Parallel Matrix Transposition Optimization

Alessandro Benassi
236027
alessandro.benassi@studenti.unitn.it

***Abstract*—This study explores the implementation and performance analysis of matrix transposition using sequential, implicit parallelism, and explicit OpenMP parallelization techniques. A fixed random $n \times n$ floating-point matrix is transposed using three methods, followed by performance benchmarking and analysis. Results include speedup, efficiency, and bottleneck discussions, highlighting the trade-offs between the different approaches.**

I. Introduction of the problem and importance

Matrix transposition is a fundamental operation applied in various fields like computer science, mathematics, robotics, signal elaboration and many more. It forms the basis of many algorithms and it's essential to manipulate data structures when talking about memory management. Because of this various applications it is crucial that the algorithm which performs the transposition is very efficient and doesn't produce bottlenecks or errors, because this will affect in particular programs or applications using very large matrices. One of the main challenges is improving how data is moved between the memory and the processor, and so minimizing the memory accesses and reducing the cache miss ratio. This project investigates three implementation strategies: sequential, implicit parallelism through vectorization and memory optimizations, and explicit parallelism using OpenMP. The aim is to evaluate their performance and scalability across varying matrix sizes also trying to fill the gap about the memory access problem. Additionally, performance is analyzed using metrics such as execution time, speedup, and efficiency.

II. State of the art

A solution found on a forum for the OpenMP version was analyzed [1] and it was taken as a starting base for the block method with OpenMP. It was a solid function that parallelized the loops with a collapse instruction but it had a problem with the inner loops which doesn't verify if the parameter exceeds the size of the matrix. The proposed solution didn't also take in account the possible difference between static and dynamic scheduling using OpenMP.

III. Methodology

A. Main flow

- The program starts by checking whether the correct number of parameters has been passed and that the size of the matrix is a power of 2 and if the number of tries is greater than one. Then the random seed is initialized and the first file containing the times of each execution is created in append mode and in the sequential program the header is appended. The central session of the program consists in a for loop that iterates for the number of time given in input as tries and it cleans the cache, allocates the matrices, do the transpose by also checking the time before and after transposing, it checks if the transposition happened correctly for low dimension matrices and then it populates an array with the time taken and prints the data on the previously opened file. At the end it frees the memory allocated for the matrices and starts again. After that the array containing the times is sorted in ascending order with a bubble sort function and only 40% of the interval is taken to compute the average so we can be sure to have a more correct value. Another file is then opened in append and write mode and the data with the average are written there. For the implicit and OpenMP program there is another part that fetches from this file the average of the time for the used size of the matrix and uses it to compute the speedup and the efficiency. This last two parameters are computed like that:

$$\text{Speedup} = \frac{\text{Sequential Time}}{\text{Parallel Time}}$$

$$\text{Efficiency} = \frac{\text{Speedup}}{\text{Number of Threads}}$$

B. Sequential Implementation

- The `checkSym` function verifies if the matrix is symmetric by computing two nested for loops which analyze only the superior part of the matrix, except the central line. The input parameters for this function are a pointer to a pointer to a float, which passes the matrix and an integer that represents the size of this last one. The condition of the first for, which scrolls the rows, is set to

  `i<n`

  where n is the size of the matrix; and the condition of the second one, which is for the columns, is set to

  `j<i`

  This nested for loop implementation will be used multiple times across this program with some changes. Inside of the two loops there is an if construct that returns zero if the element selected is different from its symmetric:

  ```
  if (M[i][j] != M[j][i]) {return 0;}
  ```

  Otherwise if the matrix is symmetric the function returns 1.
- The function `matTranspose` transposes the matrix. It takes two pointers to a pointer to a float and an integer as input; these are respectively the matrix to transpose, the matrix where it will be transposed and the dimension

of the square matrices. The first thing it does is to check whether the matrix is symmetric or not by using the previously implemented `checksym` function and if this is equal to zero we can proceed with the transpose operation. This process is based on scrolling sequentially all the elements of the matrix with two nested for loops like the ones explained before and assigning them to the other matrix with the indexes swapped to exchange the positions.

```
T[i][j] = M[j][i];
```

This time the second for loop condition extends to j¡n because we need to scroll all the elements.

C. Implicit Parallelization

- The `checkSymImp` function for this implementation has been done by dividing the matrix in blocks of size 8 and checking element by element if they are equal and if so returning zero, otherwise if the matrix is symmetric it returns one. The input parameters are the same as the `checkSym` of the previously described sequential part. The structure is composed of two outer for loops that scrolls the rows and the columns of the matrix by a factor of 8 creating different blocks and two other inner for loops that scroll the elements of the block and operate the control on it. This last check is equal to the one explained for the previous `checkSym` function. The factor 8 was chosen to be coherent to the number used in the unrolling for the `matTransposeImp` function, explained int the next paragraph.
- For the function `matTransposeImp` the main focus was to unroll the loops. The structure of the function and the input parameters are the same as the previously explained `matTranspose`. There are two nested for loops, one for the rows and one for the columns. The difference is that the j inside the second loop increments of a factor n since we want to process more elements in once, without redoing the check of the for loop condition n times. The number of the unrolls was chosen doing a tradeoff: since the aim was to work with powers of two 8 was considered the best number compared to 4 and 16, because the first was a bit small to give effective changes over big matrix dimensions and the second would equal to the sequential implementation for a dimension equal to 16. Inside the loop we transpose n elements. This was done by writing:

```
T[i][j] = M[j][i];
T[i][j + 1] = M[j + 1][i];
...
T[i][j+n] = M[j+n][i];
```

after that there is another for loop to execute also the remainers, because if the unroll is done by a factor n=4 and the matrix has for example 22 as dimension, two columns will remain out because the condition

```
j<=matrixdim-8
```

is not satisfied. This discussion about the possible remainers won't be repeated since the implementation was done paying attention to respect some size constraints decided and explained in the system description part.

D. Explicit Parallelization with OpenMP

- For the `checkSymOMP` the input parameter is the same as the `chechSym`. An integer tmp variable is set equal to one. Then two for loops are executed and inside of these there is a condition that checks whether an element of the matrix it's different from its symmetric and sets tmp to zero if this is true. All of this is parallelized by putting the pragma omp parallel instruction with

  ```
  for reduction(&&:tmp)
  ```

  before the first for loop, so the instructions are executed on a local variable for each thread and at the end of execution they are put together using the & operator in this case. (code displayed like that due to size issues)
- The `matTransposeOMP` function has been implemented in two different ways. The first one is with the worksharing method and the second one with the blocked method. The first method has the sequential function as base. The initial intention was to use the instruction

  ```
  #pragma omp for collapse(2)
  ```

  to parallelize both loops, but then the final choice was to use a

  ```
  #pragma omp parallel for
  ```

  to parallelize the outer loop and to use the

  ```
  __builtin_prefetch(&M[j + 1][i], 0, 1)
  ```

  function to prefetch data for read and write(second parameter) and optimize cache usage. For the blocksize implementation an approach similar to one found on an online forum was taken [1] as a starting point, see the discussion on its performance and issues in the previous section. The two outer for loops were handled using the collapse(2) clause and inside two variables check and check2 were declared using a newly created macro. This macro determines which one of two numbers is the smaller and gives it as output. This was used to determine the condition of the two inner for loops. This ones had to satisfy the conditions: ii less than n and ii less than i+blocksize where ii is the parameter of the inner loops, i is the one of the outer loops, which changes in j if we consider jj as the inner parameter. This is done to scroll the block so that each thread will have its set of blocks. By using the macro min we avoid doing a double control, which will result in having four of them because we have two loops and we still guarantee the correctness of the solution because we take the smaller between the out of bound condition of the block and the matrix size.

  ```
  #define min(a, b)((a)<(b)?(a):(b))
  ```

  For this implementation the decision was to set the scheduler to static, due to the fact that for an equal workload of the threads it works better, like it's written in this

document [2]. So since in both of the implementations proposed before the workload of the threads is equal, in the pbs file the command

```
export OMP_SCHEDULE=STATIC
```

was called to be sure that the scheduler was set to static.

- One of the most crucial challenges faced during the process was the cache freeing before every execution. Since the implementation chosen for executing multiple time the transpose function was to loop inside the program itself and the matrix was fixed due to the fixed seed there were problems with the cache. Part of the matrix, or also all for small sizes, was stored in the cache and so when another computation of the matrix was done the data were already in the cache and didn't need to be fetched. This led to an improvement of the performance which gave problems in testing new things. The function

  ```
  cache_flush()
  ```

  was added, which allocates an array of char of a fixed size, determined by doing the sum of the dimension of the L1 and L2 caches of the cluster in byte. The elements of the allocated array are then populated with ones and the array is then freed. Another important challenge was the one to retrieve the time of the sequential execution to compute the speedup and the efficiency. This was done by implementing a two file logic, where the first file contains all the type of the execution, the dimension of the matrix, the number of threads and the time taken for transposing. In the second files there are the same parameters plus the speedup and the efficiency computed with the averages of the executions done by taking only the 40% of the executions to avoid outliers. To retrieve the data of the sequential a while loop containing the fgets function was done. So the lines were retrieved until the end of the file. Inside this loop strtok was used to divide the elements of the line and analyze them singularly to select the correct one, which was saved in a local variable.

IV. Experimental Setup

Experiments are conducted on a system with the following specifications:

- Architecture: x86 64
- CPU op-mode(s): 32-bit, 64-bit
- Byte Order: Little Endian
- CPU(s): 96
- On-line CPU(s) list: 0-95
- Thread(s) per core: 1
- Core(s) per socket: 24
- Socket(s): 4
- NUMA node(s): 4
- Vendor ID: GenuineIntel
- CPU family: 6
- Model: 85
- Model name: Intel(R) Xeon(R) Gold 6252N CPU @ 2.30GHz
- Stepping: 7
- CPU MHz: 2300.000
- BogoMIPS: 4600.00
- Virtualization: VT-x
- L1d cache: 32K
- L1i cache: 32K
- L2 cache: 1024K
- L3 cache: 36608K
- NUMA node0 CPU(s): 0-23
- NUMA node1 CPU(s): 24-47
- NUMA node2 CPU(s): 48-71
- NUMA node3 CPU(s): 72-95

The libraries used are the stdio.h, stdlib.h which are of standard use, then there is the omp.h in all the different implementations because the time calculation is done via

```
omp_get_wtime()
```

There is also the library string.h used in the files for the strtok function to extract the average time of the sequential program and compute the average and the speedup. The gcc91 version 9.1.0 toolchain was used to compile the files. This module was loaded following the instruction written in the readme file. The matrix sizes where powers of two from 16 to 4096 and the threads were also powers of two from 2 to 64. This is important to specify as some functions have a specific design that has it's peak performance only by using this parameters above. The tests were done separately for the implicit and OpenMP implementation to facilitate the work. Then the pbs files were joined together after the tests to determine the final implementations. For the implicit implementation tests were made between some different flags: -O2, -O2 -march=native, -O2 -ftree-vectorize -march=native,-O2 -funroll-loops -march=native.The aim was to find the best flag combinations that enhance the performance. For the OpenMP implementation two different approaches were taken: the worksharing and the block method. For the second one tests were made for different sizes of the block: 8,16,32,64. All the data was gathered in two csv files: the first one is to see all the data of the simulations that have been done, the second one displays the average time of the simulations with the speedup and the efficiency related to each pair of dimension of the matrix and thread number. The files are easily manipulable because by converting the csv file in xlsx format with an online converter and opening it with excel many operations can be done on the data. Especially for the second one which contains more interesting, compact and useful data for the purpose of this project. If we open it with Excel we can select all the data and form a table. From this table we can generate different pivot tables. The ones that are more interesting are the table with the type and the number of threads on the columns and the dimension on the rows, with the speedup as value displayed and the same but with the efficiency displayed.

V. Results and Discussion

All the most important graphs and tables with the data can be found in the result folder on github. Regarding the implicit

implementation the flag combination -O2 -funroll-loops -march=native flags was chosen because compared to the other combinations it has a better speedup in almost all the points of the curve compared to the others. The results are coherent because since the implicit program was developed only using the unroll technique and doesn't involve multithreading, for bigger dimensions the speedup will inevitably decrease and the efficiency will go down with it since the values are the same because the number of threads is one. For the OpenMP part, the block based implementation has a better overall performance compared to the worksharing because in general with all the different blocksizes(8,16,32,64), in particular when talking about bigger matrix dimensions the speedup is a bit higher. On smaller dimensions the speedup is below one, this is probably due to the fact that setting up the environment takes time and it's not so efficient for small dimensions. When we have many threads we also have the problem that the threads are more than the one necessary to execute the program and so there are probably problems in dividing the workload. Since the best block was the 64, a new problem arised: the smaller matrices were executed like a sequential program since they weren't divided in blocks due to dimension issues. To avoid that a switch was inserted in the main file, which when the matrix has a size equal or smaller than 64 it sets the blocksize to half of the dimension so we have parallelization. Comparing the two ways chosen for OpenMP the block method is overall faster and has higher numbers especially for bigger dimensions and high number of threads, that is what we were searching for.

VI. Conclusion

This project highlights the strengths and trade-offs of sequential, implicit, and explicit parallelization strategies for matrix transposition. Sequential implementation is straightforward but lacks scalability. Implicit parallelization offers moderate performance improvements with minimal user effort, while explicit parallelization using OpenMP delivers the best performance for larger matrices and high thread counts, though it requires careful tuning. The initial idea to improve the cache management revealed to be good for the implicit with the memory prefetching, but for the OpenMP the block method appears to be the best related to the context and the methodology used. The results show that block-based and also worksharing OpenMP, are most effective for large-scale tasks, while implicit methods work well for smaller setups. These findings emphasize the importance of choosing the right approach based on the task and hardware to achieve optimal performance. This study provides useful benchmarks and insights to guide future work in parallel computing for similar tasks.

References

[1] Stack Overflow User. "Optimizing a Matrix Transpose Function with OpenMP." *Stack Overflow*, Jan. 23, 2022. [Online]. Available: https://stackoverflow.com/questions/70811082/optimizing-a-matrix-transpose-function-with-openmp

[2] B. Sobral. "OpenMP Dynamic Scheduling." [Online]. Available: http://www.inf.ufsc.br/~bosco.sobral/ensino/ine5645/OpenMP_Dynamic_Scheduling.pdf

[3] GeeksforGeeks. "C Program to Find Whether a No is Power of Two" [Online]. Available: https://www.geeksforgeeks.org/c-program-to-find-whether-a-no-is-power-of-two/

[4] Stack Overflow User. "Bash Shell Nested For Loop." *Stack Overflow*, Dec. 11, 2010. [Online]. Available: https://stackoverflow.com/questions/4847854/bash-shell-nested-for-loop

[5] PhoenixNAP. "Bash Check if File or Directory Exists." [Online]. Available: https://phoenixnap.com/kb/bash-check-if-file-directory-exists#:~:text=Check%20if%20a%20Directory%20Exists%20in%20Bash,-To%20check%20if&text=echo%20%24%3F,-The%20output%20is&text=d%20%2Ftmp%2Ftest-,echo%20%24%3F,test%20command%20works%20here%20too

[6] GeeksforGeeks. "fgets() vs gets() in C Language." [Online]. Available: https://www.geeksforgeeks.org/fgets-gets-c-language/

[7] Stack Overflow User. "Know Position (Person) in File with ftell and fseek in C." *Stack Overflow*, June 7, 2021. [Online]. Available: https://stackoverflow.com/questions/67862000/know-position-person-in-file-with-ftell-and-fseek-in-c.

[8] Stack Overflow User. "Min and Max in C." *Stack Overflow*, Aug. 10, 2010. [Online]. Available: https://stackoverflow.com/questions/3437404/min-and-max-in-c

[9] GeeksforGeeks. "size-t Data Type in C Language." [Online]. Available: https://www.geeksforgeeks.org/size_t-data-type-c-language/

[10] Stack Overflow User. "ftree-vectorize Option in GNU." *Stack Overflow*, Nov. 7, 2015. [Online]. Available: https://stackoverflow.com/questions/33570114/ftree-vectorize-option-in-gnu.

Appendix

A. Code Repository and Reproducibility

The code and instructions for reproducing the results are available at: `[https://github.com/ale-bena/Matrix-Transposition-Optimization.git]`