Technische Universität München
Institut für Informatik
Prof. Dr. Hans-Joachim Bungartz
Philipp Neumann, Atanas Atanasov, Christoph Kowitz

Summer Term 2012
Worksheet 4

Praktikum Wissenschaftliches Rechnen
Computational Fluid Dynamics

Worksheet 4

Parallelization

deadline: June 11, 10 AM

Part I. Introduction to Parallelization

1. Parallelization

In this worksheet, we address the parallelization of our flow simulation program. With parallelization we intend to reduce the total computing time by distributing the computational work among several processors, which perform the calculations concurrently. In addition, we often profit from a substantially larger amount of main memory compared to the memory that is available on sequential or stand-alone machines. This will allow us to solve larger problems.

In this worksheet you will implement a distributed memory parallelization for the Navier-Stokes Equations **OR** for the Lattice Boltzmann Method.
For the sake of simplicity, we will restrict ourself to the parallelization of the algorithm presented in Worksheet 1. This is a driven cavity scenario which uses an SOR iteration to solve the pressure equation.

2. Parallel Computers and Programming Environments

Before starting the exercises, we will give a little overview on parallel computers and parallel programming. If you're familiar with all this, you're welcome to skip this section.

We will start with a classification of possible (parallel) architectures. This very common classification was given by Flynn: It distinguishes between four basic types – SISD, MISD, SIMD, and MIMD – which can be explained as follows:

- Very simple machines belong to the **SISD** class (single instruction stream, single data stream), which means that all instructions are executed one at a time by a single-processor, and each instruction is executed on a single piece of data. Actually, this is exactly our straightforward sequential programming model and does not involve any kind of parallelism.

- On a **MISD** type machine (multiple instruction stream, single data stream), different instructions can simultaneously operate on a single piece of data. Pipeline architectures belong to this type.

- In contrast, the **SIMD** class (single instruction stream, multiple data stream) is one of the two main architectures used for high performance computing. Representatives of this class are, basically, all vector computers (e.g. Fujitsu's VPP700 at LRZ). Vector operations are classical examples where such a machine is useful: a single instruction will be performed for all components of a vector at once (e.g. the d multiplications required for computing $a \cdot x\ x \in \mathbf{R}^d$ are computed simultanously). Nowadays, many regular CPUs include SIMD instructions, at least on a small scale. Some basic SSE instructions (streaming SIMD extensions) on a Pentium CPU, for example, allow the parallel execution of one floating point operation on a set of up to four floating point values.

- The most important class in the supercomputing sector is that of the MIMD computers (multiple instruction stream, multiple data stream). This class can be considered to contain the "'true"' parallel computers, since here each processor has its own instruction and data streams. We can further discriminate within this category with respect to the way the memory is addressed:
  - computers with distributed memory (**MIMD distributed address space machines**)and
  - computers with a global memory (**MIMD global address space machines**).

For programming such machines, there are two important paradigms: Either all nodes work on one (virtual) global memory or the nodes communicate via messages. The first paradigm fits perfectly to global address space architectures, whereas the latter one fits to distributed memory systems. Nevertheless, each programming paradigm might be applied on any architecture. In this Practical Course, we shall concentrate on distributed memory machines programmed by message passing, since on workstation or PC clusters, the distributed programming approach is most common, i.e. multiple parallel programs are run as processes or tasks on each processor. Data allocation occurs locally on each processor, and data exchange between individual processors is programmed explicitly with the help of respective instructions (send, receive, etc.). Programming of data communication is facilitated by communication libraries such as MPI (Message Passing Interface), which presents a hardware-independent interface. Respective libraries are offered by different vendors, some of them free of charge, and are available for Fortran, C and C++ languages. We will explain the most important MPI commands throughout the next chapters and, especially, in the appendix.

To finish this introductory section, we would like to present two websites related to supercomputing. First, there is the list of the world's 500 currently fastest supercomputers

`http://www.netlib.org/benchmark/top500.html`,

and second, the website

`http://www.lrz-muenchen.de/services/compute/index.html`

will familiarize you with the high performance computers at the LRZ in Munich, where probably the Hitachi machine SR8000-F1 (maybe already the new SGI ALTIX) and the Linux cluster are the

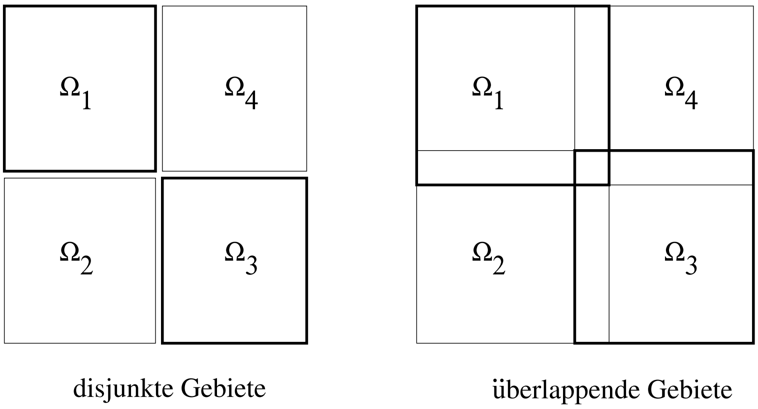

Figure 1: Domain decomposition methods with non-overlapping (left) and overlapping (right) partitions

most interesting. You can also find links for a good overview of construction, functional mechanisms and performance of parallel computers.

3. Domain Decomposition as a Parallelization Strategy

A common approach to parallelize numerical algorithms for solving partial differential equations on MIMD architectures is to divide the computational domain Ω into subdomains $\Omega_1 \ldots \Omega_N$ and to treat each subdomain on a separate processor. The first so called *domain decomposition method* was developed in 1870 by H.A. Schwarz (a large class of domain decompositions is therefore referred to as Schwarz methods). Of course, in the 19th century, he did not invent this method to solve problems on a computer – instead, he was looking for means to find analytical solutions for PDEs.

In a strict sense, the term *domain decomposition* is used for a family of numerical methods that are based on solving a problem on each subdomain separately and combine these solutions to a global solution (in an iterative scheme, if necessary). Domain decomposition methods can be classified into overlapping and non-overlapping variants, depending on whether the individual subdomains overlap or not (see Figure 1). Moreover, we distinguish between additive and multiplicative domain decomposition methods. In multiplicative methods, the solutions of the subdomains are computed one after another, and the results from the previous subdomains may be used to compute that of the next domain. In contrast, additive methods compute the solution for all subdomains at once, and combine these solutions to obtain an update of the global solution. This option sounds especially attractive for parallelization!
Thus, each subdomain is assigned to a process that approximately computes the unknowns belonging to this subdomain. Each individual process, therefore, no longer requires access to the whole data set, but rather to a part of it. All processes can work in parallel on their part of the global data. The resulting advantage is two-fold: execution time is reduced because several processors are sharing the work, and there is considerably more memory space available for the global problem than on sequential machines, because all processors can contribute their local memory.

The computig time will be reduced most efficiently, if the computing work is equally distributed among the processors. In our application, this is accomplished when the subdomains each processor has to compute are of equal size and contain the same number of unknowns.

To ensure convergence of the algorithm on the entire domain, there has to be some communication

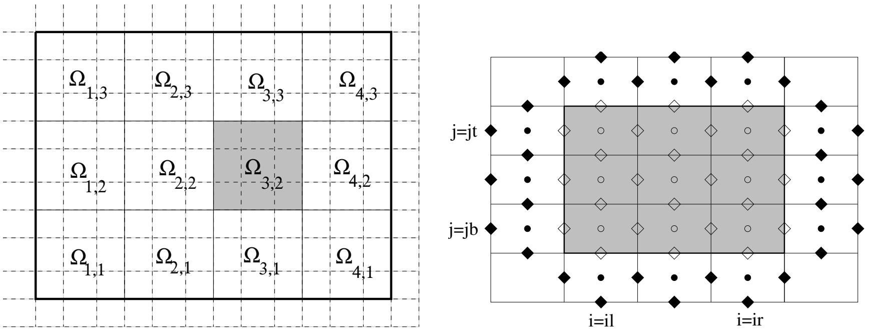

Figure 2: **left:** Partitioning of the domain Ω (`iproc`=4, `jproc`=3) **right:** Unknowns and boundary values for the subdomain $\Omega_{3,2}$

between the domains. An exchange of relevant data between processors becomes necessary at certain times. In the case of domain decomposition, this concerns primarily the values of unknowns at the boundaries of neighboring processes. Exchange of data between processors, however, is an expensive operation, which can be several orders of magnitude slower than just copying data within main memory. Thus, the amount of data that has to be exchanged should be kept as small as possible, and the number of communication steps should be reduced to a minimum since establishing a connection between processes is especially expensive. Not paying attention to communication costs can completely annihilate the desired efficiency gain. Thus, one should try to minimize the domain borders, since the number of unknowns corresponds to the size of this border.

In a broader sense, any approach that will distribute the computational domain of a problem to different processors is likely to be called a *domain decomposition* approach, even if it is not using the respective numerical schemes. The approach used in this worksheet will mainly stick to the idea of distributing the domain, too, and will not really make use of the numerical domain decomposition approach.

Part II. Navier-Stokes

4. Parallelization of the Flow Code

Let's now turn to the parallelization of the sequential program described in Worksheet 1. We begin by dividing the base domain Ω for the flow calculation along the grid lines into i_{proc} parts in the x direction and j_{proc} parts in the y direction, so that we get $i_{proc} \cdot j_{proc}$ rectangular subdomains $\Omega_{ip,jp}$, $ip = 1, \ldots, i_{proc}$, $jp = 1, \ldots, j_{proc}$. This decomposition is represented in Figure 2 where the grid lines of the staggered grid are shown as dashed lines and the subdomain boundaries as solid lines. In each process, the pressure and velocity values in the interior of the associated subdomain are calculated. To save an additional communication step, velocities lying exactly on a subdomain boundary are computed by both processes involved.
Thus, a process treating a subdomain $[(il-1)\,\delta x, ir\,\delta x] \times [(jb-1)\,\delta y, jt\,\delta y]$ will require the values shown in Figure 2. The values lying on points marked with white symbols must be computed by the process itself, whereas the values in the black points only serve as boundary data to determine the values in the interior and on the subdomain boundary. These boundary values are – unless they fall

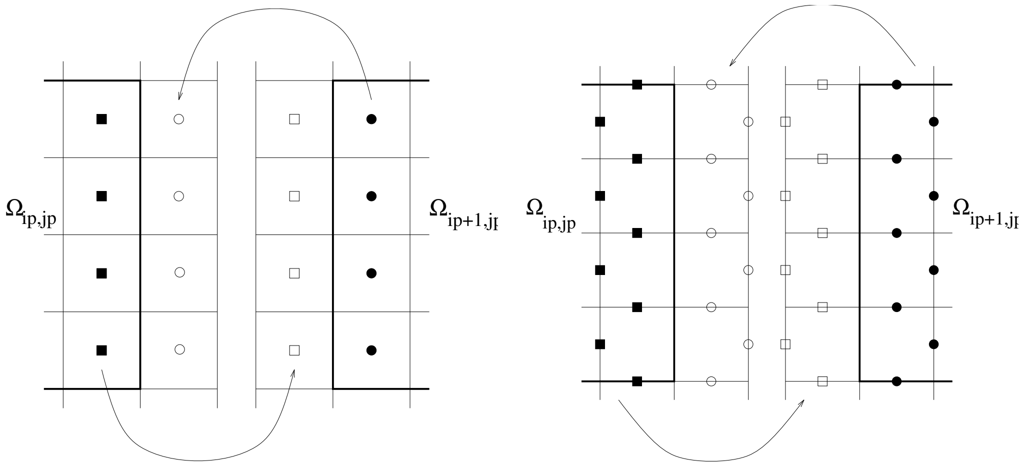

Figure 3: **left:** Exchange of pressure values **right:** Exchange of velocity values

outside the total domain Ω – calculated by the neighboring processes. The boundary values must therefore be sent, in an appropriate form and with a proper frequency, by the neighboring processes. The following communication steps have to be implemented:

- After each SOR step in the pressure iteration, the pressure values located in the boundary strips must be exchanged as represented in Figure 3, such that the computation can proceed in the subdomains with the up-to-date boundary values. To avoid network overloading with too many simultaneous data bursts, the data exchange might be performed in four steps – to the left, to the right, up, and down – send all relevant data to the respective neighbour as block of data.

- Following the data exchange of one SOR smoothing step, the termination criteria has to be evaluated: Therefore, each node computes its contribution to the global residual and, afterwards, sends the partial sum to a single process (master process). This process then decides whether to terminate the pressure iteration (tolerance ε achieved or the maximum number of iterations `itmax` exceeded) and broadcasts a message to all other processes saying whether to perform the next iteration step or to terminate the pressure iteration.

- At the end of the pressure iteration the current valid pressure values are contained in the boundary strips, so the velocity values on the subdomain boundaries can be updated in `calculate_uv` without any new communication. However, following `calculate_uv`, the updated velocity values must be exchanged as depicted in Figure 3 to enable the calculation of F and G in the next time step.

- Finally, the new time step `dt` must be determined. To find it, the maximum absolute values of `U` und `V` must be first determined in `calculate_dt` for each subdomain, and these local maxima should be sent then to the master process. The latter determines the global maxima and disseminates them over all processes. After that, the CFL condition (see Worksheet 1, (19)) returns the same value of time step `dt` on each partial process. This procedure is, in principle, analogous to that of residual computation.

We may summarize all the steps in the following algorithm:

5. The algorithm

```
Set $t := 0$, $n := 0$
Assign initial values to $u, v, p$
While $t < t_{end}$
    Set boundary values for $u$ und $v$
    Compute $F^{(n)}$ and $G^{(n)}$ according to (Worksheet 1,(9),(10),(17))
    Compute the right-hand side of the pressure equation (Worksheet 1,(11))
    Set $it := 0$
    While $it < itmax$ and $\|r^{it}\| > eps$
       Set the appropriate pressure boundary values (Worksheet 1, (16))
       Perform an SOR cycle according to Worksheet 1,(12)
       Exchange the pressure values in the boundary strips
       Compute the partial residual sum and send it to the master process
       The *master process* computes the residual norm
             of the pressure equation $\|r^{it}\|$ and broadcasts
             it to all the other processes
       $it := it + 1$
    Compute $u^{(n+1)}$ and $v^{(n+1)}$ according to Worksheet 1,(7),(8)
    Exchange the velocity values in the boundary strips
    Compute the maximum values of $u^{(n+1)}$ and $v^{(n+1)}$ for each
          process and send them to the master process (local max values)
    The *master process* computes the global maximum values of $u^{(n+1)}$
          and $v^{(n+1)}$ and broadcasts them to all other processes
    Compute the common $\delta t$ according to Worksheet 1, (19)
    $t := t + \delta t$
    $n := n + 1$
```

6. Implementation on a Network of Computers Using MPI

MPI stands for *Message Passing Interface* and defines a standard interface to incorporate many heterogeneous computers (workstations, parallel computers, vector machines) into a single virtual homogeneous parallel computer network. This standard has been established by the *MPI-Forum* (`http://www.mpi-forum.org`) and is also supported by a number of computer vendors (IBM, SGI, HP, etc.) providing their own libraries containing their native compilers and hardware extensions. Furthermore, there are also "'Public Domain Implementations"' available, as, for example, *mpich* (`http://www-unix.mcs.anl.gov/mpi/`) – the implementation used at this Chair, which is going to be employed in the present Practical Course.

The parallel program development with MPI does not imply writing a different program for each process. Instead of this, one program is distributed among all the nodes, i.e. several instances of the program are running in the cluster (*SPMD* – single program multiple data, a special case of MIMD). To each program instance, a unique variable (process ID) is assigned and all the sender / receiver identification is done using these IDs. To make MPI work properly, the directories in which the application is stored have to be the same on each node. For a network file system like the one existing at this Chair this holds.

In the parallelization of our flow code, we follow the principle of an additive, disjunct domain decomposition approach described in Section 4. Thus, in a processor treating the subdomain $[(il-1)\,\delta x, ir\,\delta x] \times [(jb-1)\,\delta y, jt\,\delta y]$, the local arrays must have the following dimensions:

- `P     [il-1,ir+1]`×`[jb-1,jt+1]`
- `U     [il-2,ir+1]`×`[jb-1,jt+1]`
- `V     [il-1,ir+1]`×`[jb-2,jt+1]` $(*)$
- `F     [il-2,ir+1]`×`[jb-1,jt+1]`
- `G     [il-1,ir+1]`×`[jb-2,jt+1]`
- `RS    [il,ir]`×`[jb,jt]`

Problems

1. Familiarize yourself with the concept of MPI-based parallel programming, MPI operations and the MPI environment on the lab's computers.

2. Copy from the Practical Course's web site the new program frames consisting of `Makefile, parallel.h, parallel.c`. Furthermore, you will need the files `init.h, init.c, boundary_val.h, boundary_val.c, uvp.h, uvp.c, visual.h, visual.c`, `helper.h, helper.c, main.c`.

3. Implement the following functions in `parallel.c` and `parallel.h`:

   -
     ```
     void init_parallel (int iproc,int jproc,int imax,int jmax,
           int *myrank,int *il,int *ir,int *jb,int *jt,
           int *rank_l,int *rank_r,int *rank_b,int *rank_t,
           int *omg_i,int *omg_j,int num_proc)
     ```

   The call of `MPI_Init(...)` must be present in the master program from the very beginning, so that in the routine to be written n processes should be taken into account.

At first, the boundaries il,ir,jb,jt and indices omg_i, omg_j for all subdomains must be computed in the master process and sent to the respectively assigned processes. For the computational domain of an arbitrary dimension and for an arbitrary number of processors (arranged in i- and j-directions), one must implement equipartitioning as uniform as possible.

Analogously, the neighborhood relationships should be determined, which means that the variables rank_l,rank_r,rank_b,rank_t should contain the process ID of the respective neighbors. The variables should be distributed for each subdomain, with the following MPI feature to be applied: If the subdomain lies, for example, at the physical boundary, then the process ID of a non-existent neighboring processes rank_l to the left can be designated by the constant MPI_PROC_NULL. Should the process identified with MPI_PROC_NULL attempt later to send/receive data, then the corresponding MPI routine would return NOP (no operation). This feature spares some extraneous case differentiation, which makes the code more compact, transparent and efficient.

-
  ```
  void pressure_comm(double **P,int il,int ir,int jb,int jt
       int rank_l,int rank_r,int rank_b,int rank_t,
       double *bufSend,double *bufRecv, MPI_Status *status, int chunk)
  ```

  This procedure will exchange pressure values between processors that treat adjacent subdomains (see Fig. 3.4). To evade a deadlock situation – for example, a situation where all processes are waiting to send or receive data, but to or from the wrong process –, one must stick to the following (or a similar) fixed order:

  send to the left — receive from the right,
  send to the right — receive from the left,
  send to the top — receive from the bottom,
  send to the bottom — receive from the top.

  Instead of the trivial solution consisting in the MPI_Send(...)/MPI_Recv(...) combination, you should, whenever possible, use more efficient variants with MPI_Sendrecv(...). Here, the vectors bufSend, bufRecv can serve as sending and receiving buffers. Be careful where to allocate the memory for your buffers bufSend and bufRecv, because these operations take a relatively long time.

  This function is called within sor before actually computing the residual.

-
  ```
  void uv_comm (double**U,double**V,int il,int ir, int jb,int jt,
       int rank_l,int rank_r,int rank_b,int rank_t,
       double *bufSend,double *bufRecv, MPI_Status *status, int chunk)
  ```

  This procedure is supposed to exchange the velocity values U und V between the processes treating adjacent subdomains (see Fig. 4.5). The data exchange should be performed in four steps as in pressure_comm, and bufSend as well as bufRecv have the same semantics as described above. The values of U and V to be sent should be saved in a sufficiently large buffer bufSend or bufRecv. This function is called at the end of calculate_uv.

4. Perform the following modifications in the functions of Worksheet 1:

   - All spatial operations (loops over i and j) must be carried out only in the respective subdomains, with the properties of staggered grids and boundary strips being taken into account.

- Since all informative `printf` outputs will be executed by each process, the read-out feature is substantially limited. A remedy for this, especially for debugging, can be found in `parallel.c`, which provides some useful routines:
  `Programm_Message(...)`, `Programm_Sync(...)`, `Programm_Stop(...)`.
  Another approach is to restrict the output to one process.
- In `visual.c`:

  ```
  void output_uvp(double **U,double **V,double **P,int il,int ir,
       int jb,int jt,int omg_i,int omg_j,char *output_file)
  ```

  Each individual process writes an output file of its own. Hereby, only elements within the process' domain are to be plotted. Elements outside (in the strips, e.g.) shall not be written to the output file. Ensure, that the output filename for each processor is unique. Paraview allows for the visualisation of multiple input files. As the different domains are disjoint, our parallel visualisation requires for no special treatment: Just load all the output files written by the visualisation.
- In `read_parameter`:
  Enhance the function in such a way that the process partitioning arguments `iproc` and `jproc` can be additionally read out from the parameter file and passed to the main program.
- In `main`:
  While allocating the memory, the arrays must have dimensions given in (∗). The main algorithm is, as described in Section 4, adjusted respectively. As the last instruction in the main program (before `return 0`), insert `Programm_Stop(...)` from the file `parallel.h`, which results in all processes to be synchronized and the `MPI_Finalize()` command to be called.
- In `boundary_values` and `spec_boundary_val`:
  The corresponding boundary values must be set when the subdomain boundary coincides with that of the whole domain.
- In `calculate_fg` und `sor`:
  The formulas for boundary values must be applied only if the subdomain boundaries coincide with those of the entire domain. Moreover, the calculation of the residual in SOR must be modified as described in Section 4.
- In `calculate_dt`:
  For the adaptive time step control applied here, current values of maximal velocities over the entire computational domain are needed. As described in Section 4, a further communication step, similar to the one required for the residual calculation, has to be introduced.
- Test the parallel algorithm taking as an example the driven cavity with the following parameters:

  ```
  imax = 300  jmax = 300   xlength = 10  ylength = 10
  dt = 0.01   t_end = 1.0  tau = 0.5     dt_value = 2.0
  eps = 0.01  omg = 1.7    alpha = 0.5   itermax = 100
  GX = 0.0    GY = 0.0     Re = 10
  UI = 0.0    VI = 0.0     PI = 0.0
  iproc = 2   jproc = 3
  ```

7. Measuring Performance

If the grid is sufficiently fine and the number of processors is high enough, impressive computing speed increases may be achieved with a parallel program as compared to a serial program. This performance increase is usually expressed using the terms *speedup* and *efficiency*. These quantities are defined as

- speedup $S(p) := T(1)/T(p)$,
- parallel efficiency $E(p) := T(1)/(p \cdot T(p)) = S(p)/p$,

where p is the number of processors in use and $T(p)$ is the execution time of the parallel computation on p processors.

When measuring the runtimes on a busy network, though, neither the individual processors nor the computer network will provide their resources exclusively to a single computing task. Instead, other processes will compete for the available computing power and network bandwidth. Thus, for the set of parameters given above, i.e. a rather small problem, it's also possible to gain only a small speed-up or even a slow-down, because the computational load is low in contrast to the time taken up by communication. The advantages of parallelization will really start to show in large computational problems. You should strive to achieve at least a noticeable speedup that improves when you increase the problem size on each partition. Otherwise, the purpose is primarily to get acquainted to the concepts and principal approaches to parallel systems and libraries as well as to illustrate these by example problems.

Part III.
Lattice Boltzmann

8. Parallelisation of the the Lattice Boltzmann Method

For the parallelisation of the Lattice Boltzmann Method we are going to use the proposed domain decomposition approach and extend it to 3D. We start by introducing a third variable k_{proc} for the z axis analog to i_{proc} and j_{proc}. Each subdomain consists of an inner part and a boundary layer of cells. The boundary layer can consist of cells with imposed boundary conditions or inner cells lying on the boundary between processes.

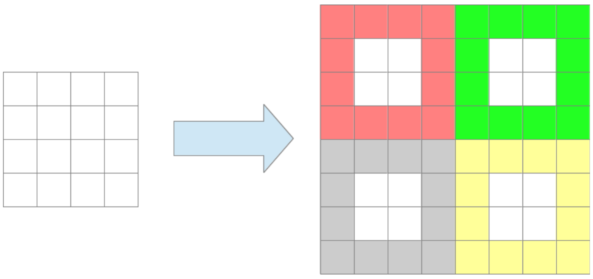

Figure 4: Overview of the data layout for a regular domain decomposition in 2D. The example shows how the initial cells are distributed among four processes. Each process holds a block of 2 by 2 cells and an additional boundary layer, representing cells with imposed boundary conditions or inner cells lying on the process boundary.

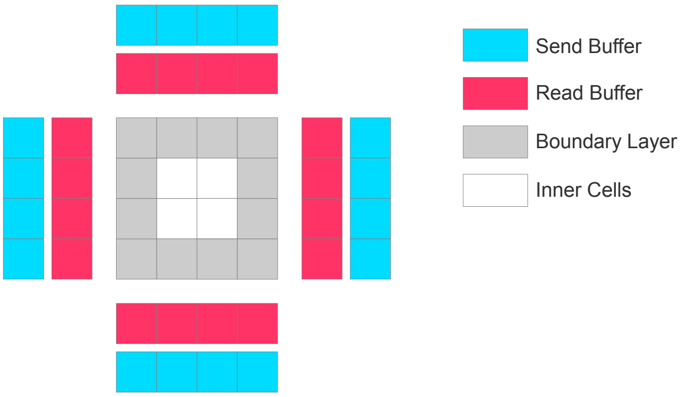

Figure 5: Buffers needed for a single process instance surrounded by eight neighbours. The process has two buffers (a read and write buffer) for each direction. For non existing neighbors the buffers should not be created. A buffer holds the data (distribution functions), which points to the corresponding neighbor.

For the convenient representation on the process boundaries we extend our flag field array with one additional flag - the PARALLEL_BOUNDARY flag, which denotes if a cell resides at the process boundary. Logically, a PARALLEL_BOUNDARY cell is an inner cell. Let's consider a 2D example with domain decomposition along the x and y axis . As in the serial case each process holds a domain plus an additional layer for boundary cells (Fig.4). If we follow the parallelisation strategy introduced for the Navier-Stokes equations we need to send all distribution functions in the cells (9 distribution functions per cell in 2D, 19 in 3D) lying on the process boundaries by the ghost-layer technique. This method has the drawback, that it sends a lot of unused data. If we consider the two main operations, streaming and collision, we know that:

- Only a subset of the distribution functions from a specific neighbor is needed to perform the streaming step correctly.
- Collision can be performed locally.

Thus, we only need to communicate a subset of those values. This subset is defined by the streaming operation. To reduce the memory overhead implied by the ghost layer technique, we introduce two different buffers:

- Send buffer – here we are going to push the values, which are going to be sent to a specific neighbor.
- Read buffer – used to store the received distribution values.

Figure 5 illustrates the 2D case of a process and its buffers needed for the communication with all 8 neighbors. It holds 4 buffers per axis (X axis - left and right , Y axis - top and bottom). To

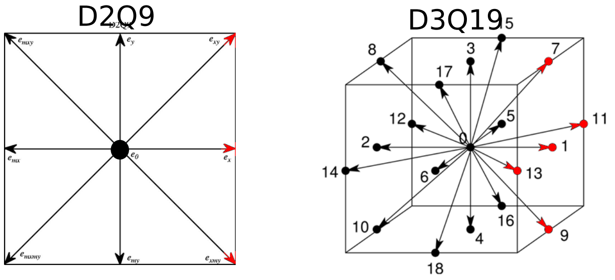

Figure 6: The example shows the needed distribution functions for the communication along the x axis for 2D and 3D. In 2D we can have at most three neighbors (one direct neighbor and two diagonal). We take only the distribution functions pointing towards those neighbors.

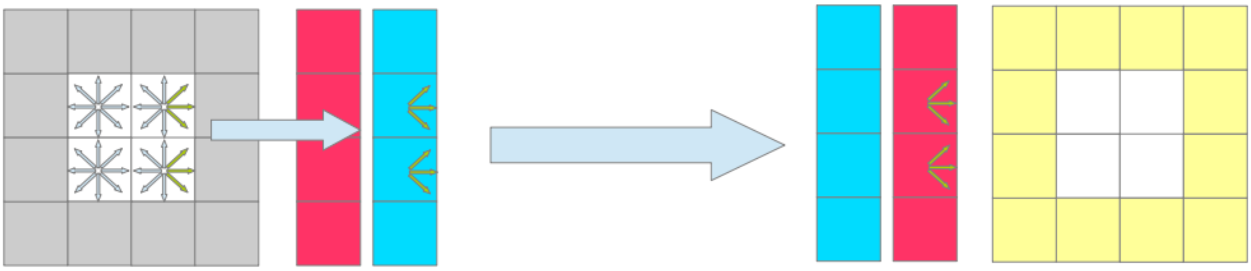

Figure 7: Data movement between two processes lying direct to each other on the X axis. We start by copying the distribution function into the send buffer (left arrow). We need only the functions pointing towards X. In the next step we send the data located in the buffer to yellow neighbor and receive data from it into the read buffer.

keep the approach as simple as possible and to avoid conflicts between diagonal neighbors during the communication, we will use buffers with exactly $\frac{xlength[0]}{i_{proc}} + 2$ entries for X axis , $\frac{xlength[1]}{j_{proc}} + 2$ for Y and $\frac{xlength[2]}{k_{proc}} + 2$ for Z. A simple entry of the buffer consists of 5 distribution functions pointing towards the specified neighbor for 3D (3 values in the 2D case - Fig.6).

The modified algorithm does a sequence of communication steps for different axis, so that the data can be propagated to all neighbors. For 2D we need two steps to propagate the data to all neighbors (in 3D 3 steps respectively). Each individual step is done on different coordinate axis and consists of the following sub-steps:

- Extraction step : Extract the needed distributions according to the communication direction and the flagField (only inner cells on process boundaries - faces) and copy them into the send buffer (Fig.7 - first arrow).
- Swap step : send the data located in the send buffer to the pointed neighbor and receive data from it in the read Buffer (Fig.7 - second arrow). That is the communication across processors.
- Injection step: copy the received data from the read buffer into the process boundary layer (Fig. 8).

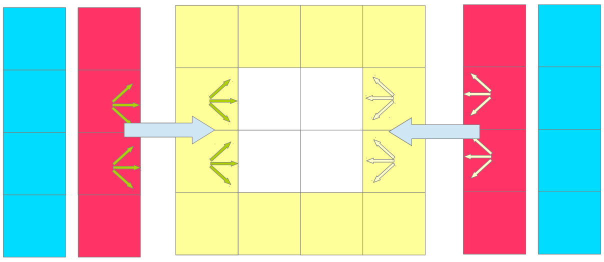

Figure 8: After receiving the data (7) we need to copy the distribution functions from the read buffer into the boundary layer.

Remarks

- The execution of three steps for each axis before the collision and streaming step in the time loop yields the correct propagation of distributions. As shown in Figure 9. After finishing all three steps for the X axis and the Y axis the distribution functions in the diagonals are transported correctly to the boundary layer of the diagonal neighbor.
- Step 1 and 2 put all distributions, that would be streamed from the inner cells of one sub-domain into a neighboring sub-domain, into a buffer. Instead of using a separate buffer for each grid cell (as it is done in the picture of step 2), it is more useful to allocate one big buffer for all boundary cells that need to be communicated to another process.
- In step 2 a send operation from process i to process j must be followed by receive operation on process i , receiving data from rank j.
- Step 3 puts these distributions into the parallel boundary cells of the other sub-domain. When the next streaming is carried out, the inner cells of each sub-domain can pull the right distributions from the parallel boundary cells.
- Once again: For the communication, it is useful to use a buffer to store all distributions from the parallel boundary cells that need to be sent/received to/from another sub-domain. This allows to only communicate the distributions that are needed by the other process (as depicted in the three images on the slides before; only communicate the pdfs that would enter the inner cells of the other process during streaming!).

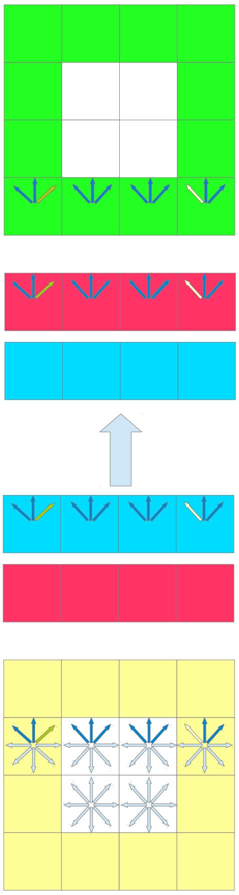

Figure 9: After finishing the communication for one axis (the X direction), we can continue with the second one. By applying the same steps as for the first direction we end up with correctly propagated data among all neighbors. The distribution functions pointing towards the diagonal neighbors were transported correctly (green and yellow).

9. The algorithm

We now look at the overall algorithm of our program:

```
int main (int argc, char *argv[]){
  double *collideField=NULL;
  double *streamField=NULL;
  int *flagField=NULL;
  int xlength;
  double tau;
  double velocityWall[3];
  int timesteps;
  int timestepsPerPlotting;
  int rank;
  int number_of_ranks;
  // send and read buffers for all possible directions :
  // [0:left,1:right,2:top,3:bottom,4:front,5:back]
  double *sendBuffer[6];
  double *readBuffer[6];

  initializeMPI(&rank,&number_of_ranks,argc,argv);

  readParameters(
    &xlength,&tau,&velocityWall,&iProc,&jProc,&kProc,timesteps,
    timestepsPerPlotting,argc, argv
  );

  // TODO: initialise pointers here with correct size for the domain decomposition!
  initialiseFields(collideField,streamField,flagField,xlength,rank,number_of_ranks);

  // allocate the buffers
  initialiseBuffers(sendBuffer,readBuffer,xlength);
  for(int t = 0; t < timesteps; t++){
    double *swap=NULL;
    do extraction , swap , injection for x (left to right)
    do extraction , swap , injection for x (right to left)
    do extraction , swap , injection for y (forth and back; back and forth)
    do extraction , swap , injection for z (down and up ; up and down)



    doStreaming(collideField,streamField,flagfield,xlength);
    swap = collideField;
    collideField = streamField;
    streamField = swap;

    doCollision(collideField,flagfield,&tau,xlength);
    treatBoundary(collideField,flagfield,velocityWall,xlength);

    if (t%timestepsPerPlotting==0){
      writeVtkOutput(collideField,flagfield,argv,t,xlength);
    }
  }
  finalizeMPI();
}
```

Task Summary

- Implement the LB-domain decompostion parallelization scheme for LID-Driven Cavity without arbitrary geometries.
- Modify the readParameters function to be able to read all three variables iProc, jProc and kProc, describing the number of processes per axis.
- Introduce a new flag PARALLEL_BOUNDARY to denote cells lying on the boundary between two processes.
- Adjust the size of the collideField, streamField and flagField in the main.c by considering the correct numbers of cells per process and the process boundary layer.
- Introduce sendBuffer and readBuffer for each process dependent on the neighbor-relationship.
- Execute the extraction, swap and injection step for every possible direction before the actual streaming and collision in each timestep .
- Extend the visualisation module for the parallel case: Use the provided feature of ParaView for appending of multiple files and extend the plotting function, so that it can plot n independent files, with n being the of processes used for the simulation! Use the process rank to insure the unique names of the files!

Tips for debugging:

- Start debugging by reducing the domain size to a small number of cells, so that you can reconstruct the result for each step of the algorithm and verify the correctness of the output.
- Beware of the communication near global domain boundaries! In such cases the corresponding processes not always have 8 for 2D or 26 for 3D neighbors. A communication with not existing neighbors can lead to deadlocks!
- Split the parallelisation into three phases: initialisation (determine size of sub-domains, allocation), sending phase and receiving phase! Check the correctness of each them independently!
- Always add the rank identifier to printf-debug statements to know where the output comes from!

Part IV. Appendix

A. Parallelization using MPI

In order to facilitate studying MPI, a short program is presented in this appendix demonstrating the transition from code to running a program. Here the program `communication.c` is given, which writes a process ID (`rank`) for each process into `stdout`, a singled-out process (e.g. with ID 0) sends the value `PI` to all others, and finally each process notifies the neighboring processes (`myrank-1, myrank+1`) of its ID. Here we also show how the both boundary cases can be handled with the aid of a special MPI constant `MPI_PROC_NULL`

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>

const double PI = 3.14;                         /* the value to be broadcasted */

int main(int argc, char* argv[]) {

   int i, myrank, nproc;
   int NeighborID_li, NeighborID_re;
   int recv_NeighborID_li, recv_NeighborID_re;
   double pi;
   MPI_Status status;


   /* initialisation */


   MPI_Init( &argc, &argv );                     /* execute n processes      */
   MPI_Comm_size( MPI_COMM_WORLD, &nproc );      /* asking for the number of processes  */
   MPI_Comm_rank( MPI_COMM_WORLD, &myrank );     /* asking for the local process id   */




   /* determine the neighbors */

   if ( 0 < myrank )
      NeighborID_li = myrank - 1;
   else
      NeighborID_li = MPI_PROC_NULL;              /* no neighbor */

   if ( (nproc - 1) > myrank )
      NeighborID_re = myrank + 1;
   else
      NeighborID_re = MPI_PROC_NULL;              /* no neighbor */


   /* broadcast of pi from process 0 to all  */

   if ( 0 == myrank )
      pi = PI;
   else
      pi = 0.0;
   printf("Proc %2d : Before the broadcast pi = %e\n", myrank, pi);

   MPI_Bcast( &pi, 1,MPI_DOUBLE, 0, MPI_COMM_WORLD );

   printf("Proc %2d : After the broadcast pi = %e\n", myrank, pi);


   /* send the ID to left/right neighbors (conventional)
      think over how the following send/receive procedure can be implemented with
      the help of a single MPI_Sendrecv command */

   /* ID of a non-existent neighbor has the value MPI_PROC_NULL  */

   MPI_Send( &myrank, 1, MPI_INT, NeighborID_li, 1, MPI_COMM_WORLD );
   MPI_Recv( &recv_NeighborID_re, 1, MPI_INT, NeighborID_re, 1,
             MPI_COMM_WORLD, &status );

   if ( MPI_PROC_NULL != NeighborID_re )
      printf("Proc %2d : ID right neighbor = %2d\n", myrank, recv_NeighborID_re);
   else
      printf("Proc %2d : ID right neighbor = MPI_PROC_NULL\n", myrank);

   MPI_Send( &myrank, 1, MPI_INT, NeighborID_re, 2, MPI_COMM_WORLD );
   MPI_Recv( &recv_NeighborID_li, 1, MPI_INT, NeighborID_li, 2,
             MPI_COMM_WORLD, &status );

   if ( MPI_PROC_NULL != NeighborID_li )
      printf("Proc %2d : ID left neighbor = %2d\n", myrank, recv_NeighborID_li);
   else
      printf("Proc %2d : ID left neighbor = MPI_PROC_NULL\n", myrank);


   /* Epilog */
   fflush(stdout);                      /* write out all output   */
   fflush(stderr);
   MPI_Barrier( MPI_COMM_WORLD );       /* synchronize all processes */
   MPI_Finalize();                      /* end the MPI session */

   return 0;
}
```

Compilation and linking are done with `mpicc`, which accepts the general C-Compiler/Linker options. The following call might be used:

```
mpicc -o communication Communication.c
```

Each MPI program will be executed with the help of the `mpirun` shell script, to which at least the desirable number of processes as well as the executable should be passed as parameters:

```
mpirun -np 4 Communication
```

Then four `Communication` processes will be started, which is quite sufficient during the development and debugging phase. If you are working on a computer with a MPI 2.0 implementation (like that at our chair, e.g.), the `mpirun` command has to be split up into three different operations: First of all, you've to start up the MPI environment using

```
mpdboot -n number_of_nodes -f machinefile --ncpus=localcpus &
```

or

`mpd --ncpus=localcpus &`.

Note, that both commands start a bunch of deamons now running in the background. Afterwards one can start the program using `mpirun`. Since the machinefile was passed to `mpdboot`, there's no need to specify it again in MPI 2.0. In the very end one should shut down MPI using the command `mpdallexit`.

The following MPI routines, types, and constants can be used for treating the Practical Course exercises:

`MPI_Init`, `MPI_Finalize`, `MPI_Comm_rank`, `MPI_Comm_size`, `MPI_Barrier`,
`MPI_Send`, `MPI_Recv`, `MPI_Sendrecv`, `MPI_Reduce`, `MPI_Bcast`,
`MPI_COMM_WORLD`, `MPI_INT`, `MPI_DOUBLE`, `MPI_MAX`, `MPI_SUM`,
`MPI_PROC_NULL`

All the methods appearing in this part of the Practical Course are defined in MPI Standard 1.1, which may serve as a documentation kit and is available under
`http://www.mpi-forum.org/docs/docs.html`.

For those who are interested in more technical details, we refer to the MPI implementation `mpich` freely available under `http://www-unix.mcs.anl.gov/mpi/`. After downloading and unpacking, you will see the installation guide, with the help of which the local `configure` parameter can be defined.