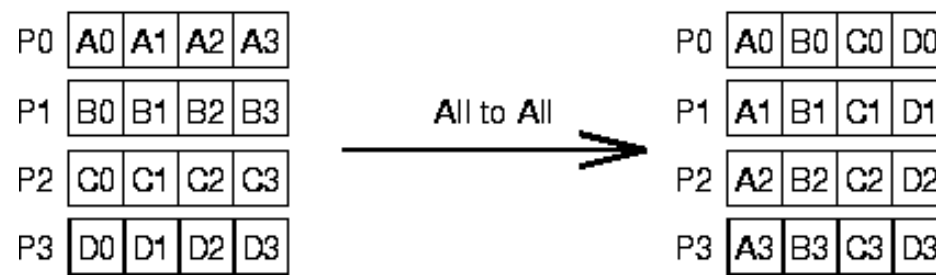
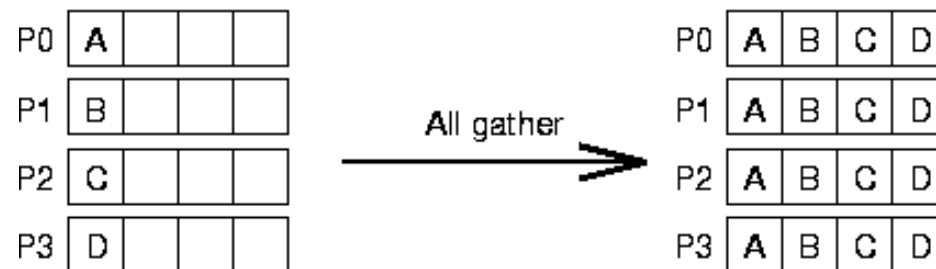
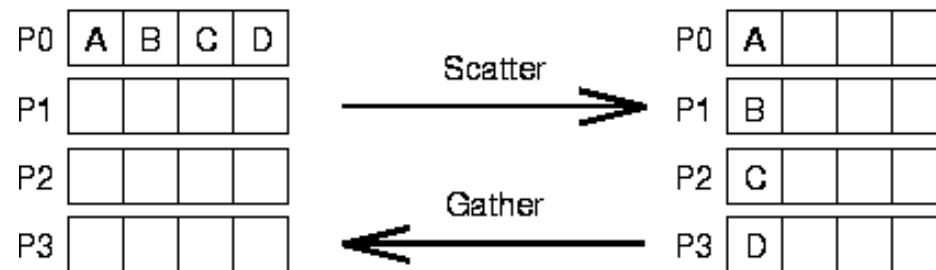
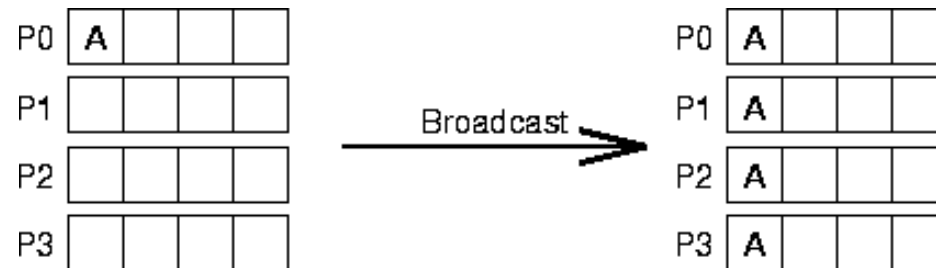


Introduction to High Performance Scientific Computing

Autumn, 2016

Lecture 16

Last time: MPI collective data movement



Today

- **Coarse grain parallelization**
- **Send and Receive**
- **Domain Decomposition**
- **Using** gather **and** gatherv

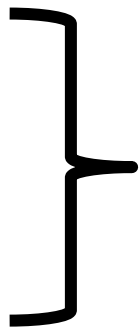
Today

- **Coarse grain parallelization**
- **Send and Receive**
- **Domain Decomposition**
- **Using** gather **and** gatherv

Part of course project

Today's lecture

Labs 8 (task 2)



All related

Coarse-grain vs fine-grain parallelism

- With OpenMP, we've used fine-grain approach
 - Look for a code segment (e.g. a loop) that can be parallelized
 - Let OpenMP do the rest (just for that segment)

Coarse-grain vs fine-grain parallelism

- With OpenMP, we've used fine-grain approach
 - Look for a code segment (e.g. a loop) that can be parallelized
 - Let OpenMP do the rest (just for that segment)
- With MPI, typically take a coarse-grain approach
 - At beginning of simulation, distribute data and tasks to processes
 - Each process works on its own problem
 - Occasionally communicating when necessary
- Can also use coarse-grain approach in OpenMP! (Project part 3.2)

Coarse-grain approach

- We have already seen a “sort-of” coarse grain approach with quadrature:

```
!set number of intervals per processor
```

```
  Nper_proc = (N + numprocs - 1)/numprocs
```

```
!starting and ending points for processor
```

```
  istart = myid * Nper_proc + 1
```

```
  iend = (myid+1) * Nper_proc
```

```
  if (iend>N) iend = N
```

Coarse-grain approach

- We have already seen a “sort-of” coarse grain approach with quadrature:

```
!set number of intervals per processor
```

```
  Nper_proc = (N + numprocs - 1)/numprocs
```

```
!starting and ending points for processor
```

```
  istart = myid * Nper_proc + 1
```

```
  iend = (myid+1) * Nper_proc
```

```
  if (iend>N) iend = N
```

```
!loop over intervals computing each interval's contribution to  
integral
```

```
  do i1 = istart,iend
```

```
    xm = dx*(i1-0.5) !midpoint of interval i1
```

```
    call integrand(xm,f)
```

```
    sum_i = dx*f
```

```
    sum_proc = sum_proc + sum_i !add contribution from interval  
to total integral
```

```
  end do
```


Coarse-grain approach

- More generally, at start of program we will:

1. Obtain *myid* and total number of processes, *numprocs*:

```
call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)
```

2. Use this information to distribute *Ntotal* points (or pieces of data) across *numprocs* processors

Coarse-grain approach

- More generally, at start of program we will:

1. Obtain *myid* and total number of processes, *numprocs*:

```
call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)
```

2. Use this information to distribute *Ntotal* points (or pieces of data) across *numprocs* processors

3. We will use a simple fortran subroutine, *MPE_DECOMP1D*:

```
call MPE_DECOMP1D( Ntotal, numprocs, myid, istart, iend)
```

```
Nlocal = iend - istart + 1
```

- Simple subroutine which assigns *istart* and *iend* to each process
- If *Ntotal*=100, *numprocs* = 2:
 - *myid* = 0 → *istart* = 1, *iend* = 50
 - *myid* = 1 → *istart* = 51, *iend* = 100

Complex parallelization

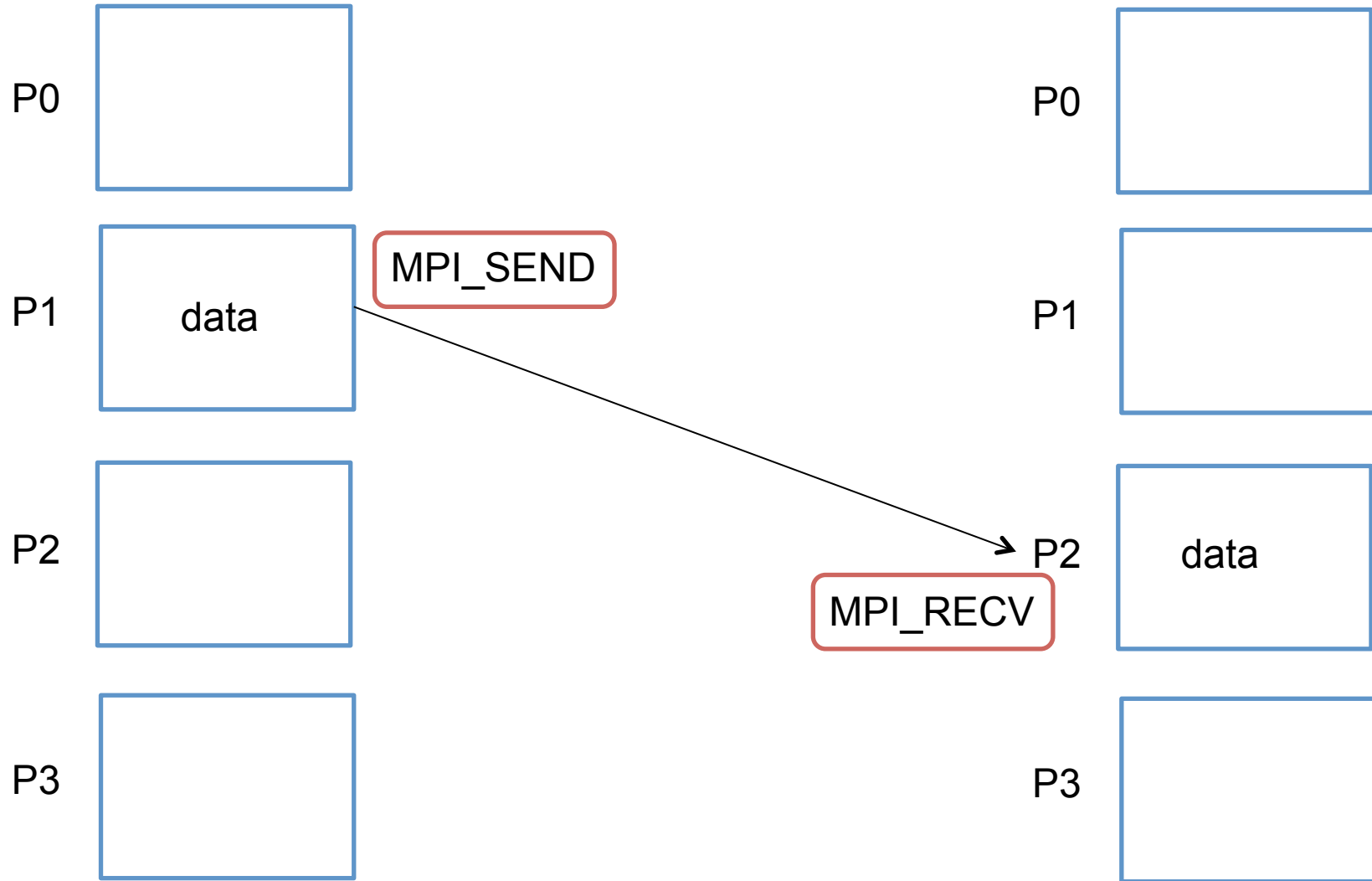
- MPE_DECOMP1D partitions data on a 'line'
- What about more complicated topologies or networks?
 - e.g. simulation of $1e7$ air molecules?
 - Advanced tools exist to do the partitioning for you
 - E.g. *ParMETIS*:

ParMETIS is an MPI-based parallel library that implements a variety of algorithms for partitioning unstructured graphs, meshes, and for computing fill-reducing orderings of sparse matrices. ParMETIS extends the functionality provided by METIS and includes routines that are especially suited for parallel AMR computations and large scale numerical simulations. The algorithms implemented in ParMETIS are based on the parallel multilevel k-way graph-partitioning, adaptive repartitioning, and parallel multi-constrained partitioning schemes developed in our lab.

MPI Send/Recv

- Bcast and Reduce are examples of *collective* communication
- *Point-to-point* communication carried out by send and recv
- Probably the most basic and most important MPI commands

MPI send/recv



MPI Send/Recv

- Bcast and Reduce are examples of *collective* communication
- *Point-to-point* communication carried out by send and recv
- Probably the most basic and most important MPI commands
- Can send data between any two processors.
- Both send and recv are needed for data transfer.
- E.g. for previous figure need: if myid==1, send data to P2 *and* If myid==2 receive data from P1

MPI Send/Recv

If (myid==1) call `MPI_SEND(n, 1, MPI_INTEGER,0, tag, MPI_COMM_WORLD,ierr)`

If (myid==0) call `MPI_RECV(n, 1, MPI_INTEGER,1,
tag, MPI_COMM_WORLD, status, ierr)`

These will send the integer **n which has size **1** from processor **1** to processor **0**.**

MPI Send/Recv

If (myid==1) call `MPI_SEND(n, 1, MPI_INTEGER, 0, tag, MPI_COMM_WORLD, ierr)`

If (myid==0) call `MPI_RECV(n, 1, MPI_INTEGER, 1,
tag, MPI_COMM_WORLD, status, ierr)`

These will send the integer **n** which has size **1** from processor **1** to processor **0**.

tag: an integer label which can contain information about the data (e.g. which molecule the data belongs to or which row in a matrix)

status: provides information about the received message (source, tag, length)

MPI Send/Recv: example

- **f90example2.f90: compute** $\text{array1} = \sin(i1)$, $i1=1,2,\dots$
- **New code:** *sendExample.f90*
- **Now, array1 is only computed on P0, and we want to send the 3rd component to P1 and store it in P1's (empty) array1**

MPI Send/Recv: example

- `f90example2.f90`: compute `array1 = sin(i1), i1=1,2,...`
- New code: `sendExample.f90`
- Now, `array1` is only computed on P0, and we want to send the 3rd component to P1 and store it in P1's (empty) `array1`
- Compute `array1` on P0 and send it to P1:

```
i1 = 3
if (myid==0) then
  call calculations(N,array1) !fill in array1
  call MPI_SEND(array1(i1),1,MPI_DOUBLE_PRECISION,1,i1,MPI_COMM_WORLD,
                                                         ierr)
```

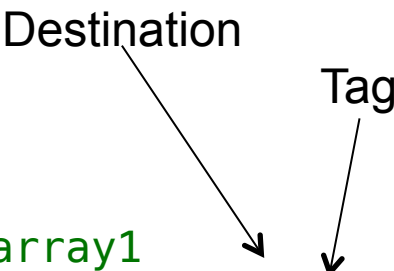
Destination

Tag

MPI Send/Recv: example

Compute array1 on P0, and send it to P1:

```
i1 = 3
if (myid==0) then
    call calculations(N,array1) !fill in array1
    call MPI_SEND(array1(i1),1,MPI_DOUBLE_PRECISION,1,i1,MPI_COMM_WORLD,
                                                           ierr)
```

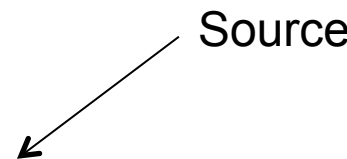


The diagram shows two labels, 'Destination' and 'Tag', with arrows pointing to the MPI_SEND function. 'Destination' points to the argument '1' (representing the destination rank), and 'Tag' points to the argument 'i1' (representing the message tag).

Must also have MPI_RECV on P1:

```
elseif (myid==1) then
    call MPI_RECV(var1,1,MPI_DOUBLE_PRECISION,0,MPI_ANY_TAG,
                                                           MPI_COMM_WORLD,status,ierr)

    j1 = status(MPI_TAG) !location where var1 will be stored in array1
    array1(j1) = var1
```



The diagram shows a label 'Source' with an arrow pointing to the argument '0' in the MPI_RECV function, which represents the source rank.

Notes:

- MPI_ANY_TAG: **The destination will accept a message with any tag**
- status(MPI_TAG) = 3; **we have used the tag to send/set the array index**

Comments on send/recv

- **Send/Recv are *blocking* operations**
 - **Code waits at send until the data has been received**
 - **But what if all processes are trying to send data to each other?**
 - **Can degrade performance or freeze the code**

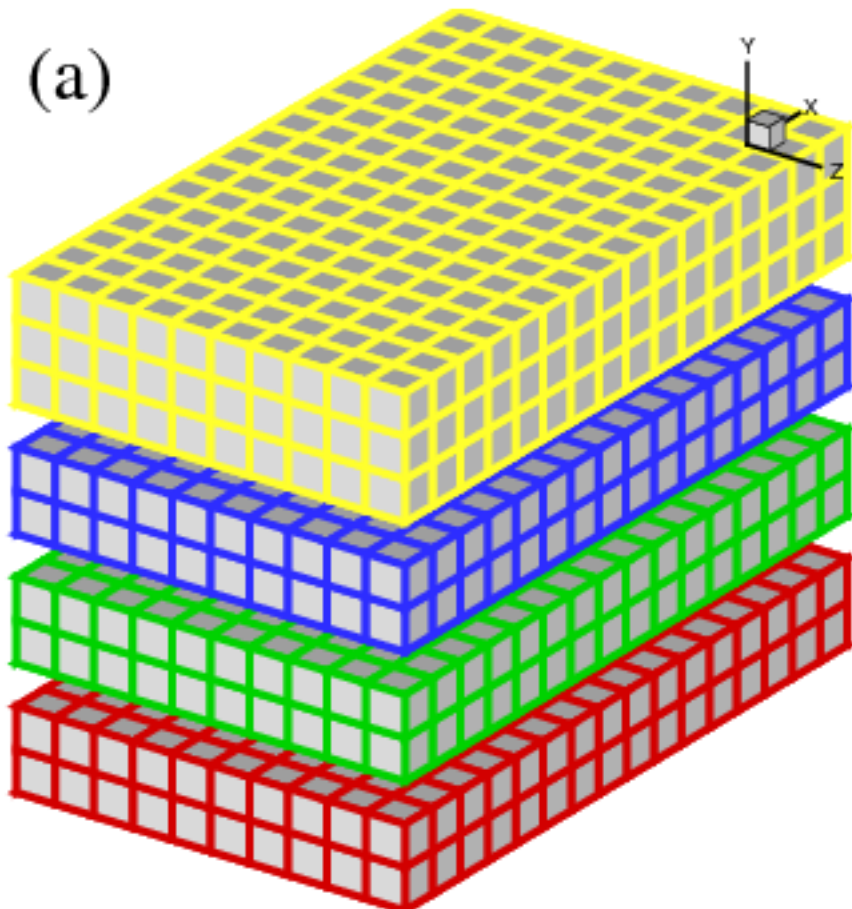
Comments on send/recv

- **Send/Recv are *blocking* operations**
 - Code waits at send until the data has been received
 - But what if all processes are trying to send data to each other?
 - Can degrade performance or freeze the code
- **Solutions:**
 - Combined send/recv: `MPI_SENDRECV`
 - Non-blocking send/recv: `MPI_ISEND`, `MPI_IRecv`
 - Usually used with `MPI_WAIT` or `MPI_TEST`
 - Buffered send: `MPI_BSEND`
 - Sender sends message and moves on
 - Message is stored in buffer until receiver is ready

Send/Recv and domain decomposition

A parallel computation computes a potential field, $f(x,y,z,t)$ on four processors.

P0, P1, P2, P3 solve for f in separate subdomains



How would you compute the gradient?

$$\nabla f = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right)$$

Computation of derivative

Equispaced grid: $x = x_1, x_2, x_3, \dots$

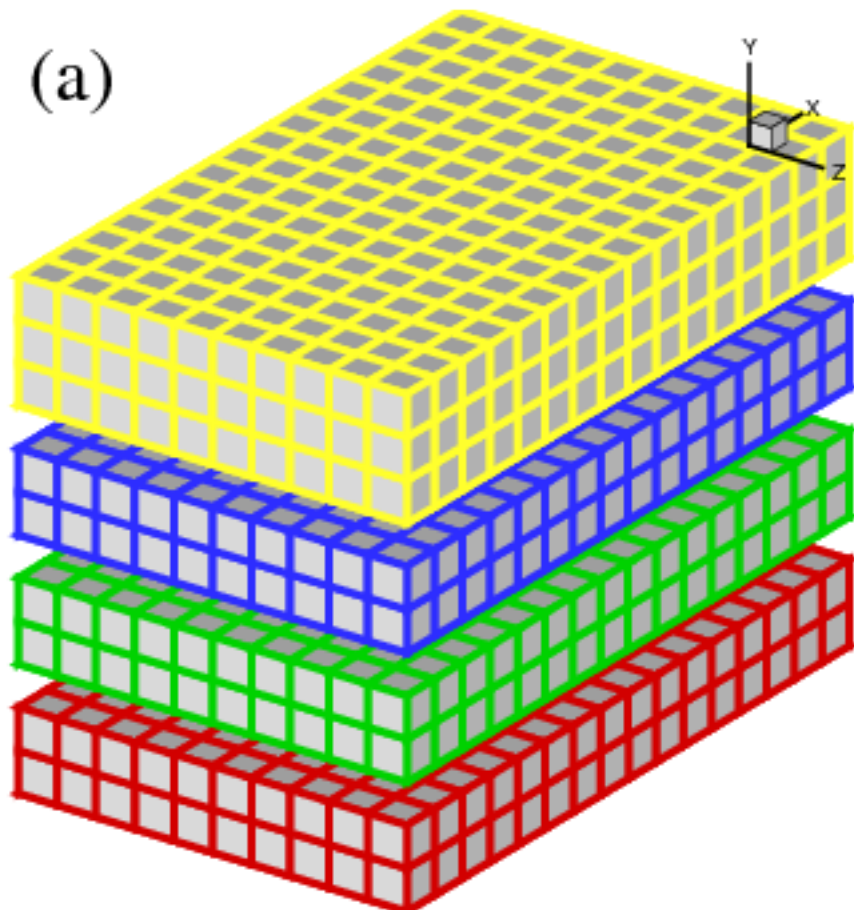
$x_{i+1} - x_i = h = \text{constant}$

Then, $\frac{df_i}{dx} \approx \frac{f_{i+1} - f_{i-1}}{2h}$

Send/Recv and domain decomposition

A parallel computation computes a potential field, $f(x,y,z,t)$ on four processors.

P0, P1, P2, P3 solve for f in separate subdomains



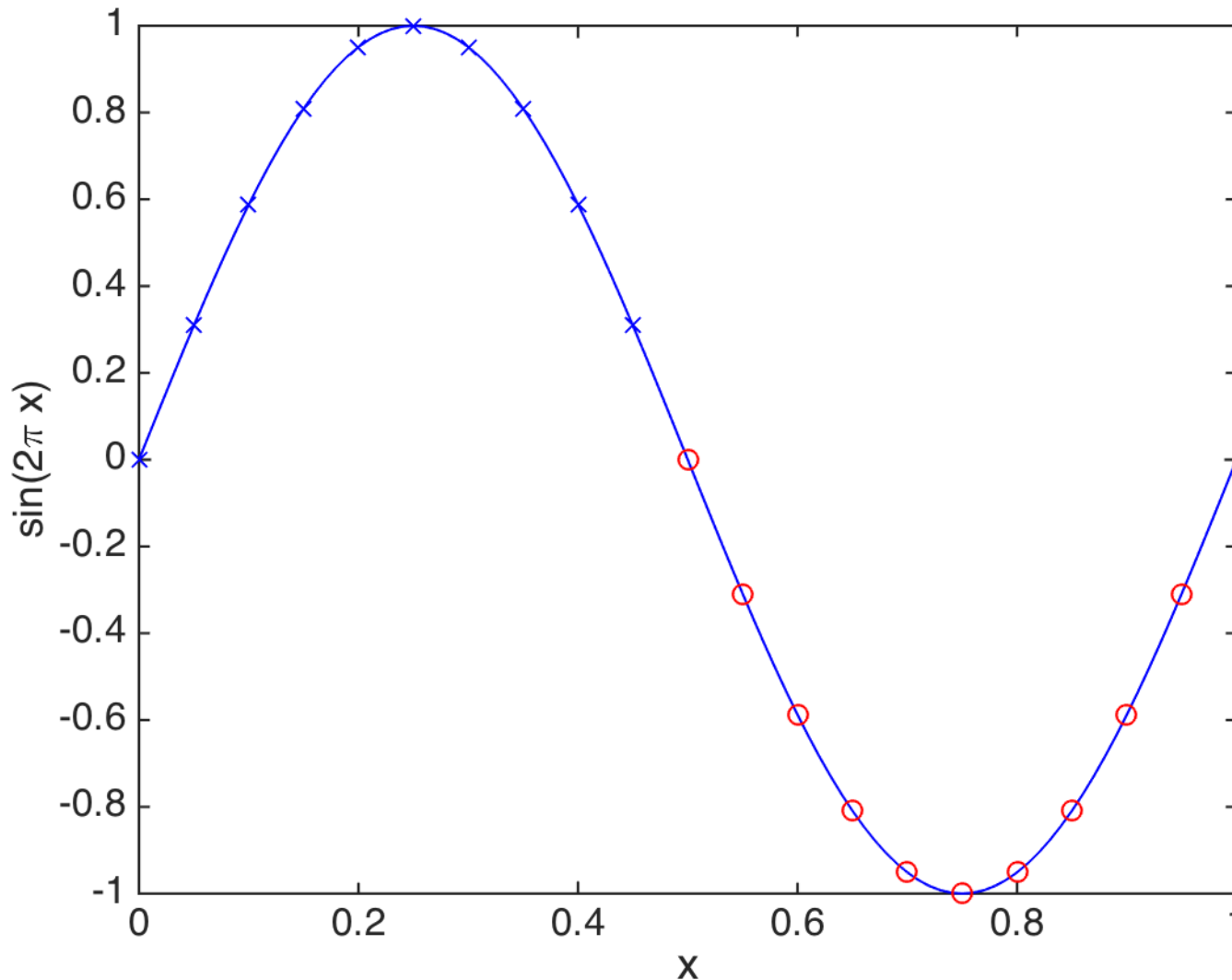
- How would you compute the gradient?

$$\nabla f = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right)$$

- No problems with x and z directions
- But what about y?

Send/Recv and domain decomposition

Let's look at a simpler 1-D problem



- 10 points on P0, 10 points on P1
- To compute df/dx for $i=10$ on P0, need to recv $f(i=11)$ from P1
- On P0, f will have 12 points:
 - The ten points shown
 - and 2 points received from each neighbor

Parallel differentiation example

- **gradient_p.f90: compute df/dx given $f=\sin(2\pi x)$ distributed across processors**

Code outline:

1. **Initialize MPI**
2. **Read N_{total} from *data.in***
3. **Construct domain decomposition – assign N_{local} points from i_{start} to i_{end} to each processor.**
4. **Make grid and field, $f=\sin(2\pi x)$, in the local subdomain**
5. **Compute derivative**
6. **Output error**

Parallel differentiation example

Key parts:

- Domain decomposition (subroutine from MPE library)

```
!construct decomposition
```

```
  call MPE_DECOMP1D( Ntotal, numprocs, myid, istart, iend)
```

```
  Nlocal = iend - istart + 1
```

Parallel differentiation example

Key parts:

- Domain decomposition (subroutine from MPE library)

```
!construct decomposition
```

```
  call MPE_DECOMP1D( Ntotal, numprocs, myid, istart, iend)  
  Nlocal = iend - istart + 1
```

- Make *local* grid and field

```
!make grid and field
```

```
  call make_grid(Ntotal,Nlocal,istart,iend,x)
```

```
  dx = x(2)-x(1)
```

```
  print *, 'proc', myid, ' has been assigned the interval x=',  
x(1),x(Nlocal)
```

```
  call make_field(Nlocal,x,f(2:Nlocal+1)) !note: f(1) and  
f(Nlocal+2) must be obtained from neighboring processors
```

Parallel differentiation example

Key parts:

- Domain decomposition (subroutine from MPE library)

```
!construct decomposition
```

```
  call MPE_DECOMP1D( Ntotal, numprocs, myid, istart, iend)  
  Nlocal = iend - istart + 1
```

- Make *local* grid and field

```
!make grid and field
```

```
  call make_grid(Ntotal,Nlocal,istart,iend,x)
```

```
  dx = x(2)-x(1)
```

```
  print *, 'proc', myid, ' has been assigned the interval x=',  
x(1),x(Nlocal)
```

```
  call make_field(Nlocal,x,f(2:Nlocal+1)) !note: f(1) and  
f(Nlocal+2) must be obtained from neighboring processors
```

- Compute derivative (with send/recv at subdomain boundaries)...

Parallel differentiation example

```
!-----  
!Send data at top boundary up to next processor  
!i.e. send f(nlocal+1) to myid+1 and store it there as f(1)  
!data from myid=numprocs-1 is sent to myid=0  
!-----
```

```
if (myid<numprocs-1) then  
    receiver = myid+1
```

```
else  
    receiver = 0
```

```
end if
```

```
if (myid>0) then  
    sender = myid-1
```

```
else  
    sender = numprocs-1
```

```
end if
```

```
call MPI_SEND(f(Nlocal+1),1,MPI_DOUBLE_PRECISION,receiver,0,  
              MPI_COMM_WORLD,ierr)
```

```
call MPI_RECV(f(1) 1,MPI_DOUBLE_PRECISION,sender,MPI_ANY_TAG,  
              MPI_COMM_WORLD,status,ierr)
```

Parallel differentiation example

```
!-----  
!Send data at top boundary up to next processor  
!i.e. send f(nlocal+1) to myid+1 and store it there as f(1)  
!data from myid=numprocs-1 is sent to myid=0  
!-----
```

```
  if (myid<numprocs-1) then  
    receiver = myid+1
```

```
  else  
    receiver = 0
```

```
  end if
```

```
  if (myid>0) then  
    sender = myid-1
```

```
  else  
    sender = numprocs-1
```

```
  end if
```

```
  call MPI_SEND(f(Nlocal+1),1,MPI_DOUBLE_PRECISION,receiver,0,  
               MPI_COMM_WORLD,ierr)
```

```
  call MPI_RECV(f(1) 1,MPI_DOUBLE_PRECISION,sender,MPI_ANY_TAG,  
               MPI_COMM_WORLD,status,ierr)
```

Parallel differentiation example

- At end of computation, each process has its own part of df/dx
- It is sometimes useful to *gather* the data onto one process (e.g. for writing data to a file)
- Easy to do if gathering ten numbers from ten processors and storing them in an array

Parallel differentiation example

- At end of computation, each process has its own part of df/dx
- It is sometimes useful to *gather* the data onto one process (e.g. for writing data to a file)
- Easy to do if gathering ten numbers from ten processors and storing them in an array

```
!gather Nlocal from each proc to array Nper_proc on myid=0  
call MPI_GATHER(Nlocal,1,MPI_INT,Nper_proc,1,MPI_INT 0,MPI_COMM_WORLD,  
ierr)
```

- Nlocal (size 1, type int) is sent into Nper_proc (rank 1 array, type int) on *myid = 0*

Parallel differentiation example

- At end of computation, each process has its own part of df/dx
- It is sometimes useful to *gather* the data onto one process (e.g. for writing data to a file)
- Easy to do if gathering ten numbers from ten processors and storing them in an array

```
!gather Nlocal from each proc to array Nper_proc on myid=0  
call MPI_GATHER(Nlocal,1,MPI_INT,Nper_proc,1,MPI_INT 0,MPI_COMM_WORLD,  
ierr)
```

- Nlocal (size 1, type int) is sent into Nper_proc (rank 1 array, type int) on *myid = 0*
- Trickier to do when gathering arrays into larger array. Then need array of locations where to place sub-arrays
 - e.g., `disps = [1, 1+Nper_proc(1), 1+Nper_proc(1)+Nper_proc(2), ...]`

Parallel differentiation example

- Trickier to do when gathering arrays into larger array. Then need array of locations where to place sub-arrays
 - e.g., $\text{disps} = [1, 1 + \text{Nper_proc}(1), 1 + \text{Nper_proc}(1) + \text{Nper_proc}(2), \dots]$
 - Then use *mpi_gatherv* with *disps* as input:
!collect df from each processor onto myid=0
call MPI_GATHERV(df, Nlocal, MPI_DOUBLE_PRECISION, df_total, Nper_proc,
disps, MPI_DOUBLE_PRECISION, 0, MPI_COMM_WORLD, ierr)
- *df* (size *Nlocal*) is gathered from each processor and stored in *df_total* (in locations determined from *Nper_proc* and *disps*)

Notes on method of lines

Lecture 14: Solving N ODEs:

$$\frac{dT_i}{dt} = S_i(t) + \frac{T_{i+1} - 2T_i + T_{i-1}}{\Delta x^2}, \quad i = 1, 2, \dots, N$$

- Can solve ODEs with *odeint* which adjusts the time step error criteria are satisfied
- Can also use time-step methods (see lab 8)
- Simplest is explicit Euler:

$$\frac{dT_i}{dt} \approx \frac{T_i(t + \Delta t) - T_i(t)}{\Delta t}$$

$$T_i(t + \Delta t) = T_i(t) + \Delta t \left[S_i(t) + \frac{T_{i+1} - 2T_i + T_{i-1}}{\Delta x^2} \right], \quad i = 1, 2, \dots, N$$

Notes on method of lines

Simplest is explicit Euler:

$$\frac{dT_i}{dt} \approx \frac{T_i(t + \Delta t) - T_i(t)}{\Delta t}$$

$$T_i(t + \Delta t) = T_i(t) + \Delta t \left[S_i(t) + \frac{T_{i+1} - 2T_i + T_{i-1}}{\Delta x^2} \right], \quad i = 1, 2, \dots, N$$

error ~ dt, very poor stability properties → requires very small time step

- **Fourth-order Runge-Kutta (RK4) is a much better fixed-time step method**
- **In course project, you are provided with routines for both methods**
 - **Will have to modify euler**
 - **But will only have to provide appropriate RHS for RK4 (as RHS was provided for *odeint*)**

Timing code

1. Lazy approach:

```
$ time mpiexec -n 2 midpointpt
```

```
real    0m0.073s
```

```
user    0m0.081s
```

```
sys 0m0.030s
```

Timing code

1. Lazy approach:

```
$ time mpiexec -n 2 midpointpt
```

```
real    0m0.073s
user    0m0.081s
sys 0m0.030s
```

2. Use *MPI_WTIME* to time particular parts of code:

```
starttime = MPI_WTIME() !***START TIMER***
!code...
!
!
endtime = MPI_WTIME() !***STOP TIMER***
print *, 'time= ',endtime - starttime, 'seconds'
```

Timing code

1. Lazy approach:

```
$ time mpiexec -n 2 midpointpt
```

```
real    0m0.073s
user    0m0.081s
sys 0m0.030s
```

2. Use *MPI_WTIME* to time particular parts of code:

```
starttime = MPI_WTIME() !***START TIMER***
!code...
!
!
endtime = MPI_WTIME() !***STOP TIMER***
print *, 'time= ',endtime - starttime, 'seconds'
```

3. But to get detailed information, use a profiler: VampirTrace, IPM, ...

Timing code

From IPM webpage (<http://ipm-hpc.sourceforge.net/>):

