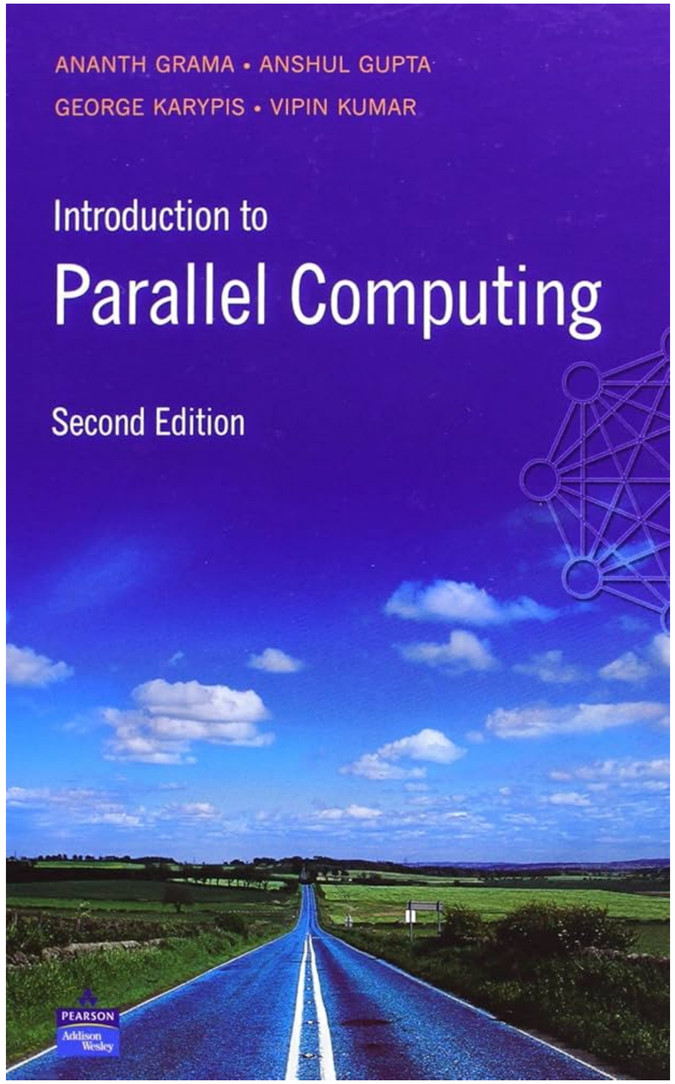
A Beginner’s Guide to MPI

By: Parsa Toopchinezhad

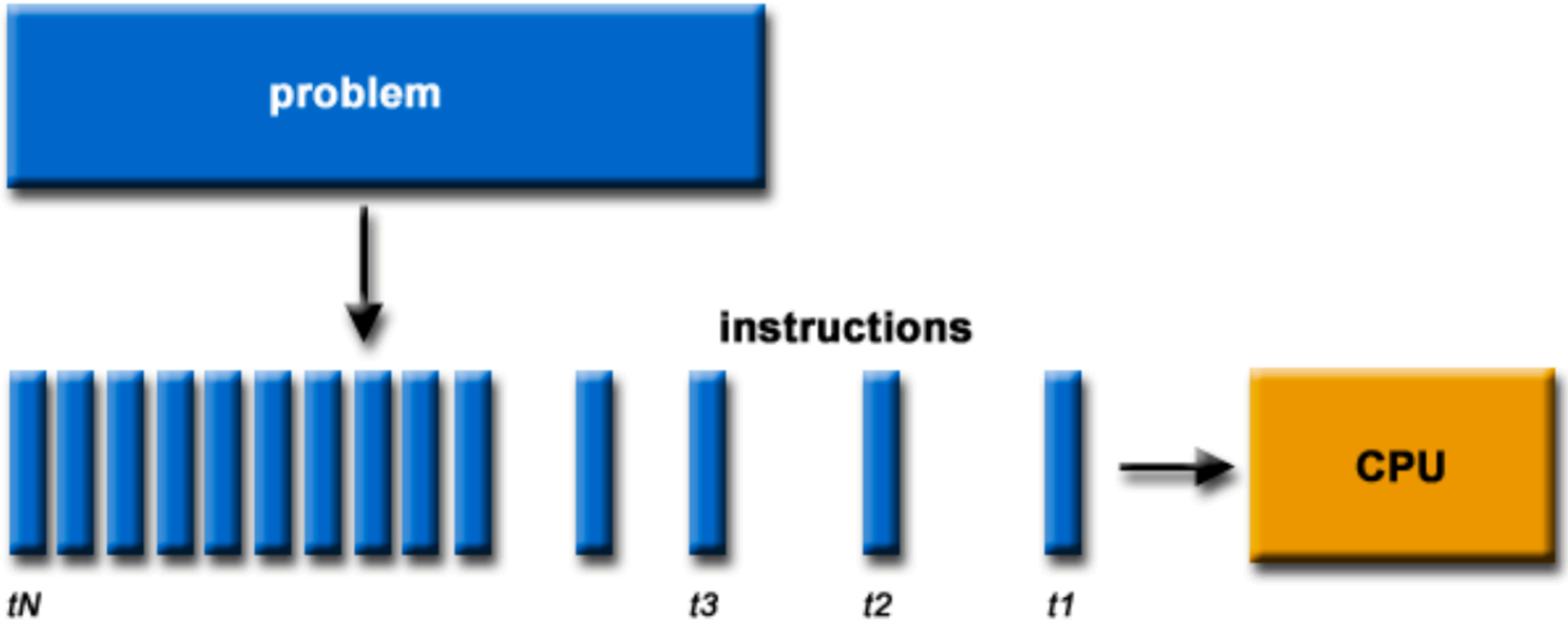
Concordia University - COMP428 - Fall 2025

Textbook


ANANTH GRAMA • ANSHUL GUPTA
GEORGE KARYPIS • VIPIN KUMAR
Introduction to
Parallel Computing
Second Edition
PEARSON
Addison Wesley

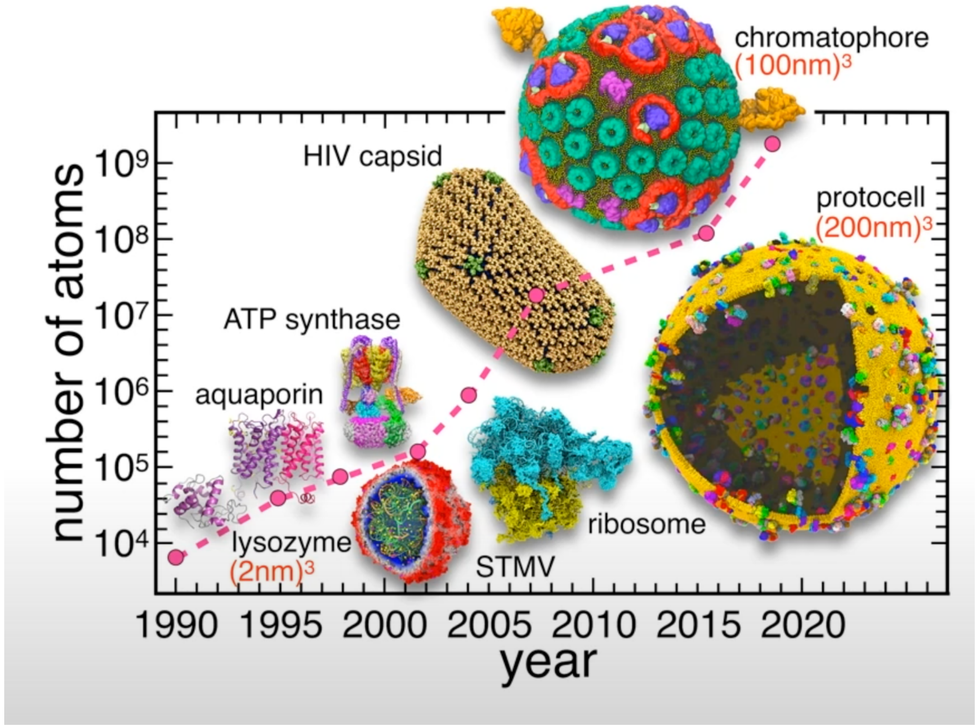
Part I: Evolution of Computing

Serial Programming



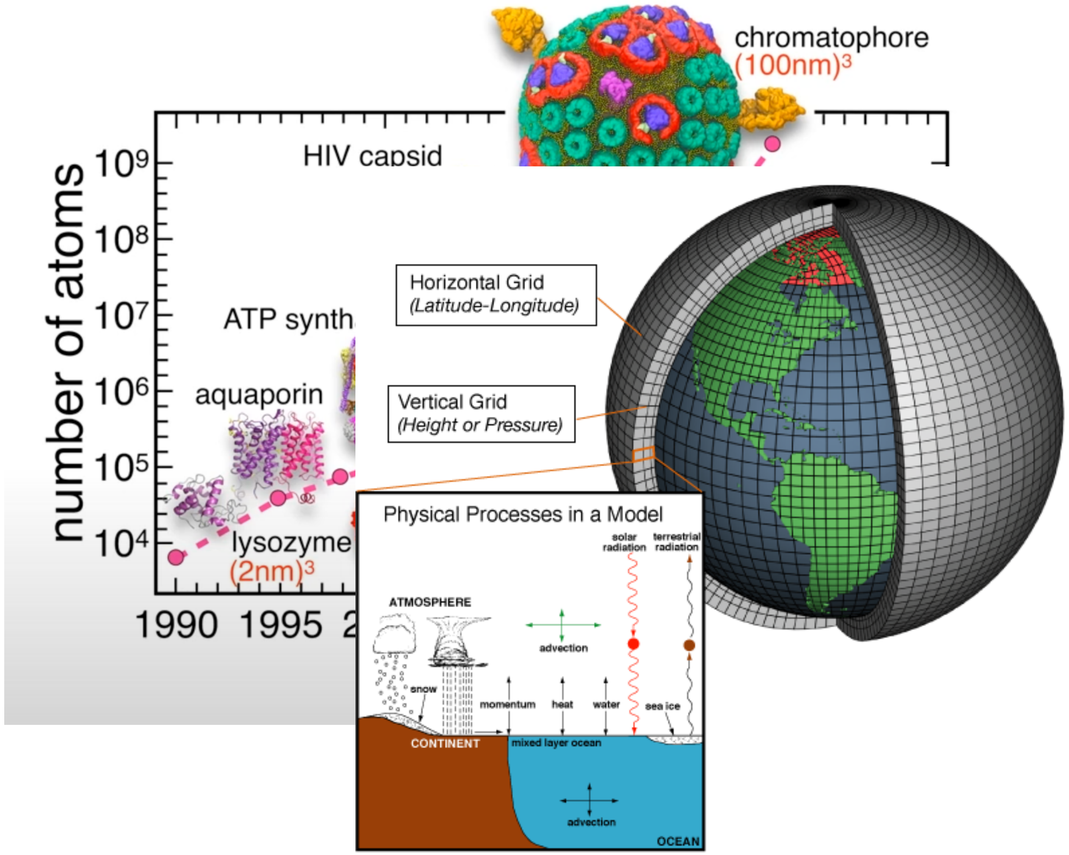
Our Never-Ending Thirst for Compute

- Important applications are compute intensive:
    - Drug Discovery



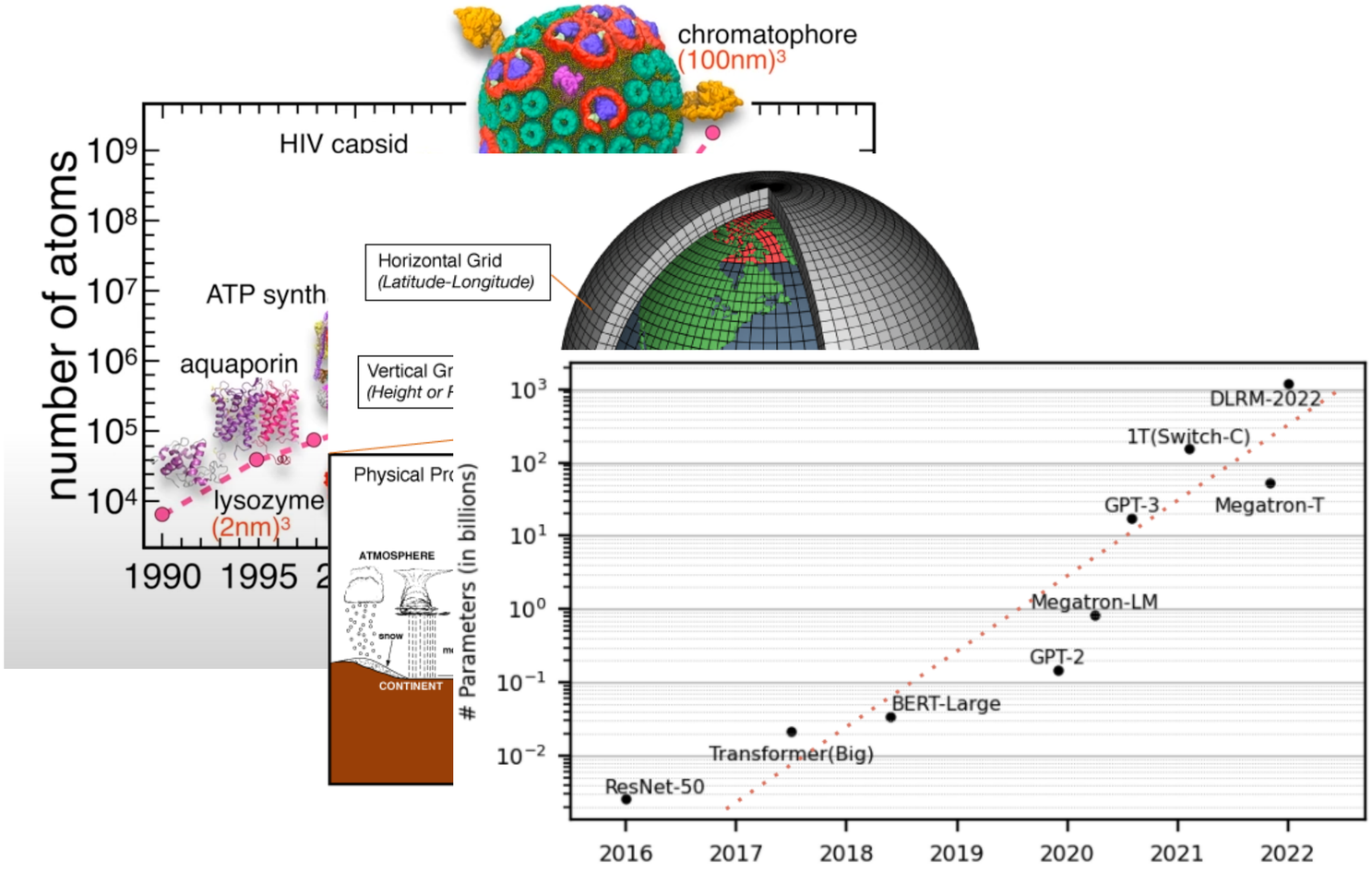
Our Never-Ending Thirst for Compute

- Important applications are compute intensive:
  - Drug Discovery
  - Climate Modeling



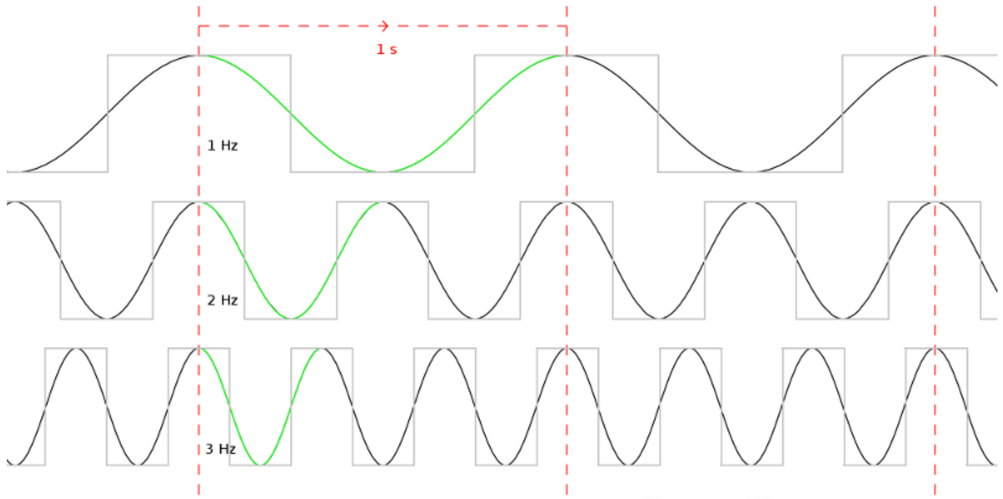
Our Never-Ending Thirst for Compute

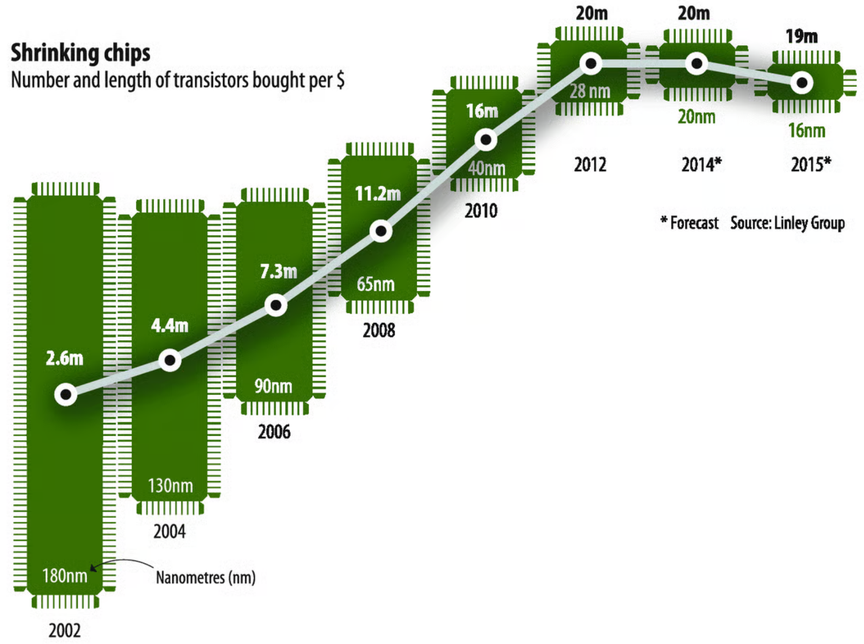
- Important applications are compute intensive:
  - Drug Discovery
  - Climate Modeling
  - Machine Learning



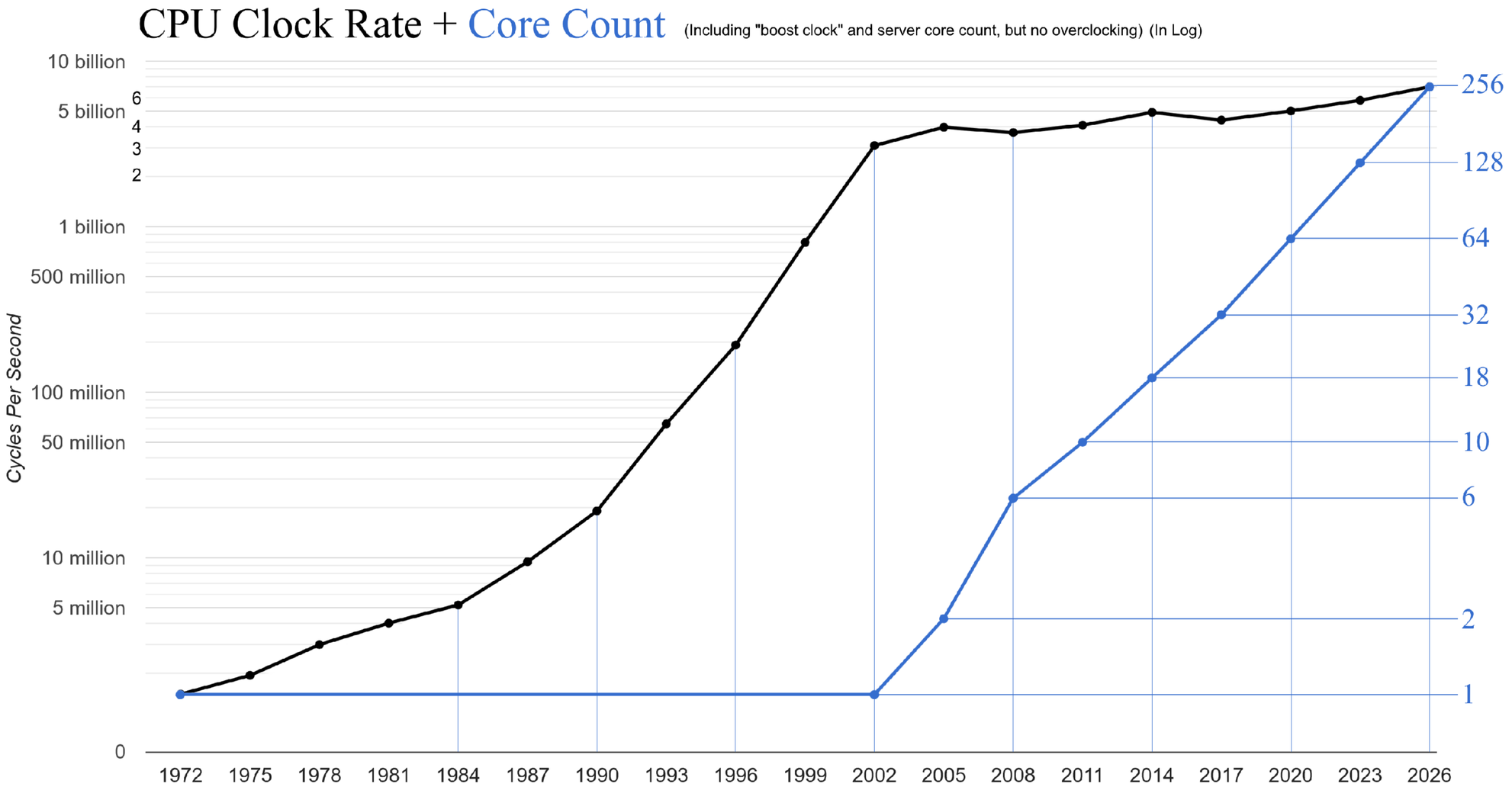
But… Has Hardware Kept Up?

- Traditional scaling via increasing CPU frequency
- We are at physical limits of transistors (post Denard scaling)
- Must find new scaling paradigm…





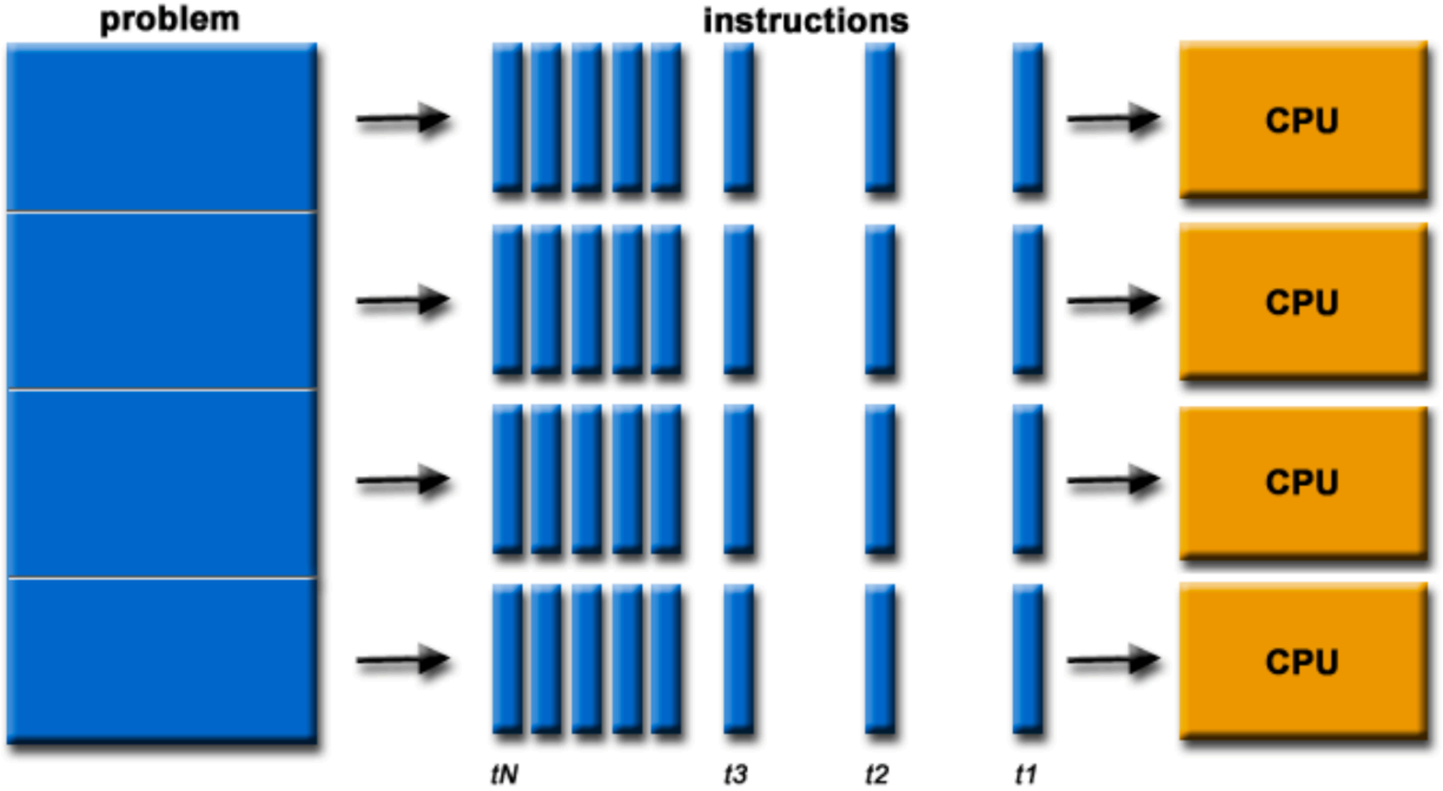
The Rise of Multicore Computers



Source: https://www.singularity.com/charts/page61.html 1976- 1998 "Microprocessor clockspeed" (modified) | 1999-2026 Manufacturer specifications
Method: Fastest / heighest (not coupled) core count CPU released (1999-2024) 2026 - Leaks + conservative prediction

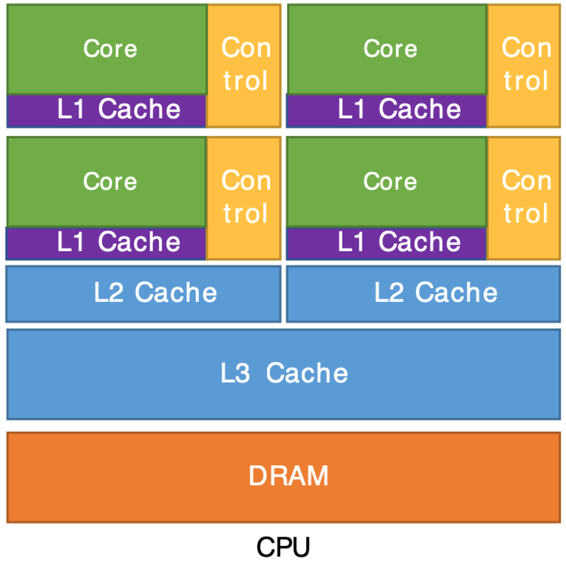
Parallel Computing

- “The simultaneous use of multiple computing resources to solve a computational problem”

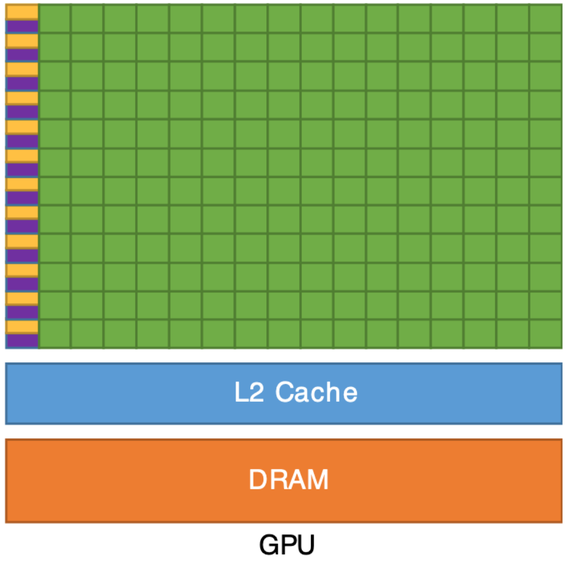


Parallelism in Everyday Life

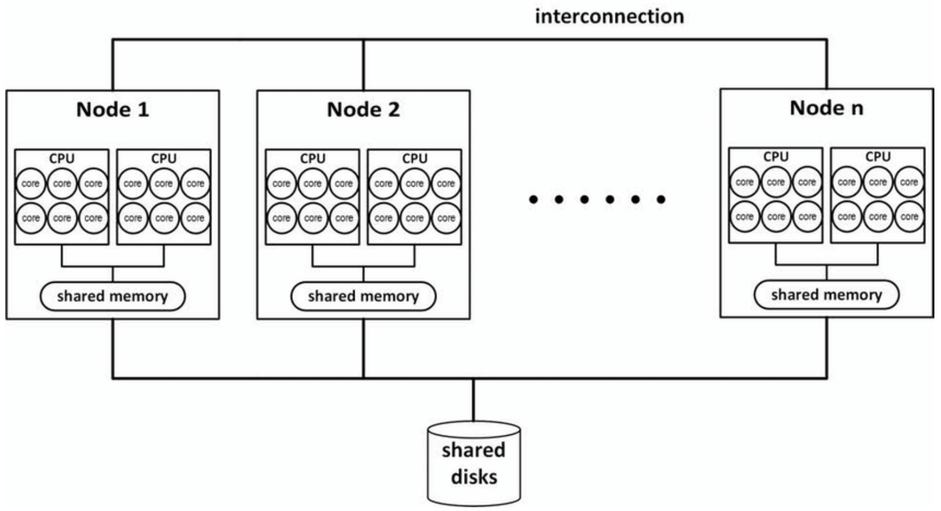
Parallel Programming is Everywhere



CPU



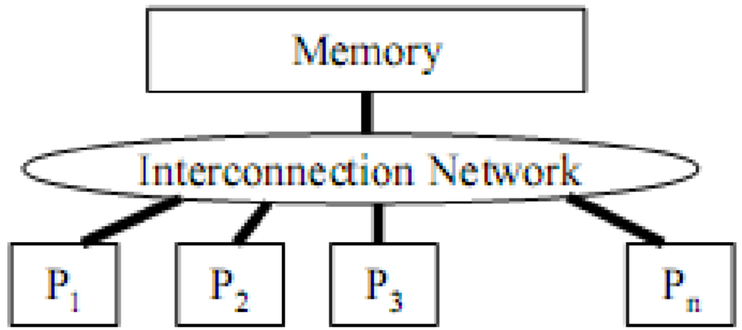
GPU

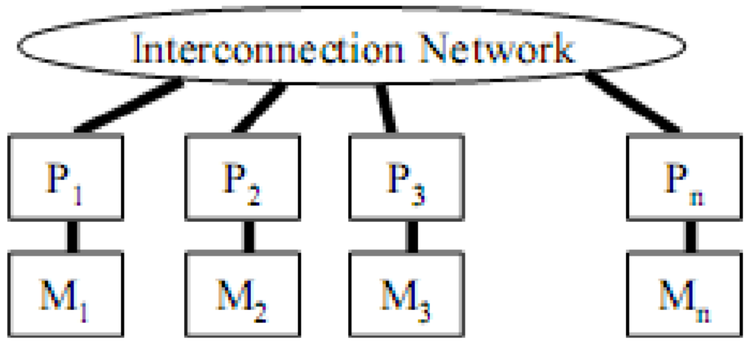


More Powerful but More Difficult

Parallel Computing Architectures

- Type 1: Shared Memory
    - Processors access the same memory space
    - OpenMP
- Type 2: Distributed memory
    - Each processor has its own private memory
    - MPI
    - **Focus of this course**





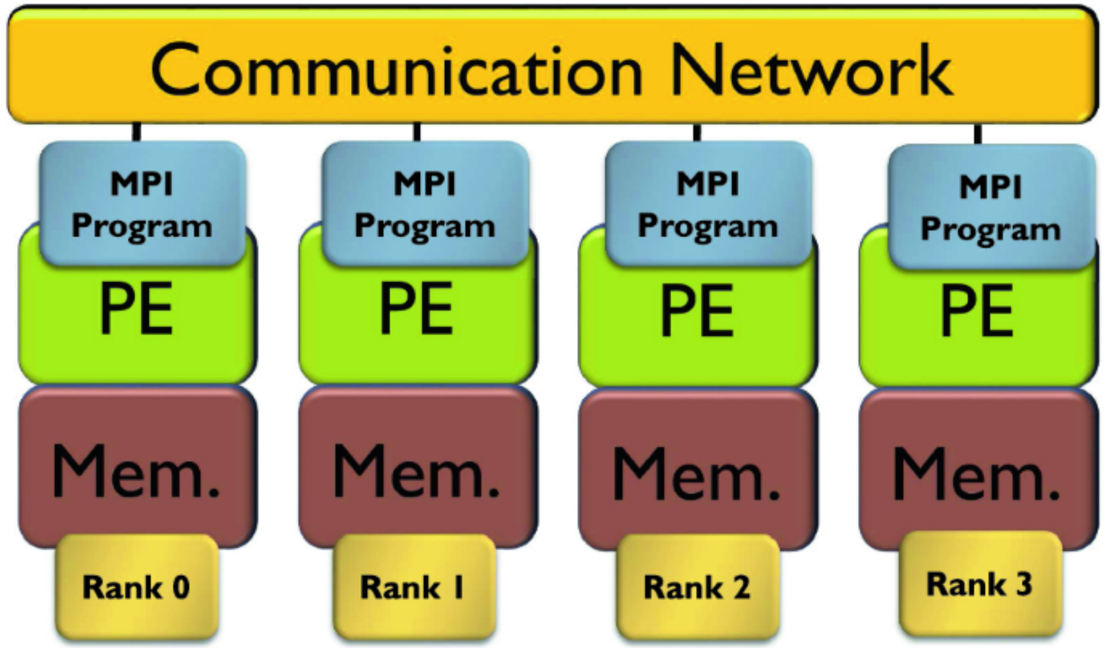
Part II: MPI Basics

Message Passing Interface 101

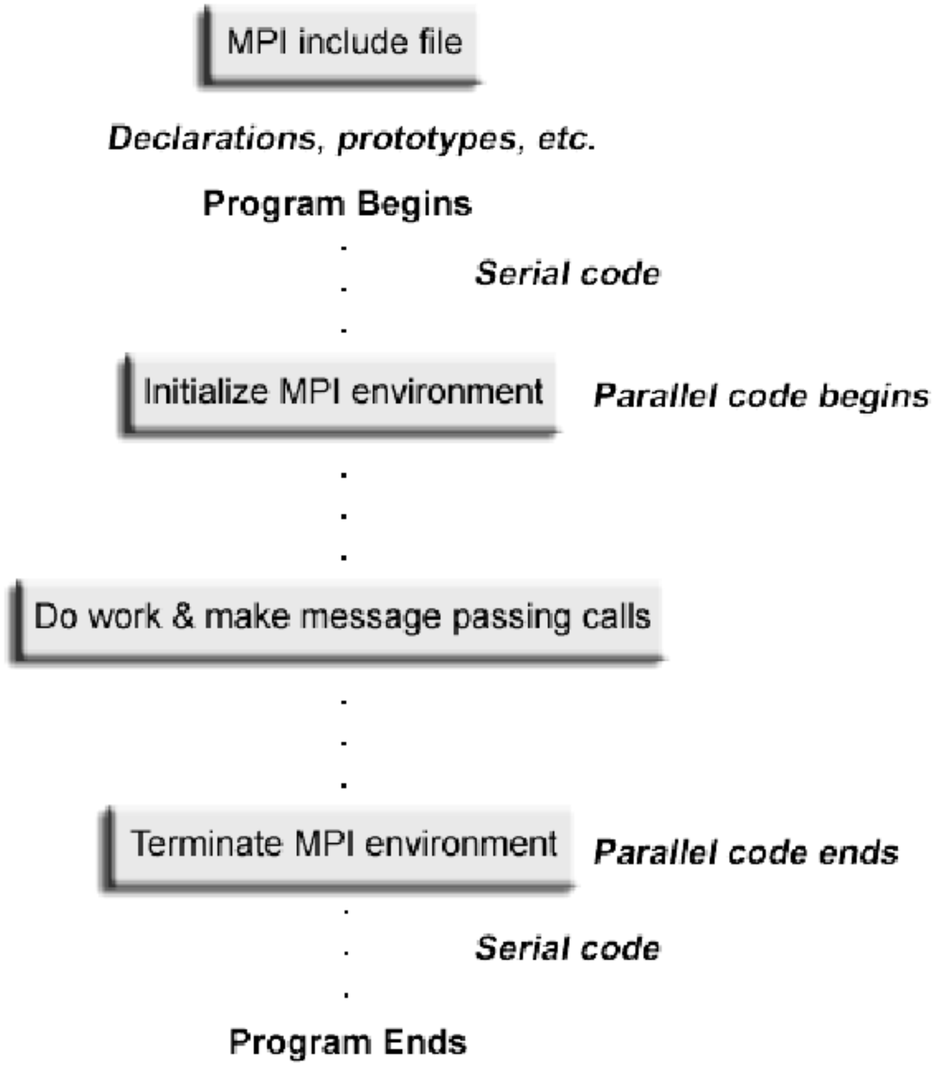
- MPI is a standard specification for the development of message passing libraries
- Before MPI, each vendor had unique communication protocols
- OpenMP is one of the most used MPI implementations
- MPI has official bindings for C/C++ & Fortran
- MPI goals:
    - Practical — **Portable** — Efficient — Flexible
- MPI consists of ~130 calls
- MPI runs on distributed, shared, and hybrid memory platforms

MPI Terminology

- **Process**: An instance of a running program on a Processing Element (PE)
- **Communicator**: A group of processes that can send and receive messages.
- **Rank**: A unique ID number assigned to each process within a communicator.
- **Message**: A chunk of data sent between two or more processes



MPI Program Structure



C Hello World (Serial)

```
#include <stdio.h>
int main(){
   printf ("Welcome to COMP428 Tutorial!\n");
   return 0;
}
```

C Hello World (Parallel)

```
#include <stdio.h>
#include”mpi.h”
int main(){
   MPI_Init (NULL, NULL); /*start MPI*/
   printf ("Welcome to COMP428 Tutorial!\n");
   MPI_Finalize(); /*shut down MPI*/
return 0;
}
```

Now… lets write code that actually does something

Essential MPI Functions

- Many parallel programs can be written using just these six functions
    - MPI_Init
    - MPI_Comm_size
    - MPI_Comm_rank
    - MPI_Send
    - MPI_Recv
    - MPI_Finalize

Essential MPI Functions

- Many parallel programs can be written using just these six functions
  - **MPI_Init**: initialize MPI runtime environment
  - **MPI_Comm_size**: returns the number of MPI processes
  - **MPI_Comm_rank**: returns unique id for each processor
  - **MPI_Send**: Sending data from one node
  - **MPI_Recv**: Receiving data from one node
  - **MPI_Finalize**: shuts down MPI runtime environment (critical!)

MPI Hello World

```
int MPI_Init(int *argc, char ***argv)
int MPI_Finalize()
```

```
int MPI_Comm_size(MPI_Comm comm, int *size)
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

```
1   #include <mpi.h>
2
3   main(int argc, char *argv[])
4   {
5     int npes, myrank;
6
7     MPI_Init(&argc, &argv);
8     MPI_Comm_size(MPI_COMM_WORLD, &npes);
9     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
10    printf("From process %d out of %d, Hello World!\n",
11           myrank, npes);
12    MPI_Finalize();
13  }
```

Part III: Sending & Receiving

The Building Blocks of MPI

Send and Receive Operations

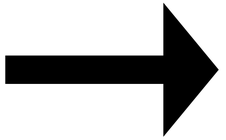
- Last week's example was a "hello world"
- We will look at how nodes communicate with each other
- In MPI, interactions are accomplished by sending and receiving messages
- MPI provides basic functions to send & receive
  - **`send`**`(void *sendbuf, int nelems, int dest)`
  - **`receive`**`(void *recvbuf, int nelems, int source)`
- It is vital to understand how these functions work under the hood

Simple Scenario

Send and Receive Operations

- Consider two processes
- P0 sends a to P1
- What will be printed?

```
P0

a = 100;
send(&a, 1, 1);
a=0;
```



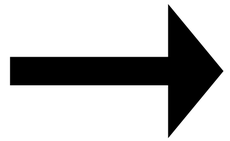
```
P1

receive(&a, 1, 0)
printf("%d\n", a);
```

Simple Scenario

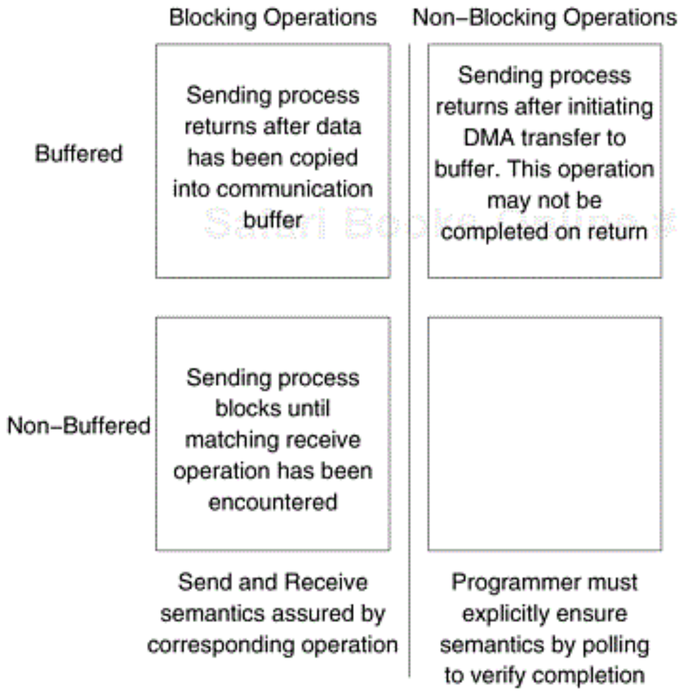
Send and Receive Operations

- Consider two processes
- P0 sends a to P1
- What will be printed?
- Depends on:
  - Use of buffer or not
  - Blocked or passing function
  - 4 possible send & recv implementations

```
P0

a = 100;
send(&a, 1, 1);
a=0;
```

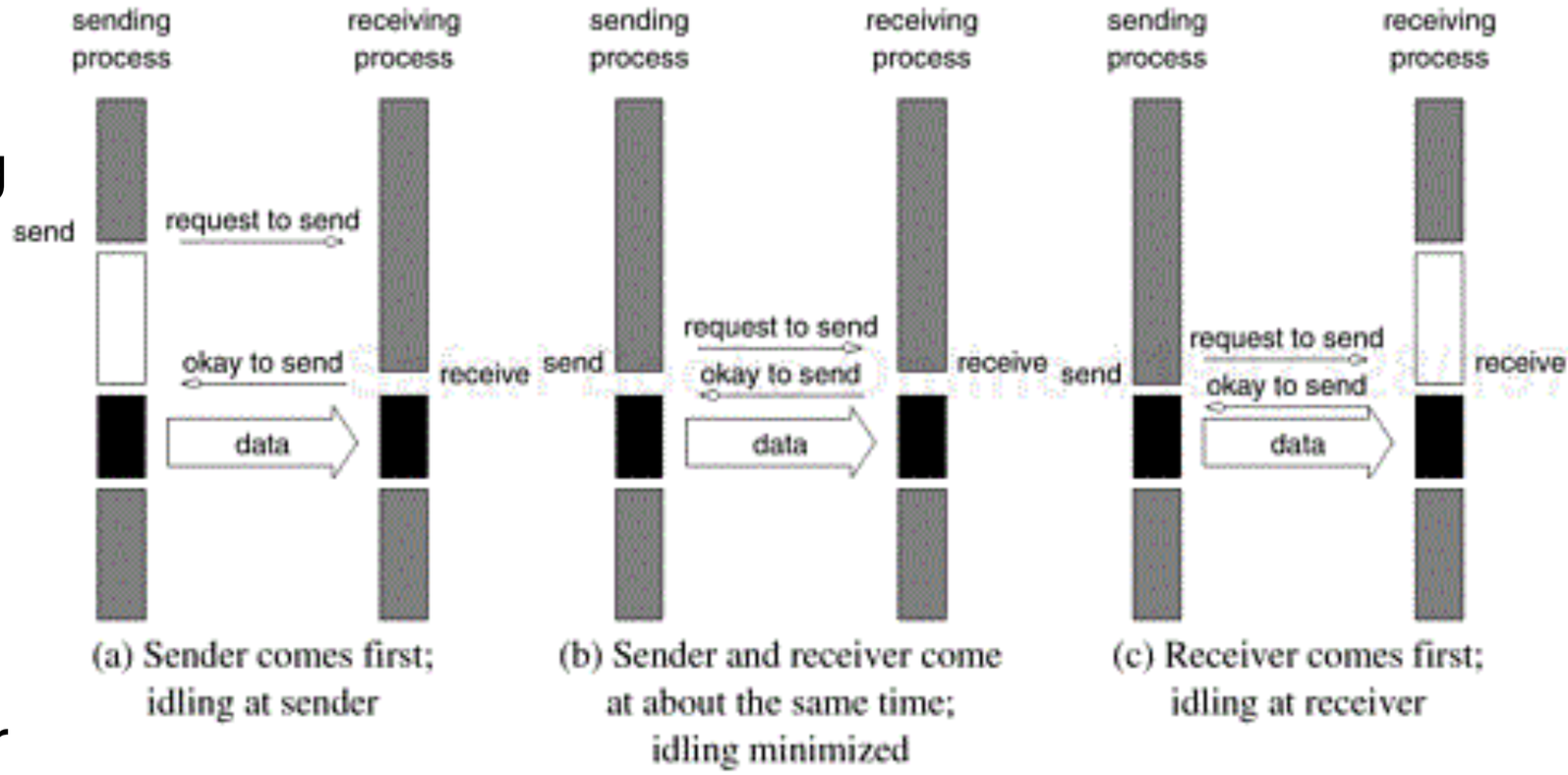


```
P1

receive(&a, 1, 0)
printf("%d\n", a);
```



Blocking Non-Buffered Send/Receive

- Send operation does not return until the matching receive has been encountered at the receiving process
- Afterwards message is sent and the send operation returns upon completion of the communication operation
- Idling may happen at sender or receiver



Blocking Non-Buffered Send/Receive

Deadlock

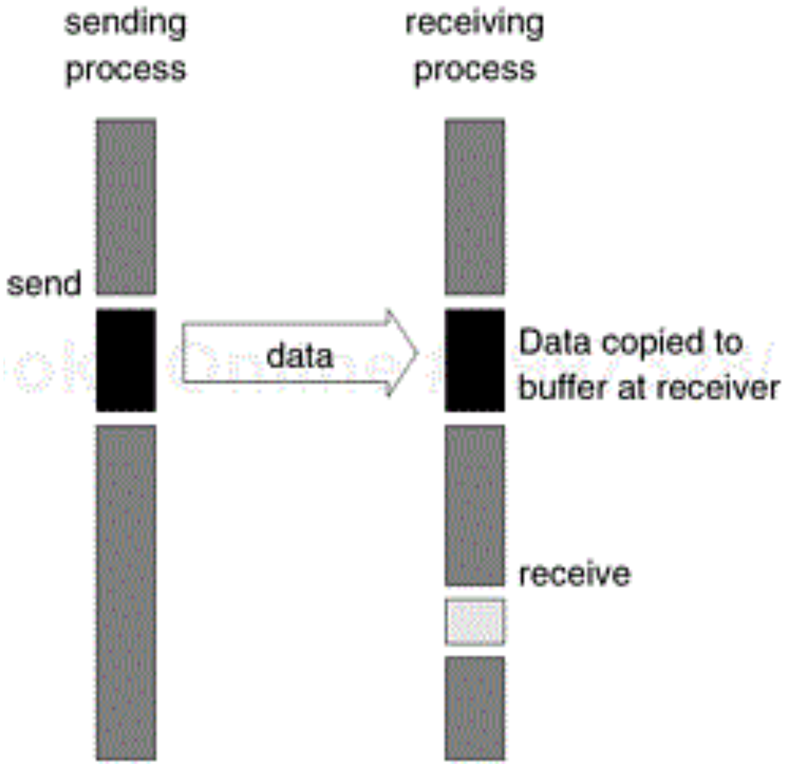
- **Deadlock**: *any situation in which computation can not proceed because each members waits for another*
- This type of send/recv can lead to deadlock
- What will happen here?

```
P0

send(&a, 1, 1);
receive(&b, 1, 1);
```

```
P1

send(&a, 1, 0);
receive(&b, 1, 0);
```

Blocking Buffered Send/Receive

- A simple solution to the idling and deadlocking problem is to used buffers
- The actual communication can be accomplished in many ways depending on the available hardware resources
- If the hardware supports asynchronous communication (independent of the CPU), then a network transfer can be initiated after the message has been copied into the buffer.
- Tradeoff: alleviate idling overheads at the cost of adding buffer management overheads



Blocking buffered transfer protocols: in the absence of communication hardware, sender interrupts receiver and deposits data in buffer at receiver end.

Blocking Buffered Send/Receive

Deadlock

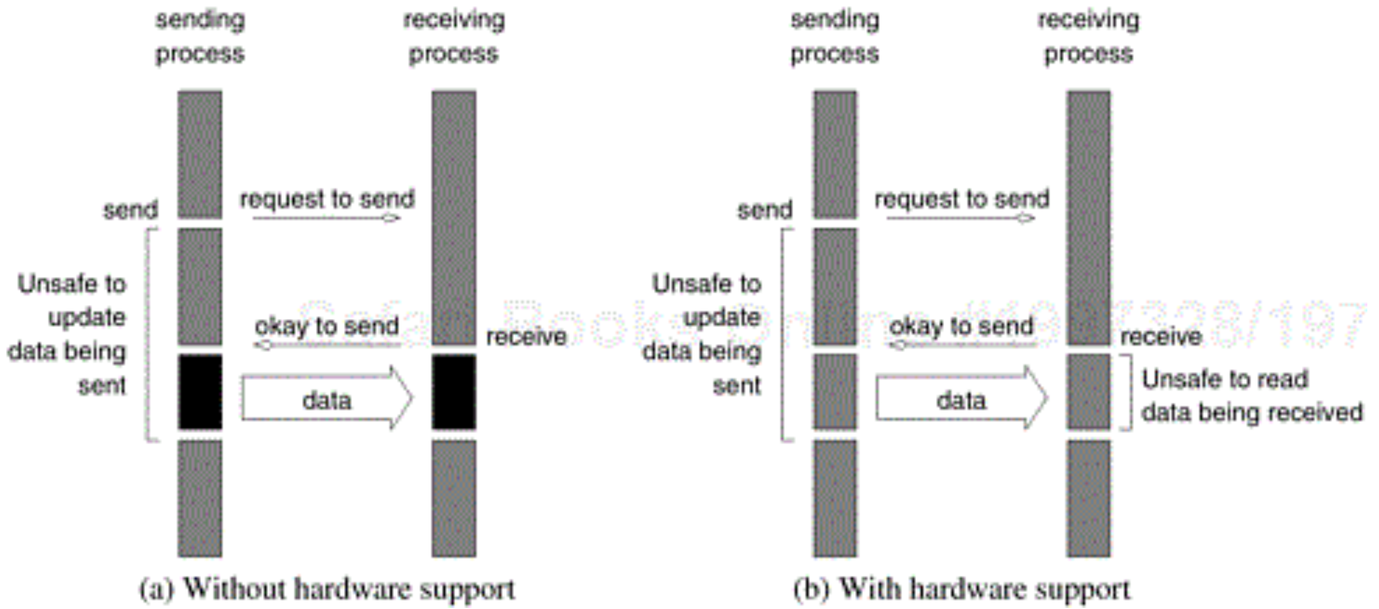
- A simple solution to the idling and deadlocking problem is to used buffers
- The actual communication can be accomplished in many ways depending on the available hardware resources
- If the hardware supports asynchronous communication (independent of the CPU), then a network transfer can be initiated after the message has been copied into the buffer.
- Tradeoff: alleviate idling overheads at the cost of adding buffer management overheads

```
P0

receive(&a, 1, 1);
send(&b, 1, 1);
```

```
P1

receive(&a, 1, 0);
send(&b, 1, 0);
```

Non-Blocking Message Passing Operations

- In blocking protocols, the overhead of guaranteeing semantic correctness was paid in the form of idling (non-buffered) or buffer management (buffered)
- In non-blocking operations, the programmer is the one who pays the price
- Have to ensure semantic correction
- Dont worry, MPI makes this easier for us



Non-blocking non-buffered send and receive operations (a) in absence of communication hardware; (b) in presence of communication hardware.

Standard Message Passing in MPI

- Send & Recv functions
- Blocking & buffered or non-buffered
- MPI provides datatypes for every C datatype + 2 extra:
  - MPI_BYTE: corresponds to a byte
  - MPI_PACKED: a collection of non-contiguous data

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

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Blocked Passing in MPI

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

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    if (world_rank == 0) {
        int number = 123;
        MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    } else if (world_rank == 1) {
        int number;
        MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process 1 received %d\n", number);
    }

    MPI_Finalize();
    return 0;
}
```

```c
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
        int dest, int tag, MPI_Comm comm)
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
        int source, int tag, MPI_Comm comm, MPI_Status *status)
```

Blocked Send & Recv Deadlock

```
1   int a[10], b[10], myrank;
2   MPI_Status status;
3   ...
4   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
5   if (myrank == 0) {
6     MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
7     MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
8   }
9   else if (myrank == 1) {
10    MPI_Recv(b, 10, MPI_INT, 0, 2, MPI_COMM_WORLD);
11    MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD);
12  }
13  ...
```

Non-Blocking Communication in MPI

- MPI provides MPI_Isend & MPI_Irecv for non-blocking operations
- These functions return immediately after being called
- It is your duty to make sure you don’t break your program
- To do that, use
  - MPI_Test(): tests whether or not the Isend or Ireceive has finished
  - MPI_Wait(): blocks until the non-blocking operation completes

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype,
        int dest, int tag, MPI_Comm comm, MPI_Request *request)
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
        int source, int tag, MPI_Comm comm, MPI_Request *request)
```

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

Non-Blocking Communication in MPI

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

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    MPI_Request request;
    MPI_Status status;

    if (world_rank == 0) {
        int number = 123;
        MPI_Isend(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &request);
        // <-- do more computations here while send is in progress
        MPI_Wait(&request, &status);
    } else if (world_rank == 1) {
        int number;
        MPI_Irecv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &request);
        // <-- do more computations here while receive is in progress
        MPI_Wait(&request, &status);
        printf("Process 1 received %d\n", number);
    }

    MPI_Finalize();
    return 0;
```

Other Useful Send&Recv Operations

- MPI provides many “quality of life” functions for common use cases:
  - Send&receive at the same time: MPI_Sendrecv()
  - Send&receive using one buffer: MPI_Sendrecv_replace()
  - Listen for messages from any source: MPI_ANY_SOURCE
  - Listen for messages from any tag: MPI_ANY_TAG
  - And much more…

Let’s Learn by Coding

- Open the Moodle page for this course
- Download “lab_2_exercise.c”
- The code sends a simple message from node 0 to node 1
- Your task:
    - Change code so it sends the message from node 0 to every other node (broadcast)
    - (Try to do it with a for loop)

Part IV: Communication Patterns

Communication Patterns

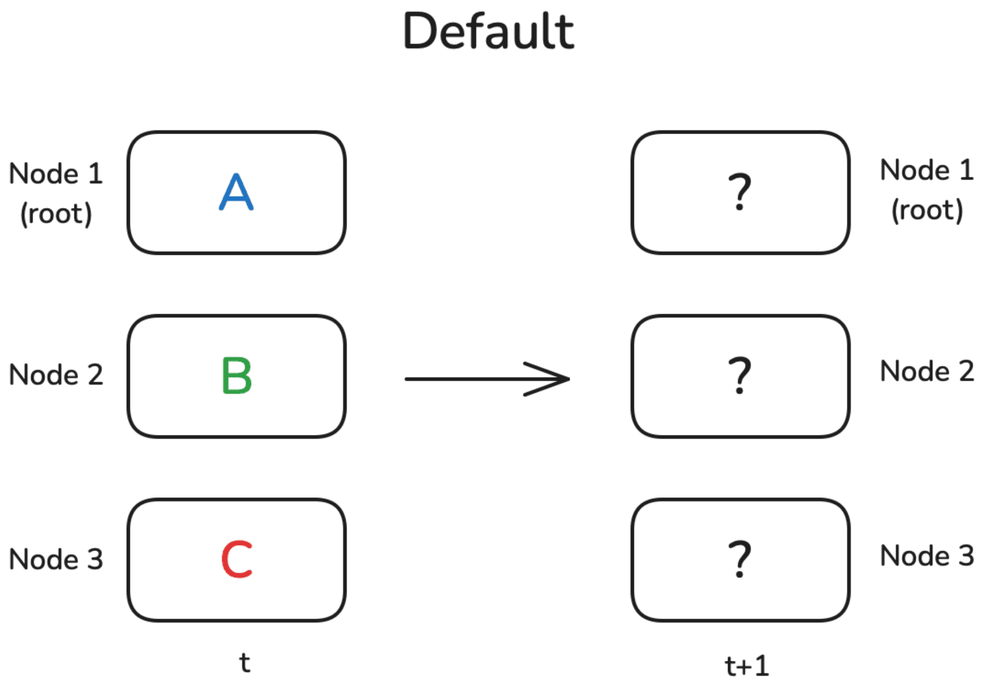
The bread and butter of MPI

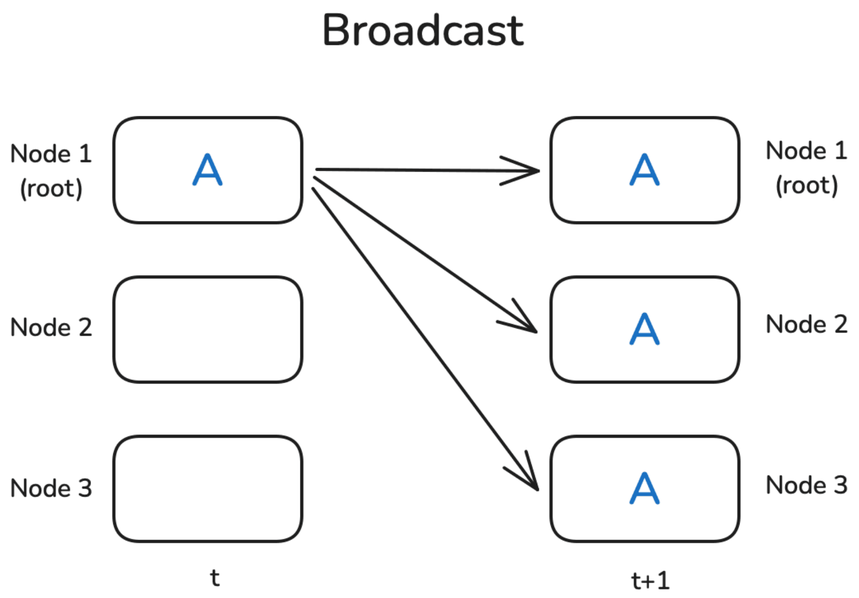
- Last week we learned how to send messages between two processes
- With just the Send and Recv functions, you can (in theory) implement any parallel algorithm
- However, decades of parallel programming have shown that most algorithms share similar patterns of sending and receiving messages between processes
- These patterns are known as the *“Basic Communication Operations”*
- MPI has prepared well-optimized functions for these patterns
- Understanding these patterns is ***essential***

One-to-All Broadcast

Pattern #1

- **Goal:** send message *m* from node *n* to all other nodes
- https://huggingface.co/spaces/nanotron/ultrascale-playbook

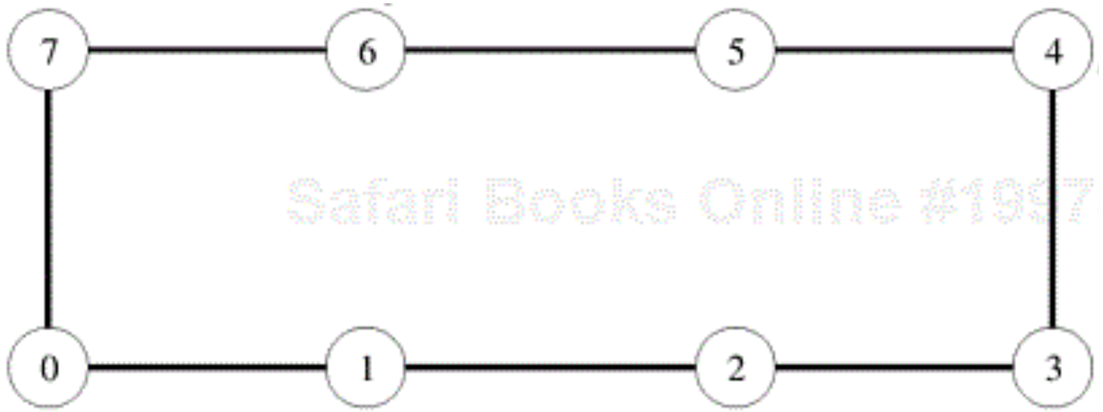




One-to-All Broadcast

Naive Implementation

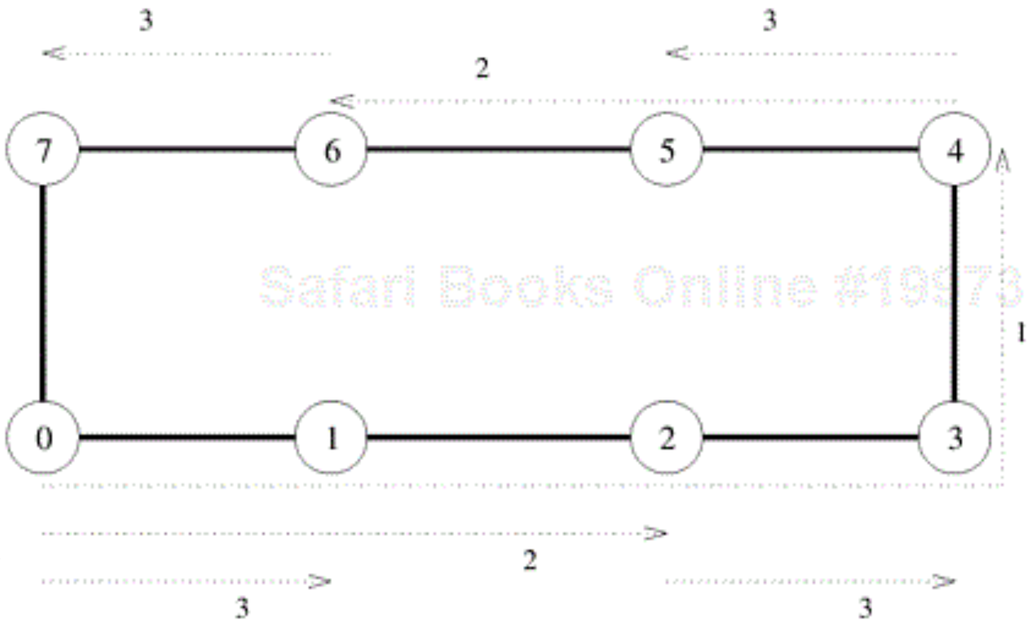
- The easiest (yet worst) way to implement one-to-all:
    - Node 0 —> Node 1
    - Node 0 —> Node 2
    - Node 0 —> Node 3
    - …
- This requires n steps of communication
- Can you think of a better way?



One-to-All Broadcast

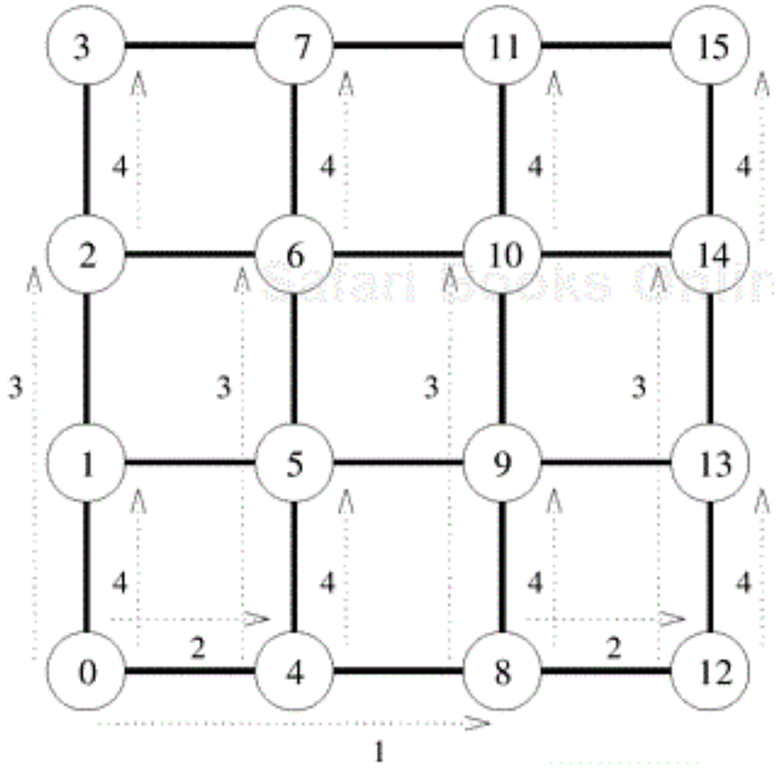
Better Implementation

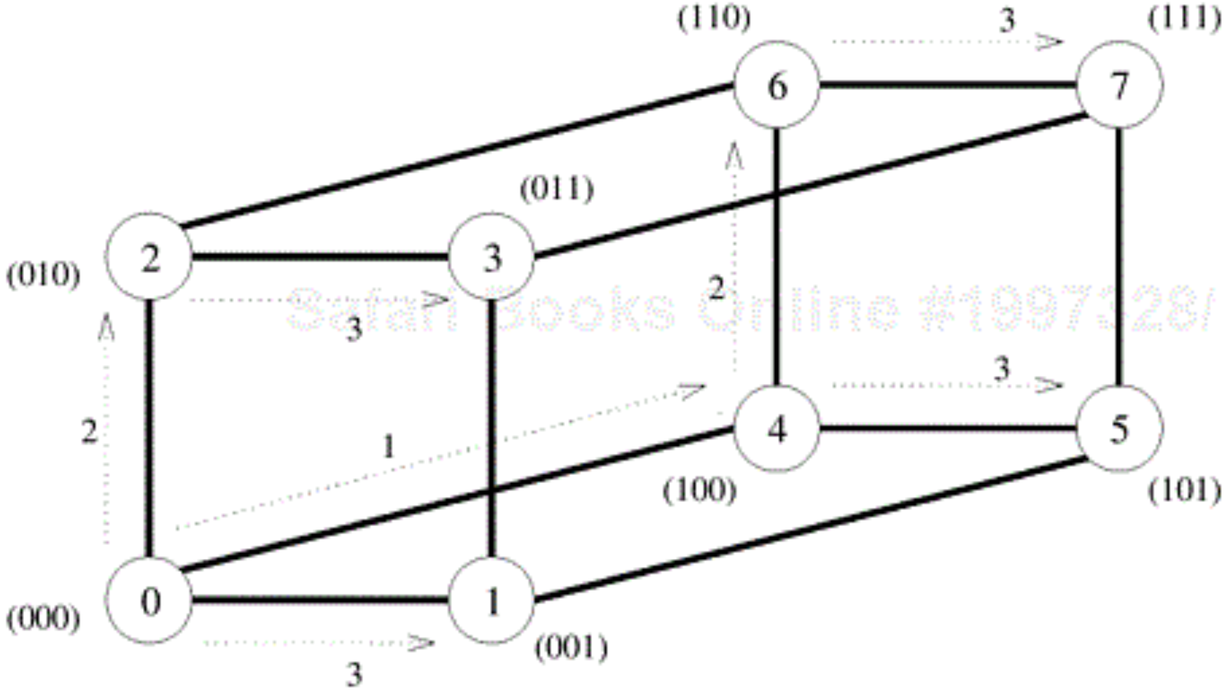
- **Key Idea —>** the nodes who have received the data can help distribute it:
  - N0->N4
  - N0->N2 | N4—>N6
  - N0->N1 | N4->N5 | N2->N3 | N6-> N7
- Requires only log(n) step!
- Sequence of nodes has to be chosen carefully
- This is to reduce congestion on a single link



One-to-All Broadcast

Other Topologies





One-to-All Broadcast

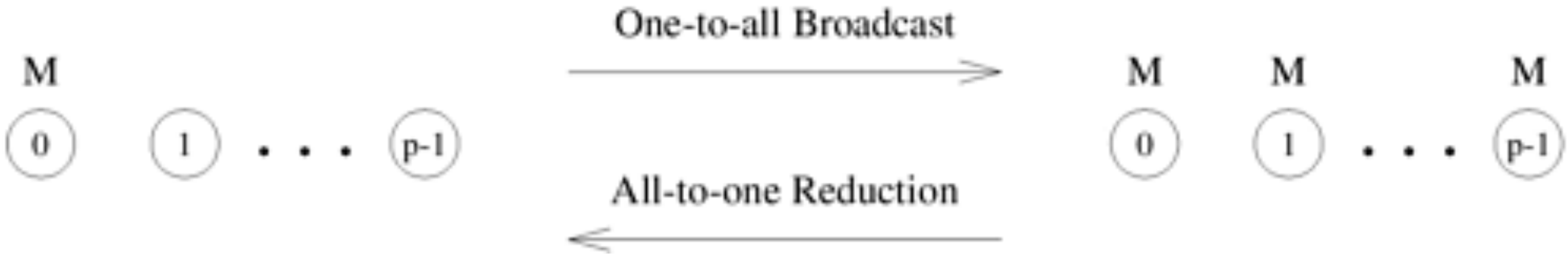
Hypercube Topology Sudocode

```
1.    procedure ONE_TO_ALL_BC(d, my_id, X)
2.    begin
3.       mask := 2^d - 1;                    /* Set all d bits of
4.       for i := d - 1 downto 0 do          /* Outer loop */
5.           mask := mask XOR 2^i;           /* Set bit i of mask
6.           if (my_id AND mask) = 0 then /* If lower i bits of
7.               if (my_id AND 2^i) = 0 then
8.                   msg_destination := my_id XOR 2^i;
9.                   send X to msg_destination;
10.              else
11.                  msg_source := my_id XOR 2^i;
12.                  receive X from msg_source;
13.              endelse;
14.          endif;
15.      endfor;
16.   end ONE_TO_ALL_BC
```

All-to-One Reduce

Pattern #2

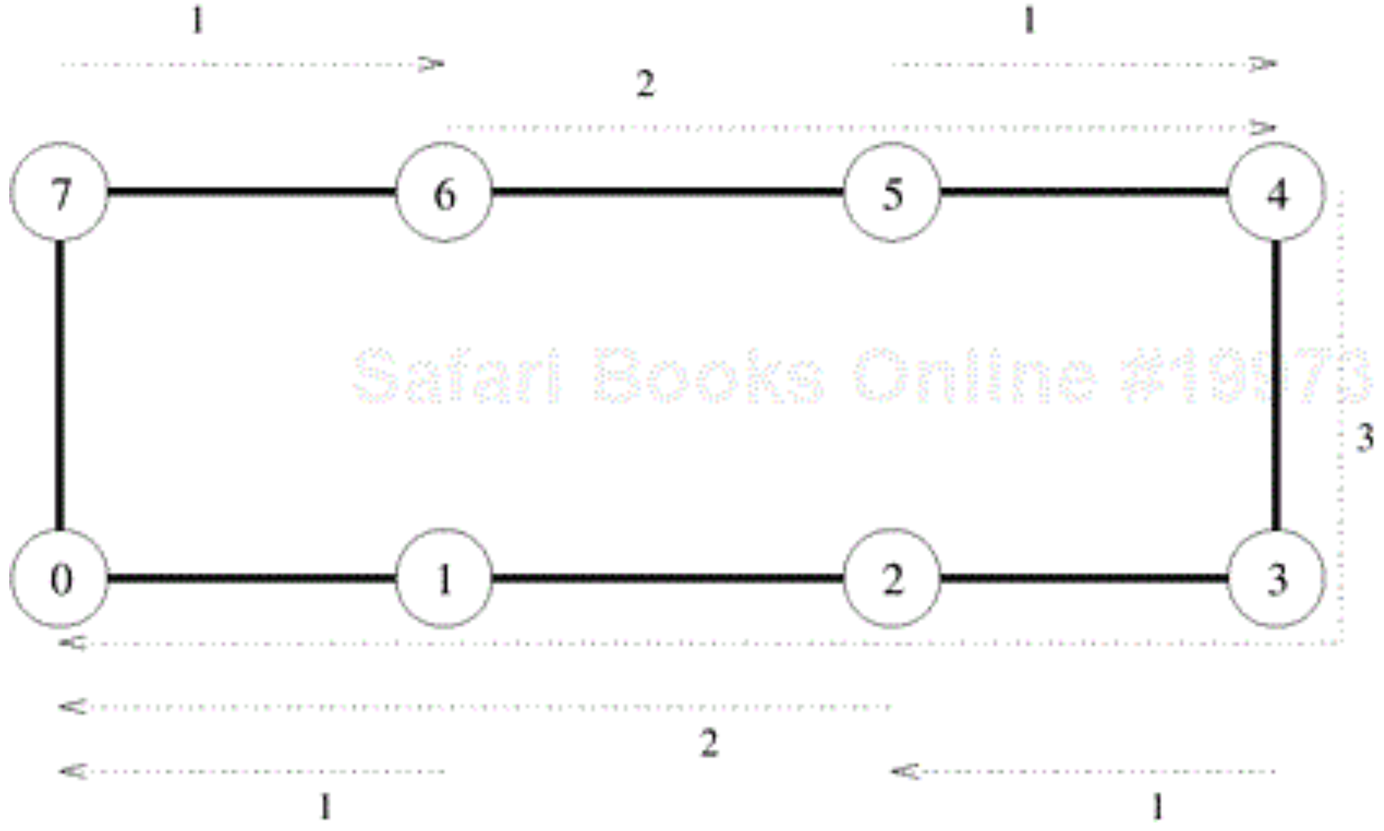
- **Goal:** send messages from every node to a single node *n*
- It is the ***dual*** of all-to-one reduce



All-to-One Reduce

Implementation

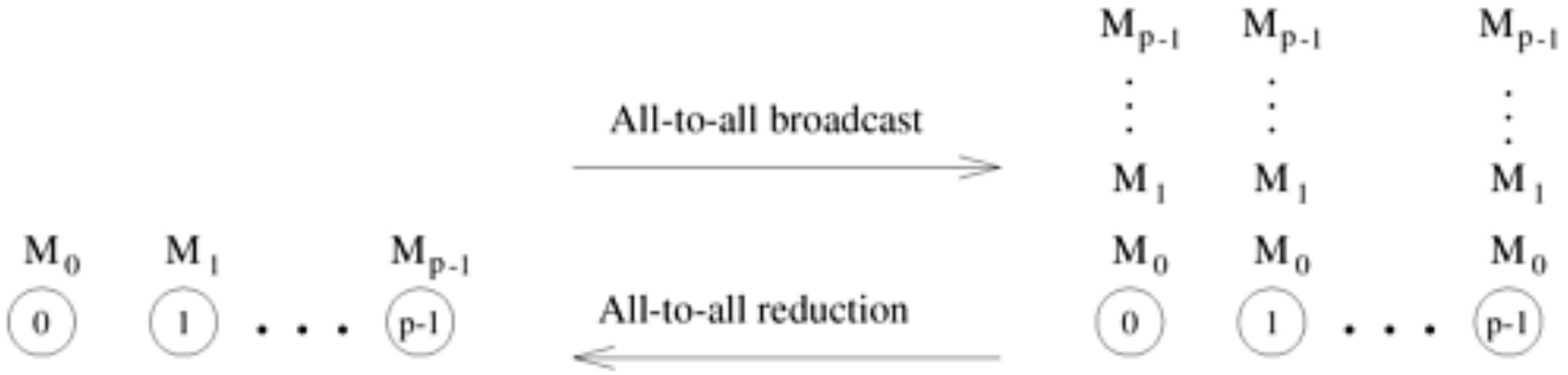
- The exact opposite steps as one-to-all broadcast



All-to-All Broadcast and Reduction

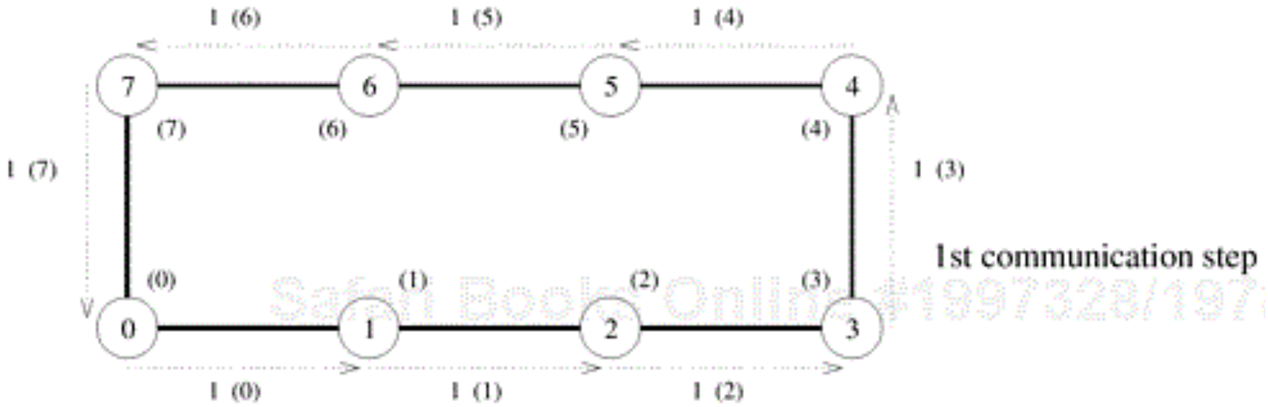
Pattern #3&4

- It is a generalization of one-to-all broadcast in which all p nodes simultaneously initiate a broadcast

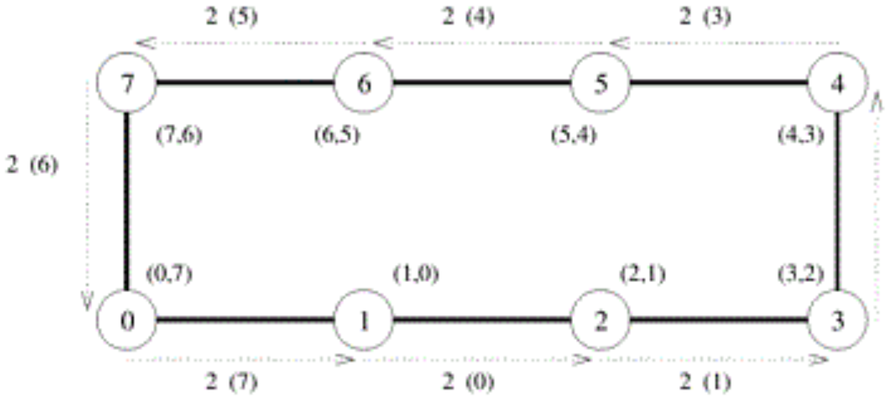


All-to-All Broadcast and Reduction

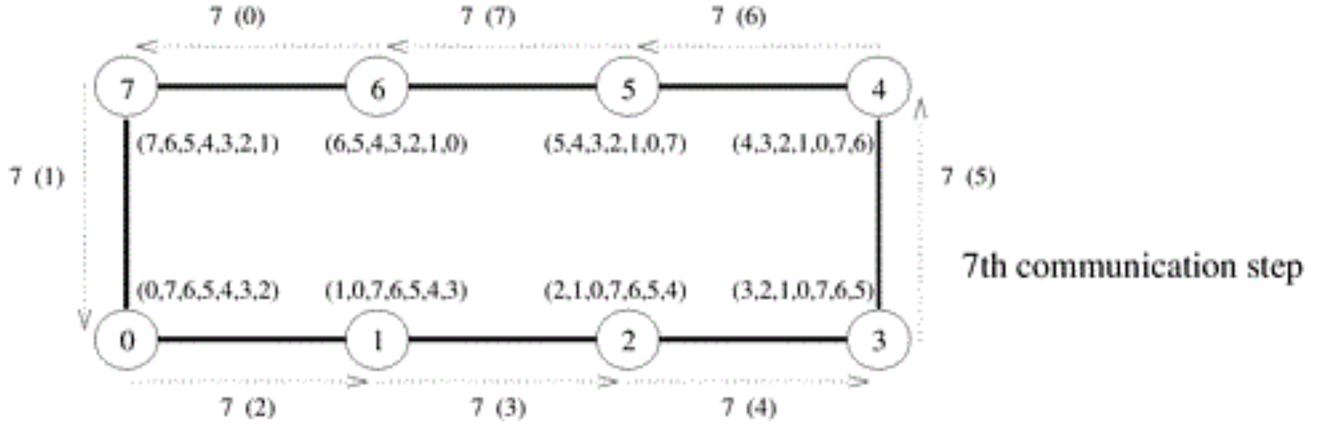
Ring Topology



1st communication step



2nd communication step

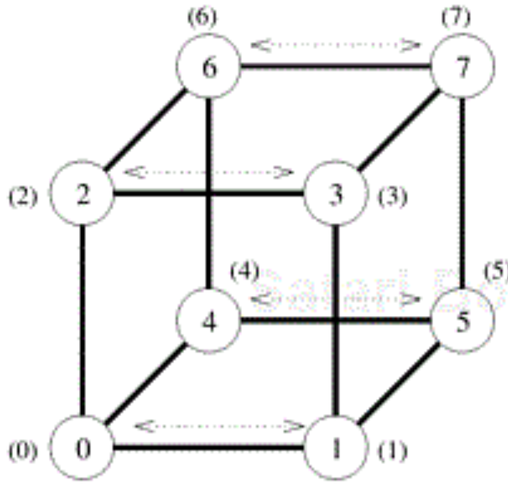


7th communication step

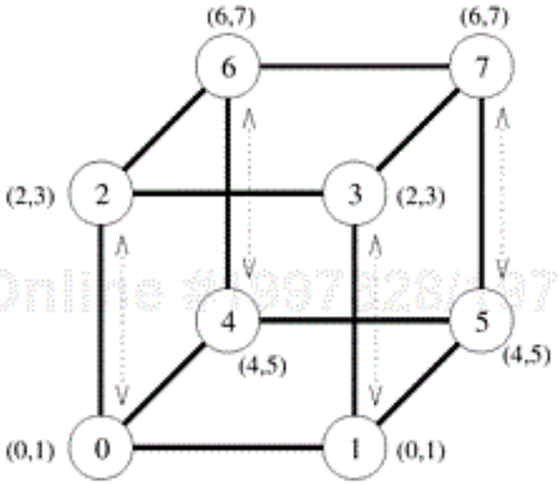
All-to-All Broadcast and Reduction

Hypercube Topology

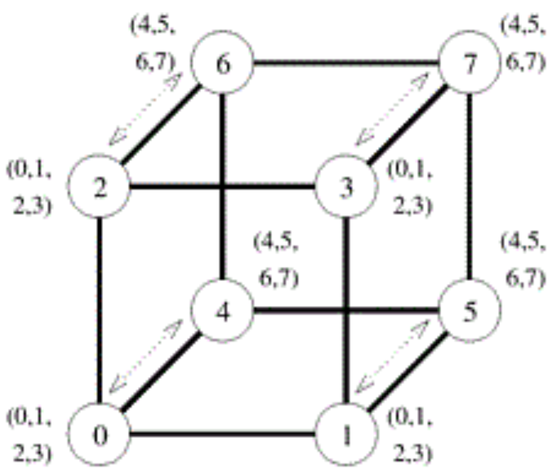
- Log(n) steps



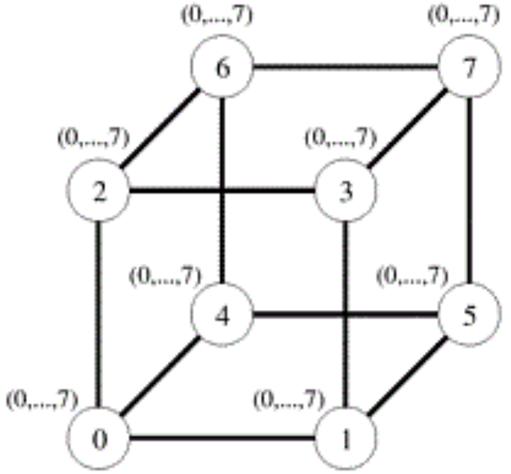
(a) Initial distribution of messages



(b) Distribution before the second step



(c) Distribution before the third step

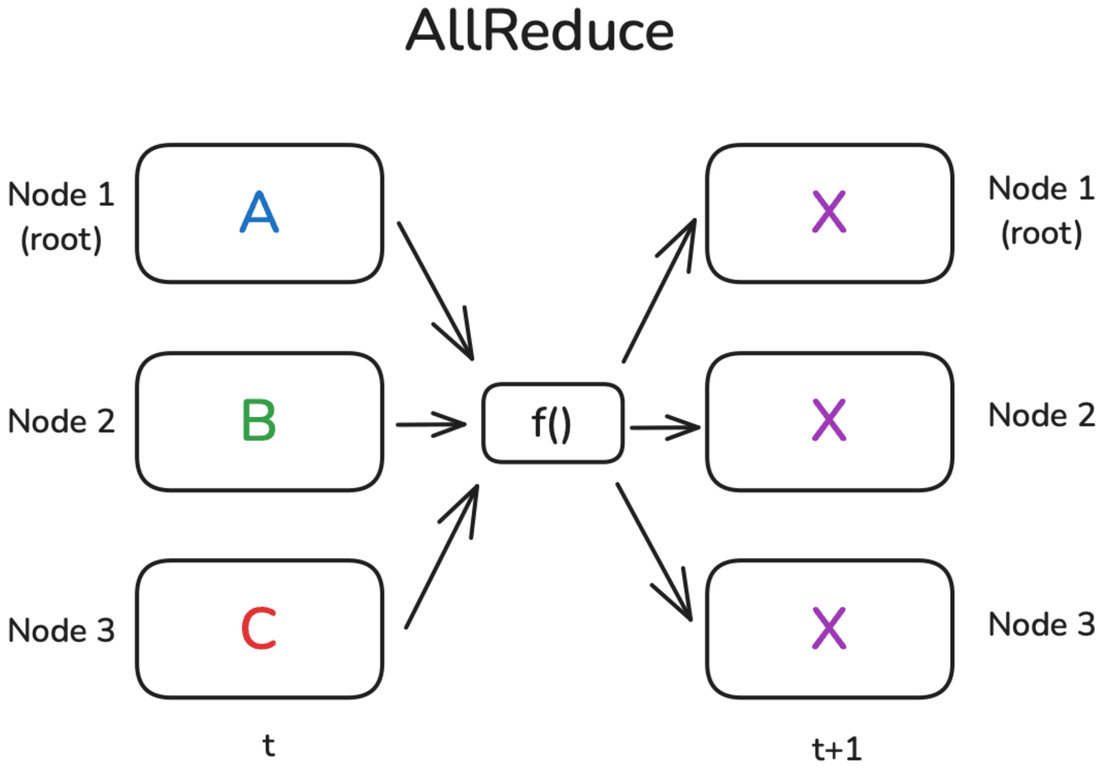


(d) Final distribution of messages

All-Reduce

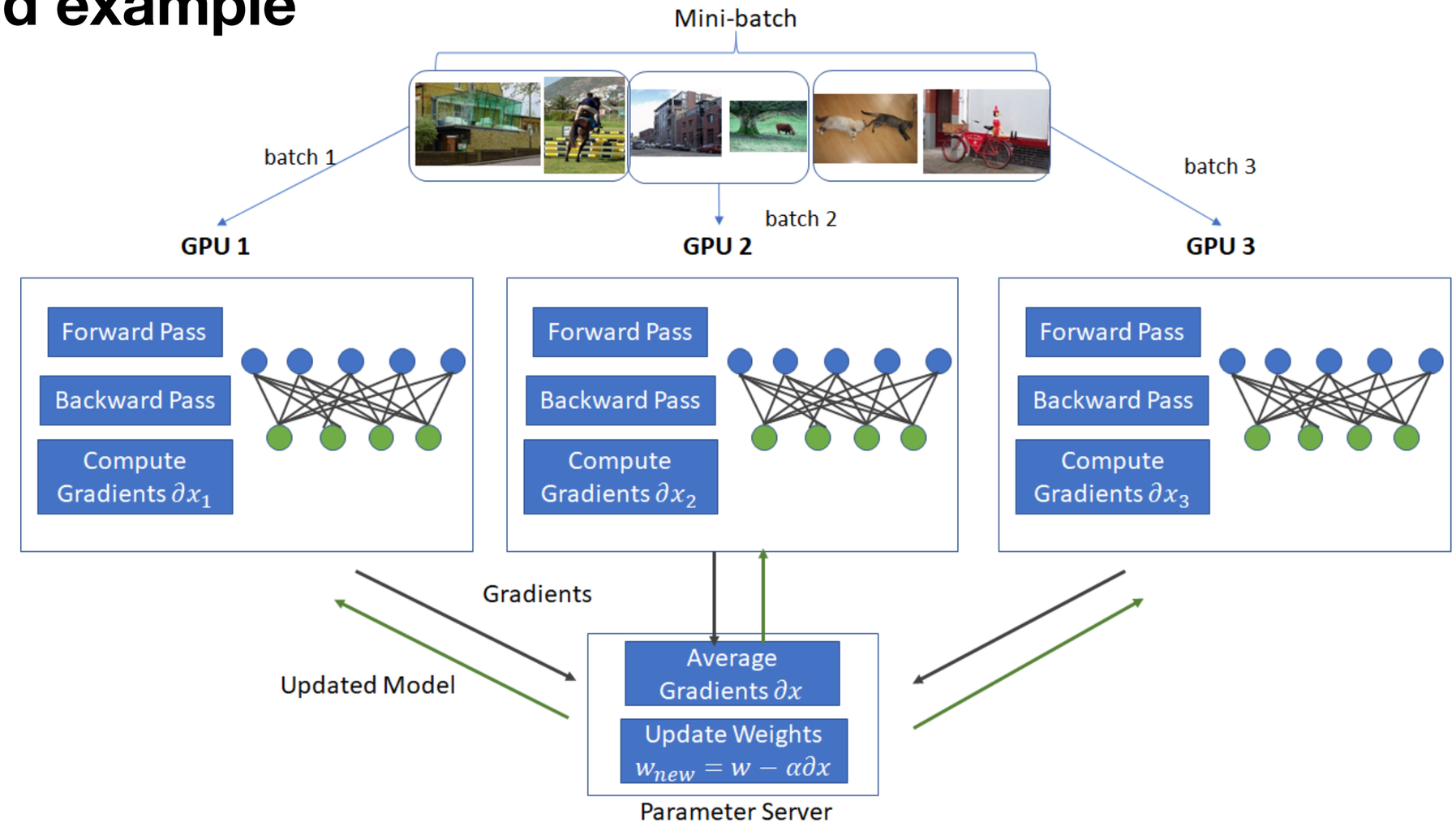
Pattern #5

- Each node begins with a buffer of size m. The final output is an identical buffer of size m on each node, created by combining the original p buffers with an associative operator.
- All-reduce is identical to performing an all-to-one reduction followed by a one-to-all broadcast of the result



All-Reduce

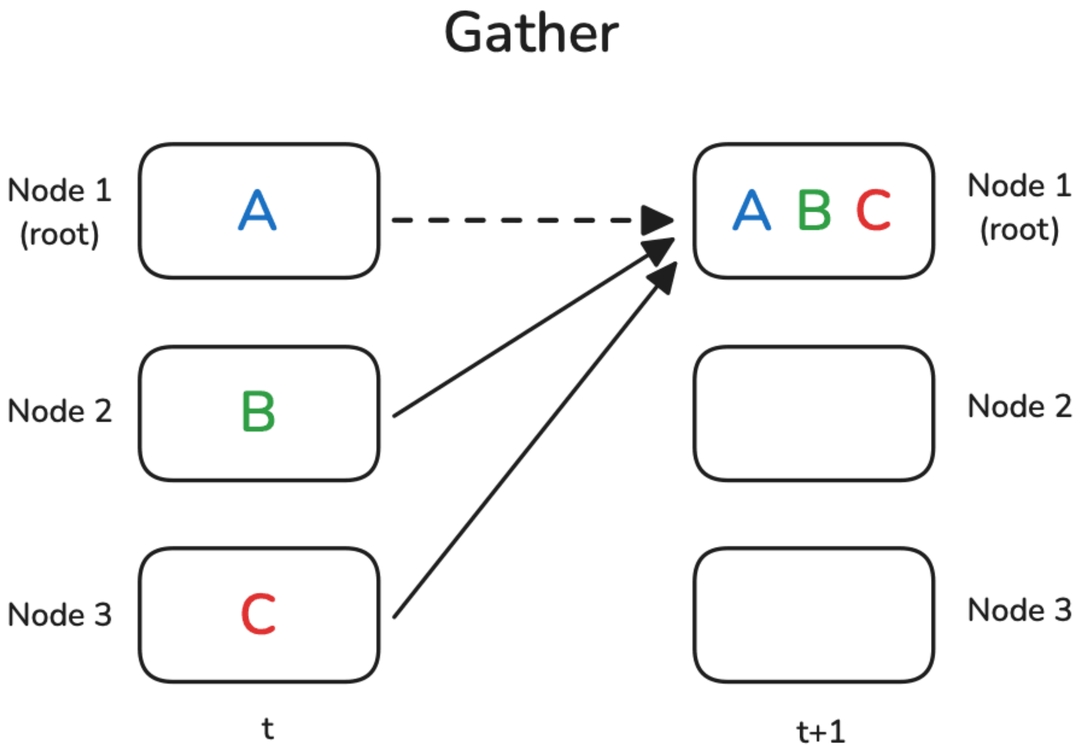
Real world example



Scatter and Gather

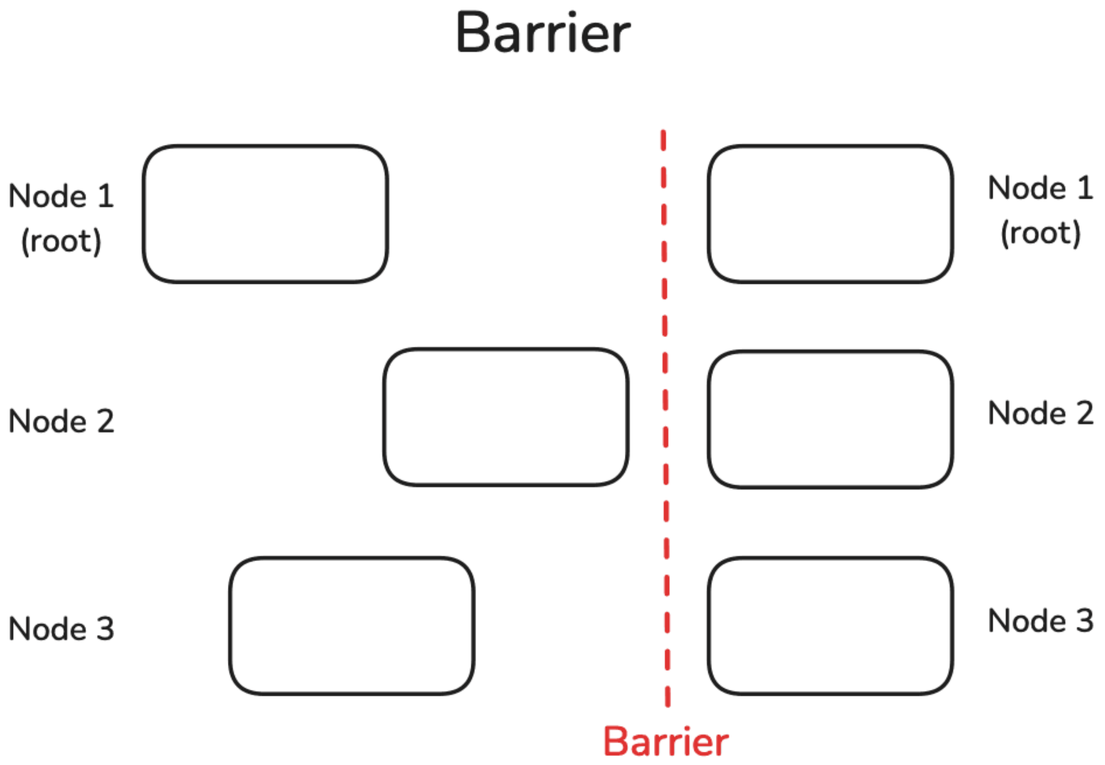
Pattern #6&7

- **Scatter**: A single "root" process divides a large data set into smaller, unique chunks and distributes them to multiple "leaf" processes.
- **Gather**: The reverse of a scatter operation, where a single "root" process collects data from multiple "leaf" processes to combine it into a single, cohesive data set.



Barrier Operation

- Sometimes, it is necessary to synchronize all nodes
- The *Barrier* operation, does this for us
- Remember: use a barrier **only** when it is absolutely **necessary** for your algorithms since it almost always causes idling



Collective Communication Operations in MPI

- It is possible to implement all communication operations using the basic send and recv functions
- However, MPI provides functions for doing this, you should use them because:
    - It makes programming easier
    - They are optimized for speed
- We will now look at these functions
- They are very important for your assignments

Barrier in MPI

```
int MPI_Barrier(MPI_Comm comm)
```

- The only argument of MPI_Barrier is the communicator that defines the group of processes that are synchronized
- The call to MPI_Barrier returns only after all the processes in the group have called this function.

Broadcast in MPI

```
int MPI_Bcast(void *buf, int count, MPI_Datatype datatype,
        int source, MPI_Comm comm)
```

- `void *buf`: The data buffer to be sent from the source process and received by all other
- `int count`: The number of elements in the buffer.
- `MPI_Datatype datatype`: The type of each element in the buffer (e.g., MPI_INT)
- `int root`: The rank of the source process that holds the data to be broadcast.
- `MPI_Comm comm`: The communicator that defines the group of processes participating in the broadcast.

Reduction in MPI

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

- `void *sendbuf`: The data buffer on each process to be combined.
- `void *recvbuf`: The buffer on the root process that will store the combined result.
- `int count`: The number of elements to combine from each process.
- `MPI_Datatype datatype`: The type of each element being combined.
- `MPI_Op op`: The operation to perform (e.g., MPI_MAX, MPI_SUM). You can define custom operations
- `int target`: The rank of the process that will receive the final result.

All-Reduce in MPI

```
int MPI_Allreduce(void *sendbuf, void *recvbuf, int count,
        MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

- Similar parameters to reduction
- But, no target since all nodes receive data

And More…

- `MPI_Scatter`
- `MPI_Scatterv`
- `MPI_Alltoall`
- …
- Read more here:
    - https://www.mpi-forum.org/docs/
    - https://www.open-mpi.org/doc/

Example

- Parallel sum

int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

```c
int main(int argc, char *argv[]) {
    int rank, size, n = 16;
    double *data = NULL;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Root initializes data
    if (rank == 0) {
        data = malloc(n * sizeof(double));
        for (int i = 0; i < n; i++) data[i] = i + 1;  // [1..16]
    }

    // Each process gets n/size numbers
    int local_n = n / size;
    double *local_data = malloc(local_n * sizeof(double));

    MPI_Scatter(data, local_n, MPI_DOUBLE,
                local_data, local_n, MPI_DOUBLE,
                0, MPI_COMM_WORLD);

    // Local sum
    double local_sum = 0.0;
    for (int i = 0; i < local_n; i++) local_sum += local_data[i];

    // Global sum using Reduce
    double global_sum;
    MPI_Reduce(&local_sum, &global_sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        printf("Global sum = %.1f\n", global_sum);  // should be 136
        free(data);
    }

    free(local_data);
    MPI_Finalize();
    return 0;
}
```

Part V: Groups & Communicators

Introduction to Groups and Communicators

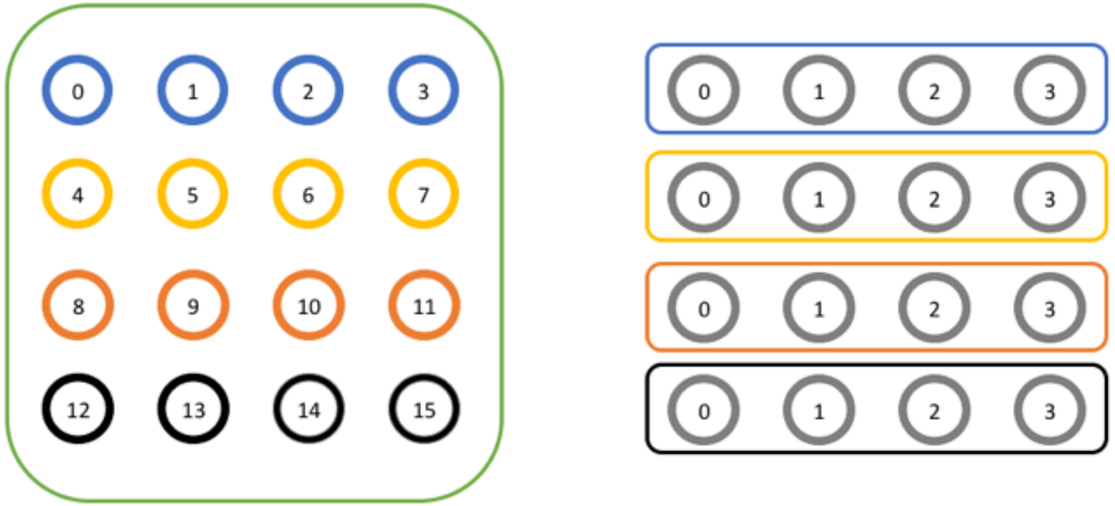
- This section is based off the following webpage:
    - https://mpitutorial.com/tutorials/introduction-to-groups-and-communicators/
- A communicator is a group of processes
- In all previous labs, we have used MPI_COMM_WORLD, which contains all nodes
- For simple applications, this is sufficient as we have a relatively small number of processes
- When applications start to get bigger, this becomes less practical and we may only want to talk to a few processes at once

What is COMM_WORLD?

- Default communicator in MPI (Message Passing Interface).
- Includes all processes in an MPI program when it starts (created automatically)
- Used to identify and manage communication among processes.
- Programmers can create sub-communicators, but MPI_COMM_WORLD always exists.
- For simple applications, it’s not unusual to do everything using MPI_COMM_WORLD

Splitting Communicators

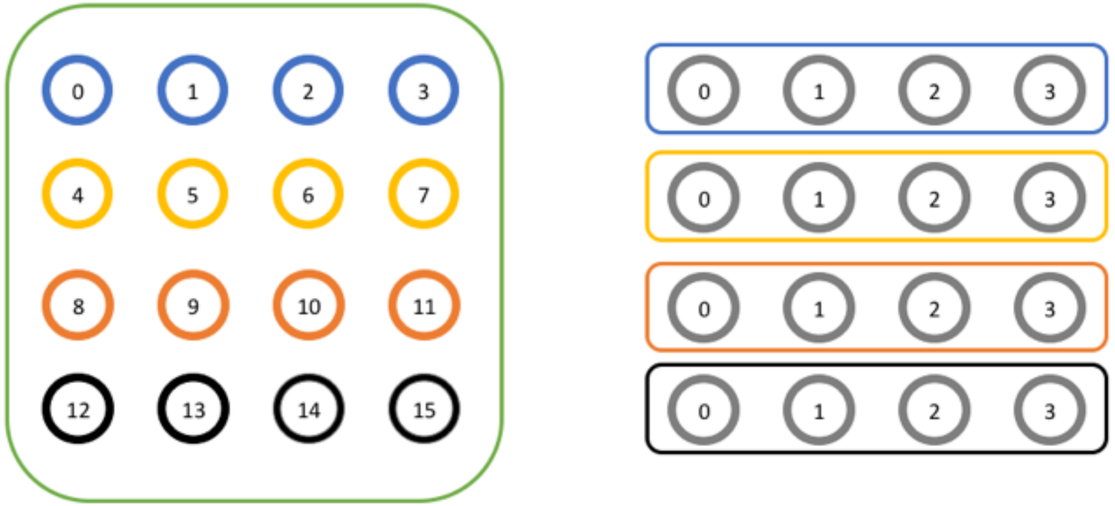
- For more complex use cases, it might be helpful to have more communicators
- The easiest way to do this is with: MPI_Comm_split




```
MPI_Comm_split(
        MPI_Comm comm,
        int color,
        int key,
        MPI_Comm* newcomm)
```

MPI_Comm_split

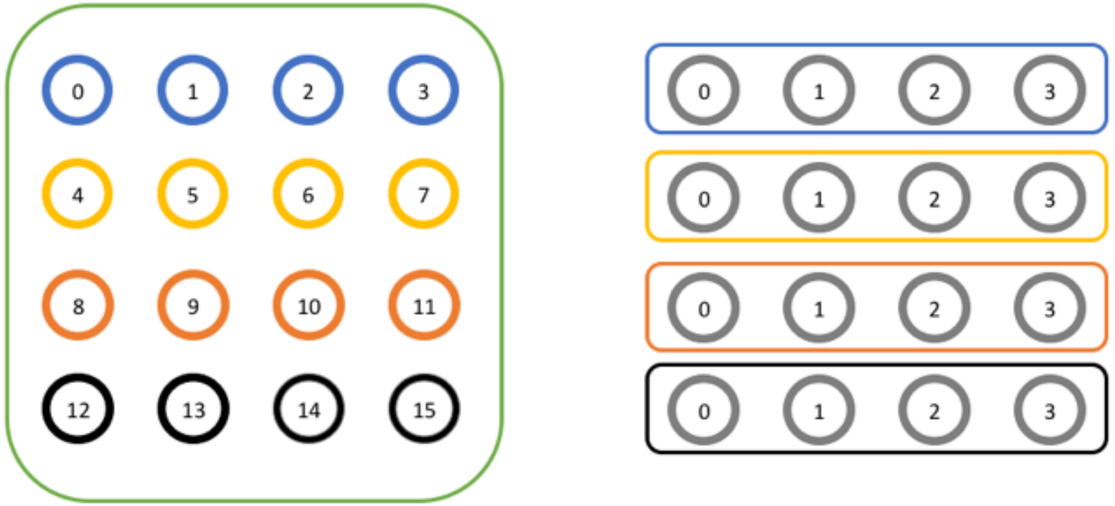
- Creates new communicators by splitting an existing communicator
- The original communicator remains unchanged
- comm is the communicator to split, e.g. MPI_COMM_WORLD
- color decides which new communicator a process belongs to
- MPI_UNDEFINED as color means the process is excluded
- key determines rank ordering within the new communicator
- Lowest key gets rank 0, next lowest gets rank 1, ties use original rank
- newcomm is the handle to the new communicator returned to the user




```
MPI_Comm_split(
        MPI_Comm comm,
        int color,
        int key,
        MPI_Comm* newcomm)
```

MPI_Comm_split | Example

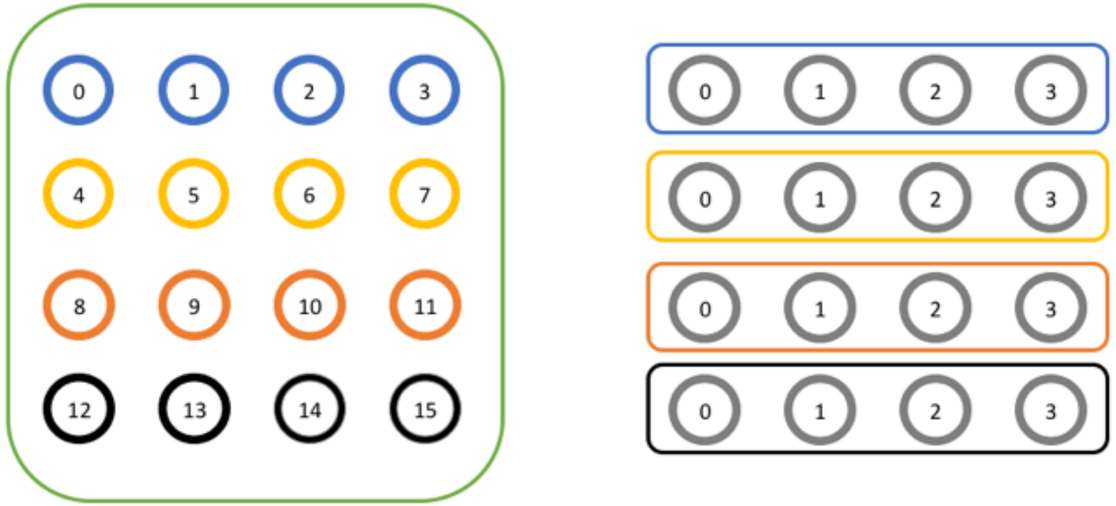
- Start with a single global communicator
- Imagine 16 processes arranged in a 4x4 grid
- Goal is to divide the grid by row
- Each row is assigned a unique color
- Processes with the same color are grouped into the same communicator
- Result is one communicator per row of the grid




```
MPI_Comm_split(
        MPI_Comm comm,
        int color,
        int key,
        MPI_Comm* newcomm)
```

MPI_Comm_split | Example

```
// Get the rank and size in the original communicator
int world_rank, world_size;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

int color = world_rank / 4; // Determine color based on row

// Split the communicator based on the color and use the
// original rank for ordering
MPI_Comm row_comm;
MPI_Comm_split(MPI_COMM_WORLD, color, world_rank, &row_comm);

int row_rank, row_size;
MPI_Comm_rank(row_comm, &row_rank);
MPI_Comm_size(row_comm, &row_size);

printf("WORLD RANK/SIZE: %d/%d \t ROW RANK/SIZE: %d/%d\n",
       world_rank, world_size, row_rank, row_size);

MPI_Comm_free(&row_comm);
```




```
MPI_Comm_split(
        MPI_Comm comm,
        int color,
        int key,
        MPI_Comm* newcomm)
```

MPI_Comm_split | Example

```c
// Get the rank and size in the original communicator
int world_rank, world_size;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

int color = world_rank / 4; // Determine color based on row

// Split the communicator based on the color and use the
// original rank for ordering
MPI_Comm row_comm;
MPI_Comm_split(MPI_COMM_WORLD, color, world_rank, &row_comm);

int row_rank, row_size;
MPI_Comm_rank(row_comm, &row_rank);
MPI_Comm_size(row_comm, &row_size);

printf("WORLD RANK/SIZE: %d/%d \t ROW RANK/SIZE: %d/%d\n",
       world_rank, world_size, row_rank, row_size);

MPI_Comm_free(&row_comm);
```

```
WORLD RANK/SIZE: 0/16    ROW RANK/SIZE: 0/4
WORLD RANK/SIZE: 1/16    ROW RANK/SIZE: 1/4
WORLD RANK/SIZE: 2/16    ROW RANK/SIZE: 2/4
WORLD RANK/SIZE: 3/16    ROW RANK/SIZE: 3/4
WORLD RANK/SIZE: 4/16    ROW RANK/SIZE: 0/4
WORLD RANK/SIZE: 5/16    ROW RANK/SIZE: 1/4
WORLD RANK/SIZE: 6/16    ROW RANK/SIZE: 2/4
WORLD RANK/SIZE: 7/16    ROW RANK/SIZE: 3/4
WORLD RANK/SIZE: 8/16    ROW RANK/SIZE: 0/4
WORLD RANK/SIZE: 9/16    ROW RANK/SIZE: 1/4
WORLD RANK/SIZE: 10/16   ROW RANK/SIZE: 2/4
WORLD RANK/SIZE: 11/16   ROW RANK/SIZE: 3/4
WORLD RANK/SIZE: 12/16   ROW RANK/SIZE: 0/4
WORLD RANK/SIZE: 13/16   ROW RANK/SIZE: 1/4
WORLD RANK/SIZE: 14/16   ROW RANK/SIZE: 2/4
WORLD RANK/SIZE: 15/16   ROW RANK/SIZE: 3/4
```

Overview of groups

- While MPI_Comm_split is the simplest way to create a new communicator, it isn't the only way to do so
- There are more flexible ways to create communicators, but they use a new kind of MPI object, **MPI_Group**
- A communicator consists of two parts:
    - **Context**: a unique ID that prevents operations from mixing across communicators
    - **Group of processes**: the set of processes in the communicator
- For MPI_COMM_WORLD, the group is all processes started by mpirun

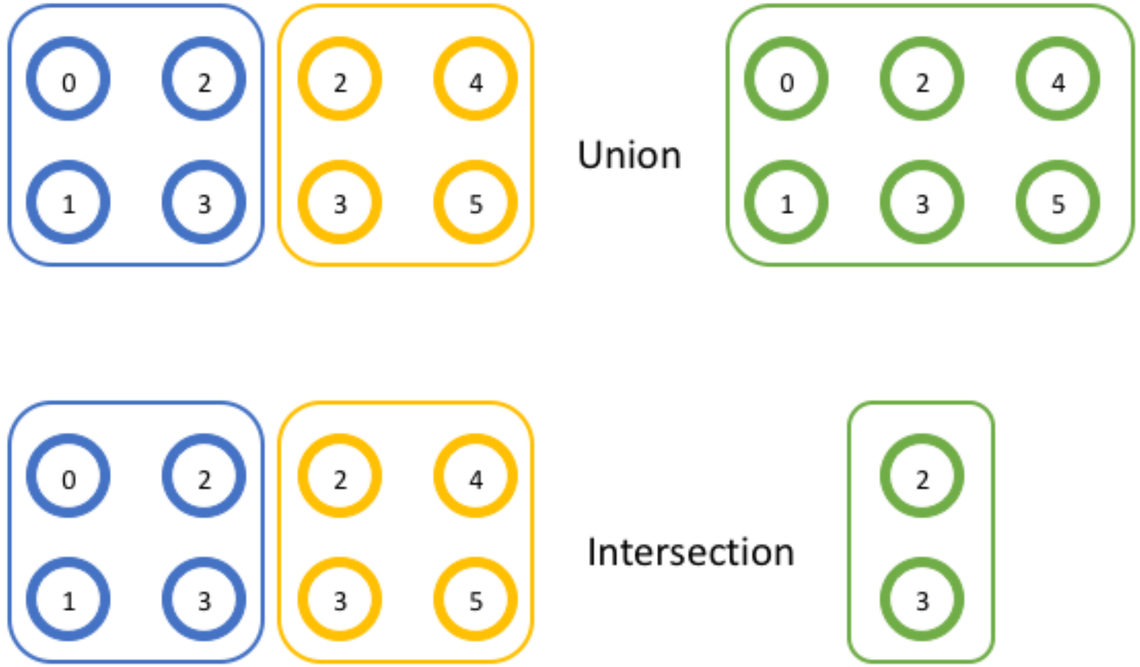
Working with Groups

- Get the group of processes in a communicator with the API call, MPI_Comm_group
- Calling MPI_Comm_group gets a reference to the group object
- The group object works the same way as a communicator object except that you can’t use it to communicate with other ranks
- You can still get the rank and size with MPI_Group_rank and MPI_Group_size

```
MPI_Comm_group(
        MPI_Comm comm,
        MPI_Group* group)
```

MPI Groups

- MPI uses these groups in the same way that set theory works
- Therefore, we can perform operations on them:
- The union of the two groups {0, 1, 2, 3} and {2, 3, 4, 5} is {0, 1, 2, 3, 4, 5}
- The intersection of the two groups {0, 1, 2, 3}, and {2, 3, 4, 5} is {2, 3}



Working with Groups

- You can do with groups things you can’t do with communicators.
- For example,
  - Union
  - Intersection

```
MPI_Group_union(
        MPI_Group group1,
        MPI_Group group2,
        MPI_Group* newgroup)
```

```
MPI_Group_intersection(
        MPI_Group group1,
        MPI_Group group2,
        MPI_Group* newgroup)
```

Creating Communicators from Groups

- This function takes an MPI_Group object and creates a new communicator that has all of the same processes as the group.

```
MPI_Comm_create_group(
        MPI_Comm comm,
        MPI_Group group,
        int tag,
        MPI_Comm* newcomm)
```

More Ways to Use Groups

- Function which allows you to pick specific ranks in a group and construct a new group containing only those ranks
- With this function, newgroup contains the processes in group with ranks contained in ranks, which is of size n

```
MPI_Group_incl(
        MPI_Group group,
        int n,
        const int ranks[],
        MPI_Group* newgroup)
```

Example

- In this example, we construct a communicator by selecting only the prime ranks in MPI_COMM_WORLD.
- This is done with MPI_Group_incl and results in prime_group.
- Next, we pass that group to MPI_Comm_create_group to create prime_comm.

```
// Get the rank and size in the original communicator
int world_rank, world_size;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

// Get the group of processes in MPI_COMM_WORLD
MPI_Group world_group;
MPI_Comm_group(MPI_COMM_WORLD, &world_group);

int n = 7;
const int ranks[7] = {1, 2, 3, 5, 7, 11, 13};

// Construct a group containing all of the prime ranks in world_group
MPI_Group prime_group;
MPI_Group_incl(world_group, 7, ranks, &prime_group);

// Create a new communicator based on the group
MPI_Comm prime_comm;
MPI_Comm_create_group(MPI_COMM_WORLD, prime_group, 0, &prime_comm);
```

The End