Introduction to Parallel Computing
A.Y. 2023/2024
Homework 3: Parallelizing matrix operations using MPI
Results report

Alessandro Iepure, 228023
alessandro.iepure@studenti.unitn.it
`https://github.com/aleiepure/Parallel-assignment3`

13 December 2023

Abstract

This document analyzes the results and methodologies used while working on the assignment in the title. All work is original and done without input from outside sources except where explicitly noted.

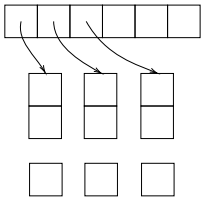
1 Parallel matrix transposition

This exercise involves the implementation of two matrix transposition algorithms using MPI (Message Passing Protocol) for parallelization. The goal is to compare the scalability and efficiency between both versions.

Transposing a matrix involves flipping it over its main diagonal, exchanging rows and columns, resulting in a new matrix where the rows of the original one become the columns of the transposed matrix and vice versa. When handling sequential codes, this is relatively straightforward. However, the introduction of MPI parallelization complicates the implementation.

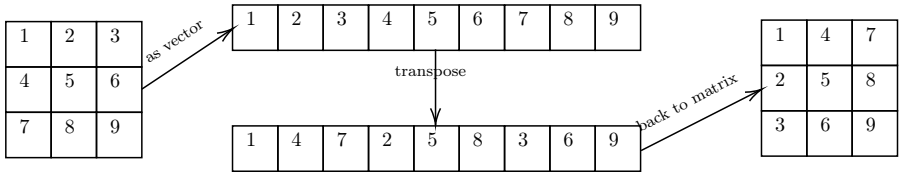
For part 1, the initial implementation mirrored that of the previous assignment. Improving it with MPI for part 2 proved more challenging. It required the evaluation of various strategies to achieve a correct data subdivision. What follows is a concise breakdown of the thinking process that led me to the implemented logic.

Figure 1: Idea 1 - Array of pointers to arrays



The first concept was to leverage the way C stores matrices as an array of pointers to other arrays (Figure 1). However, I abandoned this idea quickly due to the challenges in passing such data structure to MPI's collective operation functions (like `MPI_Scatter()`, `MPI_Gather()`, etc.). This led me to treat the matrix as a one-dimensional array (Figure 2). This turned out to be a winning approach and ultimately what I implemented.

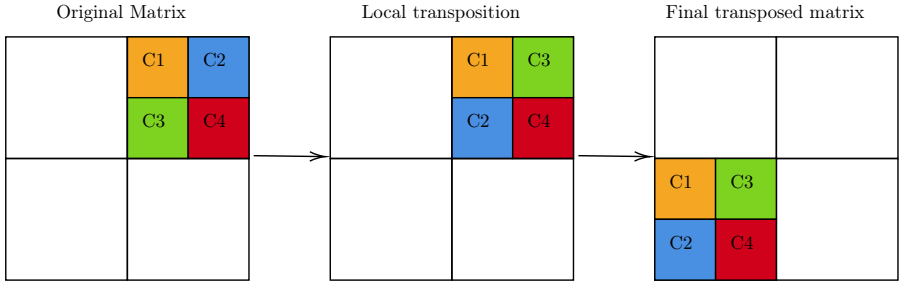
Figure 2: Idea 2 - Matrix as a 1D array



In task 3, I opted for a different approach: I divided the matrix (still represented as a 1-D array) into sizable blocks of equal dimensions. Each of these blocks is further subdivided and distributed to the available CPUs as smaller blocks. These smaller blocks are the local copy of each processor and are transposed and sent back to the main CPU. The latter is responsible for reassembling the transposed smaller portions into the transposed large block.
Before iterating to the next block, the transposed block is positioned correctly to create the final transposed matrix, also stored in a1-D array. All matrices are assumed to be square, and the number of blocks dividing

the matrix is equal to the number of processors. While both limitations are present for simplicity, with small modifications any number can be used, provided it is mathematically compatible.

Figure 3: Idea 3 - Divide into blocks, each processed individually



Although the assignment asked for a single executable with explicit functions for the sequential and parallel versions of the 2 algorithms, I could not find a reliable way to divide my logic into separate functions. Due to time constraints, I decided to write them as 2 separate programs. I interpreted the sequential requirement as the execution of the same parallel code restricted to only one CPU. Each program is called repeatedly using a series of PBS files - *bash* scripts that detail the necessary resources required for the execution on the cluster.

Result analysis

The final code and supporting files were uploaded to the University's HPC cluster and queued for execution. The nodes responsible for the computations were `hpc-c11-node12.unitn.it` (for 1-CPU and 4-CPUs executions), `hpc-c11-node01.unitn.it`, and `hpc-c04-node01.unitn.it`. The first three nodes are equipped with an Intel® Xeon® Gold 6252N CPU running at 2.30GHz and 1024GB of RAM, while the fourth has an Intel® Xeon® Gold 6140M CPU clocked at 2.30GHz and 768GB of RAM.

To meet the assignment's requirement for both strong and weak scaling, two sets of input matrix sizes were used. The weak scaling has progressively increasing workloads, while the strong scaling maintains them constant. Table 1 and table 2 report the execution times for both weak and strong scaling respectively, alongside the achieved bandwidth, calculated using Equation 1. Additionally, plot 1 and plot 2 visualize these results.

$$\text{Bandwidth} = \frac{B_r * B_w * \text{Times} * \text{size\_of}(float)}{\text{Run Time}} = \begin{cases} \frac{\text{Size}^2 * 3 * 4}{\text{Run\_Time}}, & \text{without blocks} \\ \frac{\text{Size}^2 * 4 * 4}{\text{Run Time}}, & \text{with blocks} \end{cases} \quad [\text{Byte/s}] \tag{1}$$

Unfortunately, I had difficulties obtaining results for the missing values in the tables above as the execution would stall until the allocated wall time was reached, resulting in the process being terminated by the system. This is probably happening because the assigned nodes have insufficient memory available.

Table 1: Weak scaling

Cores	Size	Normal		Blocks	
		Run Time [s]	Bandwidth [GB/s]	Run Time [s]	Bandwidth [GB/s]
1	64	0.00003	1.446	0.00012	0.570
	128	0.00013	1.512	0.00035	0.749
	256	0.00051	1.536	0.00137	0.766
	512	0.00298	1.056	0.00577	0.727
4	256	0.01306	0.060	0.01700	0.062
	512	0.03965	0.079	0.18536	0.023
	1024	0.01104	1.139	0.09559	0.176
	2048	0.06305	0.798	0.09333	0.719
16	1024	0.01504	0.837	0.01117	1.502
	2048	0.02647	1.901	0.04131	1.625
	4096	0.13280	1.516	0.15336	1.750
	8192	0.31432	2.562	0.65696	1.634
64	4096	0.22652	0.889	0.19813	1.355

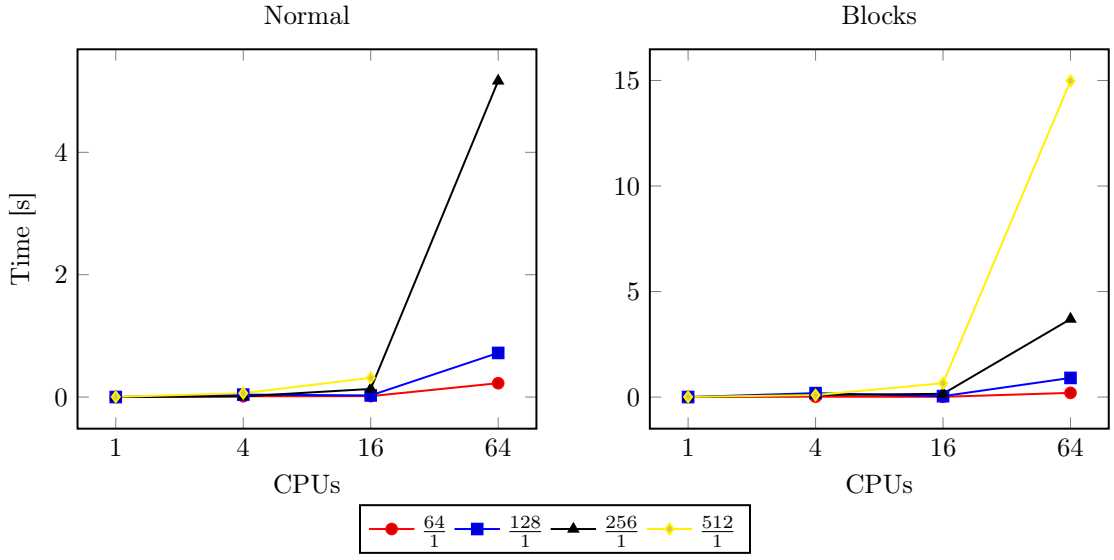
Table 1: Weak scaling

Cores	Size	Normal		Blocks	
		Run Time [s]	Bandwidth [GB/s]	Run Time [s]	Bandwidth [GB/s]
	8192	0.72118	1.117	0.90659	1.184
	16384	5.16715	0.623	3.69552	1.162
	32768	-	-	14.97982	1.147

Table 2: Strong scaling

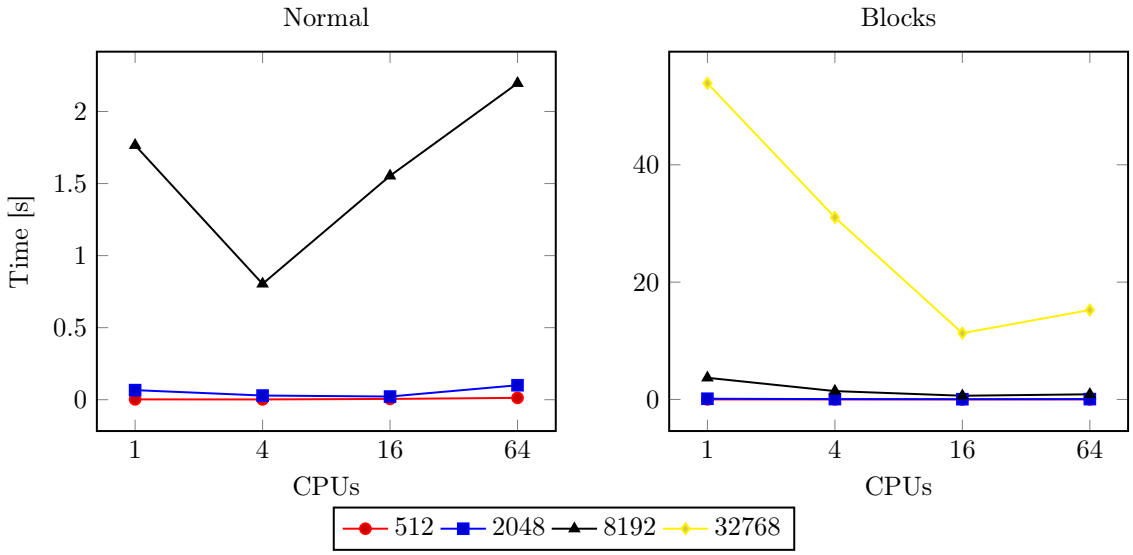
Size	Cores	Normal		Blocks	
		Run Time [s]	Bandwidth [GB/s]	Run Time [s]	Bandwidth [GB/s]
512	1	0.00230	1.366	0.00518	0.810
	4	0.00199	1.583	0.00394	1.066
	16	0.00552	0.570	0.00362	1.160
	64	0.01288	0.244	0.00858	0.489
2048	1	0.06648	0.757	0.11989	0.560
	4	0.02901	1.735	0.05956	1.127
	16	0.02207	2.280	0.04145	1.619
	64	0.10043	0.501	0.05700	1.177
8192	1	1.76583	0.456	3.69059	0.291
	4	0.80396	1.002	1.42328	0.754
	16	1.55372	0.518	0.63222	1.698
	64	2.19584	0.367	0.87797	1.223
32768	1	-	-	53.89544	0.319
	4	-	-	31.03135	0.554
	16	-	-	11.29802	1.521
	64	-	-	15.24146	1.127

Plot 1: Weak scale graph by size - CPUs ratio



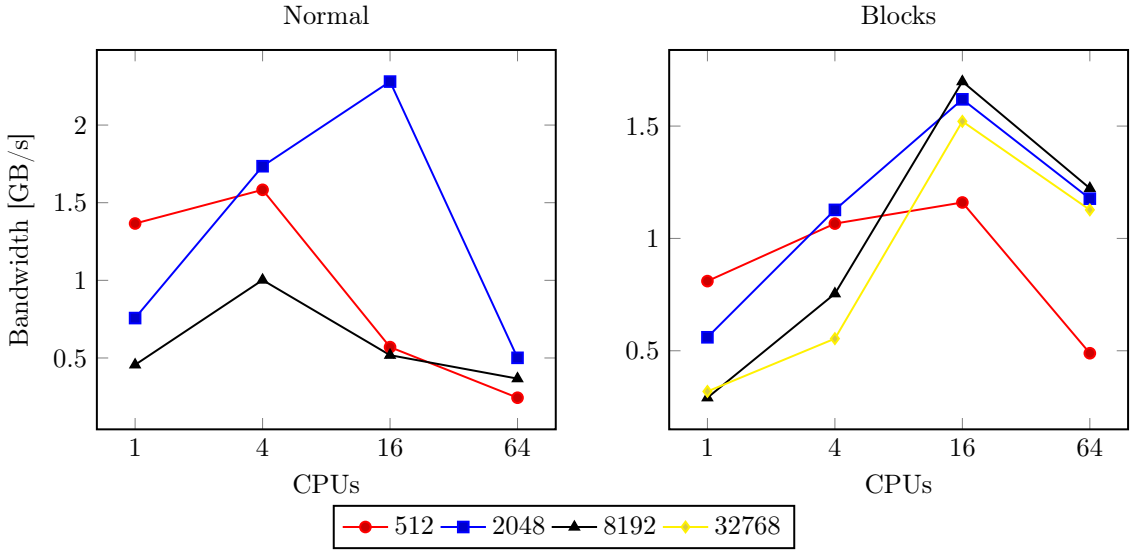
Looking at the weak scaling results (plot 1), we can observe that, although the workload of each CPU remains constant, the time required for computation increases as the number of CPUs rises. This is because communications between the processes increase as well, becoming more frequent and resource-intensive.
On the other hand, the strong scaling (plot 2) shows a general decreasing trend, pronounced for matrices of size 8192 in the normal algorithm and 32768 in the block one. In those cases, the curve starts to flatten out with executions executed by 4 and 16 CPUs respectively.

Plot 2: Strong scale graph by size of matrices



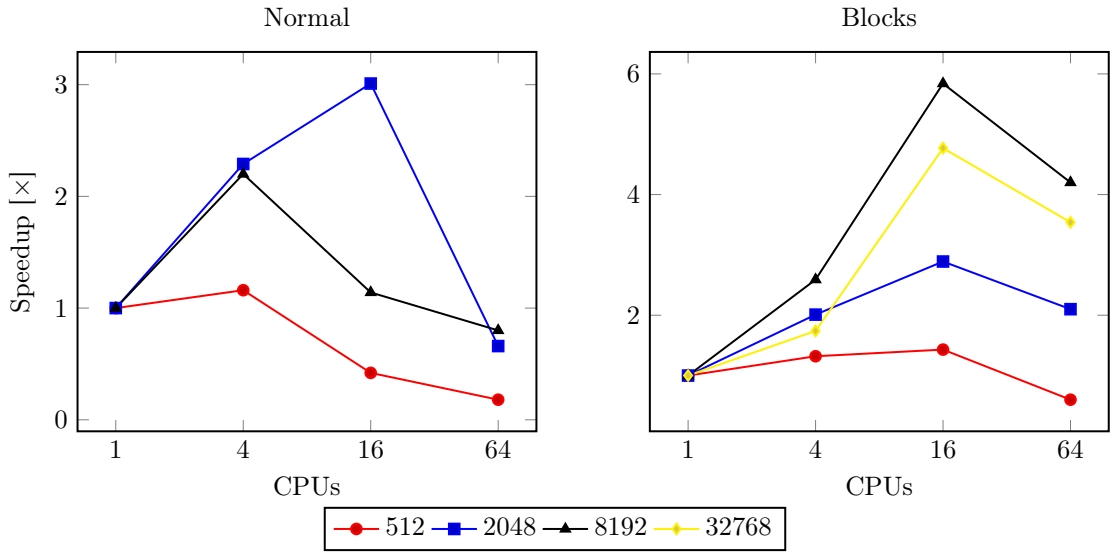
The parallel scalability of both algorithms was evaluated using the bandwidth as the performance metric. Plot 3, plot 4, and plot 5 show us a representation of the performance, speedup, and efficiency respectively.

Plot 3: Performance graph by size of matrices

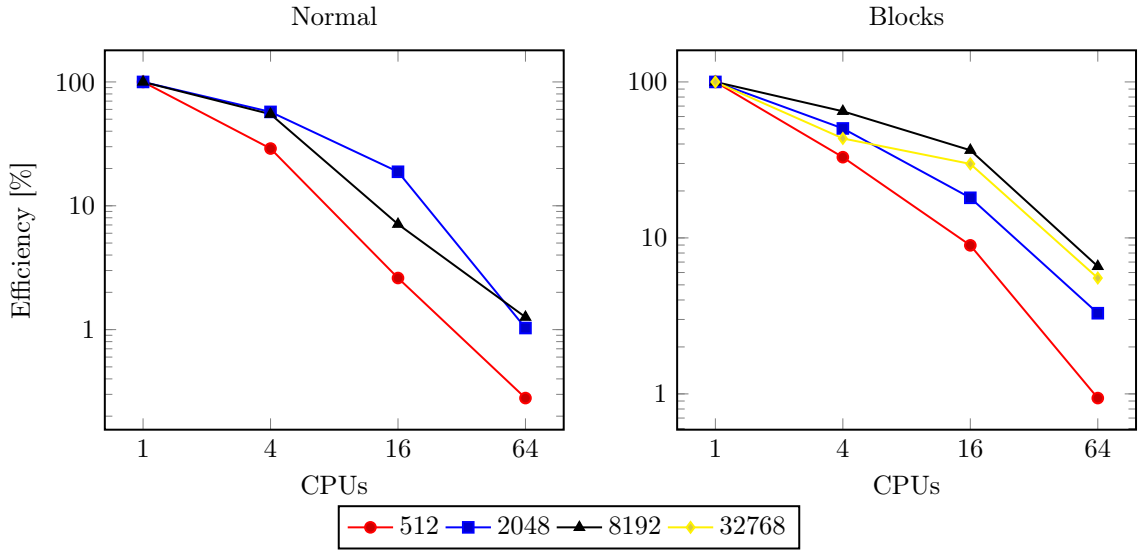


Contrary to what we would expect, the performance of both algorithms is quite irregular. Typically, the bandwidth should rise with the increase in CPUs, but in this case, it increases up to a point before declining. What is said holds valid for both the normal and the blocks algorithms. All the above is also reflected in the speedup and efficiency graphs, which show data scattered all over the place with a general downward trend with the increase in the number of CPUs.

Plot 4: Speedup graph by size of matrices



Plot 5: Efficiency graph by size of matrices



2 OpenMP and MPI Comparison

After completing all three assignments and gaining some knowledge in parallel computing techniques, I can assert that achieving a balance between OpenMP and MPI involves considering the system architecture and the specific problem at hand. OpenMP focuses on shared memory and excels in simplifying parallelization within a single node, making it ideal for tasks like loop-level parallelism and efficient utilization of multiple cores. In contrast, MPI offers explicit control over communications between distributed nodes, making it suitable for larger-scale applications that extend across clusters or supercomputers.

The decision between the two often depends on factors such as scale and the need for granular control or scalability across nodes. For smaller to medium-sized tasks on shared-memory systems, OpenMP provides the simplest and most efficient solution. However, for large-scale distributed computations requiring communication across multiple nodes, MPI becomes the preferred choice due to its explicit message-passing capabilities.

Combining both techniques into what is known as "MPI+OpenMP hybrid programming" allows leveraging the strengths of both paradigms. MPI manages communications between nodes, enabling setup across a cluster or supercomputer, while OpenMP handles in-node parallelism via its directives, maximizing multicore utilization. This hybrid model offers a flexible and scalable solution, however, implementing it requires careful planning and considerations regarding workload distribution, overheads, and load balancing to ensure good performance across the entire system.

References

[1] Professor Laura del Río Martín. *Advanced Features and Optimizations in MPI for Distributed-Memory Architectures*. Lecture 15 slides for Professor's Vella course Introduction to Parallel Computing. 2023. URL: https://didatticaonline.unitn.it/dol/pluginfile.php/1783723/mod_folder/content/0/IntroPARCO-L15.pdf?forcedownload=1.

[2] Professor Laura del Río Martín. *MPI Techniques for Distributed-Memory Architectures*. Lecture 14 slides for Professor's Vella course Introduction to Parallel Computing. 2023. URL: https://didatticaonline.unitn.it/dol/pluginfile.php/1782846/mod_folder/content/0/IntroPARCO-L14.pdf?forcedownload=1.