Introduction to Parallel Computing. Homework 1: Exploring Implicit and Explicit Parallelism with OpenMP

Riccardo Bassan
Università di Trento
238793
riccardo.bassan@studenti.unitn.it

***Abstract*—This project's main goal is to create two functions, one that checks if a matrix is symmetric and one that computes a transposed matrix. After that, it is required to parallelize them using implicit parallelization and OpenMP parallelization and analyze performances for varying matrix sizes and different numbers of threads.**

I. Introduction of the problem and importance

The project is intended to study various methods to parallelize a function with implicit and explicit parallelization (OpenMP) and analyze their performances to find the most efficient implementation.
To check the symmetry of a matrix we need to verify if each element above the main diagonal is equal to the corresponding element in the lower part. The transpose matrix of an assigned matrix is obtained by exchanging rows and columns.
In the realm of high-performance computing, optimizing algorithms for large matrices is a quintessential challenge. As datasets continue to grow in size and complexity, traditional sequential processing becomes a bottleneck.
In this assignment, I have explored implicit parallelization, which is a method that allows a compiler to automatically exploit the parallelism, and explicit parallelization through OpenMP, which is an application programming interface that supports multi-platform shared-memory multiprocessing programming in C, C++, and Fortran.

II. State-of-the-art

Matrix transposition is one of the most common methods used for matrix transformation in matrix concepts across linear algebra. It is especially useful in areas such as image processing, machine learning, data science, and cryptography. Current research on matrix transposition spans a variety of computational and application domains:

- Optimized algorithms: studies explore parallel pipelined architectures and modular designs to reduce latency and memory usage during matrix transposition.
- GPU and memory efficiency: techniques for asynchronous in-place transposition improve memory bandwidth and computational efficiency, especially for large-scale GPU workloads.
- Evolutionary algorithms: the matrix transpose operation is used in innovative algorithms like Estimation of Distribution Algorithms (EDA) for enhanced optimization in Bayesian learning and data modeling.

While recent research offers significant advances in performance and scalability, certain limitations persist, such as hardware dependency, memory constraints, and algorithm complexity.

III. Contribution and Methodology

A. Symmetry check

1) Sequential implementation: In the function checkSym I scroll through a given matrix with two nested loops and verify if every data is equal to its corresponding one on the opposite side to the main diagonal. When a value that doesn't meet this check is found, a boolean value is set to false, and, at the end of matrix scrolling, it is returned.

2) Implicit parallelization: I combined two optimization techniques to improve performance:

- #pragma simd: this directive is applied to the outer loop and instructs the compiler to vectorize the loop, enabling Single Instruction Multiple Data processing. It reduces the number of iterations by processing multiple elements at a time.
- #pragma unroll(dynamic): this directive tells the compiler to unroll the inner loop by a factor that isn't fixed but it is decided by the compiler or at runtime, based on the loop size and CPU capabilities. It reduces the loop control overhead and improves efficiency by leveraging parallel processing.

Symmetry check implicit parallelization code:

```
#pragma simd
for(int i=0; i<n; i++){
    #pragma unroll(dynamic)
    for(int j=i+1; j<n; j++){
        if(matrix[i][j]!=matrix[j][i]){
            check = false;
        }
    }
}
```

3) Explicit parallelization with OpenMP: The code is equal to the sequential implementation with two additional OpenMP directives:

- #pragma omp parallel for shared(check): creates a parallelized loop where each thread independently processes a subset of matrix rows, shared(check) indicates that the variable check is shared among threads.
- #pragma omp flush(check): ensures that the check value is visible to all threads after any updates, this prevents a thread from working with a stale value of the variable.

Symmetry check explicit parallelization code:

```
#pragma omp parallel for shared(check)
for(int i=0; i<n; i++){
   #pragma omp flush(check)
   for(int j=i+1; j<n; j++){
      if(matrix[i][j]!=matrix[j][i]){
         check = false;
      }
   }
}
```

B. Matrix transposition

1) Sequential implementation: This function performs a matrix transposition, where the rows and columns of the input matrix become respectively columns and rows of the transpose matrix. This is done using two nested loops that scroll through the matrices.

2) Implicit Parallelization: To make better use of the cache, I divided the code into blocks. Outer loops divide the $n \times n$ matrix into blocks of size $SIZE \times SIZE$, while inner loops iterate over each element in the current block. I chose a $SIZE$ of 64 based on the hardware's cache size to maximize performance, this reduces memory jumps and improves cache access efficiency.

3) Explicit parallelization with OpenMP: For the OpenMP implementation, the code is the same as the implicit parallelization, but, before the outer loop, I added the directive #pragma omp parallel for collapse(2). This allows to parallelize nested loops effectively, combining multiple loops into a single iteration space for better workload distribution and performance thanks to the collapse clause.

Matrix transposition explicit parallelization code:

```
#pragma omp parallel for collapse(2)
for(int i=0; i<n; i+=SIZE){
  for(int j=0; j<n; j+=SIZE){
    for(int ii=i; ii<i+SIZE && ii<n; ii++){
      for(int jj=j; jj<j+SIZE && jj<n; jj++){
        transpose[ii][jj] = matrix[jj][ii];
      }
    }
  }
}
```

IV. Experiments and System Description

A. System description

For the project's development, I used the UniTn HPC Cluster, which consists of a CPU and GPU-based architecture based on the PBS queue management system.
It consists of:

- 2 head nodes
- computing nodes: 142 CPU for a total of 7674 nodes, 10 GPU for a total of 448128 CUDA nodes
- Ram of 65 TB
- Ethernet, Infiniband, and Omni-Path networks
- shared storage
- total theoretical peak performance of 478.1 TFLOPs (CPU: 422.7 TFLOPs, GPU: 55.4 TFLOPs)

B. Experiments

The experiments were conducted by submitting the job in the short_cpuQ queue with these resources: 1 node, 64 CPUs, 64 threads, and 1mb memory.
I wrote the code in C programming language, and some libraries that I used for it are:

- <omp.h> that allows the use of OpenMP parallel instructions and wall clock timers.
- <stdbool.h> to use boolean values.

For every function required, I used wall clock timers to estimate the time taken from the implementations and print them to be able to do a comparison. Each matrix transposition function runs the transposition code ten times and computes an average of the times to improve the accuracy of the experiments.
Then for the explicit parallelization of the matrix transposition, I created a for that does the ten runs for every number of threads between 1 and 64 (1,2,4,8,16,32,64), and for each of them calculates the average time, the speedup, the efficiency, and the bandwidth. These data are printed in the output, but I also wrote a code that creates a .csv file and saves in it all the results of the experiments that I run multiple times for different matrix sizes from 16 to 4096.
Other functions and checks I created to help me in the experimental phase are:

- Function checkTrans: used to check if the transposition made is correct.
- Function print: to print matrices.
- Code to check if the matrix size n is a power of two.
- Initialization code to initialize the matrix with random float numbers.

I tested the program with various compiler optimization flags comparing results and discovered that the best one is $-O2$. It is a compiler optimization level that controls how the compiler optimizes the code balancing performance improvements with code size, it includes basic optimizations as well as loop unrolling, vectorization, and better instruction scheduling.

V. Results and Discussion

Table 1 shows sequential and implicit execution time for each matrix size. One can notice that my implementation of the matrix transposition implicit parallelization starts to work when sizes are larger than 256×256. This happens because the computational work grows disproportionately to the overhead, allowing parallel resources to be used to their full potential.

TABLE I
Execution time

Size	*Sequential Time*	*Implicit Time*
16x16	***0,0000002***	***0,0000002***
32x32	***0,0000006***	***0,0000007***
64x64	***0,0000022***	***0,0000026***
128x128	***0,0000098***	***0,0000112***
256x256	***0,0000394***	***0,0000452***
512x512	***0,0002703***	***0,0002055***
1024x1024	***0,0011918***	***0,0007661***
2048x2048	***0,0196110***	***0,0073994***
4096x4096	***0,0917636***	***0,0592718***

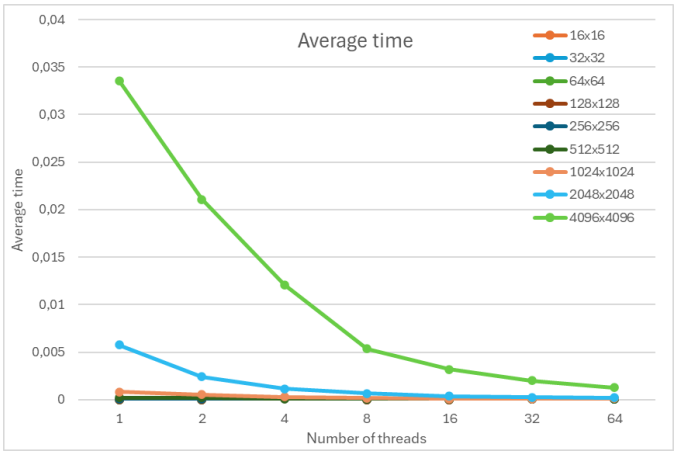

Fig. 1. Enter Caption

The first graph represents the average execution time of the parallelized matrix transposition with OpenMP compared with the number of threads used. The more threads are used, the faster the code is executed, this can be observed especially with larger matrix sizes because of a better ratio between computational work and overheads associated with parallelization.

The second table and graph describe for each matrix size his effective bandwidth which is the actual data transfer rate from memory to CPU achieved during real operations. The formula is

$$Bandwidth = \frac{Data\ Transferred}{Time\ Taken} \quad (1)$$

where $Data\ Transferred$ is $2 \times n \times n \times sizeof(float)$ and the $Time\ Taken$ is the average of the ten runs.
The table compares sequential and implicit parallelization bandwidth, it shows that the second approach is ineffective for small matrices, overcoming the sequential implementation

TABLE II
Bandwidth

Size	*Sequential Bandwidth*	*Implicit Bandwidth*
16x16	***9,9684***	***9,4785***
32x32	***12,7665***	***10,9622***
64x64	***14,3341***	***11,6520***
128x128	***13,2509***	***11,6553***
256x256	***13,3012***	***11,6082***
512x512	***7,7580***	***10,2034***
1024x1024	***7,0386***	***11,0129***
2048x2048	***1,7110***	***4,5375***
4096x4096	***1,4626***	***2,2644***

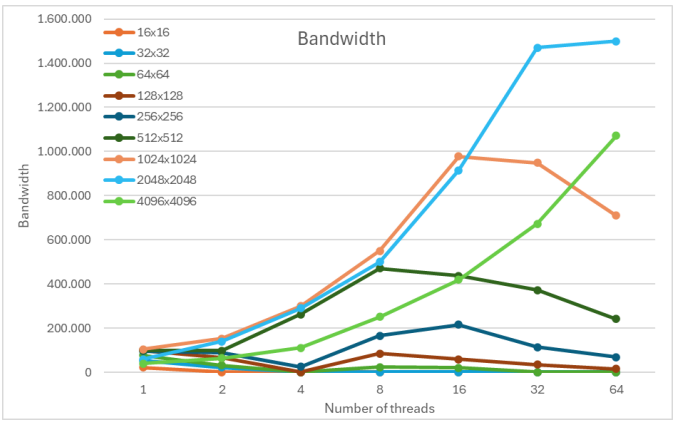

Fig. 2. Bandwidth

with bigger dimensions. The graph, instead gives us bandwidth variation due to the number of threads through which we can say that large blocks benefit more from a high number of threads until a limit is reached beyond which saturation effects appear. To optimize performance is necessary to balance the number of threads and matrix dimensions based on system capabilities.

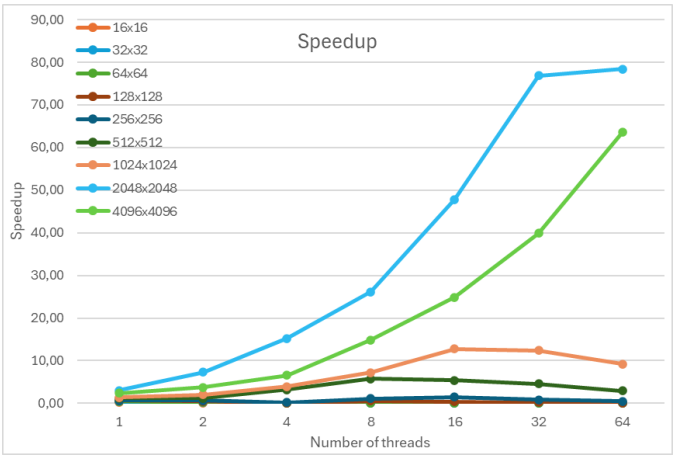

Fig. 3. Speedup

Fig.3 graph represents the speedup for every matrix size compared with the number of threads. It is calculated with the

formula:

$$Speedup = \frac{T_s}{T_p} \tag{2}$$

where T_s is the time of the sequential implementation and T_p is the time taken from the OpenMP parallelization. It measures how the execution time of a parallel application changes as the number of threads increases.
It can be observed that with smaller-size matrices the speedup variation due to the number of threads is small because of the very low execution time, but with larger matrices, it increments exponentially until 32 threads, then it tends to decrease because of the excessive number of processes.

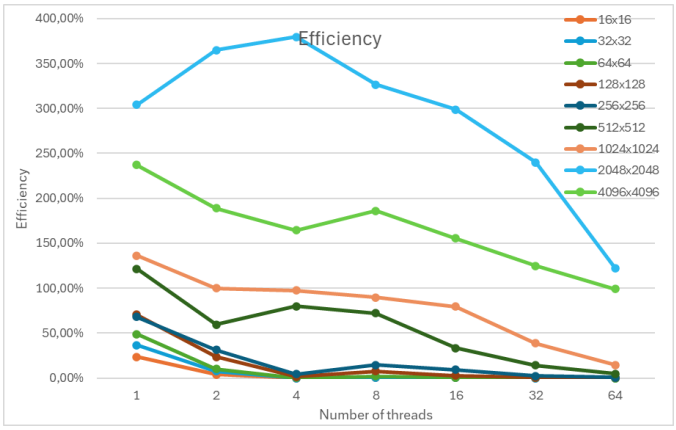

Fig. 4. Efficiency

Fig.4 shows the efficiency in percentage compared with the number of threads for every matrix size. Its formula is:

$$Efficiency = \frac{Speedup}{N_t} \times 100 = \frac{T_s}{N_t T_p} \times 100 \tag{3}$$

where N_t is the number of threads. It is a measure of how effectively the computational resources are utilized when executing the program in parallel.
One can notice that efficiency tends to decrease when the number of threads increases because of synchronization and communication overhead, and memory bottlenecks.

VI. Conclusions

The project demonstrated that parallelization is an optimal way to improve matrix transposition performance, I have experimented with two methods to do this concluding that the implicit approach provides small improvements since it is limited by factors like data dependence and an inefficient automatic memory allocation of workload. OpenMP parallelization, instead, emerges as a powerful ally in the quest for performance optimization, providing a flexible and accessible framework for parallel computing thanks to its automatic thread management, dynamic scheduling, and built-in synchronization techniques.

References

[1] Cuemath.com, "Matrix transposition". Available: https://www.cuemath.com/algebra/transpose-of-a-matrix/
[2] Daa-Won Kim, Cornell University, "Estimation of Distribution Algorithms with Matrix Transpose in Bayesian Learning". Available:https://arxiv.org/abs/2407.18257
[3] Mark Harris, "An Efficient Matrix Transpose in CUDA C/C++". Available:https://developer.nvidia.com/blog/efficient-matrix-transpose-cuda-cc/
[4] "Harnessing Parallel Computing Power of OpenMP Optimization for Large Matrices". Available: https://researchtech.net/index.php/2024/01/harnessing-parallel-computing-openmp-optimization-matrices/

GIT repository link: https://github.com/RicBas03/Exploring-Implicit-and-Explicit-Parallelism-with-OpenMP