ELEC 470 Computer System Architecture Machine Problem

John Turnbull – 20235355 – 2024/04/10

Part 1:

a)

```
//John Turnbull
//20235355

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

// We will be changing these variables frequently to evaluate the functionality
#define N 100
#define NUM_THREADS 16

double A[N], B[N];
double dot_product = 0;

// Sequential dot product function
double sequential_dot_product() {
    double sum = 0;
    int i;
    for (i = 0; i < N; i++) {
        sum += A[i] * B[i];
    }
    return sum;
}

int main() {
    int i;
    double start, end;

    // Initialize input vectors
    for (i = 0; i < N; i++) {
        A[i] = 1.0;
        B[i] = 2.0;
    }

    // Start execution time
    start = omp_get_wtime();

    // Compute dot product using parallel for with a reduction, also set number of threads
    #pragma omp parallel for reduction(+:dot_product) num_threads(NUM_THREADS)
    for (i = 0; i < N; i++) {
        dot_product += A[i] * B[i];
    }

    // End execution time
    end = omp_get_wtime();

    // Output execution time excluding sequential
    printf("Parallel execution time: %f seconds\n", end - start);

    // Compute dot product sequentially
    double sequential_result;
    sequential_result = sequential_dot_product();

    // Output the results of the dot product
    printf("Parallel DP: %f\n", dot_product);
    printf("Sequential DP: %f\n", sequential_result);

    // Check if results match, output to the console accordingly
    if (dot_product == sequential_result)
        printf("Correct results\n");
    else
        printf("Incorrect results\n");

    return 0;
}
```

```
[20jjt@ece-para-knl ~]$ ./dotprod_high
Parallel execution time: 0.001573 seconds
Parallel DP: 200.000000
Sequential DP: 200.000000
Correct results
```

This code computes the dot product of two vectors using both sequential and parallel methods. It initializes the vectors, then employs OpenMP directives for parallel computation, utilizing a **parallel for loop with a reduction** operation to sum contributions from each thread. The number of threads used, and the input vector will be altered depending on our experimental needs. After computation, it compares the results from both methods and outputs the execution times and results.

b)

Without EPC:

```
//John Turnbull
//20235355

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

// We will be changing these variables frequently to evaluate the functionality
#define ROWS 10
#define COLS 10
#define NUM_THREADS 16

double A[ROWS][COLS], B[COLS], P_parallel[ROWS], P_sequential[ROWS];

// Initialization function for the input matrix and vector value
void initialize_input() {
    int i, j;
    for (i = 0; i < ROWS; i++) {
        for (j = 0; j < COLS; j++) {
            A[i][j] = 1.0;
        }
        // I might change these variables as we go through and evaluate the functionality in part 2
        B[i] = 2.0;
    }
}

// Sequential matrix-vector multiplication function
void sequential_matrix_vector_mult() {
    int i, j;
    for (i = 0; i < ROWS; i++) {
        double dot_product = 0.0;
        for (j = 0; j < COLS; j++) {
            dot_product += A[i][j] * B[j];
        }
        P_sequential[i] = dot_product;
    }
}

int main() {
    int i, j;
    double start, end;

    initialize_input();

    // Start execution time
    start = omp_get_wtime();

    // Compute matrix-vector multiplication using parallel for with a reduction, also set number of threads
    #pragma omp parallel for reduction(+:j) num_threads(NUM_THREADS)
    for (i = 0; i < ROWS; i++) {
        double dot_product = 0.0;
        for (j = 0; j < COLS; j++) {
            dot_product += A[i][j] * B[j];
        }
            P_parallel[i] = dot_product;
    }
```

```
    // End execution time
    end = omp_get_wtime();

    // Output execution time excluding sequential
    printf("Parallel execution time: %f seconds\n", end - start);

    // Compute matrix-vector multiplication sequentially
    sequential_matrix_vector_mult();

    // Compare results and output the mismatch index (if there are any)
    int mismatch_count = 0;
    for (i = 0; i < ROWS; i++) {
        if (P_parallel[i] != P_sequential[i]) {
            printf("Mismatch at index %d: parallel=%f, sequential=%f\n", i, P_parallel[i], P_sequential[i]);
            mismatch_count++;
        }
    }

    // Check if results match, output to the console accordingly
    if (mismatch_count == 0)
        printf("Correct results\n");
      else
        printf("Incorrect results. Total mismatches: %d\n", mismatch_count);

    return 0;
}
```

```
[20jjt@ece-para-knl ~]$ ./matrixmul
Parallel execution time: 0.001648 seconds
Correct results
```

This code performs matrix-vector multiplication using both sequential and parallel methods. It initializes the input matrix and vector, then utilizes OpenMP directives for parallel computation. Specifically, it employs **a parallel for loop** with a reduction operation to distribute the work across multiple threads. The number of threads used, and the number of rows/columns will be altered depending on our experimental needs. The computation involves calculating the dot product of each row of the matrix with the vector. After computation, it compares the results from both methods and outputs any mismatches found along with the execution times.

With EPC:

```
//John Turnbull
//20235355

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

// We will be changing these variables frequently to evaluate the functionality
#define ROWS 10000
#define COLS 10000
#define NUM_THREADS 16

double A[ROWS][COLS], B[COLS], P_parallel[ROWS], P_sequential[ROWS];

// Initialization function for the input matrix and vector value
void initialize_input() {
    int i, j;
    for (i = 0; i < ROWS; i++) {
        for (j = 0; j < COLS; j++) {
            A[i][j] = 1.0;
        }
        // I might change these variables as we go through and evaluate the functionality in part 2
        B[i] = 2.0;
    }
}

// Sequential matrix-vector multiplication function
void sequential_matrix_vector_mult() {
    int i, j;
    for (i = 0; i < ROWS; i++) {
        double dot_product = 0.0;
        for (j = 0; j < COLS; j++) {
            dot_product += A[i][j] * B[j];
        }
        P_sequential[i] = dot_product;
    }
}

void parallel_matrix_vector_mult(int start, int end) {
    int i, j;
    for (i = start; i < end; i++) {
        double dot_product = 0.0;
        for (j = 0; j < COLS; j++) {
            dot_product += A[i][j] * B[j];
        }
        P_parallel[i] = dot_product;
    }
}
```

```
int main() {
    int i, j;
    double start, end;

    initialize_input();

    // Start execution time
    start = omp_get_wtime();

    // Compute matrix-vector multiplication using parallel region, also set number of threads
    #pragma omp parallel num_threads(NUM_THREADS)
    {
        int thread_id = omp_get_thread_num();
        int chunk_size = ROWS / omp_get_num_threads();
        int start_thread = thread_id * chunk_size;
        int end_thread = (thread_id == omp_get_num_threads() - 1) ? ROWS : start_thread + chunk_size;
        parallel_matrix_vector_mult(start_thread, end_thread);
    }

    // End execution time
    end = omp_get_wtime();

    // Output execution time excluding sequential
    printf("Parallel execution time: %f seconds\n", end - start);

    // Compute matrix-vector multiplication sequentially
    sequential_matrix_vector_mult();

    // Compare results and output the mismatch index (if there are any)
    int mismatch_count = 0;
    for (i = 0; i < ROWS; i++) {
        if (P_parallel[i] != P_sequential[i]) {
            printf("Mismatch at index %d: parallel=%f, sequential=%f\n", i, P_parallel[i], P_sequential[i]);
            mismatch_count++;
        }
    }

    // Check if results match, output to the console accordingly
    if (mismatch_count == 0)
        printf("Correct results\n");
      else
        printf("Incorrect results. Total mismatches: %d\n", mismatch_count);

    return 0;
}
```

```
[20jjt@ece-para-knl ~]$ ./matrixmul_epc
Parallel execution time: 0.101689 seconds
Correct results
```

In this code, the "embarrassingly parallel computation" approach is employed within the **parallel region** where each thread individually computes one or more elements of the output vector P. Specifically, each thread is responsible for calculating the dot product of a subset of rows of the matrix with the vector. The number of threads used, and the number of rows/columns will be altered depending on our experimental needs. This design allows for efficient utilization of resources, as threads work independently on their assigned data segments, eliminating the need for communication or synchronization between them. After computation, it compares the results from both methods and outputs any mismatches found along with the execution times.

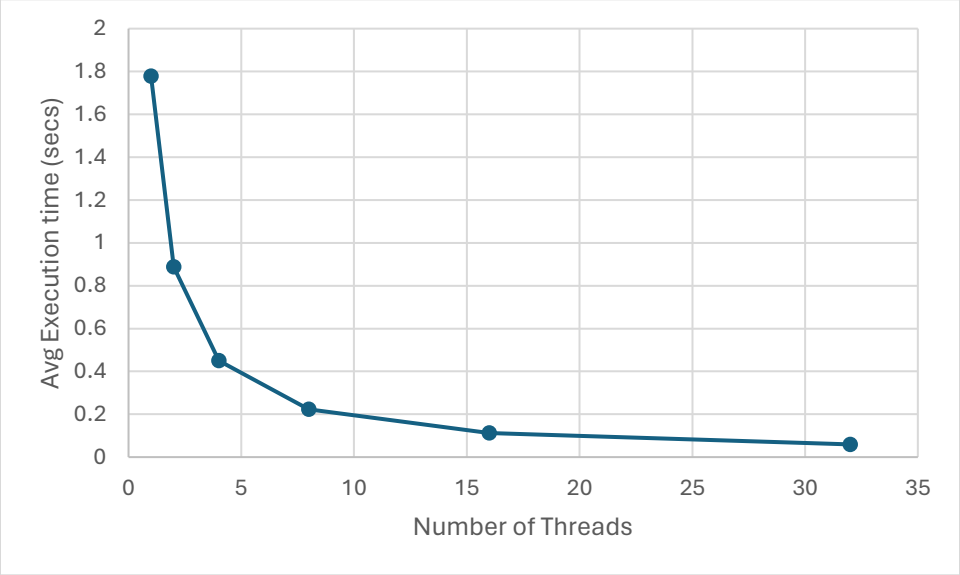
Part 2:

Strong Scaling Scenario

Dot Product:

Vector Size = 100000000

Number of Threads	Average Execution Time
1	1.7773
2	0.8883
4	0.4499
8	0.2234
16	0.1131
32	0.05936



The strong scaling scenario results for my dot product code shows a clear trend: as the number of threads increases for the dot product computation with a constant workload of 100,000,000 iterations, the average execution time decreases significantly. This reduction is nearly linear, indicating efficient parallelization through OpenMP.

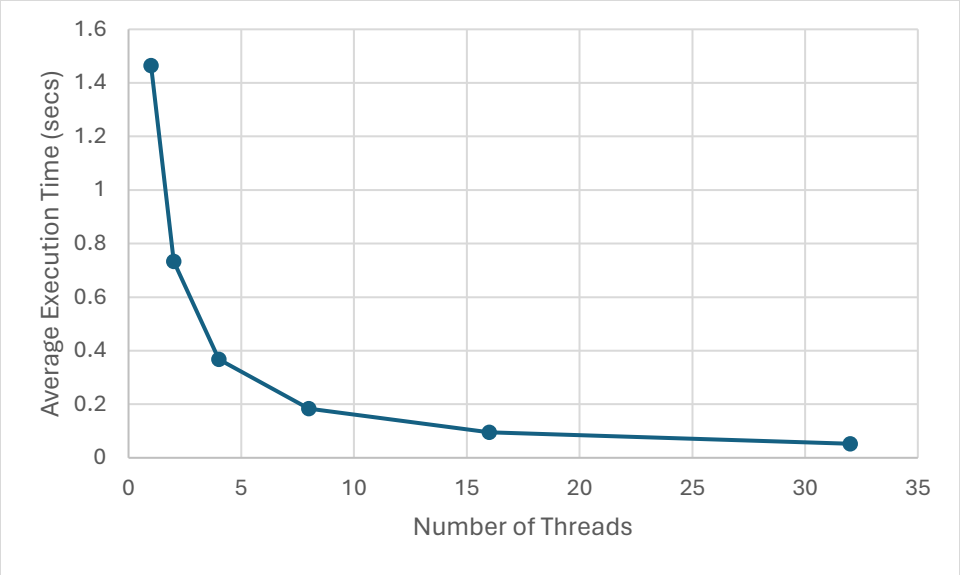
This improvement in performance can be explained by the division of the workload among multiple threads, allowing simultaneous computation of partial dot products. As each thread independently processes a portion of the data and contributes to the final result, the overall computation time is reduced.

Matrix-Vector Multiplication:

Number of Rows = 10000

Number of Columns = 10000

Number of Threads	Average Execution Time
1	1.4641
2	0.7328
4	0.3674
8	0.1841
16	0.0950
32	0.0524



For the matrix-vector multiplication scenario, the results exhibit a similar trend to the dot product computation. As the number of threads increases while maintaining a constant workload of 10,000 rows and 10,000 columns, the average execution time decreases significantly. This reduction follows a nearly linear pattern, showcasing effective parallelization through OpenMP.

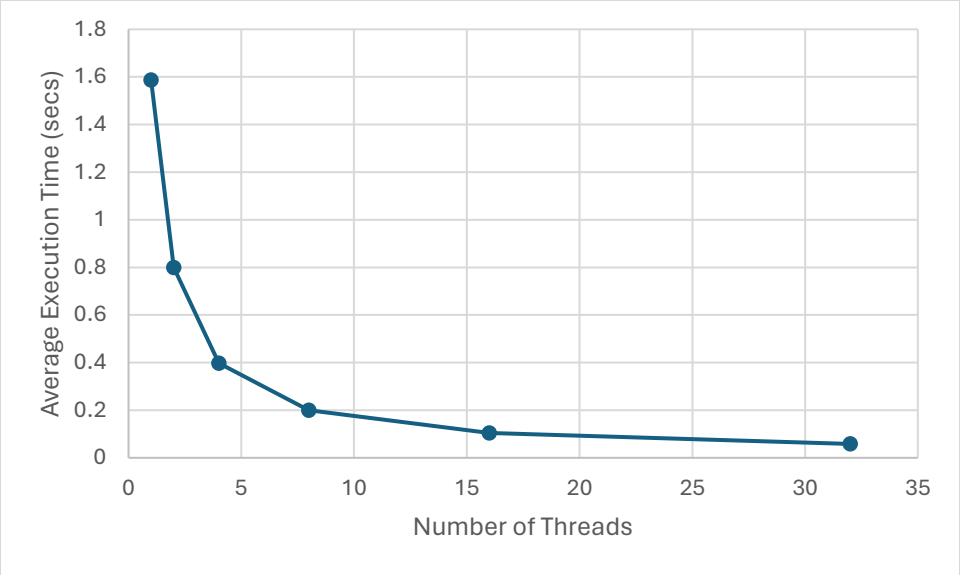
The observed performance enhancement in matrix-vector multiplication can be attributed to the parallel distribution of the workload across multiple threads. By dividing the matrix into rows and assigning each thread the task of computing the dot product of a specific row with the input vector, parallelization enables concurrent computation of these dot products. Consequently, each thread works independently on its allocated portion of the data, contributing to the final output vector. This collaborative effort leads to a reduction in the overall computation time, as the workload is effectively shared and processed simultaneously across multiple threads.

Matrix-Vector Multiplication (with embarrassingly parallel computation):

Number of Rows = 10000

Number of Columns = 10000

Number of Threads	Average Execution Time
1	1.5876
2	0.7994
4	0.3973
8	0.2000
16	0.1045
32	0.0582



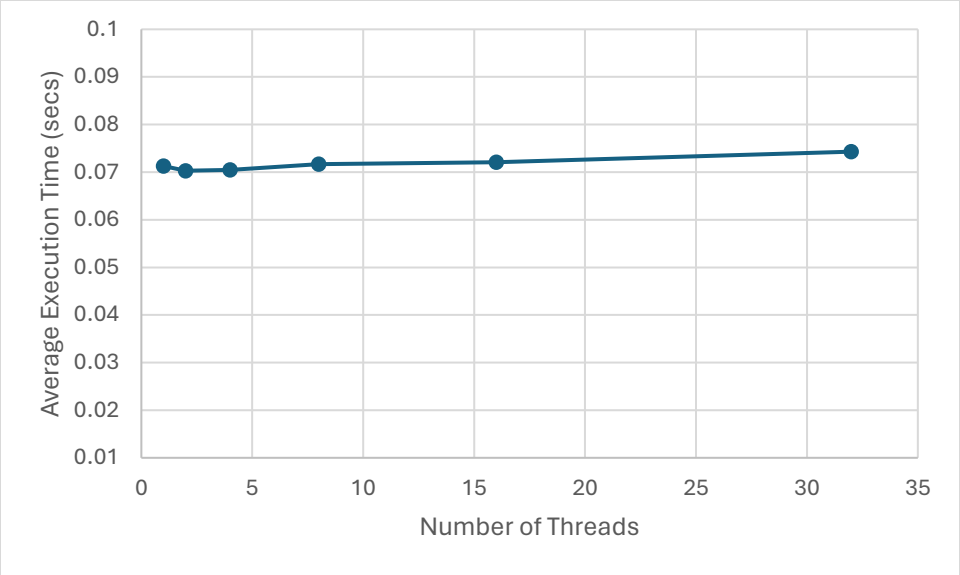
The performance gains in matrix-vector multiplication with embarrassingly parallel computation stem from efficient task allocation among threads. Each thread independently computes elements of the output vector, eliminating the need for communication or sharing. As thread count increases while maintaining a constant workload, execution time decreases significantly in a nearly linear fashion, showcasing scalability.

By assigning distinct tasks to each thread, parallelization facilitates simultaneous computation of multiple elements. Consequently, threads operate autonomously, collectively contributing to the final output matrix. This collaborative effort minimizes computation time, as the workload is efficiently processed in parallel across all threads, showcasing the effectiveness of parallelism without communication overhead.

Weak Scaling Scenario

Dot Product:

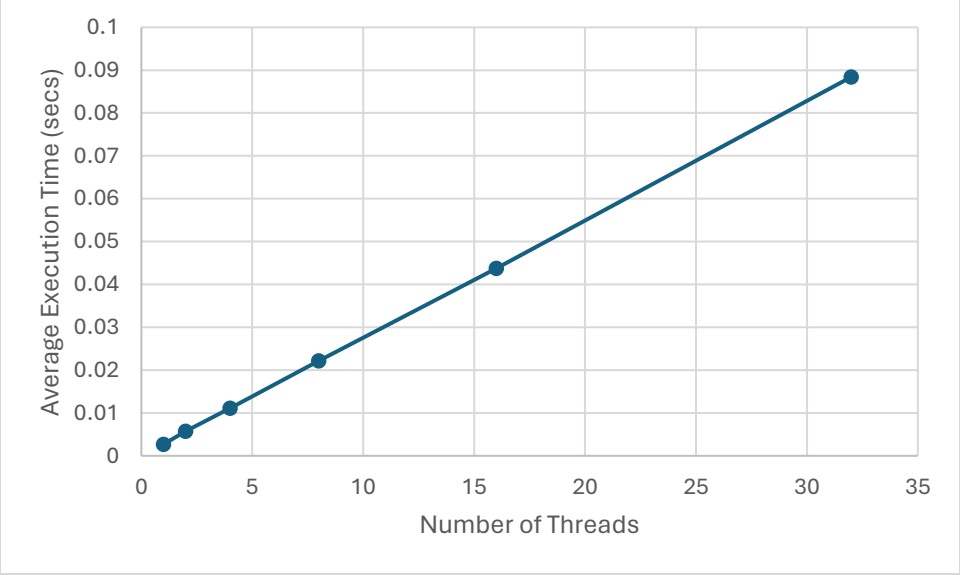
Number of Threads	Vector Size	Average Execution Time
1	4000000	0.0713
2	8000000	0.0703
4	16000000	0.0705
8	32000000	0.0717
16	64000000	0.0721
32	128000000	0.0743



As the number of threads increases from 1 to 32, the vector size proportionally grows, indicating a weak scaling scenario. Despite the increasing workload, the average execution time remains relatively stable, with minor fluctuations. This stability suggests effective load balancing and synchronization mechanisms within the OpenMP framework, ensuring optimal utilization of computational resources.

Matrix-Vector Multiplication:

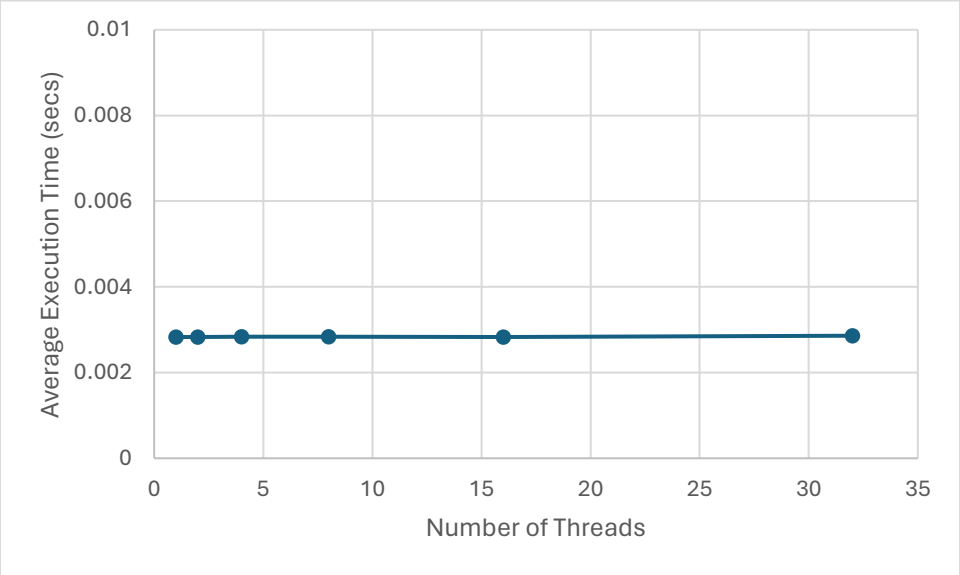
Number of Threads	Row and Column Size	Average Execution Time
1	400	0.00269
2	800	0.00573
4	1600	0.01112
8	3200	0.02217
16	6400	0.04375
32	12800	0.08841



The data for my matrix-vector multiplication operation reveals an intriguing trend in the performance of the OpenMP implementation across the weak scaling scenario. Contrary to the expected behavior, where increasing threads and matrix size proportionally typically leads to a stable execution time across the data, the results show an increasing linear relationship. This unexpected behavior suggests underlying inefficiencies in workload distribution, synchronization mechanisms, or resource utilization within the parallelization framework.

Matrix-Vector Multiplication (with embarrassingly parallel computation):

Number of Threads	Row and Column Size	Average Execution Time
1	400	0.00283
2	800	0.00283
4	1600	0.00284
8	3200	0.00284
16	6400	0.00283
32	12800	0.00286



The average execution time remains remarkably consistent across different proportional thread counts to matrix sizes. This uniformity suggests that the computational workload is efficiently distributed among threads and matrix sizes.

We do hereby verify that this simulation assignment report is our own work and contains our own original ideas, concepts, and designs. No portion of this report or code has been copied in whole or in part from another source, with the possible exception of properly referenced material.