Examples of MPI programming

A 1D example

The computational problem:

- A uniform mesh in x-direction with $M+2$ points:
  - x_0 is left boundary point, x_{M+1} is right boundary point
  - $x_1, x_2, \ldots, x_M$ are interior points
- The main computation is a time-stepping procedure. That is, for $\ell = 1, 2, \ldots$ we compute
  - $$u_i^{\ell+1} = 2u_i^\ell - u_i^{\ell-1} + C_2\left(u_{i-1}^\ell - 2u_i^\ell + u_{i+1}^\ell\right) \quad i = 1, 2, \ldots, M$$
  - u_i^ℓ denotes a value at spatial point x_i and time level ℓ
  - C_2 is a constant
  - u_0^ℓ and u_{M+1}^ℓ are given as boundary conditions (for all time levels)
- The above computation may arise from solving a 1D wave equation, but we don't need to know the mathematical/numerical details

Observations

- To compute the $u^{\ell+1}$ values on time level $\ell+1$, we need two preceding time levels: ℓ and $\ell-1$
  - We assume that the u^0 and u^1 values (for all subscript i indices) are given as initial conditions
  - The main computation is a time stepping iteration
- The value of $u_i^{\ell+1}$ does not depend on the value of $u_j^{\ell+1}$
  - The $u^{\ell+1}$ values can be computed **independently**, in any order!!!

Serial implementation

- Three 1D arrays are needed:
  - $u^{\ell+1}$: `double *up=(double*)malloc((M+2)*sizeof(double));`
  - u^{ℓ}: `double *u=(double*)malloc((M+2)*sizeof(double));`
  - $u^{\ell-1}$: `double *um=(double*)malloc((M+2)*sizeof(double));`
- Preparation work for prescribing the u^0 (`um` array) and u^1 values (`u` array)
- Main computation: A `while`-loop for doing the time steps
  - At each time step, a `for`-loop for updating the interior points

Time stepping

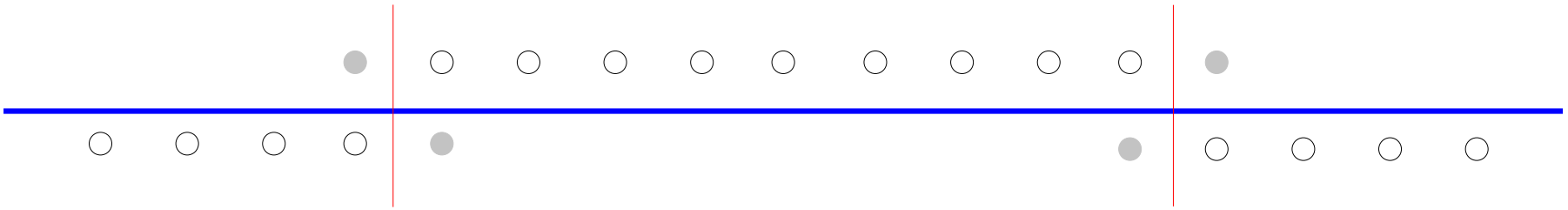
```
t = dt;
while (t<T){
  t += dt;
  for (i=1; i<=M; i++)
    up[i] = 2*u[i]-um[i]+C2*(u[i-1]-2*u[i]+u[i+1]);
  up[0] = value_of_left_BC(t);   // left boundary condition
  up[M+1] = value_of_rigt_BC(t); // right boundary condition

  /* preparation for next time step: shuffle the three arrays */
  tmp = um;
  um = u;
  u = up;
  up = tmp;
}
```

Parallelization

Parallelization starts with dividing the work

- The global domain is decomposed into P segments (subdomains)
  - actually, the M interior points are divided



MPI programming

- Each subdomain is assigned with a portion of the interior points

```
int P, my_rank, my_start, my_stop, M_local;
MPI_Comm_size (MPI_COMM_WORLD, &P);
MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
my_start = (my_rank*M)/P+1;
my_stop = ((my_rank+1)*M)/P;
M_local = my_stop - my_start + 1;
```

- In addition, each subdomain is expanded with two "ghost points"
  - if there is a neighbor subdomain over the boundary, the value of the ghost point is to be provided by the neighbor
  - if there is no neighbor subdomain over the boundary, the ghost point is actually a physical boundary point (x_0 or x_{M+1})
- Data allocation per MPI process:

```
double *up_local=(double*)malloc((M_local+2)*sizeof(double));
double *u_local =(double*)malloc((M_local+2)*sizeof(double));
double *um_local=(double*)malloc((M_local+2)*sizeof(double));
```

Computation per MPI process

```
t = dt;
while (t<T){
  t += dt;
  for (i=1; i<=M_local; i++)
    up_local[i] = 2*u_local[i]-um_local[i]
                  +C2*(u_local[i-1]-2*u_local[i]+u_local[i+1]);

  if (my_rank==0)
    up_local[0] = value_of_left_BC(t);   // left boundary condition
  if (my_rank==P-1)
    up_local[M_lcoal+1] = value_of_rigt_BC(t); // right boundary condition

  communicate1D (up_local, M_local, my_rank, P);

  /* preparation for next time step: shuffle the three arrays */
  tmp = um_lcoal;
  um_local = u_local;
  u_local = up_local;
  up_local = tmp;
}
```

Communication

```
void communicate1D (double *up_local, int M_local, int my_rank, int P)
{
  MPI_Status status;

  if (my_rank%2) { // odd-number proc exchanges with left neighbor
   if (my_rank>1) {
     MPI_Send (&(up_local[1]),1,MPI_DOUBLE,my_rank-1,1,MPI_COMM_WORLD);
     MPI_Recv (&(up_local[0]),1,MPI_DOUBLE,my_rank-1,2,MPI_COMM_WORLD, &sta
   }
  }
  else { // even-number proc exchanges with right neighbor
   if (my_rank<(P-1)) {
     MPI_Recv (&(up_local[M_local+1]),1,MPI_DOUBLE,my_rank+1,1,MPI_COMM_WOR
     MPI_Send (&(up_local[M_local]),1,MPI_DOUBLE,my_rank+1,2,MPI_COMM_WORLD
  }
```

Communication (continued)

```
  if (my_rank%2) { // odd-number proc exchanges with right neighbor
   if (my_rank<(P-1) {
     MPI_Send (&(up_local[M_local]),1,MPI_DOUBLE,my_rank+1,3,MPI_COMM_WORLD
     MPI_Recv (&(up_local[M_local+1]),1,MPI_DOUBLE,my_rank+1,4,MPI_COMM_WOR
   }
  }
  else { // even-number proc exchanges with left neighbor
   if (my_rank>1) {
     MPI_Recv (&(up_local[1]),1,MPI_DOUBLE,my_rank-1,3,MPI_COMM_WORLD,&stat
     MPI_Send (&(up_local[0]),1,MPI_DOUBLE,my_rank-1,4,MPI_COMM_WORLD);
  }
}
```

Alternative communication

An equivalent implementation that uses `MPI_Sendrecv`:

```
void communicate1D (double *up_local, int M_local, int my_rank, int P)
{
  MPI_Status status;
  int left_neigh = (my_rank>1) ? my_rank-1 : MPI_PROC_NULL;
  int right_neigh = (my_rank<(P-1)) ? my_rank+1 : MPI_PROC_NULL;

  // send to left neighbor, receive from right neighor
  MPI_Sendrecv (&(up_local[1]),1,MPI_DOUBLE, left_neigh,5,
                &(up_local[M_local+1]),1,MPI_DOUBLE, right_neigh,5,
                MPI_COMM_WORLD, &status);

  // send to right neighbor, receive from left neighor
  MPI_Sendrecv (&(up_local[M_local]),1,MPI_DOUBLE, right_neigh,6,
                &(up_local[0]),1,MPI_DOUBLE, left_neigh,6,
                MPI_COMM_WORLD, &status);
}
```

A 2D example

2D uniform grid:
$x_0, x_1, \ldots, x_M, x_{M+1}$ in the x direction
$y_0, y_1, \ldots, y_N, y_{N+1}$ in the y direction

$$
\begin{aligned}
u_{i,j}^{\ell+1} = \quad & 2u_{i,j}^{\ell} - u_{i,j}^{\ell-1} \\
+ \quad & C_{2x}\left(u_{i-1,j}^{\ell} - 2u_{i,j}^{\ell} + u_{i+1,j}^{\ell}\right) \\
+ \quad & C_{2y}\left(u_{i,j-1}^{\ell} - 2u_{i,j}^{\ell} + u_{i,j+1}^{\ell}\right) \\
& i = 1, 2, \ldots, M,\ j = 1, 2, \ldots, N
\end{aligned}
$$

Serial implementation

```
double **up, **u, **um;  // three 2D arrays
// ...
while (t<T){
  t += dt;
  for (j=1; j<=N; j++)
    for (i=1; i<=M; i++)
      up[j][i] = 2*u[j][i]-u[j][i]
                 +C2x*(u[j][i-1]-2*u[j][i]+u[j][i+1])
                 +C2y*(u[j-1][i]-2*u[j][i]+u[j+1][i]);

  enforce_left_BC (up,M,N,t);
  enforce_right_BC (up,M,N,t);
  enforce_lower_BC (up,M,N,t);
  enforce_upper_BC (up,M,N,t);

  /* preparation for next time step: shuffle the three arrays */
  tmp = um;
  um = u;
  u = up;
  up = tmp;
}
```

Parallelization approach 1

- Divide the 2D domain into horizontal blocks
  - The N interior points in the y direction is divided evenly among P processes
- Each MPI process is responsible for a subdomain of $M \times N_{\text{local}}$ interior points
- Each subdomain is extended with a surrounding layer of ghost points

MPI implementation (array allocation)

```
double **up_local, **u_local, **um_local;
int P, my_rank, my_start, my_stop, N_local;
MPI_Comm_size (MPI_COMM_WORLD, &P);
MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
my_start = (my_rank*N)/P+1;
my_stop = ((my_rank+1)*N)/P;
N_local = my_stop - my_start + 1;

up_local=(double**)malloc((N_local+2)*sizeof(double*));
up_local[0]=(double*)malloc((N_local+2)*(M+2)*sizeof(double));
for (j=1; j<=N_local+1; j++)
  up_local[j] = &(up_local[0][j*(M+2)]);

// similarly for allocating arrays u_local and um_local
```

MPI implementation (main computation)

```
while (t<T){
  t += dt;
  for (j=1; j<=N_local; j++)
    for (i=1; i<=M; i++)
      up_local[j][i] = 2*u_local[j][i]-u_local[j][i]
                  +C2x*(u_local[j][i-1]-2*u_local[j][i]+u_local[j][i+1])
                  +C2y*(u_local[j-1][i]-2*u_local[j][i]+u_local[j+1][i]);

  enforce_left_BC (up_local,M,N_local,t);
  enforce_right_BC (up_local,M,N_local,t);
  if (my_rank==0)
    enforce_lower_BC (up_local,M,N_local,t);
  if (my_rank==P-1)
    enforce_upper_BC (up_local,M,N_local,t);

  communicate2D_vertical (up_local, M, N_local, my_rank, P);

  // pointer swap for arrays up_local, u_local um_local
}
```

Communication in the vertical direction

```
void communicate2D_vertical(double **up_local, int M, int N_local,
                            int my_rank, int P)
{
  MPI_Status status;
  int lower_neigh = (my_rank>1) ? my_rank-1 : MPI_PROC_NULL;
  int upper_neigh = (my_rank<(P-1)) ? my_rank+1 : MPI_PROC_NULL;

  // send to lower neighbor, receive from upper neighor
  MPI_Sendrecv (&(up_local[1][1]),M,MPI_DOUBLE, lower_neigh,5,
                &(up_local[N_local+1][1]),M,MPI_DOUBLE, upper_neigh,5,
                MPI_COMM_WORLD, &status);

  // send to upper neighbor, receive from lower neighor
  MPI_Sendrecv (&(up_local[N_local][1]),M,MPI_DOUBLE, upper_neigh,6,
                &(up_local[0][1]),M,MPI_DOUBLE, lower_neigh,6,
                MPI_COMM_WORLD, &status);
}
```

Parallelization approach 2

- 2D domain partitioning
  - Number of subdomains is $P = Q \times R$
  - The M interior points in the x direction are divided into Q pieces
  - The N interior points in the y direction are divided into R pieces
  - Each subdomain is responsible for $M_{\text{local}} \times N_{\text{local}}$ interior points
- Communication is now needed in both horizontal (x) and vertical (y) directions
  - Each MPI process needs a double index `my_rank_x`, `my_rank_y`
  - Vertical communication is as simple as in Parallelization approach 1
  - Horizontal communication requires intermediate buffers for outgoing and incoming messages, because the data points of array `up_local` that constitute the horizontal messages are not contiguous in memory
- MPI programming details are omitted