Introduction to Parallel Computing
A.Y. 2023/2024
Homework 2: Parallelizing matrix operations using OpenMP
Results report

Alessandro Iepure, 228023
alessandro.iepure@studenti.unitn.it
`https://github.com/aleiepure/Parallel-assignment2`

22 November 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 multiplication

The first task asks for the implementation of a parallel matrix by matrix multiplication algorithm in C using OpenMP. The goal of the task is to check the scalability of the code and evaluate its efficiency.

Matrix multiplication methods vary in performance when applied to dense and sparse matrices [4]. The most trivial way to calculate this operation is the "row by column" method which involves computing each element of the resulting matrix by taking the dot product of a row of the first matrix and the corresponding column of the second one. It is commonly used for dense matrices and requires n^3 multiplications for a $n \times n$ matrix. While it also works for sparse matrices, it is less efficient due to the large portions of zero elements.
Sparse matrices offer advantages in memory usage and computation efficiency by storing only the non-zero values and their positions, reducing the occupied space [3]. Moreover, sparse matrix-oriented algorithms, like Strassen's, exploit the structure of the data to reduce the number of operations needed while also benefitting from a better cache utilization.
In my assessment, implementations of different algorithms for the two types of matrices and the respective analyses are beyond the scope of this assignment. I choose to implement the "row by column" algorithm that works for both, even if it may be less performant.

The algorithm was implemented sequentially and later I introduced OpenMP directives to obtain a parallel code.
Source code 1 shows the parallelized function with the directives used. The choice of which ones to use was made after reading the professor's material on the subject [6], OpenMP's 3.1 documentation [1, 2], and trying various combinations. In the end, I decided to use the following: `#pragma omp parallel for collapse(2) reduction(+:temp)`. In particular:

- `#pragma omp parallel`: creates a parallel region in which all code is distributed among the specified number of CPUs. The amount is controlled externally by the `OMP_NUM_THREADS` environment variable.
- `for`: `for`-loops are going to be distributed between available CPUs.
- `collapse(2)` applies to the previous `for` directive and specifies the number of nested loops to divide among the CPUs.
- `reduction(+:temp)` specifies the variable and the operation on which to perform the reduction.

Source code 1: Implemented algorithm

```
float **matMulPar(float **A, float **B, int a_rows, int a_columns, int b_rows, int b_columns) {
    double wt1, wt2;
    float temp;
    float **C = allocateMatrix(a_rows, b_columns);
    wt1 = omp_get_wtime();

#pragma omp parallel for collapse(2) reduction(+:temp)
    for (int i = 0; i < a_rows; i++) {
        for (int j = 0; j < b_columns; j++) {
            temp = 0;
            for (int k = 0; k < a_columns; k++) {
                temp += A[i][k] * B[k][j];
            }
            C[i][j] = temp;
        }
    }
    wt2 = omp_get_wtime();

    printf("%f\n", wt2 - wt1);
    return C;
}
```

Before the final run on the University's HPC cluster (GCC 4.8.5, OpenMP 3.1), the code was tested and debugged on my local machine, a laptop equipped with an Intel® Core™ i5-8300H at 2.30GHz and 16GB of RAM running Fedora 39 (GCC 13.2.1, OpenMP 4.5).
As an effort to simplify the compilation and execution of the various iterations of the code with different values, I wrote a *bash* script which later became the PBS job submission file. This script is responsible for creating the necessary directories on the file system, for compilation, and execution with the various values. The results are returned in the form of a series of `.CSV` files which are easier to work on with other programs.

Results analysis

The implemented algorithm was run with dense square matrices of sizes ranging from 200 to 2000 with up to 32 CPUs. The node responsible for the computation was `hpc-c11-node22.unitn.it` equipped with an Intel® Xeon® Gold 6252N CPU running at 2.30GHz and 256GB of RAM. The obtained execution times are reported in Table 1 alongside the FLOPS calculated using Equation 1 and expressed as GFLOPS.

$$\text{FLOPS} = \frac{2 \cdot \text{A rows} \cdot \text{A columns} \cdot \text{B columns}}{\text{execution time}} = \frac{2 \cdot \text{size}^3}{\text{execution time}} \tag{1}$$

Table 1: Matrix multiplication - Run times and FLOPS

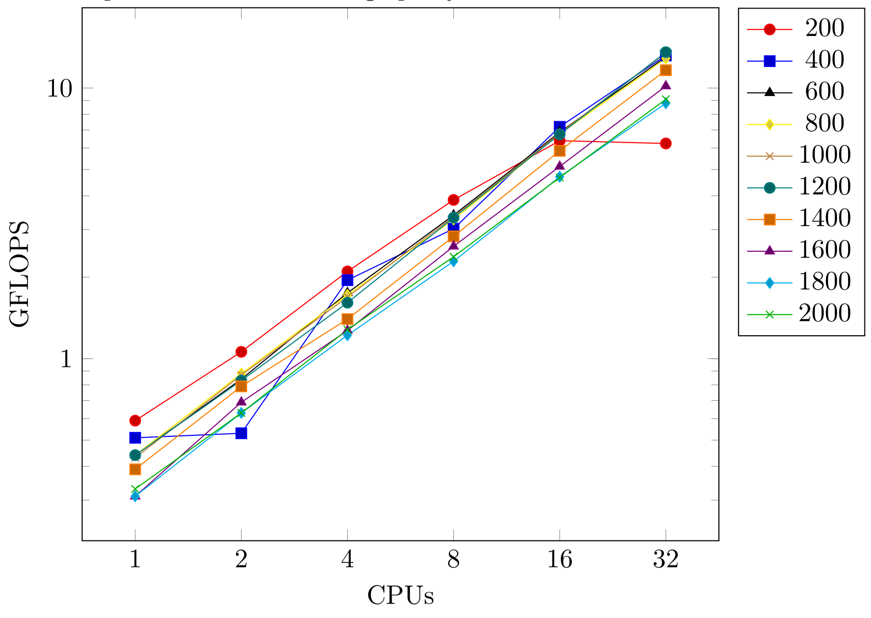
Size	Cores	Run Time [s]	GFLOPS	Size	Cores	Run Time [s]	GFLOPS
200	1	0.0272	0.59	1200	1	7.9264	0.44
	2	0.0150	1.06		2	4.1837	0.83
	4	0.0076	2.10		4	2.1500	1.61
	8	0.0041	3.86		8	1.0419	3.32
	16	0.0025	6.39		16	0.5124	6.74
	32	0.0026	6.24		32	0.2546	13.57
400	1	0.2509	0.51	1400	1	14.1042	0.39
	2	0.2405	0.53		2	6.9411	0.79
	4	0.0656	1.95		4	3.9240	1.40
	8	0.0424	3.02		8	1.9400	2.83
	16	0.0178	7.19		16	0.9362	5.86
	32	0.0097	13.19		32	0.4717	11.64
600	1	0.9862	0.44	1600	1	26.1967	0.31
	2	0.5122	0.84		2	11.8077	0.69
	4	0.2474	1.75		4	6.4396	1.27
	8	0.1273	3.39		8	3.1493	2.60
	16	0.0629	6.87		16	1.5955	5.13
	32	0.0331	13.03		32	0.8052	10.17
800	1	2.3327	0.44	1800	1	37.6769	0.31
	2	1.1591	0.88		2	18.6574	0.63

Table 1: Matrix multiplication - Run times and FLOPS

Size	Cores	Run Time [s]	GFLOPS	Size	Cores	Run Time [s]	GFLOPS
	4	0.5950	1.72		4	9.5689	1.22
	8	3.3135	3.27		8	5.1028	2.29
	16	0.1514	6.76		16	2.4839	4.70
	32	0.0769	12.87		32	1.3263	8.79
1000	1	4.6862	0.43	2000	1	48.0570	0.33
	2	2.2981	0.87		2	25.5150	0.63
	4	1.1806	1.69		4	12.4870	1.28
	8	0.5962	3.35		8	6.7303	2.38
	16	0.2936	6.81		16	3.4271	4.67
	32	0.1483	13.49		32	1.7600	9.09

The parallel scalability of the algorithm was evaluated using the FLOPS as the performance metric. Plot 1, Plot 2, and Plot 3 allow us to see a visual representation of the performance, the speedup, and the efficiency, respectively.

Plot 1: Parallel Matrix Multiplication - Performance graph by size of matrices



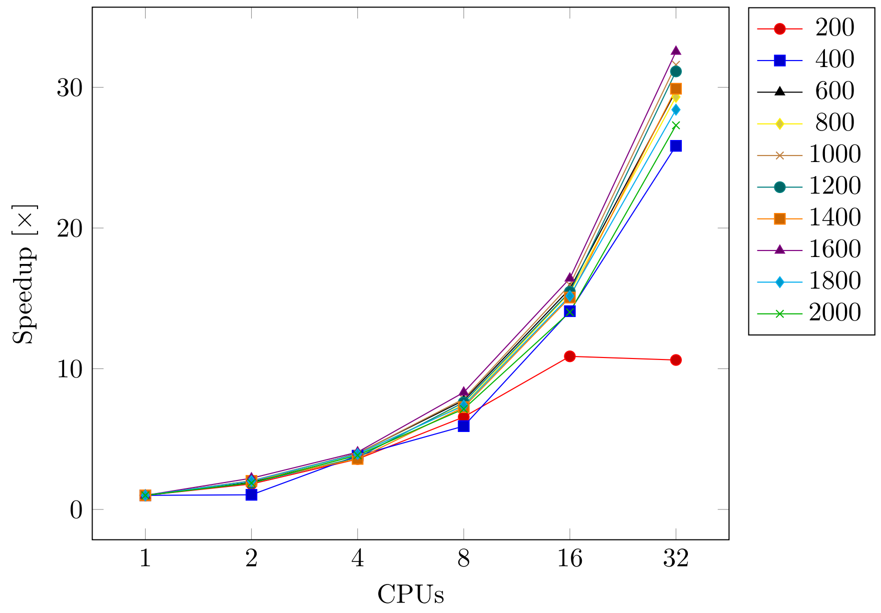
It is easy to see that the execution has a better performance (Plot 1) the more CPUs it can utilize because of the higher number of operations it can execute per second. The same consideration is also valid for the speedup (Plot 2): the more CPUs, the faster the calculations are executed.
The spikes observable in the efficiency graph (plot 3) are because parallel systems are nonlinearly scalable. We must choose either a constant run time or a constant efficiency when scaling up the problem [5]. The general downward slope can be interrupted by undefined peaks caused by imbalance problems. In case in exam, it is particularly evident with 200 × 200 matrices (red line) and 400 × 400 matrices (blue line).

2 Parallel matrix transposition

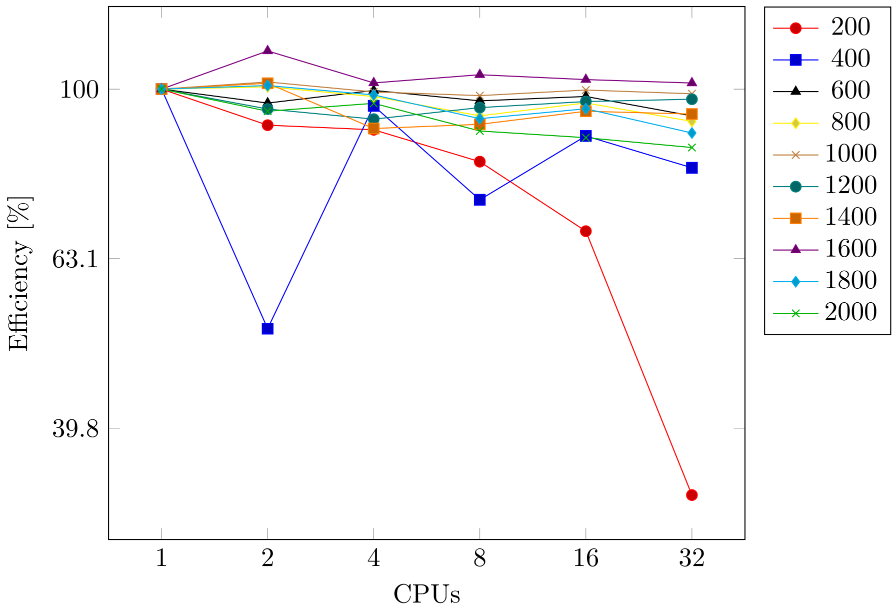
The second task asks for the implementation of a matrix transposition algorithm in two different ways, written in *C* and parallelized with OpenMP. The end goal is to compare the scalability of the two versions and evaluate their efficiency.

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. The first algorithm does verbatim what described above, while the other divides the original matrix into blocks of fixed sizes, transposes them using the same principle, and finally transposes each block's content.

Plot 2: Parallel Matrix Multiplication - Speedup graph by size of matrices



Plot 3: Parallel Matrix Multiplication - Efficiency graph by size of matrices



Both algorithms were written and tested sequentially and parallelized later by introducing the OpenMP directives visible in source code 2. The choice of which ones to use was made after reading the professor's material on the subject [6], OpenMP's 3.1 documentation [1, 2], and trying various combinations[1].

Source code 2: Implemented algorithm

```
float **matTpar(float **M, int rows, int cols) {
    double wt1, wt2;
    float **T = allocateMatrix(cols, rows);

    wt1 = omp_get_wtime();
#pragma omp parallel for collapse(2)
    for (int i = 0; i < rows; i++)
        for (int j = 0; j < cols; j++)
            T[j][i] = M[i][j];
    wt2 = omp_get_wtime();
```

[1]For an explanation of each part of the directive, refer to Task 1, Parallel matrix multiplication.

```
    printf("%f, ", wt2 - wt1);
    return T;
}

float **matBlockTpar(float **M, int rows, int cols, int b_rows, int b_cols) {
    double wt1, wt2;
    float **T = allocateMatrix(cols, rows);

    wt1 = omp_get_wtime();
#pragma omp parallel for collapse(4)
    for (int i = 0; i < rows; i += b_rows) {
        for (int j = 0; j < cols; j += b_cols) {
            for (int x = 0; x < b_rows; x++) {
                for (int y = 0; y < b_cols; y++) {
                    T[j + y][i + x] = M[i + x][j + y];
                }
            }
        }
    }
    wt2 = omp_get_wtime();

    printf("%f\n", wt2 - wt1);
    return T;
}
```

Compilation and execution are achieved, both locally (see previous task for the architecture) and on the HPC cluster, via the same PBS script used for task 1, integrated with the new commands. The outputs of the runs are returned in `CSV` files for easier analysis with external programs.

Results analysis

The implemented code was executed with square matrices of sizes between 20000 and 50000, each with square blocks of 1000, 2000, 5000, and 10000. All combinations were run with up to 32 CPUs on `hpc-c11-node22.unitn.it`[2] Execution times are reported in Table 2 and Table 3 alongside the achieved bandwidth, calculated using Equation 2. Due to the way I wrote the *C* code, the algorithm without the blocks runs four times per matrix size per number of CPUs. The bandwidth reported in Table 2 refers to the average of the runtimes.

$$\text{Bandwidth} = \frac{\text{Rows} \cdot \text{Columns} \cdot \text{size\_of}(\mathit{float})}{\text{execution time}} = \frac{\text{size}^2 \cdot 4}{\text{execution time}} \qquad [\text{B/s}] \tag{2}$$

Table 2: Matrix transposition - run times and bandwidth

Size	Cores	Run 1 [s]	Run 2 [s]	Run 3 [s]	Run 4 [s]	Average [s]	Bandwidth [GB/s]
20000	1	6.5606	6.6204	6.6265	6.5600	6.5919	0.24
	2	3.4106	3.3133	3.3296	3.3078	3.3404	0.48
	4	2.0776	1.8602	2.0055	2.0716	2.0037	0.80
	8	0.9174	0.9229	0.9655	0.9833	0.9473	1.69
	16	0.5285	0.5373	0.5404	0.5687	0.5437	2.94
	32	0.3506	0.3423	0.3365	0.3830	0.3532	4.53
30000	1	15.6335	15.5900	15.5903	16.2771	15.7727	0.23
	2	8.4734	8.1278	8.2218	8.2349	8.2645	0.44
	4	4.3465	4.8519	4.9496	4.8396	4.7469	0.76
	8	2.2628	2.4801	2.6046	2.2887	2.4091	1.49
	16	1.2341	1.8231	1.2558	1.2322	1.3863	2.60
	32	0.8240	0.8335	1.1785	0.7841	0.9050	3.98
40000	1	27.9708	28.9593	27.9213	27.9582	28.2024	0.23
	2	14.9170	14.8816	14.8110	14.9570	14.8919	0.43

[2] refer to task 1 for its architecture.

Table 2: Matrix transposition - run times and bandwidth

Size	Cores	Run 1 [s]	Run 2 [s]	Run 3 [s]	Run 4 [s]	Average [s]	Bandwidth [GB/s]
	4	8.4412	8.3528	8.6755	8.3326	8.4505	0.76
	8	4.3593	4.4382	4.2855	4.1860	4.3172	1.48
	16	3.2731	3.0261	2.2626	2.5232	2.7712	2.31
	32	3.1133	2.7508	2.5513	1.9700	2.5964	2.46
50000	1	45.9321	44.8595	45.1216	47.1102	45.7558	0.22
	2	23.8948	25.4520	25.4178	25.2524	25.0042	0.40
	4	13.8669	13.7983	13.6129	13.5577	13.7090	0.73
	8	8.6068	7.5582	8.0404	8.4306	8.1590	1.23
	16	4.1942	5.3581	5.3738	4.5863	4.8781	2.05
	32	3.3619	4.7703	3.2806	3.4283	3.7103	2.70

Table 3: Matrix transposition in blocks - run times and bandwidth

Size	Block size	Cores	Run time [s]	Bandwidth [GB/s]
20000	1000	1	2.3274	0.69
		2	1.3910	1.15
		4	1.0382	1.54
		8	0.3582	4.47
		16	0.1652	9.69
		32	0.0894	17.90
	2000	1	2.9142	0.55
		2	1.5654	1.02
		4	1.3991	1.14
		8	0.4399	3.64
		16	0.1971	8.12
		32	0.1029	15.55
	5000	1	3.3928	0.47
		2	1.9972	0.80
		4	2.1841	0.73
		8	0.5174	3.09
		16	0.2396	6.68
		32	0.1340	11.94
	10000	1	5.6197	0.28
		2	2.8618	0.56
		4	2.0601	0.78
		8	0.8063	1.98
		16	0.3628	4.41
		32	0.1888	8.47
30000	1000	1	4.8689	0.74
		2	2.9412	1.22
		4	1.3300	2.71
		8	1.0382	3.47
		16	0.4314	8.34
		32	0.1845	19.51
	2000	1	7.0540	0.51
		2	3.6089	1.00
		4	2.8164	1.28
		8	1.6397	2.20
		16	0.9963	3.61
		32	0.2399	15.00
	5000	1	7.8568	0.46
		2	4.2641	0.84
		4	3.2850	1.10

Table 3: Matrix transposition in blocks - run times and bandwidth

Size	Block size	Cores	Run time [s]	Bandwidth [GB/s]
		8	1.2891	2.79
		16	0.5993	6.01
		32	0.3935	9.15
	10000	1	13.1201	0.27
		2	7.1527	0.50
		4	5.2946	0.68
		8	2.2368	1.61
		16	0.9249	3.89
		32	0.4554	7.91
40000	1000	1	8.7225	0.73
		2	5.3019	1.21
		4	4.8171	1.33
		8	2.8220	2.27
		16	0.9196	6.96
		32	0.5399	11.85
	2000	1	12.2732	0.52
		2	6.6707	0.96
		4	4.0177	1.59
		8	2.1598	2.96
		16	1.1351	5.64
		32	0.6754	9.48
	5000	1	13.4611	0.48
		2	7.4703	0.86
		4	6.8054	0.94
		8	2.4402	2.62
		16	1.0151	6.30
		32	0.8766	7.30
	10000	1	24.0736	0.27
		2	12.3900	0.52
		4	9.0544	0.71
		8	4.3846	1.46
		16	1.7084	3.75
		32	0.8562	7.47
50000	1000	1	12.0920	0.83
		2	7.6869	1.30
		4	5.0051	2.00
		8	3.0580	3.27
		16	1.3538	7.39
		32	0.6621	15.10
	2000	1	20.3203	0.49
		2	10.6947	0.94
		4	8.7424	1.14
		8	4.4028	2.27
		16	1.6551	6.04
		32	1.1047	9.05
	5000	1	21.5054	0.47
		2	11.5888	0.86
		4	7.6854	1.30
		8	4.1386	2.42
		16	2.0332	4.92
		32	0.8638	11.58
	10000	1	37.0605	0.27
		2	19.4318	0.51
		4	13.5048	0.74

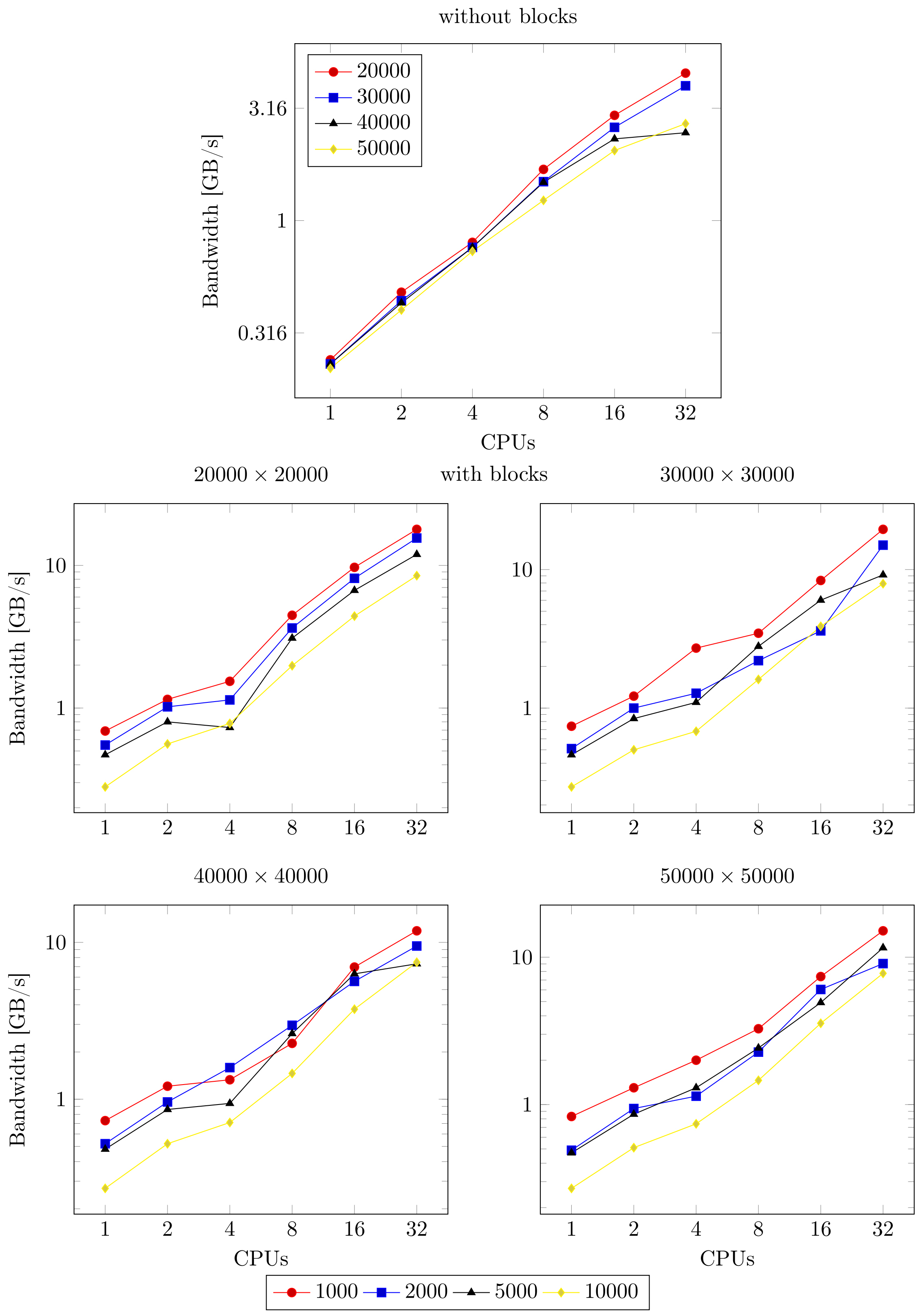
Table 3: Matrix transposition in blocks - run times and bandwidth

Size	Block size	Cores	Run time [s]	Bandwidth [GB/s]
		8	6.8553	1.46
		16	2.8058	3.56
		32	1.2871	7.77

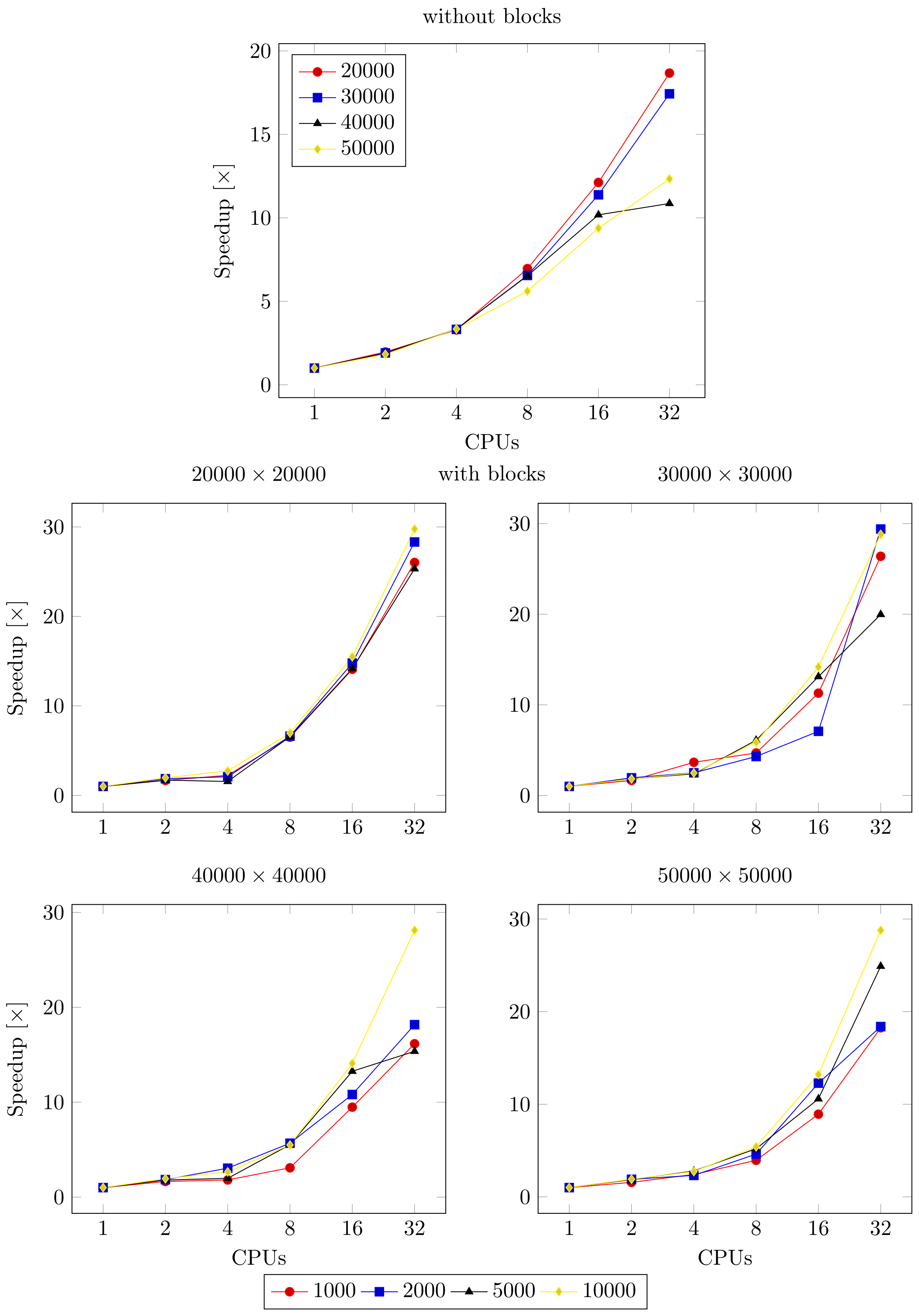
The parallel scalability of the algorithm was evaluated using the bandwidth as the performance metric. Plot 4, Plot 5, and Plot 6 visualize the performance, the speedup, and the efficiency respectively.

As expected, the algorithms achieve a better performance the more CPUs they have access to. The same can be said about the speedup: the higher the number of CPUs, the faster the calculations are completed. Unfortunately, the efficiency is all over the place because the algorithms are not linearly scalable [5]. Implementing the parallelization differently or using another algorithm altogether could be some of the possible fixes for this undefined yet mathematically correct issue.

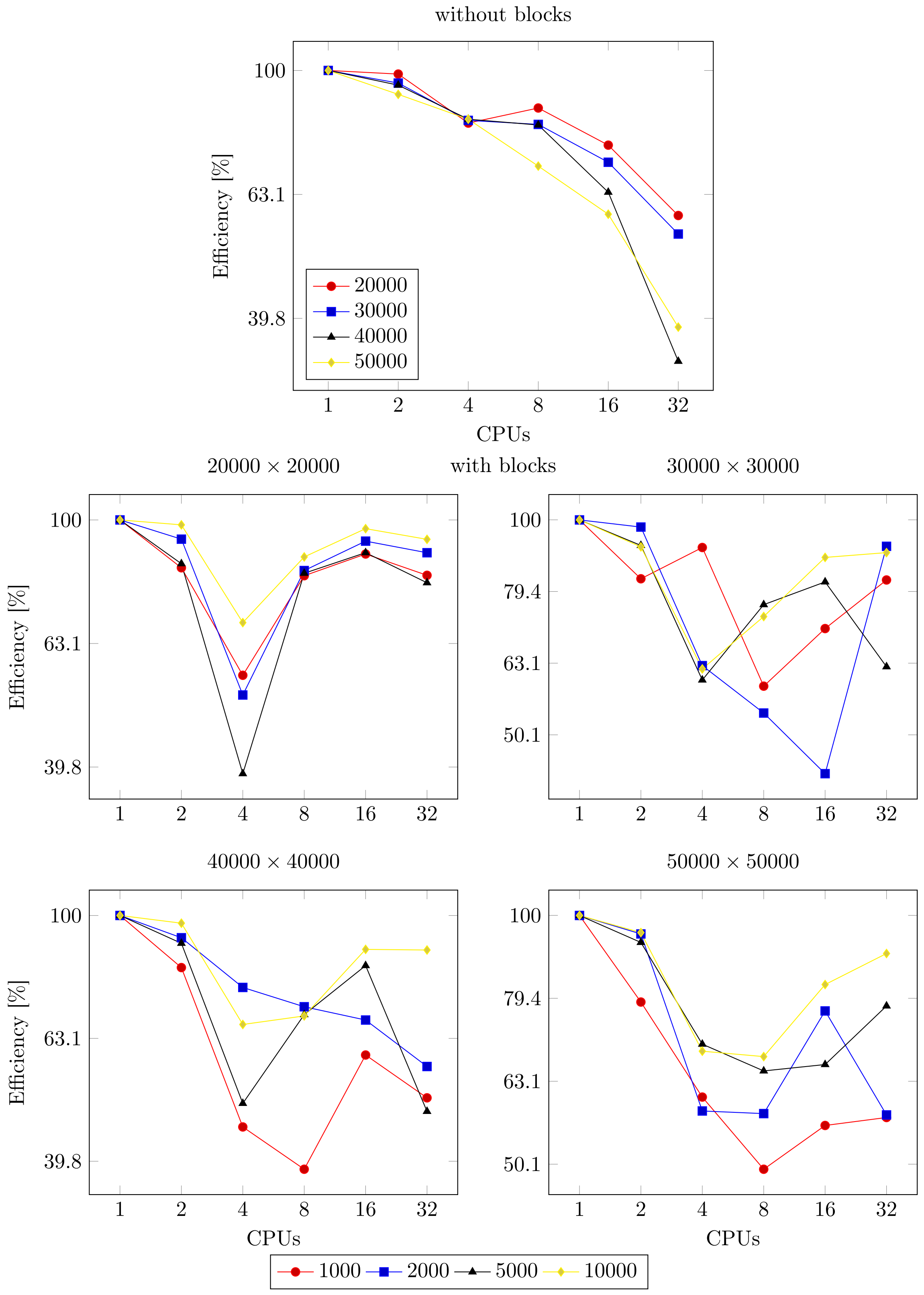
Plot 4: Parallel Matrix Transposition - Performance graph by size of matrices



Plot 5: Parallel Matrix Transposition - Speedup graph by size of matrices



Plot 6: Parallel Matrix Transposition - Efficiency graph by size of matrices



Conclusions

The analyzed problems highlight the advantages of parallel processing over sequentially run algorithms in terms of speed and achievable performance. This improvement does not strictly follow a linear pattern with efficiency challenges due to non-linear scalability. The challenges can be addressed by using different methods or more sophisticated algorithms that are optimized for the given hardware. One of the most trivial ways is to differentiate the algorithms for the specific type of matrix and perform separate scaling tests to compare them. This report provides only insights into simple academic problems that could represent real-world applications even if they are overly simplified.

References

[1] OpenMP ARB. *OpenMP 3.1 API C/C++ Syntax Quick Reference Card.* 2011. URL: `https://www.openmp.org/wp-content/uploads/OpenMP3.1-CCard.pdf`.

[2] OpenMP ARB. *OpenMP Application Program Interface.* 2011. URL: `https://www.openmp.org/wp-content/uploads/OpenMP3.1.pdf`.

[3] Noble G, Nalesh S, and Kala S. "Accelerators for Sparse Matrix-Matrix Multiplication: A Review". In: *2022 IEEE 19th India Council International Conference (INDICON).* 2022, pp. 1–6. DOI: `https://doi.org/10.1109/INDICON56171.2022.10039979`.

[4] Sumaia Al-Ghuribi and Khalid Thabit. "Matrix Multiplication Algorithms". In: *International Journal of Computer Network and Information Security* 12 (Feb. 2012), pp. 74–79.

[5] J.S. Kowalik. "Scalability of Parallel Systems: Efficiency Versus Execution Time". In: *High Performance Computing.* Ed. by J.J. Dongarra et al. Vol. 10. Advances in Parallel Computing. North-Holland, 1995, pp. 39–47. DOI: `https://doi.org/10.1016/S0927-5452(06)80005-5`.

[6] Professor Laura del Río Martín. *OpenMP Techniques for Shared Memory Architectures.* Lecture 10 slides for professor's Vella course Introduction to parallel computing. 2023. URL: `https://didatticaonline.unitn.it/dol/pluginfile.php/1776866/mod_folder/content/0/IntroPARCO-L10.pdf?forcedownload=1`.