Introduction to parallel computing, homework 2

Anne-Marthe Hvammen Sikkerboel
245997
anne.sikkerboel@studenti.unitn.it
link to git repo: https://github.com/AnneMartheH/IPC_HOMEWORK_2.git

22. november 2023

Parallel matrix multiplication

In this task the goal was to implement a program that multiplied two matrices. In this program I have one function called matMul which multiplies the matrices sequentially. Then a function called matMulPar which multiplies the matrices in parallel, by using the directive OpenMp.

Solution implemented matMul
The function Matmul takes in the matrices A and B. In order for matrix A and B to be compatible I always initialize the matrices as square matrices with the same dimensions. When the two multiplied matrices are square matrices with the same dimension, the result matrix C will also have the same dimension. Because of this decision I can make a global variable MatrixSize. MatrixSize decides the matrix size of every matrix in this script.

At the beginning of the matMul function the memory space for matrix C is allocated and initialized. This is done so the function can handle big matrices. Afterward the matrix multiplication begins. The multiplication itself consists of three nested for loops, with a sum inside the inner for loop. In this sum the multiplied value for each element is stored, before it in the second loop is stored at the right place in matrix C.

Solution implemented matMulPar
As I understood the assignment, function matMulPar should be a copy of the function matMul, but parallelized using OpenMP. Hence, the code implements the same strategics for matrix multiplication, but with a few additional lines for the parallelizeation.

As you can see in the script task1TimeTests.c I tried different implementations of OpenMP, this is also represented in table 1. Since I never have coded with OpenMP I tried different directives and clauses to see what gave improved time. Since the code consists of several for loops I thought the parallel for"and the fordirective would be reasonable to use. However I also tried with the "atomicto see if this improved the running

time.

I ended up using the matMulParV5 from the task1TimeTest.c script as this gave the best average runtime.

Performance and results

Function name	Average time	Directive	Clauses
matMulParV1	0,4319	Parallel for	private(k), collapse(2)
matMulParV2	1,0111	Parallel for	private(k)
matMulParV3	1,0188	Parallel for	firstprivate(k)
matMulParV4	0,8259	Parallel for	private(k), schedule(dynamic)
matMulParV5	0,2897	Parallel for	private(k), schedule(dynamic), ordered
matMulParV6	0,2985	Parallel, for	private(k), schedule(dynamic), ordered
matMulParV7	0,5849	Parallel, for, atomic	private(k), schedule(dynamic), ordered

Tabell 1: Time tests on the matMulPar function from file task1TimeTests.c with different OpenMP clauses and directives. The average time is calculated from 4 runs with 8 threads in the cluster.

number of threads	time[s], matrixSize = 1000	time[s], matrixSize = 3000
1	3,78	186,6
2	3,83	153,9
4	3,77	154,9
8	3,74	152,9
16	3,64	168,4
32	3,70	154,5
64	3,77	153,2

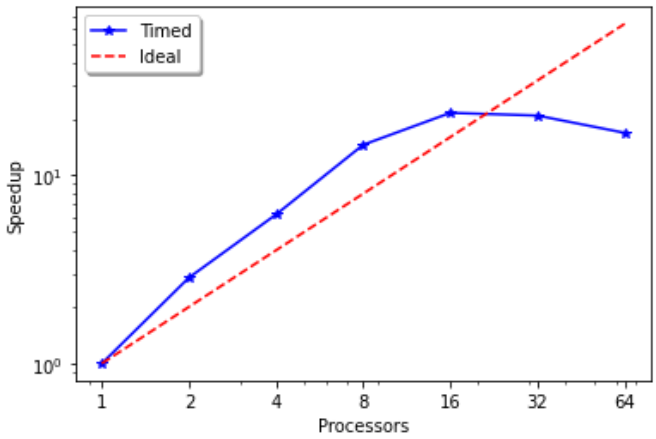
Tabell 2: Time used by function matMul from file task1_AntattBest.c on one run in the cluster

Performance discussion and analyzing
In this task the plots are a representation of the strong scalability of the function with a specific matrixSize value.

By looking at the table 1 two functions (V2 and V3) stand out as worse than the average and two stand out as better than the average (V5 and V6). The two worst functions both use the directive parallel for"with only one clause each. The two best functions both use three different clauses, but they use different directives. By forcing the threads to follow the sequential iteration order of the code and specifying how the threads should spread over the iteration of a loop we save more time, than when this is decided freely. OpenMP is limited in how much the programmer can decide over the parallelization. With MPI you can decide even more specific how the threads should spread and in which order it

number of threads	time[s], matrixSize = 1000	time[s], matrixSize = 3000
1	3,836	156,4
2	1,341	69,96
4	0,6186	33,79
8	0,2635	18,74
16	0,1782	7,217
32	0,1846	3,647
64	0,2276	4,583

Tabell 3: Time used by function matMulPar from file task1_AntattBest.c on one run in the cluster



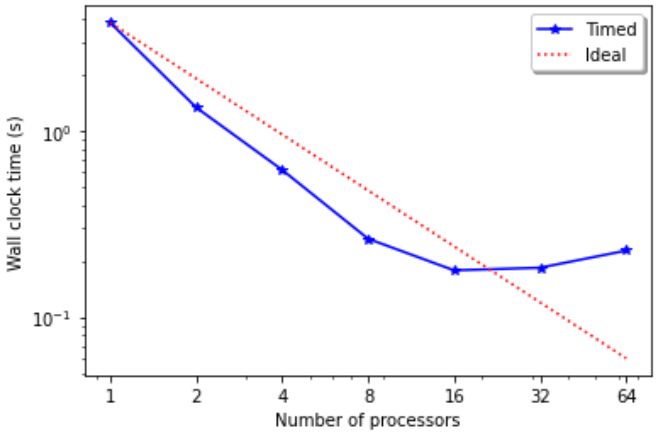
Figur 1: Speedup in matMulPar function in file task1_AntattBest.c with matrixSize = 1000 compared to ideal speedup, plotted by using professor Rio Martins script

would be possible to optimize the function.

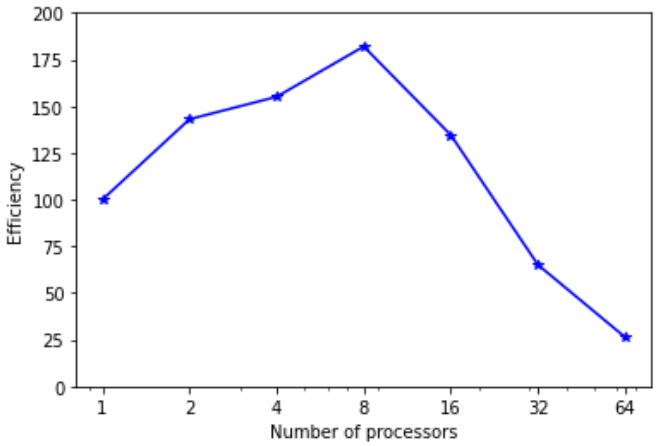
Looking at the plots for function matMulPar with matrixSize at 1000 (2 1) the timed functions is better then the ideal. Looking at the 3 one can also see that the efficiency for 8 threads is greater than 175%. From what I have understood a code is almost never better than the ideal code, and the percentage could never by higher than 100%. So these results come off as weird. By looking at the plots for matrixSize = 3000 (6, 4, 5) one can see the same problems but in smaller scale.

However looking at the table 2 and 3 there is a clear difference between the running time in the parallel and in the sequential code for both matrixSize = 1000 and matrixSize = 3000. The running time in the parallel has a clear improvement.

Since the plots and results are based on running the code one time, and not the average of running several times, this makes the results less robust than if it was calculated as



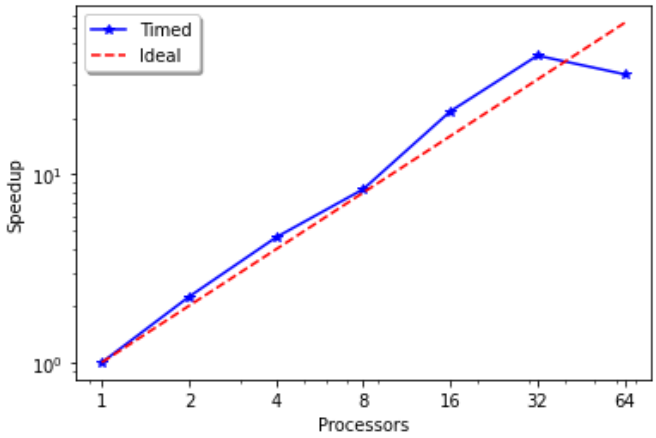
Figur 2: wall clock of timed matMulPar function in file task1_AntattBest.c with matrixSize = 1000 compared to ideal speedup, plotted by using professor Rio Martins script



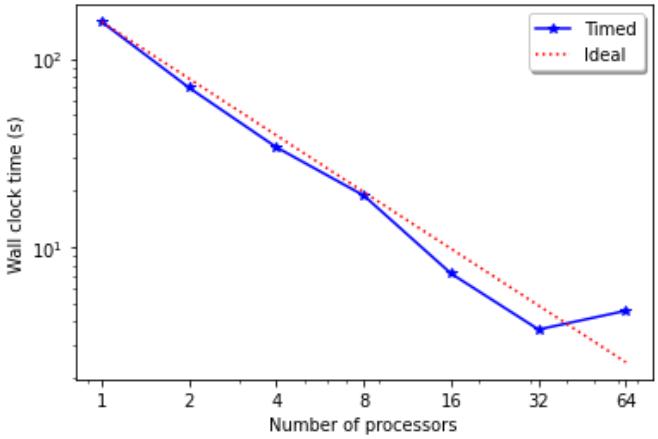
Figur 3: Efficiency of timed matMulPar function in file task1_AntattBest.c with matrixSize = 1000, plotted by using professor Rio Martins script

an average. Because of this the results my vary if the code is ran again. However, the trends in the code should be the same. The trend in this code is that the codes run-time improve a lot with parallelization.

Compering the plot for wall clock time for matrixSizez at 500 7, with the one at 1000 2 and 3000 5, one see that the bigger matrices follows the ideal line better. This is also possible to see in the trend of the timed graph. The wall clock time of the graph for the matrixSize = 3000 diverges from the ideal with 64 processors. But the wall clock time with matrixSize = 500, has a spike when running with 2 processors. This shows that larger problems are handled more efficiently, and with a better trend. Hence this can imply that program also responds well to weak scaling.



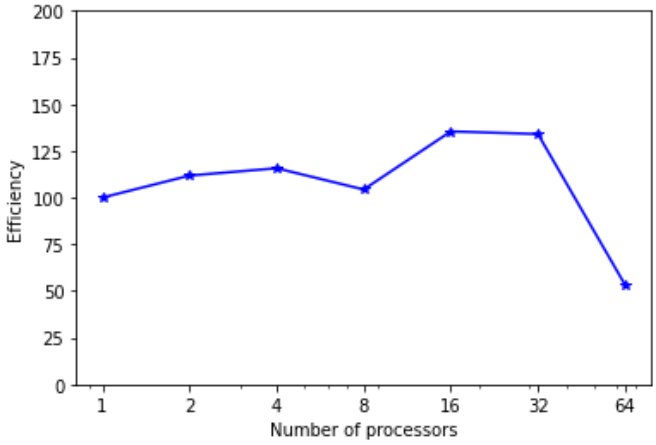
Figur 4: Speedup in matMulPar function in file task1_AntattBest.c with matrixSize = 3000 compared to ideal speedup, plotted by using professor Rio Martins script



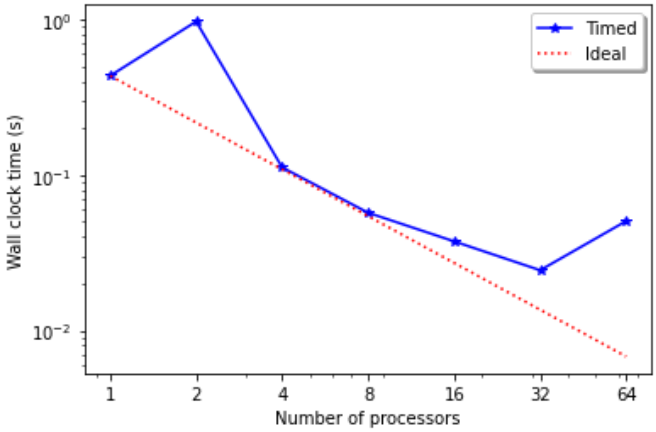
Figur 5: Wall clock of timed matMulPar function in file task1_AntattBest.c with matrix-Size = 3000 compared to ideal speedup, plooted by using professor Rio Martins script

If the two input matrices are sparse, one knows that the output matrix also will contain a lot of zeros. Hence, you can skip a lot of multiplications because the answer will be 0. If one manages to implement this efficiently into the code this will most likely improve the running time of the function (only when used on sparse matrices). This is something one can implement for further improvement of the code.

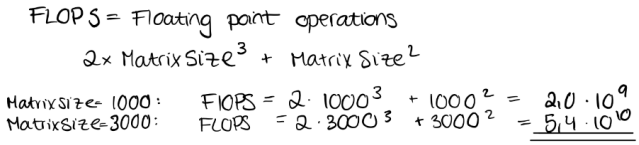
Looking at the FLOPS = floating point operations, wee see that the FLOPS is directly dependent on the MatrixSize. The calculation of the FLOPS is a result of the 2 operations inside the third for loop, added with the operation where D[i][j] is set to sum. These operations leads to the formula and calculations seen in figure 8.



Figur 6: Efficiency of timed matMulPar function in file task1_AntattBest.c with matrixSize = 3000, plotted by using professor Rio Martins script



Figur 7: Wall clock time of timed matMulPar function in file task1_AntattBest.c with matrixSize = 500, plotted against the ideal wall clock time by using professor Rio Martins script



Figur 8: FLOPS calculated for the matMul and matMulPar functions with MatrixSize = 1000 and MatrixSize = 3000

Conclusion

The plots and calculations of the functions differ from the expectations. However, by looking at the times of the different functions one see that the parallelized codes are faster than the sequential. To further parallelize the code I would try to use MPI to give even more specific instructions to the treads and how they should distribute the work.

On compering the plots and data from matrixSize 500, 1000 and 3000 it is a clear difference. The parallelized function works better on larger matrices. Hence, the program responds to weak scaling.

When looking at the plot for matrixSize = 3000 5 we can also conclude that the programs respond good to strong scaling.

Further improvement of the code could be done by identifying and handling multiplication between sparse matrices more efficiently.

Parallel matrix transposition

This task consists of transposing matrices, thought this task you will see this done by four different functions. Each function and its implemented solution will be presented below. However, the main goal of this task is to make the transposed matrices as quickly as possible by using OpenMP.

Solution implemented matT
The function matT takes a matrix A as an input, and returns the transposed AT. At the beginning of the function it allocates and initializes a new matrix called AT, which is of the same size as the input matrix. After this the timed section of the function starts. The timed part of the function is two nested for loops, that iterates through the input matrix and puts the elements in the transposed order in the AT matrix. This is done in one line, inside the two perfectly nested loops. When the program exits the two nested loops, the timer is stopped. The matrix AT is the transposed of the input matrix, and it is what the function returns.

Solution implemented matTPar
This function takes in a matrix A, and return the transposed matrix ATP.

During the implementation of the matTpar function I experimented with different OpenMP directives. Some of that experimentation is available in the file task2e1TimeTest.c. The implemented function has the same basic structure as matT, but with some additional lines for using the OpenMP. The function uses the directives parallel"and for", with the additional clauses "ordered", nowait"and schedule(static)". This means that the implicit control of threads is removed and the treads are given chunks of tasks in a sequential order. By other words I try to give the threads more specified tasks.

Solution implemented matBlockT
The function matBlockT also takes a matrix A and returns the transposed ABT. In this case the transposed matrix is allocated and initialized with the variable name ABT. In difference from the other functions this function transposes the matrix block-wise. One block of the matrix at the time. The size of these blocks are dependent of how many blocks the matrix is divided into, which is set by the variable nBlocks.

The timed part of this functions consists of four nested for loops. Where the two outer for loops are used to iterate block-wise through the matrix. The two inner for loops are used to iterate through each block. Then one line inside these four perfectly nested for loops put the elements in the right order in the transposed matrix. When these nested for loops are done, the timer is stopped. Then the transposed matrix is returned.

In this function the number of blocks are decided by the parameter nBlocks.

Solution implemented matBlockTPar
The function matBlockTPar also takes in a matrix A, and returns its transposed ABTP. This function is a parallelization of the function matTPar. In other words the logic of the function is the same as in matBlockTPar, but with additional lines for the parallelization

done by using OpenMP.

During the implementation of matBlockTPar, different approaches where tried. Some of these are available in the file task2e2TimeTest_c.c. The function I landed on is implemented by using the parallel fordirective with the clauses private(bi,bj,i,j)"and "collapse(2)". The collapse is possible to use since the loops are perfectly nested. An overview of the directives I tried and the clauses I tried are displayed in table 9.

In order to further optimize the code I tested different values of the nBlocks variable. This was done by running a file called nBlocksTest.c. The results of these tests are available in table 6.

Results

Function name	**Average time**	**Directive**	**Clauses**
matTParV1	0,00569	Parallel for	collapse(2)
matTParV2	0,00569	Parallel for	collapse(2), ordered
matTParV3	0,00375	Parallel for	collapse(2), ordered, schedule(static)
matTParV4	0,00345	Parallel, for	collapse(2), ordered, schedule(static), nowait
matTParV5	0,0346	single	-

Tabell 4: Time tests with OpenMP clauses and directives. The average time is calculated from 4 runs with 8 threads in the cluster. The functions is in the file task2e1TimeTests.c

Function name	**Average time**	**Directive**	**Clauses**
matBlockTParV1	0,00413	Parallel for	private(bi,bj,i,j), collapse(2)
matBlockTParV3	0,00558	Parallel, for	ordered, scheduled(dynamic), private(bi,bj,i,j)
matBlockTParV5	0,0148	Parallel, single	-

Tabell 5: Time tests OpenMP clauses and directives. The average time is calculated from 4 runs with 8 threads in the cluster. The functions is in the file task2e2TimeTest_c.c

Number of blocks	4	32	256	512	1024
Average time	0,248	0,127	0,189	0,156	0,142

Tabell 6: Tests run on function matBlockTParV1 to check run time with different number of block partitions. This test can be found in the file nBlocksTest.c

Performance discussion and analyzing matT vs MatTpar
In this task the plots are a representation of the strong scalability of the function with a specific matrixSize value.

Looking at the tables 7 and 8 one can see that the parallelized code for both matrixSize = 2048 and matrixSize = 16384 has improved a lot. This is also visible in the plots 6, 4 and 5. Looking at these plots one also sees that the improvement is smaller with thread

number of threads	**time[s], matrixSize = 2048**	**time[s], matrixSize = 16384**
1	0,03	5,33
2	0,03	5,30
4	0,03	4,82
8	0,03	4,78
16	0,03	5,13
32	0,03	4,34
64	0,03	4,91

Tabell 7: Time used by function matT from file task2_AntattBest.c on one run in the cluster at each matrixSize.

number of threads	**time[s], matrixSize = 2048**	**time[s], matrixSize = 16384**
1	0,0292	4,758
2	0,0149	2,231
4	0,00956	1,085
8	0,00530	0,679
16	0,00325	0,374
32	0,00309	0,308
64	0,0173	0,317

Tabell 8: Time used by function matTPar from file task2_AntattBest.c on one run in the cluster with each size of the matrix.

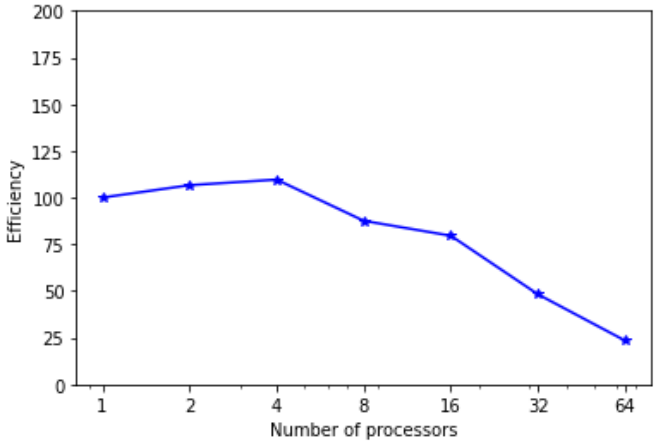
number of threads	**time[s], matrixSize = 2048**	**time[s], matrixSize = 16384**
1	0,01	24,40
2	0,02	22,41
4	0,02	23,25
8	0,02	24,49
16	0,02	24,46
32	0,02	20,55
64	0,02	23,65

Tabell 9: Time used by function matBlockT from file task2_AntattBest.c on one run in the cluster at each matrixSize

32 and 64, compered to the previous number of processors. This is evident when the timed line in the plot deviates from the ideal line in the plot in both 5 and 4. One can also see in plot 6 that the efficiency of the code drops significantly from 32 processor to 64 processors. Looking closer at table 7 and 8 with 64 and 32 processors one can see that code is slower with 64 processors for both the parallel and the sequential code.

number of threads	**time[s], matrixSize = 2048**	**time[s], matrixSize = 16384**
1	0,0164	24,670
2	0,00848	13,650
4	0,00591	6,704
8	0,00341	3,769
16	0,00197	1,914
32	0,00145	1,154
64	0,0109	1,387

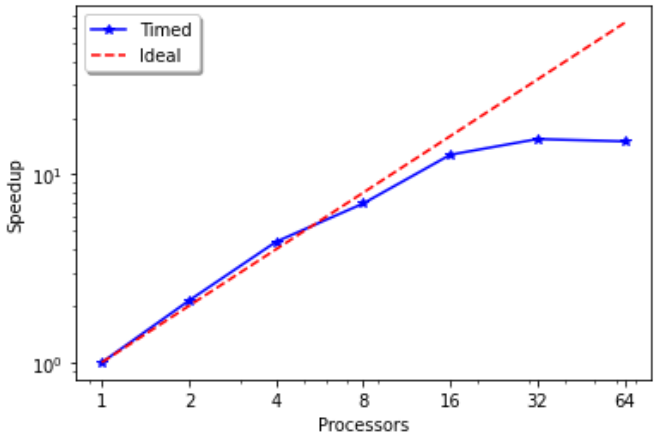
Tabell 10: Time used by function matBlockTPar from file task1.c on one run in the cluster



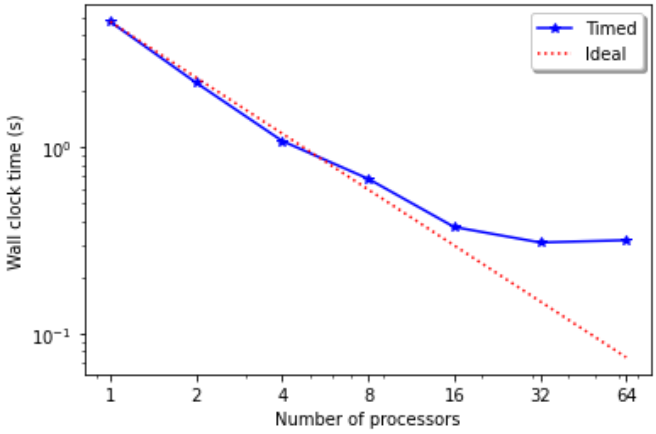
Figur 9: Efficiency in matTPar function in file task2_AntattBest.c with matrixSize = 16384, plotted by using professor Rio Martins script

One of the reasons for this increase in time can be that with 64 processors it becomes so much communication between the processors that this slows down the computation. That this happens for both the serial and the parallel code can underline this argument.

When transposing a matrix one has to flip all the rows with all the columns in the matrix. Normally when you work with dense matrices the columns and rows will consist of different values. And then you have to flip every element in all columns and rows. However if you have a sparse matrix the probability of two values that are switched or even a row and a column that are switched all consists of 0 is likely. If this is the case one can simply skip the switching of the element and just copy the elements right inside the new matrix. This could save time. However, the compatibility of the elements that will be switched also has to be checked which again is extra work for the function and then takes more time. So here the question would be if one saves time overall doing these checks or not.



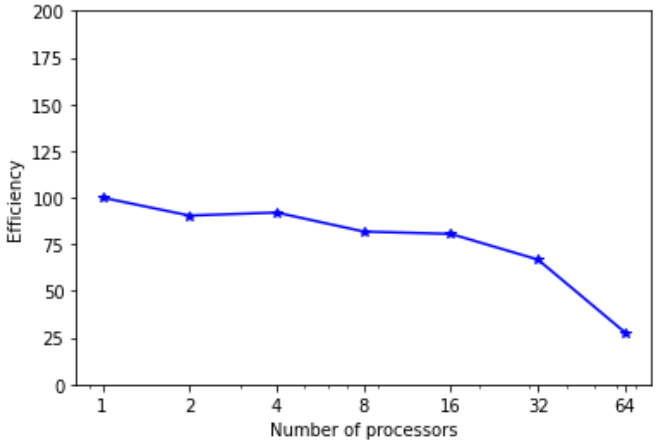
Figur 10: Speedup in matTPar function in file task2_AntattBest.c with matrixSize = 16384 compared to ideal speedup, plotted by using professor Rio Martins script



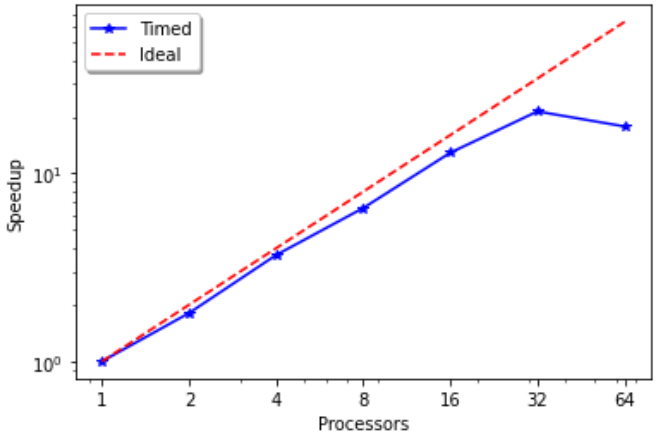
Figur 11: Wall clock time of timed matTPar function in file task2_AntattBest.c with matrixSize = 16384, plotted by using professor Rio Martins script

Performance discussion and analyzing matBlockT vs MatBlockTPar
Looking at the plots 12 for the matBlockTPar function one can see that the efficiency is high for all the processors up to 32, after this it falls with around 50%. In the speedup and wall clock time one also sees that the timed line almost follows the ideal line until 32 processors. Then at 64 processors it is quite bad in comparison. Overall the results are quite good and from the table 9 one can see that the code has improved with over 20 sec at the best from table 7.

Performance discussion and analyzing matTPar vs matBlockTPar When the matBlockTPar is compered to the matTPar plots, it is possible to see that the matBlockTPar is following the ideal line longer than matTPar.



Figur 12: Efficiency of timed matBlockTPar function in file task2_AntattBest.c with matrixSize = 16384, plotted by using professor Rio Martins script

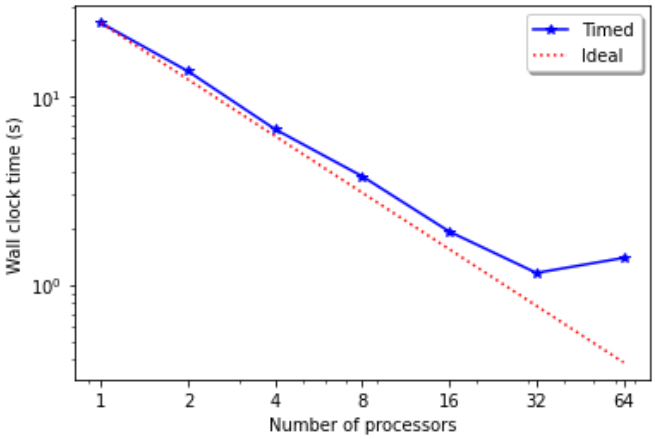


Figur 13: Speedup in matBlockTPar function in file task2_AntattBest.c with matrixSize = 16384 compared to ideal speedup, plotted by using professor Rio Martins script

In the not paralellized code one can see that the matBlockT is on average 0,1 sec faster when the matrixSize is 2048. However when the matrixSize = 16384 the matBlockT is around 20 sec slower than the matT code. This trend is also followed for the parallelized codes.

Comparing the bandwidth of the matBlockTPar in 16 and the bandwidth in matTPar 15 the matTPar has higher bandwidth for all the tested thread sizes. This is a bit unexpected as I would have expected the opposite.

In the function matBlockTPar the parallelization consists of a "collapse(2)", this means that just the other two loops are parallelized. In order to improve the code even more



Figur 14: Wall clock time of timed matBlockTPar function in file task2.c with matrixSize = 16384 compared to ideal speedup, plotted by using professor Rio Martins script

Bandwidth calculations matTPar

time 1 thread = 4,76 sec
time 8 threads = 0,68 sec
time 64 threads = 0,132 sec
Matrix size = 16 384 x 16 384
float = 4 bytes
Br+Bw = 2

Bandwidth x threads = $\frac{\text{data transferd}}{\text{time}} = \frac{2 \cdot 4 \cdot 16\,384^2 \text{ [bytes]}}{\text{time (x) [sec]}}$

Bandwidth 1 thread = $4{,}5 \cdot 10^8$ bytes/sec
Bandwidth 8 threads = $3{,}16 \cdot 10^9$ bytes/sec
Bandwidth 64 threads = $6{,}7 \cdot 10^9$ bytes/sec

Figur 15: Bandwidth calculations for matTPar

Bandwidth calculations matBlockTPar

time 1 thread = 24,67 sec
time 8 threads = 3,77 sec
time 64 threads = 1,39 sec
Matrix size = 16 384 x 16 384
float = 4 bytes
Br+Bw = 2

Bandwidth x threads = $\frac{\text{data transferd}}{\text{time}} = \frac{2 \cdot 4 \cdot 16\,384^2 \text{ [bytes]}}{\text{time (x) [sec]}}$

Bandwidth 1 thread = $8{,}7 \cdot 10^7$ bytes/sec
Bandwidth 8 threads = $5{,}7 \cdot 10^8$ bytes/sec
Bandwidth 64 threads = $1{,}5 \cdot 10^9$ bytes/sec

Figur 16: Bandwidth calculations for matBlockTPar

one idea could be to find a way to parallelize all the loops.

Conclusion
Further improvements to matBlockedTPar could be to try to parallelize all the four for loops.

In this code I have assumed that the data set will consist of dense matrices, hence the conclusion is that it would not be efficient to check the compatibility of each row and column for a direct copy.

Overall the parallelization makes the calculations faster. However, one see that the parallelization is better for 32 processors than for 64. This could be for several reasons but I think the increase in required communication is one of them.

Bibliography

here are some of the links that have been used as inspiration or to get facts to this project:

https://stackoverflow.com/questions/22634121/openmp-c-matrix-multiplication-run-slower-in-parallel

https://coliru.stacked-crooked.com/a/ee174916fa035f97

https://www.cse.iitd.ac.in/ dheerajb/OpenMP/codes/C/Omp_Matrix_Transpose.c

https://iq.opengenus.org/formula-for-flops-theoretical-max/

https://www3.nd.edu/ zxu2/acms60212-40212/Lec-12-OpenMP.pdf

https://www.programiz.com/c-programming/c-data-types#: :text=The%20size%20of%20float%20(single,d