Lab exercises (1.1)

The goal of the computer lab is to become familiar with the message passing paradigm (MPI) and the strategy to obtain high performance for a specific application. After an introductory exercise, there are three different applications or case studies. In each application some additional aspects that are important for message passing or high performance computing are introduced.

In this document we will describe the steps that can be made to create an MPI program. We do not start from scratch but use an existing program, and transform it step by step in a 'realistic' message passing program.

1.1 First exercise

1.1.1 Introduction

First exercise is on numerical solution of Poisson's equation. Such a second order elliptic differential equation is typical for many areas in physics. Though the differential equation is rather simple, there are numerous ways to solve the system of linear equations that emerge after discretization of this differential equation. We consider a two dimensional computational domain which is simply the unit square. Dirichlet type boundary conditions are imposed at the boundary of the domain.

In order to keep the route to generalizations open, or to make a somewhat more realistic problem, we impose conditions other than Poisson's equation at some interior grid points. These conditions can be interpreted as auxiliary boundary conditions or as restrictions imposed by the physics of the problem. Hence at any grid point one out of the two possible relations is imposed. First there are points where Poisson equation holds, and secondly there are points where another equation or condition is imposed. Of course grid points where simple conditions, like a constant value, are imposed can easily be removed completely from the system of equations.

The reason for this generalization is to emphasize that in realistic problems the domain of interest in general is not a square, but a more general shape. Some direct solvers, such as those that use a Fast Fourier Transform are therefore not so easily applied. Iterative methods on the other hand are almost always applicable. It does not matter much for these methods whether the equation for a grid point, a finite volume or a finite element comes from Poisson's equation or from any other condition. The types of communication involved in this exercise are rather simple. In each iteration step any grid point needs the values of its neighboring grid points. These are needed in order to evaluate the discrete Poisson equation or the sparse matrix-vector multiplication. Apart from the local communication there are possibly global dot products that need to be evaluated in some algorithms, e.g., the Conjugate Gradient method. For the evaluation of these dot products each process in a message passing system can evaluate only its own local contribution. In order to obtain the global dot product all processes need to participate in a global reduction operation.

Other data that have to be communicated are for instance related to a stopping criterion. The global residue after a certain number of iterations may be needed in order to decide whether to stop or not. Of course one wants all processes to stop in the same phase of the calculation. This involves also rather simple global communication between the participating processes.

Actually in a message passing system it is not a good idea to let a process stop once its local residue is small enough. One problem is that the residue also depends on grid points that are not owned by a process. Grid points along the boundary of a process domain need points outside that domain in order to evaluate the contribution to the residue. The value of those outside grid points may still change in their subsequent iterations.

Another possible criterion to stop is to monitor the maximum change that occurs within a single iteration for each sub-domain separately. If this maximum change is smaller than a prescribed value the solution for the corresponding sub-domain has converged. However, other processes may not have reached convergence yet. Again this is not a good idea and it does not give results any faster. We have to wait until the last process has found a converged solution for its part of the domain. It is a waste of computing power to have some idle processes. Hence a simultaneous stop of all processes is advisable.

1.1.2 The Poisson problem and Parallelism

The central problem in this first exercise is the Poisson equation and the strategy to solve it on a parallel computer. We limit ourselves to the 2-dimensional situation

$$\nabla^2\phi(x,y) = S(x,y), \qquad 0 \le \mathrm{x,y} \le 1. \tag{1}$$

Here S(x, y) is called a source function. Boundary conditions at the edges of the unit square are required. In practical situations, it may occur that not in all points the Poisson equation is satisfied, but for instance that ϕ is fixed at a prescribed value at some interior points. Instead of the discretized version of eq.(1) at such points one has the equation

$$\phi(\mathrm{x,y}) = \phi_0(\mathrm{x,y}) \tag{2}$$

or in a discretized form

$$\phi_{\mathrm{i,j}} = (\phi_0)_{\mathrm{i,j}}, \tag{3}$$

at some specific grid points, with ϕ_0 some known function. We will not mention these alternative conditions anymore in the following equations, but implicitly assume that for certain (i, j)-points another condition holds instead of the discretized Poisson equation.

With straightforward finite differencing the partial differential equation (1) is turned into the system of equations

$$4\phi_{i,j} - \phi_{i-1,j} - \phi_{i,j-1} - \phi_{i,j+1} - \phi_{i+1,j} = h^2 S_{i,j}, \quad i,j = 1,\ldots,N-1, \tag{4}$$

where a grid spacing h = 1/N is defined such that $x_i = i \bullet h$ and $y_j = j \bullet h$. The $(N-1)^2$ unknowns can be solved from these $(N-1)^2$ equations. Note that the border points $\phi_{i,j}$ with either i or j equal to 0 or N are not unknowns. They must be fixed, or at least be expressed in terms of values $\phi(i,j)$ at interior points. Source terms in eq.(1) do not cause a problem. They enter through a direct discretization of S into the right hand side of eq.(4).

It is outside the scope of the present exercise to investigate the different systems of equations that all emerge from the various discretization strategies to solve eq.(1).

There are several strategies to solve systems of linear equations like eq.(4). In this exercise we will focus on some simple iterative techniques, as well as on their parallel implementation, where the domain decomposition technique enters in a natural way.

The basic idea in many iterative solvers is to write eq.(4) in a solved form. In the $(n+1)^{th}$ iteration a (preliminary) value at grid point (i,j) is obtained

$$\tilde{\phi}_{i,j}^{(n+1)} = \frac{\phi_{i-1,j}^{(n)} + \phi_{i,j-1}^{(n)} + \phi_{i,j+1}^{(n)} + \phi_{i+1,j}^{(n)}}{4} + \frac{h^2}{4} S_{i,j} \tag{5}$$

This preliminary value $\tilde{\phi}$ together with the previous value can be combined into a new value.

$$\phi_{i,j}^{(n+1)} = \omega\tilde{\phi}_{i,j}^{(n+1)} + (1-\omega)\phi_{i,j}^{(n)} \tag{6}$$

When $\omega = 1$ this is a called the Jacobi algorithm if the iteration counter is increased from n to $n+1$ only after preliminary values for all grid points are calculated. It is called a Gauss-Seidel algorithm (also with $\omega = 1$ if the iteration counter is increased each time when a preliminary value at a grid point is obtained. With the appropriate value of ω a Gauss-Seidel algorithm is also called the SOR (Successive Over Relaxation).

For the Jacobi algorithm the sequence at which grid points are visited is irrelevant. However, for Gauss-Seidel this sequence is important.

Suppose, for instance, that one increases the iteration counter each time 1 point has been updated. Furthermore assume that one visits points in such a sequence that subsequent points are always neighbors. This is also a Gauss-Seidel scheme. Such an algorithm is essentially sequential and a parallelization strategy is doomed to fail, but it can be shown that this Gauss-Seidel iteration converges faster than Jacobi iteration.

Fortunately if one simply loops over the points in a lattice in the 'normal way', row after row and from left to right, subsequent points are not always neighbors. Sometimes one starts with a new row. This makes a parallelization techniques feasible. In literature this is called the 'wavefront' or ‘hyperplane’ approach. However, it is also possible to visit the points in a lattice such that one does not need the updated value of a grid point for a long time. This is achieved by splitting the grid into 'red' and 'black' points like in a checker board. First visit all grid points with the same color, followed by all the grid points of the other color. Now one has to update the values of grid points only twice during one sweep over all the points. When it comes to parallelization this red-black Gauss-Seidel is much easier to implement than the standard Gauss-Seidel, and only slightly more difficult than Jacobi.

We can thus distinguish various parallel iterative schemes, based on slightly different algorithms. All updates within one iteration can be performed in parallel. With N grid points in either direction, red-black Gauss-Seidel updates $N^2/2$ points in each iteration. All these points can in principle be updated in parallel.

Apart from generating computational schemes that are easily implemented on a parallel computer, it is of course more important to investigate the performance of such schemes or algorithms. Therefore we will not only restrict ourselves to these Gauss-Seidel type of iterations, but discuss also the Conjugate Gradient Method as a more generic way to solve (sparse) linear systems of equations. Here of course emphasis will also be on the parallelization and on the communication that is required.

It will be clear by now that there exists a variety of iterative methods, and that each method can be implemented in a large variety of ways.

The outline of any parallel iterative algorithm is more or less as follows:

1. **Initialize.** Within each domain an initial try $\phi^{(0)}(x, y)$ is either set and discretized or data are read from file.

2. **Communicate.** Information of boundaries between adjacent domains is transmitted.

3. **Compute.** Each domain does one or more iterations (updates) for all its interior points. Points may be updated if all the information that is needed to update it, is available.

4. **Check.** Test whether the result has converged sufficiently. If not go on with step 2.

5. **Finalize.** Save the (part of the) solution found.

A parallel program can be obtained by implementing one of the schemes, such as red-black Gauss-Seidel iteration. However, there are numerous questions or problems that are worth to investigate. For instance, is it worth to make more than 1 iteration in each domain between two communication steps? The convergence of the algorithm will be slower if measured in the number of iteration steps, but if measured in real time it may or may not be advantageous. How much data should one transfer in a communication step? Just the data along the border, or maybe more layers at once? What is the best way to partition the grid points into domains? Should these domains be as close to a square as possible, horizontal strips or vertical strips? How does the performance scale with problem size, or with increasing number of processes?

1.1.3 Description of the sequential code SEQ_Poisson.c

As a starting point we consider the code SEQ_Poisson.c that solves the Poisson equation on a rectangular domain with the red-black Gauss-Seidel scheme. This program can be downloaded from Brightspace. All the routines in this program are put together in one file.

After a few routines that are provided for timing purposes, and to facilitate debugging, the first routine of interest to us is Setup_Grid.

1. Setup_Grid. In this routine all the input is read. These are the number of grid points in the x and y direction. For stopping there is a precision goal and if this goal is not met, for instance because of slow convergence, there is still a maximum allowed number of iterations. Memory is allocated for the necessary data structures. Furthermore the initialization is done here. In the example input input.dat there 3 points indicated in the domain that are kept at a prescribed value that is also read from input.

2. Do_Step. In this routine the code to update all red or black points once is provided. While new values are being calculated the maximum change is being monitored. This can be used as a stopping criterion.

3. Solve. In this routine there is the iteration loop. It is executed until a stopping criterion is satisfied. Within each iteration there are calls to Do_Step. The red and black points are distinguished by their parity, which may be either 0 or 1. The parity of a grid point is denned as the sum of its indices modulo two. A point x_i, y_j thus has even parity if $i + j$ is equal to an even number.

4. Write_Grid. In this routine the resulting field after the last iteration is written to a file with the name output.dat. Though this has nothing to do with high performance, it is important that a realistic application produces results.

5. Clean_Up. This routine does nothing but freeing the claimed memory when it is no longer needed.

6. main. The main program calls the various components discussed above. Note that a timer start_timer is started and that the elapsed time is printed with print_timer at a specific location within the program. The timer routines (start, stop, print and resume) can be placed in almost any sequence and almost anywhere in the program to get an impression of the time the program spends in a specific phase.

1.1.4 Building a parallel program using MPI

The sequential version of the code that solves a Poisson-like problem is provided as the file SEQ_Poisson.c. In order to generate a parallel version from it that uses the MPI library, modifications have to be made and code has to be added at several places. You will already be familiar with some of these steps, since they are similar to the exercises in the introduction to MPI. In this part of the lab you will build a working parallel code step by step. Meanwhile you will become familiar with some additional functionality that is offered by the MPI library.

1.1.4.1 Step 1

Execute clones of the original code on a number of processes. You have to add calls to MPI_Init and MPI_Finalize in the main program. This is the minimum amount of additional code you need in order to use MPI. Rename the program as MPI_Poisson.c. Compile the program and run it on processors using the srun command. How do you know that you indeed executed the program twice?

A first problem you might encounter on an arbitrary parallel computer, is related to I/O. The sequential program reads data from a file and generates an output file and probably also writes some output to your screen. It is not guaranteed that each processor in a parallel computer can perform all these operations.

The easiest way to turn the sequential program into a parallel one is to change nothing, and let each process perform its I/O operations exactly as in the sequential program. In a 'loosely coupled parallel system' this approach may not work since various processes may have different file systems from where they read files. Writing results to file gives also problems, since various processes may simultaneously write or try to write, and results written by one process are easily overwritten immediately by another.

The alternative that always works is to read data by only one process and broadcast it to the others. Hence one process is designated to do all the I/O.

1.1.4.2 Step 2

Make sure that you can see which process is responsible for each line of output shown on your terminal screen. Therefore you have to know the rank of the process that prints something to standard output. You need to use MPI_Comm_rank to determine the rank of a process, and print at least this rank each time you sent something to standard output.

Declare a global variable with the name proc_rank and change the print command

```
printf("Number of iterations %i\n", count);
```

at the end of the Solve routine into something of the form

```
printf("(%i) .............\n", proc_rank, ........... );
```

that also prints the rank of the process.

1.1.4.3 Step 3

In the previous exercise you could not see how much time each process used. Replace the four timing routines in the code with the following text.

```
void start_timer()
{
  if (!timer_on)
  {
      MPI_Barrier(MPI_COMM_WORLD);
      ticks = clock();
      wtime = MPI_Wtime();
      timer_on = 1;
  }
}
```

```
void resume_timer()
{
  if (!timer_on)
  {
          ticks = clock()  -  ticks;
          wtime = MPI_Wtime() - wtime;
          timer_on = 1;
  }
}

void  stop_timer()

{
  if (timer_on)
  {
          ticks = clock()  -  ticks;
          wtime = MPI_Wtime() - wtime;
          timer_on = 0;
  }
}

void print_timer()
{
  if (timer_on)
  {
          stop_timer();
          printf("(%i) Elapsed Wtime %14.6f s (%5.1f%% CPU)\n",
             proc_rank, wtime, 100.0 * ticks * (1.0 / CLOCKS_PER_SEC) / wtime);
          resume_timer();
  }
  else
          printf("(%i) Elapsed Wtime %14.6f s (%5.1f%% CPU)\n",
             proc_rank, wtime, 100.0 * ticks * (1.0 / CLOCKS_PER_SEC) /  wtime);
}
```

This text can be found on Brightspace as mptimers.c. For each process you can now see how much time it has spent. Do not forget to define wtime as a global variable.

```
double wtime;                    /* wallclock time */
```

Run the program again with processes and verify whether the timing routines work correctly. Apart from MPI_Wtime the only MPI function that occurs in these routines is MPI_Barrier. It assures that all processes start simultaneously with their timing activities.

1.1.4.4 Step 4

Verify that 2 clones of the program do indeed produce identical results. Therefore inspect the results written to your terminal screen as well as to file. If everything works correct you should see on your terminal screen that both processes performed the same number of iterations. This is no guarantee that they generated the same result as well. Therefore you have to inspect the results written to the output file. In the file output.dat the values at the various grid points are written. You may notice that each process writes to this file simultaneously. Hence it may be a surprise to find out what is actually written to it. In order to get rid of this ambiguity we change the program such that different processes write their calculated data to different output files. You can achieve this by creating a filename that is a rank-dependent character string, for instance output0.dat for the output of process with rank 0 and output1.dat for the output of the process with rank 1.

Change in the routine Write_Grid the lines where the filename is determined and the file is opened for writing

```
if ((f = fopen("output.dat", "w")) == NULL)  Debug("Write_Grid fopen failed", 1);
```

into

```
char filename[40];

sprintf(filename, "output%i.dat", proc_rank);

if ((f = fopen(filename, "w")) == NULL)
   Debug("Write_Grid fopen failed", 1);
```

Make sure that the declaration of the new local variable filename appears at the right place in the code. The integer variable proc_rank is globally known.

Check with the Unix command **diff** whether the contents of output0.dat and output1.dat is indeed identical.

1.1.4.5 Step 5

Change the program such that only the process with rank 0 reads data from an input file, and subsequently broadcasts this data to all processes. You will have to modify Setup_Grid and use the collective communication MPI_Bcast for this purpose. In the beginning of Setup_Grid you will see the text that forms the body of the if below.

```
if (.......)         /* only process 0 may execute this if    */
{
  f = fopen("input.dat", "r");
  if (f == NULL)
      Debug("Error opening input.dat", 1);
  fscanf(f, "nx %i\n", &gridsize[X_DIR]);
  fscanf(f, "ny %i\n", &gridsize[Y_DIR]);
  fscanf(f, "precision goal %lf\n", &precision_goal);
  fscanf(f, "max iterations %i\n", &max_iter);
}
MPI_Bcast(....);  /*  broadcast the  array gridsize  in one  call */
MPI_Bcast(....);  /* broadcast precision_goal                     */
MPI_Bcast(....);  /* broadcast max_iter                           */
```

Here it is important that only the process with rank 0 reads data from the input file, but that all processes call MPI_Bcast. If you only modified the code to include this if-statement and the 3 lines with MPI_Bcast you will get an error if you try to run the program. The reason is that the end of Setup_Grid some more data is read from this input file that has to be modified as well.

```
do
{
  if (..........)       /* only process 0 may scan next line of input */
      s = fscanf(f, "source %lf %lf %lf\n", &source_x, &source_y, &source_val);

  MPI_Bcast(&s, ...... ); /* The return value of this scan is broadcast even though
                                          it is no input data */

  if (s == 3)
  {
   MPI_Bcast(.........);           /* broadcast source_x        */
   MPI_Bcast(.........);           /* broadcast source_y        */
   MPI_Bcast(.........);           /* broadcast source_val */
```

```
    x = gridsize[X_DIR] *  source_x;
    y  =  gridsize[Y_DIR]  *   source_y;
    x +=  1;
    y  += 1;
    phi[x][y] = source_val;
    source[x][y] = 1;
  }
}
while (s == 3);

if  (..........)  fclose(f);    /* only process 0 may close the file */
```

This completes the distribution of identical information from one input file over all processes with MPI_Bcast.

1.1.4.6 Step 6

Up to now each process performs exactly the same calculations as the others. The first step to change this is to make every process recognize its own specific role. Therefore you are going to develop the routine Setup_Proc_Grid. This routine is new and is absent in the sequential code. The call to MPI_Comm_rank in the main program is now changed in the line

```
Setup_Proc_Grid(argc, argv);
```

This is an important phase in the parallelization process. All the processes that are active, can be imagined to form a 2-D process grid. The sizes of this grid are provided at startup time. They are the arguments of the run-script run that is provided to facilitate the running of the program on different process grids.

We need to define some variables that are specific for each process. These variables are all global.

```
/* process specific variables */
int proc_rank;                /* rank of  current process   */
int proc_coord[2];            /* coordinates of current process in processgrid */
int proc_top, proc_right, proc_bottom, proc_left; /* ranks of neigboring procs */
```

Furthermore there are the following global variables that are the same for each process.

```
int  P;                       /* total number of processes */
int P_grid[2];                /* process grid dimensions */
MPI_Comm grid_comm;           /* grid COMMUNICATOR           */
MPI_Status  status;
```

The call to Setup_Proc_Grid should be placed immediately after MPI_Init in the main program.

```
void Setup_Proc_Grid(int argc, char **argv)
{
  int wrap_around[2];
  int reorder;

  Debug("My_MPI_Init", 0);

  /* Retrieve the number of processes */
  MPI_Comm_size(.........);   /* find out how many processes there are      */
```

```
/* Calculate the number of processes per column and per row for the grid */
  if (argc > 2)
  {
     P_grid[X_DIR] = atoi(argv[1]);
     P_grid[Y_DIR] = atoi(argv[2]);
     if (P_grid[X_DIR] * P_grid[Y_DIR] != P)
          Debug("ERROR Proces grid dimensions do not match with P ", 1);
  }
  else
     Debug("ERROR Wrong parameter input", 1);

  /* Create process topology (2D grid) */
  wrap_around[X_DIR] = 0;
  wrap_around[Y_DIR] = 0;    /* do not connect first and last process      */
  reorder = 1;  /* reorder process ranks  */

  MPI_Cart_create(.............); /* Creates a new communicator grid_comm  */

  /* Retrieve new rank and cartesian coordinates of this process */
  MPI_Comm_rank(..........); /* Rank of process in new communicator        */
  MPI_Cart_coords(.......);   /* Coordinates of process in new communicator*/

  printf("(%i) (x,y)=(%i,%i)\n", proc_rank, proc_coord[X_DIR], proc_coord[Y_DIR]);

  /* calculate ranks of neighboring processes */
  MPI_Cart_shift(grid_comm, Y_DIR,....);
                              /*  rank of processes proc_top and proc_bottom  */
  MPI_Cart_shift(......);   /*  rank of processes proc_left and  proc_right */

  if (DEBUG)
       printf("(%i) top %i,  right  %i,  bottom  %i,  left  %i\n", proc_rank, proc_top,
                      proc_right, proc_bottom, proc_left);
}
```

When the MPI job starts up a certain amount of processes, each of them has a unique rank. MPI has some elegant provisions to arrange the processes in a virtual process grid. This process grid has little to do with the way processors in a parallel system are mutually connected. With MPI_Cart_create you can create such a (cartesian) grid of processes. Each process now 'knows' where it is in the grid, and what the ranks of its neighbors are. You can use MPI_Cart_shift to find the rank of the neighbors, and MPI_Cart_coords to find the coordinates of the process in the process grid. If there is no neighbor in a certain direction, because it points to a location outside the process grid, the rank of that 'neighbor' is set to MPI_PROC_NULL. This is very useful because any attempt to communicate with a non-existing process immediately returns. So you do not have to treat the boundaries of the process grid as special cases.

It is advised to use these MPI functions, even though it is not very difficult to arrange $n_x \times n_y$ processes with ranks between 0 and $n_x \times n_y - 1$ in an $n_x \times n_y$ grid, and find out for each process what the ranks of its 4 neighbors are.

The goal of Setup_Proc_Grid is thus to generate the 4 integers that denote the ranks of the 4 neighbors, and the 2 integers that denote the coordinates of each process in the virtual process grid. This function takes care of the administration, and it is executed only once and simultaneously by all processes. Without any further changes each process still does all the work, so we do not have a parallel program yet. The print statements in this routine make it easy to verify whether each process indeed knows and the ranks of its neighbors and its own position in the grid.

From now on only the communicator grid_comm is used. This implies that MPI_COMM_WORLD should be replaced by grid_comm in any MPI call executed after MPI_Cart_create.

1.1.4.7 Step 7

The next step is to give each process its own work. Therefore it needs to obtain data. This is something that belongs to the responsibility of the Setup_Grid routine. From input the total problem size is read. This is known to all processes as discussed before. Depending on the problem size and on its position in the process grid (obtained from Setup_Proc_Grid) each process determines how much data it has to work on. This amount can be different from process to process, since only in special cases the amount of data can be evenly divided amongst the processes. Moreover, it is important that each process knows the location of the data for which it is responsible in the total domain. So if a certain point has to be set to some value, only the process that owns the region of that point should set the value of the corresponding grid point.

In Setup_Grid each process finds out what region of the domain it is going to work on. The right amount of memory space is allocated, and the initialization of the field, which is necessary for any iterative solver (the value of the zeroth iteration), is performed.

In the sequential code the single process gets 1 extra layer of grid points around the complete domain. These are called ghost points. Hence also in the sequential code the size of the arrays increases with 2 in each direction.

```
/* Calculate dimensions of grid */
dim[X_DIR] = gridsize[X_DIR] + 2;
dim[Y_DIR] = gridsize[Y_DIR] + 2;
```

In the parallel version this additional layer of points is also added. Now these points are not always ghost points, but they can also be owned by a neighboring process. Each process calculates what the index of its 'first' point in any dimension would be in the global grid. This is called its offset.

```
/* Calculate top left corner coordinates of local grid */
offset[X_DIR] = gridsize[X_DIR] * proc_coord[X_DIR] / P_grid[X_DIR];
offset[Y_DIR] = gridsize[Y_DIR] * proc_coord[Y_DIR] / P_grid[Y_DIR];
upper_offset[X_DIR] = gridsize[X_DIR] * (proc_coord[X_DIR] + 1) / P_grid[X_DIR];
upper_offset[Y_DIR] = gridsize[Y_DIR] * (proc_coord[Y_DIR] + 1) / P_grid[Y_DIR];

/* Calculate dimensions of local grid */
dim[Y_DIR] = upper_offset[Y_DIR] - offset[Y_DIR];
dim[X_DIR] = upper_offset[X_DIR] - offset[X_DIR];

/* Add space for rows/columns of neighboring grid */
dim[Y_DIR] += 2;
dim[X_DIR] += 2;
```

Note that the array offset is a global variable, whereas upper_offset is needed only local in Setup_Grid.

The next thing to take care for is the treatment of fixed points. With only one process and one domain it is sufficient in Setup_Grid to have simply the 2 lines

```
phi[x][y] = source_val;
source[x][y] = 1;
```

where x and y are indices of the fixed points. source is an array with flags that are set to 1 to indicate that the point is kept fixed. In the parallel program, at least as implemented in the example code, any fixed point is broadcast to any process. Hence each process actually has to check whether each fixed point lies in its domain or not. Therefore the previous 2 lines of code have to be replaced by the following piece of code.

```
x = x - offset[X_DIR];
y = y - offset[Y_DIR];
if ( x>0 && x < dim[X_DIR] -1 && y>0 && y < dim[Y_DIR] -1)
{                       /* indices in domain of this process */
   phi[x][y] = source_val;
   source[x][y] = 1;
}
```

After Setup_Grid is adjusted it is possible to perform independent runs. Each process gets its own problem size and its own data, and is able to solve its Poisson problem, independent of the other processes. Run the example code with 2, 3 or 4 processes. Verify that the processes are indeed doing different things. With for instance 3 processes you should see that 1 or 2 processes do not do any iteration. Do you understand why?

1.1.4.8 Step 8

In the sequential program there are the so called ghost points around the active grid points. For instance, these ghost points are fixed if one has Dirichlet boundary conditions. In the parallel implementation, the ghost points for one process may be active points for another process. So if one wants to solve the global problem, the neighboring processes have to exchange data so now and then. In a new routine Exchange_Borders this data exchange in the 4 possible directions is performed. With the help of MPI_Sendrecv it is possible to let all the processes first perform all the sends and receives that involves 'traffic' to the left. Followed by traffic to the right, top and bottom, respectively.

Each process should know in each of these 4 phases what the addresses are of the data to be sent, and what the addresses are of the ghost points that are to be received.

Since the data structure is regular, and in the data exchange complete borders between neighboring processes are transmitted, it is practical to create special data types for these border elements with MPI_Type_vector and MPI_Commit. With the code of Setup_MPI_Datatypes this new data types are denned as follows:

```
void Setup_MPI_Datatypes()
{
      Debug("Setup_MPI_Datatypes", 0);

      /* Datatype for vertical data exchange (Y_DIR) */
      MPI_Type_vector(dim[X_DIR] - 2, 1, dim[Y_DIR],
                                    MPI_DOUBLE, &border_type[Y_DIR]);
      MPI_Type_commit(&border_type[Y_DIR]);

      /* Datatype for horizontal data exchange (X_DIR) */
      MPI_Type_vector(dim[Y_DIR] - 2, 1, 1,
                                 MPI_DOUBLE, &border_type[X_DIR]);

      MPI_Type_commit(&border_type[X_DIR]);
}
```

A call to Setup_MPI_Dataypes has to be placed in the main program, and the global array border_type with 2 elements of type MPI_Datatype - one for exchange in the x-direction and one for exchange in the y-direction - has to be declared as well.

Of these new 'border' data types only one element is sent/received in a single MPI_Sendrecv call. Just as elementary data types, these new data types have an address, which is simply the address of its first entry.

The exchange of data along the 4 borders can be performed before or after each iteration.

Since the sequential version uses a red-black ordering scheme it is necessary to perform data exchange after each iteration of red as well as black grid points.

The skeleton code of the Exchange_Borders routine is:

```
void Exchange_Borders()
{
        Debug("Exchange_Borders", 0);

        MPI_Sendrecv(&phi[1][.... ], 1, border_type[Y_DIR], proc_top, 0,
                     &phi[1][.....], 1, border_type[Y_DIR], proc_bottom, 0,
                     grid_comm,  &status); /*  all  traffic in  direction  "top"     */

        MPI_Sendrecv( ..........
                      ----------
                     ...........    ); /* all traffic in direction "bottom" */

        MPI_Sendrecv( ..........
                      ----------
                     ...........    ); /* all traffic in direction "left" */
        MPI_Sendrecv( ..........
                      ----------
                     ...........    ); /* all traffic in direction "right" */

}
```

Make sure you provide the correct starting addresses in each MPI_Sendrecv call. With the Exchange_Borders routine implemented, you have a version of the Poisson solver where the various processes cooperate in solving the global problem. However, it is unlikely that the program will run, since the processes do not communicate amongst each other when to stop. Run the example code with processes. When only one process finishes you will have to kill the other process manually (when you run the program interactively type Ctrl-C to kill the process). However, on the Delftblue, you submit the program to a batch queue, you can better set a time limit for the program using the option –time, e.g., --time 00:00:60 (60seconds).

1.1.4.9 Step 9

Next one has to assure that a sufficient accurate solution of the global problem is found. Now each process performs iterations and exchanges data until it obtains an accuracy of its own (local) solution that is sufficient. Then it will decide to stop. However, the other processes may not be aware of that fact, and may still want to exchange data and go on with more iterations. That was the reason why the program did not finish normally in the previous exercise.

A possible solution to this problem is that a process may only stop, if a global convergence criterion is met. This happens to be also the convergence criterion in the sequential program. The additional work in the parallel version is the calculation (possibly after a certain number of iterations) of the global error from the local errors of all processes. If each process knows the value of the global error, they can all decide simultaneously to stop if this error is sufficiently small.

In Solve you thus have to define besides the local variable delta also another variable global_delta.

With the help of MPI_Allreduce it is possible and rather simple to perform a so called collective reduction operation. Therefore complete the following line of code in Solve. It is inserted after delta has been given a value.

```
MPI_Allreduce(&delta, ....  ,  ....  ,  MPI_DOUBLE, ....  ,  grid_comm);
```

Furthermore change delta into global_delta in Solve on all places where you think it is necessary.

Every process has an error, but wants to know the maximum of all the errors of all the processes. Other norms used as a convergence criterion can be implemented with the same ease.

So in Solve you have to add/modify code to assure that all processes leave the while-loop if a global convergence criterion is reached. Check whether the number of iterations that is required for convergence is still the same.

1.1.4.10 Step 10

Now everything seems to work fine, but the following two points may need some more attention.

- Several output files are generated: one by each process. The routine Write_Grid that collects output hardly changed compared to the sequential version. The indices of the points written to file are the local indices. They have to be changed to global indices. Therefore one has to add offset[X_DIR] and offset[Y_DIR] to the x and y index, resp., that is written to output. Ghost points are not written to the output files, so each point is written only to one file.

  Next there is the dilemma whether to change the program such that only one global output file is generated, or leave it as it is. We choose for the easy approach. But if you want to generate one output file that contains all the data calculated by the various processes you can have a look at the file writegrid.c.

- Results that are now generated need not be identical to the results of the sequential program. There may be small differences. Check this by looking at the calculated function value, obtained with the current code, at a particular point, and compare it with the value obtained with the sequential program. The reason is that one process may start with a 'red' grid point at the local [1, 1] position, whereas another process may have a 'black' grid point at its local [1, 1] position. In a parallel program these points are updated in the same phase, whereas in the sequential program these points are updated in a different phase. The solution is to let each process keep track of the global parity of its grid points. Modify the conditional line

  ```
  if ((x+y) % 2 == parity && source[x][y] != 1)
  ```

  in Do_Step to solve this.