HPC Lab

Project

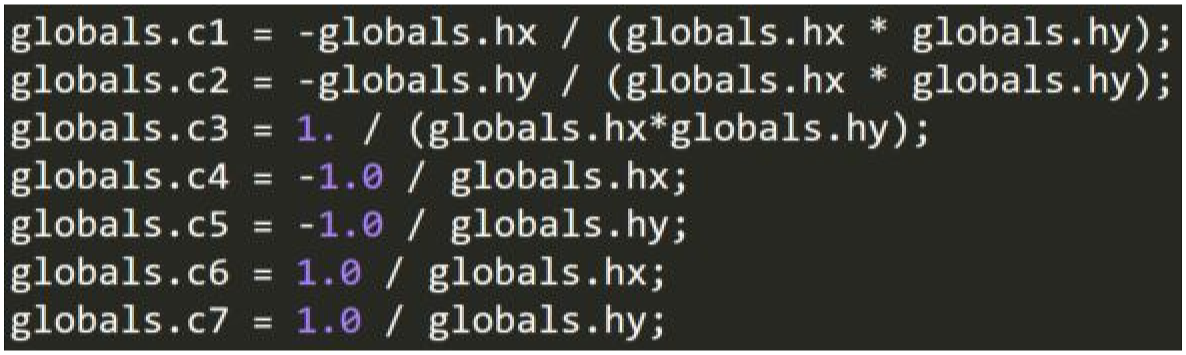
Erkin Kirdan - Manuel Rothenberg - Gabrielle Poerwawinata

For the assigned Project we split the task in three categories. Single-Core Optimization, OpenMP and MPI. To measure the optimization success, we take the time of the execution of the simulation function, because that's where the real calculations take place. Also we measure the time in seconds and not GFLOPs or something similar, because some optimizations result in less Operations needed and this would not improve the resulting Measurement regarding the comparability. Also what counts in the end is the needed time to solve the problem.

Single-Core Optimization

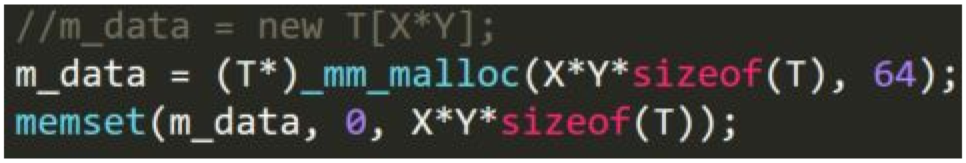
For the optimization of the operations that happen on a single core we first analyzed the given code. Most of the time the kernel functions are called, which themselves call the DGEMM function mostly.

1.) Elimination of Divisions and unnecessary calculations
Also during the simulation, there are division of constants. The results of these calculations never change. So we searched for such calculations and precalculate them in the beginning and just get the result later from these precalculated values. For this, we added seven constants to the global structure:



```
globals.c1 = -globals.hx / (globals.hx * globals.hy);
globals.c2 = -globals.hy / (globals.hx * globals.hy);
globals.c3 = 1. / (globals.hx*globals.hy);
globals.c4 = -1.0 / globals.hx;
globals.c5 = -1.0 / globals.hy;
globals.c6 = 1.0 / globals.hx;
globals.c7 = 1.0 / globals.hy;
```

2.) Alignment of memory
To vectorize more easy, we aligned all variables which are build with the grid-Template.



```
//m_data = new T[X*Y];
m_data = (T*)_mm_malloc(X*Y*sizeof(T), 64);
memset(m_data, 0, X*Y*sizeof(T));
```

3.) GlobalMatrices Access
The constant matrices in the file GlobalMatrices.cpp are heavily used in Simulator.cpp and Kernels.cpp. These matrices are only used directly in DGEMM() or in computeFlux() where they are also used in DGEMM(). They always end up being used as the parameter A in DGEMM(). In DGEMM() the parameter A is used to calculate multiplications and this happens not straight through the memory. In the most inner loop in DGEMM() the index used to access A is always

incremented by the width of the matrix, so there are heavy jumps in memory. That's why we transposed the matrices in GlobalMatrices.cpp and added a new function opt_DGEMM1(), which is only used for DGEMM with matrices out of GlobalMatrices. This functions goes through the matrix like it is stored in memory, incrementing the index only by one for each loop.

4.) GlobalMatrices Vectorization/Intrinsics
Now we go through GlobalMatrices one by one in the order the values are stored in memory. To get a better performance, we used Intel intrinsics to vectorize the implementation. Now we calculate 8 multiplications per cycle. For this, we also padded each row in the matrices of GlobalMatrices, to get rows which are divisible by 8. For example for the order 10 each row, had 55 entries, we padded each row by one zero, to get 56 entries per row. Since we padded it with zero, the result of this multiplication will also be zero, which therefore has no effect on the actual result we want to calculate.

5.) Temporary Matrix
In computeAder(), computeVolumeIntegral() and computeFlux() there is a temporary matrix tmp, which is used first to store results of DGEMM() and then used as input to DGEMM(). We transposed this matrix as well and also padded it with zeros to get rows of 8 values instead of just 3. We changed opt_DGEMM1() to store values in this changed matrix in the correct way. Then we added a new function opt_DGEMM2(), which is used for calculations where this tmp matrix is used as input. It also works with Intel intrinsics and loads one row of tmp instead of three separate calculations. With the help of this we could eliminate the most inner loop for these calculations.
In computeAder() and computeVolumeIntegral() there are DGEMM() calls where this temporary matrix as well as the matrix A or B is used as input. These matrices A and B are small and sparse, that's why we use two more functions opt_DGEMM4A() and opt_DGEMM4B() which are just used for calculations with these matrices and tmp as input. A and B just have 2 non-zero values, so in opt_DGEMM4A() and opt_DGEMM4B() we just calculate the rows, where non-zero values are existing. These two functions are like opt_DGEMM2(), but the inner loop is unrolled and adapted to just perform the necessary calculations.

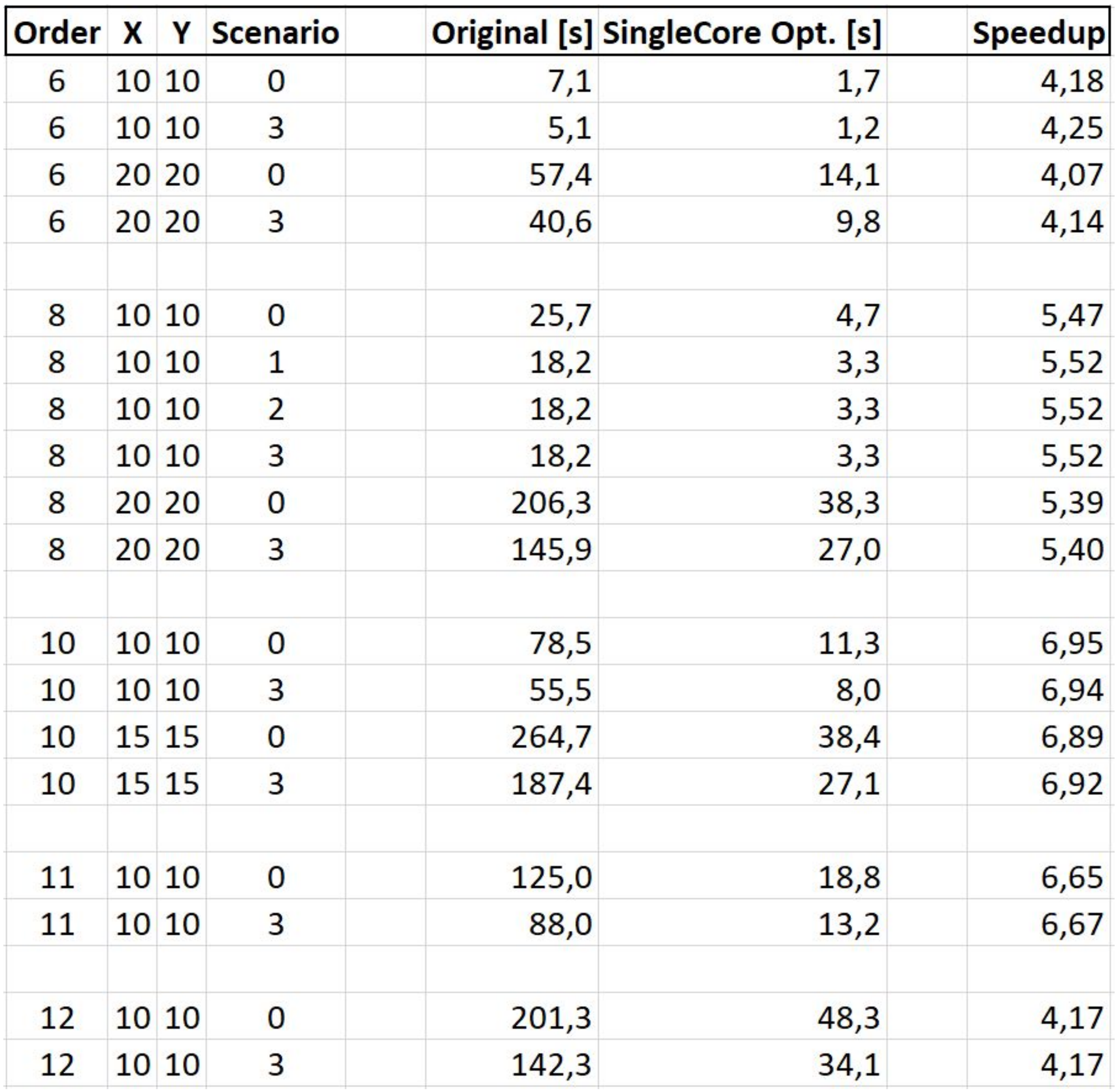
6.) More optimization
We now have 4 different DGEMM implementations for specific scenarios. These functions have the parameter beta which is used as a multiplication factor. In the most cases, this parameter is either zero or one, that's why we removed beta in the code and just add the value if it was multiplied by one and removed the addition if it would have been multiplied by zero.
Also the B matrix in all new DGEMM-functions is pre-loaded in registers only once and reused if possible, since in most cases, this matrix is smaller and never changed.

7.) Not implemented optimization ideas
The matrices in GlobalMatrices are often very sparse. That's why we tried to find a pattern, to just perform the necessary calculations. Since we use vectorization and there is no real (implementable) pattern, we didn't implement this.

Also the matrices KetaT ,Keta, KxiT, Kxi only have values in the top right or bottom left half. We tried to exploit this, by just going through these filled halfs, but didn't get a performance improvement. The program even was slightly slower, probably because of more Cache misses.

Results

With all the improvements explained in this chapter, we could speed up the execution of the simulation() function by up to ~7 times. Some examples comparing the Single-Core Optimization to the initial condition are shown in the following table.



Order	X	Y	Scenario	Original [s]	SingleCore Opt. [s]	Speedup
6	10	10	0	7,1	1,7	4,18
6	10	10	3	5,1	1,2	4,25
6	20	20	0	57,4	14,1	4,07
6	20	20	3	40,6	9,8	4,14
8	10	10	0	25,7	4,7	5,47
8	10	10	1	18,2	3,3	5,52
8	10	10	2	18,2	3,3	5,52
8	10	10	3	18,2	3,3	5,52
8	20	20	0	206,3	38,3	5,39
8	20	20	3	145,9	27,0	5,40
10	10	10	0	78,5	11,3	6,95
10	10	10	3	55,5	8,0	6,94
10	15	15	0	264,7	38,4	6,89
10	15	15	3	187,4	27,1	6,92
11	10	10	0	125,0	18,8	6,65
11	10	10	3	88,0	13,2	6,67
12	10	10	0	201,3	48,3	4,17
12	10	10	3	142,3	34,1	4,17

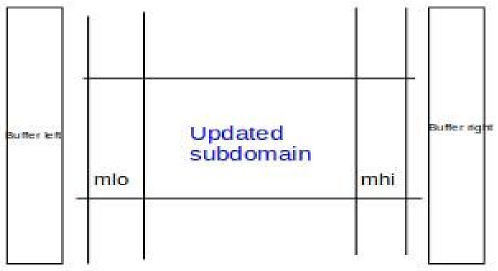
The initial implementation took very long for higher orders and higher X and Y, that's why we compare only up to X=Y=20. But nevertheless, we see, that the speedup depends mostly on the order and not on the szenario or X and Y.

MPI

We are providing MPI parallelisation by using asynchronous communication and domain decomposition with copy and ghost layer.

Divide timeIntegratedGrid into several subdomain
Each of subdomain is updated by a single MPI process, provided that the boundary cells from adjacent subdomains are available. This boundary cells need to be exchanged between adjacent subdomains after each outer iteration. The size of each subdomain is defined by globals.Y * (globals.X/P), where P is the number of processor. Domain decomposition is based on 1D column wise domain decomposition.

If we run the application with 2 processor, when the P1 need adjacent value from another cell in P2, instead the P2 retrieved the individual value every time it is needed, they can send one column to P1. Hence, subdomain are extended by ghost-layers which hold boundary data from adjacent subdomains. In this case we are divide the grid into column wise subdomain grid.
In each of the subdomain has mlo and mhi index as the boundary with the neighbor.



The definition of mlo and mhi depend on the value of rank they have.

```
if(rank ==0){
     mlo =0;
     mhi = mlo + stripsize-1;
}else if(rank==ntasks-1){
    mlo =(rank * stripsize) -1;
    mhi = globals.X -1;
}else{
```

```
            mlo = (rank * stripsize)-1;
            mhi = mlo + stripsize-1;
        }
```

There are additional buffer to store mlo column (buffer left) and mhi column (buffer right) for communication with the neighbour.
We define a variable called *stripsize* which is a size of column subdomain that each of the process has.

```
        stripsize = globals.X/P;
```

There are 2 loop in the simulate function, the first loop is doing computation only on its local cell without communication with the neighbor. The second loop, timeIntegratedGrid of one cell will communicate with other cell. Therefore MPI asynchronous communication is placed on the second loop.
Original simulate function were done in grid *x* {0..globals.X} and *y* {0 .. globals.Y}. In parallel version we can divide the grid accordingly with the number of processor.
Serial without ghost cell decomposition

```
 for (int y = 0; y < globals.Y; ++y) {
        for (int x = 0; x < globals.X; ++x) {
                // doing some computation here
        }
}
```

With decomposition of P processor, the iteration of coordinate x is changed.

```
for (int y = 0; y < globals.Y; ++y) {
        for (int x = mlo; x <= mhi; ++x) {
                // doing some computation here
        }
}
```

Iteration above only applied to the first loop, because on the second loop we applied a ghost cell, which the inner subdomain lowest *x* index become mlo+1 and the highest index become mhi-1.

Overlapping communication and computation

1. Compute the first loop in simulate function based on the size of each process.
2. Receive a column from left rank for buffer left and receive a column from right for buffer right.

```
         //receive from the left
        MPI_Irecv(&timeIntegratedGridBuffLeft.get(0,0), globals.Y, MPI_DOUBLE, left, 1, MPI_COMM_WORLD, &request[0]);
        //receive from the right
        MPI_Irecv(&timeIntegratedGridBuffRight.get(0,0), globals.Y, MPI_DOUBLE, right, 0, MPI_COMM_WORLD,
&request[1]);
```

3. Compute 2 halo/boundary columns

For the halo column on the left the value of *timeIntegratedGrid.get(x-1,y*) can be obtained from *timeIntegratedGridBuffLeft.get(0,y)*. The halo column in the right when it need *timeIntegratedGrid.get(x+1,y)* the value can be obtained from *timeIntegratedGridBuffRight(0,y)*.

4. Copy the result of left halo to buffer left and result of right halo to buffer right in order to send communication with the neighbor.
5. Do MPI_Test and MPI_Wait to guarantee that the receive request has finished.
6. Send the value of buffer to the neighbour

```
//send to the right
MPI_Isend(&timeIntegratedGridBuffRight.get(0,0), globals.Y, MPI_DOUBLE, right, 1, MPI_COMM_WORLD, &request[2]);
//send to the left
MPI_Isend(&timeIntegratedGridBuffLeft.get(0,0), globals.Y, MPI_DOUBLE, left, 0, MPI_COMM_WORLD, &request[3]);
```

7. Compute the remaining grid of inner subdomain
8. Do MPI_Test and MPI_Wait to guarantee that the send request has finished.

The modification for asynchronous communication and ghost cell implementation were done in simulate procedure in *Simulator.cpp.*

Guarantee the completeness of process

Because it uses asynchronous communication, a non-blocking send or receive must make sure that the operation has completed before it proceeds with its computations. To check the completion of non-blocking send and receive operations, MPI provides functions of MPI_Test and MPI_Wait. MPI_Test test whether the non-blocking send or receive operation identified by certain request has finished. It will return a flag = true if it completed, otherwise the flag = false. MPI_Wait blocks until the non-blocking operation identified by request completes.

```
    .. halo columns cmputation

  //waiting on receive buffer
    MPI_Wait(&request[0], &status[0]);
    MPI_Wait(&request[1], &status[1]);
    MPI_Test(&request[0], &flag[0], &status[0]);
    MPI_Test(&request[1], &flag[1], &status[1]);

     .. inner subdomain without halo columns computation

    //waiting on send to release buffer
    MPI_Wait(&request[2], &status[2]);
    MPI_Wait(&request[3], &status[3]);
    MPI_Test(&request[2], &flag[2], &status[2]);
    MPI_Test(&request[3], &flag[3], &status[3]);
```

L2 error MPI_Allreduce

There are 3 attributes of error that have to be computed (pressure, x-velocity, y velocity), for each of them we put MPI_Allreduce for the sum operation.

```
for (unsigned q = 0; q < NUMBER_OF_QUANTITIES; ++q) {
        MPI_Allreduce(&l2error[q],&l2error_local[q], 1, MPI_DOUBLE, MPI_SUM,    MPI_COMM_WORLD);
    l2error[q] = sqrt(l2error_local[q]);
 }
```

Results
When we applied domain decomposition and ghost cell our l2error are off the original l2error. We tried a lot of things, but were not able to make the MPI with ghost cell running until the deadline of this assignment was given. We put the not working ghost cell with MPI implementation in file Simulator_MPI.cpp.

OMP

In Simulator.cpp within the simulation() function are two big nested for-loop constructions. Both are going through the grid to perform the necessary calculations. We parallelized these for-loops to distribute the workload to all cores on the node. This was done with the following pragma before the two outer loops.

```
#pragma omp parallel for collapse(2)
for (int y = 0; y < globals.Y; ++y) {
        for (int x = 0; x < globals.X; ++x) {
```

Here, the collapse option ensures two nested for loops to be parallelized. With HyperThreading it is possible to share the workload between 64*4=256 OMP threads. That's why maximum speedup is observed when number of threads is set to 256. After these pragmas it is observed that for particular convergence orders, like 6 or 10, it worked fine, but for some others, like 8 or 11, Pressue (p), X-Velocity (u), and Y-Velocity (v) are -nan.

In GEMM.cpp, OMP parallelization of for loops are also tried in optimized DGEMM functions, but, in each try the time is either stayed same or increased. So, no OMP parallelization is used in GEMM.cpp.

Results

order	x	y	Scen ario	After some single core optimization	After some single core optimization + openmp, threads = 256
6	20	20	0	14367.6ms	482.773ms
6	20	20	1	10129.9ms	428.636ms
6	20	20	2	9957.05ms	417.728ms
8	20	20	0	38154.4ms	986.731ms
8	20	20	1	27330.3ms	737.231ms
8	20	20	2	26985.1ms	715.826ms
10	20	20	0	91577.5ms	2280.89ms
10	20	20	1	64824.6ms	1711.85ms
10	20	20	2	64796.9ms	1734.34ms
10	50	50	0	<very long>	20393ms
10	100	100	0	<very long>	151024ms

Execution Instructions

Our optimized implementation can be found in the _project folder of the GIT repository. The building with scons and execution is the same like with the initial given program. The only change is, that before execution the OMP_NUM_THREADS environment variable has to be set to the wanted OMP Thread-Number. To use all cores of a node in the best way it has to be set to 256.