MPI Programming Part 1

Wei Feinstein

HPC@LSU

Outline

- Why MPI
- MPI programing basics
- Point-to-point communications

Why Parallel Computing

As computing tasks get larger and larger, may need to enlist more computer resources

- Bigger: more memory and storage
- Faster: each processor is faster
- More: do many computations simultaneously

Memory System Models for Parallel Computing

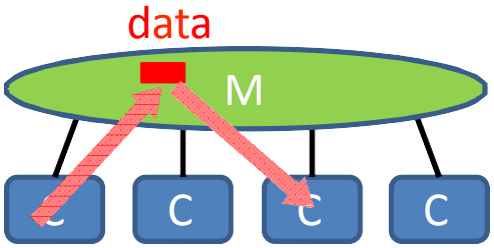
Different ways of sharing data among processors

- Shared Memory
- Distributed Memory
- Hybrid (shared + distributed)
- PGAS (Partitioned Global Address Space)

Shared Memory Model

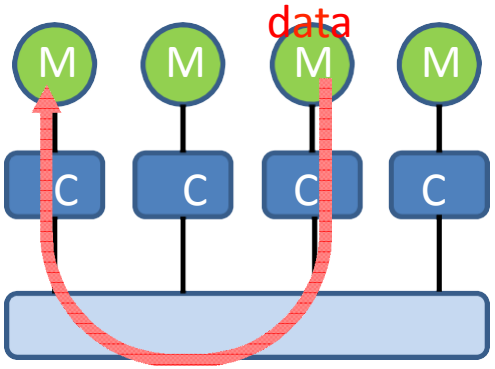
- All threads/processes have access to global address space
- Data sharing achieved via read/ write to the same memory location
- Within single computer

e.g., OpenMP



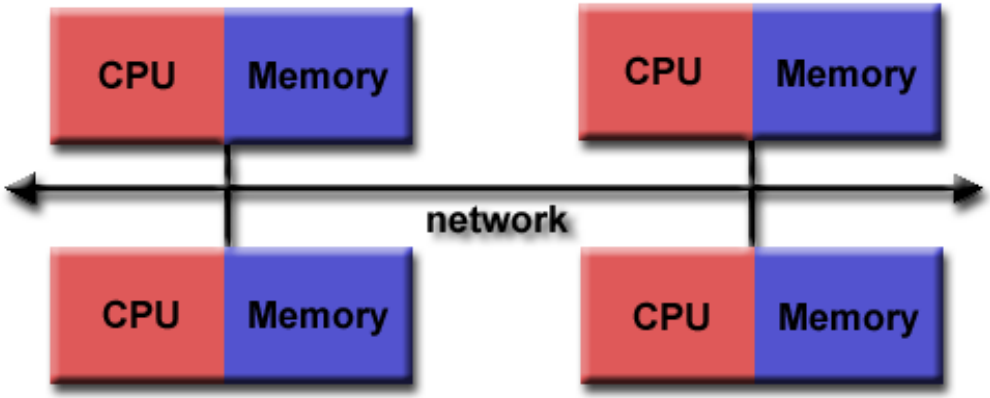
Distributed Memory Model

- Each process has its own address space locally
- Data sharing achieved via explicit message passing (through network)
- Inter- and intra computers
- Example: MPI (Message Passing Interface)

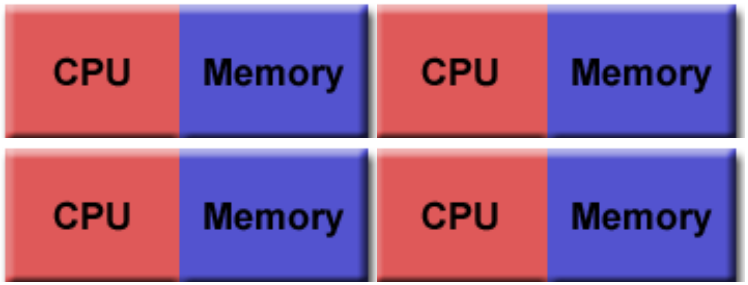


MPI Programming Models

- Distributed

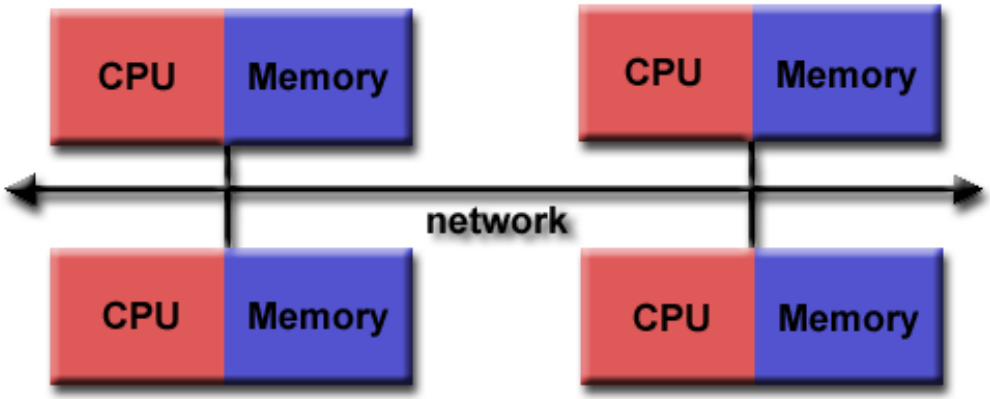


- Single node

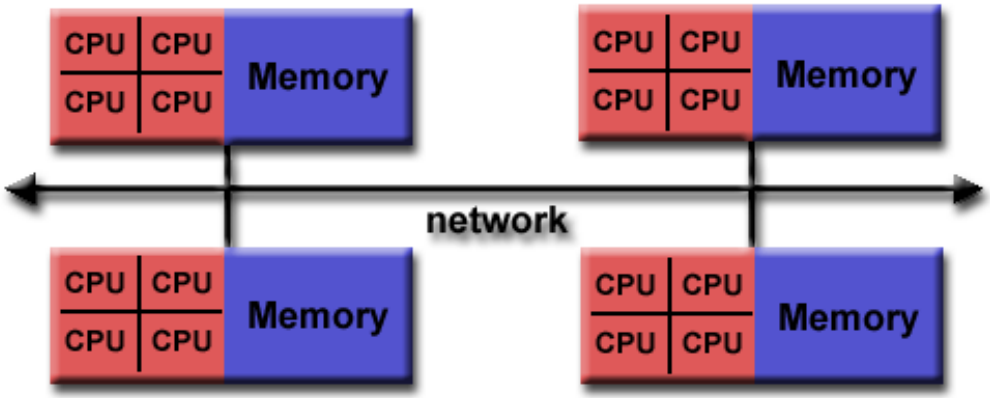


MPI Programming Models

- Distributed

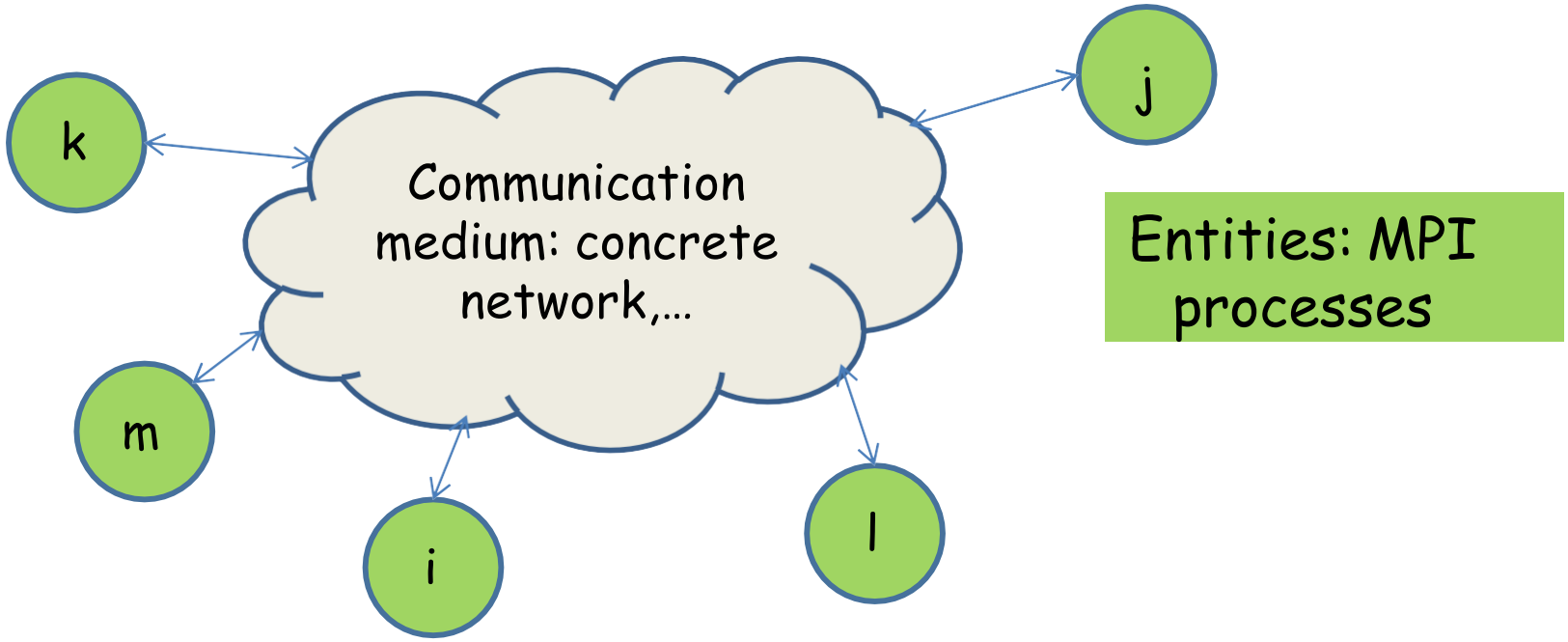


- Hybrid MPI+OpenMP



Message Passing

Any data to be shared must be explicitly transferred from one to another



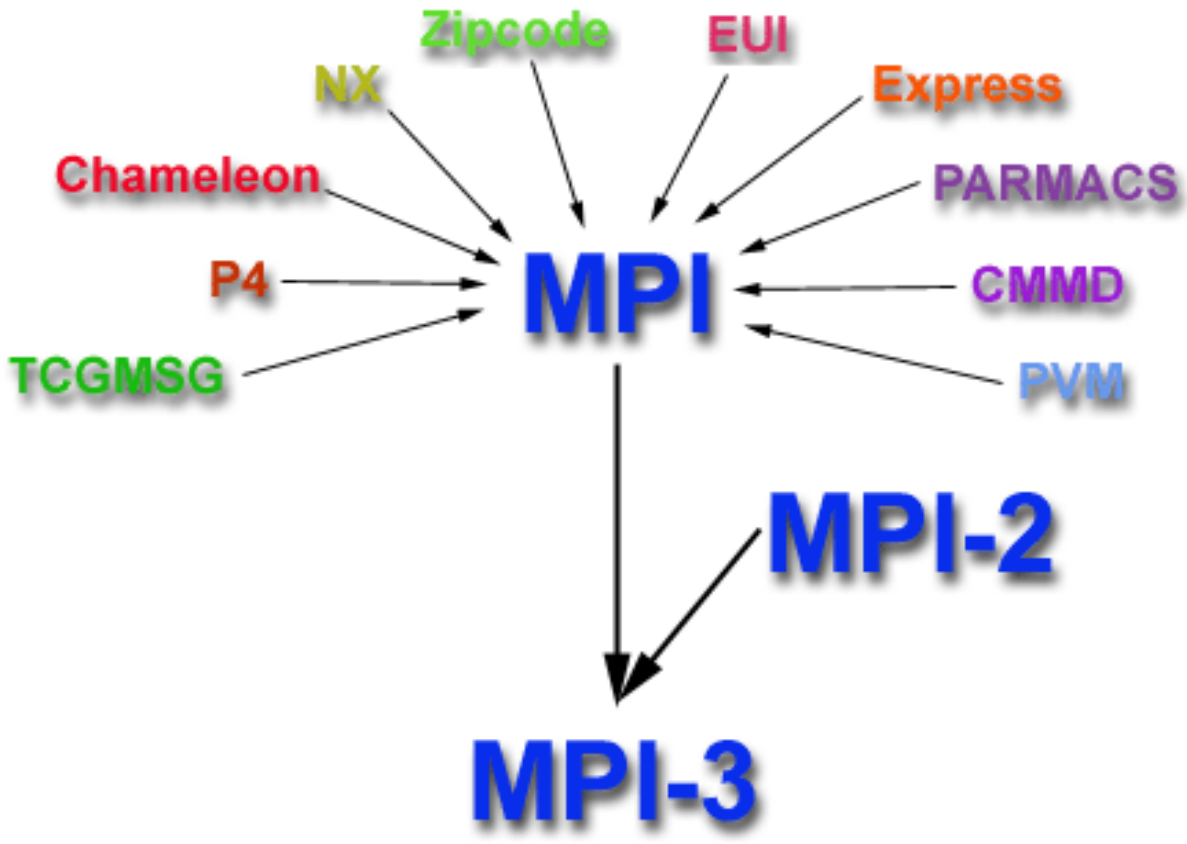
Why MPI?

- Universality:
  - Works on separate processors connected by any network or shared memory systems
  - Most modern parallel supercomputers have right hardware
  - Most useful on distributed memory machines

- Performance:
  - Scalability keeps MPI as a permanent component of HPC
  - Each message passing process only directly uses its local data, avoiding complexity of process-shared data, cache contention

MPI History

- 1980-1990
- 1994:MPI-1
- 1998:MPI-2
- 2012:MPI-3



MPI Standards

- MPI defines the portable message-passing standard designed by a group of researchers from academia and industry to function on a wide variety of parallel computing architectures
- MPI defines standard APIs for message passing
  - The standard includes
    - What functions are available
    - The syntax of those functions
    - What outcome to expect from those functions
  - The standard does NOT include
    - Implementation details (e.g. how the data transfer occurs)
    - Runtime details (e.g. how many processes the code run with etc.)
- MPI provides C/C++ and Fortran bindings
- Several third-party bindings for Python, R and more other languages

Various MPI Implementations

- OpenMPI: open source, portability and simple installation and configuration
- MPICH: open source, portable
- MVAPICH2: MPICH derivative
  InfiniBand, iWARP and other RDMA-enabled interconnects (GPUs)
- Intel MPI (IMPI): vendor-supported MPICH from Intel

More about MPI

- A MPI Program launches separate processes (tasks)

- Each task has its own address space
  – Requires partitioning data across tasks
  – Without data decomposition, the same task run N times
  – Data is explicitly moved from task to task by message passing

- Two classes of message passing
  - Point-to-Point communication: involving only two tasks
  - Collective communication: involving many tasks simultaneously

Let's try it

- `$ whoami`
- `$ mpirun –np 4 whoami`

What just happened?

- mpirun launched 4 processes
- Each process ran `whoami`
- Each process ran independently
- Usually launch no more MPI processes than #processors
- Use multiple nodes:

```
mpirun –hostfile machine.lst
-np/-npp 4 app.exe
```

Outline of a MPI Program

1. Initialize communications
   MPI_INIT initializes the MPI environment
   MPI_COMM_SIZE returns the number of processes
   MPI_COMM_RANK returns this process's index (rank)

2. Communicate to share data between processes
   MPI_SEND sends a message
   MPI_RECV receives a message

3. Exit in a "clean" fashion when MPI communication is done
   MPI_FINALIZE

Hello World (C)

```
include "mpi.h"

int main(int argc, char* argv[]){
int nprocs, myid;
...
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &myid);
printf("Hi %s, Hello World from process %d/%d
\n", name, myid, nprocs);
MPI_Finalize();
…
}
```

Header file

Initialization

Communication

Termination

Hello World (C)

```
include "mpi.h"

int main(int argc, char* argv[]){
int nprocs, myid;
...
MPI_Init(&argc,
MPI_Comm_size(M
MPI_Comm_rank(M
printf("Hello W
\n", myid, nproc

MPI_Finalize();
…
}
```

```
[wfeinste@shelob1 hello]$ mpicc hello.c
[wfeinste@shelob1 hello]$ mpirun -np 4 a.out
 Dear dear, Hello World from process 3/4
Dear dear,  Hello World from process 0/4
Dear dear, Hello World from process 2/4
Dear dear, Hello World from process 1/4
```

Hello World (Fortran)

```
include "mpif.h"

integer::nprocs, ierr, myid
integer::status(mpi_status_size)

call mpi_init(ierr)
call mpi_comm_size(mpi_comm_world, nprocs, ierr)
call mpi_comm_rank(mpi_comm_world, myid, ierr)

write(*, '("Hello World from process ",I3," /",I3)') myid,
nprocs

call mpi_finalize(ierr)

…
```

Header file

Initialization

Computation & communication

Termination

Hello World (Fortran)

```
include "mpif.h"

integer::nprocs, ierr, myid
integer::status(mpi_status_size)

call mpi_init(ierr)
call mpi_comm
call mpi_comm

write(*, '("Hel
nprocs

call mpi_finalize(ierr)

…
```

Header file

Initialization

utation &

unication

ination

```
[wfeinste@shelob1 hello]$ mpif90 hello.f90
[wfeinste@shelob1 hello]$ mpirun -np 4 ./a.out
Hello World from process   3 /  4
Hello World from process   0 /  4
Hello World from process   1 /  4
Hello World from process   2 /  4
```

Naming Signature (C/Fortran)

- Function name convention
  - C: `MPI_Xxxx(arg1,…)`

    ```
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs)
    ```
  - Fortran: `mpi_xxx( ) :` not case sensitive

    ```
    mpi_comm_size(mpi_comm_world, nprocs, ierr)
    ```

- Error handles
  If rc/ierr == MPI_SUCCESS, then the call is successful.

  - C: `int rc = MPI_Xxxx(arg1,…)`
  - Fortran: `call mpi_some_function(arg1,…,ierr)`

What is a MPI Communicator?

```
MPI_Comm_size(MPI_Com MPI_COMM_WORLD, int &nprocs)
MPI_Comm_rank(MPI_Com MPI_COMM_WORLD, int &myid)
```

- A group identifier of MPI processes that can send and receive messages to each other

Communicators

- MPI_COMM_WORLD: the default communicator contains all processes running a MPI program
  - Point-point and collective communication
  - When a function is called to send data to all processes, MPI needs to understand what "all" means
- A process can belong to multiple communicators
- There can be many communicators

```
e.g., MPI_Comm_split(MPI_Comm comm, int
color, int, kye, MPI_Comm* newcomm)
```

Ranks and Size

- Rank: unique process id within a communicator (start from 0)
  - C: `MPI_Comm_Rank(MPI_Comm comm,int *rank)`
  - Fortran: `MPI_COMM_RANK(COMM,RANK,ERR)`

- Size: total process # within a communicator
  - C: `MPI_Comm_Size(MPI_Comm comm, int *size)`
  - Fortran: `MPI_COMM_SIZE(COMM,SIZE,ERR)`

Compiling MPI Programs

Not a part of the standard

- Could vary from platform to platform
- Or even from implementation to implementation on the same platform
- mpicc/mpicxx/mpif77/mpif90: wrappers to compile MPI code and auto link to startup and message passing libraries.

MPI Compilers

Language	Script Name	Underlying Compiler
C	mpicc	gcc
	mpigcc	gcc
	mpiicc	icc
	mpipgcc	pgcc
C++	mpiCC	g++
	mpig++	g++
	mpiicpc	icpc
	mpipgCC	pgCC
Fortran	mpif90	f90
	mpigfortran	gfortran
	mpiifort	ifort
	mpipgf90	pgf90

Compiling and Running MPI Programs

– Compile

- C: **mpicc -o <executable name> <source file>**
- Fortran: **mpif90 -o <executable name> <source file>**

– Run

mpirun -hostfile $PBS_NODEFILE -np <number of procs> <executable name> <input parameters>

Compile: **mpicc -o hello hello.c**
Run: **mpirun -np 16 hello Marry**

In Class Exercises

- Exercises
  - Track a: Process color
  - Track b: Matrix multiplication
  - Track c: Laplace solver

- Your tasks:
  - Fill in blanks to make MPI code work under /exercise directory
  - Solutions are provided in /solution directory

Request an Interactive Node

- Never run your jobs on a cluster head node
- qsub -I -A hpc_training_2018 -l walltime=4:0:0 -l nodes=1:ppn=16

Exercise a1: Process Color

mpi_part1_2018/color/exercise

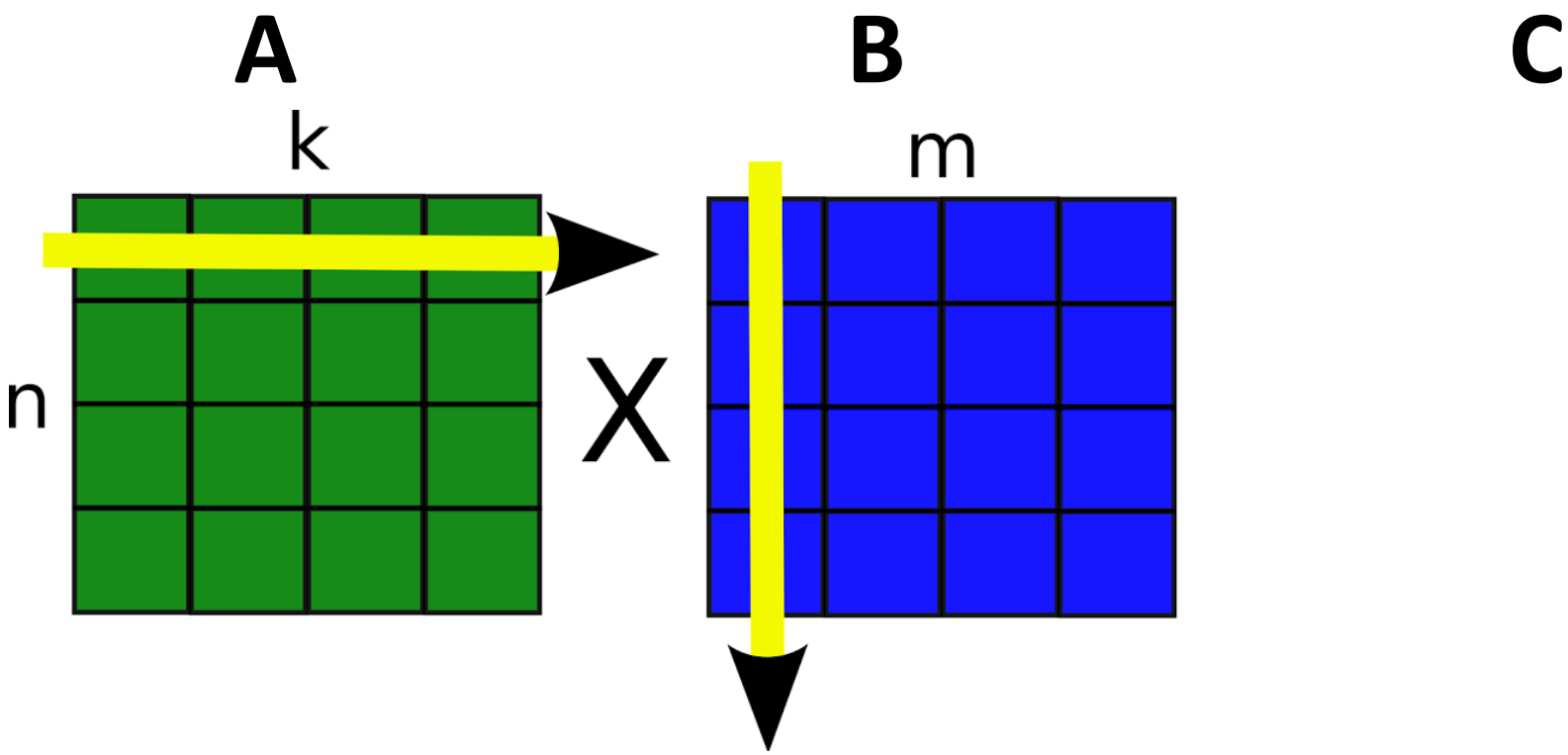
- Compile and run **ex0a.c** (C) or **ex0a.f90** (Fortran)
- What do you get?
- Processes with odd rank print to screen "Process x has the color green"
- Processes with even rank print to screen "Process x has the color red"
- Modify **ex1a.c** (C) or **ex1a.f90** (Fortran)

Exercise a1: Process Color

```
...
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);

  char host[10];
  gethostname(host, 255);
  if (myid % 2 == 0) // myid with even numbers
      printf("Process %d from %s has color red\n",myid, host);
  else               // myid with odd numbers
      printf("Process %d from %s has color green\n",myid, host);

  MPI_Finalize();
...
```

```
mpicc ex1a_solution.c
mpirun –np 24 –hostfile $PBS_NODEFILE a.out
```

Exercise b1: Matrix Multiplication



Exercise b1: Matrix Multiplication

```
for(i=0;i<row;i++)     //row of first matrix
   for(j=0;j<col;j++)  //column of second matrix
      for(k=0;k<col;k++)
         C[i][j] += A[i][k]*B[k][j];
```

Exercise b1: Matrix Multiplication

- Goal: Distribute the work load among processes in 1-d manner
  - Each process initializes its own copy of A and B
  - Each process computes part of the workload
    - Determine how to decompose (which process deals which rows or columns)
    - Make sure dimensions of A and B can be evenly divided by number of MPI processes
  - Validate the result C[rpeek][cpeek]

```
mpicc ex1a_solution.c
mpirun –np 16 a.out
```

Workload non-sharing vs. sharing

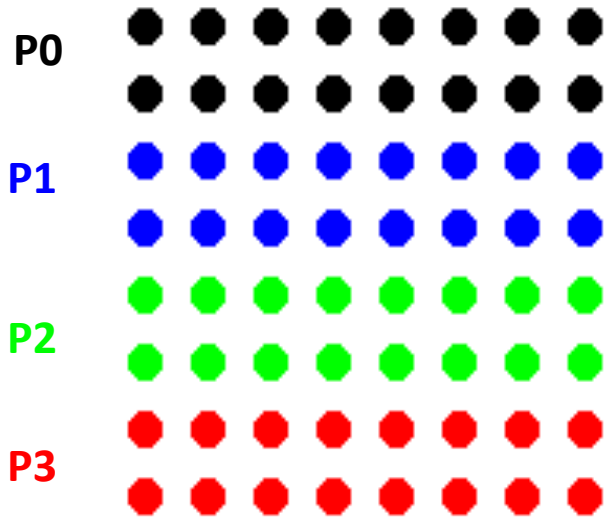
```
for (i=0; i<nra; i++){
   for (j=0;j<ncb;j++)
     for (k=0;k<nca;k++)
         C[i][j] += A[i][k]*B[k][j];
 }
```

Demo

```
int rows_per_process = nra/nprocs;
Int r_start = myrank * rows_per_process
Int r_end = (myrank+1)* rows_per_process - 1;

for (i=r_start; i<=r_end; i++)
   for (j=0; j<ncb; j++)
        for (k=0; k<nca; k++)
           C[i][j] += A[i][k]*B[k][j];
```

mpirun –np 16 a.out 100 200

How to divide workload among ranks/processes

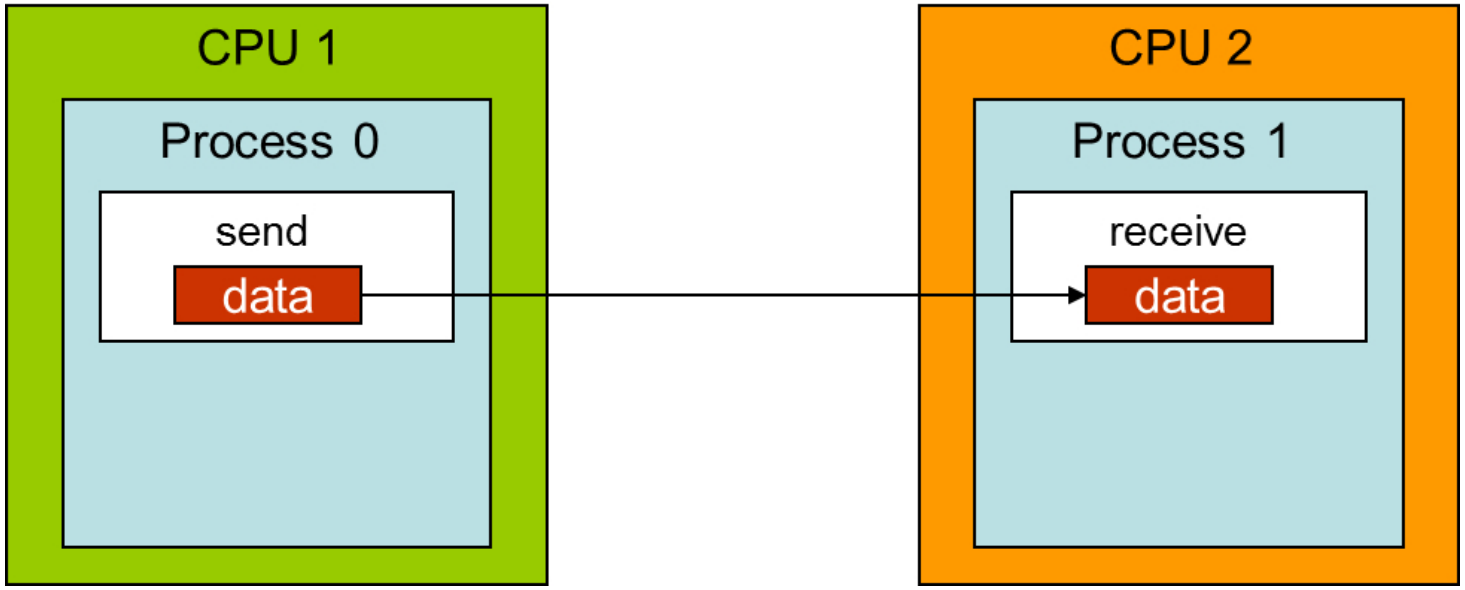



```
int nra = 8;            // 8x8 matrix
int nprocs = 4;         / /by colors
int r_per_prc = nra/nprocs;  //rows per rank

//starting point per rank
int ira_start = myrank*r_per_prc;
//end point per rank
int ira_end = (myrank+1)*r_per_prc-1;

for (i=ira_start; i<=ira_end; i++)
   for (j=0;j<ncb;j++){
      for (k=0;k<nca;k++){
         C[i][j]+= A[i][k]*B[k][j];
```

MPI Data Communication

- Environment management functions
  - Initialization and termination
- Point-to-point communication functions
  - Message transfer from one process to another
- Collective communication functions
  - Message transfer involving all processes in a communicator

Point-to-point Communication



Blocking vs. non-blocking

- Communication between a pair of processes
  - Blocking: not return until conditions met
    - Send: when buffer is available for reuse
    - Receive: returns only after it contains the data in its buffer
  - Non-blocking:
    - Create request for send/receive, get back a handle and terminate.
    - Returns from call without waiting for task to complete
    - Sender side allows overlapping computation with communication

Send/Receive Modes (blocking)

- Standard: message sent (no guarantee that the receive has started). It is up to MPI to decide what to do

  int **MPI_Send**(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
- Buffered: int **MPI_Bsend**(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
- Synchronous: A send will not complete until a matching receive has been posted

  int **MPI_Ssend**(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
- Ready: int **MPI_Rsend**(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

One blocking receive:

int **MPI_Recv**(void *buf, int count, MPI_Datatype dtype, int source, int tag, MPI_Comm comm, MPI_Status *status)

Blocking send/receive

```
int MPI_Send(void *buf, int count, MPI_Datatype dtype,
int dest, int tag, MPI_Comm comm);

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

Blocking send/receive

```
int MPI_Send(void *buf, int count, MPI_Datatype dtype,
int dest, int tag, MPI_Comm comm);

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

- A MPI message consists of **two parts**
  - Message itself: data body

Blocking send/receive

```
int MPI_Send(void *buf, int count, MPI_Datatype dtype,
int dest, int tag, MPI_Comm comm);

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

- A MPI message consists of **two parts**
  - Message itself: data body
  - Message envelope: routing infomation
- **`status`**: information about received message

What is A Message ?

- Collection of data (array) of MPI data types
  - Basic data types such as int /integer, float/real
  - Derived data types
- Message "envelope" – source, destination, tag, communicator

Standard send/receive (Blocking)

```
int MPI_Send(void *buf, int count, MPI_Datatype dtype,
int dest, int tag, MPI_Comm comm);

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

- buf - address of sending/receive buffer
- count - maximum number of elements in receive buffer
- datatype - datatype of each sending/receive buffer element
- dest/source - rank of dest / source
- tag - message tag
- comm – communicator
- status - status object,

  e.g.,status.MPI_SOURCE, status.MPI_TAG)

Standard send/receive (Blocking)

- The sending process calls the MPI_SEND function
  - C: `int MPI_Send(void *buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm)`
  - Fortran:

```
MPI_SEND(BUF, COUNT, DTYPE, DEST, TAG,COMM,IERR)
```

- The receiving process calls the MPI_RECV function
  - C: `int MPI_Recv(void *buf, int count, MPI_Datatype dtype, int source, int tag, MPI_Comm comm, MPI_Status *status)`
  - Fortran:

```
MPI_RECV(BUF, COUNT, DTYPE, SOURCE, TAG, COMM,
  STATUS, IERR)
```

MPI Data Types (C)

MPI_CHAR
MPI_SHORT
MPI_INT
MPI_LONG MPI_UNSIGNED_CHAR
MPI_UNSIGNED_SHORT MPI_UNSIGNED_LONG
MPI_UNSIGNED MPI_FLOAT MPI_DOUBLE
MPI_LONG_DOUBLE MPI_BYTE
MPI_PACKED

MPI Data Types (Fortran)

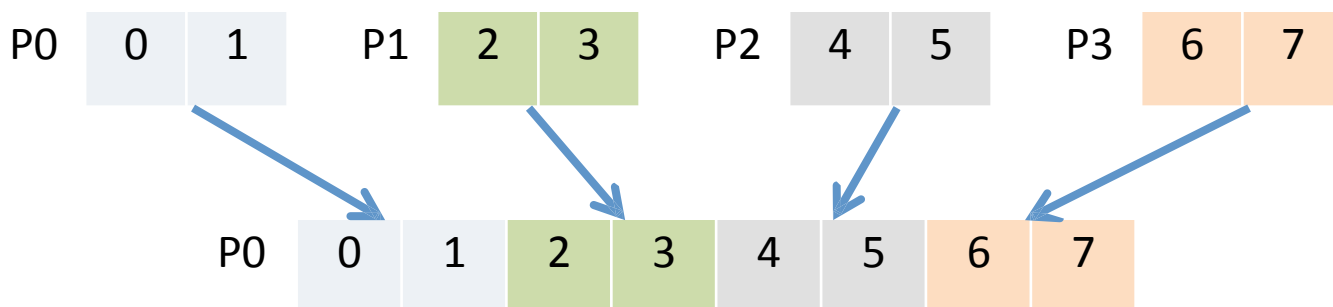
- MPI_INTEGER – INTEGER
- MPI_REAL – REAL
- MPI_DOUBLE_PRECISION – DOUBLE PRECISION
- MPI_CHARACTER – CHARACTER(1)
- MPI_COMPLEX – COMPLEX
- MPI_LOGICAL – LOGICAL
- . . .

Example 1

```
..
MPI_Status Stat;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) {
   dest = 1; source = 1;
   rc = MPI_Send(&outmsg,1,MPI_CHAR,dest,tag,MPI_COMM_WORLD);
   rc = MPI_Recv(&inmsg,1,MPI_CHAR,source,tag,MPI_COMM_WORLD,&Stat);
}
else if (rank == 1) {
   dest = 0; source = 0;
   rc = MPI_Recv(&inmsg,1,MPICHAR,source,tag,MPI_COMM_WORLD,&Stat);
   rc = MPI_Send(&outmsg,1,MPI_CHAR,dest,tag,MPI_COMM_WORLD);
}
rc = MPI_Get_count(&Stat, MPI_CHAR, &count);
printf("Task %d: Received %d char(s) from task %d with tag %d \n",
rank, count, Stat.MPI_SOURCE, Stat.MPI_TAG);
MPI_Finalize(); }
```

Example 2: Gathering Array Data

Gather some array data from each process and place it in the memory of the root process



Example: Gathering Array Data

```
…
integer,allocatable :: array(:)
! Initialize MPI
call mpi_init(ierr)
call mpi_comm_size(mpi_comm_world,nprocs,ierr)
call mpi_comm_rank(mpi_comm_world,myid,ierr)
! Initialize the array
allocate(array(2*nprocs))
array(1)=2*myid  array(2)=2*myid+1
! Send data to the root process
if (myid.eq.0) then do
  i=1,nprocs-1
    call mpi_recv(array(2*i+1),2,mpi_integer,i,i,status,ierr)
  enddo
  write(*,*) "The content of the array:"
  write(*,*) array
else
  call mpi_send(array,2,mpi_integer,0,myid,ierr)
endif
```

Source

dest

tag

MPI_Barrier

MPI_Barrier(MPI_COMM_WORLD)

- Force all the processes within a communicator to wait for each other
- All processes halt until every single one has reached the barrier

MPI_Wtime()

MPI_Wtime()
Returns an elapsed time on the calling processor
Time in seconds since an arbitrary time in the past

MPI_Get_count()

```
MPI_Get_count(MPI_Status* status,
              MPI_Datatype datatype, int* count)
```

MPI_Status: Information about status

e.g., status.MPI_SOURCE, status.MPI_TAG

MPI_Probe()

```
MPI_Probe(int source, int tag, MPI_Comm comm,
          MPI_Status* status)
```

- Similar to MPI_Recv() without receiving
- Used to allocate receiving memory dynamically

Blocking Operations

- MPI_SEND and MPI_RECV are blocking operations
  - They will not return from the function call until the communication is completed
  - When a **blocking send** returns, the value(s) stored in the variable can be **safely overwritten**
  - When a **blocking receive** returns, the data has been received and is **ready to be used**

Deadlock (1)

Deadlock occurs when both processes await each other to make progress

```
//    Exchange data  between two processes
if  (process 0)
        Receive data from process 1
        Send data to process 1
if  (process 1)
        Receive data from process 0
        Send data to process 0
```

- Guaranteed deadlock!
- Both receives wait for data, but no send can be called until the receive returns

Deadlock (2)

- How about this one?

```
//   Exchange data  between two processes
If  (process 0)
        Receive data from process 1
        Send data to process 1
If  (process 1)
        Send data to process 0
        Receive data from process 0
```

- No deadlock !
- P0 receives the data first, then sends the data to P1
- There will be performance penalty due to serialization of potentially concurrent operations.

Deadlock (3)

- And this one?

```
//    Exchange data  between two processes
If  (process 0)
        Send data to process 1
        Receive data from process 1
If  (process 1)
        Send data to process 0
        Receive data from process 0
```

- It depends
- If one send returns, then we are OKAY - most MPI implementations buffer the message, so a send could return even before the matching receive is posted.
- If the message is too large to be buffered, deadlock will occur.

Blocking vs. Non-blocking

- Blocking operations are data corruption proof
- Until a matching receive has signaled that it is ready to receive, a blocking send may continue to wait
  - Possible deadlock
  - Performance penalty
- Non-blocking operations allow overlap of completion and computation
  - The sender process can work on other tasks between the initialization and completion
  - Should be used whenever possible

Non-blocking Operations (asynchronous)

- Separate initialization of a send or receive from its completion

- Two calls are required to complete a send or receive
  - Initialization
    - Send: `MPI_ISEND`
    - Receive: `MPI_IRECV`
  - Completion: `MPI_WAIT or MPI_TEST`

Non-blocking Isend and Irecv

✧ MPI_ISEND function

- C: `int MPI_Isend(void *buf,int count,MPI_Datatype dtype, int dest, int tag, MPI_Comm comm, MPI_Request *request)`
- Fortran: `MPI_ISEND(BUF,COUNT,DTYPE,DEST,TAG,REQ,COMM,IERR)`

✧ MPI_IRECV function

- C: `int MPI_Irecv(void *buf, int count, MPI_Datatype dtype, int source, int tag, MPI_Comm comm, MPI_Status*status, MPI_Request *request)`
- Fortran: `MPI_IRECV(BUF, COUNT, DTYPE, SOURCE, TAG,COMM, STATUS, REQ, IERR)`

Waiting for Completion: MPI_Wait

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

- request: communication request
- Status: status object

- Force the process in "blocking mode" waiting to finish on a given request
- As soon as the request is complete, a status instance is returned in status.

Non-blocking Example (C)

```
if (myrank == 0) {
   MPI_Isend(&x, 1, MPI_INT, 1, tag1, MPI_COMM_WORLD, &send_req);
    printf("Process %d receiving from process 1\n", me);
    MPI_Irecv(&x, 1, MPI_INT, 1, tag2, MPI_COMM_WORLD, &recv_req);
// do computations here while waiting for communication
      …
   MPI_Wait(&send_req, &status);
   MPI_Wait(&recv_req, &status);
   printf("Process %d ready\n", myrank);
}
else if (myrank == 1)  {
   MPI_Irecv(&y, 1, MPI_INT, 0, tag1, MPI_COMM_WORLD,  &recv_req);
   MPI_Isend(&y, 1, MPI_INT, 0, tag2, MPI_COMM_WORLD, &send_req);
    MPI_Wait(&recv_req, &status)
    MPI_Wait(&send_req, &status);
}
```

Non-blocking Example (Fortran)

```
integer reqids,reqidr
integer status(mpi_status_size)
if (myid.eq0) then
    call mpi_isend(to_p1,n,mpi_integer,1,100,mpi_comm_world,reqids,ierr)
    call mpi_irecv(from_p1,n,mpi_integer 1,101,mpi_comm_world,reqidr,ierr)
else if (myid.eq.1) then
    call mpi_isend(to_p0,n,mpi_integer,0,101,mpi_comm_world,reqids,ierr)
    call mpi_irecv(from_p0,n,mpi_integer,0,100,mpi_comm_world,reqidr,ierr)
endif

call mpi_wait(status, reqitds, ierr)
Call mpi_wait(status, reqidr, ieer)
```

Test for Completion: MPI_Test

```
int MPI_Test(MPI_Request *request,int *flag,
             MPI_Status *status)
```

- request: communication request
- flag: true if operation completed
- status: status object

- Check if the request can be completed.
- If it can, the request is automatically completed and data is transferred

Example: MPI_Test

```
MPI_Irecv(&data, 1, MPI_INT, MPI_ANY_SOURCE, tag,
          MPI_COMM_WORLD, &mpi_request);
MPI_Test(&mpi_request, &req_complete, &status);

while (! req_complete) {
     //Do other stuff before message
    ...
     MPI_Test(&mpi_request,&req_complete,&status);
}
// Do stuff after getting message
...
```

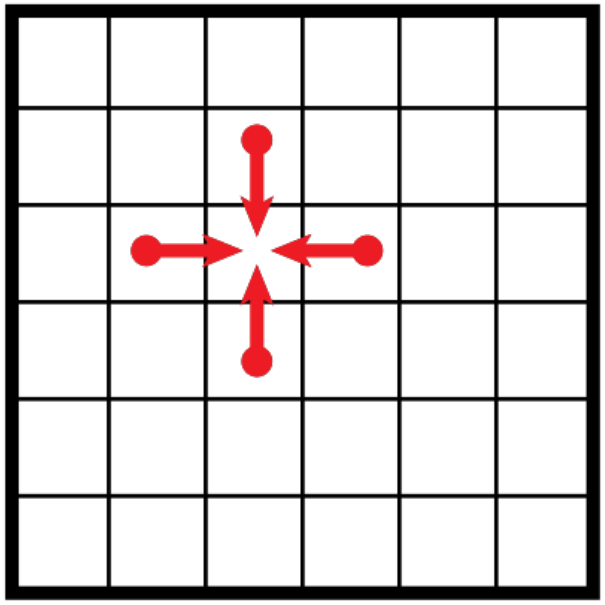
Exercise a2: Find Global Maximum

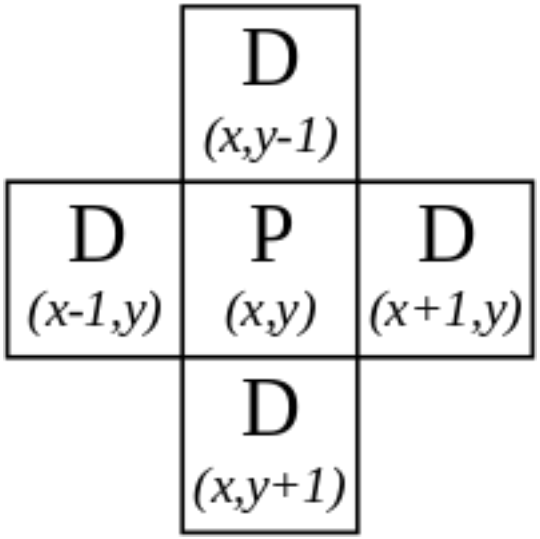
- Goal: Find the global maximum
  - Each process (myrank) randomly generate 10 numbers
  - Each process (myrank) finds its own max_local
  - Root (rank=0) collects max_local from each process to get the max_global

Exercise b2: Matrix Multiplication

- Modify b1 so that each process sends its partial results to the root process
  - The root process should now have the whole matrix
  - Note: b1: each process holds its own part of calculation locally
- Validate the result at the root process

Last Exercise: Laplace Solver



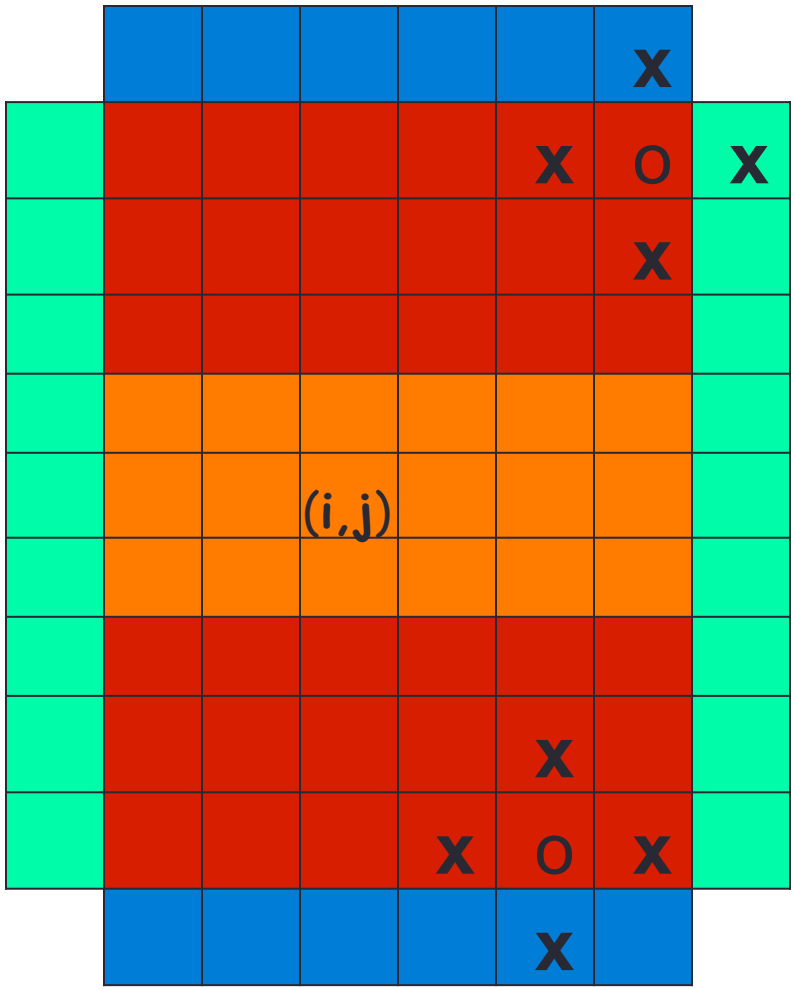


$$P_{x,y} = (D_{x-1,y} + D_{x,y-1} + D_{x+1,y} + D_{x,y+1}) / 4$$

Five-point Finite-Difference Stencil (border cells)

				X	(0,5)
X					X
(2,0)	X				
X					
		(i,j)			
				X	
			X	(7,4)	X

Domain Decomposition



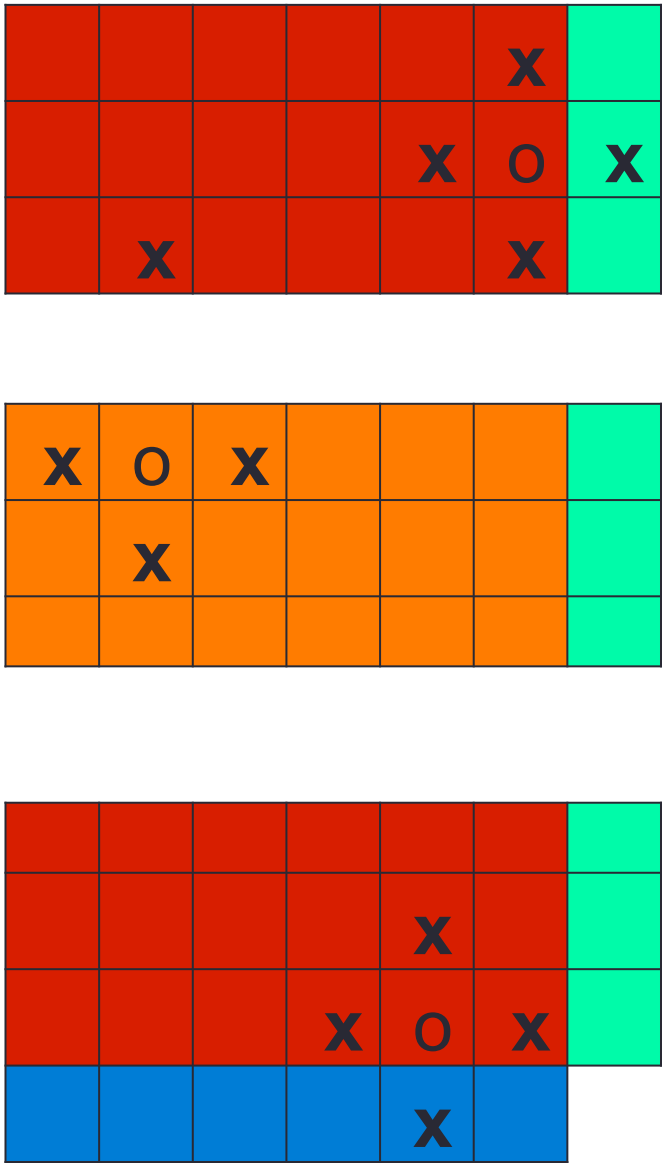
thread 0

thread 1

thread 2

Unknowns at Border Cells – 1D

Five-point finite-difference stencil applied at thread domain border cells require cells from neighboring threads and/or boundary cells.

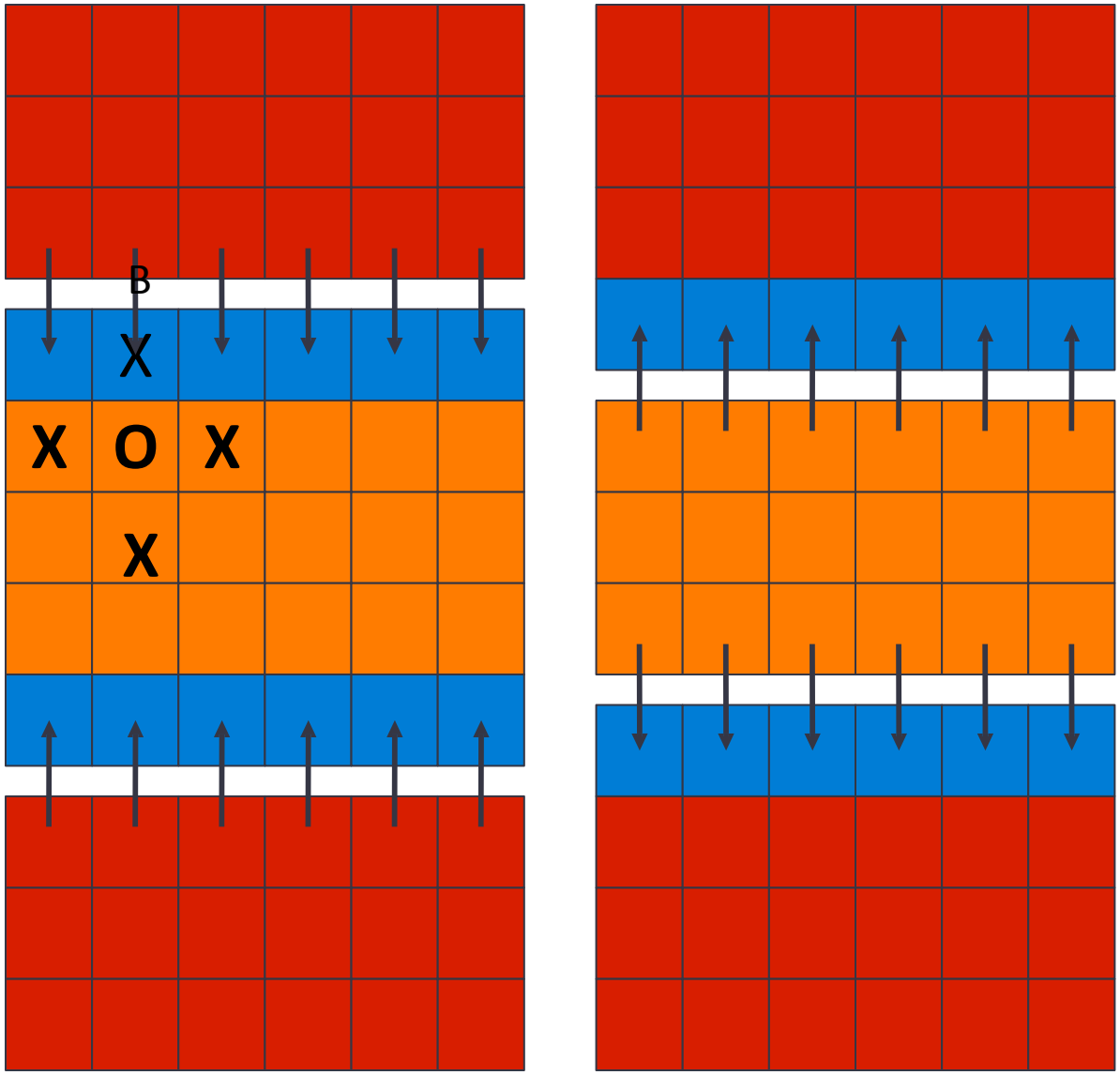


thread 0

thread 1

current thread

thread 2



Exercise c: Laplace Solver

- Goal: develop a working MPI Laplace solver
  - Distribute the workload in 1D manner
  - Initialize the sub-matrix at each process and set the boundary values
  - Calculate new values
  - At the end of each iteration
    - Exchange boundary data with neighbors
    - Find the global convergence error and distribute to all processes
    - Update entire matrix using new values

Conclusions

- Standardized
  - With efforts to keep it evolving (MPI 3.0)
- Portability
  - MPI implementations are available on almost all platforms
- Scalability
  - In the sense that it is not limited by the number of processors that can access the their local memory space
- Popularity
  - De Facto programming model for distributed memory machines
- Nearly every big academic or commercial simulation or data analysis running on multiple nodes uses MPI directly or indirectly

Continue…

- MPI Part 2: Collective communications
- MPI Part 3: Understanding MPI applications