PARALLEL AND DISTRIBUTED COMPUTING

Assignment 2

SAMEET ASADULLAH
SECTION D
18I-0479

TASK-1

(a)

Loop 1: Parallelizable (No Dependencies)

Distance Vector = (0)

Direction Vector = (=)

Loop 2: Not Parallelizable (Loop carried dependency + True/Flow Dependency)

Distance Vector = (1)

Direction Vector = (<)

Loop 3: Parallelizable (Loop carried dependency + Output Dependency)

Note: There is an if condition in this loop which can be considered as a control flow dependency but with few transformations, we can parallelize it.

Distance Vector = (4)

Direction Vector = (<)

(b)

```
#include <iostream>
#include <omp.h>
using namespace std;
#define N 1024

int main() {
    int i, k = 10;
    int a[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    int c[1000];
    int b[N][N];
    int loc = -1;
    int tmp = -1;

    #pragma omp parallel for schedule(static, 1)
        for(i = 0; i < k; i++)
            b[i][k] = b[a[i]][k];

    printf("%d %d", a[0], b[0][0]);

    for(i = 0; i < 1000; i++) {
        tmp = tmp + 1;
        c[i] = tmp;
    }

    #pragma omp parallel for schedule(static, 1)
        for(i = 0; i < 1000; i++) {
            if (c[i] % 4 == 0) {
                #pragma omp critical
                {
                    if (i > loc)
                        loc = i;
                }
            }
        }
    return 0;
}
```

(c)

Sequential Execution Time = 0.0225s

Parallel Execution Time (2) = 0.0195s

Parallel Execution Time (4) = 0.0197s

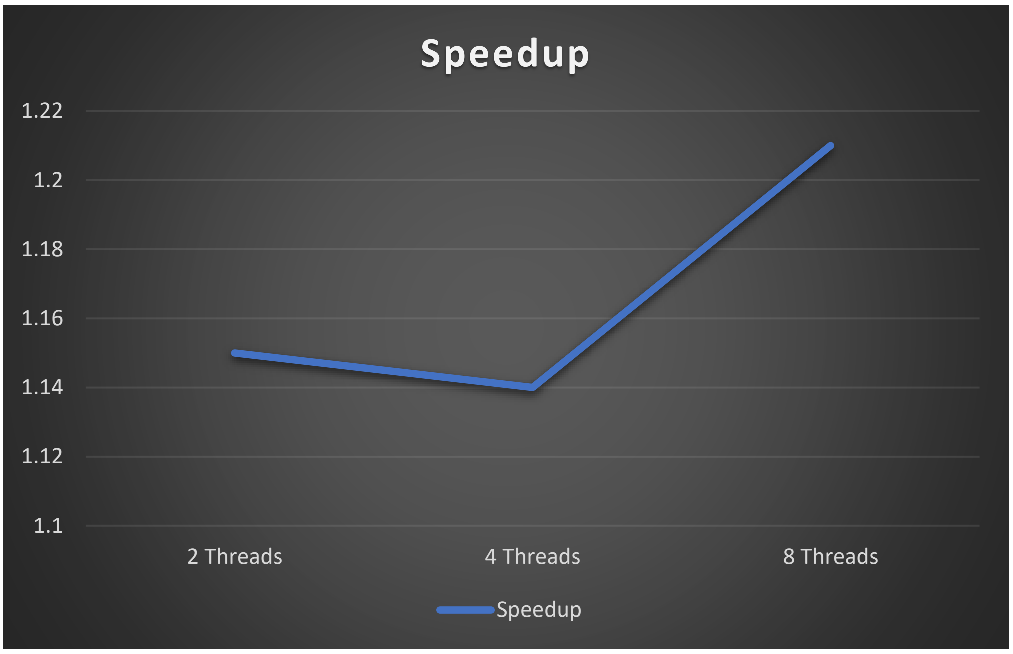
Parallel Execution Time (8) = 0.0185s

Note: Sequential Execution time and parallel Execution time is average of first six observations

Speedup (2) = Sequential Time/Parallel Time = 0.0225/0.0195 = 1.15

Speedup (4) = 0.0225/0.0197 = 1.14

Speedup (8) = 0.0225/0.0185 = 1.21



As there is a significant difference in execution time of sequential time and parallel time, which means that the performance of the code has been improved. The speedup using 2 and 4 threads is almost similar while it has been also increased significantly while using 8 threads.

TASK-2

(a)

Loop 1: Parallelizable (No Dependencies)

Distance Vector = (0, 0)

Direction Vector = (=, =)

Loop 2: Parallelizable (No Dependencies)

Distance Vector = (0, 0)

Direction Vector = (=, =)

(b)

```
#include <iostream>
#include <omp.h>
using namespace std;
#define N 800

int function_call(int j) {
    int a;
    a = 2 * 2 + j;
    return a;
}

int main() {
    int i, j;
    int a[N][N];
    int b[N][N];
    int c[N][N];

    #pragma omp parallel for private(i, j), schedule(static, 1), collapse(2)
        for(i = 1; i <= N; i++)
                for(j = 0; j < N; j++)
                    b[i - 1][j] = function_call(j);

    #pragma omp parallel for private(i, j), schedule(static, 1), collapse(2)
        for(j = 0; j < N - 10; j++)
            for(i = 0; i < N-10; i++) {
                a[i][j + 2] = b[i + 2][j];
                c[i + 1][j] = b[i][j + 3];
            }
    return 0;
}
```

(c)

Sequential Execution Time = 0.0315s

Parallel Execution Time (2) = 0.0313s

Parallel Execution Time (4) = 0.0280s

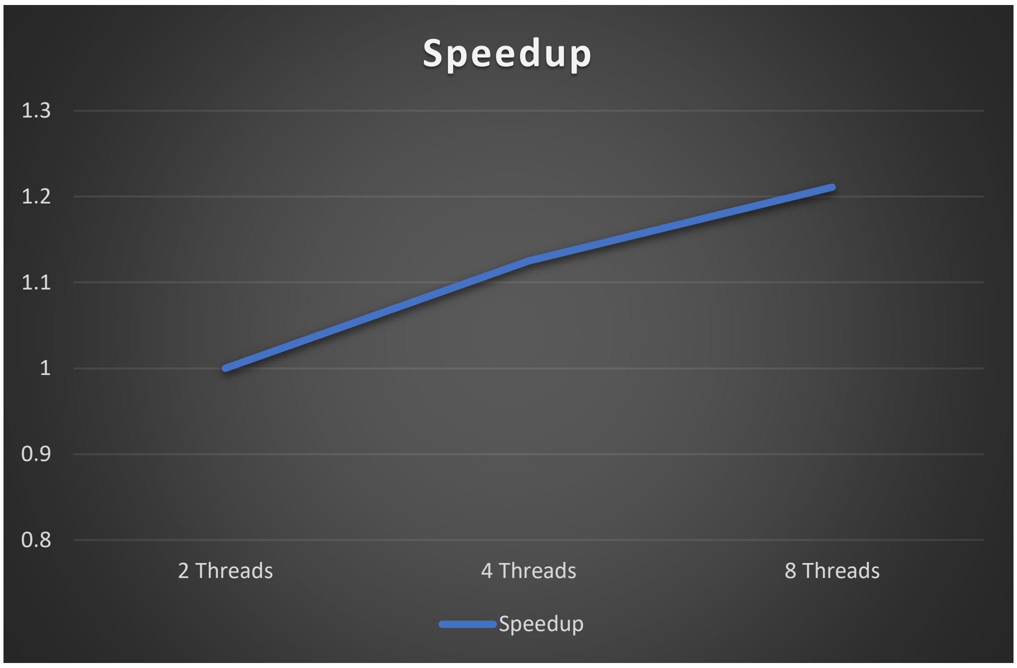
Parallel Execution Time (8) = 0.0260s

Note: Sequential Execution time and parallel Execution time is average of first six observations

Speedup (2) = Sequential Time/Parallel Time = 0.0315/0.0313 = 1.00

Speedup (4) = 0.0315/0.0280 = 1.125

Speedup (8) = 0.0315/0.0260 = 1.211



In this case, the sequential execution time and parallel execution time incase of two threads is almost same so there’s not much performance achieved but if we increase number of threads then speedup also increases and as well as performance.

TASK-3

(a)

Loop 1: Outer Loop Not Parallelizable (Loop Carried Dependency + True/Flow Dependency)

Inner Loop Parallelizable (No Dependencies)

Distance Vector = (1, 0)

Direction Vector = (<, =)

Loop 2: Parallelizable (No Dependencies)

Distance Vector = (0, 0)

Direction Vector = (=, =)

(b)

```cpp
#include <iostream>
#include <omp.h>
using namespace std;
#define N 1024

int main()
{
    int i, j;
    int X[N][N];
    int Y[N];
    int Z[N];
    int k = 1;

    for(i = 0; i < N; i++){
        Y[i] = k;
        k = k * 2;
        Z[i] = -1;
        #pragma omp parallel for schedule(static, 1)
            for(j = 0; j < N; j++)
                X[i][j] = 2;
    }

    #pragma omp parallel for private(i, j), schedule(static, 1), collapse(2)
        for(i = 0; i < N; i++)
            for(j = 0; j < N; j++) {
                Z[i] += Y[j] + X[i][j];
            }
    return 0;
}
```

(c)

Sequential Execution Time = 0.0245s

Parallel Execution Time (2) = 0.0301s

Parallel Execution Time (4) = 0.0260s

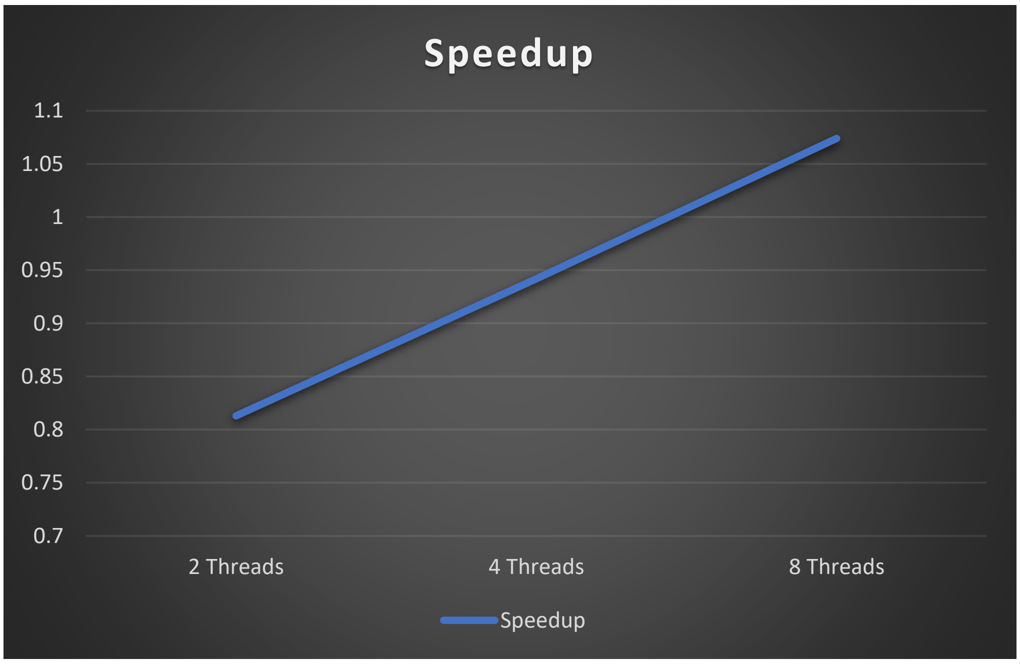
Parallel Execution Time (8) = 0.0228s

Note: Sequential Execution time and parallel Execution time is average of first six observations

Speedup (2) = Sequential Time/Parallel Time = 0.0245/0.0301 = 0.813

Speedup (4) = 0.0245/0.0260 = 0.942

Speedup (8) = 0.0245/0.0228 = 1.074



In this task, overall speedup is increased using two, four and eight threads but the execution time of sequential code is less than the execution time of parallel code using two and four threads, that means no performance has been improved but incase of eight threads, the execution time is less than the sequential execution time, that means performance is only going to improve if we use eight threads otherwise the sequential code is better than the parallelized code using two and four threads.

TASK-4

(a)

Original Code:

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

int main() {
    int m = 4, n = 16384;
    int i, j;
    double a[m][n];
    double s[m];

    #pragma omp parallel for private(i, j), shared(s, a)
        for(i = 0; i < m; i++) {
            s[i] = 1.0;
            for(j = 0; j < n; j++)
                s[i] = s[i] + a[i][j];
        }
    printf("%f", s[1]);
}
```

Improved Code:

```
#include <iostream>
#include <omp.h>
using namespace std;

int main() {
    int m = 4, n = 16384;
    int i, j;
    double a[m][n];
    double s[m];

    #pragma omp parallel for schedule(static, 1)
        for(i = 0; i < m; ++i) {
            s[i] = 1.0;
        }

    #pragma omp parallel for private(i,j), shared(s,a), schedule(static, 1), collapse(2)
        for(i = 0; i < m; i++) {
            for(j = 0; j < n; j++){
                s[i] = s[i] + a[i][j];
            }
        }
    printf("%f", s[1]);
}
```

(b)

Sequential Execution Time = 0.0138s

Original Code - Parallel Execution Time (1) = 0.0120s

Original Code - Parallel Execution Time (2) = 0.0113s

Original Code - Parallel Execution Time (4) = 0.0121s

Original Code - Parallel Execution Time (8) = 0.0126s

Improved Code - Parallel Execution Time (1) = 0.010s

Improved Code - Parallel Execution Time (2) = 0.0111s

Improved Code - Parallel Execution Time (4) = 0.011s

Improved Code - Parallel Execution Time (8) = 0.011s

Note: Sequential Execution time and parallel Execution time is average of first six observations

Original Code - Speedup (1) = Sequential Time/Parallel Time = 0.0138/0.0120 = 1.15

Original Code - Speedup (2) = 0.0138/0.0113 = 1.22

Original Code - Speedup (4) = 0.0138/0.0121 = 1.14

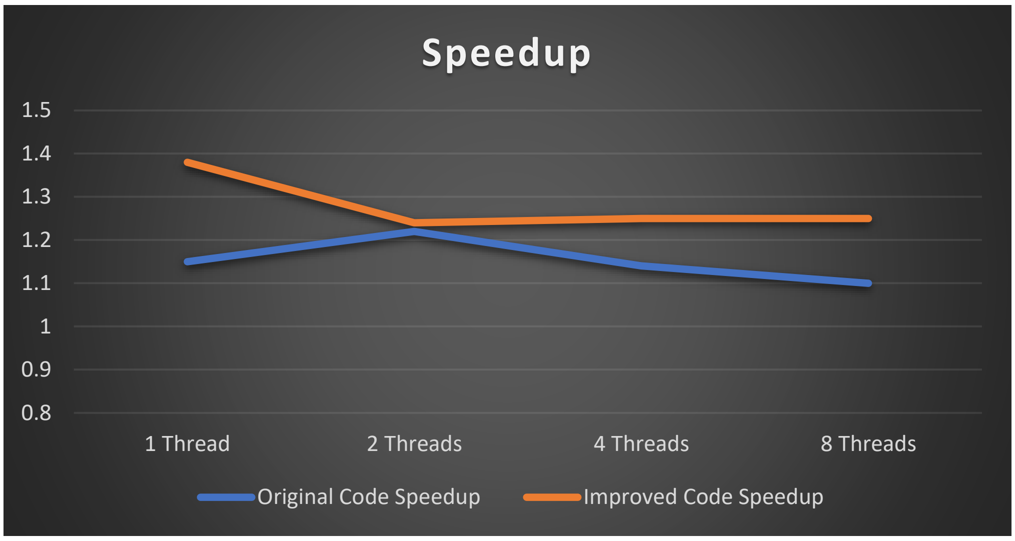
Original Code - Speedup (8) = 0.0138/0.0126 = 1.10

Improved Code - Speedup (1) = 0.0138/0.010 = 1.38

Improved Code - Speedup (2) = 0.0138/0.0111 = 1.24

Improved Code - Speedup (4) = 0.0138/0.011 = 1.25

Improved Code - Speedup (8) = 0.0138/0.011 = 1.25



The speedup of the improved code is greater than the original code while in case of two threads it’s almost same. Execution time of sequential code is greater than the execution time of parallel code so the performance has been improved.