PD_10_Kushagra Suryawanshi

HPC Lab Assignment No: 4

Title: Two large vectors Addition/Subtraction/Multiplication

Aim: Design and implement a parallel algorithm to add/subtract/multiply two large vectors using C programming language.

Objective:

1.Write sequential vector addition/subtraction/multiplication program.

2. Calculate complexity of the program.

3. Measure time taken by the sequential program.

4. Write program for Parallel vector addition/subtraction/multiplication

5.Measure time taken by the parallel program.

6. Measure and compare time taken by the parallel vs serial program.

Theory:

1. **Write about the serial vector addition/subtraction/multiplication.**

ANS:

- Vector addition can be done by sequentially accessing the elements of both the

  large vectors and adding them and store result in result vector(array).

- The following v_add function inside v_add.c performs an element-by-element

  addition of two vectors x and y and places the result in a third vector z;

```
void v_add(double* x, double* y, double* z)
{
    for(int i=0; i<ARRAY_SIZE; i++)
        z[i] = x[i] + y[i];
}
```

2. **Write about the parallel vector addition/subtraction/multiplication.**

ANS:

- Vector addition is inherently a very parallel computation.
- The v_add function inside v_add.c performs an element-by-element addition of two vectors x and y and places the result in a third vector z parallely , as shown below:

```
void v_add(double* x, double* y, double* z)
{
  # pragma omp parallel
   {
    for(int i=0; i<ARRAY_SIZE; i++)
        z[i] = x[i] + y[i];
    }//end_pragma section
}
```

3. **Write which constructs of OPENMP are used for parallel vector addition, subtraction, multiplication.**

ANS:

The ‘for’ construct of openMP is used for parallel openMP addition.

```
#pragma omp parallel for
    for(i=0; i<n; i++)
    {
        c[i] = a[i]+b[i];
        printf("Thread %d works on element%d\n", omp_get_thread_num(), i);
    }
```

Test on data set of sufficiently large size.

Compute Total cost and Efficiency as: -

Total Cost = Time complexity × Number of processors used

Efficiency = Execution time of sequential algorithm/Execution time of the parallel algorithm

Efficiency = 0.76

Mention the number of processors / processor cores of your machine: 8

Consider data points as follows: -

Sr no	Data points/Data values for vectors	Time Taken for serial approach	Time Taken for parallel approach
1	500	0.000002	0.0042441
2	1000	0.000007	0.002412
3	10000	0.000091	0.001008
4	30000	0.000253	0.000915
5	50000	0.000416	0.001801

6	70000	0.000572	0.001650
7	100000	0.000719	0.000946

Auto Generate the data points/values through :-

a) Giving a specific pattern of data value generation

b) Randomizing the data points/values

(Also a file may be maintained for reading the values stored)

Input: Two large vectors.

Output: Addition/multiplication/subtraction of two large vectors.

Platform: Windows

Conclusion: Thus, successfully studied, analyzed addition/subtraction/multiplication of two

large vectors.

FAQs:

1. What is the Complexity of Strassen's Matrix Multiplication?

ANS: Time complexity of Strassen’s Matrix Multiplication is:

O(n^log2(7)) = O(n^2.81)

2. What do you understand by Speedup and Efficiency?

ANS: The speedup is defined as the ratio of the serial runtime of the best sequential algorithm for solving a problem to the time taken by the parallel algorithm to solve the same problem on p processors. The efficiency is defined as the ratio of speedup to the number of processors.

Code:

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

#define NUM_THREADS 4

double addVector(double *a, double *b, double *ans, int n)
{
    double start = omp_get_wtime();
    for(int i=0; i<n; i++)
    {
        ans[i] = a[i] + b[i];
    }

    double end = omp_get_wtime();
    printf("Time Taken for Serial : %f\n", end-start);
    double T1 = end-start;
    return T1;
}

double addVectorPar(double *a, double *b, double *ans, int n)
{
    double start = omp_get_wtime();

    #pragma omp parallel for
    for(int i=0; i<n; i++)
    {
        ans[i] = a[i] + b[i];
        //printf("Addition%lf + %lf = %lf performed by thread %d \n", a[i], b[i],
ans[i], omp_get_thread_num());
    }

    double end = omp_get_wtime();
    printf("Time Taken for Parallel : %f\n", end-start);
    double Tp = end-start;
    return Tp;
}

int main()
{
    int n = 70000000;
    printf("\nFor vector of size %d\n",n);
    double *array1 = (double *) malloc(sizeof(double) * n), *array2 = (double *)
malloc(sizeof(double) * n), *array3 = (double *) malloc(sizeof(double) * n), *array4
= (double *) malloc(sizeof(double) * n);

    for (int i=0; i<n; i++)
    {
        array1[i] = rand() % 1000;
```

```
        array2[i] = rand() % 1000;
    }

    omp_set_num_threads(NUM_THREADS);
    // addVector(array1, array2, array3, n);
    // addVectorPar(array1, array2, array4, n);
    double a = addVector(array1, array2, array3, n);
    double b = addVectorPar(array1, array2, array4, n);
    double cores = omp_get_num_procs();
    double speedup = a/b;
    double efficiency = (1/cores)*speedup;
    printf("Speedup = %f",speedup);
    printf("\nEfficiency = %lf",efficiency);

}
```

Output:

```
For vector of size 100000000
Time Taken for Serial : 1.362000
Time Taken for Parallel : 1.228000
Speedup = 1.109121
Efficiency = 0.138640
PS D:\HPC> gcc -fopenmp -o vector hpcvector.c
PS D:\HPC> ./vector.exe

For vector of size 50000000
Time Taken for Serial : 0.157000
Time Taken for Parallel : 0.079000
Speedup = 1.987343
Efficiency = 0.248418
PS D:\HPC> gcc -fopenmp -o vector hpcvector.c
PS D:\HPC> ./vector.exe

For vector of size 70000000
Time Taken for Serial : 0.254000
Time Taken for Parallel : 0.129000
Speedup = 1.968989
Efficiency = 0.246124
PS D:\HPC>
```