Malay Pandey
CS 610A

Assignment III

Roll No: 22111090
October 10, 2023

KD Workstation System Configuration

- **System Number:** CSEWS3
- **CPU Max MHz:** 4600
- **CPU Min MHz:** 800
- **L1-d Cache:** 192 KiB (6 instances) = 32 KiB (1 instance)
- `getconf LEVEL1_DCACHE_LINESIZE`: 64 (Line Size is 64 bytes)

Problem 1

```
#define N (1 << 12)

double x[N], y[N], z[N], A[N][N]
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        y[j] = y[j] + A[i][j] * x[i];
        z[j] = z[j] + A[j][i] * x[i];
    }
}
```

The double data type occupies 8 B, and we've set the value of N to be 2^{12}, which implies that the 1D arrays **x, y, & z** have a size of 2^{12+3} B, equivalent to 2^{15} B or 32 KiB each. In contrast, the 2D array **A** has a size of 2^{12*2+3} B, which equals 2^{27} B or 128 MB. Modern computers typically have enough capacity in L1 cache to hold the entire **x, y, & z** arrays in memory, so we need not be concerned about cache misses for these arrays. However, the 2D array **A** is too large to fit entirely within the cache, so optimizing its access will require making efficient use of spatial and temporal locality of reference.

Within the loop indexed by j, we are accessing the elements of the array A[i][j]. This access pattern exhibits spatial locality of reference because, on modern computers with a typical cache line size of 64 bytes, a single cache miss would load 8 consecutive array elements into the cache. When we update the elements of the array z, we attempt to load elements from the array A in a column-major access pattern (A[j][i]). This would result in cache misses at each iteration because it doesn't take advantage of the spatial locality of reference. Since array A is quite large, it means that for the next iteration of the outer loop indexed by i, the cache will likely not contain the necessary elements from array A, further leading to cache misses and potentially impacting performance negatively.

Assignment III

We observe that statements 1 and 2 within the inner loop (indexed by j) perform updates to distinct memory locations, indicating that we don't have any dependence between both statements. Furthermore, there exists a **self loop-independent anti-dependence** for both statements (read before write for y and z). Consequently, we have the opportunity to optimize by **vectorizing** the innermost loop. However, vectorization alone may not yield the desired speedup due to the unresolved issue of the column-major access pattern in statement 2.

Given that both statements update separate memory locations, we can use loop fission to split the loops into two distinct nested for loops. Each of these nested loops will be responsible for updating the variables y and z, respectively. This approach can help enhance performance by mitigating the impact of the column-major access pattern in statement 2, since now we can change the order of the loops in this case. Furthermore, we can use `#pragma ivdep` to vectorize the innermost loop in both the cases. We can also use loop tiling for both loops (since there is no carried dependence, loop tiling is valid), to better utilize spatial and temporal locality of reference.

Please note that in the tables mentioned below, the version Optimized(AVX2 *) utilized the intrinsics version where tiling hasn't been applied, but it did provide the most speedup amongst all possible versions.

```
#define N (1 << 12)

double x[N], y[N], z[N], A[N][N]
for (int i = 0; i < N; i++) {
    #pragma ivdep
    for (int j = 0; j < N; j++) {
        y[j] = y[j] + A[i][j] * x[i];
    }
}

for (int j = 0; j < N; j++) {
    #pragma ivdep
    for (int i = 0; i < N; i++) {
        // Column major access turned to row major access
        z[j] = z[j] + A[j][i] * x[i];
    }
}
```

For optimization level: -O2, the best speedup was provided with **AVX2 vectorization** without any tiling, while for optimization level: -O3, the best speedup was provided with `#pragma ivdep` **Optimized version**, with tiling of **Block Size = 4**.

Version	**B = 4**	**B = 8**	**B = 16**
Reference	0.69019 sec	0.71582 sec	0.70874 sec
Optimized	0.07536 sec	0.09392 sec	0.09708 sec
Optimized(AVX2)	0.11028 sec	0.07621 sec	0.07898 sec
Optimized(AVX2 *)	**0.06157 sec**		

Optimization Level: -O2

Version	**B = 4**	**B = 8**	**B = 16**
Reference	0.41100 sec	0.39304 sec	0.37382 sec
Optimized	**0.05099 sec**	0.08102 sec	0.09032 sec
Optimized(AVX2)	0.11557 sec	0.07992 sec	0.07632 sec
Optimized(AVX2 *)	0.05963 sec		

Optimization Level: -O3

Command for running the code:

```
g++ -O2 -mavx -mavx2 -fopenmp -march=native 22111090-prob1.cpp
-o prob1
./prob1
```

Problem 2

a

In order to implement a version of OpenMP parallelism without reductions, we can either use `#pragma omp atomic` to store the values directly into the shared variable (seq_sum), or compute local sums for each thread, followed by the aggregation of these individual sums to obtain the total sum for the array. Consequently, it is necessary to declare an array with a size of `num_threads` to accommodate these local sums. Additionally, it is crucial to determine the appropriate padding size to prevent false sharing, as failing to do so can hinder the effective utilization of parallel programming advantages. The return type of the sum functions is `uint64_t`, representing an 8-byte or 64-bit integer. To ascertain the cache line size, the following command can be utilized: `getconf LEVEL1_DCACHE_LINESIZE`. In most modern systems, the cache line size is 64 bytes, hence, it maybe better to pad the local thread array with $\frac{64-8}{8} = 7$ more elements, to mitigate false sharing issues.

b

To implement OpenMP parallelism with reductions, one straightforward approach is to utilize the compiler directive `#pragma omp for reduction(+: sum)`. This directive informs the compiler that the variable `sum` is shared among threads, and the threads should automatically create their own necessary local variables, along with code generation to combine all these local variables.

c

In the context of implementing OpenMP task-based parallelism, the objective is to compute the sum of elements within a specified array range denoted as `[l, r]`, where l represents the leftmost boundary, and r signifies the rightmost boundary of the segment. To achieve this, we can store the intermediate task results within an array, with the length of this array corresponding to the number of concurrently generated threads. Subsequently, these individual task results are combined to yield the ultimate output.

To ensure that each task operates on a sub-array of fixed size 1024 (GRANULARITY), static scheduling can be utilized. The array is partitioned into uniform segments, and subsequently, the tasks individually choose and operate on distinct sub-portions of these segments. The selection of these sub-portions adheres to the granularity level (GRANULARITY).

Version	**Time**
Sequential	0.0116972 sec
Parallel (atomic)	0.690538 sec
Parallel (thread-local)	0.0045018 sec
Parallel (work-sharing)	**0.0044803 sec**
Parallel (OpenMP tasks)	0.0058528 sec

The Parallel version using reductions performed the best, and the worst performance using `#pragma omp atomic`, since we are putting locks for every array index, which means we are essentially carrying out sequential sum on the original array, in some jumbled fashion, and introducing overhead using locks as well, which leads to performance dip.

Command for running the code:

```
g++ -O2 -fopenmp 22111090-prob2.cpp -o prob2
./prob2
```

Problem 3

To implement an inclusive parallel prefix algorithm, we can split the array into small blocks, calculate local prefix sums, and then do a second pass where we adjust the local values by adding the sum computed in the previous block. We perform $\mathbf{\log n}$ iterations, where the k^{th} iteration makes sure that the i^{th} element a_i is equal to the sum of the segment $[i-2^k, i]$. Since we have max $k = \mathbf{\log n}$, this means that at the last iteration, we'll have the inclusive prefix sum for the entire array (reference).

Version	**Time** (on N = 2^{15})
Serial	55
OMP	37
SSE	**20**
AVX2	124
AVX512	28

The SSE version performs the best, maybe due to the fact that the implementation for SSE took only one pass, while AVX2 and AVX512 on the other hand took multiple passes (2 passes) to compute the inclusive prefix sum of the entire array.

Command for running the code:

```
g++ -msse4 -mavx2 -mavx512f -march=native -O3
-fopt-info-vec-optimized -fopt-info-vec-missed -o prob3 prob3.cpp
./prob3
```

Problem 4

The critical observation in this context is that due to the sorted nature of the iteration space $r_1, r_2, \ldots, r_{10}$, the values $x_1, x_2, \ldots, x_{10}$ are also stored in sorted order in the results file. Consequently, it is possible to initially store these values in a data structure such as a set, vector, array, etc., without sorting them during the parallel implementation. After the parallel execution is completed, the results will remain unchanged, allowing us to sort these values afterward to achieve the desired sorted order in the results.

It's important to observe that while an individual loop may not be particularly large on its own, the cumulative effect of having multiple loops with almost identical limits results in a substantial iteration space. Hence, when utilizing a parallel region, it becomes imperative to identify an optimal balance where employing the `#pragma omp parallel for collapse(_)` directive does not result in a performance decline. This is particularly critical because incorporating the expressions for x_k within the inner loops effectively increases the number of floating-point operations. Dynamic

scheduling was chosen as the preferred approach because it offered the fastest speedup, primarily due to its ability to maintain a proper workload balance among the various threads. Furthermore, the highest level of performance was achieved when collapsing three for loops into one.

- **Approach I**: Utilise `#pragma omp critical` to save all the favorable points in the vector dictionary, sort the dictionary, and then save all the files to `results.txt`
- **Approach II**: Given the huge iteration space, we can store favorable points for each thread within individual local thread vectors, rather than using `#pragma omp critical`. Subsequently, these results can be aggregated and combined from all the thread vectors into the output file `results.txt`. An essential consideration in this context is the need to prevent false sharing. To achieve this, it's necessary to apply padding to the vectors for each local thread. In our case, we have a collection of 10 double values to be saved, with each double consuming 8 bytes of memory. This means that each individual favorable data point occupies 80 bytes. Given the cache line size of 64 bytes, it is advisable to incorporate padding into our array. Specifically, adding an extra $\frac{64\times2-80}{8} = 6$ elements will help us effectively avoid false sharing issues and optimize memory access patterns.

Version	**Time**
Original	380 sec
OMP (Approach I)	**67.53 sec**
OMP (Approach II)	70.28 sec
Best Speedup	$\sim$ 5.62

Commands for running the files are mentioned at the top of each file.

References

- Algorithmica - Prefix Sum with SIMD
- ADMS2020

End.