MPI: Essentials of Message Passing

Thomas Hauser, thomas.hauser@colorado.edu
Nick Featherstone, feathern.colorado.edu
University of Colorado Boulder

Mar 14-16, 2017

Outline

- Overview
- Blocking Communication
- Load-balancing
- Non-blocking Communication

Message passing

- Most natural and efficient paradigm for distributed-memory systems
- Two-sided, send and receive communication between processes
- Efficiently portable to shared-memory or almost any other parallel architecture:
  - "assembly language of parallel computing" due to universality and detailed, low-level control of parallelism

More on message passing

- Provides natural synchronization among processes (through blocking receives, for example), so explicit synchronization of memory access is unnecessary
- Sometimes deemed tedious and low-level, but thinking about locality promotes
  - good performance,
  - scalability,
  - Portability
- Dominant paradigm for developing portable and scalable applications for massively parallel systems

Message sending and receiving

- Which process is sending the message?
- Where is the data on the sending process?
- What kind of data is being sent?
- How much data is there?
- Which process is going to receive the message?
- Where should the data be stored on the receiving process?
- What amount of data is the receiving process prepared to accept?

Blocking send

- call MPI_SEND(

message,	e.g., my_partial_sum,
count,	number of values in msg
data_type,	e.g, MPI_DOUBLE_PRECISION,
destination,	e.g., myid + 1
tag,	some info about msg, e.g., store it
communicator,	e.g., MPI_COMM_WORLD,
ierr	error tag (return value)

)

All arguments are inputs (except ierr).

Fortran MPI Data Types

MPI_CHARACTER
MPI_COMPLEX, MPI_COMPLEX8, also 16 and 32
MPI_DOUBLE_COMPLEX
MPI_DOUBLE_PRECISION
MPI_INTEGER
MPI_INTEGER1, MPI_INTEGER2, also 4 and 8
MPI_LOGICAL
MPI_LOGICAL1, MPI_LOGICAL2, also 4 and 8
MPI_REAL
MPI_REAL4, MPI_REAL8, MPI_REAL16

Numbers = numbers of bytes
Somewhat different in C—see text or Google it

C MPI Datatypes

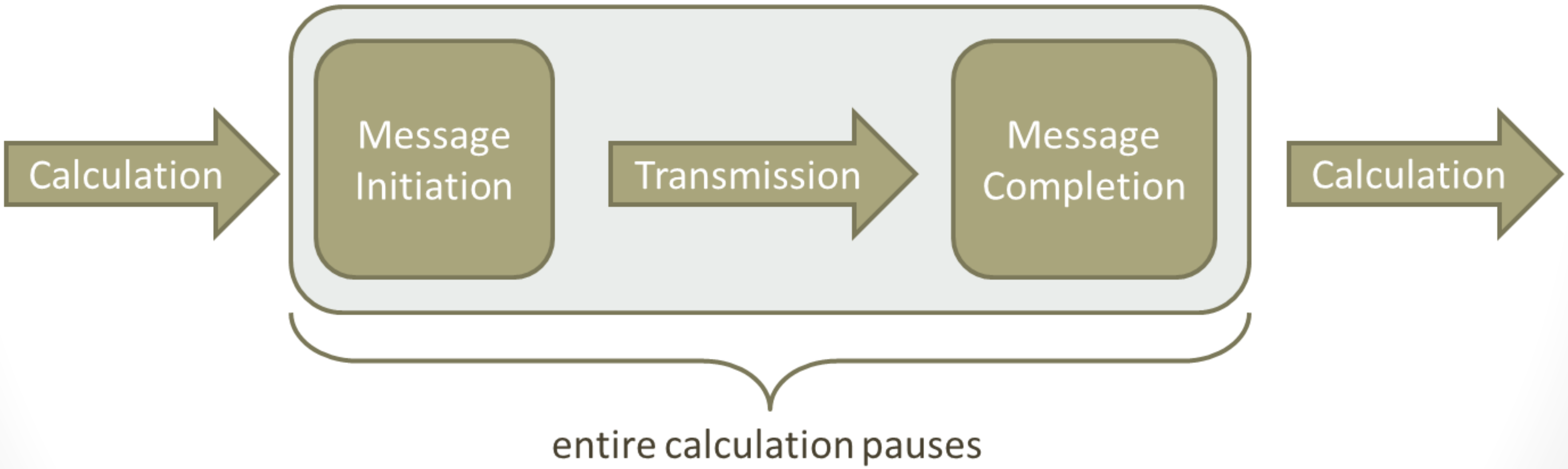
MPI_CHAR	8-bit character
MPI_DOUBLE	64-bit floating point
MPI_FLOAT	32-bit floating point
MPI_INT	32-bit integer
MPI_LONG	32-bit integer
MPI_LONG_DOUBLE	64-bit floating point
MPI_LONG_LONG	64-bit integer
MPI_LONG_LONG_INT	64-bit integer
MPI_SHORT	16-bit integer
MPI_SIGNED_CHAR	8-bit signed character
MPI_UNSIGNED	32-bit unsigned integer
MPI_UNSIGNED_CHAR	8-bit unsigned character
MPI_UNSIGNED_LONG	32-bit unsigned integer
MPI_UNSIGNED_LONG_LONG	64-bit unsigned integer
MPI_UNSIGNED_SHORT	16-bit unsigned integer
MPI_WCHAR	Wide (16-bit) unsigned character

Blocking?

- MPI_send
  - does not return until the message data and envelope have been buffered in matching receive buffer or temporary system buffer.
  - can complete as soon as the message was buffered, even if no matching receive has been executed by the receiver.
  - MPI buffers or not, depending on availability of space
  - **non-local**: successful completion of the send operation may depend on the occurrence of a matching receive.

Blocking Communication: Program Flow

- Programs written using blocking sends & receives possess portions similar to schematic below:



Blocking receive

- Process must wait until message is received to return from call.

- Stalls progress of program BUT
  - blocking sends and receives enforce process synchronization
  - so enforce consistency of data

Blocking receive

- call MPI_RECV(

message,	e.g., my_partial_sum,
count,	number of values in msg
data_type,	e.g, MPI_DOUBLE_PRECISION,
source,	e.g., myid - 1
tag,	some info about msg, e.g., store it
communicator,	e.g., MPI_COMM_WORLD,
status,	info on size of message received
ierr	

)

The arguments

- outputs: message, status
- count*size of data_type determines size of receive buffer:
  --too large message received gives error,
  --too small message is ok
- status must be decoded if needed
  - MPI_Get_Count(status, datatype, ierror)
  - status(MPI_SOURCE)
    status.MPI_SOURCE
  - status(MPI_TAG) status.MPI_TAG
  - status(MPI_ERROR) status.MPI_ERROR

Wildcards

- MPI_ANY_SOURCE
- MPI_ANY_TAG

- Send must send to specific receiver
- Receive can receive from arbitrary sender

Example Program

- Some examples of point-to-point communication:

  MPI/Lab/session2/examples/messages_blocking.f90

- Start an interactive session:

  Salloc –A training –n 24 –t 120 –p UV - - reservation=training

- Build and run the code with 2 cores:
  - module load mpi/mpich-3.2-intel   intel
  - make
  - srun –mpi=pmi2 –n 2 ./message_blocking.out

Example Program

- Some examples of point-to-point communication:

  MPI/Lab/session2/examples/messages_blocking.f90

- All MPI subdirectories contain a Makefile
  - Type “make” to build all programs in a directory
  - Exercises have solutions in the “solutions” directory

- For exercises, Makefiles generate code compiled with:
  - optimization flags (e.g., ex1.out)
  - debugging flags (e.g., ex1.dbg)

Example Program

- Some examples of point-to-point communication:

  MPI/Lab/session2/examples/messages_blocking.f90

- Transmitting single data values:
  - SWAP_DOUBLE()
  - SWAP_INTEGER()

- Communicating multiple data values:
  - SWAP_MULTI_DOUBLE()

Deadlock

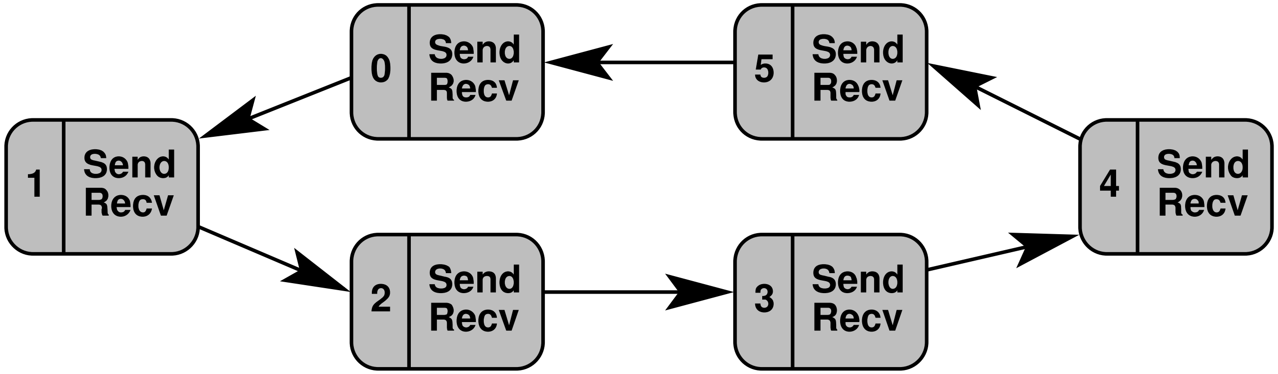
- Deadlock: process waiting for a condition that will never become true
- Easy to write send/receive code that deadlocks
  - Two processes: both receive before send
  - Send tag doesn't match receive tag
  - Process sends message to wrong destination process
- QUICK EXERCISE:
  - **Produce** a deadlock by swapping the order of **one** send/receive **pair** in messages_blocking.f90

Deadlock Exercise

- Build and run the code in:

  MPI/Lab/session2/exercise1/ex1.f90
- Fix the deadlock in this code…
- But what is this program intended to do anyway?

Sending data in a ring



- Store the data in array of size nprocs x n
- Each process sends message to neighbor with higher rank
    - N elements to id+1
- Receives values from neighbor with lower rank
    - N elements from id -1
- At the end sum up and print local results

Ring Data Layout

- Nprocs=4
- N=3

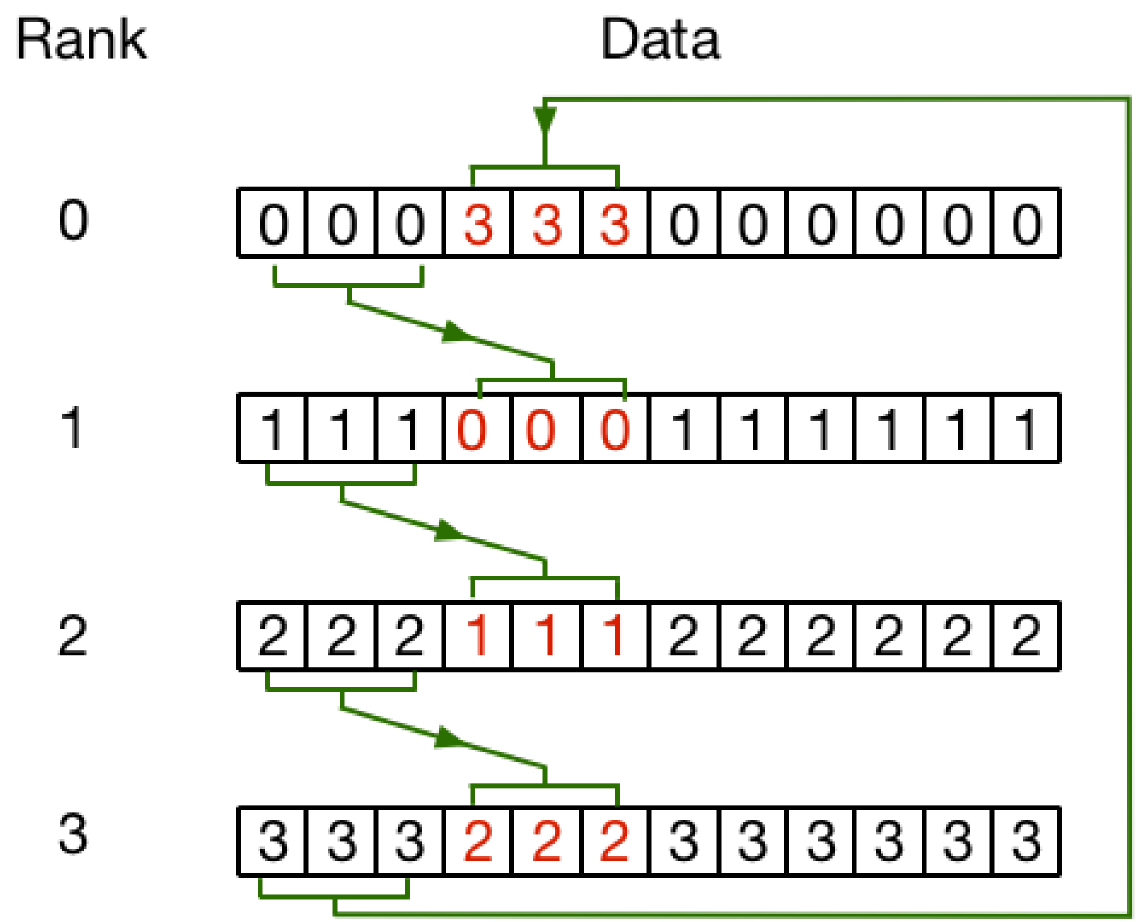
Ring Data Layout

- Nprocs=4
- N=3

Rank	Data											
0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3	3	3	3	3	3

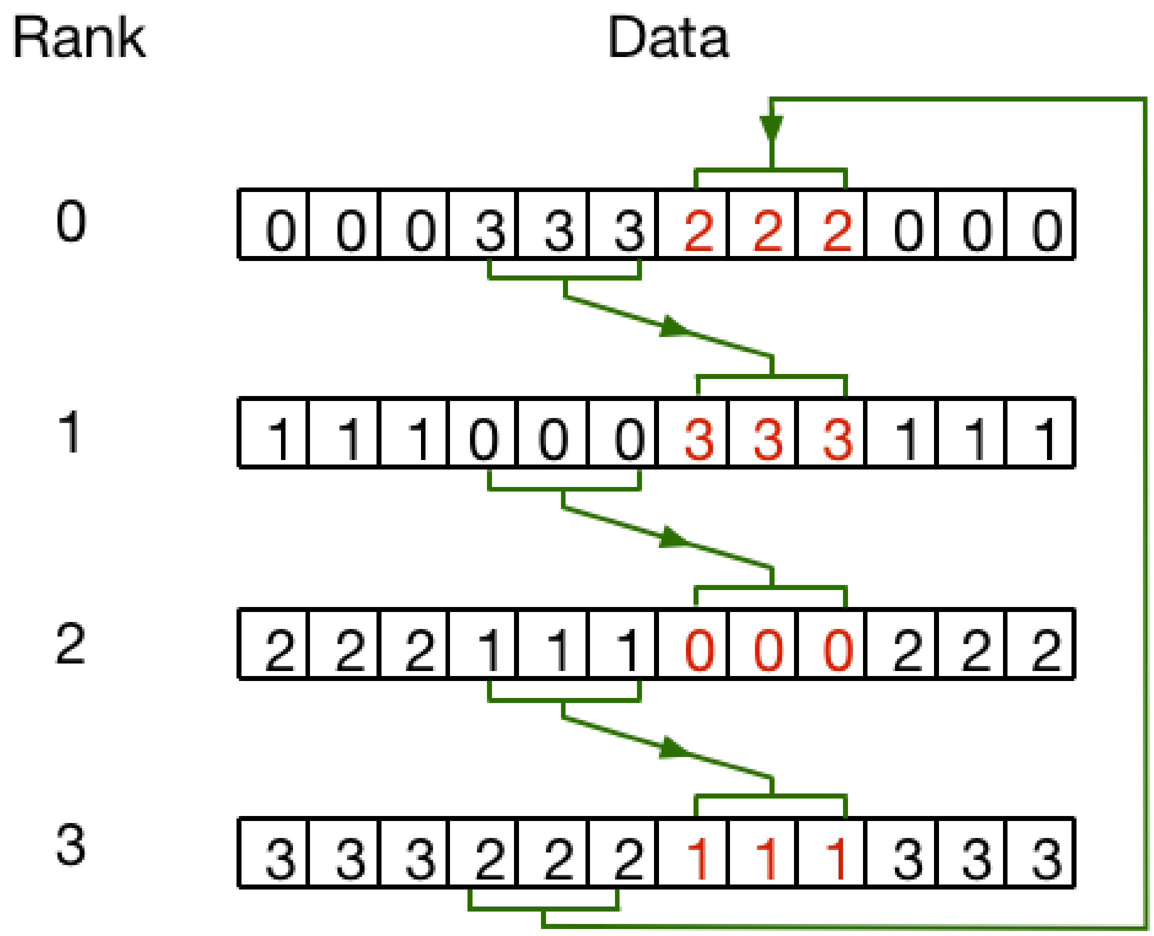
Ring Data Layout

- Nprocs=4
- N=3



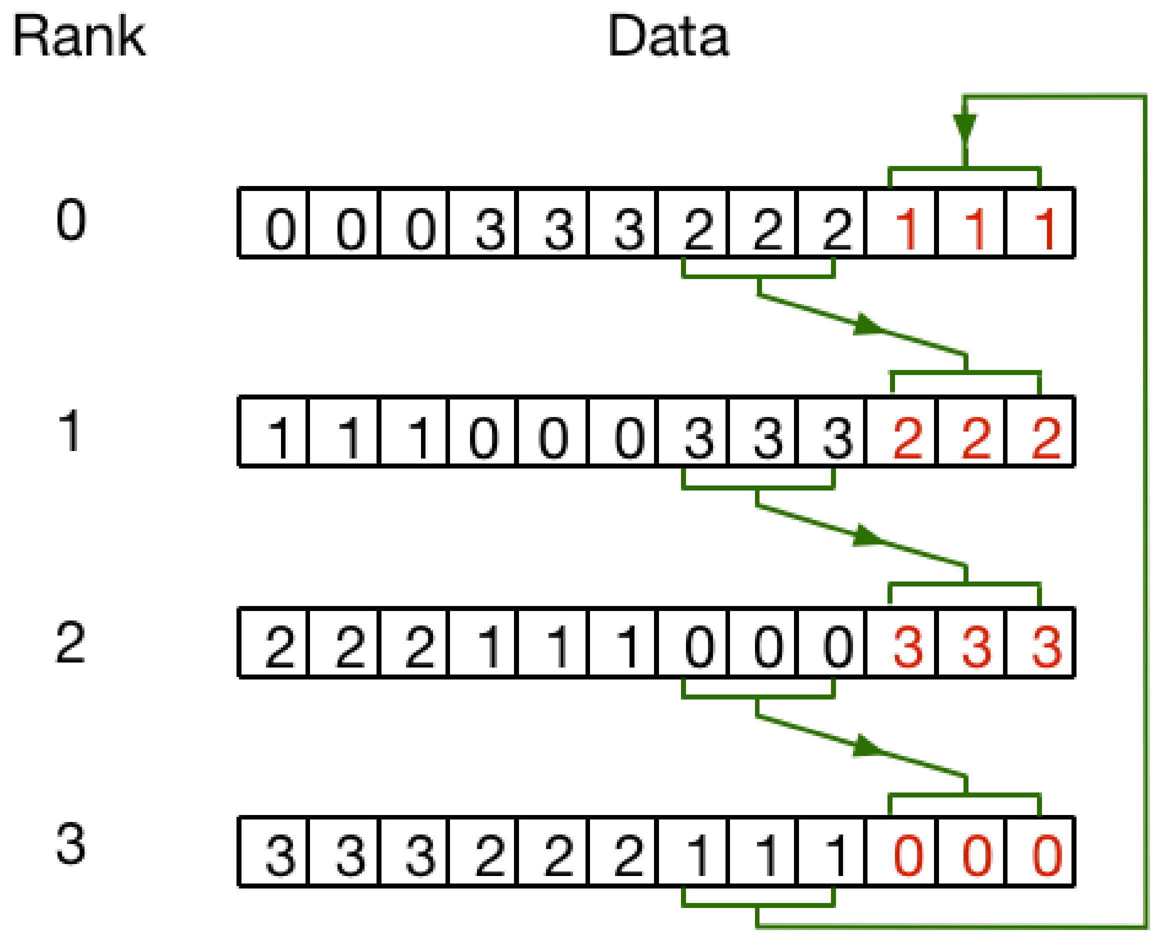
Ring Data Layout

- Nprocs=4
- N=3



Ring Data Layout

- Nprocs=4
- N=3



Ring Data Layout

- Nprocs=4
- N=3

Rank	Data											
0	0	0	0	3	3	3	2	2	2	1	1	1
1	1	1	1	0	0	0	3	3	3	2	2	2
2	2	2	2	1	1	1	0	0	0	3	3	3
3	3	3	3	2	2	2	1	1	1	0	0	0

Deadlock Exercise

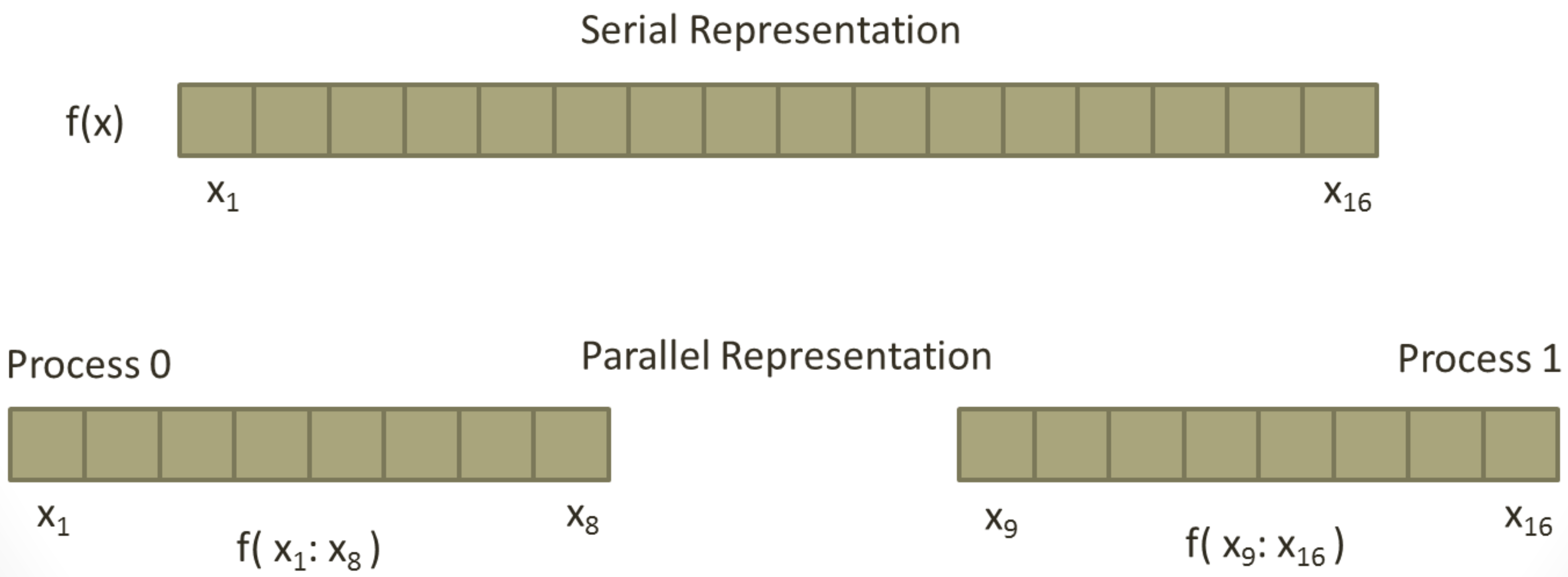
- Build and run the code in:

  MPI/Lab/session2/exercise1/ex1.f90
- Fix the deadlock in this code…

Load-Balancing

- When parallelizing a program, we split up the work.

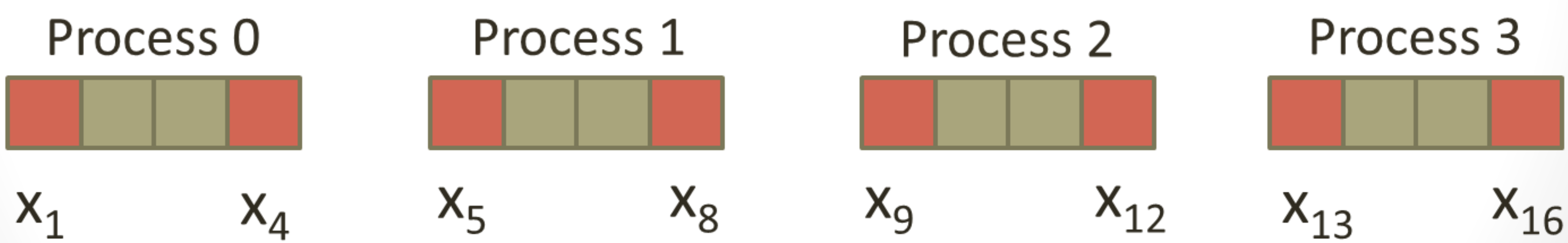
- Many physics applications:
  physical variables distributed based on spatial grid coordinates



1-D Diffusion Problem

$$f_{x,t+1} = ½ \left(f_{x-1,t} + f_{x+1,t} \right)$$

- Commonly encountered computational kernel
- Shaded regions cannot be updated without communication

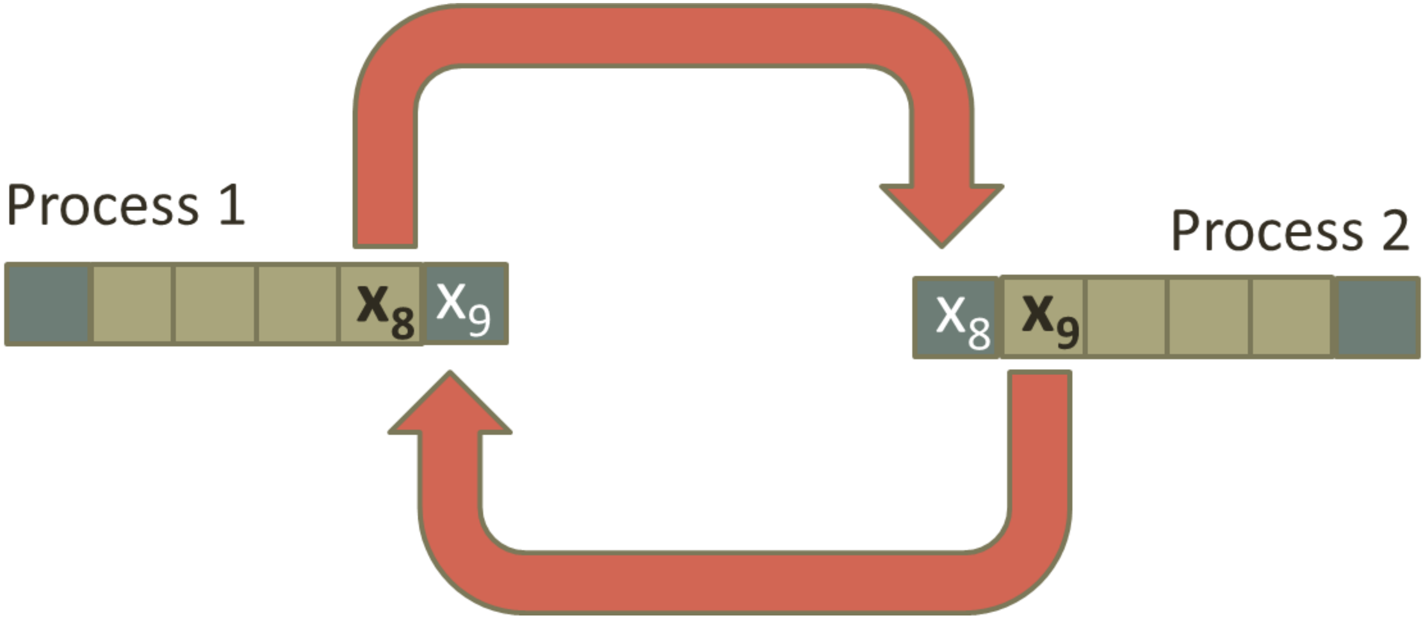


Solution: Ghost Zones

- Neighboring processes hold overlapping data: **ghost zones**
- Update via send/receive pairs during each time step

Solution: Ghost Zones

- Neighboring processes hold overlapping data: **ghost zones**
- Updated during each time step via send/receive pairs



1-D Diffusion Exercise

- Build and run the code in:

  MPI/Lab/session2/exercise2/ex2.f90

- Let's examine the code…

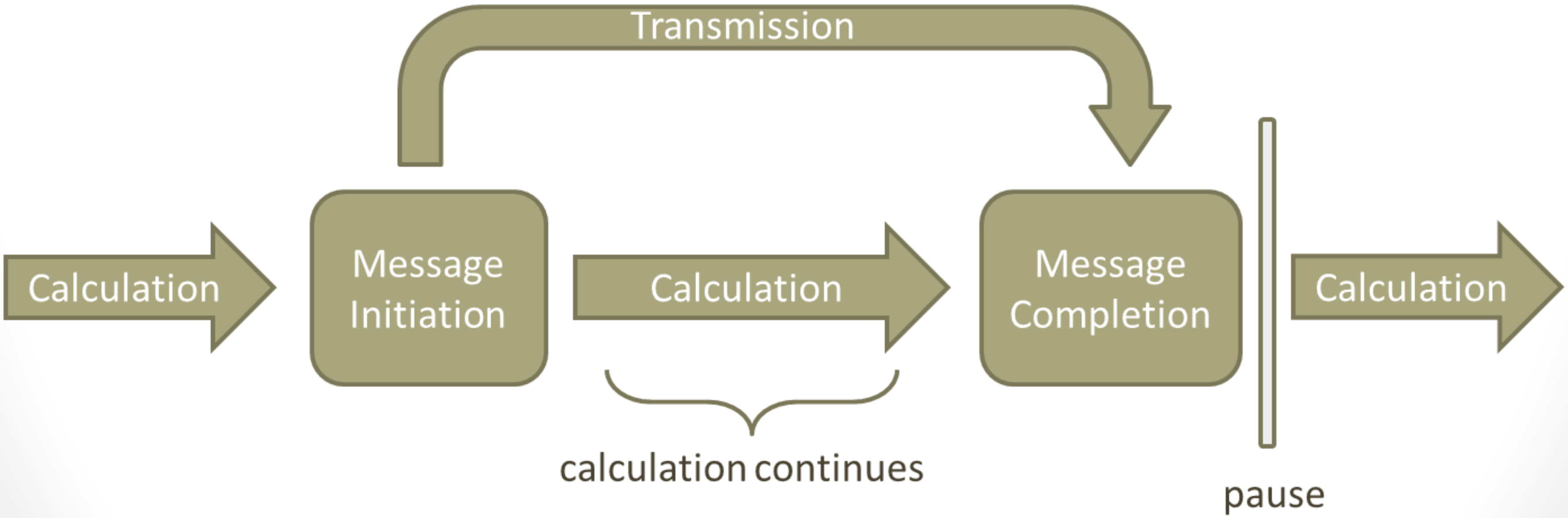
- Modify the program so that
  - The **nx** gridpoints are distributed across the **ncpu** MPI ranks.
  - Ghostzones are communicated correctly

Non-Blocking Send & Receive

- Same syntax as MPI_Send() and MPI_Recv()
  - Addition of a request handle argument.
- Calls return immediately
- Data in the buffer (send and receive) may not be accessed until operations is complete.
- Send and receive are completed by
  - MPI_Test
  - MPI_Wait

Non-Blocking Communication: Program Flow

- Programs written using ISends & IReceives possess portions that are schematically similar to:



MPI_ISEND (buf, cnt, dtype, dest, tag, comm, request, ierr)

- Same syntax as MPI_SEND with the addition of a request handle
- Request is a handle (int in Fortran; MPI_Request in C) used to check for completeness of the send
- This call returns immediately
- Data in buf may not be accessed until the user has completed the send operation
- The send is completed by a successful call to MPI_TEST or a call to MPI_WAIT

MPI_IRECV(buf, cnt, dtype, source, tag, comm, request, ierr)

- Same syntax as MPI_RECV except status is replaced with a request handle
- Request is a handle (int in Fortran MPI_Request in C) used to check for completeness of the recv
- This call returns immediately
- Data in buf may not be accessed until the user has completed the receive operation
- The receive is completed by a successful call to MPI_TEST or a call to MPI_WAIT

MPI_WAIT (request, status, ierr)

- Request is the handle returned by the non-blocking send or receive call
- Upon return, status holds source, tag, and error code information
- This call does not return until the non-blocking call referenced by *request* has completed
- Upon return, the request handle is freed
- If *request* was returned by a call to MPI_ISEND, return of this call indicates nothing about the destination process

MPI_WAITALL (count, requests, statuses, ierr)

- *requests* is an array of handles returned by non-blocking send or receive calls
- *count* is the number of requests
- This call does not return until all non-blocking call referenced by *requests* have completed
- Upon return, *statuses* hold source, tag, and error code information for all the calls that completed
- Upon return, the request handles stored in *requests* are all freed

Example Program

- Some examples of ***non-blocking*** point-to-point communication:

  MPI/Lab/session2/examples/messages_nonblocking.f90

- Transmitting single data values:
  - SWAP_DOUBLE()
  - SWAP_INTEGER()

- Communicating multiple data values:
  - SWAP_MULTI_DOUBLE()

Ring-Communication Exercise

- Examine the code in:
  MPI/Lab/session2/exercise3
- Rewrite this code using non-blocking ISends and IRecvs

1-D Diffusion Problem Exercise

- Examine the code in:

  MPI/Lab/session2/exercise4
- Revise this code so that it uses ISends and IRecvs