KWAME NKRUMAH UNIVERSITY OF SCIENCE AND TECHNOLOGY, KUMASI

FACULTY OF PHYSICAL AND COMPUTATIONAL SCIENCES DEPARMENT OF MATHEMATICS

PROJECT REPORT: PART TWO(MPI)

Contents

Chapter 1

Introduction

1.1 Overview

This report presents the results, methodology, and implementation of a parallel programming project using MPI in C. The project focuses on vector-vector multiplication, where the workload is distributed efficiently across processes, regardless of whether the distribution is even or uneven.

1.2 Concepts

Vector-Vector Multiplication: This operation, also known as the dot product, computes a scalar value from two vectors. It is essential in various scientific and engineering computations.

Parallelization: To optimize performance on multi-core processors, the vector-vector multiplication was parallelized using MPI. The workload was distributed efficiently among processes, even when the distribution was not even.

Chapter 2

Algorithms

2.1 Mathematical Description

Given two vectors $A[i]$ and $B[i]$ of size N, the dot product is computed as:

$$R = \sum_{i=1}^{N} A[i] \times B[i]$$

The complexity of this operation is $O(N)$, where N is the length of the vectors.

2.2 Algorithm 1: Efficient Vector-Vector Multiplication Using MPI

```
1: Data: A[N], B[N]
2: Result: R ← 0
3: Split A and B into subarrays for each process
4: for each process do
5:     Calculate partial dot product for assigned subarray
6: end for
7: Use MPI_Reduce to sum all partial results to the root process
```

Chapter 3

Implementation

This section details the implementation of the vector-vector multiplication using MPI in C. The code efficiently distributes the workload across multiple processes, ensuring each process contributes to the overall computation, even when the distribution is not uniform.

3.1 Code Overview

The implementation is divided into the following key steps:

1. **Initialization:**

   - MPI is initialized, and each process retrieves its rank and the total number of processes.
   - The root process (rank 0) initializes the vectors vecA and vecB, and distributes portions of these vectors to all other processes.

   ```
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   ```

2. **Data Distribution:**

   - The vector size (N) is divided among the processes.
   - Each process receives a portion of the vectors vecA and vecB to handle, using MPI_Scatterv.

   ```
   // Determine the number of elements each process will handle
   int local_n = (rank < N % size) ? (N / size + 1) : (N / size);

   MPI_Scatterv(vecA, sendcounts, displs, MPI_DOUBLE, local_vecA, local_n,
                MPI_DOUBLE, 0, MPI_COMM_WORLD);
   MPI_Scatterv(vecB, sendcounts, displs, MPI_DOUBLE, local_vecB, local_n,
                MPI_DOUBLE, 0, MPI_COMM_WORLD);
   ```

3. **Computation:**

- Each process computes a partial result of the dot product for its assigned portion of the vectors.
- These partial results are then reduced to the root process using `MPI_Reduce`.

```
local_result = vector_vector(local_n, local_n, local_vecA, local_vecB);
MPI_Reduce(&local_result, &total_result, 1, MPI_DOUBLE, MPI_SUM, 0,
           MPI_COMM_WORLD);
```

4. **Finalization:**

   - The total result is gathered, the time for operation is printed by the root process and was saved to a file.
   - All dynamically allocated memory is freed, and MPI is finalized.

```
if (rank == 0) {
    printf("%9d %4d %9.4f\n", N, size, total_time);
    fprintf(file, "%9d %4d %9.4f\n", N, size, total_time);
}
free(local_vecA);
free(local_vecB);
MPI_Finalize();
```

Choice of MPI_Scatterv over MPI_Scatter: The **MPI_Scatterv** function was chosen over **MPI_Scatter** because it allows for the distribution of non-uniform data sizes across processes. This flexibility is essential when dealing with uneven distributions of data, ensuring that each process receives the appropriate amount of work based on its rank, thereby optimizing the overall computation and load balancing. For example at core 2 and 3.

This implementation efficiently manages both even and uneven distributions of data, ensuring all processes contribute effectively to the overall computation.

Chapter 4

Test and Results

A vector of size 2.5×10^8 was used to test the performance. The run times were recorded for different numbers of cores, and the results are summarized in Table 4.1.

Table 4.1: Execution time with varying core counts

N	**Cores**	**vv/s (seconds)**
250000000	1	0.6697
250000000	2	0.4596
250000000	3	0.2876
250000000	4	0.2502
250000000	5	0.1808
250000000	6	0.1404
250000000	7	0.1096
250000000	8	0.0853

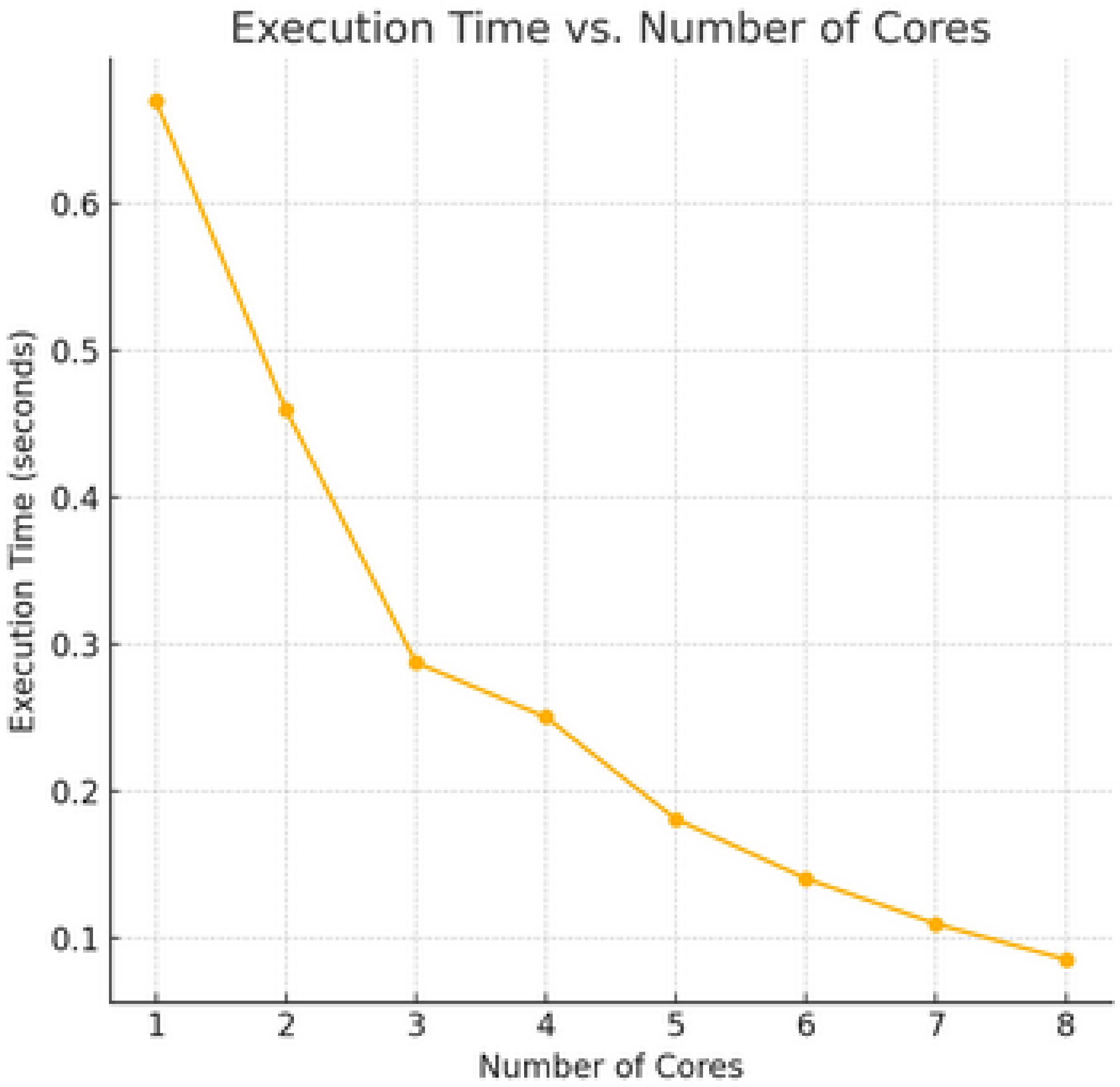

Figure 4.1: Execution Time vs. Number of Cores

Chapter 5

Analysis

5.1 Performance Analysis

The results indicate that as the number of cores increases, the execution time decreases, demonstrating the efficiency of parallel processing. The uneven distribution of workload does not negatively impact the performance, due to efficient use of MPI.

5.2 Speed up Analysis

The speed-up of a parallel algorithm is a measure of the improvement in execution time when using multiple processors compared to a single processor. It is defined as the ratio of the time taken to execute a task with one processor to the time taken with n processors. The formula for calculating speed-up is given by:

$$\text{Speed up (n cores)} = \frac{\text{Time with 1 core}}{\text{Time with n cores}} \tag{5.1}$$

This metric helps in understanding the efficiency and scalability of parallel algorithms. A higher speed-up indicates better performance as more processors are utilized.

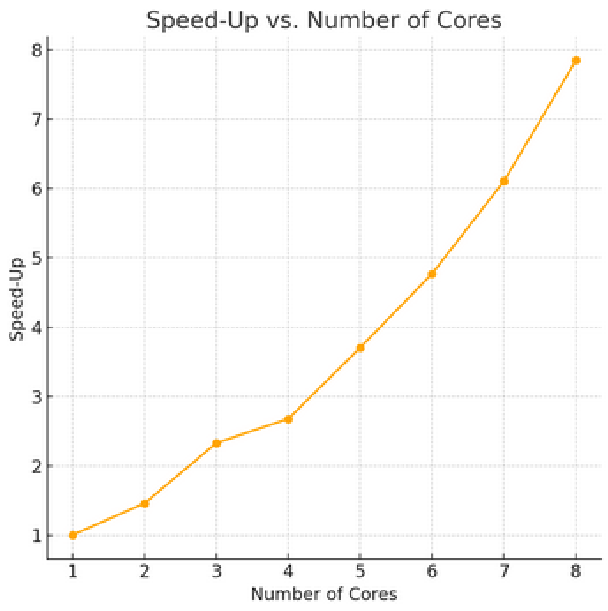

Figure 5.1: Speed-Up vs. Number of Cores

Chapter 6

Conclusion

The results demonstrate that parallelizing the vector-vector multiplication using MPI significantly reduces execution time. The implementation efficiently handles both even and uneven distributions of data, resulting in good scalability. Further optimizations could include fine-tuning the data distribution or exploring hybrid parallelization strategies.

Chapter 7

References

- MPI Standard Documentation: `https://www.mpi-forum.org/docs/`
- ”Parallel Programming in C with MPI and OpenMP” by Michael J. Quinn
- ”Introduction to High-Performance Scientific Computing” by Victor Eijkhout