Homework 1: Exploring Implicit and Explicit Parallelism with OpenMP.

Dimitri Corraini
237963
dimitri.corraini@studenti.unitn.it

***Abstract*—This project explores both implicit and explicit parallelization techniques by implementing a matrix transpose operation and a symmetry verification. We will focus our analysis on square matrices of size $n \times n$ with n ranging from 2^4 to 2^{12}.**

I. Introduction and Background

A. Matrix Transposition and Its Importance

Matrix transposition is a fundamental operation that plays a critical role in various fields, including machine learning, computational fluid dynamics and computer graphics.

B. Importance of Matrix Symmetry

Verifying matrix symmetry is as important, if not more, than transposition. This is due to the fact that if a matrix A is symmetric, its transpose will follow $A^T = A$, which saves valuable computation time by eliminating the need to swap values along the diagonal.

C. Project Objectives

The main objectives that this project aims to achieve include developing a solid understanding of parallelism, compare various techniques to identify the most efficient one and analyze how each proposed solution scales as the workload increases.

D. Project Repository

All the code and additional resources can be found on GitHub: Project Repository.

II. State-of-the-Art

After conducting online research, it is evident that the state-of-the-art in large matrix operations involves parallelizing tasks using GPUs [9] [10]. GPUs excel in this domain due to their numerous cores, enabling efficient hardware-level parallelism. In contrast, OpenMP employs software-level parallelism, leveraging the capabilities of multicore CPUs. However, relying on GPU parallelism can be a limitation for systems that lack sufficiently capable graphics processors. Therefore, my project focuses on such systems, aiming to provide robust parallelization solutions without the need for a GPU.

III. Contribution and Methodolgy

As stated previously, my project aims to achieve strong parallelism without relying on GPUs. While this goal may not be groundbreaking, considering my status as an undergraduate and the fact that many aspects are still new to me, I am committed to giving my best effort. After this introduction, we will explore the three methods I used to implement implicit parallelism:

1) Implicit V1
   This method utilizes the serial code and applies to it the compile flag `-O2`
2) Implicit V2
   This method is similar to the previous one but also adds flags enabling more optimization:
   `-ftree-loop-if-convert -ftree-loop-distribution -floop-interchange -ftree-vectorize -fvect-cost-model=unlimited -funroll-loops -march=native -ffast-math -mprefer-vector-width=512 -fprefetch-loop-arrays`
3) implicitBlock
   This method aims to create parallelism by utilizing by diving data into blocks to exploit data locality and improve cache hit rates. This implementation however is not cache aware.

Matrix Transposition with Block-Based Parallelism

Input: Matrix M of size $N\_SIZE \times N\_SIZE$
Output: Transposed matrix T
Initialize block_size as $N\_SIZE/4$
for each i from 0 to $N\_SIZE - 1$ with step 2 **do**
  for each j from 0 to $N\_SIZE - 1$ with step block_size **do**
    for each k from 0 to block_size - 1 **do**
      $T[j+k][i] = M[i][j+k]$
      $T[j+k][i+1] = M[i+1][j+k]$
    end for
  end for
end for

Matrix Symmetry with Block-Based Parallelism

Input: Matrix M of size $N\_SIZE \times N\_SIZE$

Output: Boolean value indicating if the matrix is symmetric
Initialize block_size as $N\_SIZE/4$
Initialize symmetric as `true`
for each i from 0 to $N\_SIZE - 1$ with step size block_size **do**
  for each k from 0 to block_size $- 1$ **do**
    for each j from $i+1+k$ to N_SIZE-1 **do**
      if $M[i+k][j] \neq M[j][i+k]$ **then**
        Set symmetric to `false`
      end if
    end for
  end for
end for

return symmetric

To implement OpenMP parallelism, I focused on two main approaches. The first involves enhancing the classic serial code with OpenMP directives, allowing for parallel execution of loops and tasks. The second where I modified the block approach seen before to add to it OpenMP parallelism. The following pseudo codes are the OpenMP block functions.

Matrix Transposition with Block-Based OpenMP Parallelism

Input: Matrix M of size $N\_SIZE \times N\_SIZE$
Output: Transposed matrix T
Initialize block_size as $N\_SIZE/4$
#pragma omp parallel for schedule(static) collapse(2)
for each i from 0 to $N\_SIZE - 1$ with step 2 **do**
  for each j from 0 to N_SIZE-1 with step block_size **do**
    for each k from 0 to block_size - 1 **do**
      $T[j+k][i] = M[i][j+k]$
      $T[j+k][i+1] = M[i+1][j+k]$
    end for
  end for
end for

Matrix Symmetry with Block-Based OpenMP Parallelism

Input: Matrix M of size $N\_SIZE \times N\_SIZE$
Output: Boolean value indicating if the matrix is symmetric
Initialize symmetric as `true`
#pragma omp parallel for schedule(static) reduction(&&: symmetric)
for each i from 0 to $N\_SIZE - 1$ **do**
  for each j from $i+1$ to $N\_SIZE - 1$ **do**
    symmetric = symmetric&&$(M[i][j] == M[j][i])$
  end for
end for

return symmetric

The most challenging aspect of choosing the right approach has been the overwhelming number of potential options, as well as the considerable time spent determining whether a solution is viable. The process involved not only evaluating each possibility carefully but also weighing the practical implications of each approach to ensure that it would effectively address the project's needs.

IV. Experiments and System Description

The experiments for this project were conducted on a high-performance computing (HPC) system, specifically on the node `hpc-c11-node23`. This system is built on a 64-bit x86_64 architecture and utilizes an Intel(R) Xeon(R) Gold 6252N CPU operating at a base frequency of 2.30GHz. The system is equipped with four sockets, each containing 24 physical cores, for a total of 96 CPU cores.

A. Experimental Setup and Procedures

For the purpose of these experiments, a maximum of 64 cores were utilized, with a total of 200 MB of memory allocated to the tasks. The code was compiled using g++ version 9.1.0. Each proposed solution is implemented in a single `.cpp` file, and depending on the compilation flag used, the Unit Under Test (UUT) is selected. Since where are dealing with variable size matrices, compilation flag are also used to specify the matrix length n. To test our functions, a job is submitted using a PBS file. This file handles the compilation of all tests and their execution. The results are then compared against a test function to verify correctness and are saved on convenient `.csv` files, to ease automatic data processing.

B. Experiment Design

The Experiment design for this project is quite straighforward. We begin by recording a time instance before invoking each function, once the function concludes its execution, a second time instance is recorded. The total run time is then determined by the different between the two timestamps. After collecting enough data, the times are averaged together and visualized through graphs to present the project results.

V. Results and Discussion

Due to limited space we will discuss only the most significant results, specifically those concerning a matrix size of 4096×4096 in the OpenMP section. The results presented will follow this structure:

- Serial vs Implicit
- Serial vs OpenMP

A. Serial vs Implicit

We first analyze the results obtained for the matrix transposition function As shown in Figure 1 for smaller matrix sizes, the implicit approach achieves the shortest execution time. However, as the matrix size increases, the difference in execution time between the serial and the two implicit versions becomes negligible. At size $n = 4096$ the faster approach is the implicitBlock being merely a little more faster than the other three.

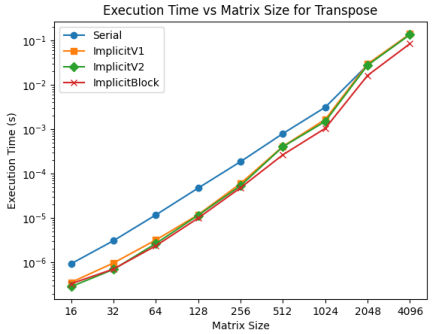

Fig. 1. Matrix Transposition

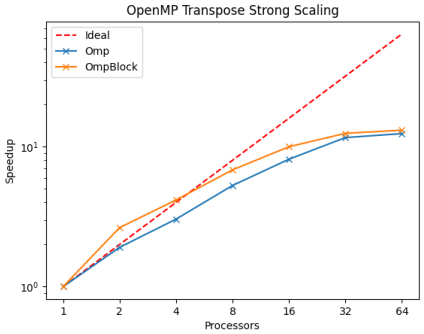

Fig. 3. Matrix Transposition Speedup OpenMP size $n = 4096$

Let's now analyze the results obtained for matrix symmetry. As shown in Figure 2, the results for the symmetry functions closely resemble those of the transposition function. In smaller matrix sizes the implicit approach dominates; however, as the matrix size increases, the performance margin becomes nearly negligible.

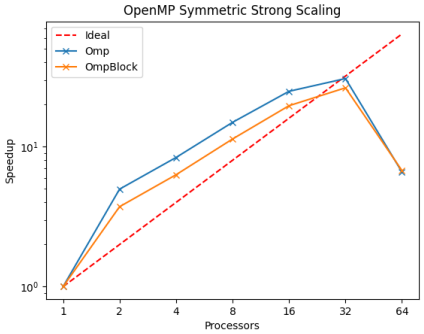

Fig. 4. Matrix Symmetry Speedup OpenMP size $n = 4096$

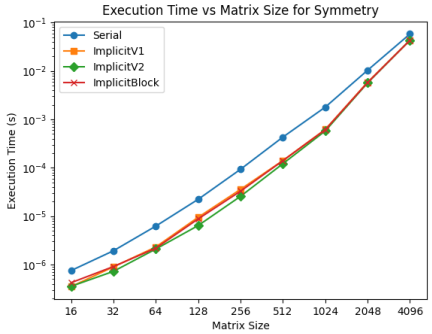

Fig. 2. Matrix Transposition

The speedup is therefor reflected in the execution time where a few points in the transpose function are lower than the ideal line, whereas in the symmetry graph most of the points are well under

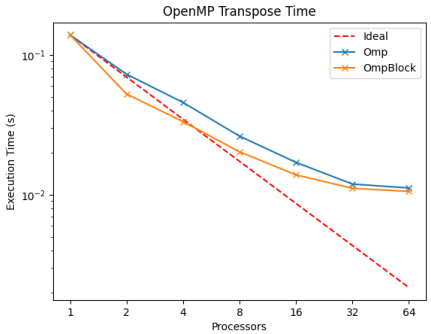

Fig. 5. Matrix Transposition Time OpenMP size $n = 4096$

We can now focus our attention on the OpenMP results. We can start by looking at the strong scaling of Figure 3 and of Figure 4. Figure 3 shows good strong scaling for the OpenMP block approach by having a couple of point where the function is over the ideal line. The biggest immprovement, however is on the symmetry strong scale where for most of the processors the function stays over the ideal line for both the approaches.

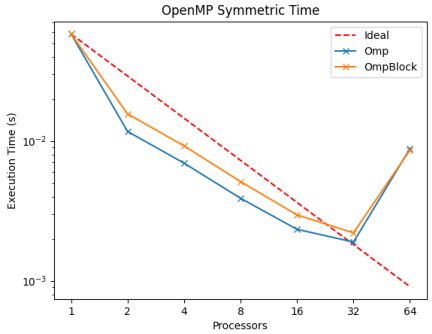

Fig. 6. Matrix Symmetry Time OpenMP size $n = 4096$

By looking at the efficiency graph we can observe a good curve for the transposition, however the symmetry graph seems a little too good to be true leaving me a little skeptical about the efficiency value obtained. One thing can be said for sure, by having too much active cores our Efficiency level plummets as the thread will start to starve themselves out.

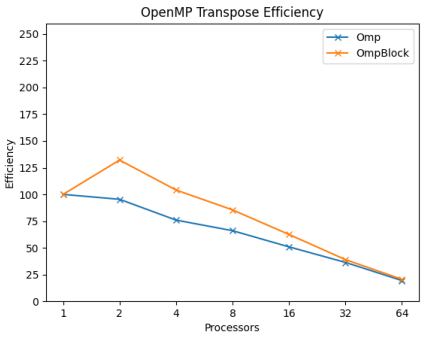

Fig. 7. Matrix Transposition Efficiency OpenMP size $n = 4096$

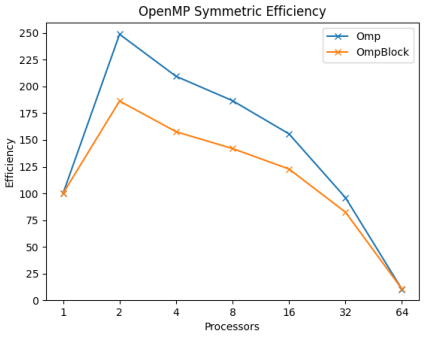

Fig. 8. Matrix Symmetry Efficiency OpenMP size $n = 4096$

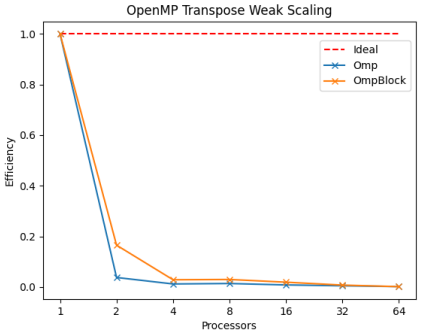

Fig. 9. Matrix Transposition Weak Scaling OpenMP size $n = 4096$

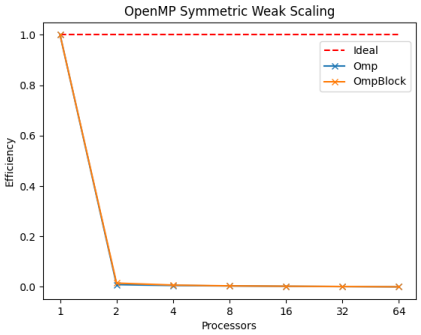

Fig. 10. Matrix Symmetry Weak Scaling OpenMP size $n = 4096$

VI. Conclusion

To conclude this project, I can say that while my goal is partially achieved, the data does not show promising weak scaling. Although the results may seem adequate for a project, they lack the potential to develop into a leading matrix transpose parallel algorithm. This outcome encourages me to explore new approaches to the problem. Additionally, this project has highlighted my current limited understanding of HPC and has fueled my desire to deepen my knowledge in the field.

References

References

[1] A. Grama, A. GUPTA, G. KARYPIS and V. KUMAR, Introduction to Parallel Computing, 2nd ed., Pearson Education, 2003.
[2] R. Robey and Y. Zamora, Parallel and High Performance Computing, Manning, 2021.
[3] Michael. J. Quinn, Parallel Programming in C with MPI and OpenMP, McGraw-Hill, 2004.
[4] Wen-mei W. Hwu, David B. Kirk and Izzat El Hajj , Programming Massively Parallel Processors A Hands-on Approach, 4th ed., 2023.
[5] F. Gebali, Algorithms and Parallel Computing, Wiley, 2011.
[6] Joseph JáJá, An Introduction to Parallel Algorithms, Addison-Wesley, 1992.
[7] F. Thomson Leighton, Introduction to Parallel Algorithms and architectures: Arrays·Trees·Hypercubes, Morgan Kaufmann, 1992.
[8] GCC Online Documentation, "Optimize Options - SIMD Cost Model"

[9] Khan, A., Al-Mouhamed, M., Fatayar, A., Almousa, A., Baqais, A., and Assayony, M., "Padding free bank conflict resolution for CUDA-based matrix transpose algorithm," in 15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2014, pp. 1-6, doi: 10.1109/SNPD.2014.6888709.

[10] Gorawski, M., and Lorek, M., "General in-situ matrix transposition algorithm for massively parallel environments," in 2014 International Conference on Data Science and Advanced Analytics (DSAA), 2014, pp. 379-384, doi: 10.1109/DSAA.2014.7058100.