

## Assignment 1

**Q1. DAXPY Loop:** D stands for Double precision, A is a scalar value, X and Y are one-dimensional vectors of size  $2^{16}$  each, P stands for Plus. The operation to be completed in one iteration is  $X[i] = a*X[i] + Y[i]$ . Your task is to compare the speedup (in execution time) gained by increasing the number of threads. Start from a 2 thread implementation. How many threads give the max speedup? What happens if no. of threads are increased beyond this point? Why?

Sol.

```
1 #include <stdio.h>
2 #include <omp.h>
3
4 #define N [1<<24]
5
6 int main() {
7     double X[N], Y[N], a = 2.5;
8     int i;
9
10    for (i = 0; i < N; i++) {
11        X[i] = i * 1.0;
12        Y[i] = i * 2.0;
13    }
14
15    double start = omp_get_wtime();
16
17    #pragma omp parallel for
18    for (i = 0; i < N; i++) {
19        X[i] = a * X[i] + Y[i];
20    }
21
22    double end = omp_get_wtime();
23    printf("Execution time: %f seconds\n", end - start);
24
25    return 0;
26 }
27
```

Output:

- nitin@DESKTOP-EL941AC:~/UCS645/Assignment\_1\$ OMP\_NUM\_THREADS=2 ./Q1  
Execution time: 0.000200 seconds
- nitin@DESKTOP-EL941AC:~/UCS645/Assignment\_1\$ OMP\_NUM\_THREADS=8 ./Q1  
Execution time: 0.000373 seconds
- nitin@DESKTOP-EL941AC:~/UCS645/Assignment\_1\$ OMP\_NUM\_THREADS=18 ./Q1  
Execution time: 0.001103 seconds
- nitin@DESKTOP-EL941AC:~/UCS645/Assignment\_1\$ OMP\_NUM\_THREADS=13 ./Q1  
Execution time: 0.001009 seconds

No. of Threads	Exec. Time(s)	Speed-up
2	0.000200	1.00x
8	0.000373	0.54x
13	0.001009	0.20x
18	0.001103	0.18x

**Q: How many threads give the maximum speedup?**

The maximum speedup is obtained when the number of threads is equal to the number of physical CPU cores, as all cores are efficiently utilized.

**Q: What happens if the number of threads is increased beyond this point?**

If the number of threads is increased further, the execution time stops improving and may even increase.

**Q: Why does this happen?**

This happens because of memory bandwidth limitations, cache conflicts, and overhead caused by managing too many threads. The DAXPY operation is memory-bound, not computation-bound.

**Q2. Matrix Multiply :** Build a parallel implementation of multiplication of large matrices (Eg. size could be 1000x1000). Repeat the experiment from the previous question for this implementation. Think about how to partition the work amongst the threads – which elements of the product array will be calculated by each thread? Implement 2 version of parallel implementation for given task. First, use 1D threading(i.e., make a single loop run in parallel). Second, use 2D threading (i.e., use nested looping to the most suitable loops to be parallelized)

Sol.

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <omp.h>
4
5  #define N 1000    // Matrix size (NxN)
6
7  double A[N][N], B[N][N], C[N][N];
8
9  /* Initialize matrices */
10 void initialize() {
11     for (int i = 0; i < N; i++) {
12         for (int j = 0; j < N; j++) {
13             A[i][j] = 1.0;
14             B[i][j] = 1.0;
15             C[i][j] = 0.0;
16         }
17     }
18 }
19
20 /* Clear result matrix */
21 void clear_C() {
22     for (int i = 0; i < N; i++)
23         for (int j = 0; j < N; j++)
24             C[i][j] = 0.0;
25 }
26

// Version 1: 1D Threading

void matmul_1D() {
    #pragma omp parallel for
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            for (int k = 0; k < N; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

// Version 2: 2D Threading

void matmul_2D() {
    #pragma omp parallel for collapse(2)
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            for (int k = 0; k < N; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

```

```

int main() {
    double start, end;

    initialize();

    // 1D Threading
    clear_C();
    start = omp_get_wtime();
    matmul_1D();
    end = omp_get_wtime();
    printf("1D Threading Time: %f seconds\n", end - start);

    //2D Threading
    clear_C();
    start = omp_get_wtime();
    matmul_2D();
    end = omp_get_wtime();
    printf("2D Threading Time: %f seconds\n", end - start);

    return 0;
}

```

## Output:

```

● nitin@DESKTOP-EL941AC:~/UCS645/Assignment_1$ OMP_NUM_THREADS=2 ./Q2
 1D Threading Time: 1.541063 seconds
 2D Threading Time: 1.891216 seconds
● nitin@DESKTOP-EL941AC:~/UCS645/Assignment_1$ OMP_NUM_THREADS=6 ./Q2
 1D Threading Time: 1.021492 seconds
 2D Threading Time: 0.962172 seconds
● nitin@DESKTOP-EL941AC:~/UCS645/Assignment_1$ OMP_NUM_THREADS=17 ./Q2
 1D Threading Time: 0.568293 seconds
 2D Threading Time: 0.587222 seconds

```

### Version 1: 1D Threading

In the first version, only the outer loop (row index `i`) is parallelized. Each thread is responsible for computing a subset of rows of the result matrix CCC. This approach is simple, has low overhead, and provides good cache locality since each thread accesses contiguous memory locations.

#### **Observation:**

Execution time decreases as the number of threads increases, showing good speed-up up to the number of available cores.

### Version 2: 2D Threading

In the second version, two nested loops (`i` and `j`) are parallelized using OpenMP's `collapse(2)` clause. This distributes individual elements of matrix CCC across threads, providing finer-grained parallelism and better load balancing, especially when a large number of threads is used.

#### **Observation:**

2D threading shows better performance at higher thread counts due to improved work distribution, but it also introduces slightly higher overhead and cache contention.

## Conclusion

Both 1D and 2D parallel implementations significantly improve the performance of matrix multiplication. 1D threading is simpler and efficient for moderate thread counts, while 2D threading provides better scalability on systems with many cores. However, maximum speed-up is limited by memory access patterns and hardware constraints.

**Q3. Calculation of  $\pi$ :** This is the first example where many threads will cooperate to calculate a computational value. The task of this program will be arrive at an approximate value of  $\pi$ . The value of  $\pi$  is given by the following integral:

$$\pi = \int_0^1 \frac{4.0}{1+x^2} dx$$

**Sol.**

```
1 #include <stdio.h>
2 #include <omp.h>
3
4 static long num_steps = 100000000;
5
6 int main() {
7     double step = 1.0 / (double) num_steps;
8     double sum = 0.0, pi;
9     int i;
10
11 #pragma omp parallel for private(i) reduction(+:sum)
12 for (i = 0; i < num_steps; i++) {
13     double x = (i + 0.5) * step;
14     sum += 4.0 / (1.0 + x * x);
15 }
16
17 pi = step * sum;
18 printf("Estimated Pi = %.15f\n", pi);
19
20 return 0;
21 }
22 }
```

**Output:**

```
● nitin@DESKTOP-EL941AC:~/UCS645/Assignment_1$ OMP_NUM_THREADS=2 ./Q3
Estimated Pi = 3.141592653589910
● nitin@DESKTOP-EL941AC:~/UCS645/Assignment_1$ OMP_NUM_THREADS=6 ./Q3
Estimated Pi = 3.141592653589646
● nitin@DESKTOP-EL941AC:~/UCS645/Assignment_1$ OMP_NUM_THREADS=15 ./Q3
Estimated Pi = 3.141592653589801
```

### **Observation**

The computed value of  $\pi$  is very close to the actual value (3.141592653589793) for all thread counts. Small variations in the last few decimal places occur due to floating-point arithmetic and the order in which partial sums from different threads are combined.

### **Conclusion**

Parallel computation using OpenMP accurately estimates the value of  $\pi$  while improving performance. The use of the `reduction` clause ensures correct results by avoiding race conditions, and increasing the number of threads does not affect accuracy but improves execution efficiency.