Parallel & Distributed Computing

Lab Assignment -1

Saksham Mittal-102303135

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?

Code:

```
C Ques1.c ×
Ubuntu > home > saksham > parallel-assignments > Lab1 > C Ques1.c
#include<stdio.h>
#include<omp.h>
#include<math.h>

int main(){
    int n=65536;
    double a=10;
    double x[n],y[n];
    double st,et;
    for(int i=0;i<n;i++){
        x[i]=i;
        y[i]=i;
    }
    st=omp_get_wtime();
    for(int i=0;i<n;i++){
        x[i]=a*x[i]+y[i];
    }
    et=omp_get_wtime();
    double seq_duration=et-st;
    printf("Sequential execution time: %f\n",seq_duration);
    for(int threads=2;threads<=20;threads++){
        for(int i=0;i<n;i++){
            x[i]=i;
        }
        omp_set_num_threads(threads);
        st=omp_get_wtime();
        #pragma omp parallel for
        for(int i=0;i<n;i++){
            x[i]=a*x[i]+y[i];
        }
        et=omp_get_wtime();
        double par_duration=et-st;

        printf("Threads: %d  Duration:%f  Speedup:%f\n",threads,par_duration,seq_duration/par_duration);
    }
}
```

Output:

```
(base) saksham@Saksham:~/parallel-assignments/Lab1$ ./Ques1
Sequential execution time: 0.000146
Threads: 2  Duration:0.000176  Speedup:0.829161
Threads: 3  Duration:0.000187  Speedup:0.782727
Threads: 4  Duration:0.000137  Speedup:1.066238
Threads: 5  Duration:0.000323  Speedup:0.452891
Threads: 6  Duration:0.000080  Speedup:1.817881
Threads: 7  Duration:0.006920  Speedup:0.021114
Threads: 8  Duration:0.007945  Speedup:0.018390
Threads: 9  Duration:0.000414  Speedup:0.353122
Threads: 10  Duration:0.000419  Speedup:0.348640
Threads: 11  Duration:0.000310  Speedup:0.470790
Threads: 12  Duration:0.000241  Speedup:0.606091
Threads: 13  Duration:0.000169  Speedup:0.865430
Threads: 14  Duration:0.000500  Speedup:0.292149
Threads: 15  Duration:0.000458  Speedup:0.319259
Threads: 16  Duration:0.000621  Speedup:0.235453
Threads: 17  Duration:0.000293  Speedup:0.499027
Threads: 18  Duration:0.000314  Speedup:0.465552
Threads: 19  Duration:0.000587  Speedup:0.249074
Threads: 20  Duration:0.000328  Speedup:0.445117
```

Observations:

The speedup follows an irregular pattern and actual decrease in execution time is obtained only at threads= 4 and threads=6 where speedup>1. For the rest of the parallel executions, speedup < 1 which means that the sequential execution time is less than the time taken to execute code parallelly.

Q2) Matrix Multiplication

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)

Code:

C Ques1.c   C Ques2.c ×

Ubuntu > home > saksham > parallel-assignments > Lab1 > C Ques2.c

```c
#include <stdio.h>
#include <omp.h>

double A[1000][1000], B[1000][1000], C[1000][1000];

int main()
{
    int N=1000;


    int i, j, k;
    double st, et, seq_duration;

    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            A[i][j] = i;
            B[i][j] = i*2;
            C[i][j] = 0.0;
        }
    }

    printf("Sequential:\n");

    st = omp_get_wtime();

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

    et = omp_get_wtime();
    seq_duration = et - st;

    printf("Sequential Time = %f\n", seq_duration);

```

Ques1.c Ques2.c

Ubuntu > home > saksham > parallel-assignments > Lab1 > Ques2.c

```c

    printf("\n1D Parallel:\n");

    for (int threads = 2; threads <= 20; threads++) {

        double par_duration;

        omp_set_num_threads(threads);

        /* reset C */
        for (i = 0; i < N; i++)
            for (j = 0; j < N; j++)
                C[i][j] = 0.0;

        st = omp_get_wtime();

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

        et = omp_get_wtime();
        par_duration = et - st;
        printf("Threads=%d  Time=%f  Speedup=%.2f\n", threads, par_duration, seq_duration / par_duration);
    }
```

Ques1.c Ques2.c

Ubuntu > home > saksham > parallel-assignments > Lab1 > Ques2.c

```c

    printf("\n2D Parallel:\n");

    for (int threads = 2; threads <= 20; threads++) {

        double par_duration;

        omp_set_num_threads(threads);

        for (i = 0; i < N; i++)
            for (j = 0; j < N; j++)
                C[i][j] = 0.0;

        st = omp_get_wtime();

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

        et = omp_get_wtime();
        par_duration = et - st;
        printf("Threads=%d  Time=%f  Speedup=%.2f\n", threads, par_duration, seq_duration/par_duration);
    }

    return 0;
}
```

Output:

```
(base) saksham@Saksham:~/parallel-assignments/Lab1$ gcc Ques2.c -fopenmp -o Ques2
(base) saksham@Saksham:~/parallel-assignments/Lab1$ ./Ques2
Sequential:
Sequential Time = 2.606031

1D Parallel:
Threads=2  Time=1.361691  Speedup=1.91
Threads=3  Time=1.029791  Speedup=2.53
Threads=4  Time=0.926276  Speedup=2.81
Threads=5  Time=0.883619  Speedup=2.95
Threads=6  Time=0.780665  Speedup=3.34
Threads=7  Time=0.745771  Speedup=3.49
Threads=8  Time=0.644200  Speedup=4.05
Threads=9  Time=0.684581  Speedup=3.81
Threads=10  Time=0.689898  Speedup=3.78
Threads=11  Time=0.659996  Speedup=3.95
Threads=12  Time=0.661578  Speedup=3.94
Threads=13  Time=0.697833  Speedup=3.73
Threads=14  Time=0.675871  Speedup=3.86
Threads=15  Time=0.664709  Speedup=3.92
Threads=16  Time=0.645827  Speedup=4.04
Threads=17  Time=0.702229  Speedup=3.71
Threads=18  Time=0.667410  Speedup=3.90
Threads=19  Time=0.640276  Speedup=4.07
Threads=20  Time=0.607424  Speedup=4.29
```

```
2D Parallel:
Threads=2  Time=1.362602  Speedup=1.91
Threads=3  Time=1.150884  Speedup=2.26
Threads=4  Time=0.903615  Speedup=2.88
Threads=5  Time=0.850063  Speedup=3.07
Threads=6  Time=0.782795  Speedup=3.33
Threads=7  Time=0.756267  Speedup=3.45
Threads=8  Time=0.659305  Speedup=3.95
Threads=9  Time=0.673681  Speedup=3.87
Threads=10  Time=0.669165  Speedup=3.89
Threads=11  Time=0.667870  Speedup=3.90
Threads=12  Time=0.629386  Speedup=4.14
Threads=13  Time=0.676703  Speedup=3.85
Threads=14  Time=0.672839  Speedup=3.87
Threads=15  Time=0.649911  Speedup=4.01
Threads=16  Time=0.629467  Speedup=4.14
Threads=17  Time=0.679035  Speedup=3.84
Threads=18  Time=0.637407  Speedup=4.09
Threads=19  Time=0.645831  Speedup=4.04
Threads=20  Time=0.662997  Speedup=3.93
(base) saksham@Saksham:~/parallel-assignments/Lab1$ |
```

Observations:

For 1D parallel program, the speedup increases till threads=8 and stays almost stable between 3.80-4.00. Whereas, for 2D parallel program, the speedup increases till threads=12 and then stabilizes at 4.00. Important thing to note is that in 2D parallel program, the speedup is overall better than the 1D parallel program but the maximum speedup is achieved in 1D parallel program with 20 threads, speedup=4.29.

Q3) Calculation of Pi

The task of this program will be arrive at an approximate value of π. The value of π is given by the following integral:

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

This can be approximated as the sum of the areas of rectangles under the curve.

Code:

```c
#include <stdio.h>
#include <omp.h>

int main() {
    int NUM_STEPS=100000;
    double step = 1.0 / (double)NUM_STEPS;
    double x, pi, sum;
    double st, et, seq_duration;

    sum = 0.0;

    st = omp_get_wtime();

    for (int i = 0; i < NUM_STEPS; i++) {
        x = (i + 0.5) * step;
        sum += 4.0 / (1.0 + x * x);
    }

    pi = step * sum;

    et = omp_get_wtime();
    seq_duration = et - st;

    printf("Sequential Result\n");
    printf("Pi = %.10f\n", pi);
    printf("Time = %f sec\n\n", seq_duration);

    printf("Parallel Results\n");

    for (int threads = 2; threads <= 20; threads++) {

        double par_duration, speedup;
        sum = 0.0;

        omp_set_num_threads(threads);

        st = omp_get_wtime();

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

        pi = step * sum;

        et = omp_get_wtime();
        par_duration = et - st;

        printf("Threads = %d    Pi = %.10f    Time = %f    Speedup = %.2f\n",threads, pi, par_duration, seq_duration / par_duration);
    }

    return 0;
}
```

Output:

```
(base) saksham@Saksham:~/parallel-assignments/Lab1$ gcc Ques3.c -fopenmp -O3
(base) saksham@Saksham:~/parallel-assignments/Lab1$ ./Ques3
Sequential Result
Pi = 3.1415926536
Time = 0.000173 sec

Parallel Results
Threads = 2   Pi = 3.1415926536   Time = 0.000191   Speedup = 0.91
Threads = 3   Pi = 3.1415926536   Time = 0.000288   Speedup = 0.60
Threads = 4   Pi = 3.1415926536   Time = 0.000189   Speedup = 0.92
Threads = 5   Pi = 3.1415926536   Time = 0.000144   Speedup = 1.21
Threads = 6   Pi = 3.1415926536   Time = 0.000143   Speedup = 1.22
Threads = 7   Pi = 3.1415926536   Time = 0.000107   Speedup = 1.62
Threads = 8   Pi = 3.1415926536   Time = 0.000123   Speedup = 1.41
Threads = 9   Pi = 3.1415926536   Time = 0.000166   Speedup = 1.04
Threads = 10   Pi = 3.1415926536   Time = 0.000237   Speedup = 0.73
Threads = 11   Pi = 3.1415926536   Time = 0.000385   Speedup = 0.45
Threads = 12   Pi = 3.1415926536   Time = 0.000417   Speedup = 0.42
Threads = 13   Pi = 3.1415926536   Time = 0.000297   Speedup = 0.58
Threads = 14   Pi = 3.1415926536   Time = 0.000490   Speedup = 0.35
Threads = 15   Pi = 3.1415926536   Time = 0.000517   Speedup = 0.34
Threads = 16   Pi = 3.1415926536   Time = 0.000515   Speedup = 0.34
Threads = 17   Pi = 3.1415926536   Time = 0.000529   Speedup = 0.33
Threads = 18   Pi = 3.1415926536   Time = 0.000502   Speedup = 0.35
Threads = 19   Pi = 3.1415926536   Time = 0.000708   Speedup = 0.24
Threads = 20   Pi = 3.1415926536   Time = 0.000484   Speedup = 0.36
```

Observations:

The speedup is negligible as the time taken to perform sequentially is already very less(0.000173 secs). This indicates that parallelism does not necessarily improve performance. This happens because the problem complexity is very less and small problems do not benefit from parallelism.