1. **Problem Statement**

Calculate the Matrix-Vector Multiplication of an M=32,768 by N = 16,384 matrix. Create a program which runs this calculation in C programming language using up to no more than 32 nodes. Design and implement a program with as optimal performance as possible using MPI and OpenMP. Document performance and include a Karp-Flatt metric.

1. **Approach**

Before designing the solution to this problem, the current academic references for an optimum design were examined in the analysis phase. Using Parallel Programming in C with MPI and OpenMP as a guide, there were three distinct approaches to the problem: row-wise, column-wise, and block-wise decomposition. Each approach had a different method of solving the problem, but row-wise and column-wise decomposition proved to be not very scalable. As such, block-wise decomposition was selected as the prime candidate for this application, being more scalable that the other two implementations .

Block-wise decomposition, or “Checkerboard” decomposition, breaks down the problem size into smaller blocks which are assigned to each process. Before venturing into the details of block-wise decomposition, a simple row-wise decomposition must be understood. Take the following example in Figure 1.

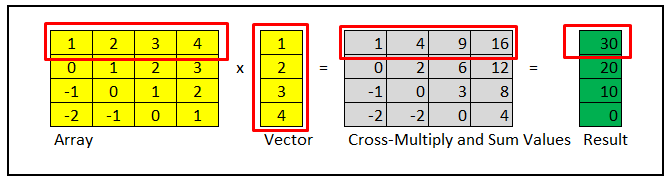


Figure 1. Row-wise decomposition illustration

Row-wise decomposition would simply assign each row to a process along with the corresponding vector to compute each element in the result. However, because of high communication overhead, this method is not ideal. Block-wise decomposition, however, breaks the problem into blocks as illustrated in Figure 2.

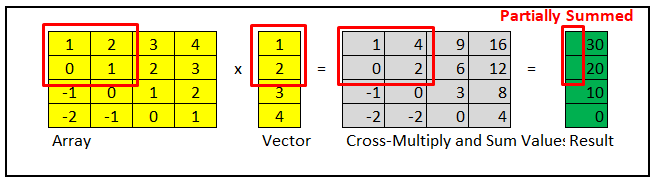


Figure 2. Block-wise decomposition illustration

As shown, blocks are multiplied in a processor and partial sums are computed. This is quite a bit more complex than row-wise or column-wise decomposition, but results in a better time complexity for communication overhead. A complete illustration for how this is workload is divided is shown in Figure 3.

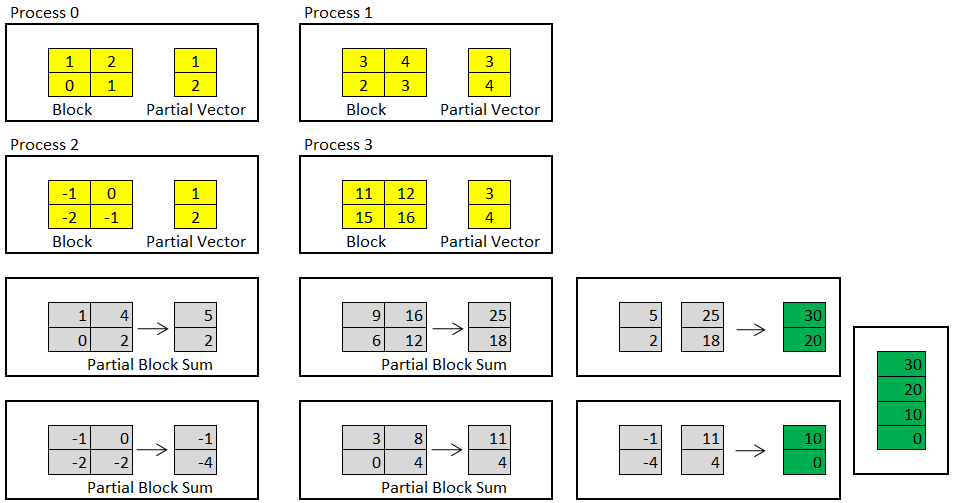


Figure 3. Illustration showing how workload is divided among processes rank 0 through 3

This method requires a Cartesian communicator using a 2D grid for processes to communicate through MPI. Within each processor, the multiplication task can be further improved using OpenMP to get the partial sum.

1. **Solution**

During the implementation of the checkerboard block-wise decomposition approach to solving matrix-vector multiplication problems, it was soon discovered that the design became increasingly complex with many setbacks and risks due to lack of knowledge and experience in MPI implementation. I was able to successfully implement a Cartesian Communicator that accepting any range of processors. The matrix problem size was also successfully divided into blocks with block dimensions varying depending on the location of the processor in the Cartesian Communicator.

However, since the problem size may not be perfectly square and the sub-matrices in each block may be of varying size, this made the solution substantially more difficult. One possible approach may be to implement MPI\_Datatypes for each uniform block and handle remainders separately. This would allow for an MPI\_Scatterv call followed by a separate one for the remainders. Another approach would be the one that was partially implemented in this first attempt, which involved creating two additional communicators using MPI\_Comm\_split, one for root communicators at the head of each column, and another for column communication across each column. However, even after completing this, there would be a need for a custom MPI\_Datatype to handle blocks for the first MPI\_Scatterv across columns followed by the MPI\_Scatterv down the column.

Ultimately, due to the risks involved in limited time for this project, the executive decision was made to scrap the first attempt. A partial solution using the checkered decomposition approach can be found in matrix\_vector\_checkered.c and compiled as follows:

**mpicc matrix\_vector\_checkered.c -lm -o matrix\_vector\_checkered**

A sample run showing the decomposition of the problem domain, Cartesian communicator and two MPI split communicators can be performed using:

**mpirun -n 4 ./ matrix\_vector\_checkered 5 5**

A sample output can be found below in Figure 4.

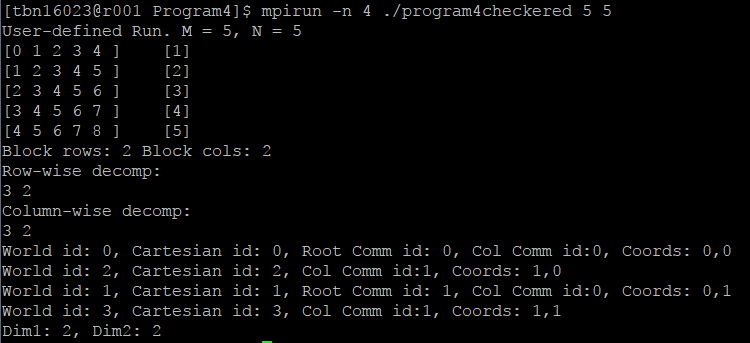


Figure 4. Checkerboard block-wise decomposition first attempt at approaching Matrix-Vector Multiplication

After scrapping the first attempt, I approached the problem with row-wise decomposition. Although not a scalable, this solution was less complex allowing me to use experience from prior projects with the added new MPI techniques learned. The first step was to decompose the problem size, row-wise, with remainder rows evenly distributed depending on communicator rank. then, the send count, receive count, send displacement, and receive displacement arrays must be created and populated for using MPI\_Scatterv. Once this was completed, MPI\_Scatterv can be used to distribute the matrix rows. Then, using MPI\_Bcast, the vector can be distributed, as well.

With this completed, all processors simply need to perform matrix multiplication which can be optimized using OpenMP. Using an OpenMP parallel for pragma with the reduction clause, this improved performance. Finally, with each process work completed, the final result was gathered using MPI\_Gatherv. After some debugging, the successful result can be found below in Figure 5 and Figure 6.

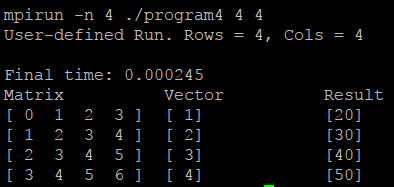


Figure 5. Row-wise decomposition approach result to verify correctness

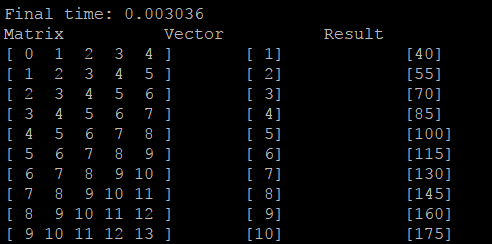


Figure 5. Another row-wise decomposition approach result to verify correctness

A Makefile was included with the following commands to re-test this solution:

**make**  make complete executable solution

**make debug** test solution with debugger showing values

**mpirun –n <process count> ./ matrix\_vector\_rowwise m n DEBUG**

. manual input to run the program with custom process count, m and n values, and DEBUG to printout result

Finally, Figure 6 shows the processed results to analyze the performance of the implemented row-wise decomposition algorithm in the solution. This was run using a matrix with m = 30000 rows and n = 16000 columns. Keeping the problem size constant, the number of processors was increased from 1-10 processors. The outputted execution time is shown with the calculated speedup below.

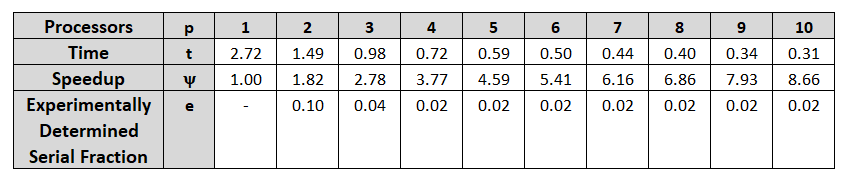


Figure 6. Karp-Flatt metric - Analyzing the performance of the implemented solution

While speedup is clearly increasing directly with the number of processors operating in parallel, it can be difficult to see if there are inefficiencies or problems that affect scaling. The Karp-Flatt metric is a powerful tool, because it allows us to detect sources of inefficiency and overhead which can affect scale-up. After 3 processors operating in parallel, the experimentally determined serial fraction is shown to remain constant when it should be steadily increasing. This indicates that the source keeping the algorithm from even greater performance increase/speedup is due to the large serial fraction. As such, by further improving the algorithm to improve parallel operations, we can improve this algorithm even further in the future.