NTNU – Trondheim
Norwegian University of
Science and Technology

Collective operations

Jan.Christian.Meyer@ntnu.no

Today’s topic

- We’ve looked at how the rank of a process can be used to make it act differently from every other, by branching into statements that only apply to 1 rank
- We’ve also looked at how calculating different arguments based on rank can make the same statement do different things by different ranks
- Today, we’ll look at MPI calls that are specifically made to be used in the second manner

We have already seen collective *code*

- Instead of writing something like

```
if ( rank == 0 )
    for ( int_t i=0; i<N/2; i++ )
        c[i] = a[i] * b[i];
else if ( rank == 1 )
    for ( int_t i=N/2; i<N; i++ )
        c[i] = a[i] * b[i];
```

we can write

```
bounds[2] = { rank * N/2, (rank+1)*N/2 };
for ( int_t i=bounds[0]; i<bounds[1]; i++ )
    c[i] = a[i] * b[i];
```

and make both ranks use the same code for different data.

Collective operations

- MPI's collective operations are function calls that expect this style of program
- All ranks in a communicator must participate in its collective operations
  - The idea is to place it somewhere in the control flow where all the ranks will come through, and make the exact same call
  - You *can* put it inside conditional code as well, but there has to be a copy along every code path
- If fewer than all ranks make the call, it will hang until it times out and crashes

The simplest collective:

```
MPI_Barrier ( MPI_COMM_WORLD );
```

- This does not actually *do* anything
  - It just requires all ranks in WORLD to call it
  - Nobody returns from their call before everyone has made it
- It's a synchronization feature of sorts
  - The ranks won't actually all return from Barrier at exactly the same moment
  - It'll be close, though
  - If one or more ranks were lagging behind, this will definitely bring them up to speed (at the expense of waiting for them)

MPI_Barrier is <u>not</u> a memory fence

- A *memory fence* is an operation that forces all committed work to be completed before continuing
- MPI_Barrier does no such thing
  - It just makes everyone exchange some empty messages to check on each other's progress
  - If you have background messages in transit, they may still be in transit after the barrier
  - If you have written data that is waiting in a buffer, barrier will not flush it
  - If you have pending requests for work, barrier will not clean them up, you still have to wait for their completion to finalize them

What’s it for, then?

- It can help a lot with instrumenting your program’s performance
  - More on that in a minute
- It can help a little with debugging
  - You can use it to guarantee that everyone has reached a particular point in the program (as long as you remember what that means)
- I have not seen a program that depended on a barrier in order to get the right answer
  - If you make one, you’re probably inventing something strange

Broadcast

- Here's a more interesting collective:

  int MPI_Bcast (

  void *buffer,

  int count,

  MPI_Datatype datatype,

  int root,

  MPI_Comm communicator

  );
- You'll recognize the first three arguments, they're just like for Send and Recv
- The last one isn't surprising either

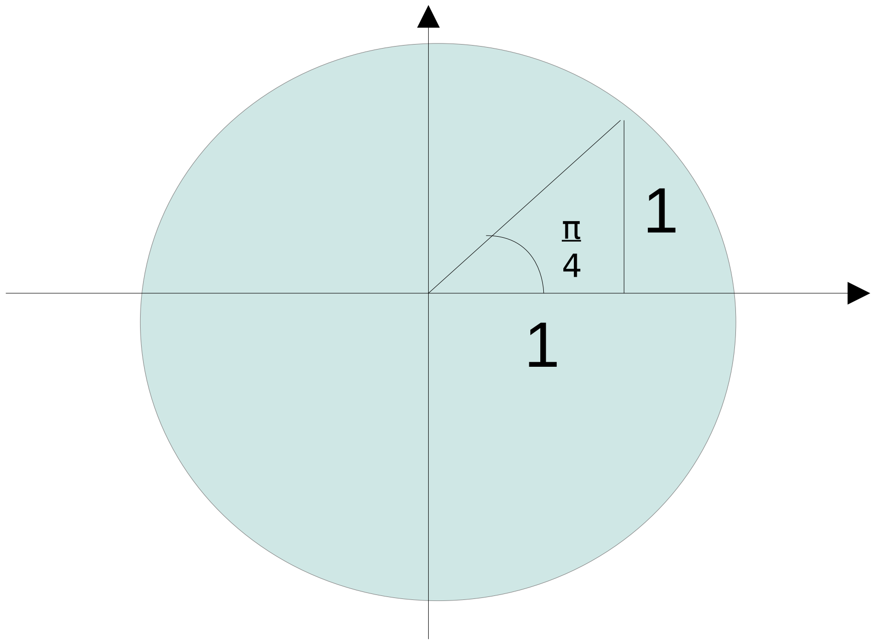
MPI_Bcast is a *rooted* collective

- The *root* argument designates a rank that acts as the 'master' rank for the operation
- Broadcast, as the name implies, takes data from one rank and gives it to everyone
  - On the root rank, the memory buffer will be read and transmitted
  - On all the other ranks, data will be received and written into the memory buffer

(by contrast, MPI_Barrier has no root rank, everybody's equal)

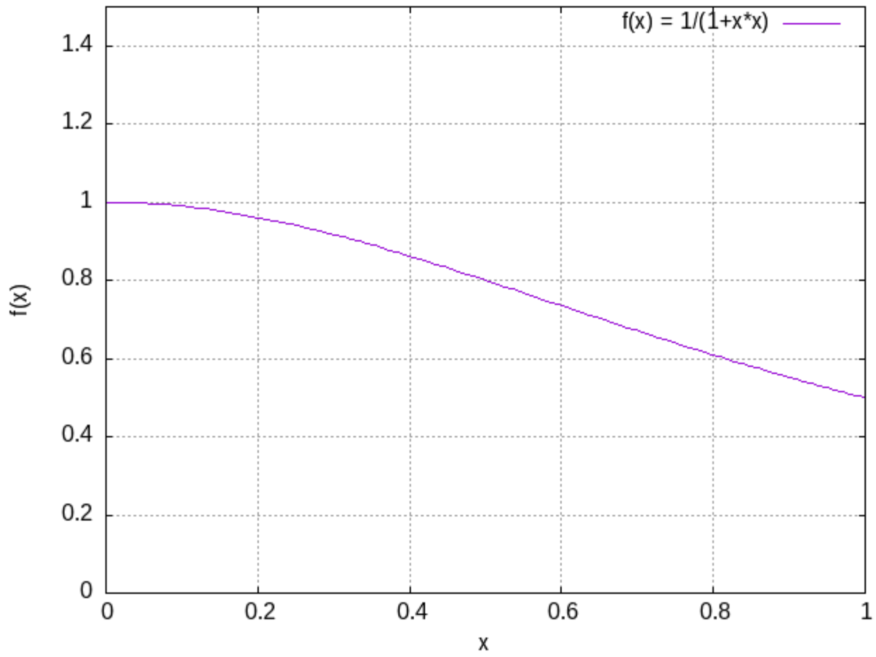
Let’s do a global sum

- We can calculate something simple without the complexities of neighbor points, border exchanges, boundary conditions, *etc.*
- The arctangent of 1 is pi divided by 4:



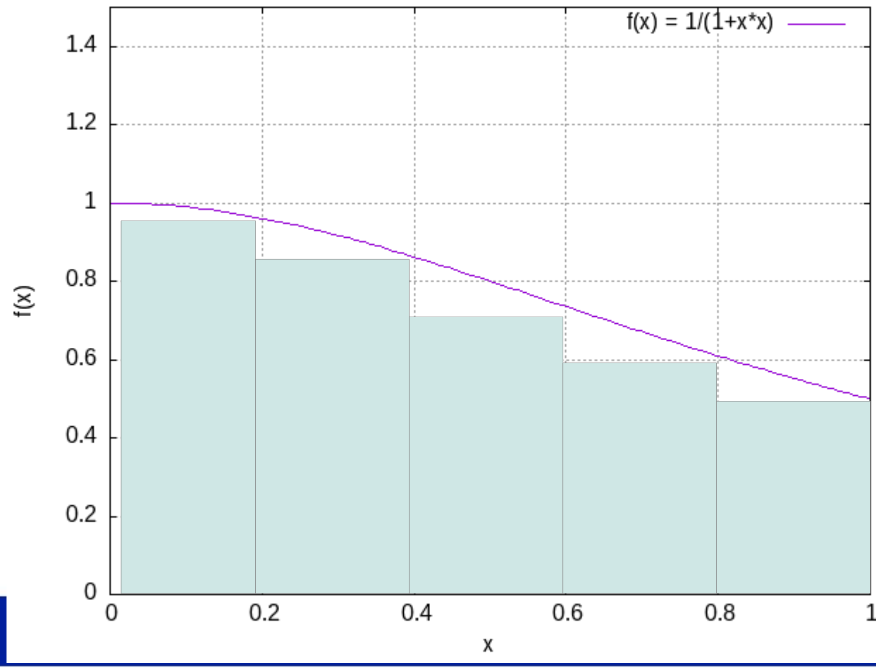
The derivative of arctan(x) is $1/(1+x^2)$

- Here we have it, from x=0 to x=1
- The area between this curve and the x-axis is π/4



We have an integration engine

- We can estimate the area under the curve with a bunch of rectangles
  - They can have width h
  - Their height will be $1/(1+x^2)$ at the end of the interval
  - That gives us the area



NTNU – Trondheim
Norwegian University of Science and Technology

Tales from the code archive

- estimate_pi.c does what we did with the file saving issue in last lesson's parallel advection eq. solver:
  - Rank 0 includes its own partial result first
  - Rank 0 then waits for messages from all the rest, in order
  - All other ranks send their partial results to 0
- It works, but
  - It's long-winded and error prone to write
  - It forces a sequence upon the reception of messages
  - Adding up a global sum is a common thing to do, so we can use a collective operation instead

Rooted reductions

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

- The send-buffer, count, datatype, and communicator are like the MPI_Send arguments, and point out the data that each rank will contribute to the total

NTNU – Trondheim
Norwegian University of Science and Technology

Rooted reductions

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

- The recv-buffer only has to exist at the *root* rank, it is where the total of all contributions will be placed
- It won't be used on the other ranks, you can make it NULL there if you wish

Rooted reductions

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

- The MPI_Op is the name of an operation that can be applied to combine the contributions from arbitrary pairs of ranks
- There’s a list of them, including MPI_SUM, MPI_PROD, MPI_MAX, MPI_BAND (‘bitwise and’), and so on...
- The main thing is that they have to be commutative

The Pi example with reduction

- estimate_pi_reduction.c replaces our point-to-point construct with a collective op. that takes a single line of code
- There is also an unrooted MPI_Allreduce
- It's the same as Reduce, except that
  - There's no *root* argument
  - recv-buffer has to be allocated on all participants, because everyone gets a copy of the result
- estimate_pi_allreduce.c uses that instead

There are quite a few collectives

- Scatter ← partition data into equal-size chunks
  - Scatterv ← or chunks of individual, different sizes
- Gather ← collect equal-size chunks into a whole
  - Gatherv ← or chunks of individual, different sizes
- Scan ← Accumulate intermediate parts
- Allgather ← Gather by everyone
- Alltoall ← Total exchange (from everyone to everyone)
  - Alltoallv ← also available with different-sized chunks

- Some include computations along the way, others are just data movement

There's another 6-function MPI

- Everything MPI can do can also be implemented using selected collectives
  - MPI_Init
  - MPI_Finalize
  - MPI_Comm_rank
  - MPI_Comm_size
  - MPI_Bcast
  - MPI_Reduce
- The example we didn't implement last time where all ranks have large, same-size allocations does the job
  - Point-to-point messages can be done by designating an area per process, and reducing the entire global array every so often

Collectives hide different complexities

- Some are terribly expensive
- Others are not so expensive
- We’ll look at estimating their cost next time
  - It’s not an accurate science, but ballpark estimates are already useful