www.locuz.com

Distributed Computing with MPI

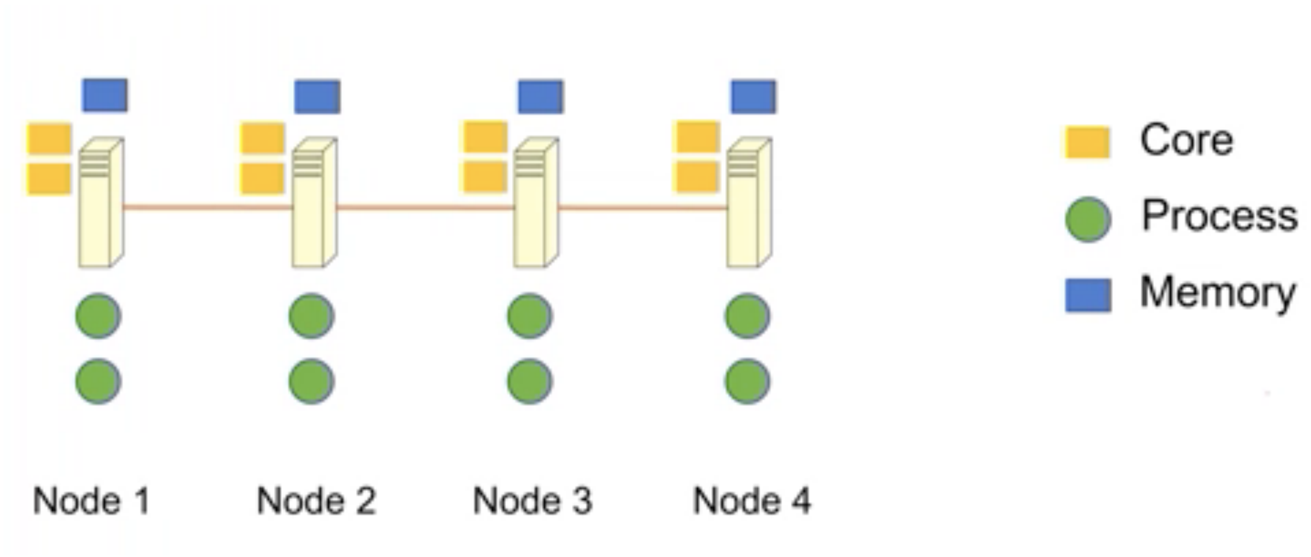
Presented By: Mandeep Kumar

Converge to the Cloud

Agenda

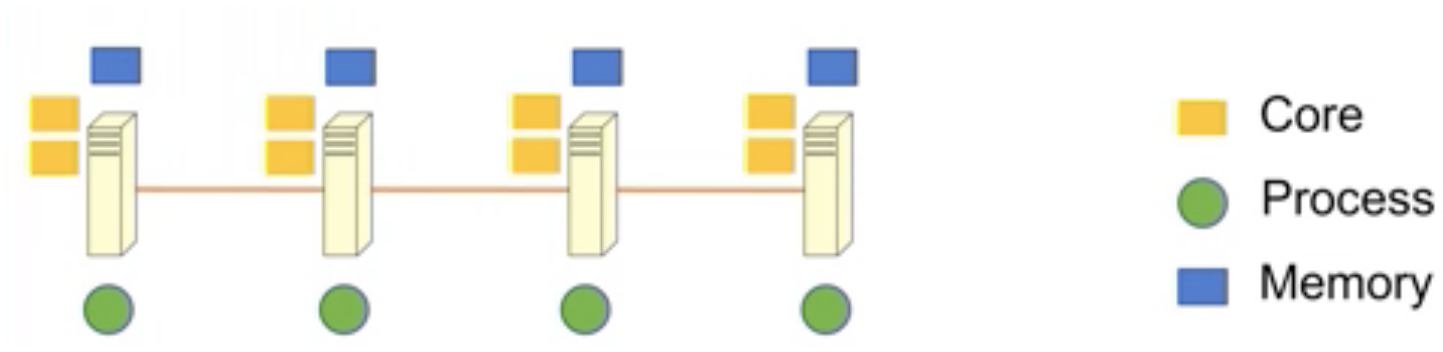
- MPI Introduction
- Communicator and communication
- Point to point communication
- Collective communication

System Details



Distributed Memory (each process has its own address space)

Parallel Execution



Instruction 1
Instruction 2
…

Instruction 1
Instruction 2
…

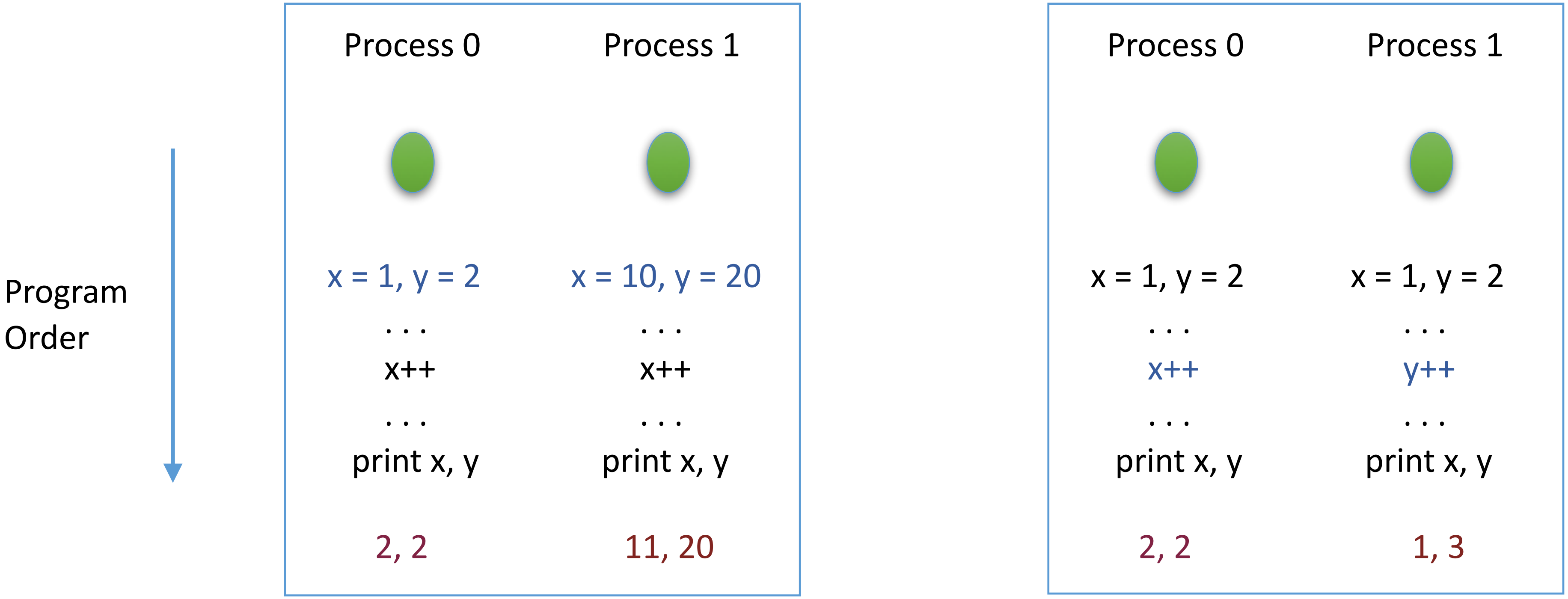
Instruction 1
Instruction 2
…

Instruction 1
Instruction 2
…

parallel.x parallel.x parallel.x parallel.x

parallel.x

Distinct Process Address Space



Program Order

Process 0	Process 1
x = 1, y = 2	x = 10, y = 20
. . .	. . .
x++	x++
. . .	. . .
print x, y	print x, y
2, 2	11, 20

Process 0	Process 1
x = 1, y = 2	x = 1, y = 2
. . .	. . .
x++	y++
. . .	. . .
print x, y	print x, y
2, 2	1, 3

What is MPI?

- MPI stands for Message Passing Interface and is a library specification for message-passing, proposed as a standard by a broadly based committee of vendor, implementors, and users.

- API for distributed- memory programming
  - parallel code that runs across multiple computers (nodes)

- MPI consists of
  - a header file mpi.h
  - a library of routines and functions, and
  - a runtime system

- MPI is for parallel computers, clusters, and heterogeneous networks.

- Use from C/C++, Fortran, Python, R, …

- More than 200 routines

- Using only 10 routines are enough is many cases
  — Problem dependent

Example MPI routines

The following routines are found in nearly every program that uses MPI

- MPI_Init() starts the MPI runtime environment.
- MPI_Finalize() shuts down the MPI runtime environment.
- MPI_Comm_size() gets the number of processes, N_p.
- MPI_Comm_rank() gets the process ID of the current process which is between 0 and $N_p - 1$, inclusive.

(These last two routines are typically called right after MPI_Init().)

Parallel hello to you!

```
#include <stdio.h>
#include "mpi.h"

int main(int argc, char *argv[])
{

  // initialize MPI
  MPI_Init (&argc, &argv);

  printf (“Parallel hello to you!\n");


  // done with MPI
  MPI_Finalize();

}
```

Initialization

Finalization

MPI_Init

- gather information about the parallel job
- set up internal library state
- prepare for communication

MPI Code Execution Steps

- **Compile**

  ```
  mpicc -o program.x program.c
  ```

- **Execute**

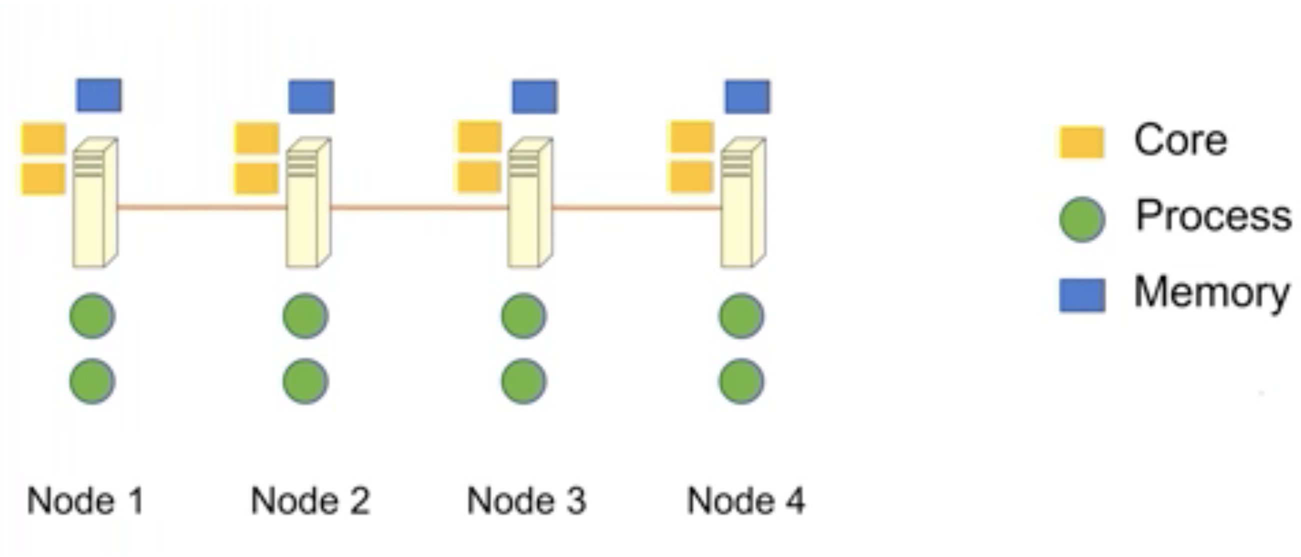
  - `mpirun -np 1 ./program.x` (or mpiexec in place of mpirun)
    — Run 1 process on launch node

  - `mpirun -np 6 ./program.x`
    — Run 6 process on launch node

  - `mpirun -np 6 -f hostfile ./program.x`
    — Run 6 process on the nodes specified in the hostfile

```
<hostfile>

Node1:2
Node2:2
Node3:2

…
```

Multiple Processes on Multiple Cores and Nodes



mpirun -np 8 -f hostfile ./program.x

Communicator

- Communication handle among a group/collection of processes
  — processes are numbered 0,1,… to N-1

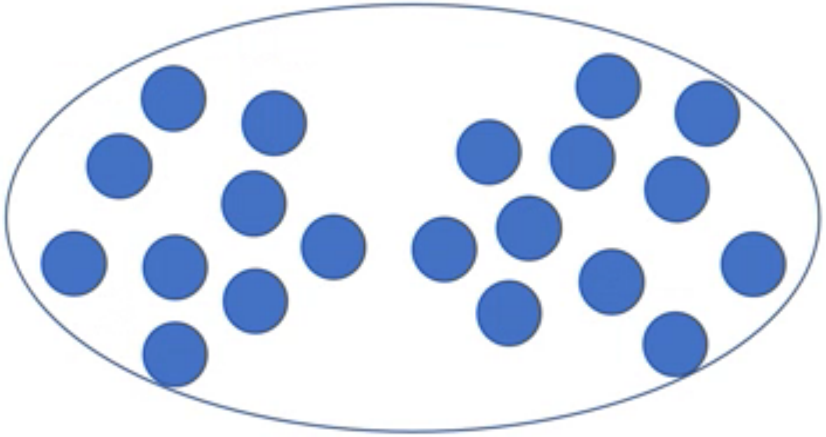
- Default Communicator:
  - MPI_COMM_WORLD
    — contains all processes

- Contains a mapping from MPI process ranks to processor ids

- Memory proportional to #processes in the group

- Query functions:
  - How many processes in total?
    MPI_Comm_size(MPI_COMM_WORLD, &nproc)
  - What is my process ID?
    MPI_Comm_rank(MPI_COMM_WORLD, &rank)

MPI_COMM_WORLD



Required in every MPI communication

Process Identification

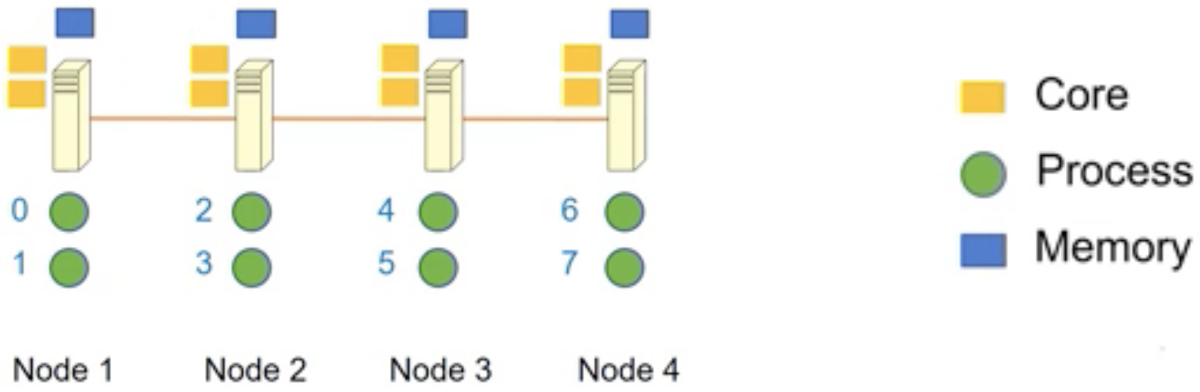
- **MPI_Comm_size:** get the total number of process
- **MPI_Comm_rank:** get my rank

```
#include <stdio.h>
#include “mpi.h"
int main(int argc, char *argv[])
{
  int numtasks, rank, len;
  char hostname[MPI_MAX_PROCESSOR_NAME];

  // initialize MPI
  MPI_Init (&argc, &argv);

  // get number of tasks
  MPI_Comm_size (MPI_COMM_WORLD, &numtasks);

  // get my rank
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);

  // this one is obvious
  MPI_Get_processor_name (hostname, &len);

  printf ("Number of tasks=%d My rank=%d Running on %s\n", numtasks, rank, hostname);
  // done with MPI
  MPI_Finalize();
}
```

Total number of processes

Rank of process

Multiple Processes on Multiple Cores and Nodes



```
mpirun -np 8 -f hostfile ./program.x
```

```
Node1:2
Node2:2
Node3:2
Node4:2
```

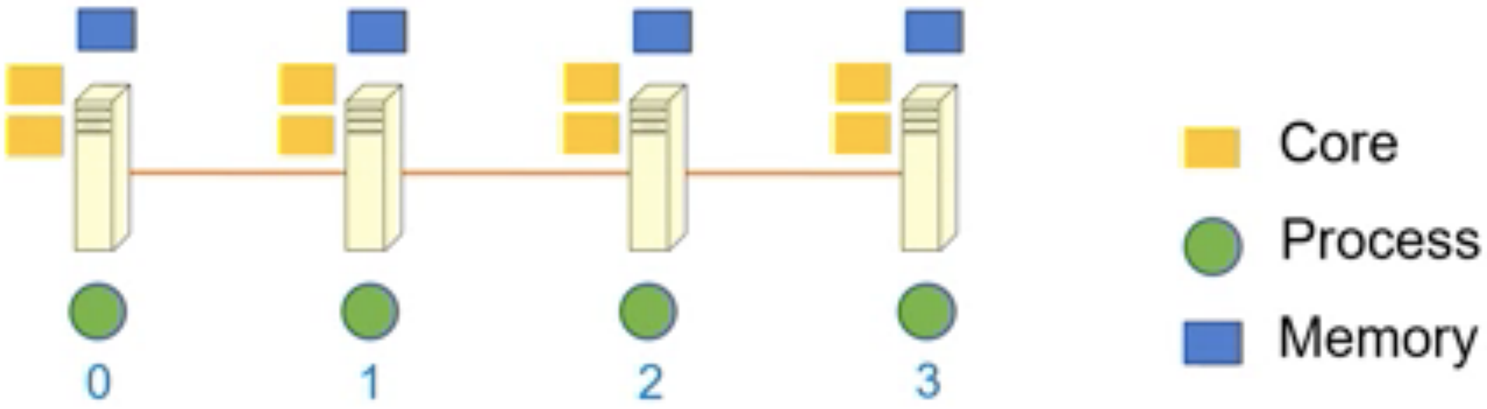
Note:

- All MPI processes (normally) run the same executable
- Each MPI process knows which rank it is
- Each MPI process knows how many processes are part of same job
- The processes run in a non-deterministic order

Sum of Squares of N Numbers

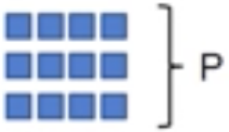
Serial

```
for i = 1 to N
sum += a[i] * a[i]
```



Parallel

```
for i = 1 to N/P
sum += a[i] * a[i]
```



```
for i = 1 to N/P
sum += a[i] * a[i]
```

```
for i = 1 to N/P
sum += a[i] * a[i]
```

```
for i = 1 to N/P
sum += a[i] * a[i]
```

```
for i = 1 to N/P
sum += a[i] * a[i]
```

```
for i = N/P * rank; i < N/P * (rank+1); i++
   localsum += a[i] * a[i]
Collect localsum, add up at one of the ranks
```

Parallel Sum of Array

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "mpi.h"
#define N 100000000

int main(int argc, char *argv[])
{
  int numtasks, rank, len, rc, i, sidx;
  double etime, stime, a[N], value, localsum;
  char hostname[MPI_MAX_PROCESSOR_NAME];

  // initialize MPI
  MPI_Init (&argc, &argv);

  // get number of tasks
  MPI_Comm_size (MPI_COMM_WORLD, &numtasks);

  // get my rank
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
```

```c
  MPI_Get_processor_name (hostname, &len);

 // random initialization
  srand(time(NULL));
  value = abs((numtasks-rank)*(20210401-rand()))%N;
  sidx = rank*N/numtasks;
  for (i=sidx; i<sidx+N/numtasks ; i++)
      a[i] = value;

  // compute local sum
  localsum=0.0;
  stime = MPI_Wtime();
  for (i=sidx; i<sidx+N/numtasks ; i++)
      localsum += a[i];
  etime = MPI_Wtime();

  printf ("%d: Time to sum: %lf\n", rank, etime - stime);
  // done with MPI
  MPI_Finalize();
}
```

Scalability Bottleneck: Memory Bound

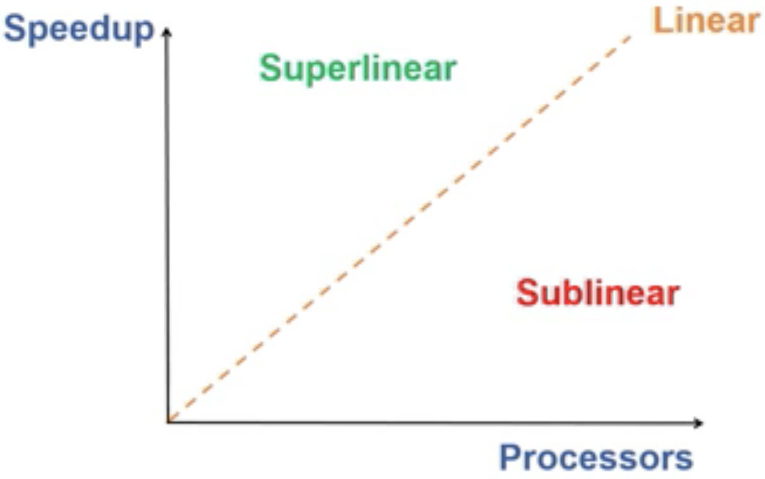
Performance

- Speedup

$$S_P = \frac{\text{Time (1 processor)}}{\text{Time (P processors)}}$$

- Efficiency

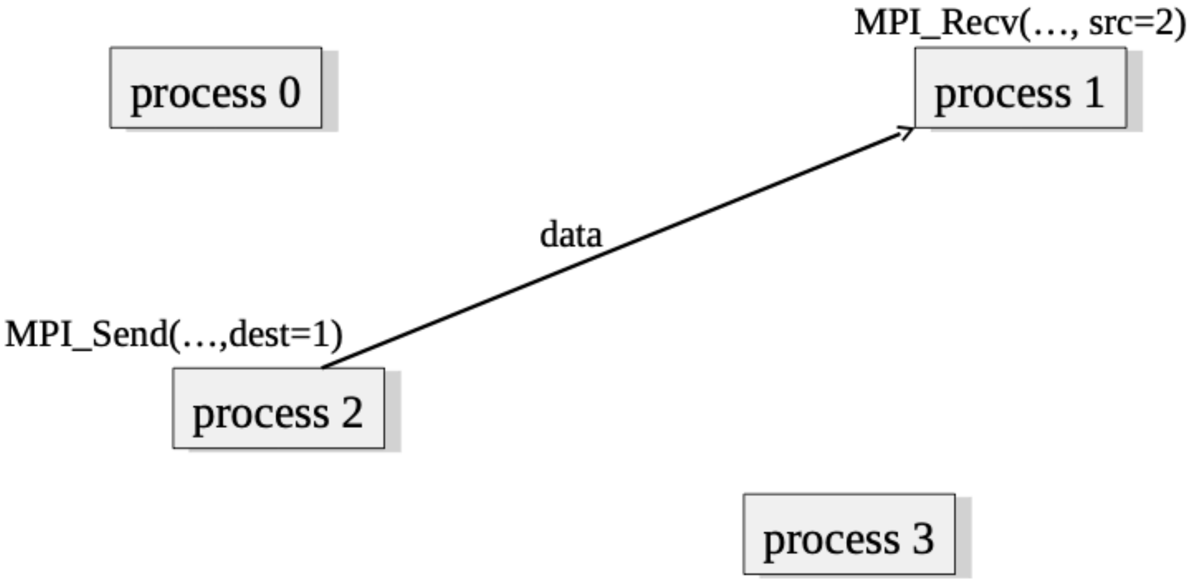
$$E_P = \frac{S_P}{P}$$



MPI Communication Types

- Point-to-point
- Collective

Point-to-Point Communication



MPI_Send

- Send data to another process

```
MPI_Send (buffer, count, datatype, dest, tag, comm)
```

Arguments	**Meanings**
buf	starting address of send buffer
count	# of elements
datatype	data type of each send buffer element
dest	process ID (rank) destination
tag	message tag
comm	communicator

- Examples:

```
C/C++: MPI_Send(&x, 1, MPI_INT, 5 , 0, MPI_COMM_WORLD);
Fortran: MPI_Send(x, 1, MPI_INTEGER, 5, 0, MPI_COMM_WORLD, ierr)
```

MPI_Recv

- Receive data from another process

```
MPI_Recv (buffer, count, datatype, src, tag, comm, status)
```

Arguments	Meanings
buf	starting address of send buffer
count	# of elements
datatype	data type of each send buffer element
dest	process ID (rank) source
tag	message tag
comm	communicator
status	status object (an integer array in Fortran)

- Examples:

```
C/C++: MPI_Recv(&x, 1, MPI_INT, 1 , 0, MPI_COMM_WORLD, &status);
Fortran: MPI_Recv(x, 1, MPI_INTEGER, 1, 0, MPI_COMM_WORLD, status, ierr)
```

Notes on MPI_Recv

- A message is received when the followings are matched:
  - Source (sending process ID/rank)
  - Tag
  - Communicator (e.g. MPI_COMM_WORLD)

- Wildcard values may be used:
  - MPI_ANY_TAG
    (Don’t care what the tag value is)
  - MPI_ANY_SOURCE
    (Don’t care where it comes from; always receive)

Send/Recv Example (C)

- Send an integer array f[N] from process 0 to process 1

```
int f[N], src=0, dest=1;
MPI_Status status;
// ...
MPI_Comm_rank( MPI_COMM_WORLD, &rank);

if (rank == src)           // process “dest” ignores this
MPI_Send(f, N, MPI_INT, dest, 0, MPI_COMM_WORLD);

if (rank == dest)         // process “src” ignores this
MPI_Recv(f, N, MPI_INT, src, 0, MPI_COMM_WORLD, &status);

//...
```

Send/Recv Example (F90)

- Send an integer array f(1:N) from process 0 to process 1

```
integer f(N), status(MPI_STATUS_SIZE), rank, src=0, dest=1, ierr
// ...
call MPI_Comm_rank( MPI_COMM_WORLD, rank, ierr);

if (rank == src) then                !process “dest” ignores this
call MPI_Send(f, N, MPI_INT, dest, 0, MPI_COMM_WORLD, ierr)
end if

if (rank == dest) then               !process “src” ignores this
call MPI_Recv(f, N, MPI_INT, src, 0, MPI_COMM_WORLD, status, ierr)
end if

//...
```

Send/Recv Example (cont'd)

- **Before**

process 0 (send)	**process 1 (recv)**
f[0]=0	f[0]=0
f[1]=1	f[1]=0
f[2]=2	f[2]=0

- **After**

process 0 (send)	**process 1 (recv)**
f[0]=0	f[0]=0
f[1]=1	f[1]=1
f[2]=2	f[2]=2

Blocking

- Function call does not return until the communication is complete MPI_Send and MPI_Recv are blocking calls

- Calling order matters
  - it is possible to wait indefinitely, called “deadlock”
  - improper ordering results in serialization (loss of performance)

Deadlock

- This code always works:

```
MPI_Comm_rank(comm, &rank);

if (rank == 0) {
     MPI_Send(sendbuf, count, MPI_INT, 1, tag, comm);
     MPI_Recv(recvbuf, count, MPI_INT, 1, tag, comm, &stat);
} else { // rank==1
     MPI_Recv(recvbuf, count, MPI_INT, 0, tag, comm, &stat);
     MPI_Send(sendbuf, count, MPI_INT, 0, tag, comm);
}
```

Deadlock

- This code deadlocks:

```
MPI_Comm_rank(comm, &rank);

if (rank == 0) {
     MPI_Recv(recvbuf, count, MPI_INT, 1, tag, comm, &stat);
     MPI_Send(sendbuf, count, MPI_INT, 1, tag, comm);
} else { // rank==1
     MPI_Recv(recvbuf, count, MPI_INT, 0, tag, comm, &stat);
     MPI_Send(sendbuf, count, MPI_INT, 0, tag, comm);
}
```

reason: MPI_Recv on process 0 waits indefinitely and never returns.

Non-blocking

- Function call returns immediately, without completing data transfer
  - Only “starts” the communication (without finishing)
  - MPI_Isend and MPI_Irecv
  - Need an additional mechanism to ensure transfer completion (MPI_Wait)
- Avoid deadlock
- Possibly higher performance
- Example: MPI_Isend & MPI_Irecv

MPI_Isend

```
MPI_Isend (buffer, count, datatype, dest, tag, comm, request)
```

- Similar to MPI_Send, except the last argument “request”
- Typical usage:

```
MPI_Request request_X, request_Y;
MPI_Isend(…, &request_X);
MPI_Isend(…, &request_Y);

// … some ground-breaking computations …

MPI_Wait(&request_X, …);
MPI_Wait(&request_Y, …);
```

MPI_Irecv

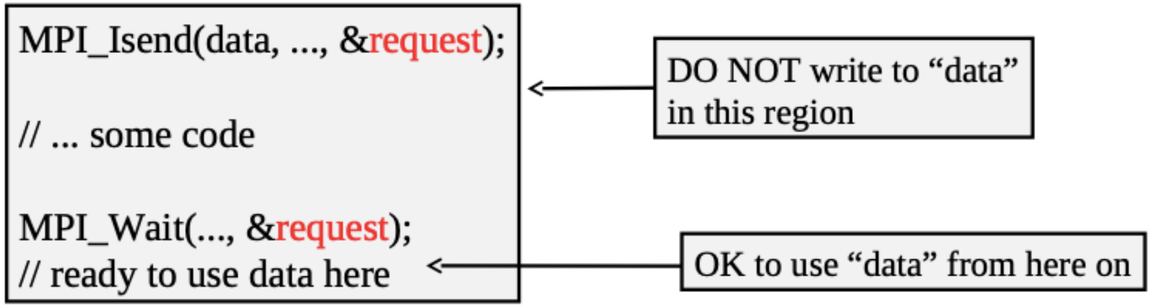
```
MPI_Irecv (buffer, count, datatype, dest, tag, comm, request)
```

- Similar to MPI_Recv, except the last argument “request”
- Typical usage:

```
MPI_Request request_X, request_Y;
MPI_Irecv(…, &request_X);
MPI_Irecv(…, &request_Y);

// … more ground-breaking computations …

MPI_Wait(&request_X, …);
MPI_Wait(&request_Y, …);
```

Caution about MPI_Isend and MPI_Irecv

- The sending process should not access the send buffer until the send completes



```
MPI_Isend(data, ..., &request);

// ... some code

MPI_Wait(..., &request);
// ready to use data here
```

DO NOT write to “data” in this region

OK to use “data” from here on

MPI_Wait

```
MPI_Wait (MPI_Request, MPI_Status)
```

- Wait for an MPI_Isend/recv to complete
- Use the same "request" used in an earlier MPI_Isend or MPI_Irecv
- If they are multiple requests, one can use
  MPI_Waitall(count, request[], status[]);
  request[] and status[] are arrays.

Other variants of MPI Send/Recv

- MPI_Sendrecv
  - send and receive in one call
- Mixing blocking and non-blocking calls
  - e.g. MPI_Isend + MPI_Recv
- MPI_Bsend
  - buffered send
- MPI_Ibsend
- ... (see MPI standard for more)

Synchronization (MP_Barrier)

- MPI_Barrier (comm)
- Every rank needs to call this function (for true synchronization)
- Caller returns only after all processes have entered the call

```
printf(“Before barrier”);
MPI_Barrier (MPI_COMM_WORLD);
printf(“After barrier”);
```

Collective Communication

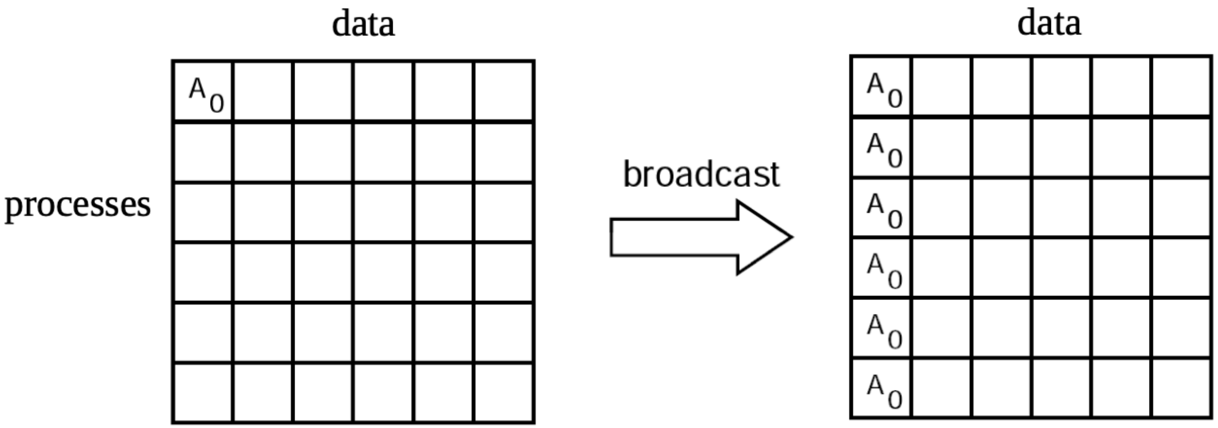
- One to all
  - MPI_Bcast, MPI_Scatter
- All to one
  - MPI_Reduce, MPI_Gather
- All to all
  - MPI_Alltoall

Implicit Synchronization

MPI_Bcast

```
MPI_Bcast (buffer, count, datatype, root, comm)
```

- Broadcasts a message from “root” process to all other processes in the same communicator



MPI_Bcast Example

- Broadcast 100 integers from process “3” to all other processes

C/C++

```
MPI_Comm comm;
int array[100];
// …
MPI_Bcast(array, 100, MPI_INT, 3, comm);
```

Fortran

```
INTEGER comm
integer array(100)
// …
call MPI_Bcast(array, 100, MPI_INTEGER, 3, comm, ierr)
```

MPI_Bcast vs MPI_Send+MPI_Recv

```
if (world_rank == root) {
   // If we are the root process, send our data to everyone
  for (int i = 0; i < world_size; i++) {
        if (i != world_rank) { MPI_Send(data, count, datatype, i, 0, communicator);}
     }
 } else { // if we are a receiver process, receive the data from the root
    MPI_Recv(data, count, datatype, root, 0, communicator, MPI_STATUS_IGNORE);
   }
```

Vs

```
MPI_Bcast(data, num_elements, MPI_INT, 0, MPI_COMM_WORLD);
```

A simple way to speedup:

— Use tree based communication

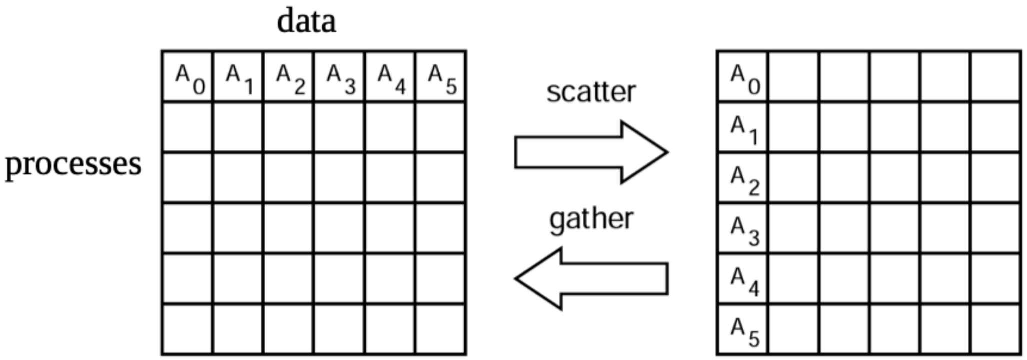
MPI_Gather & MPI_Scatter

- Gathers values from all processes to a root process

```
MPI_Gather (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)
```

- Scatters values to all processes from a root process

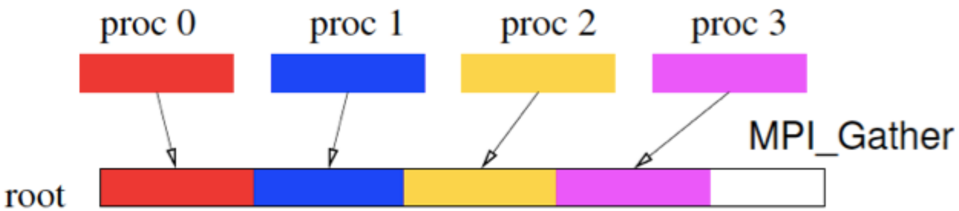
```
MPI_Scatter (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)
```

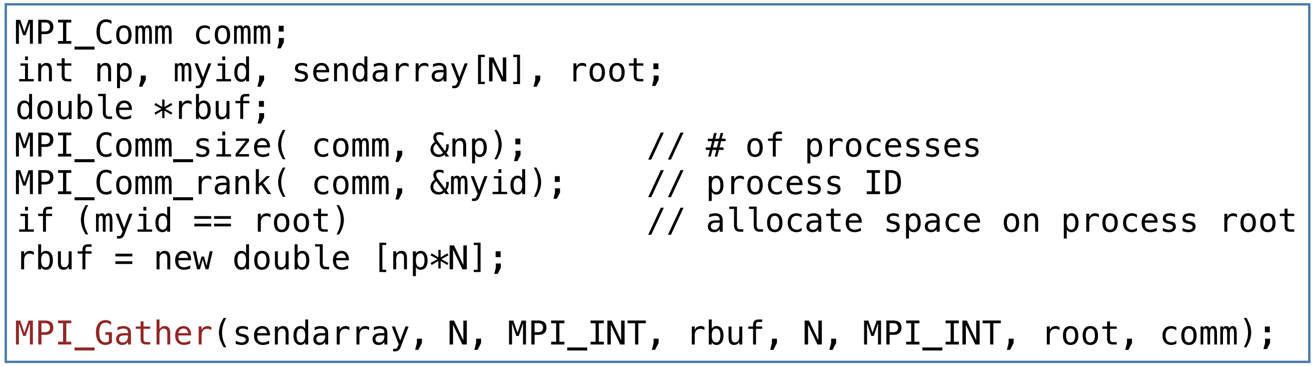


- When gathering, make sure the root process has big enough memory to hold the data (especially when you scale up the problem size).

Non blocking versions: MPI_IGather and MPI_IScatter

MPI_Gather Example





```
MPI_Comm comm;
int np, myid, sendarray[N], root;
double *rbuf;
MPI_Comm_size( comm, &np);      // # of processes
MPI_Comm_rank( comm, &myid);    // process ID
if (myid == root)               // allocate space on process root
rbuf = new double [np*N];

MPI_Gather(sendarray, N, MPI_INT, rbuf, N, MPI_INT, root, comm);
```

Scatter Gather Example

```
if (world_rank == 0) {rand_nums = create_rand_nums(elements_per_proc ∗ world_size);}

// Create a buffer that will hold a subset of the random numbers
float ∗sub_rand_nums = malloc(sizeof(float) ∗ elements_per_proc);

// Scatter the random numbers to all processes
MPI_Scatter(rand_nums, elements_per_proc, MPI_FLOAT, sub_rand_nums, elements_per_proc, MPI_FLOAT,
0, MPI_COMM_WORLD);

// Compute the average of your subset
float sub_avg = compute_avg(sub_rand_nums, elements_per_proc);

// Gather all partial averages down to the root process
float ∗sub_avgs = NULL;

if ( world_rank == 0) {sub_avgs = malloc(sizeof(float) ∗ world_size);}

MPI_Gather(&sub_avg, 1, MPI_FLOAT, sub_avgs, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);

// Compute the total average of all numbers
if (world_rank == 0) {float avg = compute_avg(sub_avgs, world_size);}
```

Variations of MPI_Gather/Scatter

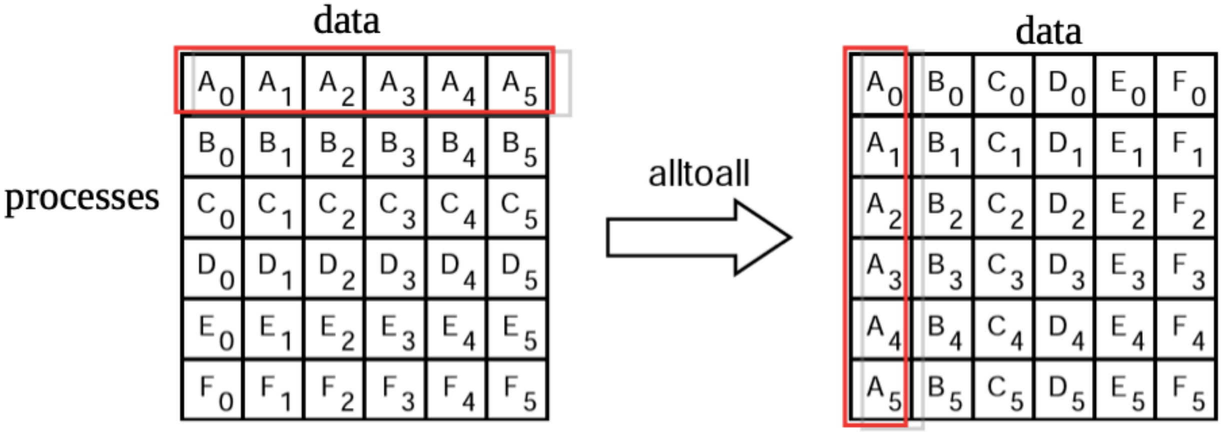
- Variable data size
  - MPI_Gatherv
  - MPI_Scatterv

- Gather + broadcast (in one call)
  - MPI_Allgather All processes send same amount of data
  - MPI_Allgatherv Processes may send variable amount of data

MPI_Alltoall

```
MPI_Alltoall (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)
```

The j-th block sendbuf from process i is received by process j and is placed in the i-th block of recvbuf:



A_0	A_1	A_2	A_3	A_4	A_5
B_0	B_1	B_2	B_3	B_4	B_5
C_0	C_1	C_2	C_3	C_4	C_5
D_0	D_1	D_2	D_3	D_4	D_5
E_0	E_1	E_2	E_3	E_4	E_5
F_0	F_1	F_2	F_3	F_4	F_5

A_0	B_0	C_0	D_0	E_0	F_0
A_1	B_1	C_1	D_1	E_1	F_1
A_2	B_2	C_2	D_2	E_2	F_2
A_3	B_3	C_3	D_3	E_3	F_3
A_4	B_4	C_4	D_4	E_4	F_4
A_5	B_5	C_5	D_5	E_5	F_5

MPI_Reduce

```
MPI_Reduce (sendbuf, recvbuf, count, datatype, op, root, comm)
```

- Apply operation op to sendbuf from all processes and return result in the recvbuf on process “root”.
- op: MIN, MAX, SUM, PROD, …
- Example:

MPI_Reduce (…, MPI_MAX, …) Output: 21
MPI_Reduce (…, MPI_MIN, …). Output: 1

21
5
1
8
3
2
13

MPI Resources on the Net

1. MPI Tutorial

Introduction, background, and basic information. Topics include MPI Environment Management, Point-to-Point Communications, and Collective Communications routines
https://computing.llnl.gov/tutorials/mpi/

2. Laplace Exercise

Serial and parallel exercise, domain decomposition, C and Fortran template codes.

https://docplayer.net/61527559-Laplace-exercise-john-urbanic-parallel-computing-scientist-pittsburgh- supercomputing-center-copyright-2017.html

3. Home page of Prof. William D. Gropp

http://wgropp.cs.illinois.edu/

4. Book: Using MPI- Portable Parallel Programming with the Message-Passing Interface by William Gropp, Ewing Lusk and Anthony Skjellum (MIT Press, Third edition)
https://ieeexplore.ieee.org/book/6267273

5. Book: Introduction to Parallel Computing by Gramma, Gupta, Karypis and Kumar (Pearson)

https://www-users.cs.umn.edu/~karypis/parbook/

6. Book: Parallel Programming in C with MPI and OpenMP by Michale J Quinn (McGrawhill)

https://www.amazon.in/Parallel-Programming-C-MPI-OpenMP/dp/0072822562

7. Book: Iterative Methods for Sparse Linear Systems by Yousef Saad

https://www-users.cs.umn.edu/~saad/IterMethBook_2ndEd.pdf

8. NPTEL Course on Matrix Solvers- Somnath Roy

Topic covered are Direct solvers, Iterative solvers, Krylov methods, Preconditioners, Domain Decomposition and Multigrid Methods
https://nptel.ac.in/courses/111/105/111105111/

Thanks!