Performance Comparison of Serial and Parallelized Matrix Transposition Using Different Parallelization Techniques

Cornel Grigor, id° 235986
Department of Information Engineering and Computer Science
University of Trento, Trento, Italy
cornel.grigor@studenti.unitn.it

***Abstract*—This report is about researching, benchmarking, and performance analysis of different methodologies used to implement the transposition of a squared matrix, a fundamental operation widely used in many computations. The goal was to optimize the algorithms that were tested by using implicit and explicit, OpenMP in this case, parallelization techniques. Numerous and exhaustive tests were conducted on each of the implementations used to gather data to compare the scalability and efficiency of the implicit and OpenMP-based versions across various matrix sizes, different optimization flags and thread configurations. Results demonstrate significant performance improvements over the serial version by using implicit optimizations that lead to a better overall performance while OpenMP showed its massive scalability potential when working with certain configurations.**

***Index Terms*—Matrix Transposition, Parallel Computing, Implicit parallelization, OpenMP, Performance Analysis**

I. Introduction of the problem and importance

Matrix Transposition is a linear algebra operation that rearranges elements of a two-dimensional array so that rows become columns and columns become rows. This operation is commonly used in order to allow numerical algorithms to access data contiguously, rather than with a stride [1]. This permits the exploitation of cache locality and memory access patterns. Some examples of the scientific applications where it gets utilized are: Graph Theory where the transpose of the adjacency matrix is used to compute the transpose graph [2]; Discrete Fast Fourier Transform (FFT) which is a highly efficient way to compute the Discrete Fourier Transform (DFT) [1]. To take full advantage of it the operation itself must be efficient. The effect of different parallelization techniques, implicit and explicit, were studied in order to make it faster and scalable on multi-core CPU architectures which allows for a division of the workload across multiple processing units.

II. State-of-the-Art

While both implicit optimizations and explicit techniques, such as OpenMP, are not new, they are still evolving. The field of Parallel Computing research is very active to this day and is even more relevant now due to significant development in multi-core processors, GPUs and distributed systems. These techniques are put in practice to speedup any kind of workload and the main goal is to utilize the resources that the system provides in the most efficient way within its limitations.

A. OpenMP

The latest OpenMP version 5.0 takes advantage of this by implementing full support for accelerator devices like GPUs that are more efficent at running massive parallelism and it includes mechanisms for unified shared memory and dynamic reverse offloading that allow computations to be offloaded from the CPU to an accelerator [3]. This also allowed porting existing CPU code to GPUs with some small adjustments to achieve the best results [9]. In a matrix transposition, as the matrix size increases, memory bandwidth becomes more impactful for performance, this is because a transposition requires, in the case of a in-place transposition, moving lots of data from a memory location to another and vice-versa. GPUs usually have higher memory bandwidth than what we can find in modern CPUs and this, combined with having a number of cores in the thousands, is why they are used as accelerators for heavy parallel tasks. When working with out-of-palce matrix transposition algorithms, which is when the result of the transposition is copied to another memory location, the consumption of memory is double and this becomes a problem when memory capacity is limited, such as in GPUs as opposed to CPUs [3] [4].

B. Implicit

On the other hand implicit optimizations do their best to take advantage of memory access patterns and hardware-level parallelism like Instruction-Level Parallelism (ILP) or Data-level parallelism (DLP) dynamically without requiring explicit changes by the programmer. This can be accomplished by using modern Compiler's optimization techniques. Some examples are: Automatic Vectorization, this functionality applies Single Instruction Multiple Data (SIMD) instructions to loops that can exploit DLP [8]. Other Implicit optimizations focus on improving cache locality and memory pre-fetching. [5]

C. Limitations

Currently both implementations have some limitations. Implicit optimizations can't be generalized for all architectures. For example every SIMD Instruction Set Architecture (ISA) requires its own programming approach developed and optimized [8]. The lack of fine tuning can result in missed optimization opportunities. OpenMP provides direct control

over parallelization, but load distribution, memory synchronization, schedule strategies and many more directives need to be manually optimized to find the right compromise for the best results and this increases the implementation complexity. Major overheads also need to be taken into account when scaling applications with OpenMP, the system will eventually reach some bottlenecks in the shared memory or in the thread management.

D. Identifying the gap

After some research there seems to be no practical studies and actual comparison between these three squared matrix transpose algorithms implemented in serial, implicit and explicit versions that take into account all of the aspects that are proposed in this report.

III. Contribution and Methodology

From an educational point of view this project works on providing real-world benchmarks comparing implicit optimizations and explicit OpenMP parallelization across different squared matrix transposition algorithms, configurations and test scenarios while providing some insights on the bottlenecks and implementation challenges that would arise in this scenario.

A. Methodology

Transpose algorithms for squared matrices are usually straightforward and between different implementation there are just some small changes on how the memory is accessed and how the actual data is being processed. For having a wider view of the subject I decided to do my testing on three different versions of the matrix transposition. Keep in mind that algorithms for non-squared matrices can get much more complex but this subject will not be treated during this report [4]. These are the algorithms that were tested as independent programs:

1) Naive out-of-place matrix transposition algorithm [M1]
2) Triangular in-place matrix transposition algorithm [M2]
3) Block-based matrix transposition algorithm [M3]

Each algorithm was analyzed in the same way to keep data consistent and comparable and there are no dependencies between the versions. All versions have their own serial, implicit, explicit methods of the *checkSym()* function and the *matTranspose()* function in this form:

- *checkSymSEQ()*
- *matTransposeSEQ()*
- *checkSymIMP()*
- *matTransposeIMP()*
- *checkSymOMP()*
- *matTransposeOMP()*

The *checkSym()* checks if the matrix that was generated is not symmetric, this is done because the transpose of a symmetric matrix is the original matrix it self, thus calculating the transpose of such a matrix using the transpose algorithms would be a waste of time.

All the versions implement the same algorithm for checking the symmetry in the 3 implementation methods that just vary in optimization mechanisms (SEQ,IMP,OMP).

Algorithm 1 Checks if Matrix is Symmetric

Ensure: **true** if the matrix is symmetric, otherwise **false**
  for $i = 0$ to n **do**
    for $j = 0$ to n **do**
      if $mat[i][j] \neq mat[j][i]$ **then**
        return **false**
  return **true**

All the test were performed on a squared matrix **M** represented by a 2D vector filled with float numbers with 3 decimals that were randomly generated, the goal was to keep the original matrix untouched and to generate a new **T** transposed matrix. The basic performance metric that was used during the whole testing was the execution time of just the *matTranspose()* ignoring the time taken by the *checkSym()* since that introduced another variable and made the data more inconsistent to compare between implementations. OpenMP's wall clock time function was used to take all the time measurements that get saved inside a CSV file for later processing and plotting of the data. To ensure the transpose correctness, every **T** transposed matrix is checked by using a function.

B. Sequential Implementation

Every algorithm has it's own serial implementation, that is the algorithm in the most simple and plain form that gets executed without any optimization or parallelism. Below are pseudo-codes of the three algorithms tested:

1) Naive out-of-place [M1]: The most basic and straightforward approach. The original matrix and the destination for its transposed are passed by reference.

Algorithm 2 Naive (Out-of-Place) transposition

  for $i = 0$ to n **do**
    for $j = 0$ to n **do**
      $trans[j][i] \leftarrow mat[i][j]$

2) Triangular in-place [M2]: This was implemented as in-place which means that the transpose is done on the same memory space without a second matrix where the result gets copied to. To avoid over-writing the original matrix the function is called by passing by reference a copy of the matrix. This is called triangular as it only checks the upper triangle of the matrix to avoid redundant operations.

Algorithm 3 Triangular (In-Place) Transposition

  for $i = 0$ to n **do**
    for $j = i + 1$ to n **do**
      Swap $mat[i][j]$ and $mat[j][i]$
  return `mat`

3) Block-based out-of-place [M3]: This approach divides the matrix into blocks of the same size before being actually transposed, the block size choice impacts performance, see more on this later.

Algorithm 4 Block-Based Matrix Transposition (Out-of-Place)

```
for i = 0 to n step blockSize do
  for j = 0 to n step blockSize do
    for bi = i to min(i + blockSize, n) do
      for bj = j to min(j + blockSize, n) do
        trans[bj][bi] ← mat[bi][bj]
```

C. Implicit Parallelization

The implicit methods were applied directly to the serial version of algorithms for each version. It can be argued that the block-based version is an implicit optimization itself since it is supposed to optimize memory accesses [7] but for consistency reasons it was studied the same way by comparing an identical serial code with no extra compilation flags to one that was optimized using gcc's appropriate flags.

D. Explicit Parallelization with OpenMP

OpenMP was used to parallelize the serial methods in the "OMP" versions of the algorithms by using the Pragma directives. This was done by exploiting the nested for loops and reduction of the boolean variable in the case of the *checkSymOMP()* function. This exception was needed to avoid race conditions since a boolean variable is used in the OMP implementation of this function and in this way every thread updates its own private variable until the end of the loop where a logical AND is applied to all the private variables to get the final result.

In the Naive and Block-Based approach it was possible to utilize `collapse(2)` as the inner for loops had no dependencies on the outer one but this was not the case in the Triangular approach where only the outer loop was parallelized. Race conditions are managed by the for clause and there's no need to specify shared or private variables as iterators inside the parallel region are private by default and the rest of the parameters needed, the matrix and the matrix size, are shared and managed by the for directive. The goal is to avoid race conditions at all costs.

IV. Experiments and System Description

A. System Specifications

The testing was done on the HPC cluster that UniTN provided for this experience, specifically by running an Interactive session on the `short_cpuQ` queue on the nodes that rely on 4x `Intel Xeon Gold 6252N` (96 cores) processors from the "Cascade Lake" family of Intel's processors running the x86-64 architecture. Each processor has 35.75MB of L3 cache so in total 143MB of L3 cache then each core has 1MB of L2 and 32KB of L1 cache. The cluster runs on a linux environment and the compiler utilized was `gcc 9.1.0`. Stack memory limit is set to unlimited in the cluster's nodes.

B. Experimental Setup

The tests were conducted on matrices with size ranging from 2 to 4096 with power of two increments. Every execution time for every test scenario was taken 5 times so that a mean time could be elaborated inside the plotting program, written in python, for more uniform results. Since the cluster's performance was not so consistent, times that were considered outliers were removed manually during the making of the graphs. For the OpenMP implementations data was acquired for multiple number of thread configurations ranging from 2 and increasing in powers of two up to 96. To avoid unpredictable behaviour the maximum number of threads utilized was purposefully chosen to be the same as the maximum number of physical cores available. Note that 96 core's performance was very inconsistent on the cluster.

After some research the flags that were tested for the implicit implementations are:

- `-O1`
- `-O1 -march=native`
- `-O2`
- `-O2 -march=native -funroll-loops`

I used O1 and O2 optimizations since these are one of the most common optimizations you can apply. When used in reality they enable a collection of multiple optimization flags [11]. For example O2 enables all O1 optimizations(like `-fprefetch-loop-array`) plus other like `-ftree-loop-vectorize` that performs vectorization on trees [11]. The flag `-funroll-loops` forces the compiler to unroll loops, this reduces loops overhead at the cost of code size [12]. The last flag tested, `-march=native` optimizes the code for the specific cpu family , `native` detects the cpu family and applies the right optimization ("cascadelake" in this case) [10].

C. Hypotesis

The implicit version should be significantly faster but since the code we are running is not very complex there might be little difference between the flags that were tested.

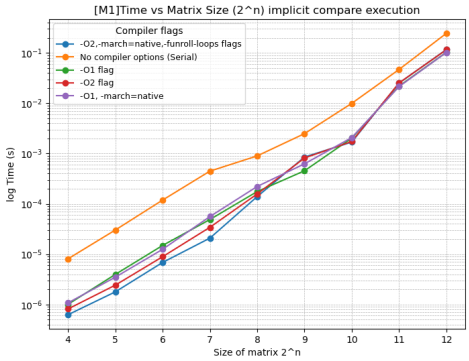
For OpenMP the expected result varies according to the matrix size tested, for smaller matrices the single thread version can be faster since it avoids additional overhead caused by thread management, the bigger the matrix gets the more speed up is expected when running parallelism.

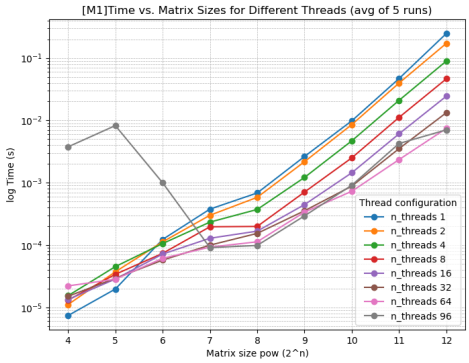
Block-size is also crucial, even more so when running the OpenMP version to evenly distribute the workload, since the goal is to minimize cache misses by choosing the right size that will fit inside the CPU cache to avoid as much as possible accesses to main's memory (major overhead) and to exploit cache locality [7]. For the best performance block-size should be adjusted for each configuration to get the best performance. The cache sizes of the processor plays an important role when choosing the block size, if we had a bigger cache L1 or L2 then bigger block sizes would be more efficient and would reduce the overhead of switching from one block to another. We can solve this issue by using bigger block sizes that would fit into L3 cache at the compromise of a lower memory bandwidth.

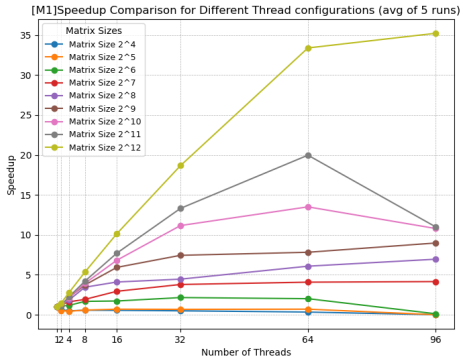
V. Results and Discussion

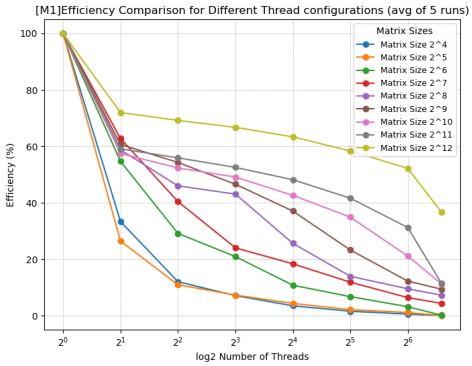
A. Performance Graphs

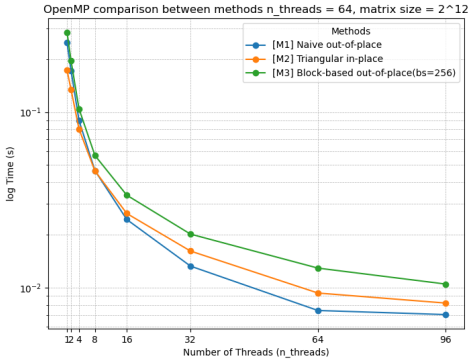
The graphs of the [M1] are shown but the trends apply with small differences to all of them.











B. Analysis and Comparison

We see small differences between the optimization flags that become almost none as the matrix size increases, with the best overall result being `-O2 -march=native -funroll-loops`.

As expected for OpenMP we see an improvement in time when using matrices bigger than $2^6 * 2^6$. The bigger the matrix, the higher efficiency and speedup was obtained, this is because the scalability outweighed the thread overhead with bigger workloads.

For the Block-based, after a performance comparison, a block size of 256 was chosen, this was done to have a good enough compromise that would work for smaller and bigger matrices. Bigger block sizes were tested but didn't improve performance in a noticeable way.

Cache misses were checked using the Linux `perf` tool, it showed how the % of misses increases as the size of the matrix grows. Roughly the same % of cache misses were observed for all three methods going from as low as 20% to 55/60% for bigger matrices, this is probably due to memory bottlenecks as transposition is a memory-bound application [13] [14].

Thanks to the comparison we can see that overall [M1] was the one that scaled better and this could be thanks to the parallelization of the two nested for loops.

VI. Conclusions

In conclusion the tests conducted gave an understanding of the system response to different optimizations and configurations. We saw that parallelism is not always faster for simple tasks with small workloads and that optimizations need fine-tuning for the best results [7] [8]. Also we saw that throwing as many cores as you have on a task doesn't always scale up the way you think and added overheads destroy the efficiency expected from the parallelized code. Specific sized workloads need an appropriate amount of resources to make the most out of them.

Git and Instructions for Reproducibility

In the Git repository below you can find the code tested with any details about reproducibility including other graphs about the other methods performance:

https://github.com/cornelGrg/IPCdeliverable1

REFERENCES

[1] https://www.sciencedirect.com/topics/computer-science/matrix-transposition

[2] https://en.wikipedia.org/wiki/Transpose_graph

[3] https://www.openmp.org/press-release/openmp-5-0-is-a-major-leap-forward/

[4] J. Gómez-Luna, I. -J. Sung, L. -W. Chang, J. M. González-Linares, N. Guil and W. -M. W. Hwu, "In-Place Matrix Transposition on GPUs," in IEEE Transactions on Parallel and Distributed Systems, vol. 27, no. 3, pp. 776-788, 2016.

[5] M. O. Karsavuran, K. Akbudak and C. Aykanat, "Locality-Aware Parallel Sparse Matrix-Vector and Matrix-Transpose-Vector Multiplication on Many-Core Processors," in IEEE Transactions on Parallel and Distributed Systems, vol. 27, no. 6, pp. 1713-1726, 1 June 2016

[6] P. D. Michailidis and K. G. Margaritis, "Computational Comparison of Some Multi-core Programming Tools for Basic Matrix Computations," in *2012 IEEE 14th International Conference on High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems*, Liverpool, UK, 2012, pp. 143-150.

[7] S. Chatterjee and S. Sen, "Cache-efficient matrix transposition," Proceedings Sixth International Symposium on High-Performance Computer Architecture. HPCA-6 (Cat. No.PR00550), Touluse, France, 2000, pp. 195-205

[8] H. Amiri, A. Shahbahrami, A. Pohl, and B. Juurlink, "Performance evaluation of implicit and explicit SIMDization," *Microprocessors and Microsystems*, vol. 63, pp. 158-168, Nov. 2018.

[9] J. Huber et al., "Efficient Execution of OpenMP on GPUs," 2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), Seoul, Korea, Republic of, 2022, pp. 41-52.

[10] https://gcc.gnu.org/onlinedocs/gcc/x86-Options.html#index-march-15

[11] https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

[12] https://en.wikipedia.org/wiki/Loop_unrolling

[13] V. Rajput and A. Katiyar, "Proactive Bottleneck Performance Analysis in Parallel Computing Using OpenMP," International Journal of Advanced Studies in Computer Science & Engineering (IJASCSE), Vol. 2(5), 2013, pp. 46-53.

[14] https://github-pages.ucl.ac.uk/research-computing-with-cpp/07performance/sec02Memory.html#:~:text=A%20straight%2Dforward%20example%20of,and%20place%20them%20in%20another.