Implementation of the Bubble-Sort Algorithm Using A Message-Passing Model

Karan Abraham [†]and Bonga Njamela [‡]
EEE4120F 2024
University of Cape Town
South Africa
[§]ABRKAR004 and [¶]NJMLUN002

***Abstract*—In MPI, each process runs in an isolated memory-space. This report describes an investigation into the performance of a bubble-sort algorithm implemented using MPI (Message Passing Interface) compared to a MATLAB implementation. The MATLAB implementation employs a serial approach, while MPI utilizes parallelization to distribute the workload across multiple processors. The investigation focused on analyzing the speedup achieved by the MPI program compared to the MATLAB implementation as the size of an input nxn matrix increased. The speedup of the program was also found to be related to the number of processors in the communicator.**

I. Introduction

Unlike shared memory, the message-passing model (or the MP model) utilizes a set of tasks that employ separate memory-spaces during computations. In shared memory architectures, all processors have access to all memory as global address space [1]. Distributed memory models require a communication network to facilitate memory interactions between processors and there tasks as shown in figure 1 [1].

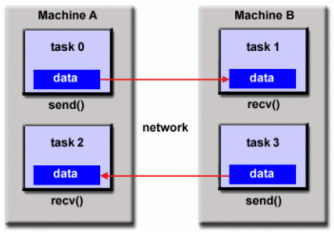

Fig. 1: Showing an implementation of the distributed/message-passing memory model with a set of tasks that can occupy the same device and communicate contents of their local memory through a distributed network.

Distributed memory can be implemented on any underlying hardware using the Message-Passing Interface (MPI) which is compiler-based.

Communications are typically not required for embarrassingly parallel solutions. Applications that require a communication network need to share results or boundary information, such as in the case of solving for systems described by differential equations. Any communication between tasks introduces some kind of overhead. This is because resources such as CPU cycles and memory that could be employed in computation are diverted to packaging and transmitting data. Furthermore, parallel tasks require synchronization to prevent race conditions and data races, leading to an increase in execution time as tasks wait for each other to reach a certain point in the process.

This paper describes the investigation into the performance of a program that employed a distributed memory architecture to execute the bubble-sort algorithm on matrices with random floating-point numbers as entries. The performance was benchmarked using a MATLAB version of the bubble-sort algorithm as a golden measure.

The bubble-sort algorithm was considered to be an embarrassingly parallel algorithm since sorting in the columns can be done separately. This implied that the investigation did not consider communication overheads that typically arise from implementation of MPI on parallel processes. In this investigation, the MPI program and the golden measure were executed on the same CPU using a scatter and gather memory partitioning scheme as described in the following methodology section.

II. Methodology

A. Hardware

An MPI and MATLAB bubble-sort programs were executed on a workstation running Ubuntu Linux 22.04 LTS, equipped with an Intel Core i7-1165G7 CPU. The CPU features 8 cores and supports 16 threads with a base clock speed of 2.58GHz. The performance of programs depended on the availability of processing units, with each core capable of executing multiple threads simultaneously. The multiple-threading capabilities facilitated by the CPU architecture ensures efficient parallelization of computations which contributes to improved performance for solving problems that require parallel and concurrent state management techniques.

B. Implementation

An MPI program, written in C for performing bubble-sort, was executed on a single CPU by implementing the `mpi.h` header file which handles MPI library calls. The bubble-sort algorithm was employed on a matrices of random floating-point numbers which were imported from CSV files.

The CSV files were generated using the MATLAB as shown below.

Listing 1: CSV file generator

```
for n = [10 100 200 500 1000 2000 5000 10000]
    M = rand(n)
    writematrix(M, sprintf('CSV-files/%dx%d.csv', [n n]))
end
```

The investigation used an implementation of the bubble-sort algorithm in MATLAB as shown in listing 2. In MATLAB, the numeric array of randomly generated floating-point numbers was expected to be allocated to a contiguous block of memory [2]. MATLAB uses `double` as the default numeric data type for storing floating-point numbers, which use 8-bytes of memory.

Listing 2: Insert MATLAB Bubblesort Code Here

```
function sorted_matrix = bubbleSort(matrix)
    % Loop through each column of the matrix
    num_cols = size(matrix,2);
    num_rows = size(matrix,1);
    for col = 1:num_cols
        % Apply bubble sort to the current column
        for i = 1:num_rows-1
            for j = 1:num_rows-i
                if matrix(j, col) > matrix(j+1, col)
                    % Swap elements
                    temp = matrix(j, col);
                    matrix(j, col) = matrix(j+1, col);
                    matrix(j+1, col) = temp;
                end
            end
        end
    end

    % Assign the sorted matrix to the output variable
    sorted_matrix = matrix;
end
```

The MPI program used a communicator to compute the bubble-sort algorithm. A communicator is a collection of CPUs that send messages to each other. Information about the number of processors and the rank of the processor were retrieved using the code in listing 3 below. The master processor was expected to be the processor with rank 0. The listing also shows the allocation of memory to store the nxn matrix. The rank 0 processor was used to handle the distribution, printing and gathering of the matrix entries, while the other processors focused solely on computing the bubblesort algorithm on a section of the matrix. To ensure that no data contention existed and that data races did not occur while reading the file, the `MPI_Barrier` was used to ensure that the processes were synchronized.

Listing 3: Code snippet showing the initialization of the MPI program in the `main` function.

```
int rc = MPI_Init(NULL, NULL);
    if (rc != MPI_SUCCESS) {
        printf("Program␣terminated.␣Could␣not␣create␣MPI␣program.\n");
        MPI_Abort(MPI_COMM_WORLD, rc);
    }

    MPI_Comm_size(MPI_COMM_WORLD, &number_of_process);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank_of_process);

    double (*matrix)[n] = malloc(n * sizeof(double[n]));
    if (!matrix) {
        printf("Error:␣Memory␣allocation␣failed.\n");
        return EXIT_FAILURE;
    }

    // Load the matrix from the file on root process
    if (rank_of_process == 0) {
        loadMatrixArray(filename, n, matrix);
        printMatrixArray(n, matrix, 0);
    }

    // Barrier to synchronize processes
    MPI_Barrier(MPI_COMM_WORLD);
```

The investigation used the Scatter and Gather partitioning scheme for distributing the computation in the communicator. This partititioning scheme was expected to distribute an nxn matrix stored in the csv file among the processors as illustrated in figure 2. Since the code was expected to use 4 processors, each segment of the matrix had 4 columns and n rows before scattering.

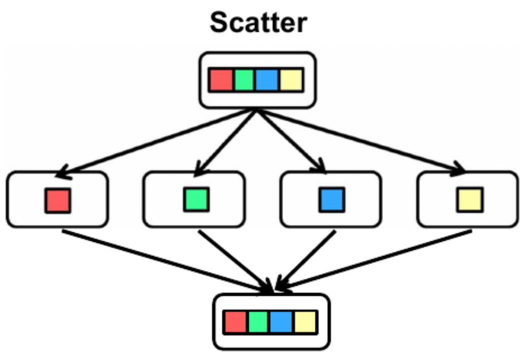

Fig. 2: Showing an illustration of the Scatter and Gather partitioning scheme for distributing the matrix memory block among the processors for computing the bubble sort algorithm.

After sorting the columns of the segments of the scattered matrix, the sorted segments were recombined to form the sorted matrix. Listing 4 shows the `scatterAndSortMatrices` function which was expected to conduct scattering and gathering before merging the segments into an array of `sortedMatrices`.

Listing 4: Code snippet showing the implementation of MPI Scatter and Gather for partitioning the allocated memory for the matrix to each processor.

```
// Function to scatter and sort m matrices
void scatterAndSortMatrices(int m, int n, double (*matrices)[m][n]
                            , double (*sortedMatrices)[m][n]) {
    int number_of_process, rank_of_process;
    MPI_Comm_size(MPI_COMM_WORLD, &number_of_process);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank_of_process);

    int columns_per_matrix = 4;
    int padded_columns = (n % columns_per_matrix == 0) ? n
        : (n + (columns_per_matrix - n % columns_per_matrix));
    int columns_per_process = padded_columns / m;

    double matrixCol[columns_per_matrix][n];
    double sortedMatrixCol[columns_per_matrix][n];

    for (int i = 0; i < m; i++) {
        // Scatter the matrix
        MPI_Scatter(matrices[i], n, MPI_DOUBLE, matrixCol, n,
                                MPI_DOUBLE, 0, MPI_COMM_WORLD);

        // Sort each column
        for (int j = 0; j < columns_per_matrix; j++) {
            bubblesort(n, matrixCol[j]);
        }

        // Gather the sorted columns
        MPI_Gather(matrixCol, n, MPI_DOUBLE, sortedMatrixCol, n,
                                    MPI_DOUBLE, 0, MPI_COMM_WORLD);

        if (rank_of_process == 0) {
            // Store the sorted columns
            for (int j = 0; j < columns_per_matrix; j++) {
                for (int k = 0; k < n; k++) {
                    sortedMatrices[i][j][k] = sortedMatrixCol[j][k];
                }
            }
        }
    }
}
```

Since MPI Scatter divides the memory blocks by each row in the matrix, the matrix was first transposed using the `transposeMatrix` function seen in listing 5 below. This was expected to contribute insignificantly to the execution time of the problem because of the simplicity of transposing a square matrix.

Listing 5: Code snippet showing the transpose function for transposing the nxn matrix.

```
void transposeMatrix(int n, double matrix[n][n]) {
    double temp;
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            temp = matrix[i][j];
            matrix[i][j] = matrix[j][i];
            matrix[j][i] = temp;
        }
    }
}
```

The program was timed using the `MPI_Wtime` function which uses wall clock time to measure the execution time of the program. This wall clock execution time of the MPI bubble-sort program was compared to the wall clock execution time bubble sort program in MATLAB to investigate the performance of parallelization using message-passing in the distributed memory model.

To run the program with 4 cores, the command in listing 6 below. The invesigation used 2, 3, and 4 processors to run the MPI program and characterise the performance as the number of available processors increased.

Listing 6: Code snippet showing the transpose function for transposing the nxn matrix.

```
/usr/bin/mpicc sortingmpi.c -o sortingmpi

/usr/bin/mpirun -np 4 ./sortingmpi
/usr/bin/mpirun -np 3 ./sortingmpi
/usr/bin/mpirun -np 2 ./sortingmpi
```

The speedup was expected to increase as the number of processors increased. To investigate the performance of the MPI program based on the number of processors, the investigation used a 100x100 matrix.

III. Results and Discussion

The results in figure 4 show the speedup graph of the MPI program when executed on 4 cores of the Intel CPU.

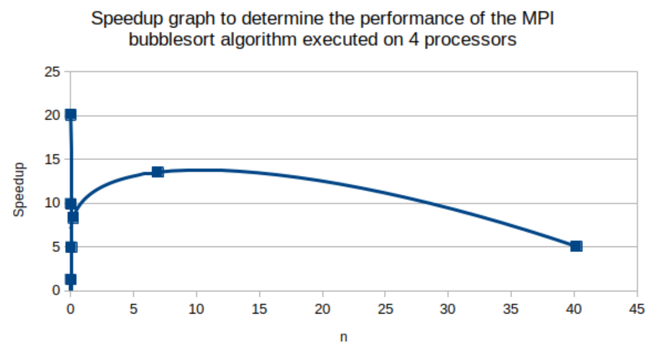

Fig. 3: Speedup graph shows exponential improvements in performance as the size of the matrix increases.

The significant performance difference between the MPI bubble-sort algorithm and the bubble-sort algorithm in MATLAB can be attributed to several factors, including parallelization and memory distribution. MPI (Message Passing Interface) allows for parallelization of the bubble sort algorithm by distributing the workload across multiple processing units (cores or nodes).

Each MPI process operated independently on the segments of the data, sorting a portion of the matrix concurrently with other processes. This parallel execution enabled faster sorting of larger matrices as illustrated by the results. Compared to the serial execution in MATLAB, where only a single thread or core is utilized, scattering the matrix among CPUs improved the speedup of the MPI significantly. The graph shows that when the size of the nxn matrix was small, the speedup was high at about 20. This was accounted for by the fact that the size of intra-communicator messages was relatively small.

In MPI, the matrix was divided into smaller chunks and distributed among the MPI processes using MPI Scatter. Each process worked on its portion of the matrix to reduce the need for data movement and reducing memory contention. Contrary to MPI, MATLAB operated in a shared-memory model, where the contiguous memory block was accessible to all threads or workers. This could lead to contention for memory access and slower performance. The impact of these bottlenecks became more evident as the size of the matrix increased. The distributed memory model of MPI allowed for better scalability and better implementation of resources, resulting in improved performance for sorting operations.

While MPI leads to communication overhead for exchanging matrix data between processes, this overhead is often outweighed by the benefits of parallel execution and reduced memory contention. MATLAB's shared-memory model could have incurred overhead in managing data access and synchronization between threads during the automatic multi-threading operation.

MPI provides low-level control over communication and synchronization, allowed for the optimization of the bubble-sort algorithms for the specific hardware. MATLAB was easier to use, however, it did not provide the same level of optimization and control, leading to suboptimal performance for certain computational tasks. Overall, the maximum average speedup achieved was 1902.

Figure **??** compares the speedup achieved when the program was run on a different number of processors.

The graph shows that overhead incurred by executing on multiple processors. This is demonstrated by the global maxima of the curves showing the average speedup. The largest global maxima was achieved for 2 processors (corresponding to the yellow curve), indicating that 2 processors was the optimal number of processors to perform

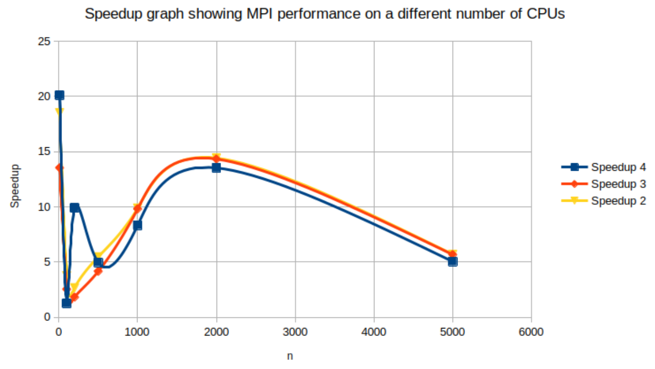

Fig. 4: Speedup graph showing the performance of the MPI program when executed on communicators of different sizes.

the MPI Scattering operation. The decreasing global maxima as the number of processors increased indicated that having to divide and send messages to multiple incurred losses in the performance of the MPI program.

IV. Conclusion

The investigation highlighted the effectiveness of MPI parallelization in improving the performance of sorting algorithms compared to serial implementations like MATLAB. The MPI program achieved a maximum average speedup of 20 compared to the MATLAB implementation, showcasing the benefits of parallel computing in optimizing computational tasks. Peak performance was achieved when executing the program on 2 processors - as illustrated by a large global maximum in the speedup curve of the MPI program . Overall, MPI offered better scalability, resource utilization, and performance optimization opportunities, making it a suitable choice for parallelizing computationally intensive algorithms on modern hardware architectures.

References

[1] L. Computing, "Introduction to parallel computing tutorial," https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial Last Accessed: 13 April 2024.

[2] MathWorks. How MATLAB Allocates Memory. [Online]. Available: https://www.mathworks.com/help/matlab/matlab$_{prog/memory-allocation.html}$