Introduction to parallel computing, homework 3

Anne-Marthe Hvammen Sikkerboel
245997
anne.sikkerboel@studenti.unitn.it
link to git repo: https://github.com/AnneMartheH/H3_IPC_AM.git

8. januar 2024

Task 2, Parallel Matrix Transposition

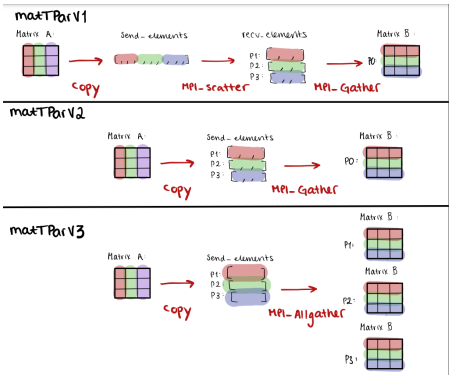
I chose task 2, parallel matrix multiplication.

Solution implemented matT (2.1)
The sequential code is the same as in a previous homework, explanation is found there.

Explored implementations for matTPar (2.2)
While making this function I experimented with different implementations in order to test which is the fastest. The first drafts of this function is available in the file *matTParV1*. In this file an array named $send\_elemets[MatrixSize * MatrixSize]$ is allocated. MatrixSize is the row size of the matrix. Through two for loops the matrix-values is copied into send_elements. Under this movement of values the first column is stored as the first elements of the matrix, and then the second column as the next values in the send_elements array. This leads to the matrix being stored in a column major order in the send_elements array. The next step i did was to MPI_scatter this array row wise out through the processors. Each processor then had a *recv_elements[MatrixSize]* array, with one column. Then the MPI_Gather function was called and all the recv_elements was collected into matrix B. Since the recv_elements each contained a column the matrix B became a transition of matrix A. Picture figure 1 is an illustration of this file.

After realising that communication is time-demanding, I looked for a solution demanding less collective communication. If each column of matrix A was directly copied into the processor handling it, one could skip the MPI_scatter function, and go directly to the MPI_Gather. This is illustrated in /textbfBilde A ref her. This function can be found in file *matTParV2*

Wanting to experiment some more with the function file *matTParV3* was created. Here matrix A is copied directly into *send_elements* as above, the different here is however what happens afterwards. In this file it is created a *MPI_Type_contiguous* called row_type. Which is used to send one row as one element of the datatype *row_type*, instead of



Figur 1: Illustration of differences in file matTParV1, matTParV2 and matTParV3

MatrixSize elements of *MPI_INT*. Afterwards the *MPI_Allgather* is used to put the processors together to matrix B. In comparison to the *MPI_Gather* the *MPI_Allgather* gives all the processor a filled in B matrix. Which means more communication than in file *matTParV2*.

In the files explained above the amount of computation is about the same, but the amount of communication is different. File matTParV2 has least computation and was therefore the quickest one.Hence, only results of testing on this file will be presented below.

Results

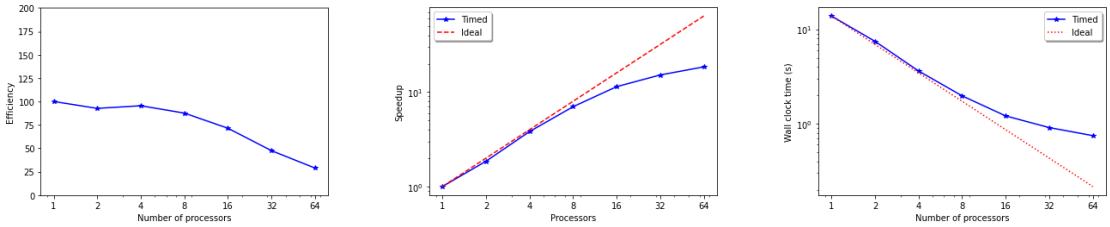
number of processors	**time[s], matrixSize = 4 096**	**time[s], matrixSize = 16 384**
1	0,1255	3,69

Tabell 1: Average time used by the sequential function matT on four runs, from file matrixT_sequential.c

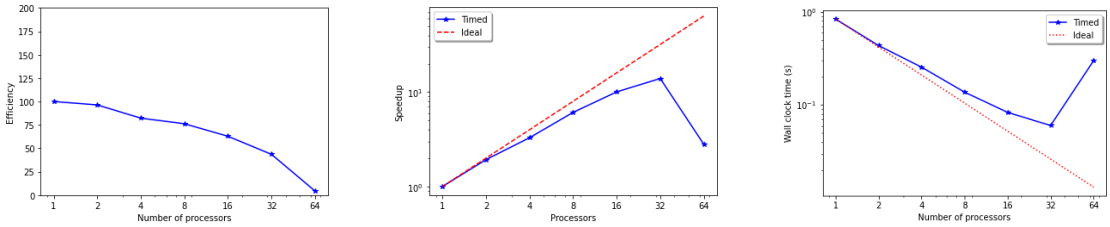
number of processors	**time[s], matrixSize = 4 096**	**time[s], matrixSize = 16 384**
1	0,84	13,83
2	0,44	7,46
4	0,26	3,62
8	0,14	1,98
16	0,083	1,21
32	0,060	0,91
64	0,30	0,75

Tabell 2: Average time used by function matTPar from file matTParV2.c, on four runs

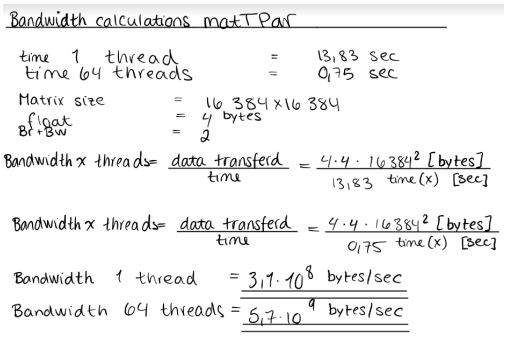
Observations and discussions



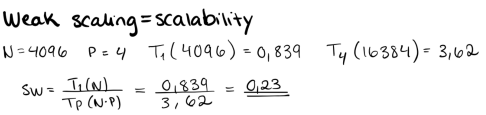
Figur 2: Efficiency, speedup and wallclock time from file matTParV2.c with matrixSize = 16384



Figur 3: Efficiency, speedup and wallclock time from file matTParV2.c with matrixSize = 4096



Figur 4: Bandwidth calculations for function matBlockTPar with MatrixSize = 16384



Figur 5: Weak scaling for matBlockTPar

Firstly commenting the results of the matTParV2.c file with a MatrixSize = 16384. Comparing the sequential time by function matT seen in table 1 with the time of matTPar run with 64 processors in table 2, one see that function matTpar is about 5 times faster. However when the function matTPar is running with 2 or less processors it is a lot slower than function matT.

Looking at the plots of function matTPar plot 2, one can see that the function has around 75% efficiency from 1 to 16 processors. Looking at the speedup one see the same here, with 32 and 64 processors the speedup diverges from the ideal line. This trend is even stronger when the MatrixSize is smaller. This is visable in figure 3. This could be because communication between more than 32 processor is so time demanding, the it starts to slow down the process.

Secondly commenting the results of the matTParV2.c file with a MatrixSize = 4096. Comparing the sequential time by function matT seen in table 1 with the time of matTPar run with 64 processors in table 2, one see that function matT is about 2,4 times faster. The function matTPar is only faster than matT when it runs with 16 and 32 processors. This could be because the MatrixSize is relatively small, which leads to less time consuming computations. Hence, the communication may take longer than the computations and we wont have any improvements in time.

Looking at figure 4 one see that that the bandwidth of the function matBlockTPar with 64 processors is greater than the bandwidth for 1 processor (sequential). In 5 one can see the calculation for the weak scaling.

Conclusion
From the results MPI is faster than sequential code, but only under certain conditions. In order for MPI to be quick the problems have to be computational large or time demanding. Otherwise the communication between the processor will add too much time.

Parallel matrix transposition

Solution implemented matT (2.3)
In the file task2BlockTransp.c you find the function called MatBlockTPar. This functions task was to transform a matrix, blockwise using MPI. To solve this task I have chosen that the number of blocks must be equal to the number of processors running. This means that when the matrixSize is big, the blockSize will also be big. Additionally, a fixed MatrixSize will not have a fixed BlockSize when running with different amount of processors. This implementation choice therefor effect the way I was able to produce results.

In the implementation of this code all memory was allocated statically. A new MPI_type called block_type gets initialized, so that it is simpler to send matrices. The first thing that is done in the timed section of this code is to split the array into transposed sub

arrays. All ranks, except rank 0, take its corresponding subarray and transpose it and save it into it's own personal sub_matrix. From here the processors send their sum_matrix to processor 0. Processor 0 starts by copying its own section of matrix A, then put the transposed version into matrix B. After this is done rank 0 is receiving all the sub_matrix from the other processors. Rank 0 receives this in order, meaning that it receives from processor 1 first, then processor 2, and then continues until it has received from all processors. After rank 0 has received a sub_matrix it copies this matrix into the bigger transposition matrix B.

Results

matBlockTPar	strong scaling		
MatrixSize	BlockSize	processors	average time
16384	16384	1	32,2508788
16384	8192	4	18,0465448
16384	4096	16	20,5605778
16384	2048	64	10,2932878

matBlockT	strong scaling		
MatrixSize	BlockSize	processors	average time
16384	16384	1	33,1231758
16384	8192	1	22,4403903
16384	4096	1	30,690445
16384	2048	1	21,3416888

matBlockTPar	weak scaling		
MatrixSize	BlockSize	processors	average time
4096	1024	1	1,30391675
16384	2048	64	10,2932878

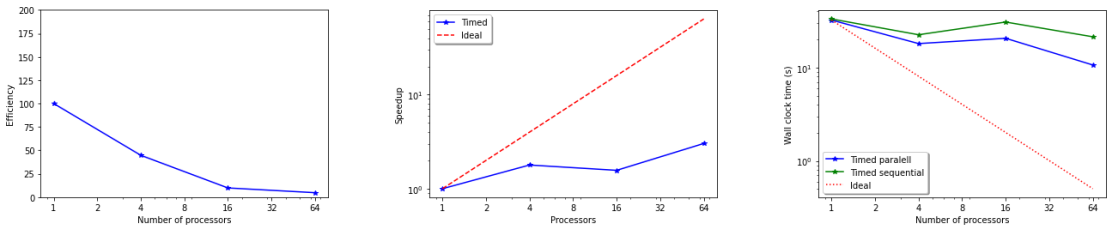
Figur 6: Runtime data used to calculate strong and weak scaling from function matBlockTPar from file task2BlockTransp.c and matBlockTPar from file task2BlockSeq.c

Discussion

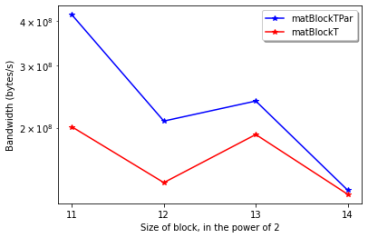
From looking at the plots in figure 7 we see that the function matBlockT's efficiency is quite bad. By looking at the plot for the speedup, the line representing the functions calculation time is quite far off from the ideal line. However, looking at the wallclock time we see that the function matBlockTPar is faster than the sequential matBlockT. In figure 9 you see the weak scaling. This value should be as close to 1 as possible, so this is also quite far off. In figure 10 one see that the function matTPar is faster than the matBlockTPar, this is not expected.

As seen in fig 8 the blockSize affects the bandwidth of the function a lot. The bandwidth 2^{11} is about four times better than the one for 2^{14}. This means that when I run the matBlockTPar with variable blockSizes to test the efficiency, speedup and wallclock time, the obtained results also varies a lot because of the different BlockSizes.

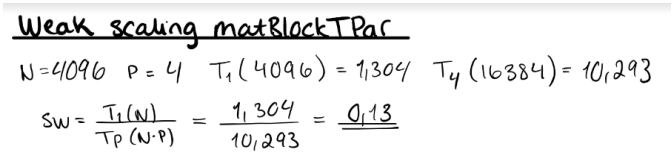
In figure 11 you see matBlockTPar's MPI_Recv function. This code snippet has quite a huge bottleneck in the first for-loop. Since the code is iterating through all the processors



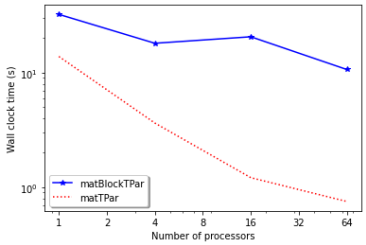
Figur 7: Efficency, speedup and wallclock time for function matBlockTPar. $BlockSize = 16384$ and $blockSize = MatrixSize/size$. The blockSize is not constant.



Figur 8: Bandwidth for matBlockTPar and matBlockT



Figur 9: Weak scaling for matBlockTPar



Figur 10: matBlockTPar compared to matTPar

```
//putting sub matrix in right place in glob matrix
    if (rank == 0){
        for (source = 1; source < size; source++){ //this is a bottleneck, especially if process 1 is slow
            int start_position_j_recv = (source % blocks_per_row) *BlockSize;
            int start_position_i_recv = (source / blocks_per_row) *BlockSize;
            MPI_Recv(&local, 1, block_type, source, source, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            for(i = 0; i < BlockSize; i++){
                for(j = 0; j < BlockSize; j++){
                        B[start_position_j_recv + j][start_position_i_recv + i]=local[j][i];
```

Figur 11: Bottleneck in code matBlockTPar

and receiving from one at the time you loose the parallelism. This is because the for-loop only takes one processor at the time. Then all the processors have to stand in a line and wait. If one of the processors take very long, this delays the entire process.

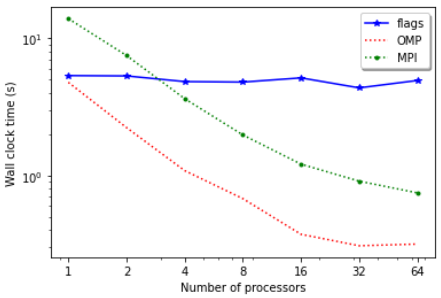
Conclusion
To improve this code I would start by allowing each processor to take several blocks. This would remove the dependencies of the blockSize when analyzing the code. Then one could also utilize the most efficient blockSize to make the code as fast as possible. Removing the bottleneck around the MPI_Recv would also increase the code. This can for example be done with a OpenMpi parallel forsection.

If the suggestions over had been followed, the code would have had a lot better performance. By taking advantage of the block division, I would expect the matBlockTPar function to be faster than the matTPar function.

Implicit parallelism, OpenMP and MPI

Performance
Since I in this task chose task 2. Matrix Transposition I am going to compare matrix transposition without blocks.

From figure 12 one can observe that the best runtime is dependent on how many processors the OMP and MPI code is running. For one processor implicit parallelism and OMP is about equal. For less than four processors MPI would not be a time optimal choice,



Figur 12: Illustration of matrix transposition parallelized differently, and how this affects the wallclock time. MatrixSize = 16384

because it is slower than both implicit parallelism and parallelism with OMP.

Overall the OMP gives the best run time. This might be because MPI requires a lot of communication, since they do not have shared memory. A lot of this is avoided in the OMP as it is a shared memory solution. Implicit parallelism is only ran on one processor, and can therefore not distribute work to other processors.

To maximise time-optimization it could be clever to combine MPI and OpenMP, so that you can use the advantages of both the methods. And specify more precisely some parts of OpenMp, with the MPI directive.

Scalability
From plot 12 on can see that the both MPI and OMP is more effective when using more processors. However this is not the case for implicit parallelism with flags.

One can argue that for small problems implicit parallelism is good, because it does not demand a lot of communication, or writing to variables in the same memory area. Hence, it can cause less problems.

However, when the problems demand more computational power, implicit parallelism will not give a huge improvement, therefore using OMP and MPI is a good idea. This is also in line with my experiences with small matrices in this and the previous homework. The efficiency and speedup tend to go further away from the optimal.

Ease of implementation
Implementing the implicit parallelism was easy. I used the builtin prefetching function and compilation flags to activate implicit parallelism. The builtin prefetch function is one, maximum two lines and the use of compilation flags is some additional words when you compile the code.

OpenMPI was also reasonable easy to implement, here you start the parallel sections and then use again prebuilt functions and clauses to make the code parallel. Here you decide what part of the code you want to run in parallel and also how it is to be parallelized, compared to with implicit parallelism. Hence you use several lines in the code for this,

but its still fairly reasonable.

Compared to implicit parallelism and OpenMP, implementing MPI is a very hard. In MPI you decide more parts of the parallelization, which means you have to tell all the processors exactly what they should do. This is quite time and logic demanding.

Conclusion
Which parallelization method I would have chosen is dependent on the task and available processors. For only on processor I would go for implicit parallelism since this is the equally good as OMP on one processor.

However, if performance is very important I would combine OpenMP with MPI, because this will give the fastest result. Then you can combine shared memory and distribution over multiple processors. However, if the performance is only a bit important I would go for the OMP.

If it was important to have absolute control over the parallelism, the only logical choice would be MPI.

Based on a general evaluation to speed up a code I would prefer OMP, as it is quite simple and quite good.

Different hardware architectures
SMP, symetric multiprocessing architecture, is an architecture with multiple cores sharing the same memory. Since OpenMP based on threads accessing shared memory, I think it would have good performance on SMP.

On the other side you have MPI, which is based on several processors communicating, and passing information to each other. For this is natural to have a system consisting of several processors. This can typically be found in clusters.

The speedup from implicit parallelism depends on the compilers ability to detect parts or patterns of the code that can be implicit parallelised. This might be easier when all parts of memory is located at one place. Hence clusters may not be optimal for implicit parallelism.

Bibliography

- https://www.mpich.org/static/docs/v3.1/www3/MPI_Type_create_subarray.html
- http://personalpages.to.infn.it/m̃ignone/MPI/lecture4.pdf
- https://www.mpich.org/static/docs/v3.4.x/www3/MPI_Gatherv.html
- https://scicomp.stackexchange.com/questions/1573/is-there-an-mpi-all-gather-operation-for-matrices
-https://www.tutorialspoint.com/cprogramming/c_multi_dimensional_arrays.htm
- https://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/
- https://stackoverflow.com/questions/42474048/mpi-c-gather-2d-array-segments-into-one-global-array