IN3200/IN4200: MPI programming by the example of Jacobi computations

Parts of the content are from the textbook: Hager & Wellein, *Introduction to High Performance Computing for Scientists and Engineers*

Goal

- Demonstrate MPI programming through parallelizing Jacobi computations
    - 1D, 2D, 3D
    - Domain decomposition
    - Use of basic MPI commands

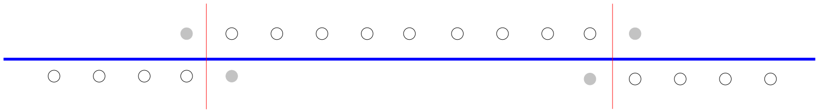
Let's start with 1D serial Jacobi computation

phi and phi_new: 1D arrays of length imax

```
// ...
/* initialization (example) */
for (i=1; i<imax-1; i++) {
  x = i*dx;                // coordinate of the grid point
  phi[i] = sin(M_PI*x);    // suppose we use the sin function
}

maxdelta = 1.0; eps = 1.0e-14;

while (maxdelta > eps) {
  maxdelta = 0.;

  for (i=1; i<imax-1; i++) {
    phi_new[i] = (phi[i-1]+phi[i+1])*0.5;
    maxdelta = max(maxdelta, abs(phi_new[i]-phi[i]));
  }

  /* pointer swapping */
  temp_ptr = phi_new; phi_new = phi; phi = temp_ptr;
}
```

Domain decomposition as the foundation of parallelization

- Note: total number of grid points is `imax`, but the left and right boundary points do not need computation
- Divide the `imax-2` interior points evenly among P processes
- For programming convenience, each process gets in addition two *ghost points* (also called *halo points*)



Decomposition and array allocation on each process

`P`: total number of processes
`my_rank`: unique rank of a process

```
my_start = my_rank*(imax-2)/P;
my_stop = (my_rank+1)*(imax-2)/P;
my_imax = my_stop-my_start+2;  // including the two ghost points

my_phi_new = (double*)malloc(my_imax*sizeof(double));
my_phi = (double*)malloc(my_imax*sizeof(double));
```

The P processes are logically lined up from "left" to "right".

The above code should work for any value of $1 \leq P \leq$`imax-2`, and the work division is as even as possible. (Can you verify?)

Initializing my_phi

Each process initializes its my_phi array:

```
for (i=1; i<my_imax-1; i++) {
  x = (my_start+i)*dx;     // coordinate of the grid point
  my_phi[i] = sin(M_PI*x); // same formula as in the serial computation
}
```

Note: each process initializes for all its subdomain interior points

Computational work of each process per iteration

```
my_maxdelta = 0.;

for (i=1; i<my_imax-1; i++) {
  my_phi_new[i] = (my_phi[i-1]+my_phi[i+1])*0.5;
  my_maxdelta = max(my_maxdelta, abs(my_phi_new[i]-my_phi[i]));
}
```

The same as the serial computation, except that the computation is now within the subdomain of each process

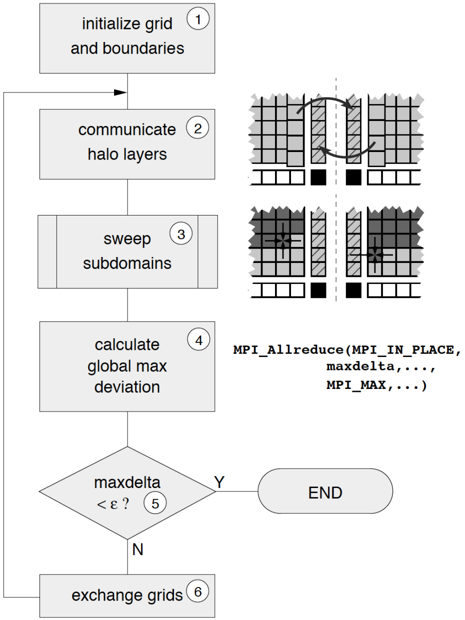
What else should each process do per iteration?

- Before computation: Must get the ghost point values (by communication)
    - Receive one value from the left neighbor, into `my_phi[0]`
    - Send `my_phi[1]` to the left neighbor
    - Receive one value from the right neighbor, into `my_phi[my_imax-1]`
    - Send `my_phi[my_imax-2]` to the right neighbor
- After computation: Find out the global maximum among all the subdomain values of `my_maxdelta`

Entire work per process

```
while (maxdelta > eps) {
  if (my_rank>0)
    MPI_Irecv(&my_phi[0],1,MPI_DOUBLE,myrank-1,0,MPI_COMM_WORLD,&req_left);
  if (my_rank<P-1)
    MPI_Irecv(&my_phi[my_imax-1],1,MPI_DOUBLE,myrank+1,0,MPI_COMM_WORLD
              &req_right);
  if (my_rank>0)
    MPI_Send(&my_phi[1],1,MPI_DOUBLE,myrank-1,0,MPI_COMM_WORLD);
  if (my_rank<P-1)
    MPI_Send(&my_phi[my_imax-2],1,MPI_DOUBLE,myrank+1,0,MPI_COMM_WORLD);
  if (my_rank>0)
    MPI_Wait(&req_left,&status_left);
  if (my_rank<P-1)
    MPI_Wait(&req_right,&status_right);

  my_maxdelta = 0.;
  for (i=1; i<my_imax-1; i++) {
    my_phi_new[i] = (my_phi[i-1]+my_phi[i+1])*0.5;
    my_maxdelta = max(my_maxdelta, abs(my_phi_new[i]-my_phi[i]));
  }

  MPI_Allreduce(&my_maxdelta,&maxdelta,1,MPI_DOUBLE,MPI_MAX,MPI_COMM_WORLD);

  /* pointer swapping */
  temp_ptr = my_phi_new; my_phi_new = my_phi; my_phi = temp_ptr;
}
```

Flow chart of parallel Jacobi computation


initialize grid and boundaries 1
communicate halo layers 2
sweep subdomains 3
calculate global max deviation 4
maxdelta $< \varepsilon$? 5
Y
N
END
exchange grids 6

```
MPI_Allreduce(MPI_IN_PLACE,
              maxdelta,...,
              MPI_MAX,...)
```

Serial 2D Jacobi computation

Now, phi and phi_new: 2D arrays of dimension jmax×imax

```
/* initialization (example) */
for (j=1; j<jmax-1; j++) {
  y = j*dy;  // y coordinate
  for (i=1; i<imax-1; i++) {
    x = i*dx;  // x coordinate
    phi[j][i] = sin(M_PI*y)*sin(M_PI*x);
  }
}

maxdelta = 1.0; eps = 1.0e-14;

while (maxdelta > eps) {
  maxdelta = 0.;

  for (j=1; j<jmax-1; j++)
    for (i=1; i<imax-1; i++) {
      phi_new[j][i] = (phi[j-1][i]+phi[j][i-1]+phi[j][i+1]+phi[j+1][i])*0.25;
      maxdelta = max(maxdelta, abs(phi_new[j][i]-phi[j][i]));
    }

  /* pointer swapping */
  temp_ptr = phi_new; phi_new = phi; phi = temp_ptr;
}
```

Parallelizing 2D Jacobi computation

- 2D domain decomposition
- The P processes are organized as a 2D $N \times M$ process topology (assume $P = N \times M$)
- The interior points are evenly divided among the processes
- Instead of two ghost points per process (in 1D), we need now a layer of ghost points around each 2D subdomain
- Each process has up to 4 neighbors, the messages to be sent/received will be arrays of values

Some "bookkeeping" is needed to find out which MPI processes are neighbors in 2D!

Virtual topologies

- MPI suits very well for implementing domain decomposition (Section 9.2.5) on distributed-memory parallel computers.
- However, setting up the "logical" process grid and keeping track of which ranks have to exchange halo data is nontrivial.
- MPI contains some functionality to support this recurring task in the form of *virtual topologies*.
- To provide a convenient process naming scheme, which fits the required communication pattern.

Cartesian topologies

Example: a 2D global Cartesian mesh of size 3000×4000. Suppose we want to use $3 \times 4 = 12$ MPI processes to divide the global mesh, with each process holding a piece of 1000×1000 submesh.

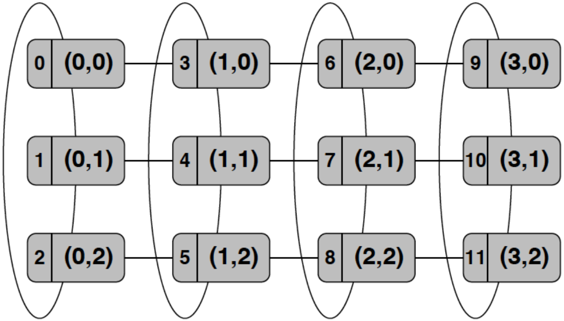


Figure 9.6: Two-dimensional Cartesian topology: 12 processes form a 3×4 grid, which is periodic in the second dimension but not in the first. The mapping between MPI ranks and Cartesian coordinates is shown.

Cartesian topologies (2)

- As shown in the preceding figure, each process can either be identified by its rank or its Cartesian coordinates.
- Each process has a number of neighbors, which depends on the grid's dimensionality. (In our example, the number of dimensions is two, which leads to at most four neighbors per process.)
- MPI can help with establishing the mapping between MPI ranks and Cartesian coordinates in the process grid.

MPI_Cart_create

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, const int dims[],
                    const int periods[], int reorder, MPI_Comm * comm_cart)
```

- A new, "Cartesian" communicator `comm_cart` is generated, which can be used later to refer to the topology.
- The `periods` array specifies which Cartesian directions are periodic, and the reorder parameter allows, if true, for rank reordering so that the rank of a process in communicators `comm_old` and `comm_cart` may differ.
- Here, MPI merely keeps track of the topology information.

Functions of MPI_Cart_coords & MPI_Cart_rank

```
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int coords[])
```

```
int MPI_Cart_rank(MPI_Comm comm, const int coords[], int *rank)
```

- These are two “service” functions responsible for the translation between Cartesian process coordinates and an MPI rank.
- MPI_Cart_coords() calculates the Cartesian coordinates for a given MPI rank.
- The reverse mapping, i.e., from Cartesian coordinates to an MPI rank, is performed by MPI_Cart_rank().

2D $N \times M$ domain decomposition

```
MPI_Comm comm_cart;
int dim[2], period[2], reorder, my_coord[2];

dim[0]=M; period[0]=0; // x direction
dim[1]=N; period[1]=0; // y direction
reorder=1;

MPI_Cart_create(MPI_COMM_WORLD, 2, dim, period, reorder, &comm_cart);
MPI_Comm_rank(comm_cart, &my_rank);
MPI_Cart_coords(comm_cart, my_rank, 2, my_coord)

my_xstart = my_coord[0]*(imax-2)/M;
my_xstop = (my_coord[0]+1)*(imax-2)/M;
my_imax = my_xstop-my_xstart+2;

my_ystart = my_coord[1]*(jmax-2)/N;
my_ystop = (my_coord[1]+1)*(jmax-2)/N;
my_jmax = my_ystop-my_ystart+2;
```

2D Jacobi: communication in *x*-direction

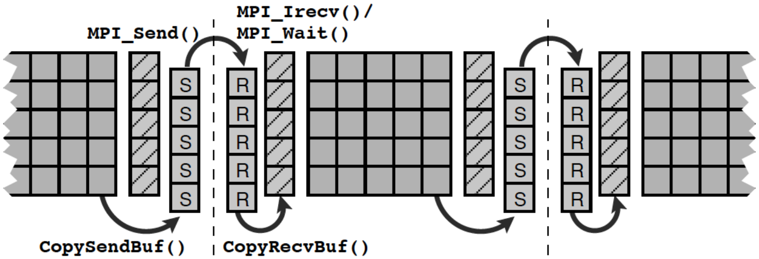


Figure 9.9: Halo communication for the Jacobi solver (illustrated in two dimensions here) along one of the coordinate directions. Hatched cells are ghost layers, and cells labeled “R” (“S”) belong to the intermediate receive (send) buffer. The latter is being reused for all other directions. Note that halos are always provided for the grid that gets read (not written) in the upcoming sweep. Fixed boundary cells are omitted for clarity.

2D Jacobi: communication in x-direction (2)

- Need two send-buffers (downward & upward), both of length `my_jmax-2`
  - Need to pack outgoing messages before send
- Need also two receive-buffers, of length `my_jmax-2`
  - Need to unpack incoming messages after completed receive

```
void copySendBuf0 (double **array, double *sbuf0, int my_imax, int my_jmax)
{
  for (int j=1; j<my_jmax-1; j++)
    sbuf0[j-1] = array[j][1];
}

void copyRecvBuf0 (double **array, double *rbuf0, int my_imax, int my_jmax)
{
  for (int j=1; j<my_jmax-1; j++)
    array[j][0] = rbuf0[j-1];
}
```

2D Jacobi: communication in y-direction

- No need for send-buffers and receive-buffers
- The data points that constitute an outgoing y-direction message already reside contiguously in memory
- Similarly, an incoming y-direction message can be stored directly to the target data array

MPI_Cart_shift

A regular task with domain decomposition is to find out, for each process, their neighbors (more specfcially, the neighbors' MPI ranks) along a certain Cartesian dimension.

```
int MPI_Cart_shift(MPI_Comm comm, int direction, int disp, int *rank_source,
                   int *rank_dest)
```

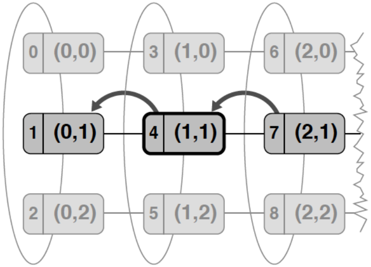


Figure 9.7: Example for the result of `MPI_Cart_shift()` on a part of the Cartesian topology from Figure 9.6. Executed by rank 4 with `direction`=0 and `disp`=−1, the function returns `rank_source`=7 and `rank_dest`=1.

Implementing *x*-direction communication

```
MPI_Cart_shift(comm_cart, 0, -1, &rank1, &rank0);

if (rank1!=MPI_PROC_NULL)
  MPI_Irecv(rbuf1,my_jmax-2,MPI_DOUBLE,rank1,0,comm_cart,&req_x1);
if (rank0!==MPI_PROC_NULL)
  MPI_Irecv(rbuf0,my_jmax-2,MPI_DOUBLE,rank0,0,comm_cart,&req_x0);

if (rank0!=MPI_PROC_NULL) {
  copySendBuf0(phi,sbuf0,my_imax,my_jmax);
  MPI_Send(sbuf0,my_jmax-2,MPI_DOUBLE,rank0,0,comm_cart);
}
if (rank1!=MPI_PROC_NULL)  {
  copySendBuf1(phi,sbuf1,my_imax,my_jmax);
  MPI_Send(sbuf1,my_jmax-2,MPI_DOUBLE,rank1,0,comm_cart);
}

if (rank1!=MPI_PROC_NULL) {
  MPI_Wait(&req_x1,&status_x1);
  copyRecvBuf1(phi,rbuf1,my_imax,my_jmax);
}
if (rank0!=MPI_PROC_NULL) {
  MPI_Wait(&req_x0,&status_x0);
  copyRecvBuf0(phi,rbuf0,my_imax,my_jmax);
}
```

Implementing *y*-direction communication

```
MPI_Cart_shift(comm_cart, 1, -1, &rank1, &rank0);

if (rank1!=MPI_PROC_NULL)
  MPI_Irecv(&(phi[my_jmax-1][1]),my_imax-2,MPI_DOUBLE,
            rank1,0,comm_cart,&req_y1);

if (rank0!==MPI_PROC_NULL)
  MPI_Irecv(&(phi[0][1]),my_imax-2,MPI_DOUBLE,
            rank0,0,comm_cart,&req_y0);

if (rank0!=MPI_PROC_NULL)
  MPI_Send(&(phi[1][1]),my_imax-2,MPI_DOUBLE,rank0,0,comm_cart);

if (rank1!=MPI_PROC_NULL)
  MPI_Send(&(phi[my_jmax-2][1]),my_imax-2,MPI_DOUBLE,rank1,0,comm_cart);

if (rank1!=MPI_PROC_NULL)
  MPI_Wait(&req_y1,&status_y1);

if (rank0!=MPI_PROC_NULL)
  MPI_Wait(&req_y0,&status_y0);
```

How to parallelize 3D Jacobi?

- Need to construct 3D Cartesian topology
- 3D decomposition of all the interior points
- Each process has up to 6 neighbors
- The entire ghost layer consists of up to 6 sides
- Communication in z-direction is the simplest, no need to pack/unpack messages
  - actually communicates a little bit more than strictly necessary
- For x and y-directions, need to pack outgoing messages before send, and unpack incoming messages after completed receive