Introduction to MPI

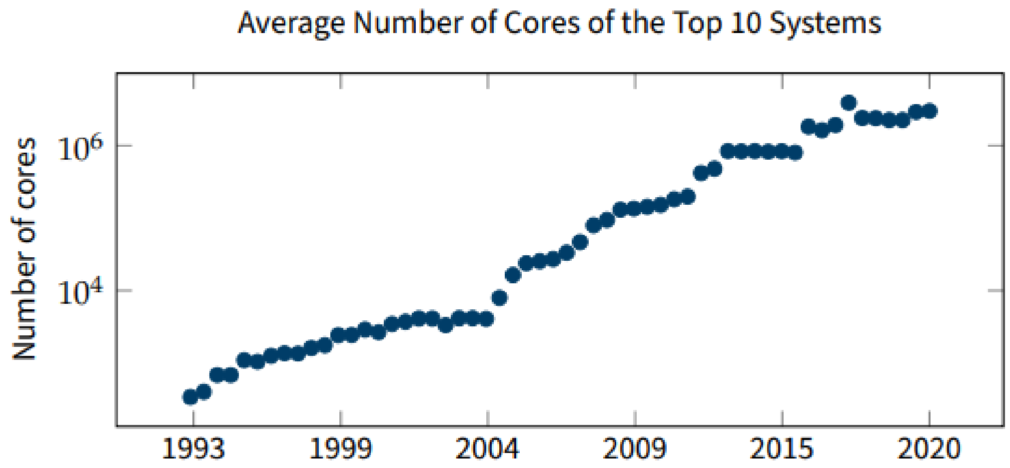
Luis Alejandro Torres Niño
SC3 Supercomputación y Cálculo Científico
Universidad Industrial de Santander

FUNDAMENTALS OF PARALLEL COMPUTING

PARALLEL COMPUTING

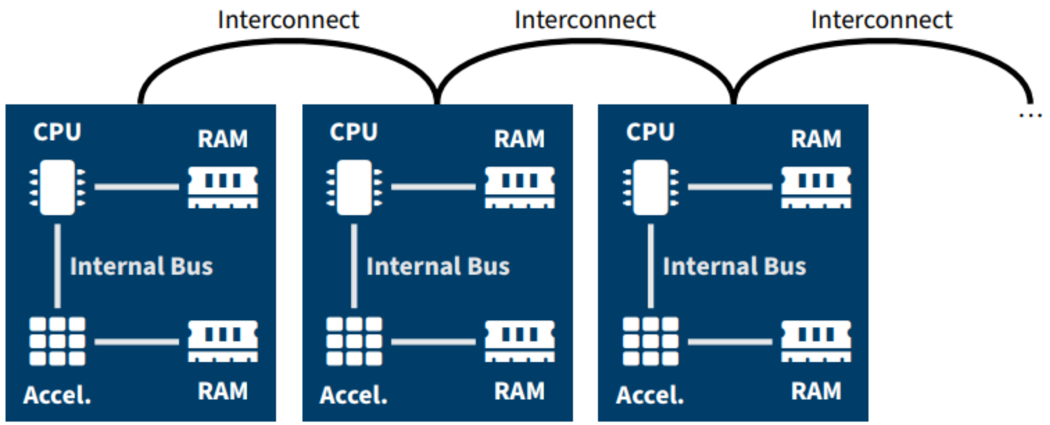
Parallel computing is a type of computation in which many calculations or the execution of processes are carried out simultaneously.

PARALLELISM IN THE TOP 500 LIST

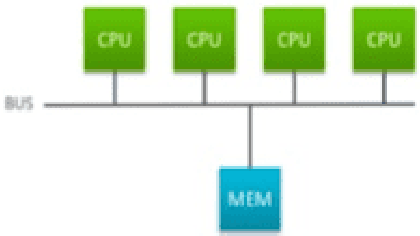


FUNDAMENTALS OF PARALLEL COMPUTING

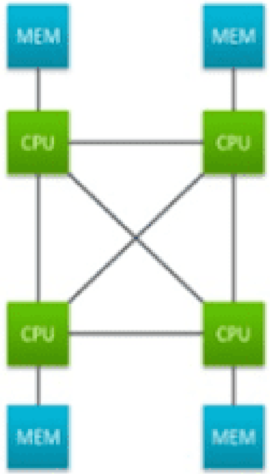
A MODERN SUPERCOMPUTER



Uniform Memory Access (UMA)



Non-Uniform Memory Access (NUMA)



FUNDAMENTALS OF PARALLEL COMPUTING

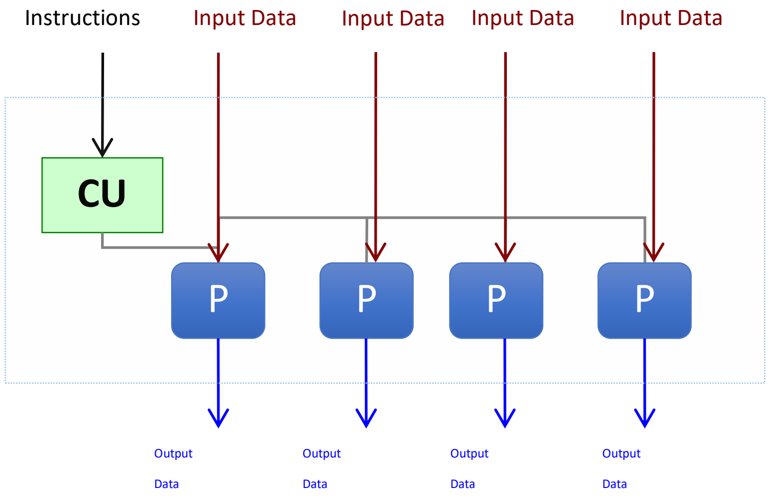
PARALLEL COMPUTING UNITS

- Implicit Parallelism
  - Parallel execution of different (parts of) processor instructions
  - Happens automatically
  - Can only be influenced indirectly by the programmer
- Multi-core / Multi-CPU
  - Found in commodity hardware today
  - Computational units share the same memory
- Cluster
  - Found in commodity hardware today
  - Independent systems linked via (fast) interconnect
  - Each system has its own memory
- Accelerators
  - Strive to perform certain tasks faster than is possible on a general-purpose CPU
  - Make different trade-offs
  - Often have their own memory
  - Often not autonomous
- Vector Processors / Vector Units
  - Perform the same operation on multiple pieces of data simultaneously
  - Making a come-back as SIMD unit in commodity CPUs (AVX-512) and GPGPU

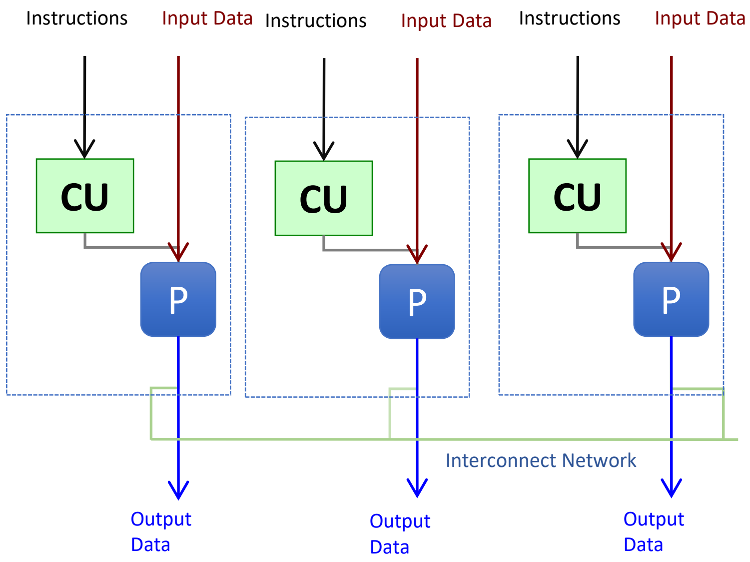
FUNDAMENTALS OF PARALLEL COMPUTING

SPECIAL FEATURES OF ARCHITECTURE

- Concurrency
- SIMD and MIMD Architectures



SIMD



MIMD

FUNDAMENTALS PARALLEL COMPUTING

MEMORY DOMAINS

- Shared Memory
  - All memory is directly accessible by the parallel computational units
  - Single address space
  - Programmer might have to synchronize access
- Distributed Memory
  - Memory is partitioned into parts which are private to the different computational units
  - “Remote” parts of memory are accessed via an interconnect
  - Access is usually nonuniform

FUNDAMENTALS PARALLEL COMPUTING

PROCESSES, THREADS, AND TASK

Abstractions for the independent execution of (part of) a program.

- Process
  - Usually, multiple processes can coexist, each with its own associated set of resources.
- Thread
  - Typically, “smaller” than processes
  - Often multiple threads per one process
  - Threads of the same process can share resources
- Task
  - Typically, “smaller” than threads
  - Often multiple tasks per one thread
  - User-level construct

FIRST STEPS WITH MPI

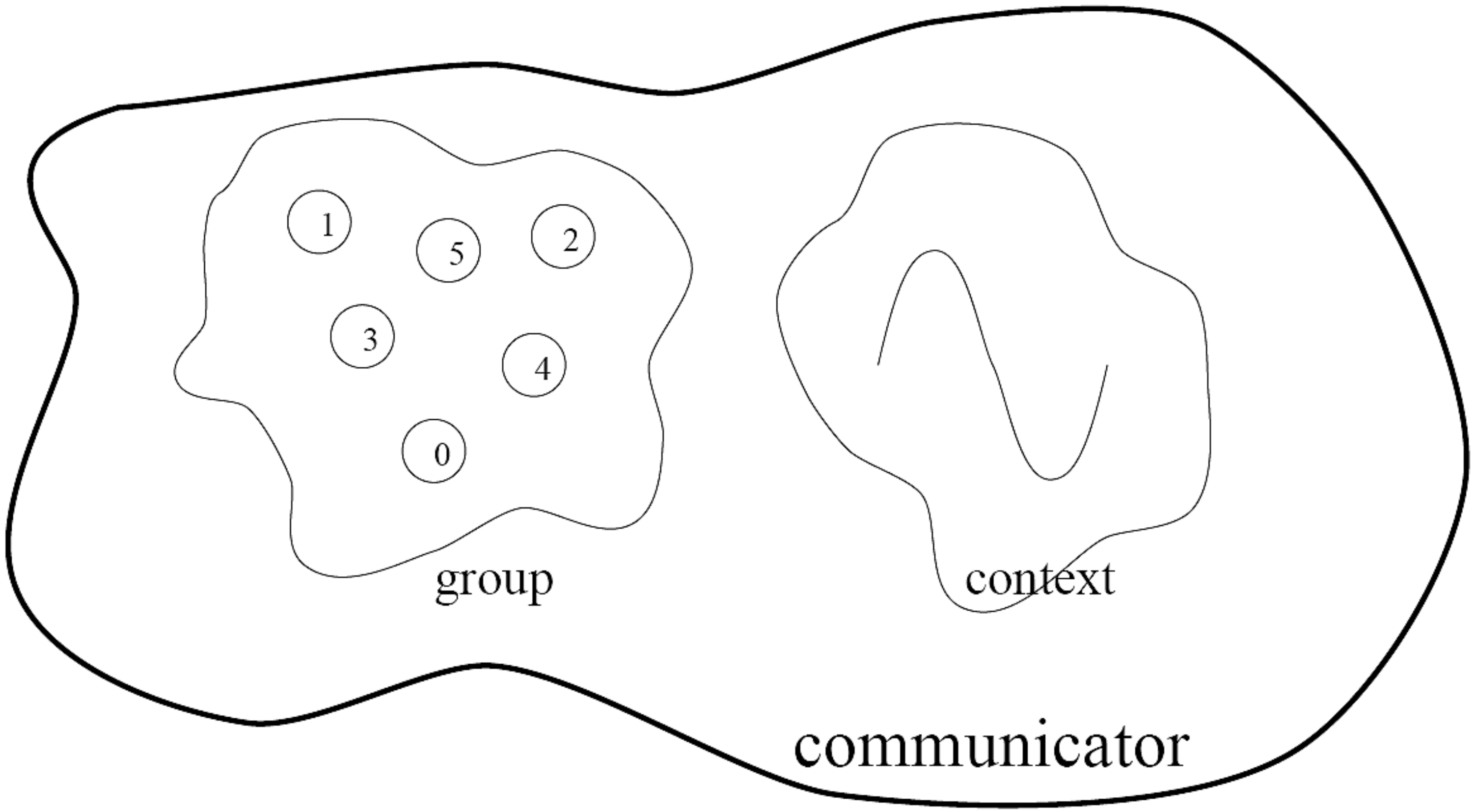
WHAT IS MPI?

- MPI **(M**essage-**P**assing **I**nterface**)** is a message-passing library interface specification.
- **MPI** addresses primarily the message-passing parallel programming model, in which data is moved from the address space of one process to that of another process through cooperative operations on each process.

TERMINOLOGY

- **Process:** An MPI program consists of autonomous processes, executing their own code in a MIMD style.
- **Rank:** A unique number assigned to each process within a group (start at 0)
- **Group:** An ordered set of process identifiers
- **Context:** A property that allows the partitioning of the communication space. It is like the frequency in radio communications.
- **Communicator:** Scope for communication operations within or between groups, combines the concepts of groups and context.

FIRST STEPS WITH MPI



FIRST STEPS WITH MPI

MPI PROGRAMMING

```c
#include <stdio.h>

int main(int argc, char **argv) {
        printf("Hello World!\n");
        return 0;
}
```

```
mpicc 0_hello.c -o 0_hello
./0_hello
```

FIRST STEPS WITH MPI

MPI PROGRAMMING

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

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

        printf("Hello World!\n");

        // Finalize MPI
        MPI_Finalize();

        return 0;
}
```

```
mpicc 1_hello_mpi.c -o 1_hello_mpi
./1_hello_mpi
mpirun ./1_hello_mpi
```

FIRST STEPS WITH MPI

MPI PROGRAMMING

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

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

        printf("Hello from %d.\n", rank);

        MPI_Finalize();
        return 0;
}
```

FIRST STEPS WITH MPI

MPI PROGRAMMING

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

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

	printf("Hello from %d.\n", rank);

	MPI_Finalize();
	return 0;
}
```

FIRST STEPS WITH MPI

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

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

        printf("Hello from %d.\n", rank);

        MPI_Finalize();
        return 0;
}
```


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

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

        printf("Hello from %d.\n", rank);

        MPI_Finalize();
        return 0;
}
```


Hello from process 0 of 4.
Hello from process 3 of 4.
Hello from process 1 of 4.
Hello from process 2 of 4.


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

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

        printf("Hello from %d.\n", rank);

        MPI_Finalize();
        return 0;
}
```

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

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

        printf("Hello from %d.\n", rank);

        MPI_Finalize();
        return 0;
}
```

FIRST STEPS WITH MPI

GENERAL OBSERVATIONS

- We must include the header file **“mpi.h”**
- The MPI calls always start with **MPI_**
- **MPI_Init** starts MPI
- **MPI_Finalize** ends MPI
- Default communicator **MPI_COMM_WORLD** groups all process
- All non-MPI instructions are executed locally in each of the processes
- By default, an error abort all processes
- All MPI functions return error codes or **MPI_SUCCESS**
- Users can program their own error handling routines

FIRST STEPS WITH MPI

THE SAME CODE?

- All nodes run the same code independently.
- But they can use their rank id to diverge in behavior as much as they like.
- This is an extreme case where each rank executes different instructions:

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

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

    int rank, size;

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

    if (rank == 0)
        printf("I am the Boss with id %d\n",
    rank);
    if (rank == 1)
        printf("I am a slave with id %d\n",
    rank);
    if (rank == 2)
        printf("I am other slave with id %d\n",
    rank);
    if (rank == 3)
        printf("I am last slave with id %d\n",
    rank);

    MPI_Finalize();
    return 0;
}
```

FIRST STEPS WITH MPI

COMMON CASE

The most common use case is using a unique rank for some coordination tasks and the others to run the same code with different data.

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

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

    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0)
        printf("I am the Boss with id %d\n",
    rank);
    else
        printf("I am a slave with id %d\n",
    rank);
    MPI_Finalize();

    return 0;
}
```

FIRST STEPS WITH MPI

SENDING AND RECEIVING

```
MPI_SEND(void *start, int count,MPI_DATATYPE datatype, int dest, int
  tag, MPI_COMM comm)
```

- The message buffer is described by (start, count, datatype).
- dest is the rank of the target process in the defined communicator.
- tag is the message identification number.
- count is the number of items to send. If we send a vector of 10 int's, the first parameter would be the memory address of the variable of the array and set this to the size of the array.

FIRST STEPS WITH MPI

SENDING AND RECEIVING

```
MPI_RECV(void *start, int count, MPI_DATATYPE datatype, int source,
  int tag, MPI_COMM comm, MPI_STATUS *status)
```

- Source is the rank of the sender in the communicator.
- The receiver can specify a wildcard value for the source (MPI_ANY_SOURCE) and/or a wildcard value for tag (MPI_ANY_TAG), indicating that any source and/or tag are acceptable
- Status is used for extra information about the received message if a wildcard receive mode is used.
- If the count of the message received is less than or equal to that described by the `MPI_RECV` command, then the message is successfully received else it is considered as a buffer overflow error.

FIRST STEPS WITH MPI

DATA TYPES

MPI Datatypes are necessary because the communications take place between heterogeneous machines which could have different data representations and different memory sizes.

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

FIRST STEPS WITH MPI

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

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

    int rank, size;
    int numberToSend = 42;
    int numberToReceive;

    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0) {
        MPI_Recv(&numberToReceive, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
    MPI_COMM_WORLD, &status);
        printf("Number received is: %d\n", numberToReceive);
    } else {
        MPI_Send(&numberToSend, 1, MPI_INT, 0, 10, MPI_COMM_WORLD);
    }
    MPI_Finalize();
    return 0;
}
```

FIRST STEPS WITH MPI

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

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

    int rank, size;
    int numberToSend = 42;
    int numberToReceive;

    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0) {
        MPI_Recv(&numberToReceive, 1, MPI_INT, 0, 10, MPI_COMM_WORLD, &status);
        printf("Number received is: %d\n", numberToReceive);
    } else {
        MPI_Send(&numberToSend, 1, MPI_INT, 0, 10, MPI_COMM_WORLD);
    }
    MPI_Finalize();
    return 0;
}
```

FIRST STEPS WITH MPI

Exercise 1. Write a code that simulates the game of Ping Pong.

Exercise 2. Write a code that the rank zero should send the number 10 and rank one should receive it and divide it by two. This must send it back to zero range to be printed on the screen.

FIRST STEPS WITH MPI

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

int main(int argc, char **argv) {
	const int PING_PONG_LIMIT = 10;
	int ping_pong_count = 0;
	int partner_rank;
	int size, rank, count;
	MPI_Init(&argc, &argv);
	MPI_Comm_rank(MPI_COMM_WORLD, &rank);
	MPI_Comm_size(MPI_COMM_WORLD, &size);
	if (size != 2) {
		printf("World size must be two for %s\n", argv[0]);
		MPI_Abort(MPI_COMM_WORLD, 1);
	}
	ping_pong_count = 0;
	partner_rank = (rank + 1) % 2;
	while (ping_pong_count < PING_PONG_LIMIT) {
		if (rank == ping_pong_count % 2) {
			// Increment the ping pong count before you send it
			ping_pong_count++;
			MPI_Send(&ping_pong_count, 1, MPI_INT, partner_rank, 0, MPI_COMM_WORLD);
			printf("%d sent and incremented ping_pong_count %d to %d\n", rank, ping_pong_count,
partner_rank);
		} else {
			MPI_Recv(&ping_pong_count, 1, MPI_INT, partner_rank, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
			printf("%d received ping_pong_count %d from %d\n", rank, ping_pong_count, partner_rank);
		}
	}
	MPI_Finalize();
	return 0;
}
```

FIRST STEPS WITH MPI

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

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

        int rank;
        int number = 10;
        int result;
        int dataToReceive;

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

        if (rank == 0) {
                MPI_Send(&number, 1, MPI_INT, 1, 10, MPI_COMM_WORLD);
                MPI_Recv(&result, 1, MPI_INT, 1, 10, MPI_COMM_WORLD, &status);
                printf("Number %d divide between 2 is = %d\n", number, result);
        }

        if (rank == 1) {
                MPI_Recv(&dataToReceive, 1, MPI_INT, 0, 10, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                dataToReceive = dataToReceive / 2;
                MPI_Send(&dataToReceive, 1, MPI_INT, 0, 10, MPI_COMM_WORLD);
        }

        MPI_Finalize();

        return 0;
}
```

FIRST STEPS WITH MPI

Exercise 3. Rewrite this code in MPI

```c
#include <stdio.h>

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

        int sum = 0;
        int i;

        for (i=1; i<1000; i++)
                sum = sum + i;

        printf("The sum from 1 to 1000 is: %d\n", sum);

        return 0;
}
```

NOTE:

```c
startVal = 1000 * rank / size + 1;
endVal = 1000 * (rank + 1)/size;
```

FIRST STEPS WITH MPI

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

int main(int argc, char** argv) {
        int sum;
        int rank, size;
        int startVal, endVal, accum;
        int i, j;

        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        sum = 0;
        startVal = 1000 * rank / size + 1;
        endVal = 1000 * (rank + 1)/size;
        for (i = starVal; i < endVal; i++)
                sum = sum + i;

        if (rank != 0)
                MPI_Send(&sum, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
        else {
                for (j = 1; j < size; j++) {
                        MPI_Recv(&accum, 1, MPI_INT, j, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                        sum = sum + accum;
                }
        }

        if (rank == 0)
                printf("The sum from 1 to 1000 is: %d\n", sum);

        MPI_Finalize();

        return 0;
}
```

FIRST STEPS WITH MPI

Point-to-Point Communication

Blocking Operations:

- **Send**
- **Recv**
- SendRecv
- Bsend
- Ssend
- Rsend

Communications Modes
The communication mode is indicated by a one-letter prefix:

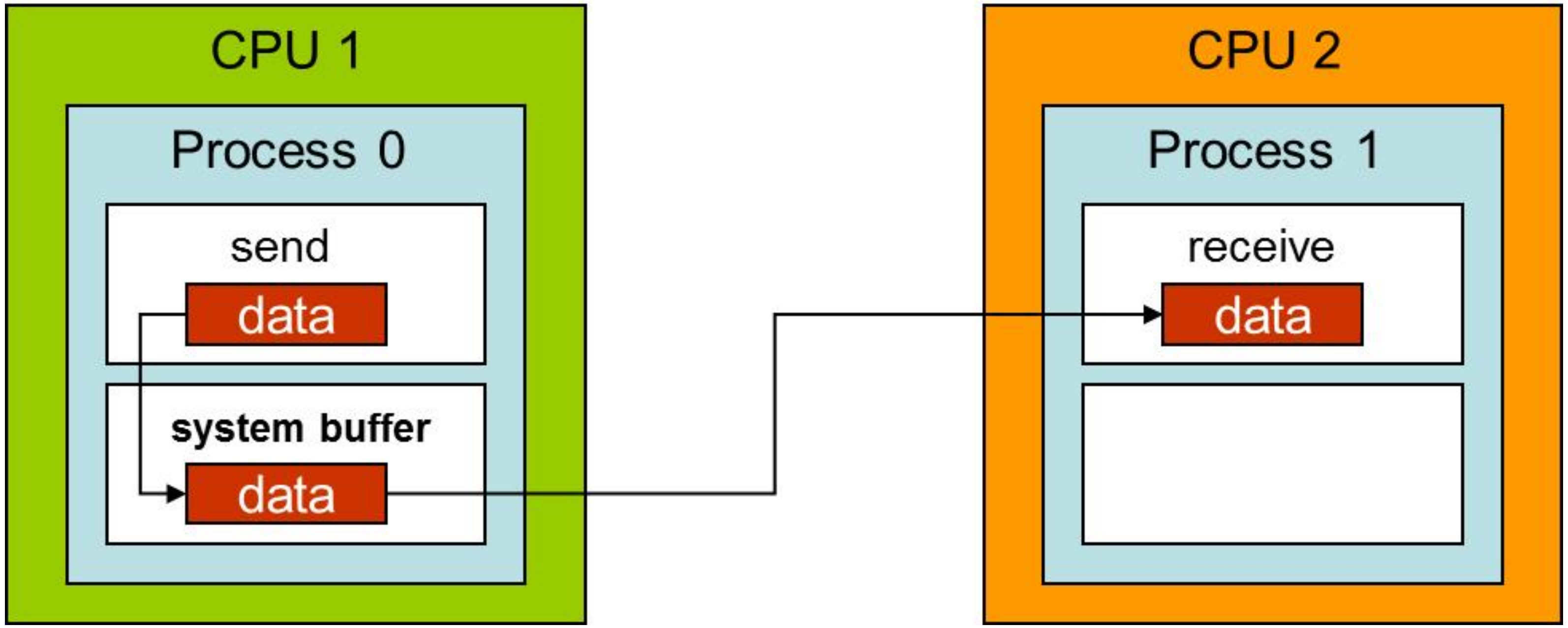
- **B**: buffered
- **S**: Syncronous
- **R**: Ready

Non-Blocking Operations:

- Isend
- Irecv

FIRST STEPS WITH MPI

Buffered Mode




```
MPI_Bsend(const void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
```

FIRST STEPS WITH MPI

Buffered Mode

```c
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char **argv) {
        int numtasks, rank, dest, source, rc, count;
        char *inmsg;
        char *outmsg = "Testing";

        MPI_Status Stat;

        int bufsize = strlen(outmsg) * sizeof(char);
        char *buf = malloc(bufsize);
        inmsg = (char *) malloc(10 * sizeof(char));

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

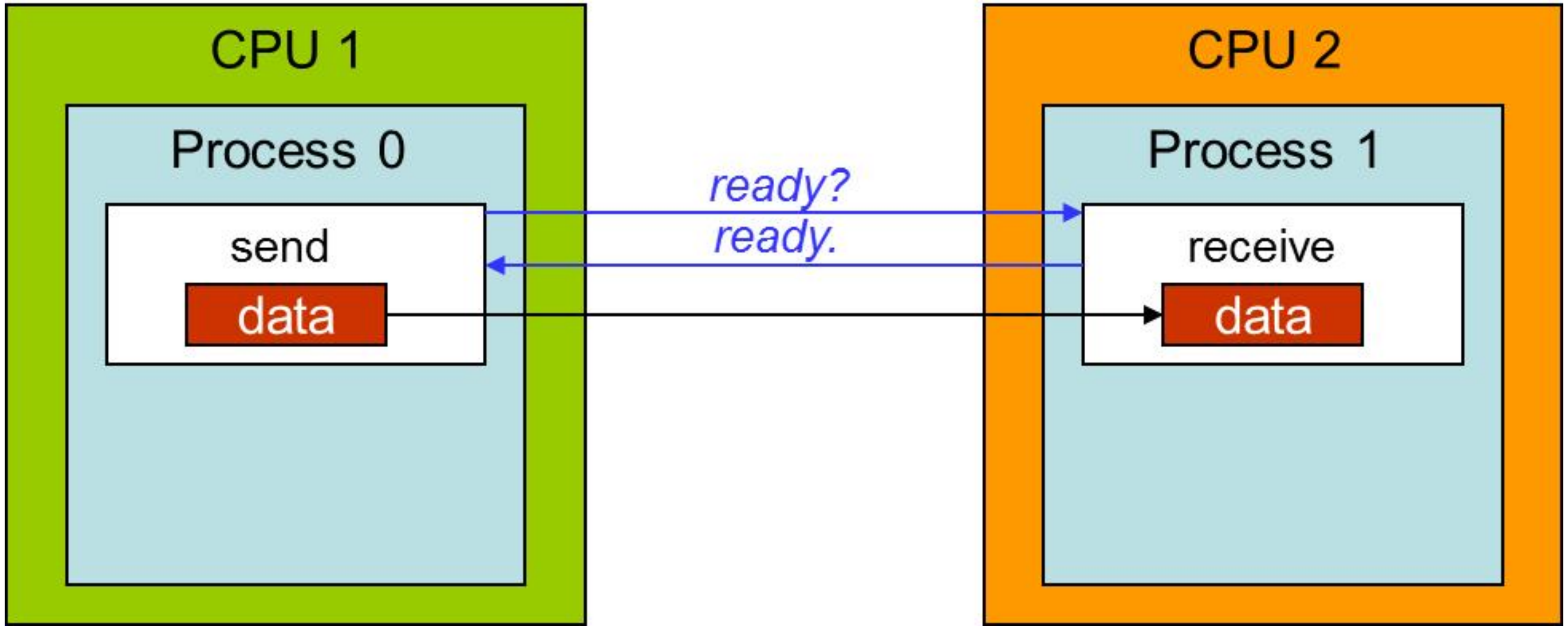
FIRST STEPS WITH MPI

Buffered Mode

```
        if (rank == 0) {
                MPI_Buffer_attach( buf, bufsize );
                int time = MPI_Wtime();
                MPI_Bsend(&outmsg, strlen(outmsg), MPI_CHAR, 1, 1, MPI_COMM_WORLD);
                float etime = MPI_Wtime() - time;
                printf("Task %d: buffered send (1) buffered , process will block on detach. Time: %1.2f\n",rank,etime);
                MPI_Buffer_detach( &buf, &bufsize );
                time = MPI_Wtime();
                MPI_Send(&outmsg, strlen(outmsg), MPI_CHAR, 1, 1, MPI_COMM_WORLD);
                etime = MPI_Wtime() - time;
                printf("Task %d: buffered send (2), process may will block here. Time: %1.2f\n",rank, etime);
        } else {
                MPI_Recv(&inmsg, strlen(outmsg), MPI_CHAR, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &Stat);
                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_Recv(&inmsg, strlen(outmsg), MPI_CHAR, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &Stat);
                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();
}
```

FIRST STEPS WITH MPI

Synchronous Mode




```
MPI_Ssend(const void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
```

FIRST STEPS WITH MPI

Synchronous Mode

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

int main(int argc, char **argv) {
        int rank, size, i;
        int buffer[10];

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

        if (size < 2) {
                printf("Please run with two processes.\n");
                fflush(stdout);
                MPI_Finalize();
                return 0;
        }
```

FIRST STEPS WITH MPI

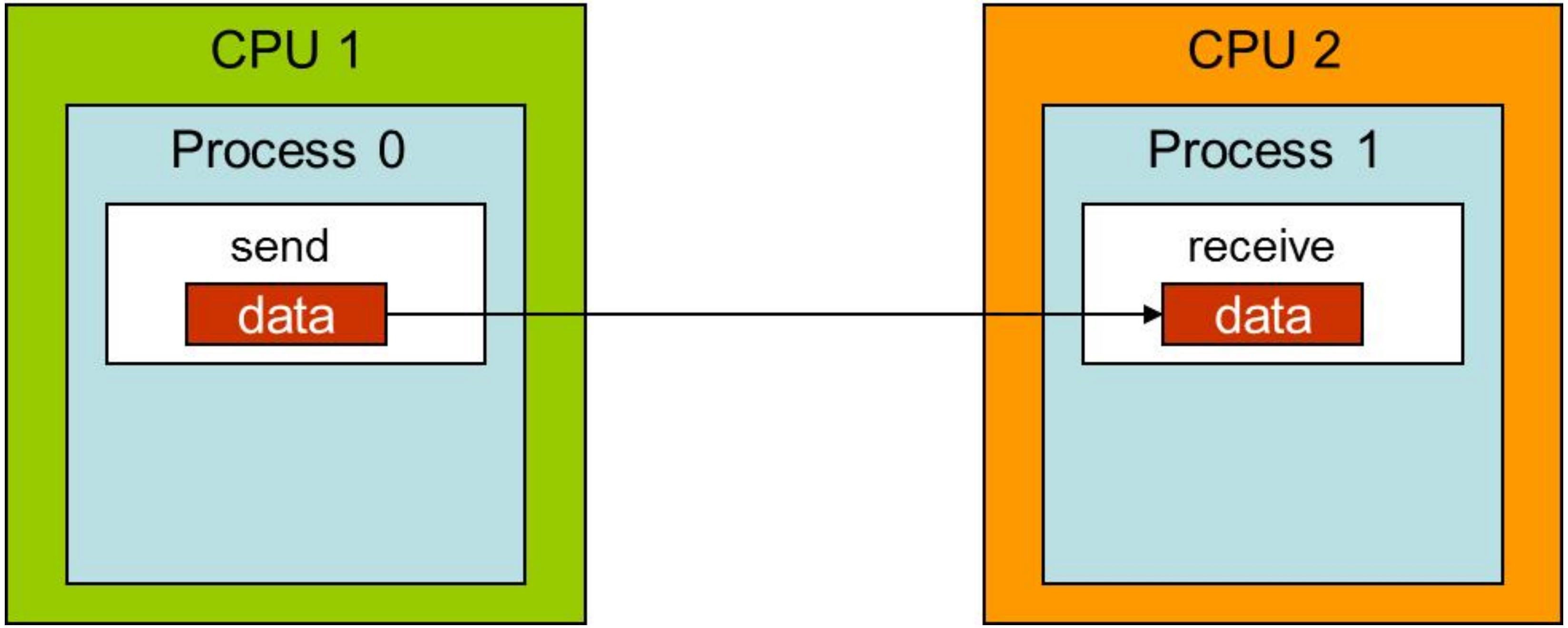
Synchronous Mode

```
    if (rank == 0) {
            for (i=0; i<10; i++)
                    buffer[i] = i;
            MPI_Ssend(buffer, 10, MPI_INT, 1, 123, MPI_COMM_WORLD);
    }

    if (rank == 1) {
            for (i=0; i<10; i++)
                    buffer[i] = -1;
            MPI_Recv(buffer, 10, MPI_INT, 0, 123, MPI_COMM_WORLD, &status);
            for (i=0; i<10; i++) {
                    if (buffer[i] != i)
                            printf("Error: buffer[%d] = %d but is expected to be %d\n", i, buffer[i], i);
            }
            fflush(stdout);
    }
    MPI_Finalize();
    return 0;
}
```

FIRST STEPS WITH MPI

Ready Mode




```
MPI_Rsend(const void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
```

FIRST STEPS WITH MPI

Collective Communications

Collective communication is defined as communication that involves a group or groups of processes.

Categories:

- All-To-One
- One-To-All
- All-To-All
- Other

Routines

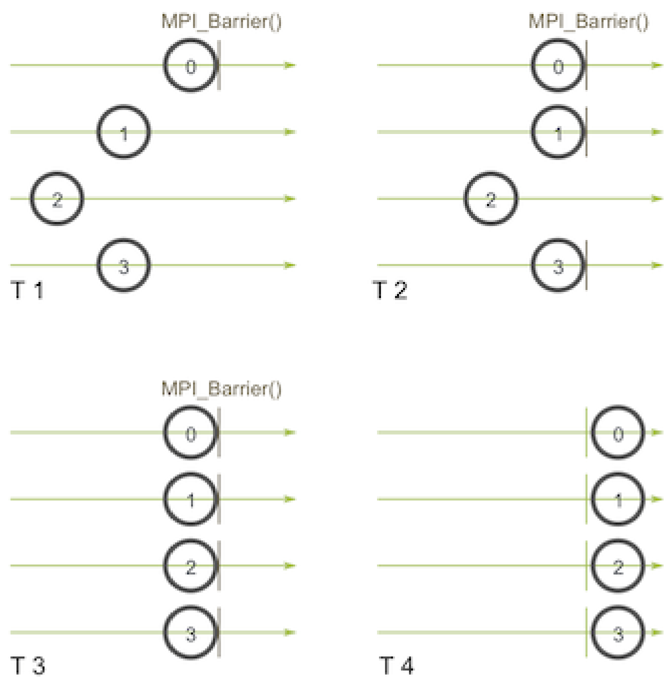
- Barrier
- Bcast
- Gather
- Scatter

FIRST STEPS WITH MPI

Barrier

blocks the caller until all group members have called it. The call returns at any process only after all group members have entered the call.

```
MPI_Barrier(MPI_Comm comm)
```



FIRST STEPS WITH MPI

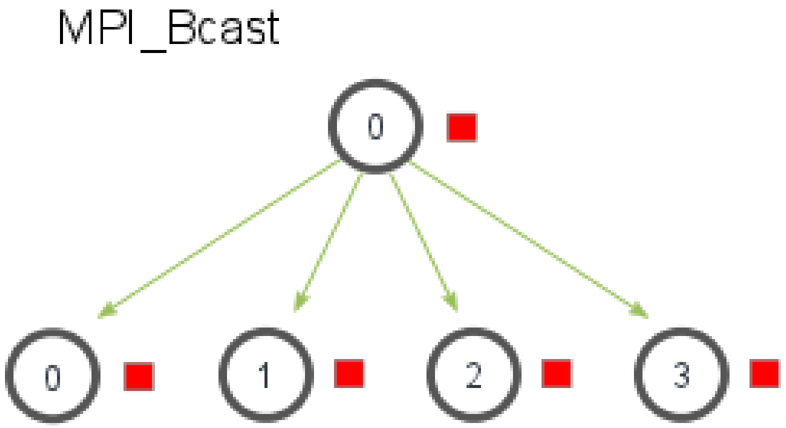
Barrier

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

int main(int argc, char **argv) {
        int rank, nprocs;
        MPI_Init(&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
        MPI_Comm_rank(MPI_COMM_WORLD,&rank);
        MPI_Barrier(MPI_COMM_WORLD);
        printf("Hello, world. I am %d of %d\n", rank, nprocs);
        fflush(stdout);
        MPI_Finalize();
        return 0;
}
```

FIRST STEPS WITH MPI
Broadcast

broadcasts a message from the process with rank root to all processes of the group, itself included




```
MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
```

FIRST STEPS WITH MPI

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

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

        int rank, value;

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

        if (rank == 0) {
                printf("Enter a number to broadcast:\n");
                scanf("%d", &value);
        } else {
                printf("process %d: Before MPI_Bcast, value is %d\n", rank, value);
        }

        MPI_Bcast(&value, 1, MPI_INT, 0, MPI_COMM_WORLD);

        printf("process %d: After MPI_Bcast, value is %d\n", rank, value);

        MPI_Finalize();
        return 0;
}
```

FIRST STEPS WITH MPI

And the rest?

HAVE FUN WITH MPI

[Let's start to have fun with MPI - Have fun with MPI in C (codingame.com)](https://www.codingame.com/playgrounds/47058/have-fun-with-mpi-in-c/lets-start-to-have-fun-with-mpi)

https://www.codingame.com/playgrounds/47058/have-fun-with-mpi-in-c/lets-start-to-have-fun-with-mpi