since the first set of 1024 elements aren't transformed. We execute chunks of size 1024 in parallel (Example, index 1024 to 2047 are executed in parallel during the first iteration). The process goes on until it reaches the end of iteration. This parallel execution with code transformation seems to reduce the processing time by more than a half when compared to the sequential solution.

```
void parallelSolution_1(int *A),
{
for (int i = 1; i < ITER; ++i)
   #pragma omp parallel for
   for (int n = i * STRIDE; n < (i+1) * STRIDE; ++n)
         A[n] = transform(A[n - STRIDE]);
}
```

2. **Parallel solution 2**

   Since parallelizing the inner loop alone is not ideal, we transformed the code to parallelize the outer loop. When we iterate through STRIDE in the first loop and ITER in the interloop, the elements are transformed in the order 1024, 2048… during the first iteration and 1025,2049... in the second iteration and so on.

   Due to the above discussed execution pattern, the same thread jumps through the array and causes a high cache miss rate, resulting in poor cache locality. The enormous amount of cache misses causes the execution time of this approach to be worse than that of the sequential approach.

```
void parallelSolution_2(int *A),
{
 #pragma omp parallel for
 for (int i = 0; i < STRIDE; i++) {
      for (int n = 0; n < ITER-1; n++) {
              A[i + (n+1)*STRIDE] = transform(A[i + n*STRIDE]);
      }
   }
}
```

3. **Parallel Solution 3**

In order to address the high cache miss rate of the above scenario, we further transform the code such that dependencies are removed or reduced, large pieces of code can be executed in parallel with known loop bounds.

In this approach, the elements are transformed in the order 1024, 1025… 2047 during the first iteration 2048, 2049... 2071 in the second iteration and so on. Since the adjacent elements are executed one after the other, we maintain the intent of the code with low synchronization and good cache locality.

```
void parallelSolution_3(int *A),
{
   #pragma omp parallel for
   for (int i = 1; i < ITER; ++i) {
      int idx = i * STRIDE;
      for (int n = 0; n < STRIDE; ++n) {
         A[idx + n] = transform(A[n]);
      }
   }
}
```

4. **Parallel solution 4 (Best Execution Time)**

In this section we tried to experiment with various schedulers and variable sharing mechanisms provided by openMP. We used dynamic scheduling (chunk size of 16) with private variable declaration used by 4 threads.

We used dynamic scheduling because, though we know each iteration of the thread is going to perform the same operation, its execution time might vary depending upon the value on which transformation is being applied. We chose 16 as batch size to maintain a good cache locality. And since the CPU has only 4 cores, we went with 4 threads. This approach performed the best among all the strategies that we previously implemented.

```
void parallelSolution_4(int *A)
{
    int idx;
    #pragma omp parallel for schedule(dynamic,16) private(idx) num_threads(4)
    for (int i = 1; i < ITER; ++i) {
        idx = i * STRIDE;
        for (int n = 0; n < STRIDE; ++n) {
            A[idx + n] = transform(A[n]);
        }
    }
}
```

Performance Comparison

Solution Type	**Execution Time(seconds)**
Serial	8.401317
Parallel Solution 1	3.323311
Parallel Solution 2	7.512820
Parallel Solution 3	2.945508
Parallel Solution 4	2.691166