Introduction to Parallel Programming with MPI

Lecture #3*: Collective Communications*

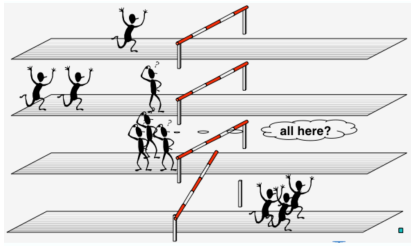
Andrea Mignone[1]

[1]Dipartimento di Fisica- Turin University, Torino (TO), Italy

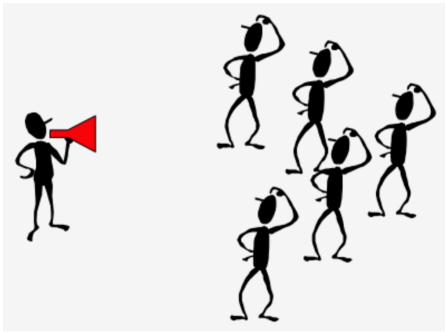
Collective Communications

- Collective communication operations are composed of several point-to-point operations.
- They are optimized internal implementations (e.g. tree algorithms) used by MPI and completely transparent to the user.
- Some important examples are:

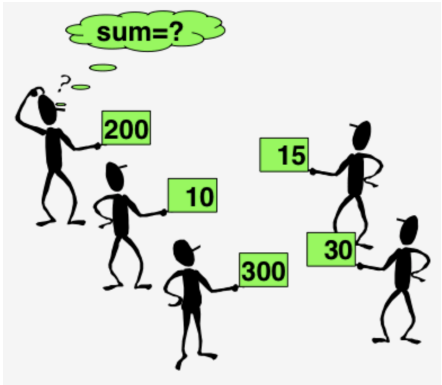
- *Barrier*: synchronize all processes:



- *Broadcast*: one process communicates with several others in the same group:



- *Reduction*: combine data from several processes into a single result:



Data Communication

- It is important to realize that collective operations must fulfill some properties:
  - All processes in the communicator must call the same collective routine and must communicate;
  - All processes must be able to start the collective routine;
  - Collective operations can be *blocking* or *non-blocking*;
  - No message tags allowed;
  - Receive buffers on all processes must have exactly the same size;

Collective Routines

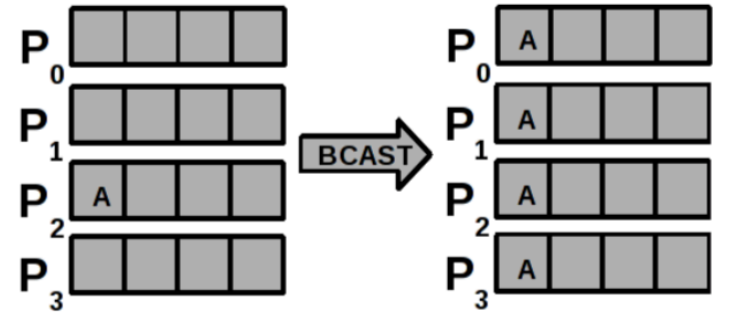
- MPI provides several collective functions. Some of them are

Function	Description
`MPI_Gather(), MPI_Gatherv()`	Collects data from tasks
`MPI_Allgather(), MPI_Allgatherv()`	Collects data from tasks and distribute them
`MPI_Reduce()`	Reduce values on all processes to a single value
`MPI_Allreduce()`	Same as before, but also distribute them
`MPI_Scatter(), MPI_Scatterv()`	Send data from one process to all others
`MPI_Alltoall(), MPI_Alltoallv()`	Send data from all to all processes
`MPI_Bcast()`	Broadcast message from one proc to all
`MPI_Barrier()`	Block until all procs have reached the same point

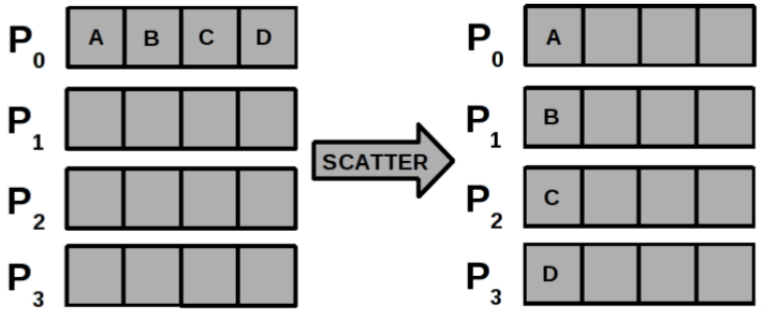
- Many functions come with the “v” version allowing data chunks to have different sizes.

Collective Routines

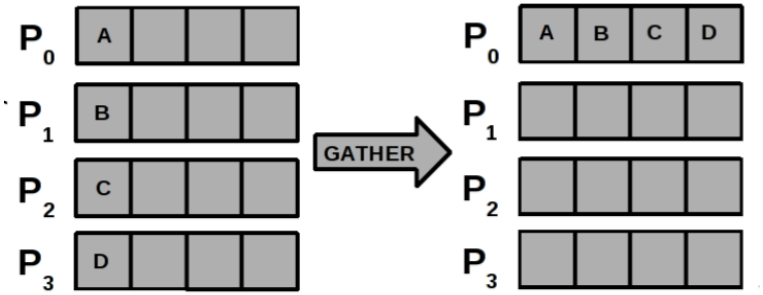
MPI_Bcast()



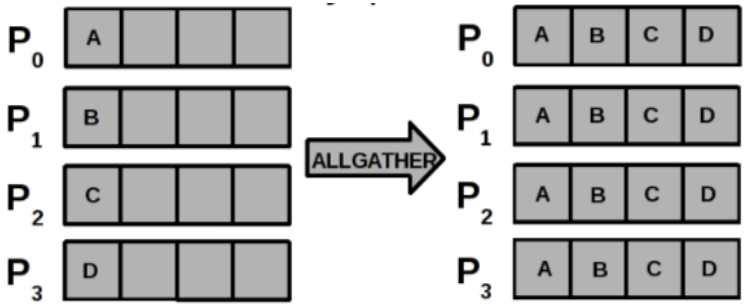
MPI_Scatter()



MPI_Gather()



MPI_AllGather()



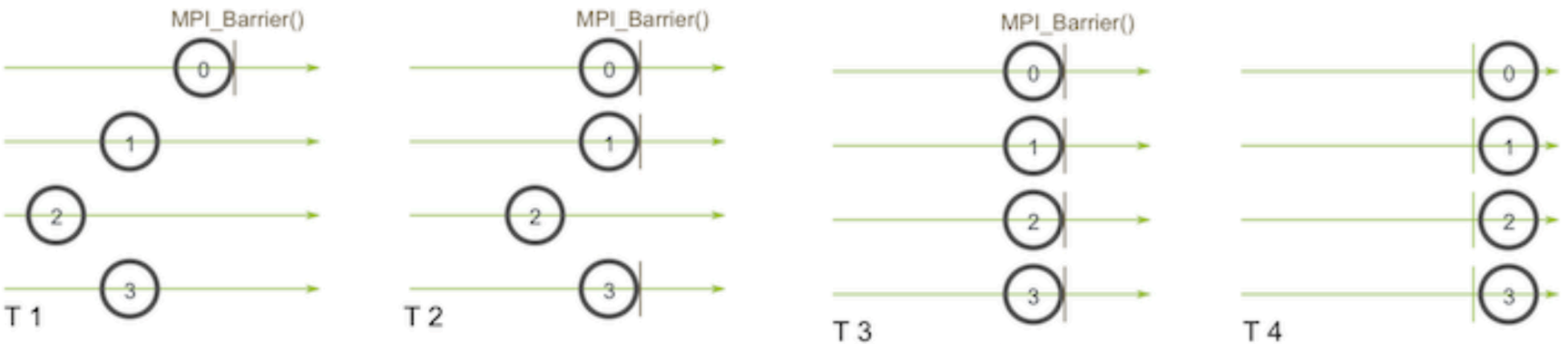
Synchronization Point: Placing Barriers

- Collective calls implies that processes are synchronized at the time of the call.

- MPI has a dedicated function for synchronizing processes:

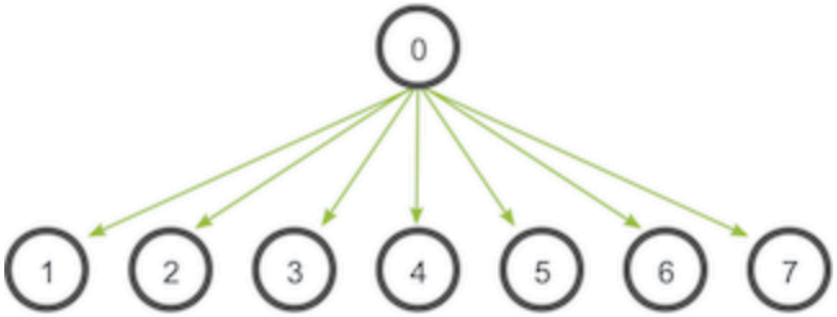
```
MPI_Barrier(MPI_Comm comm);
```

- As the name suggests, this function places a barrier so that no processes in the communicator can pass the barrier until all of them call the function.



Broadcast

- During a broadcast operation, one process sends data to all processes in the communicator:



- Broadcast is commonly used to send the user input / problem configuration to all processes.
- The function syntax is the following:

```
MPI_Bcast(void* buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
```

where

- `buf`: starting address of the buffer to send;
- `count`: number of elements in the buffer;
- `datatype`: the datatype of the buffer;
- `root`: rank of broadcast process;
- `comm`: communicator;

Example #1: `MPI_Bcast()`

- Naively, you may achieve a broadcast operation using the `MPI_Send()` and `MPI_Recv()` functions:

```
...
int main(int argc, char ** argv)
{
  int i, rank, size;
  int buf[NELEMENTS];

  ...   // Initialize the MPI execution environment

  if (rank == 0){  // Process #0 broadcast to all processes with MPI_Send() and MPI_Recv()
    int dest;
    for (i = 0; i < NELEMENTS; i++) buf[i] = 1 + i*i; // Fill buffer

    for (dest = 1; dest < size; dest++){
      MPI_Send(buf, NELEMENTS, MPI_INT, dest, 0, MPI_COMM_WORLD);
    }
  }else{  // Receive from rank #0
    MPI_Recv(buf, NELEMENTS, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  }

  MPI_Finalize();
  return 0;
}
```

- The previous code will dispatch `buf` to all processes.

Example #1: Broadcast

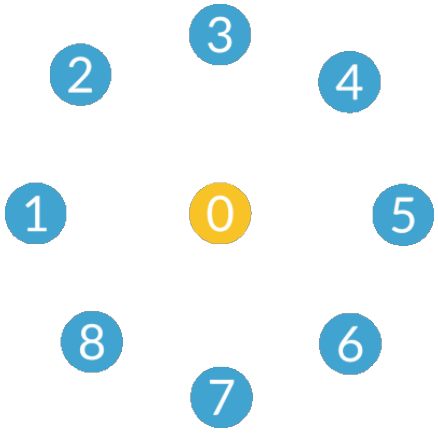
- There's a simpler and more efficient way to achieve the same result using `MPI_Bcast()`:

```
int main(int argc, char ** argv)
{
  ...
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  if (rank == 0){
    for (i = 0; i < NELEMENTS; i++) buf[i] = 1 + i*i;   /* Fill buffer  */
  }
  MPI_Bcast(buf, NELEMENTS, MPI_INT, 0, MPI_COMM_WORLD);

  MPI_Finalize();
  return 0;
}
```

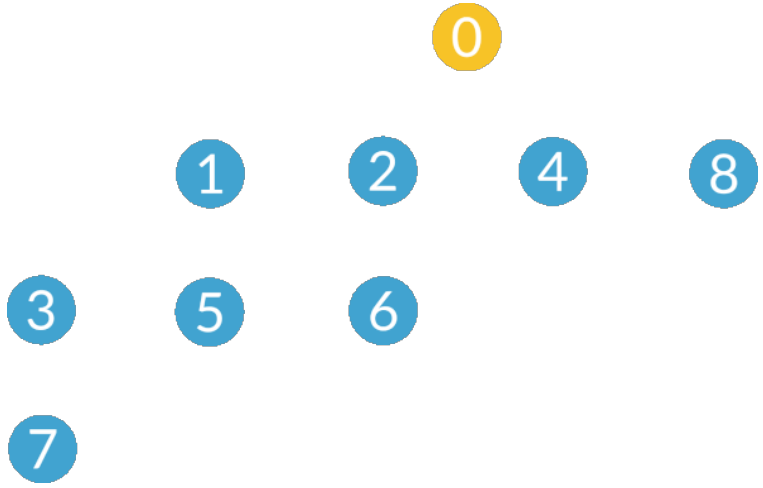
- It is important to realize that `MPI_Bcast()` is *not* a simple wrapper around `MPI_Send()` and `MPI_Recv()` function.
- `MPI_Bcast()` is actually much more efficient. Let's see why.

Broadcast using MPI_Send/Recv or MPI_Bcast

- In the 1st approach rank #0 sends buffer to all processes sequentially:



- The 2nd approach (`MPI_Bcast()`) is based on a tree-based communication algorithm that can use more of the available network links at once:



Example #1b: Using `MPI_Bcast()` to sort procs output

- How can you use `MPI_Bcast()` so that all processes can print to `stdout` in an ordered manner ?
- The code output should be:

```
Hello, I'm processor #0
Hello, I'm processor #1
Hello, I'm processor #2
Hello, I'm processor #3
...
```

- Write a code that does that.

MPI_Gather() and MPI_Scatter()

- `MPI_Scatter()` is a collective function used to send data from a given root process to all processes in a communicator. In this sense `MPI_Scatter()` is very similar to `MPI_Bcast()`.
- However, while `MPI_Bcast()` sends the same piece of data to all processes, `MPI_Scatter()` sends chunks of an array to different processes:

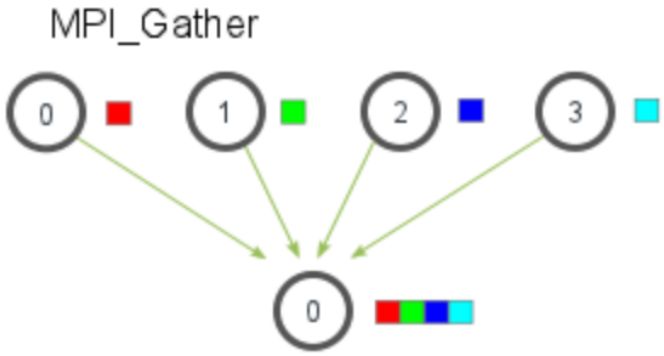

```
MPI_Scatter(void* send_buf, int send_count, MPI_Datatype send_datatype,
            void* recv_buf, int recv_count, MPI_Datatype recv_datatype,
            int root, MPI_Comm comm)
```

MPI_Scatter()

- Here `send_buf` is an array of data that resides on the root process.
- `send_count` and `send_datatype` specify how many elements of a specific MPI datatype will be sent to each process: if `send_count == 1` and `send_datatype == MPI_INT`, then process #0 gets the 1st integer of the array, process #1 gets the 2nd integer, and so on. If `send_count == 2`, then process #0 gets the 1st and 2nd integers, process #1 gets the 3rd and 4th and so on.
- In practice, `send_count` equals the number of elements in the array divided by the number of processes.
- The receiving parameters are nearly identical in respect to the sending parameters: `recv_buf` parameter is a buffer of data that can hold `recv_count` elements with datatype `recv_datatype`.
- The last parameters, `root` and `comm`, indicate the root process that is scattering the array of data and the communicator in which the processes reside.

MPI_Gather()

- `MPI_Gather()` is the inverse of `MPI_Scatter()`: it takes elements from many processes and gathers them to one single process. This routine is highly useful to many parallel algorithms, such as parallel sorting and searching.



- The function prototype is identical to `MPI_Scatter()`:

```
MPI_Gather(void* send_buf, int send_count, MPI_Datatype send_datatype,
           void* recv_buf, int recv_count, MPI_Datatype recv_datatype,
           int root, MPI_Comm communicator)
```

- Beware that only the root process needs to have a valid receive buffer. All other calling processes can pass `NULL` for `recv_buf`.

Example #2: Gathering and Scattering

- Let's write a code with the following tasks:

  T1- the root process collects numbers (`rank`^{2}`+1`) from all processes through and `MPI_Gather()` operation. This part of the code can be written as

```
...
int main(int argc, char ** argv)
{
  int    n, rank, size;
  double data;
  double *send_buf, *recv_buf;

  ...  // Initialize the MPI execution environment

  recv_buf = (double *) malloc(size*sizeof(double)); // allocate memory
  send_buf = (double *) malloc(size*sizeof(double));

  data = rank*rank + 1.0; // generate data on different procs
  MPI_Gather(&data,    1, MPI_DOUBLE,
             recv_buf, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

  if (rank == 0){
    printf ("[Gather()]:\n");
    for (n = 0; n < size; n++) printf (”data[%d] = %f\n",n,recv_buf[n]);
  }

  ...
```

Example #2: Gathering and Scattering

- We then continue with the next task:

  T2- the root process scatters numbers (n^2-1 with `n=0..size-1`) to all other processes through an `MPI_Scatter()` operation:

```
...

if (rank == 0){
  for (n = 0; n < size; n++) send_buf[n] = n*n - 1.0; // Generate “size” random numbers
}
MPI_Scatter(send_buf, 1, MPI_DOUBLE,
            &data,    1, MPI_DOUBLE, 0, MPI_COMM_WORLD);

printf ("[Scatter, proc #%d] = %f\n",rank,data);

...
```

- The output should look like

```
[Gather()]:
data[0] = 1.000000
data[1] = 2.000000
data[2] = 5.000000
data[3] = 10.000000
[Scatter, proc #0] = -1.000000
[Scatter, proc #1] = 0.000000
[Scatter, proc #2] = 3.000000
[Scatter, proc #3] = 8.000000
```

Data Reduction: `MPI_Reduce()`

- A reduction operation involves reducing a set of numbers into a smaller set of numbers through some kind of operation (e.g. `max()`, `sum()`, and so forth).
- Reductions are indeed very simple operations applied on all the buffers of all processes.
- Operations are usually pre-defined (e.g. `MPI_MAX`, `MPI_SUM`, etc...) but can also be user-defined (more advanced). Usually, the predefined operations are largely sufficient for most applications.
- The function `MPI_Reduce()` takes an array of input elements on each process and returns an array of output elements to the root process. The output elements contain the reduced result. The prototype for `MPI_Reduce()` looks like this:

```
MPI_Reduce(void* send_buf, void* recv_buf, int count, MPI_Datatype datatype,
           MPI_Op op, int root, MPI_Comm comm)
```

Data Reduction

- Here `send_buf` and `recv_buf` are, respeticely, the send buffer and the output results.
- The next arguments, `count` and `datatype`, identify the number of elements and the data type we are working on (`int`, `double`, etc...).
- The operation is specified by the argument "`op`" which can be one of

op	Action
`MPI_MAX`	Returns the maximum element
`MPI_MIN`	Returns the minimum element
`MPI_SUM`	Sum all the elements
`MPI_PROD`	Multiply all elements
`MPI_LAND`	Perform a logical "*and*" across all elements
`MPI_LOR`	Perform a logical "*or*" across all elements
`MPI_BAND`	Perform a bitwise "*and*" across all elements
`MPI_BOR`	Perform a bitwise "*or*" across all elements

- Finally, the rank of the receiving process is given by `root`.

Example #2: Random Number Distribution

- Let's make an example: we want each process to generate independent, uniformly distributed N_p floating-point random numbers in the interval (0,100).
- We then wish to compute the average and the maximum of the distribution among all processes:

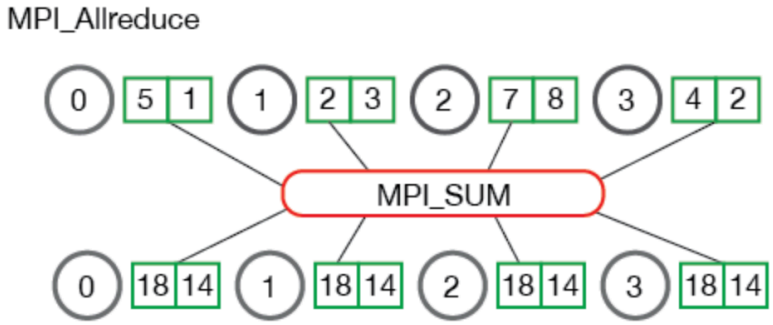
```
  ...
  srand48(time(NULL) + rank);   // Seed random sequence (different seed used for each process)
  for (i = 0; i < NELEMENTS; i++) buf[i] = drand48()*100.0;  // Fill buffer

  bmax_loc = bsum_loc = 0.0;
  for (i = 0; i < NELEMENTS; i++) {               // Compute local sum & max
    bsum_loc += buf[i];
    if (buf[i] > bmax_loc) bmax_loc = buf[i];
  }

  MPI_Reduce (&bmax_loc, &bmax, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD); // Reduce among
  MPI_Reduce (&bsum_loc, &bsum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD); // processors

  if (rank == 0){                                 // Print output
    bsum /= NELEMENTS*size;                       // There's NELEMENTS*size elements in total
    printf ("Distribution max     = %f\n",bmax);
    printf ("Distribution average = %f\n",bsum);
  }
  MPI_Finalize();
...
```

Data Reduction: MPI_Allreduce()

- Some parallel applications will require accessing the reduced results across all processes rather than the root process.
- To this end, the function `MPI_Allreduce()` will reduce the values and distribute the results to all processes. The function prototype is the following:

```
MPI_Allreduce(void* send_buf, void* recv_buf, int count, MPI_Datatype datatype,
              MPI_Op op, MPI_Comm comm)
```

- `MPI_Allreduce()` is identical to `MPI_Reduce()` with the exception that it does not need a root process id (since the results are distributed to all processes).



- Indeed `MPI_Allreduce()` is equivalent to an `MPI_Reduce()` followed by `MPI_Bcast()`.

Example #3: Evaluation of π

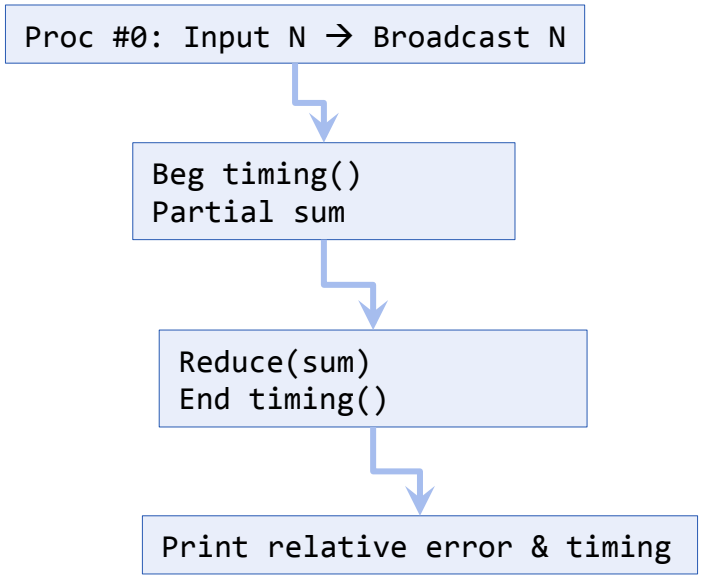
- Compute π using a numerical approximation in evaluating the integral

$$\int_0^1 \frac{4}{1+x^2}\,dx = \pi$$

- A simple numerical quadrature rule is, e.g., the midpoint rule:

$$\int_a^b f(x)\,dx \approx h\sum_{k=1}^{N} f(x_k) \qquad \text{where} \quad \begin{cases} h &= \frac{b-a}{N} \\ x_k &= a + \left(k - \frac{1}{2}\right) h \end{cases}$$

- The program takes as input the number of intervals `N` (quit if `N = 0`).
- Each process does a partial summation on `N/size` elements (do not use arrays!!).
- Reduction is done at the end and the relative error is output from proc #0.



Example #3: Evaluation of π

- In order to measure the code execution time use the `MPI_Wtime()` function (which returns a double):

```
tbeg = MPI_Wtime();  // Timer starts
...
tend = MPI_Wtime();  // Timer stops
...
Elapsed = tend - tbeg
```

- For N = 32 the error is ≈ 2.5904e-5. Repeat this using 1, 2 and 4 processors. Is the error always the same ?
- Measure the speedup using 1,2 and 4 procs with N=1234567890.
- Is the error the same with all processors ? Explain.

Application Example: 1D Heat Equation

- We now wish to solve the 1D partial differential equation (PDE)

$$\frac{\partial u}{\partial t} = D\frac{\partial^2 u}{\partial x^2}$$

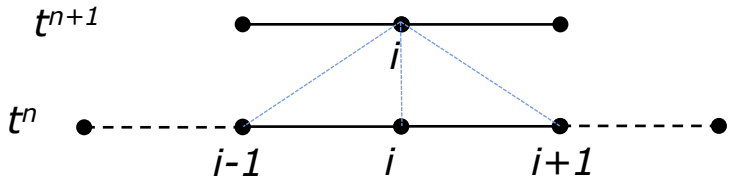
where D is a diffusion coefficient.

- Using finite differences we adopt a 1st order explicit method

$$u_i^{n+1} = u_i^n + D\frac{\Delta t}{\Delta x^2}\left(u_{i-1}^n - 2u_i^n + u_{i+1}^n\right)$$

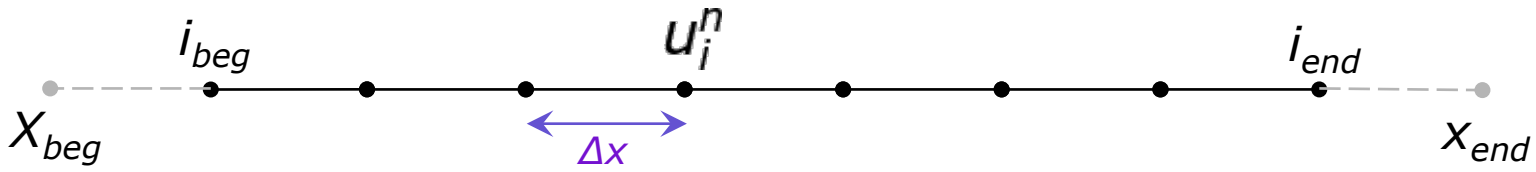
where *i* is the *grid* index while *n* is the *temporal* index.

- The time step is restricted by $\Delta t = C\frac{\Delta x^2}{D}$ with $C < \frac{1}{2}$
- This equation is used to model a diffusion process such as the conduction of heat on a finite size rod (*u* = internal energy or temperature).
- Note that the update formula involves a three-point stencil:



1D Heat Eq: Grid construction

- We construct a discrete mesh consisting of N_x interior points



where $\Delta x = (x_{end} - x_{beg})/(N_x+1)$.

- The domain endpoints (x_{beg} and x_{end}) constitute the boundary conditions and are called "ghost zones" (for our scheme only $N_{gh} = 1$ ghost zone is needed on each side).

- Therefore our arrays will have dimensions $N_x + 2N_{gh}$. <u>*Only interior points need to be updated*</u> while boundary conditions in the ghost zones must be prescribed at the beginning of each time step.

  - interior points are spanned by $beg \le i \le end$ ($beg = N_{gh}$, $end = beg + N_x - 1$);
  - Ghost zones on the left are defined by $i = 0...beg-1$.
  - Ghost zones on the right are defined by $i=end+1...end+N_{gh}$.

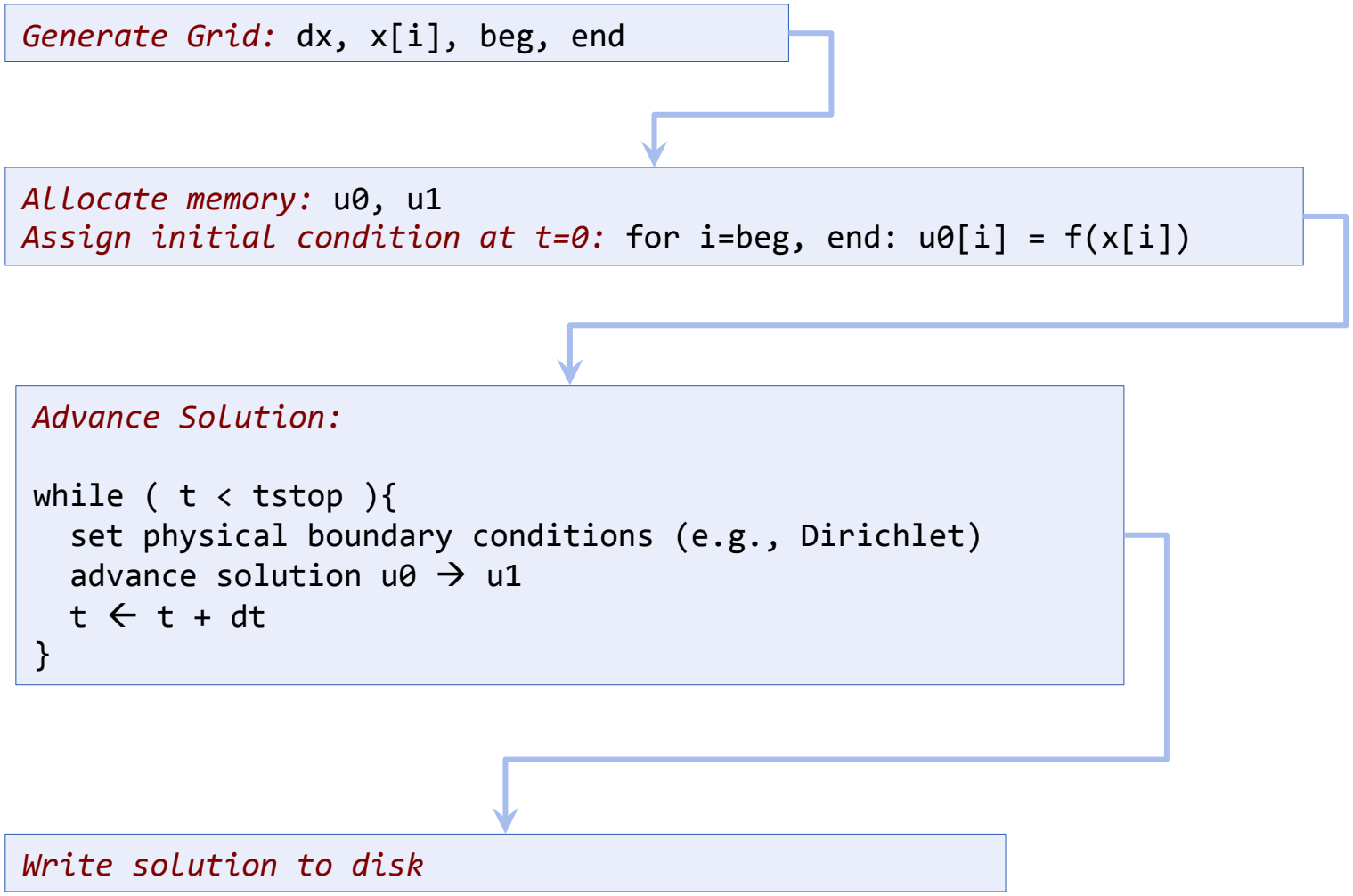
1D Heat Eq: Initial & Boundary Conditions

- The diffusion equation admits several analytical solution that can be used to assess the validity of the implementation.
- A useful one is $u(x,t) = A\exp^{-D\mu^2 t}\sin(\mu x + B) + C$

  where *A*, *B*, *C* and μ are arbitrary constants while D is the diffusion coeff.
- Here we use $A = 1$, $B = C = 0$, $D = 1$ and $\mu = \pi$.
- We consider a computational domain with $x_{beg} = 0$, $x_{end} = 1$;
- At t=0 we initialize, on $0<x<1$, $u_i^n = u(x_i, 0)$
- Since the solution is periodic and has an extremum at the domain endpoints, the boundary conditions can be either:
  - Periodic: $u(x) = u(x + L)$ where $L = x_{end} - x_{beg}$
  - Dirichlet b.c.: $u(0,t) = u(L,t) = 0$

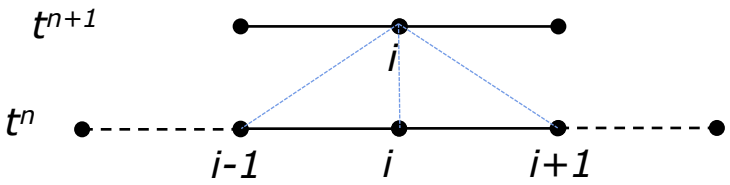
1D Heat Eq: Serial Code

- The serial implementation consists of the following 4 steps:



1D Heat Eq: The Advance Step

- Remember: advancing the solution from t^n to t^{n+1} always requires 3 points to be defined at the old time level: u_{i-1}, u_i and u_{i+1}.



- At the leftmost and rightmost interior points, boundary conditions must therefore be specified using ghost (guard) cells (`i = beg-1` and `i = end+1`)

- The update algorithm on a single processor looks like

```
while (t < tstop){

  u0[beg-1] = ... // Left boundary condition
  u0[end+1] = ... // Right boundary condition

  for (i = beg; i <= end; i++){   // Evolve interior points by dt
    u1[i] = u0[i] + dt/(dx*dx)*(u0[i-1] - 2.0*u0[i] + u0[i+1]);
  }
  t += dt;

  for (i = beg; i <= end; i++) u0[i] = u1[i]; // Copy array for next time level
}
```

- We carry out integration until $\mathtt{t}_{\mathtt{stop}} = \mathtt{0.1}$.

1D Heat Eq: Writing Data

- In the serial implementation, output data consists of a 2-column ascii data file,

```
x[beg]    u[beg]

x[beg+1]  u[beg+1]

...       ...

x[end]    u[end]
```

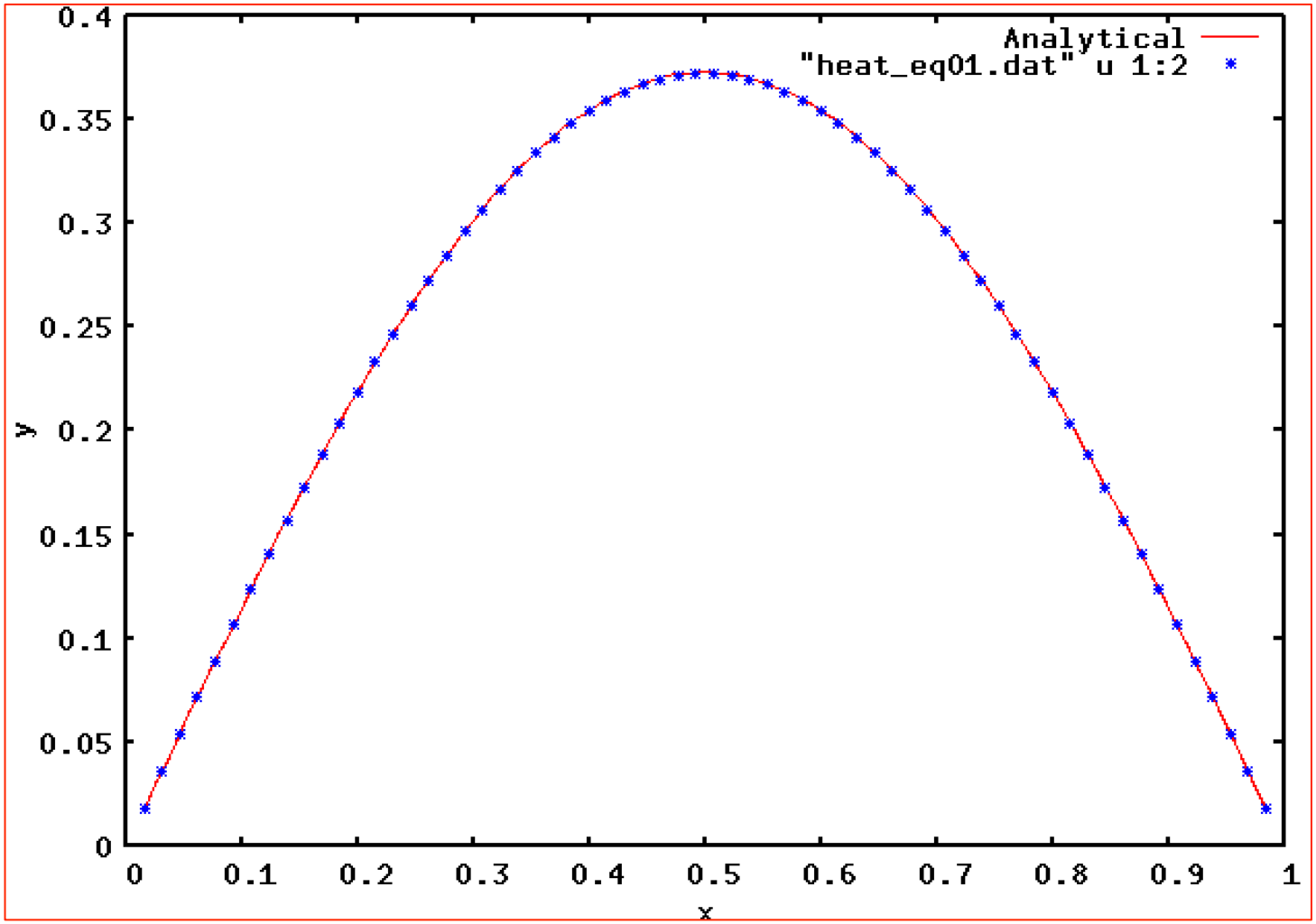
- To achieve this we implement the output function as

```
void Write (double *x, double *u, int beg, int end)
{
// x = local process grid
// u = local process solution array
// beg, end = local process start and ending indices
//            (interior points)
}
```

- Data can be written at the beginning (`t=0`) and at the end ($t=t_{stop}=0.1$).

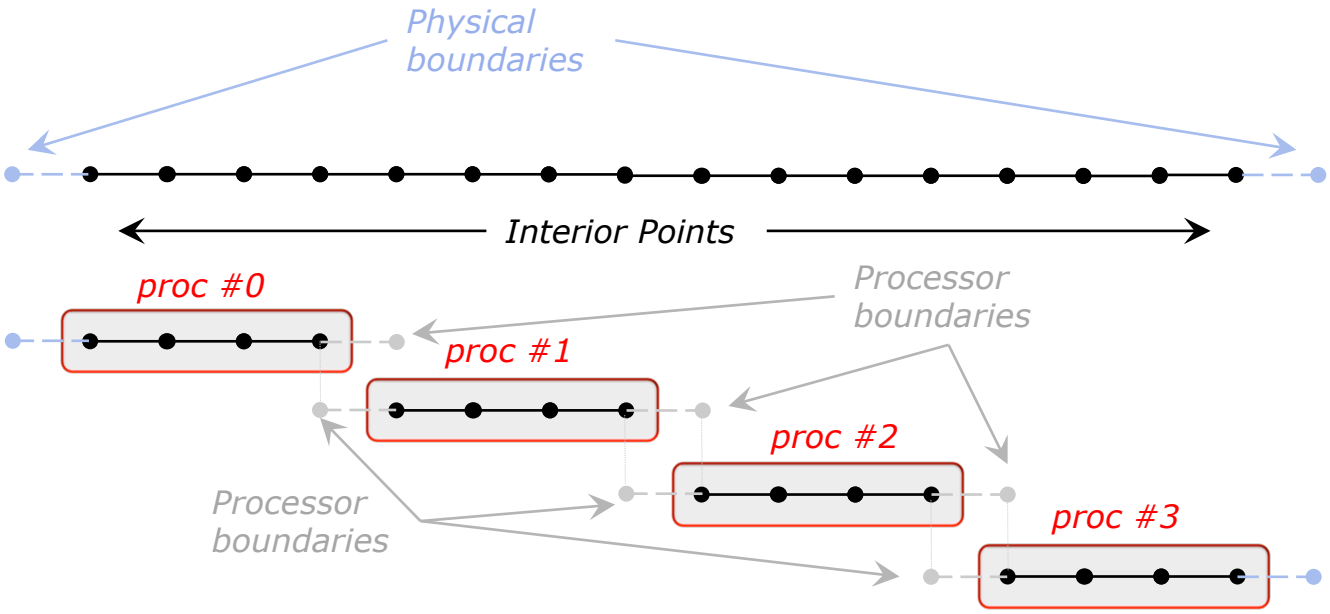
1D Heat Eq: Solution at t =0.1

- At t =0.1 we can overplot the analytical solution together with the numerical solution using, e.g., 64 points:



Parallel Implementation

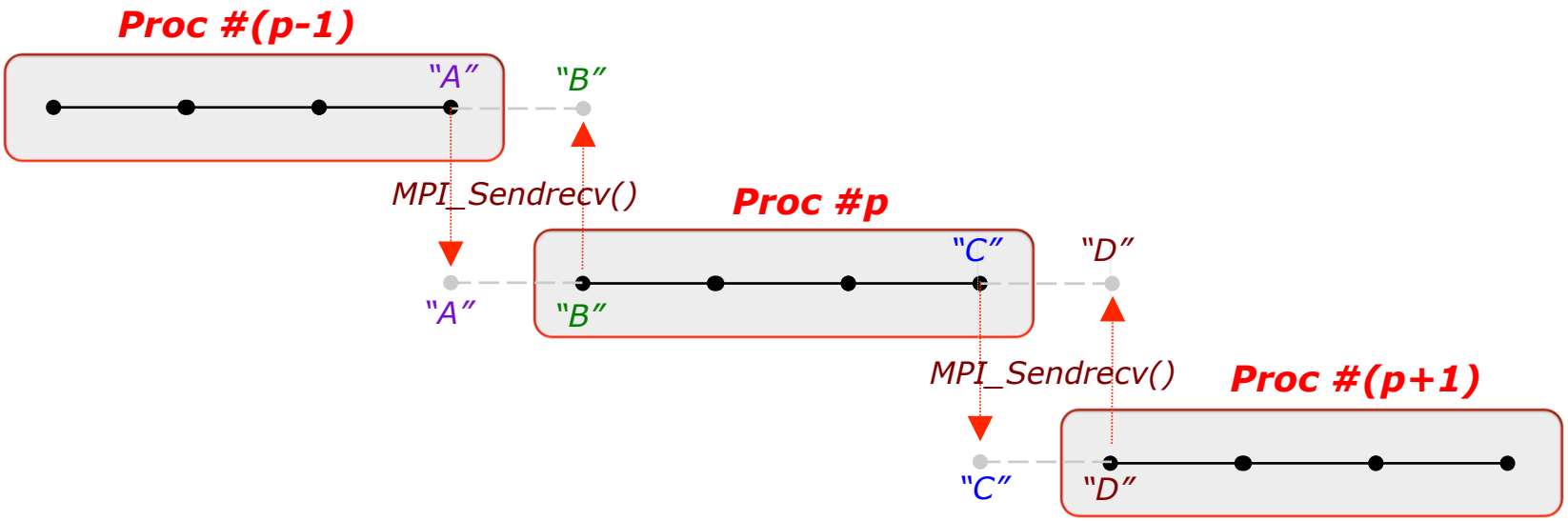
- In parallel we wish to achieve a uniform workload on all processors.
- In an explicit scheme (like the one we are using), we split the computational grid into (possibly) equal-sized meshes:



- If the global grid has N_g points, we want to assign to each process a local grid of $N_x=N_g/n_p$ points where n_p is the total number of processes.
- Process boundary can be either internal of physical.

Boundary Conditions:

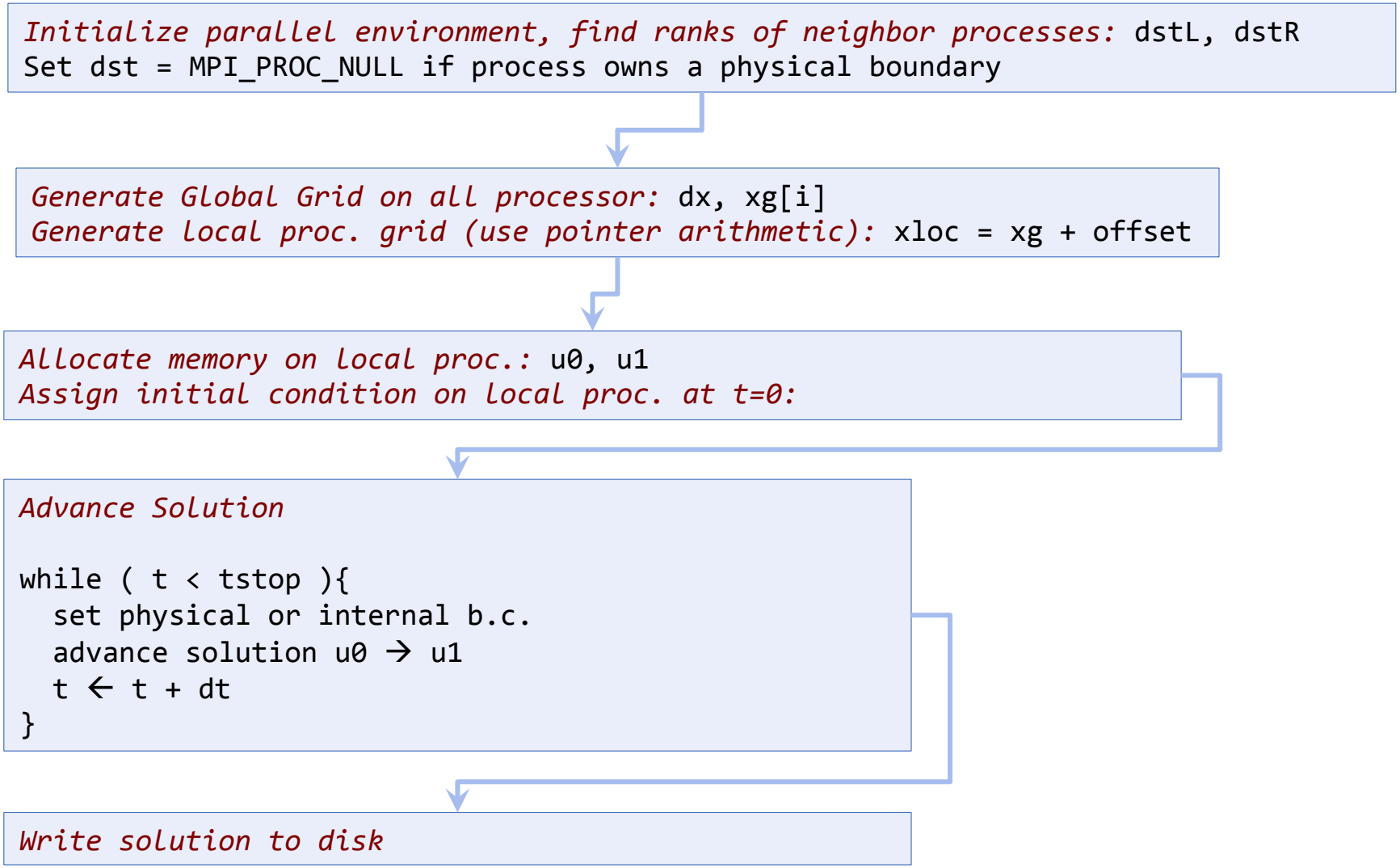
- Let's take a closer look at the inter-processor b.c.:



- Process ***#p*** sends "C" to proc ***#(p+1)*** and receives "A" from proc ***#(p-1)***.
- Process ***#p*** sends "B" to proc ***#(p-1)*** and receives "A" from proc ***#(p+1)***.
- At a physical b.c., no communication should take place unless periodicity is invoked.

Parallel Implementation flowchart

- The serial implementation consists of the following 4 steps:



1D Heat Eq: Writing Data in Parallel

- In parallel writing is less straightforward as different processes may access the same file in scrambled order.
- We can devise two possible solutions:
  1. Each process opens a different file (many files will be generated, one per process at each time step we wish to output);
  2. Process #0 gathers data from other processes and does the write (less files file will be generated, but communications are required):

```
int nx_loc = end - beg + 1;   // Local grid size
static double *recv_buf;
// Allocate memory for the recv buffer (this should be large enough to contain all data points)
if (recv_buf == NULL) recv_buf = (double *) malloc((NX_GLOB + 2*NGHOST)*sizeof(double));

MPI_Gather (u + beg, nx_loc, MPI_DOUBLE, recv_buf + beg, nx_loc, MPI_DOUBLE, 0, MPI_COMM_WORLD);
if (rank == 0){   // rank #0 does the writing
  sprintf (fname,"heat_eq%02d.dat",n);
  fp = fopen (fname,"w");
  for (i = beg; i < beg+NX_GLOB; i++) fprintf (fp, "%f  %f\n", x[i], recv_buf[i]);
  fclose(fp);
}
```

- A more sophisticated approach will be discussed later on, in this course.