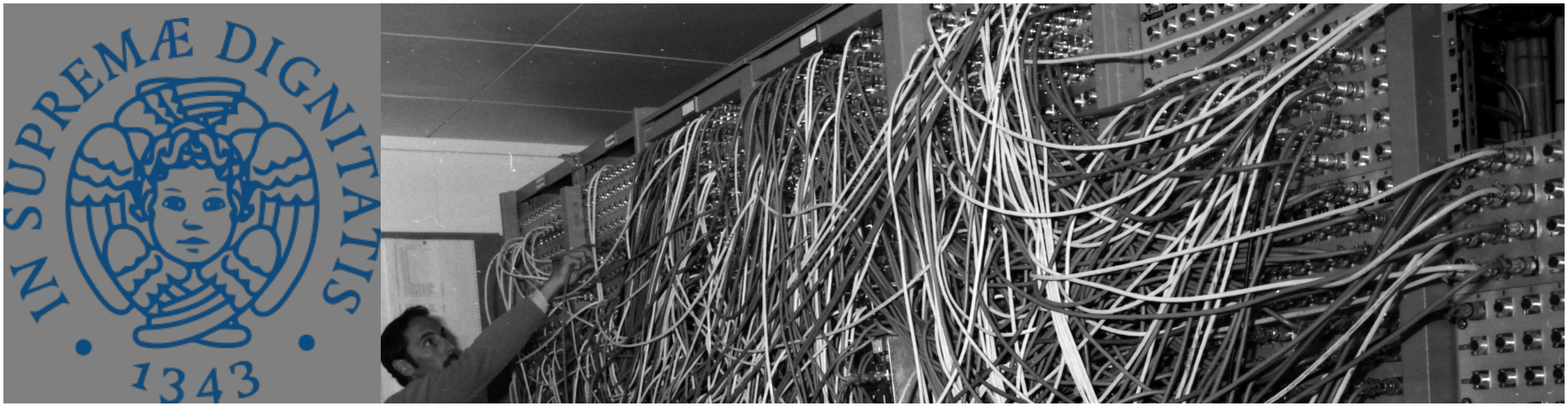


Calcolo Parallelo : Lezione 2

Fabio Durastante

Dipartimento di Matematica, Università di Pisa

Master in Scienze e Tecnologie Spaziali, 2021

1. A First Scientific Computation

2. Collective Communications
   - Broadcast, Gather and Scatter
   - Modifying the 1^{st} derivative code
   - All-to-All Scatter/Gather
   - Global reduce operation

3. Some computations using collective communications
   - Computing Integrals
   - Random number generation: Montecarlo type algorithms

4. Timers and Synchronization

The blocking send and receive

```
int MPI_Send(void *message, int count,
MPI_Datatype datatype, int dest, int tag,
MPI_Comm comm)
```

`void *message` points to the message content itself, it can be a simple scalar or a group of data,

`int count` specifies the number of data elements of which the message is composed,

`MPI_Datatype datatype` indicates the data type of the elements that make up the message,

`int dest` the rank of the destination process,

`int tag` the user-defined tag field,

`MPI_Comm comm` the communicator in which the source and destination processes reside and for which their respective ranks are defined.

The blocking send and receive

```
int MPI_Recv (void *message, int count,
MPI_Datatype datatype, int source, int tag,
MPI_Comm comm, MPI_Status *status)
```

`void *message` points to the message content itself, it can be a simple scalar or a group of data,

`int count` specifies the number of data elements of which the message is composed,

`MPI_Datatype datatype` indicates the data type of the elements that make up the message,

`int source` the rank of the source process,

`int tag` the user-defined tag field,

`MPI_Comm comm` the communicator in which the source and destination processes reside,

`MPI_Status *status` is a structure that contains three fields named `MPI_SOURCE` , `MPI_TAG`, and `MPI_ERROR`.

Basic MPI Data Types

Of the inputs in the previous slides the only one that is specific to MPI is the `MPI_Datatype`, these corresponds to a C data type

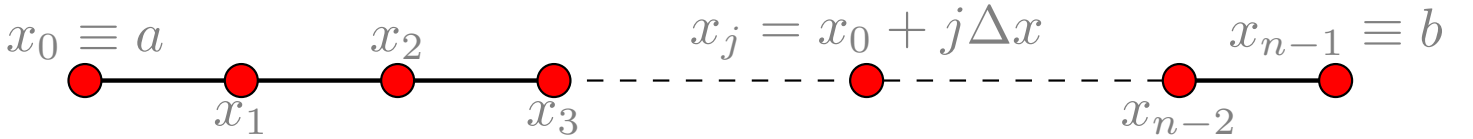
`MPI_CHAR`	`signed char`
`MPI_SHORT`	`signed short int`
`MPI_INT`	`signed int`
`MPI_LONG`	`signed long int`
`MPI_FLOAT`	`float`
`MPI_DOUBLE`	`double`
`MPI_LONG_DOUBLE`	`long double`
`MPI_UNSIGNED_CHAR`	`unsigned char`
`MPI_UNSIGNED_SHORT`	`unsigned short int`
`MPI_UNSIGNED`	`unsigned int`
`MPI_UNSIGNED_LONG`	`unsigned long int`

Note: we will see in the last lecture how to `send`/`receive` user–defined data structures.

The 1st derivative of a function with finite differences

Given a function $f(x) : [a, b] \to \mathbb{R}$ we want to approximate $f'(x)$ on a (uniform) grid on the $[a, b]$ interval by using a finite difference scheme in parallel.

- Given an integer $n \in \mathbb{N}$ we can subdivide the interval $[a, b]$ into intervals of length $\Delta x = {}^{(b-a)}/_{n-1}$ with grid points $\{x_j\}_{j=0}^{n} = \{x_j = a + j\Delta x\}_{j=0}^{n-1}$:



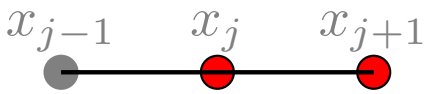
,

- and consider the values $\{f_j\}_{j=0}^{n-1} = \{f(x_j)\}_{j=0}^{n-1}$
- We can approximate the values of $f'(x_j)$, for $j = 1, \ldots, n-2$, by using only the values of f at the knots $\{f_j\}_{j=0}^{n-1}$

The 1st derivative of a function with finite differences

- The first derivative of f at $x = x_j$ can be expressed by using knots for $j' > j$

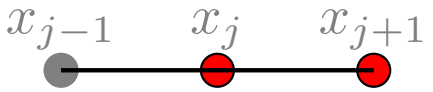
$$f'(x_j) \triangleq \lim_{\Delta x \to 0} \frac{f_{j+1} - f_j}{\Delta x} \approx \frac{f_{j+1} - f_j}{\Delta x} \triangleq D_+ f_j,$$



The 1st derivative of a function with finite differences

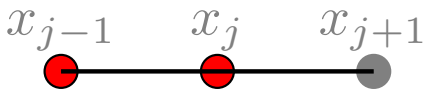
- The first derivative of f at $x = x_j$ can be expressed by using knots for $j' > j$

$$f'(x_j) \triangleq \lim_{\Delta x \to 0} \frac{f_{j+1} - f_j}{\Delta x} \approx \frac{f_{j+1} - f_j}{\Delta x} \triangleq D_+ f_j,$$



- or equivalently by using knots for $j' < j$

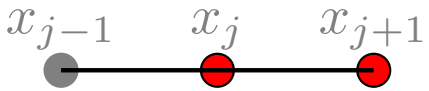
$$f'(x_j) \triangleq \lim_{\Delta x \to 0} \frac{f_j - f_{j-1}}{\Delta x} \approx \frac{f_j - f_{j-1}}{\Delta x} \triangleq D_- f_j,$$



The 1st derivative of a function with finite differences

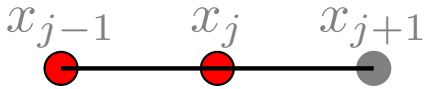
- The first derivative of f at $x = x_j$ can be expressed by using knots for $j' > j$

$$f'(x_j) \triangleq \lim_{\Delta x \to 0} \frac{f_{j+1} - f_j}{\Delta x} \approx \frac{f_{j+1} - f_j}{\Delta x} \triangleq D_+ f_j,$$



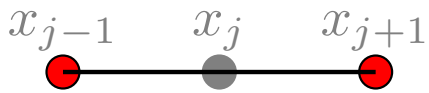
- or equivalently by using knots for $j' < j$

$$f'(x_j) \triangleq \lim_{\Delta x \to 0} \frac{f_j - f_{j-1}}{\Delta x} \approx \frac{f_j - f_{j-1}}{\Delta x} \triangleq D_- f_j,$$



- at last we can consider the arithmetic mean of previous two:

$$f'(x_j) \approx D_0 f_j \triangleq \frac{1}{2}(D_- f_j + D_+ f_j) = \frac{f_{j+1} - f_{j-1}}{2\Delta x},$$



Writing the sequential algorithm

The sequential algorithms needs to break the approximation process into three parts

1. evaluate the derivative $f'(x_i)$ for $i = 1, \ldots, n-2$,
2. evaluate the derivative at the left–hand side $f'(x_0)$,
3. evaluate the derivative at the right–hand side $f'(x_{n-1})$.

To have the same *order of approximation* at each point of the grid we need to use a one–sided formula for the steps 2. and 3., specifically

$$f'(x_0) \approx \frac{-3f_0 + 4f_1 - f_2}{2\Delta x}, \quad f'(x_{n-1}) \approx \frac{3f_{n-1} - 4f_{n-2} + f_{n-3}}{2\Delta x}$$

Writing the sequential algorithm

Then the sequential algorithm can be written as

```
void firstderiv1D_vec(int n, double dx, double *f, double *fx){
double scale;
scale = 1.0/(2.0*dx);
for (int i = 1; i < n-1; i++){
fx[i] = (f[i+1] - f[i-1])*scale;
}
fx[0] = (-3.0*f[0] + 4.0*f[1] - f[2])*scale;
fx[n-1] = (3.0*f[n-1] - 4.0*f[n-2] + f[n-3])*scale;
return;
}
```

The function takes as input

- the number of grid points is n,
- the amplitude of such intervals Δx,
- the array containing the evaluation of f (intent: input),
- the array that will contain the value of the derivative (intent: output)

Writing the parallel algorithm

To implement the sequential differencing functions in parallel with MPI, we have to perform several steps

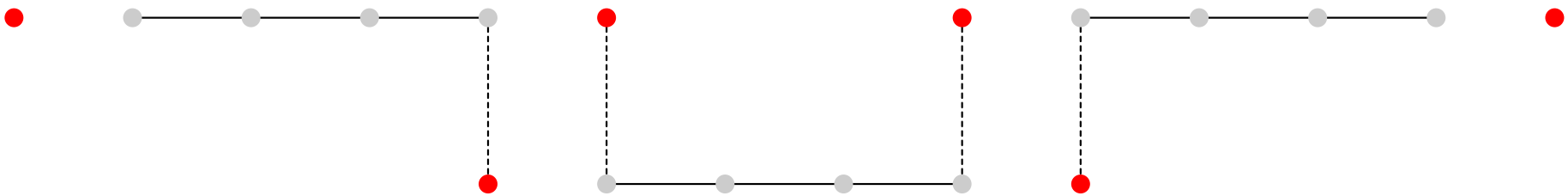
1. partition our domain $[a, b]$ among the processors,
2. each processor then computes the finite differences for all the points contained on that processor

Writing the parallel algorithm

To implement the sequential differencing functions in parallel with MPI, we have to perform several steps

1. partition our domain $[a, b]$ among the processors,
2. each processor then computes the finite differences for all the points contained on that processor

To actually perform the second step, we need to observe that the end-points on each subdomain needs information that is not contained on the processor, but that resides on a different one, we need to communicate boundary data!



Red dots are *halo* data, the one we need to communicate, while gray dots are data owned by the process.

Writing the parallel algorithm

The prototype of the function we want to write can be, in this case,

```
void firstderiv1Dp_vec(int n, double dx, double *f,
double *fx, int mynode, int totalnodes)
```

where

- `int n` is the number of points per process,
- `double dx` the amplitude of each interval,
- `double *f, double *fx` the local portions with the values of $f(x)$ (input) and $f'(x)$ (output),
- `int mynode` the rank of the current process,
- `int totalnodes` the size of the communicator

We declare then the variables

```
double scale = 1.0/(2.0*dx);
double mpitemp;
MPI_Status status;
```

Writing the parallel algorithm

Then we can treat the case in which we are at the beginning or at the end of the global interval

```
if(mynode == 0){
fx[0] = (-3.0*f[0] + 4.0*f[1] - f[2])*scale;
}
if(mynode == (totalnodes-1)){
fx[n-1] = (3.0*f[n-1] - 4.0*f[n-2] + f[n-3])*scale;
}
```

this approximate the derivative at the first and last point of the global interval.

Writing the parallel algorithm

Then we can treat the case in which we are at the beginning or at the end of the global interval

```
if(mynode == 0){
fx[0] = (-3.0*f[0] + 4.0*f[1] - f[2])*scale;
}
if(mynode == (totalnodes-1)){
fx[n-1] = (3.0*f[n-1] - 4.0*f[n-2] + f[n-3])*scale;
}
```

this approximate the derivative at the first and last point of the global interval.

Then, we can compute the inner part (the gray points) of the local interval by doing:

```
for(int i=1;i<n-1;i++){
fx[i] = (f[i+1]-f[i-1])*scale;
}
```

Writing the parallel algorithm

The other case we need to treat is again the particular case in which we are in the first, or in the last interval. In both cases we have only one communication to perform

```
if(mynode == 0){
mpitemp = f[n-1];
MPI_Send();
MPI_Recv();
fx[n-1] = (mpitemp - f[n-2])*scale;
}
else if(mynode == (totalnodes-1)){
MPI_Recv();
fx[0] = (f[1]-mpitemp)*scale;
mpitemp = f[0];
MPI_Send();
}
```

Writing the parallel algorithm

The other case we need to treat is again the particular case in which we are in the first, or in the last interval. In both cases we have only one communication to perform

```
if(mynode == 0){
mpitemp = f[n-1];
MPI_Send(&mpitemp,1,MPI_DOUBLE,1,1,MPI_COMM_WORLD);
MPI_Recv(&mpitemp,1,MPI_DOUBLE,1,1,MPI_COMM_WORLD,&status);
fx[n-1] = (mpitemp - f[n-2])*scale;
}
else if(mynode == (totalnodes-1)){
MPI_Recv(&mpitemp,1,MPI_DOUBLE,mynode-1,1,MPI_COMM_WORLD,
&status);
fx[0] = (f[1]-mpitemp)*scale;
mpitemp = f[0];
MPI_Send(&mpitemp,1,MPI_DOUBLE,mynode-1,1,MPI_COMM_WORLD);
}
```

Writing the parallel algorithm

Finally, the only remaining case is the one in which we need to communicate both the extremes of the interval

```
else{
MPI_Recv();
fx[0] = (f[1]-mpitemp)*scale;
mpitemp = f[0];
MPI_Send();
mpitemp = f[n-1];
MPI_Send();
MPI_Recv();
fx[n-1] = (mpitemp-f[n-2])*scale;
}
```

Writing the parallel algorithm

Finally, the only remaining case is the one in which we need to communicate both the extremes of the interval

```
else{
MPI_Recv(&mpitemp,1,MPI_DOUBLE,mynode-1,1,MPI_COMM_WORLD,
&status);
fx[0] = (f[1]-mpitemp)*scale;
mpitemp = f[0];
MPI_Send(&mpitemp,1,MPI_DOUBLE,mynode-1,1,MPI_COMM_WORLD);
mpitemp = f[n-1];
MPI_Send(&mpitemp,1,MPI_DOUBLE,mynode+1,1,MPI_COMM_WORLD);
MPI_Recv(&mpitemp,1,MPI_DOUBLE,mynode+1,1,MPI_COMM_WORLD,
&status);
fx[n-1] = (mpitemp-f[n-2])*scale;
}
```

And the routine is complete!

Writing the parallel algorithm

A simple (and not very useful) principal program for this routine can be written by first initializing the parallel environment, and discovering who we are.

```
MPI_Init( &argc, &argv );
MPI_Comm_rank( MPI_COMM_WORLD, &mynode );
MPI_Comm_size( MPI_COMM_WORLD, &totalnodes );
```

Then we build the local values of the f function

```
globala = 0;
globalb = 1;
a = globala + ((double) mynode)*(globalb - globala)
/( (double) totalnodes);
b = globala + ((double) mynode+1)*(globalb - globala)
/( (double) totalnodes);
f  = (double *) malloc(sizeof(double)*(n));
fx = (double *) malloc(sizeof(double)*(n));
dx = (b-a)/((double) n);
for( int i = 0; i < n; i++){
f[i] = fun(a+((double) i)*dx);
}
```

Finally we invoke our parallel computation

```
firstderiv1Dp_vec( n, dx, f, fx, mynode, totalnodes);
```

Writing the parallel algorithm

To check if what we have done makes sens we evaluate the error in the $\|\cdot\|_2$ norm on the grid, i.e., $\sqrt{\Delta x}\|\mathbf{f}' - \mathbf{fx}\|_2$ on every process

```
error = 0.0;
for(int i = 0; i < n; i++){
error += pow( fx[i]-funprime(a+((b-a)*((double) i))
/((double) n)),2.0);
}
error = sqrt(dx*error);
printf("Node %d ||f' - fx||_2 = %e\n",mynode,error);
```

Then we clear the memory and close the parallel environment

```
free(f);
free(fx);
MPI_Finalize();
```

Collective Communications

A collective communication is a communication that involves a group (or groups) of processes.

- the group of processes is represented as always as a communicator that provides a context for the operation,
- Syntax and semantics of the collective operations are consistent with the syntax and semantics of the point-to-point operations,
- For collective operations, the amount of data sent must exactly match the amount of data specified by the receiver.

Collective Communications

A collective communication is a communication that involves a group (or groups) of processes.

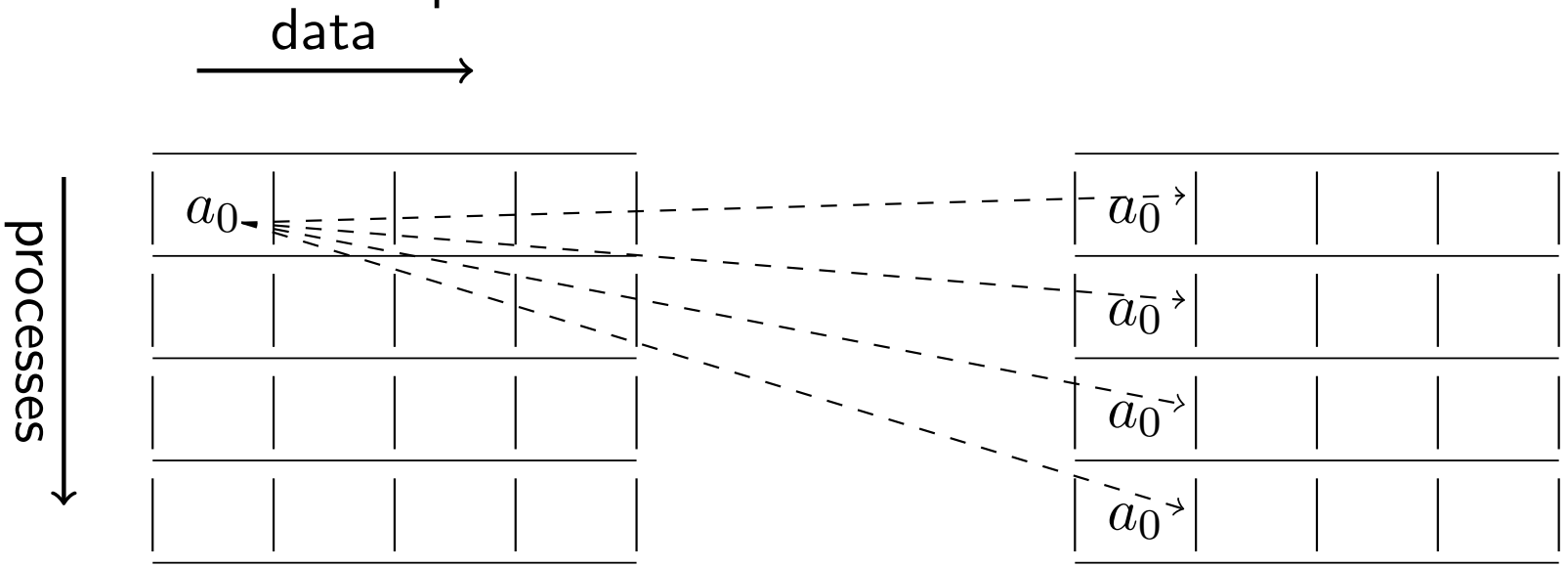
- the group of processes is represented as always as a communicator that provides a context for the operation,
- Syntax and semantics of the collective operations are consistent with the syntax and semantics of the point-to-point operations,
- For collective operations, the amount of data sent must exactly match the amount of data specified by the receiver.

Mixing type of calls

Collective communication calls may use the same communicators as point-to-point communication; Any (conforming) implementation of MPI messages guarantees that calls generated on behalf of collective communication calls will not be confused with messages generated by point-to-point communication.

Taxonomy of collective communications

- The broadcast operation



In the broadcast, initially just the first process contains the data a_0, but after the broadcast all processes contain it.

- This is an example of a **one-to-all** communication, i.e., only one process contributes to the result, while all processes receive the result.

Taxonomy of collective communications: Broadcast

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

Broadcasts a message from the process with rank `root` to all processes of the group, itself included.

`void* buffer` on return, the content of root's buffer is copied to all other processes.

`int count` size of the message

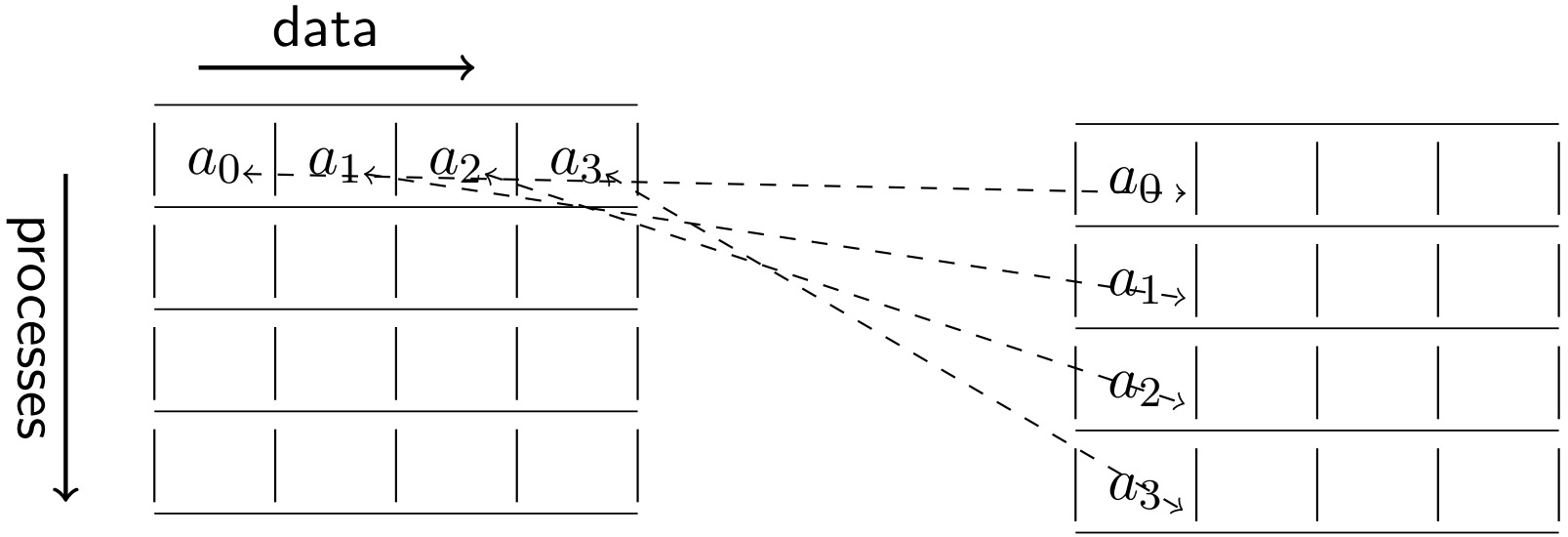
`MPI_Datatype datatype` type of the `buffer`

`int root` `rank` of the process broadcasting the message

`MPI_Comm comm` communicator grouping the processes involved in the broadcast operation

Taxonomy of collective communications: Scatter and Gather

- The scatter and gather operations



- In the **scatter**, initially just the first process contains the data $a_0, \ldots, a_3$, but after the **scatter** the jth process contains the a_j data.
- In the **gather**, initially the jth process contains the a_j data, but after the **gather** the first process contains the data $a_0, \ldots, a_3$

Taxonomy of collective communications: Gather

Each process (root process included) sends the contents of its send buffer to the root process. The latter receives the messages and stores them in rank order.

```
int MPI_Gather(const void* sendbuf, int sendcount,
  MPI_Datatype sendtype,  void* recvbuf, int recvcount,
  MPI_Datatype recvtype, int root, MPI_Comm comm)
```

`const void* sendbuf` starting address of send buffer

`int sendcount` number of elements in send buffer

`MPI_Datatype sendtype` data type of send buffer elements

`void* recvbuf` address of receive buffer

`int recvcount` number of elements for any single receive (and not the total number of items!)

`MPI_Datatype recvtype` data type of received buffer elements

`int root` rank of receiving process

`MPI_Comm comm` communicator

Taxonomy of collective communications: Gather

Each process (root process included) sends the contents of its send buffer to the root process. The latter receives the messages and stores them in rank order.

```
int MPI_Gather(const void* sendbuf, int sendcount,
  MPI_Datatype sendtype,  void* recvbuf, int recvcount,
  MPI_Datatype recvtype, int root, MPI_Comm comm)
```

`const void* sendbuf` starting address of send buffer

`int sendcount` number of elements in send buffer

`MPI_Datatype sendtype` data type of send buffer elements

`void* recvbuf` address of receive buffer

`int recvcount` number of elements for any single receive (and not the total number of items!)

`MPI_Datatype recvtype` data type of received buffer elements

`int root` rank of receiving process

`MPI_Comm comm` communicator

These are significant only at `root`!

Taxonomy of collective communications: Gather

Observe that

- The type signature of `sendcount`, `sendtype` on each process must be equal to the type signature of `recvcount`, `recvtype` at all the processes.
- The amount of data sent must be equal to the amount of data received, pairwise between each process and the root.

Taxonomy of collective communications: Gather

Observe that

- The type signature of `sendcount`, `sendtype` on each process must be equal to the type signature of `recvcount`, `recvtype` at all the processes.
- The amount of data sent must be equal to the amount of data received, pairwise between each process and the root.

Therefore, if we need to have a varying count of data from each process, we need to use instead

```
int MPI_Gatherv(const void* sendbuf, int sendcount,
 MPI_Datatype sendtype, void* recvbuf,
 const int recvcounts[], const int displs[],
 MPI_Datatype recvtype, int root, MPI_Comm comm)
```

where

`const int recvcounts[]` is an array (of length group size) containing the number of elements that are received from each process,

`const int displs[]` is an array (of length group size). Entry `i` specifies the displacement relative to `recvbuf` at which to place the incoming data from process `i`.

Taxonomy of collective communications: Gather

If we need to have the result of the *gather* operation on every process involved in the communicator we can use the variant

```
int MPI_Allgather(const void* sendbuf, int sendcount,
 MPI_Datatype sendtype, void* recvbuf, int recvcount,
 MPI_Datatype recvtype, MPI_Comm comm)
```

- All processes in the communicator `comm` receive the result. The block of data sent from the jth process is received by every process and placed in the jth block of the buffer `recvbuf`.
- The type signature associated with `sendcount`, `sendtype`, at a process must be equal to the type signature associated with `recvcount`, `recvtype` at any other process.

Taxonomy of collective communications: Gather

If we need to have the result of the *gather* operation on every process involved in the communicator we can use the variant

```
int MPI_Allgather(const void* sendbuf, int sendcount,
 MPI_Datatype sendtype, void* recvbuf, int recvcount,
 MPI_Datatype recvtype, MPI_Comm comm)
```

- All processes in the communicator `comm` receive the result. The block of data sent from the jth process is received by every process and placed in the jth block of the buffer `recvbuf`.
- The type signature associated with `sendcount`, `sendtype`, at a process must be equal to the type signature associated with `recvcount`, `recvtype` at any other process.

This function has also the version for gathering messages with different sizes:

```
int MPI_Allgatherv(const void* sendbuf, int sendcount,
 MPI_Datatype sendtype, void* recvbuf, const int recvcounts[],
 const int displs[], MPI_Datatype recvtype, MPI_Comm comm)
```

and works in a way analogous to the `MPI_Gatherv`.

Taxonomy of collective communications: Scatter

This is simply the *inverse* operation of `MPI_Gather`

```
int MPI_Scatter(const void* sendbuf, int sendcount,
  MPI_Datatype sendtype, void* recvbuf, int recvcount,
  MPI_Datatype recvtype, int root, MPI_Comm comm)
```

`const void* sendbuf` address of send buffer

`int sendcount` number of elements sent to each process

`MPI_Datatype sendtype` type of send buffer elements

`void* recvbuf` address of receive buffer

`int recvcount` number of elements in receive buffer

`MPI_Datatype recvtype` data type of receive buffer elements

`int root` rank of sending process

`MPI_Comm comm` communicator

Taxonomy of collective communications: Scatter

This is simply the *inverse* operation of `MPI_Gather`

```
int MPI_Scatter(const void* sendbuf, int sendcount,
  MPI_Datatype sendtype, void* recvbuf, int recvcount,
  MPI_Datatype recvtype, int root, MPI_Comm comm)
```

`const void* sendbuf` address of send buffer

`int sendcount` number of elements sent to each process

`MPI_Datatype sendtype` type of send buffer elements

`void* recvbuf` address of receive buffer

`int recvcount` number of elements in receive buffer

`MPI_Datatype recvtype` data type of receive buffer elements

`int root` rank of sending process

`MPI_Comm comm` communicator

This choices are significant only at `root`!

Taxonomy of collective communications: Scatter

Observe that

- The type signature of `sendcount`, `sendtype` on each process must be equal to the type signature of `recvcount`, `recvtype` at the root.
- The amount of data sent must be equal to the amount of data received, pairwise between each process and the root.

Taxonomy of collective communications: Scatter

Observe that

- The type signature of `sendcount`, `sendtype` on each process must be equal to the type signature of `recvcount`, `recvtype` at the root.
- The amount of data sent must be equal to the amount of data received, pairwise between each process and the root.

Therefore, if we need to have a varying count of data from each process, we need to use instead

```
int MPI_Scatterv(const void* sendbuf, const int sendcounts[],
 const int displs[], MPI_Datatype sendtype, void* recvbuf,
 int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

where

`const int sendcounts[]` is an array (of length group size) containing the number of elements that are sent to each process,

`const int displs[]` is an array (of length group size). Entry `i` specifies the displacement relative to `recvbuf` from which to take the outgoing data to process `i`.

Modifying the 1st derivative code

Let us perform the following modification to our first derivative code:

1. Taking from input the number of points to use in each interval,
2. Collecting the whole result on one process and print it on file.

For the first step we use the `MPI_Bcast` function,

```
if(mynode == 0){
 if(argc != 2){
  n = 20;
 }else{
  n = atoi(argv[1]);
 }
}
MPI_Bcast(&n,1,MPI_INT,
 0,MPI_COMM_WORLD);
```

- We read on `rank` 0 the number `n` from command line,
- Then we broadcast it with `MPI_Bcast`, pay attention to the fact that the broadcast operations happens on all the processes!

Modifying the 1st derivative code

Then we *gather* all the derivatives from the various processes and collect them on process 0.

```
if(mynode == 0)
 globalderiv = (double *)
   malloc(sizeof(double)
   *(n*totalnodes));

MPI_Gather(fx,n,MPI_DOUBLE,
  globalderiv,n,MPI_DOUBLE,
  0,MPI_COMM_WORLD);
```

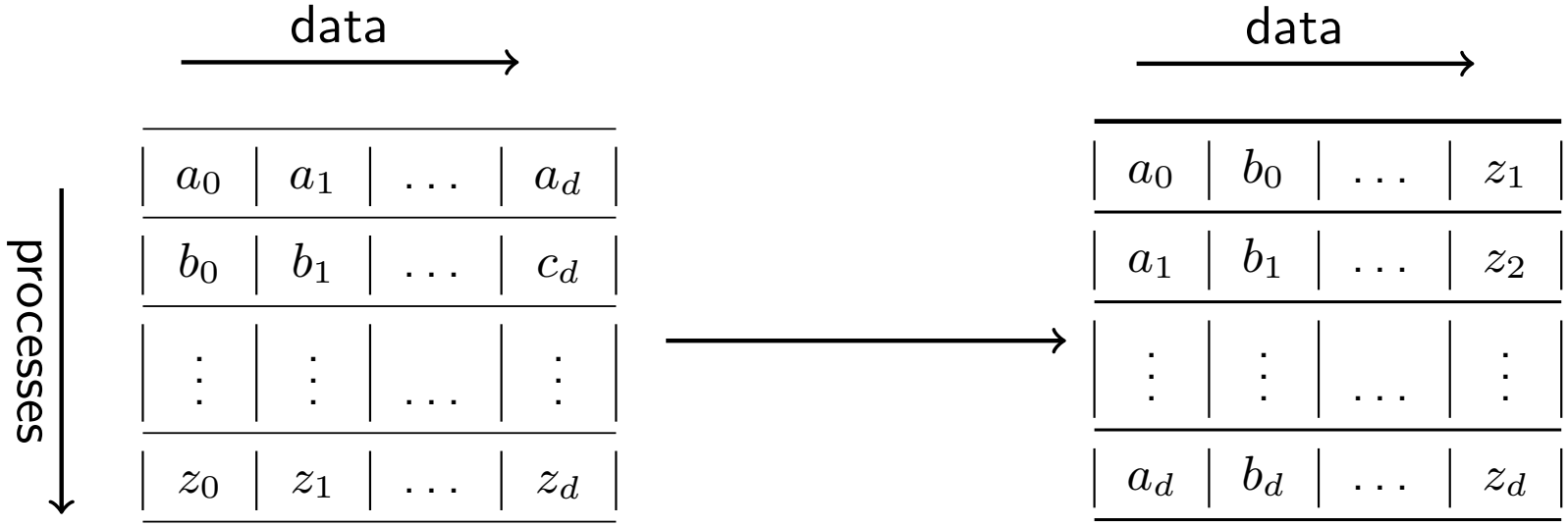
- we allocate on `rank 0` the memory that is necessary to store the whole derivative array,
- then we use the `MPI_Gather` to gather all the array `fx` (of `double`) inside the `globalderiv` array.

At last we print it out on file on `rank 0`

```
if(mynode == 0){
 FILE *fptr; fptr = fopen("derivative", "w");
 for(int i = 0; i < n*totalnodes; i++)
  fprintf(fptr,"%f %f\n",globala+i*dx,globalderiv[i]);
 fclose(fptr); free(globalderiv);}
```

All-to-All

`MPI_ALLTOALL` is an extension of `MPI_ALLGATHER` to the case where each process sends distinct data to each of the receivers.




```
int MPI_Alltoall(const void* sendbuf, int sendcount,
 MPI_Datatype sendtype, void* recvbuf, int recvcount,
 MPI_Datatype recvtype, MPI_Comm comm)
```

- The jth block sent from process i is received by process j and is placed in the ith block of `recvbuf`.
- The type signature for `sendcount`, `sendtype`, at a process must be equal to the type signature for `recvcount`, `recvtype` at any other process.

All-to-All different data size

If we need to send data of different size between the processes

```
int MPI_Alltoallv(const void* sendbuf, const int sendcounts[],
 const int sdispls[], MPI_Datatype sendtype, void* recvbuf,
 const int recvcounts[], const int rdispls[],
 MPI_Datatype recvtype, MPI_Comm comm);
```

`const void* sendbuf` starting address of send buffer

`const int sendcounts[]` array specifying the number of elements to send to each rank

`const int sdispls[]` entry j specifies the displacement (relative to `sendbuf`) from which to take the outgoing data destined for process j

`void* recvbuf` array specifying the number of elements that can be received from each rank

`const int recvcounts[]` integer array. Entry i specifies the displacement (relative to `recvbuf`) at which to place the incoming data from process i

`const int rdispls[]` entry i specifies the displacement (relative to `recvbuf`) at which to place the incoming data from process i

The reduce operation

The reduce operation for a given operator takes a data buffer from each of the processes in the communicator group and combines it according to operator rules.

```
int MPI_Reduce(const void* sendbuf, void* recvbuf,
 int count, MPI_Datatype datatype, MPI_Op op,
 int root, MPI_Comm comm);
```

`const void* sendbuf` address of send buffer

`void* recvbuf` address of receive buffer

`int count` number of elements in send buffer

`MPI_Datatype datatype` data type of elements of send buffer

`MPI_Op op` reduce operation

`int root` rank of root process

`MPI_Comm comm` communicator

The reduce operation

The value of `MPI_Op op` for the reduce operation can be taken from any of the following operators.

`MPI_MAX`	Maximum	`MPI_MAXLOC`	Max value and location
`MPI_MIN`	Minimum	`MPI_MINLOC`	Minimum value and location
`MPI_SUM`	Sum	`MPI_LOR`	Logical or
`MPI_PROD`	Product	`MPI_BOR`	Bit-wise or
`MPI_LAND`	Logical and	`MPI_LXOR`	Logical exclusive or
`MPI_BAND`	Bit-wise and	`MPI_BXOR`	Bit-wise exclusive or

The reduce operation

The value of `MPI_Op op` for the reduce operation can be taken from any of the following operators.

`MPI_MAX`	Maximum	`MPI_MAXLOC`	Max value and location
`MPI_MIN`	Minimum	`MPI_MINLOC`	Minimum value and location
`MPI_SUM`	Sum	`MPI_LOR`	Logical or
`MPI_PROD`	Product	`MPI_BOR`	Bit-wise or
`MPI_LAND`	Logical and	`MPI_LXOR`	Logical exclusive or
`MPI_BAND`	Bit-wise and	`MPI_BXOR`	Bit-wise exclusive or

Moreover, if a different operator is needed, it is possible to create it by means of the function

```
int MPI_Op_create(MPI_User_function* user_fn, int commute,
MPI_Op* op)
```

In C the prototype for a `MPI_User_function` is

```
typedef void MPI_User_function(void* invec, void* inoutvec,
        int *len, MPI_Datatype *datatype);
```

Global reduce operation – All-Reduce

As for other collective operations we may want to have the result of the reduction available on every process in a group.
The routine for obtaining such result is

```
int MPI_Allreduce(const void* sendbuf, void* recvbuf,
 int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

`const void* sendbuf` address of send buffer

`void* recvbuf` address of receive buffer

`int count` number of elements in send buffer

`MPI_Datatype datatype` data type of elements of send buffer

`MPI_Op op` reduce operation

`MPI_Comm comm` communicator

This instruction behaves like a combination of a *reduction* and *broadcast* operation.

Global reduce operation – All-Reduce-Scatter

This is another variant of the reduction operation in which the result is *scattered* to all processes in a group on return.

```
int MPI_Reduce_scatter_block(const void* sendbuf,
 void* recvbuf, int recvcount, MPI_Datatype datatype,
 MPI_Op op, MPI_Comm comm);
```

- The routine is called by all group members using the same arguments for `recvcount`, `datatype`, `op` and `comm`.
- The resulting vector is treated as `n` consecutive blocks of `recvcount` elements that are scattered to the processes of the group `comm`.
- The ith block is sent to process i and stored in the receive buffer defined by `recvbuf`, `recvcount`, and `datatype`.

Global reduce operation – All-Reduce-Scatter

Of this function also a variant with variable block–size is available

```
int MPI_Reduce_scatter(const void* sendbuf, void* recvbuf,
const int recvcounts[], MPI_Datatype datatype, MPI_Op op,
MPI_Comm comm);
```

- This routine first performs a global element-wise reduction on vectors of `count` $= \sum_{i=0}^{n-1}$ `recevcounts[i]` elements in the send buffers defined by `sendbuf`, `count` and `datatype`, using the operation `op`, where `n` is the size of the communicator.
- The routine is called by all group members using the same arguments for `recvcounts`, `datatype`, `op` and `comm`.
- The resulting vector is treated as `n` consecutive blocks where the number of elements of the ith block is `recvcounts[i]`.
- The ith block is sent to process i and stored in the receive buffer defined by `recvbuf`, `recvcounts[i]` and `datatype`.

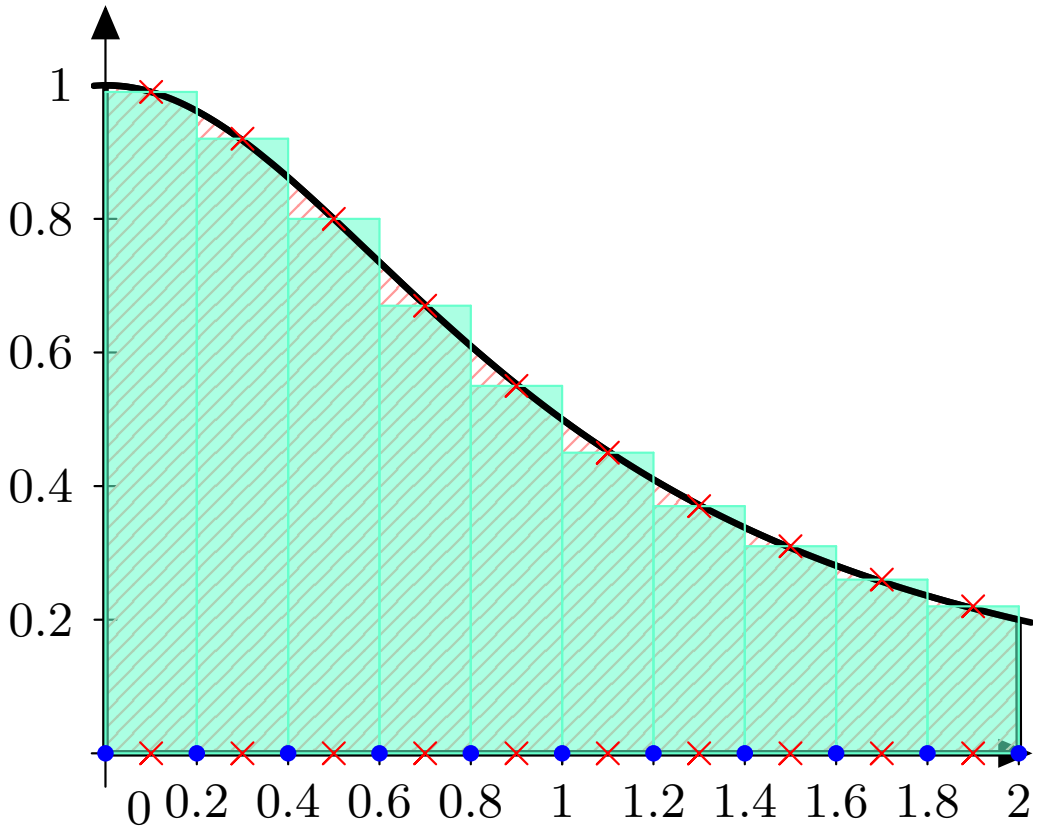
Computing integrals with parallel midpoint quadrature rule

For an integrable function $f : [a, b] \to \mathbb{R}$ the *midpoint* rule (sometimes *rectangle* rule) is given by

$$\int_a^b f(x)dx \approx I_1 = (b-a)f\left(\frac{a+b}{2}\right),$$

This is a very crude approximation, to make it more accurate we may break up the interval $[a, b]$ into a number n of non-overlapping subintervals $[a_k, b_k]$ such that $[a, b] = \cup_k [a_k, b_k]$,

$$I_n = \sum_{k=0}^{n} (b_k - a_k) f\left(\frac{a_k + b_k}{2}\right)$$



Computing integrals with parallel midpoint quadrature rule

If we want to transform this computation in a parallel computation we can adopt the following sketch:

1. `if (mynode == 0)` get number of intervals for quadrature
2. broadcast number of intervals to all the processes
3. assign the non-overlapping intervals to the processes
4. sum function values in the center of each interval
5. reduce with operator sum the integral on process 0.

As a test function for the parallel integration routine we can use

$$f(x) = \frac{4}{1+x^2}; \qquad I = \int_0^1 \frac{4}{1+x^2} dx = \pi.$$

To evaluate the error we can use the value :

```
double PI25DT = 3.14159265358979323846264338;
```

Computing integrals with parallel midpoint quadrature rule

```
h   = 1.0 / ((double) n*totalnodes);
sum = 0.0;
for (i = 1+mynode*n;
         i <= n*(mynode+1);
         i++){
 x = h * ((double)i - 0.5);
 sum += f(x);
}
mypi = h * sum;
MPI_Reduce(&mypi, &pi, 1,
         MPI_DOUBLE,
         MPI_SUM, 0,
         MPI_COMM_WORLD);
```

- We assume that all the intervals have the same size, thus the scaling `h = 1.0 / (double) n`,
- We compute all the value x that are in the local process and increment the local sum,
- in conclusion we perform an `MPI_Reduce` to sum together all the local sums.

You can then print out the obtained value of π and the error with respect to `PI25DT` as

```
if (mynode == 0){
 printf("pi is approximately %.16f, Error is %e\n",
         pi, fabs(pi - PI25DT));
}
```

Computing π Montecarlo Style

Montecarlo methods are algorithms that rely on a procedure of repeated random sampling to obtain numerical results[1].

A generic Montecarlo algorithm can be described by the following 4 steps

1. define a domain of possible samples
2. generate the samples from a probability distribution over such domain
3. perform a deterministic computation on the inputs
4. aggregate the results

[1] For some historical information about this idea: http://shorturl.at/mAWY8

Computing π Montecarlo Style

Montecarlo methods are algorithms that rely on a procedure of repeated random sampling to obtain numerical results[1].

A generic Montecarlo algorithm can be described by the following 4 steps

1. define a domain of possible samples
2. generate the samples from a probability distribution over such domain
3. perform a deterministic computation on the inputs
4. aggregate the results

Let us use it to approximate π

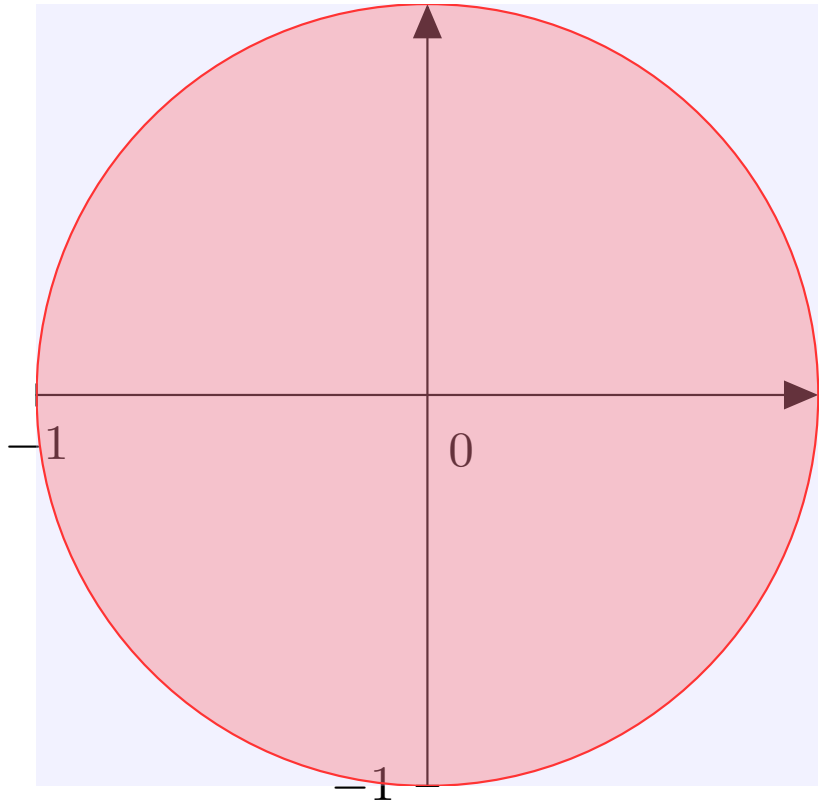
1. we consider the square $S = [-1, 1] \times [-1, 1]$,
2. we generate samples in S from a uniform probability distribution
3. we count the number of points (x, y) that are such that $x^2 + y^2 \leq 1$,
4. the approximation of π is given by the ratio

$$\pi \approx |\{(x,y): x^2+y^2 \leq 1\}| / |\{(x,y) \in S\}|$$

[1] For some historical information about this idea: http://shorturl.at/mAWY8

Computing π Montecarlo Style

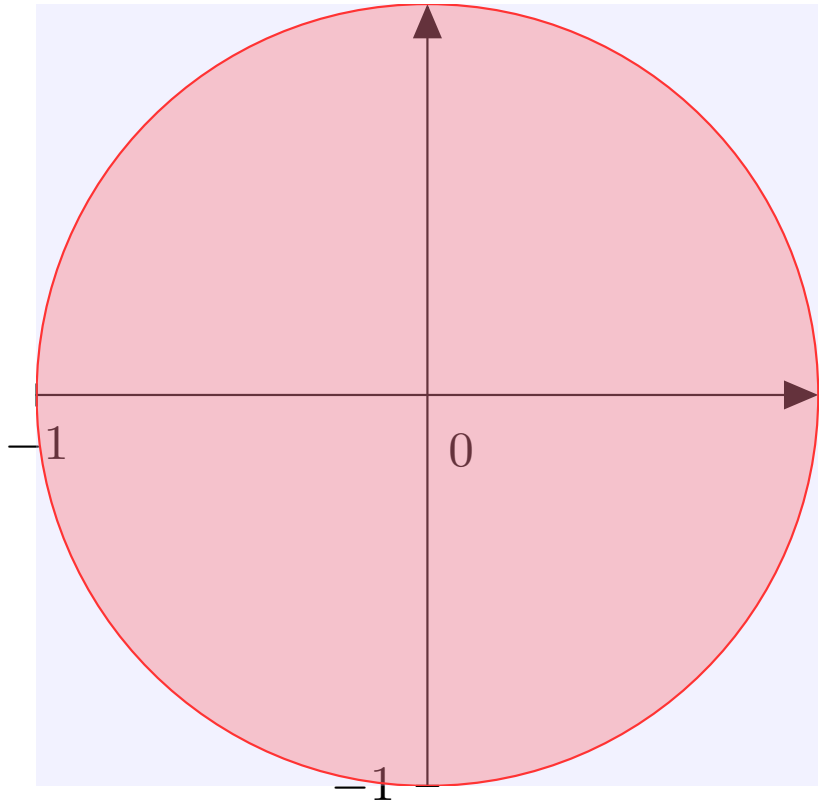
We can write the parallel version of such algorithm in the following way



Computing π Montecarlo Style

We can write the parallel version of such algorithm in the following way

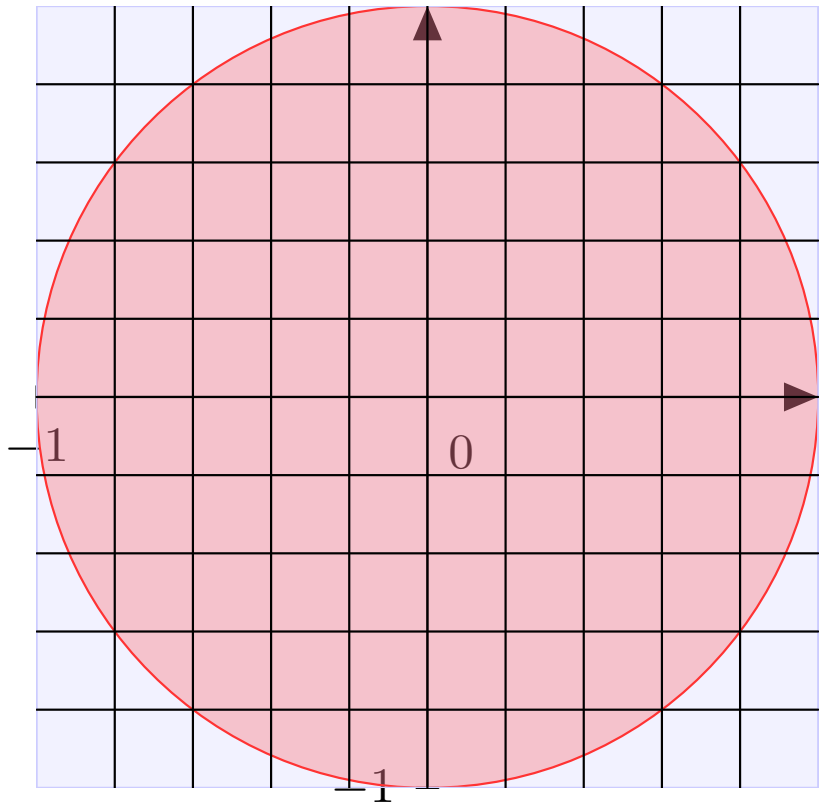
1. we divide a square in an number of parts equal to the number of processes we have,



Computing π Montecarlo Style

We can write the parallel version of such algorithm in the following way

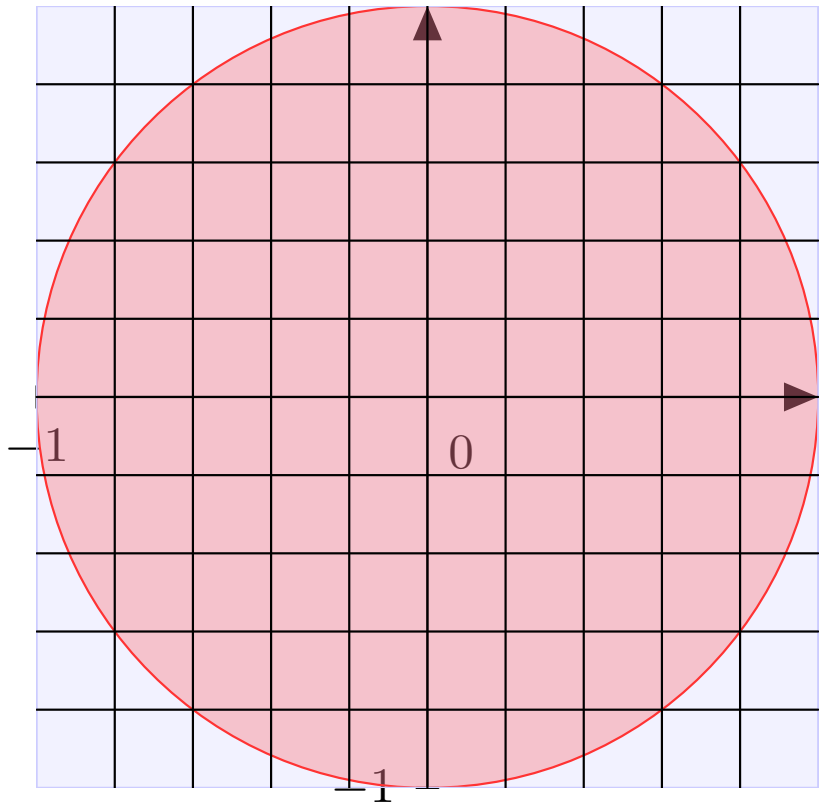
1. we divide a square in an number of parts equal to the number of processes we have,
2. we generate a number of random points (x, y) in the area owned by each process,



Computing π Montecarlo Style

We can write the parallel version of such algorithm in the following way

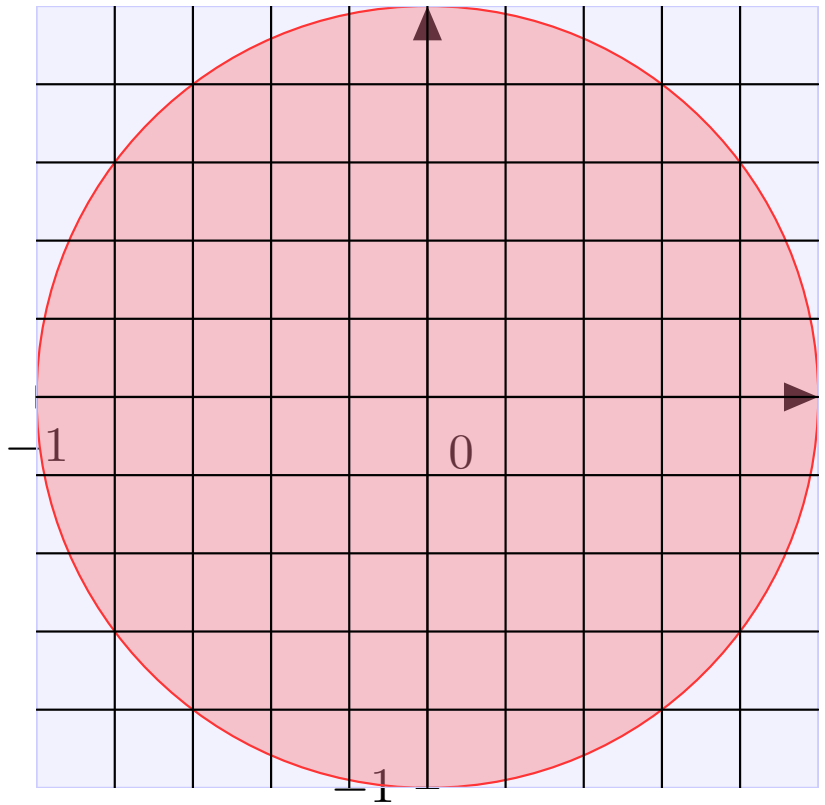
1. we divide a square in an number of parts equal to the number of processes we have,
2. we generate a number of random points (x, y) in the area owned by each process,
3. we compute how many points fall in the circle



Computing π Montecarlo Style

We can write the parallel version of such algorithm in the following way

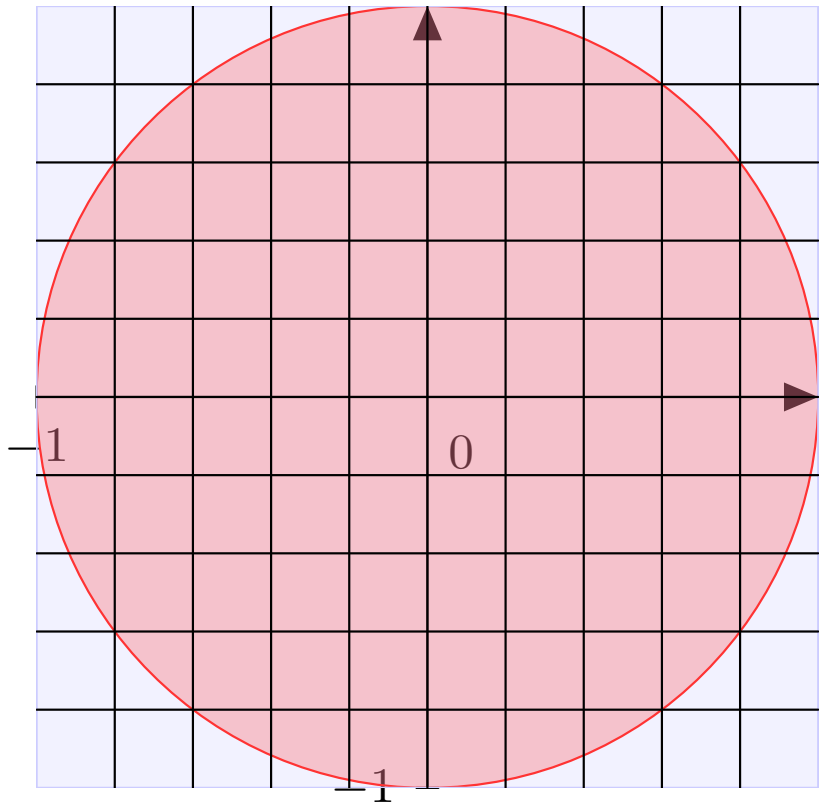
1. we divide a square in an number of parts equal to the number of processes we have,
2. we generate a number of random points (x, y) in the area owned by each process,
3. we compute how many points fall in the circle
4. sum-reduce the number of points in the square and in the circle



Computing π Montecarlo Style

We can write the parallel version of such algorithm in the following way

1. we divide a square in an number of parts equal to the number of processes we have,
2. we generate a number of random points (x, y) in the area owned by each process,
3. we compute how many points fall in the circle
4. sum-reduce the number of points in the square and in the circle
5. divide the two numbers on process 0 to get the approximation



A word of caution

Generating random numbers in a meaningful way requires some care. This is true in general, but it is true in particular in a parallel environment. On each different process we need to be sure of generating independent sequences of random numbers. This can be achieved by using different seeds for the random number generator.
We achieve this by using the `#include <time.h>` and `#include <rand.h>` libraries

```
unsigned int seed = (unsigned) time(NULL);
```

generates an integer on each process depending on the time of the call, then we build a seed by using `srand(seed + mynode)` that guarantees different seeds for the processes and thus independent random numbers:

```
srand(seed + mynode);
x = rand_r(&seed); x = x / RAND_MAX; x = x1 + x * (x2 - x1);
y = rand_r(&seed); y = y / RAND_MAX; y = y1 + y * (y2 - y1);
```

Computing π Montecarlo Style

We can generate on each node the sampling on the reference square by

```
h   = 2.0 / (double) totalnodes;
x1 = -1.0 + mynode * h;
x2 = x1 + h;
y1 = -1.0;
y2 = 1.0;
my_SqPoints  = 0;
my_CiPoints  = 0;

for (i = 1; i <= n; i += totalnodes){
 srand(seed + mynode);
 x = rand_r(&seed); x = x / RAND_MAX; x = x1 + x * (x2 - x1);
 y = rand_r(&seed); y = y / RAND_MAX; y = y1 + y * (y2 - y1);
 my_SqPoints++;
 if ( ( x*x + y*y ) <= 1.0 ) my_CiPoints++;
}
```

Computing π Montecarlo Style

Then we perform the reduction by doing

```
SqPoints = 0;
CiPoints = 0;
MPI_Reduce(&my_SqPoints, &SqPoints, 1, MPI_INT, MPI_SUM, 0,
 MPI_COMM_WORLD);
MPI_Reduce(&my_CiPoints, &CiPoints, 1, MPI_INT, MPI_SUM, 0,
 MPI_COMM_WORLD);
```

and print the approximation

```
if (mynode == 0){
pi = 4.0 * (double)CiPoints / (double)SqPoints;
printf("Pi is approximately %.16f, Error is %e\n"
 ,pi, fabs(pi - PI25DT));
}
```

Timers and Synchronization

- A timer is specified even though it is not an instruction based on "message-passing,": timing parallel programs is important for inquiring on the "performances" of your code.

Timers and Synchronization

- A timer is specified even though it is not an instruction based on "message-passing,": timing parallel programs is important for inquiring on the "performances" of your code.
- the timer returns a floating-point number of seconds, representing elapsed wall-clock time since *some time in the past*:

```
double MPI_Wtime(void);
```

the *time in the past* is guaranteed not to change during the life of the process.

Timers and Synchronization

- A timer is specified even though it is not an instruction based on "message-passing,": timing parallel programs is important for inquiring on the "performances" of your code.
- the timer returns a floating-point number of seconds, representing elapsed wall-clock time since *some time in the past*:

```
double MPI_Wtime(void);
```

  the *time in the past* is guaranteed not to change during the life of the process.
- the usual application of a timer is something of the form:

```
double starttime, endtime;
starttime = MPI_Wtime();
< --- foolish things happen here --- >
endtime = MPI_Wtime();
printf("That took %f seconds\n",endtime-starttime);
```

Timers and Synchronization

- A timer is specified even though it is not an instruction based on "message-passing,": timing parallel programs is important for inquiring on the "performances" of your code.
- the timer returns a floating-point number of seconds, representing elapsed wall-clock time since *some time in the past*:

  ```
  double MPI_Wtime(void);
  ```

  the *time in the past* is guaranteed not to change during the life of the process.
- the usual application of a timer is something of the form:

  ```
  double starttime, endtime;
  starttime = MPI_Wtime();
  < --- foolish things happen here --- >
  endtime = MPI_Wtime();
  printf("That took %f seconds\n",endtime-starttime);
  ```
- There exists a tag `MPI_WTIME_IS_GLOBAL` that is 1 if clocks at all processes in `MPI_COMM_WORLD` are synchronized, 0 otherwise.

Timers and Synchronization

- MPI offers a *barrier* function that blocks the caller until all processes in the communicator have called it

  ```
  int MPI_Barrier(MPI_Comm comm)
  ```

  that is, the call returns at any process only after all members of the communicator have entered the call.

Timers and Synchronization

- MPI offers a *barrier* function that blocks the caller until all processes in the communicator have called it

  ```
  int MPI_Barrier(MPI_Comm comm)
  ```

  that is, the call returns at any process only after all members of the communicator have entered the call.
- It can be used together with the `MPI_Wait` function to force a synchronization point in the program.

Timers and Synchronization

- MPI offers a *barrier* function that blocks the caller until all processes in the communicator have called it

  ```
  int MPI_Barrier(MPI_Comm comm)
  ```

  that is, the call returns at any process only after all members of the communicator have entered the call.
- It can be used together with the `MPI_Wait` function to force a synchronization point in the program.
- It can be used to regulate the access to an external resource (e.g., a file) in such a way that every processor accesses it in an order way: if you are interested in writing file in parallel you can look at Chapter 13 of the MPI guide[2]

[2] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, Version 3.1. https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf, High Performance Computing Center Stuttgart (HLRS).

Evaluating performances

You can use the `MPI_Wtime()` to give a simple evalaution of the performances of your program.

Consider, e.g., the two programs for the computation of the π constant. You can evaluate the weak scalability of your code by looking at the time spent in doing the whole computation for growing size of processor numbers and samples.

We can compute the efficiency of the code by measuring:

$$E = t(1)/t(N) \in [0, 1]$$

where

- $t(1)$ is the amount of time to complete a work unit with 1 processing element,
- $t(N)$ is the amount of time to complete N of the same work units with N processing elements.

Further modifications

For the derivative program:

- In every case the function `void firstderiv1Dp_vec` wants to exchange information between two adjacent processes, i.e., every process wants to "swap" is halo with its adjacent process. We can rewrite the whole function by using the `MPI_Sendrecv_replace` point-to-point communication routine.
- We can rewrite the entire program in an "embarrassing parallel" way, if every process has access to f, and are assuming that all the interval are partitioned the same way, by using the knowledge of our `rank` we can compute what are the boundary elements at the previous and following process. Thus, no communication at all!

For the π programs,

- Make a graph of the timings to evaluate the weak scaling efficiency.

– Try this at home! (Maybe here, if there is still time. . .) –