Report of Homework 1: Exploring Implicit and Explicit Parallelism with OpenMP

Marco Miglioranza
ID Student: 235630
marco.miglioranza@studenti.unitn.it

I. Abstract

The aim of the project is the implementation and analysis of a number of parallelization techniques that are applicable to the problem of matrix transposition and matrix symmetry verification. Making comparisons among a sequential implementation, implicit parallelism provided by optimization techniques, and explicit parallelism based on OpenMP, the main goal is to measure the execution time taken, scalability, and efficiency of different approaches.

The three methods have been tested in the range of matrix dimensions from 16×16 to 4096×4096.

The principal findings indicate that OpenMP realizes considerable enhancements in performance when applied to larger matrices, especially for the complexity of used strategies, whereas implicit methods yield only moderate improvements. This report provides the data development, execution, and evaluation of the proposed methods, concentrating on system performance including speedup, efficiency gains, and possible bottlenecks.

II. Introduction of the problem and importance

Matrix transposition is a fundamental operation in various domains such as robotics, mathematics and physics. Also are important in many computational domains, including scientific computing, machine learning, and data processing, all of which require fast data manipulation. The effective execution of matrix transposition holds considerable importance within high-performance computing (HPC) environments and extensive systems, where even small optimization can lead substantial advancements in the execution time and resource usage. Hence, it is necessary to develop some technique that improves the performance since conventional sequential techniques, although simple, are progressively insufficient as data volumes expand and computational frameworks transition towards multi-core and parallel processing. The goals that project goes through are:

1) Design and implement three methods of matrix transpose and symmetry verification: sequential, implicit parallelization, and explicit parallelization using OpenMP
2) Identify performance bottlenecks of each approach and suggest possible optimizations
3) Illustrate which is the best technique to use for the project problem

III. State of the art

A. Introduction

Matrix transposition and related operations are critical operation for the HPC systems as it often involve processing massive datasets and require a highly efficient computation. Several current research point toward cache-aware methods, parallel processing, and hybrid approaches that include CPU-GPU integrations. The tools and techniques used are listed as follows:

- **Cache Optimization** in dense matrix operation is one of the most used techniques. It is possible to optimize the use of the cache by
  - **Blocking/Tiling:** Tile matrices into submatrices to be more compatible with CPU cache and reduce memory access letancy.
  - **Contiguous Memory Allocation:** Arranges rows or columns of a matrix into consecutive addresses of memory in order to avoid cache misses
- **OpenMP** it is used for optimizing performance on CPU-level operations by distribuiting the workload among threads. Advanced optimizations, thus, using SIMD and loop tiling further enhance this performance. In the modern CPU architecture also it is possible to use **Hyperthreading** (multiple threads runs on a single core simultaneously) and **Vectorialization**.
- **GPU Acceleration: CUDA** process thousands of threads simultaneously. Modern approaches tries to have an hybrid solution with GPU-CPU use with OpenMP.
- **User Libraries** pretty efficient runtime-optimized implementations of the most inner functions of linear algebra by making use of hardware-specific optimizations. Some example are **LAPACK** or **BLAS** for linear algebra operation and **Armadillo** or **Eigen** for interface and template header for matrix and vector operations.

B. Limitations of Existing Techniques

Although research on dense matrix operation are well-studied, there are still visible limitation of the existing techniques.

- Scalability Issues: This restricts OpenMP, relying on shared memory, to single-node systems, whereas MPI can tackle higher scales in systems with distributed memory effectively

- Native functions: Native libraries do not possess inherent parallelizability, necessitating custom implementations to utilize parallel processing
- Memory Bottolnecks: Large, dense matrices are extremely demanding in terms of memory resources
- Gaps: Exist areas where the research has not fully discover the potential and limitation of the current techniques used:
  - Systematic Benchmarking: This area lacks in-depth research in comparing a number of parallelization methodologies for transposing dense matrices on an extensive range of hardware.
  - Hybrid approaches: Integration with OpenMP and GPU-based frameworks remains largely unexploited.
  - Energy Efficiency: While the performance metrics are well-studied, energy consumption has received limited attention in parallel implementations.

C. Contributions of This Work

This project will attempt to bridge some of these gaps by:
- Implementing and benchmark parallel transposition and simmetry verification algorithms with OpenMP for dense matrices.
- Exploring the performance on various optimization techniques, including blocking and thread scheduling.
- Provide insight into the scalability of the OpenMP implementations and propose strategies that will reduce any bottlenecks found

IV. Contribution and Methodology

The functions of symmetry verification and matrix transposition were implemented in C++, enhanced with different optimization techniques and OpenMP pragmas in order to utilize implicit/explicit parallelism capabilities of modern computing systems. The following subsections describes the contributions, computational enviroments, tools and the pseudo code of the various algorithms

A. Tools and Technologies:

1) Explicit parallelism - Tools: The only tool used is OpenMP that thanks to his optimizations on CPU level gives key benefits:

- Transposition:
  - Parallelization: OpenMP shares the workload among several CPU cores; hence, it is able to handle big matrices efficiently.
  - Efficient resource utilization: leverages all available CPU cores, thereby scaling up throughput on multi-core systems
- Symmetry Verification:
  - Same Benefits of Transposition
  - Load Balancing: OpenMP offers scheduling strategies to distruibute the workload across the threads
  - Synchronization: OpenMP provides several mechanism to ensure the correct behaviour when threads do write and read operation in shared resources

2) Explicit parallelism - Algorithm:

matTransposeOMP(T, M, N)
Input M: matrix N x N
Input N: size of matrix
output T: matrix N x N
BEGIN
[IN PARALLEL]
FOR i = 0 **TO** N - 1, i+=BLOCK_SIZE **DO**
FOR j = 0 **TO** N - 1, j+= BLOCK_SIZE **DO**
FOR ii = i **TO** $min(i + BLOCK\_SIZE, N) - 1$ **DO**
FOR jj = j **TO** $min(j + BLOCK\_SIZE, N) - 1$ **DO**
T[j][i] = M[i][j]
END FOR
END FOR
END FOR
END FOR
END The complexity of this algorithm is O(n^2). The IN PARALLEL clause indicates that the instruction to which it is associated will be executed in parallel. In the code the parallelization use the pragma `#pragma omp for parallel collapse(2)` that combines the outer loops (i, j).

checkSymOMP(M, N)
Input M: matrix N x N
Input N: size of matrix
output sym: Boolean for matix symmetry
BEGIN
BEGIN PARALLEL REGION
local_sym = true
[IN PARALLEL]
FOR i = 0 **TO** N - 2 **DO**
IF sym = false **THEN**
CONTINUE
ENDIF
FOR j = i + 1 **TO** N - 1 **DO**
IF M[i][j] $\neq$ M[j][i] **THEN**
local_symmetry = false
symmetry = false
BREAK
ENDIF
END FOR
END FOR
OMP Critical: $symmetry = symmetry \ \& \ local\_symmetry$
END PARALLEL REGION
END

The complexity of this algorithm is O(n^2). In the code the parallelization use the pragma `#pragma omp parallel` for the parallel regione and `#pragma omp for schedule(guide) nowait` for the loop.

3) Implicit parallelism - Tools: The only tools used for the implicit parallelism are the AVX and SSE instruction set extension that combines with the optimization technique gives results similar to OpenMP. The key benefits are:

- Portability: VX/SSE optimizations make sure that performance scales properly across CPUs having different numbers of cores and SIMD widths
- Efficient for Dense data: Implicit functions are good at dealing with large, dense matrices. Their vectorized nature makes them perform well even with a lot of data, where the performance does not degrade much

4) Implicit parallelism - Algorithms: **transposeIMP(T, M, N)**

```
Input: M: matrix N x N
Input: N size of matrix
Output T: matrix N x N
BEGIN
IF N ≤ 128 THEN
SERIAL_TRANSPOSITION WITH:
PREFETCHBY4 ∧ (VECTBY4 ∨ LOOPBY4)
ELIF N ≤ 1024 THEN
BLOCKING_TRANSPOSITION WITH:
PREFETCHBY4 ∧ (VECTBY4 ∨ LOOPBY4)
ELSE BLOCKING_TRANSPOSITION WITH:
PREFETCHBY8 ∧ (VECTBY8 ∨ LOOPBY8)
```

checkSymIMP(T, M, N)

```
Input: M: matrix N x N
Input: N size of matrix
Output T: matrix N x N
BEGIN
IF N ≤ 128 THEN
SERIAL_SYMMETRY_CHECK WITH:
PREFETCHBY4 ∧ (VECTBY4 ∨ LOOPBY4)
ELIF N ≤ 1024 THEN
BLOCKING_SYMMETRY_CHECK WITH:
PREFETCHBY4 ∧ (VECTBY4 ∨ LOOPBY4)
ELSE BLOCKING_SYMMETRY_CHECK WITH:
PREFETCHBY8 ∧ (VECTBY8 ∨ LOOPBY8)
```

Notes: The terms **PREFETCHBY**, **VECTBY**, and **LOOPBY** are not actual implementation details but **descriptive acronyms** used to indicate the techniques applied:

- **PREFETCHBY**: Represents the prefetching mechanism
- **VECTBY**: Refers to the vectorization of operations using instruction.
- **LOOPBY**: Loop Unrolling
- The number 8 or 4 represent the number of data or istruction done from the techniques (EG. LOOPBY8 indicates a set 8 istruction of unrolling)

The techniques perform the basilar algorithm of the transpose $T[i][j] = M[j][i]$, and the symmetry of the matrix by checking if $M[i][j] = M[j][i]$ for all i, j.

B. Methodology applied and challenges faced

1) Explicit Parallelism:

- Transposition: The two outer loops in matrix are collapsed by using the OpenMP directive pragma omp parallel for collapse(2); that allows the row and column loops to form one large iteration space to distribute the threads efficiently. It is used the collapse(2) directive because using collapse(4) would result in excessive granularity and thread management overhead. Furthermore, the innermost loops perform operation which depends on which block they are (the inner loops are depending on the outerloop). It also combines the blocking technique with a friendly cache dimension (32x32). This ensures a better exploitation of the cache and reduced memory latency
- Symmetry: It is used the nowait pragma in order to enhances performance by deleting the implicit barrier at the end of the for and prevents the threads from waiting for other threads to finish their iterations. This will be beneficial when there is no dependency like in the case of check symmetry. The parallel region is used instead of relying solely on collapse to create a persistent team of threads that can execute multiple work-sharing constructs. This minimizes the overhead of repeatedly creating and destroying threads for each loop, improving efficiency. Combining parallel with collapse ensures optimal thread utilization while maintaining scalability for large workloads.

2) Implicit Parallelism:: The use of AVX-512 and SSE instruction extensions ensures code portability on different hardware architectures while attaining high performance in a more implicit implementation. For small matrices, this overhead from reliance entirely on AVX-512 could be costly since a larger vector width is not being entirely used. Following this, the implementation has been supplemented by fallback provisions using AVX or SSE instructions for older systems without AVX-512.

The pointer-to-pointer structure used for the matrix is double**, purposely chosen to have contiguous memory layouts. Contiguous data is the only thing that SSE and AVX extensions can operate efficiently on and cannot be guaranteed.in the case of vector-of-vector structures (std::vector).

This typology from a programming point of view takes better advantage of any SIMD instructions to the fullest extent possible regarding parallelism, even in smaller matrix sizes, while effectively holding strong performance and compatibility across all platforms.

V. EXPERIMENTS

The algorithm were run on the HPC server, in the CPU 11 that is an Intel(R) Xeon(R) Gold 6252N @ 2.30 GHz with 4 Cores. The architetcure of the memory is x86_64 with the virtualization enabled. The compiler used for the test was GCC and the flag used are reported as follows:

- march=native in order to enable all the set instruction extension (AVX and SSE are implemented in the CPU, visible through the command `lscpu`)
- ffast-math to operate aggressive operation on floating point
- funroll-loops in order to unroll loops O2 Gives reasonable compilation time and optimize the performance
- fopenmp to enable OpenMP
- There were not used flag for vectoralization for the follows two main reason:
  - The flag needed for the loop -ftree-loop-vectorize is already enabled in O2. It was observed that other vectorization flags worsen performance. It is hypothesized that this performance degradation is due to nested functions in the code, which handle vectorization, prefetching, and related operations.
  - Adding other flags could get worse performance also for the other functions since more flags require more compilation time and reduce the precision of the translation of the code into machine language

1) Hypotesis:

- Sequential Implementation: Expected to show linear scaling towards size of the matrix with an O(N^2) complexity. performance worsens tremendously as the matrix grows. Hence, sequential methods become less and less efficient with increased matrix size.

- Implicit Parallelism (SIMD): SIMD startup overhead and small memory reductions fit well to light operations for small matrices and there is a significantly reduced execution time through parallel access to multiple elements. For large matrices instead performance enhancement may reach a plateau driven by bandwidth limits and cache efficiency reductions.
- Explicit Parallelism (OpenMP): For small matrices OpenMP will not perform due to thread management overhead, although the amount of work per thread is inadequate to make up the cost of the threads being parallelized. Then, for large matrices OpenMP's performance, should surpass SIMD and sequential. By increasing the number of thread the time of execution should decrease drastically
- General Aspected Trends: Small Matrices SIMD will beat OpenMP which is even more worse than sequential. For Medium Matrices the perfomance should be comparable. Large matrices OpenMP will beat SIMP especially when added more threads.

2) Results:

TABLE I
AVERAGE RESULTS WITH 8 THREADS

Algorithm	Size of the N:				
	16	128	1024	2048	4096
Transpose	5,01E-07	3,20E-05	4,84E-03	4,04E-02	1,84E-01
TransposeIMP	2,99E-06	1,67E-05	1,13E-03	5,08E-03	2,89E-02
TransposeOMP	3,68E-04	3,88E-04	1,70E-03	4,97E-03	1,97E-02
checkSym	2,98E-07	4,91E-05	2,43E-03	2,58E-02	1,15E-01
checkSymIMP	3,35E-07	5,17E-06	3,82E-04	2,45E-03	1,83E-02
checkSymOMP	5,42E-05	5,40E-05	3,76E-04	2,84E-03	1,30E-02

TABLE II
AVERAGE RESULTS IN A MATRIX OF 4096X4096 WITH DIFFERENT NUMBER OF THREADS

Algorithm	Number of threads:				
	1	4	8	32	128
TransposeOMP	1,04E-01	3,81E-02	1,97E-02	8,06E-03	1,05E-02
checkSymOMP	5,52E-02	2,40E-02	1,30E-02	4,70E-03	3,72E-03

A. Discussion of the results

The results of the experiments conducted are presented in Tables I and II. Table I shows the average execution times for different matrix sizes and algorithms, while Table II highlights the impact of varying the number of threads on performance.

As shown in Table I, the performance trends observed align closely with the initial hypotheses outlined in Sequential methods exhibit linear scaling but suffer from inefficiency as matrix size increases, whereas implicit parallelism shows moderate improvements with diminishing returns for larger matrices. Explicit parallelism demonstrates superior performance for large matrices.

Table II illustrates that, as the number of threads increases, the execution time improves significantly. However, the speedup and efficiency observed deviate from the theoretical expectations.

The theoretical speedup S_{teo} for a perfectly parallelizable algorithm is directly proportional to the number of threads p, as given by $S_{\text{teo}} = p$. Consequently, the theoretical efficiency E_{teo} should remain constant at 100%: $E_{\text{teo}} = \frac{S_{\text{teo}}}{p} = 1$.
Using the data in Table II, the observed speedup S_{obs} and efficiency E can be calculated as: $S_{\text{obs}} = \frac{T_1}{T_p}, \quad E = \frac{S_{\text{obs}}}{p}$.

TABLE III
SPEEDUP AND EFFICIENCY RESULTS IN TRANSPOSITION

Threads	Speedup Teo	Speedup	Efficiency
1	1	1	1
2	2	2.08	0.69
4	4	2.74	0.66
8	8	5.29	0.59
16	16	9.37	0.40
32	32	12.96	0.09

TABLE IV
SPEEDUP AND EFFICIENCY FOR SYMMETRY CHECK

Threads	Speedup Teo	Speedup	Efficiency
1	1	1.00	1
2	2	1.36	0.68
4	4	2.30	0.57
8	8	4.24	0.53
16	16	7.84	0.49
32	32	11.75	0.37

The degradation becomes more pronounced with an increasing number of threads, as seen with 128 threads, where the efficiency diminishes significantly due to the limited scalability of the shared memory architecture.

VI. CONCLUSION

OpenMP excels for large matrices, while SIMD techniques are more effective for small matrices. As described in the state-of-the-art and hypotheses, SIMD techniques significantly improve performance, benefiting both explicit and implicit parallelism. However, the results highlight limitations: OpenMP shows reduced efficiency as the number of threads increases (scalability issues), and vector-of-vector data structures were avoided due to their incompatibility with prefetching and vectorization in small matrices, requiring contiguous memory. Armadillo was also avoided since the library was not installed in the HPC server. Hybrid integrations were not explored due to the lack of optimized research techniques. Large matrices revealed memory bottlenecks, limiting performance gains.

1) Key findings: The key findings of the project are:

- Integration of SIMD and OpenMP demonstrated complementary strengths, ensuring portability with fallback solutions for systems lacking AVX-512.
- Data structures ensuring contiguous memory (e.g., double**) optimized SIMD performance and aligned with hardware capabilities.
- Comprehensive testing validated scalability and effectiveness, contributing to advancements in high-performance matrix computations.

REFERENCES

[1] "IJCS: Parallel Matrix Multiplication Using OpenMP," International Journal of Computer Science, vol. 50, no. 2, pp. 243–249, 2023. [Online]. Available: https://www.iaeng.org/IJCS/issues_v50/issue_2/IJCS_50_2_27.pdf.

[2] Vini2, "Parallel Matrix Multiplication Using OpenMP," GitHub Repository. [Online]. Available: https://github.com/Vini2/ParallelMatrixMultiplicationUsingOpenMP.

[3] GNU Compiler Collection, "Optimize Options." [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

[4] C. P. Jones, "Parallel Matrix Transposition and Vector Multiplication Using OpenMP," in *Conference on High Performance Computing*, 2013, pp. 243–249. [Online]. Available: https://link.springer.com/chapter/10.1007/978-1-4614-6747-2_30.

[5] "std::vector," cppreference.com. [Online]. Available: https://en.cppreference.com/w/cpp/container/vector.

[6] CINECA, "KNL Vectorization," in *26th Summer School on Parallel Computing*, Bologna, 2017. [Online]. Available: https://hpc-forge.cineca.it/files/ScuolaCalcoloParallelo_WebDAV/public/anno-2017/26th_Summer_School_on_Parallel_Computing/Bologna/SCP-KNL-vectorization.pdf.

[7] C. Sanderson, "Armadillo: An Open Source C++ Linear Algebra Library for Fast Prototyping and Computationally Intensive Experiments," in *Proceedings of the International Congress on Mathematical Software*, 2010. [Online]. Available: https://www.researchgate.net/profile/Conrad-Sanderson/publication/49514965_Armadillo_An_Open_Source_C_Linear_Algebra_Library_for_Fast_Prototyping_and_Computationally_Intensive_Experiments/links/5d42789ea6fdcc370a714d0e/Armadillo-An-Open-Source-C-Linear-Algebra-Library-for-Fast-Prototyping-and-Computationally-Intensive-Experiments.pdf.