# Parallel and Distributed Computing

## Assignment 1

**Submitted to: Dr Saif Nalband**
**Submitted by: Anushka Gupta**
**Roll No:102303358**
**Course: UCS645**

## Question 1: DAXPY Loop

Problem Statement:

DAXPY Loop: D stands for Double precision, A is a scalar value, X and Y are one-dimensional vectors of size 2^16 each, and P stands for Plus. The operation performed in one iteration is:

X[i] = a × X[i] + Y [i]

The objective is to compare the speedup (in execution time) gained by increasing the number of threads starting from a 2-thread implementation.

### Implementation

The DAXPY loop was implemented using OpenMP with a parallel for directive. Execution time was measured using `omp_get_wtime()`.

[5] ✓ 0s

```
%%writefile daxpy.c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

#define N 50000000

int main()
{
    double *X, *Y, a = 2.5;
    double start, end;

    X = (double*)malloc(N * sizeof(double));
    Y = (double*)malloc(N * sizeof(double));

    for(int i=0;i<N;i++)
    {
        X[i] = i;
        Y[i] = i;
    }

    start = omp_get_wtime();

    #pragma omp parallel for
    for(int i=0;i<N;i++)
    {
        X[i] = a*X[i] + Y[i];
    }

    end = omp_get_wtime();

    printf("Time: %f seconds\n", end-start);

    return 0;
}
```

## Experimental Results:

[6] ✓ 0s

```
!gcc daxpy.c -fopenmp -O2 -o daxpy
```

[7] ✓ 1s

```
!OMP_NUM_THREADS=2 ./daxpy
!OMP_NUM_THREADS=4 ./daxpy
```

```
Time: 0.055876 seconds
Time: 0.087043 seconds
```

### Observation

The execution time increased when the number of threads was increased from 2 to 4.
This happened due to thread creation overhead and limited CPU resources in Google Colab.

The DAXPY loop is memory-bound and executes very fast; hence, overhead dominates computation.

### Conclusion

Best performance was achieved using fewer threads. Increasing threads beyond this point reduced performance.

# Question 2: Matrix Multiplication

### Problem Statement

To implement parallel matrix multiplication of large matrices using:

- 1D threading
- 2D threading

and compare their performance.

### Implementation

Matrix multiplication was implemented using triple nested loops.
Parallelisation was done using OpenMP `parallel for` and `collapse(2)`.

```
%%writefile mat1d.c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

#define N 600

int main()
{
    static double A[N][N], B[N][N], C[N][N];
    double start, end;

    for(int i=0;i<N;i++)
        for(int j=0;j<N;j++)
        {
            A[i][j]=1;
            B[i][j]=1;
            C[i][j]=0;
        }

    start = omp_get_wtime();

    #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];

    end = omp_get_wtime();

    printf("Time: %f\n", end-start);

    return 0;
}
```

## Experimental Results (1D threading)

```
!gcc mat1d.c -fopenmp -O2 -o mat1d
```

```
!OMP_NUM_THREADS=2 ./mat1d
!OMP_NUM_THREADS=4 ./mat1d
```

```
Time: 0.313603
Time: 0.291012
```

```
%%writefile mat2d.c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

#define N 600

int main()
{
    static double A[N][N], B[N][N], C[N][N];
    double start, end;

    for(int i=0;i<N;i++)
        for(int j=0;j<N;j++)
        {
            A[i][j]=1;
            B[i][j]=1;
            C[i][j]=0;
        }

    start = omp_get_wtime();

    #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];

    end = omp_get_wtime();

    printf("Time: %f\n", end-start);

    return 0;
}
```

## Experimental Results (2D Threading)

```
!gcc mat2d.c -fopenmp -O2 -o mat2d
```

```
!OMP_NUM_THREADS=2 ./mat2d
!OMP_NUM_THREADS=4 ./mat2d
```

```
Time: 0.427361
Time: 0.289436
```

### Observation

In 1D implementation, each thread computes a subset of rows, resulting in low overhead.

In 2D implementation, work is distributed more evenly but scheduling overhead increases.

Cache effects and memory bandwidth limit perfect scaling.

### Conclusion

Both 1D and 2D implementations show limited speedup due to hardware constraints.
Moderate improvement is observed with increased threads.

## Question 3: Calculation of π

### Problem Statement

The value of π is computed using numerical integration:

$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx$$

The rectangle method is used and parallelized using OpenMP reduction.

### Implementation

Parallelization was done using `reduction(+:sum)` to avoid race conditions.

```
%%writefile pi.c
#include <stdio.h>
#include <omp.h>

#define STEPS 100000000

int main()
{
    double step = 1.0/STEPS;
    double sum = 0.0, pi;
    double start, end;

    start = omp_get_wtime();

    #pragma omp parallel for reduction(+:sum)
    for(int i=0;i<STEPS;i++)
    {
        double x = (i+0.5)*step;
        sum += 4.0/(1.0+x*x);
    }

    pi = step * sum;

    end = omp_get_wtime();

    printf("PI = %f\n", pi);
    printf("Time = %f\n", end-start);

    return 0;
}
```

## Experimental Results

```
!gcc pi.c -fopenmp -O2 -o pi
```

```
!OMP_NUM_THREADS=2 ./pi
!OMP_NUM_THREADS=4 ./pi
```

```
PI = 3.141593
Time = 0.215071
PI = 3.141593
Time = 0.217215
```

### Observation

Speedup was very small because the sequential execution time was already low.
Synchronisation and reduction of overhead affected performance.

### Conclusion

Parallel computation of π is beneficial only up to a limited number of threads.
Beyond that, overhead dominates.

## Overall Conclusion

OpenMP provides an easy way to implement parallel programs.
However, performance improvement depends on hardware capability and workload size.

Using too many threads can reduce performance due to synchronisation, memory contention, and scheduling overhead.