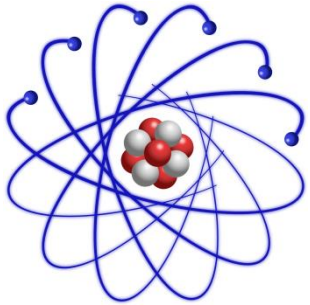
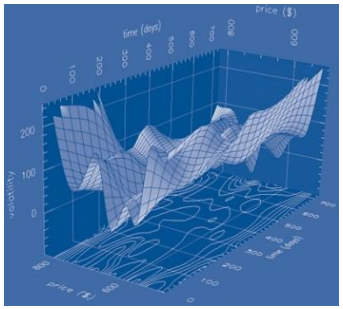
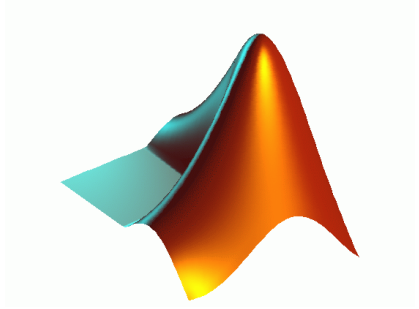




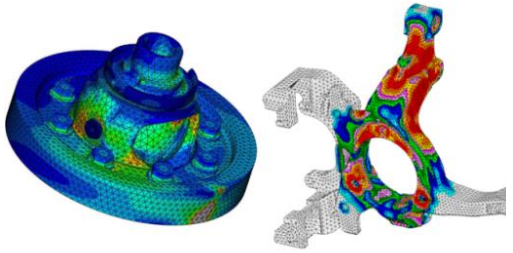


"Message Passing Interface (MPI) Programming model: Distributed Parallel Simulations"

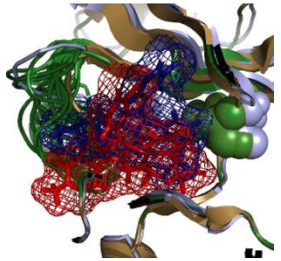
Dott. Costantino Zazza

What is a Cluster ?

- Generally speaking we have that…*”A Cluster is a group of servers/workstations interconnected through a dedicated network designed to act like a single IT processing resource/infrastructure. Clustering improves performance and the system's availability to users actually by distributing workload across all connected servers (multitasking/multiuser environment).”*

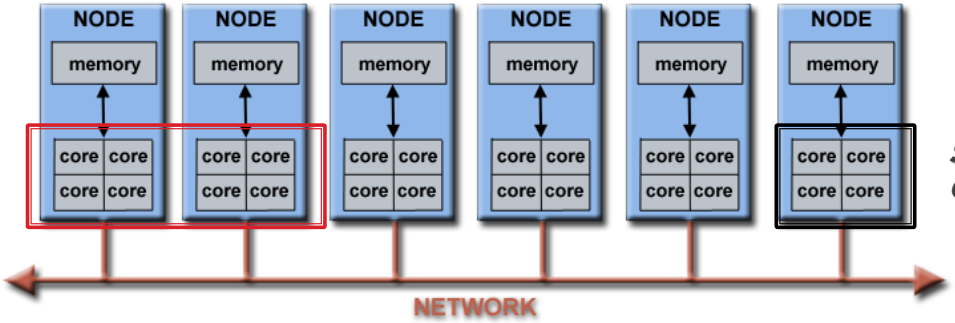


High Performance Computing



What does Parallel Computing really mean ?

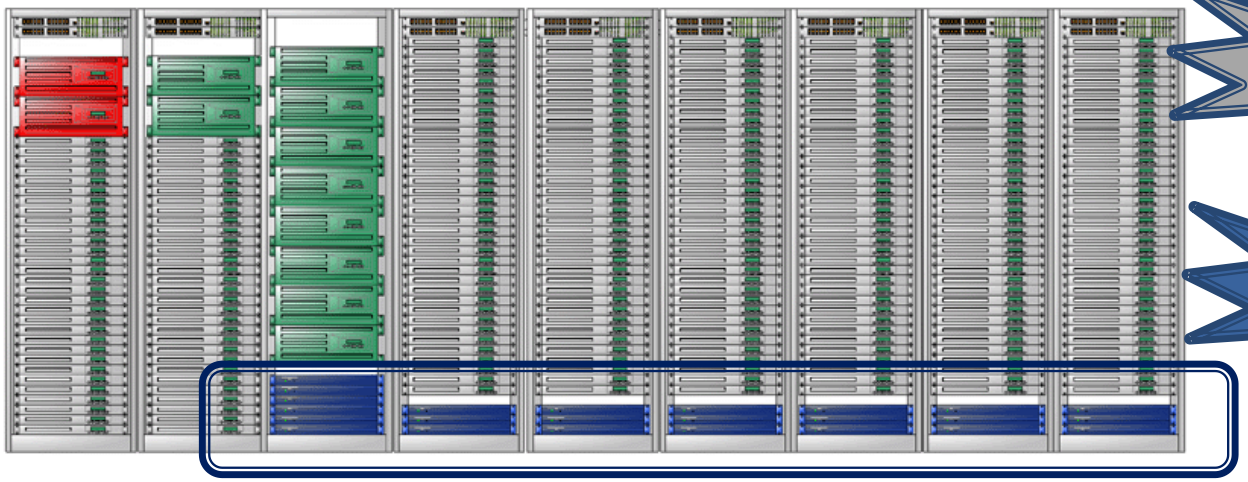
- ***Shared vs. Distributed (Memory) Parallel Computing:***



Distributed Parallel Computation

Shared Parallel Computation

CLUSTER VIEW :



Single-node simulation !

Multi-node simulation !

compute node

infiniband switch

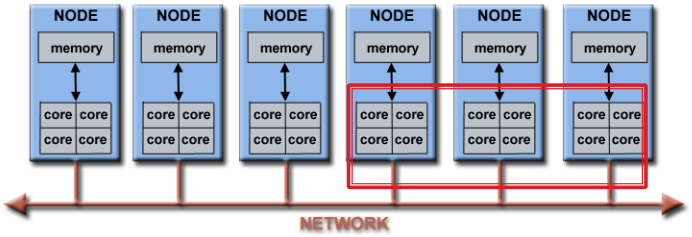
management hardware

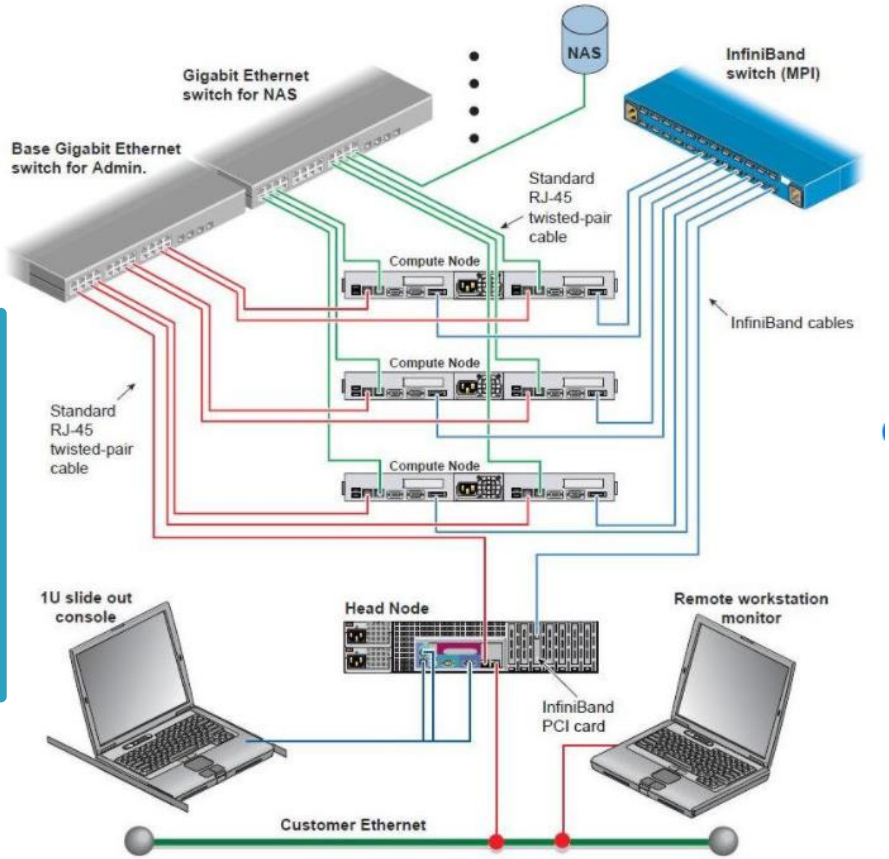
login / remote partition server node

gateway node

What does Parallel Computing really mean ?

- ***Distributed (Memory) Parallel Computing:***





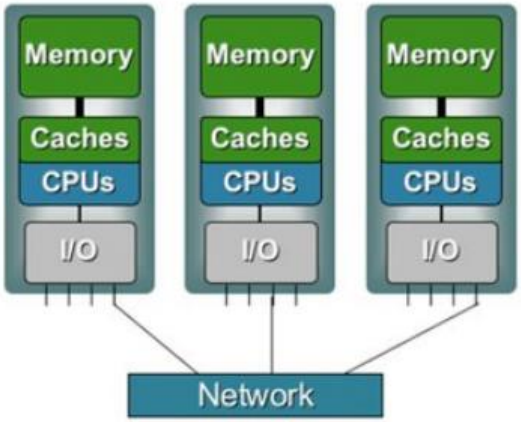
Dual Ethernet and Infiniband switch cluster configuration example

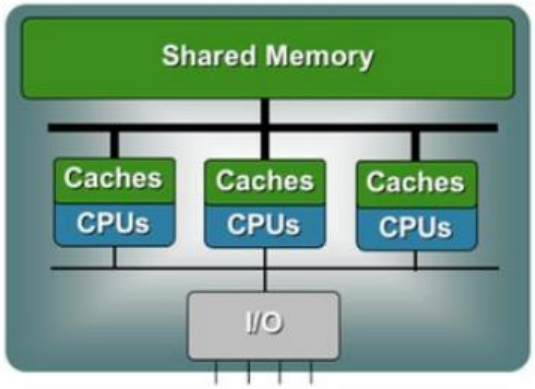
3 compute nodes 1 head node 3 switches

PS. Global Array provides a friendly API for shared-memory programming on distributed-memory computers for multidimensional arrays **but performance may be strongly affected by non local data availability..**

Shared Vs Distributed Memory architectures

- All processors may access the whole main – RAM – memory **at run time…**





- Non-Uniform Memory Access
  - Memory access time is non-uniform

NUMA

- Uniform Memory Access
  - Memory access time is uniform

UMA

What is MPI ?

An Interface Specification

M P I = **M**essage **P**assing **I**nterface

MPI is a specification for the developers and users of message passing libraries. By itself, it is NOT a library - but rather the specification of what such a library should be.

MPI primarily *addresses the message-passing parallel programming model*: data is moved from the address space of one process to that of another process through cooperative operations on each process.

Simply stated, the goal of the Message Passing Interface is to provide a widely used standard for writing message passing programs. The interface attempts to be:

- Practical
- Portable
- Efficient
- Flexible

The MPI standard has gone through a number of revisions, with the most recent version being MPI-3.x

Interface specifications have been defined for C and Fortran90 language bindings:

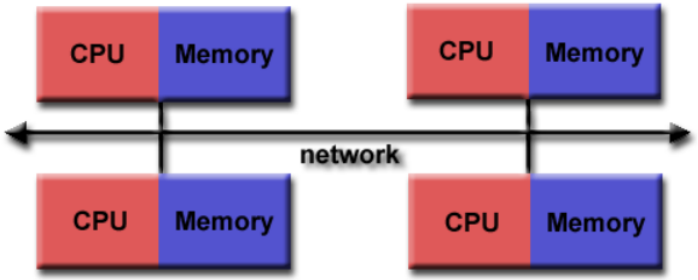
- C++ bindings from MPI-1 are removed in MPI-3
- MPI-3 also provides support for Fortran 2003 and 2008 features

Actual MPI library implementations differ in which version and features of the MPI standard they support. Developers/users will need to be aware of this.

What is MPI ?

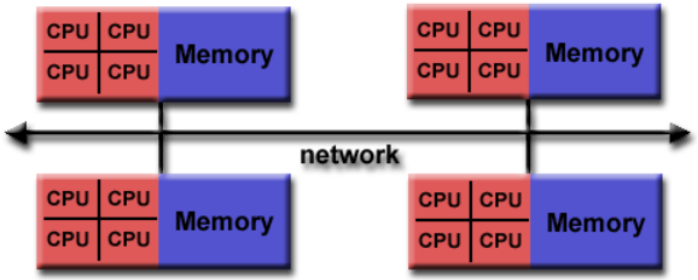
Programming Model

Originally, MPI was designed for distributed memory architectures, which were becoming increasingly popular at that time (1980s - early 1990s).



As architecture trends changed, shared memory SMPs were combined over networks creating hybrid distributed memory / shared memory systems.

MPI implementors adapted their libraries to handle both types of underlying memory architectures seamlessly. They also adapted/developed ways of handling different interconnects and protocols.



Today, MPI runs on virtually any hardware platform:

- Distributed Memory
- Shared Memory
- Hybrid

The programming model *clearly remains a distributed memory model* however, regardless of the underlying physical architecture of the machine.

Reasons for Using MPI

- **Standardization -** MPI is the only message passing library that can be considered a standard. It is supported on virtually all HPC platforms. Practically, it has replaced all previous message passing libraries.
- **Portability -** There is little or no need to modify your source code when you port your application to a different platform that supports (and is compliant with) the MPI standard.
- **Performance Opportunities -** Vendor implementations should be able to exploit native hardware features to optimize performance. Any implementation is free to develop optimized algorithms.
- **Functionality -** There are over 430 routines defined in MPI-3, which includes the majority of those in MPI-2 and MPI-1.
  - NOTE: Most MPI programs can be written using a dozen or less routines
- **Availability -** A variety of implementations are available, both vendor and public domain.

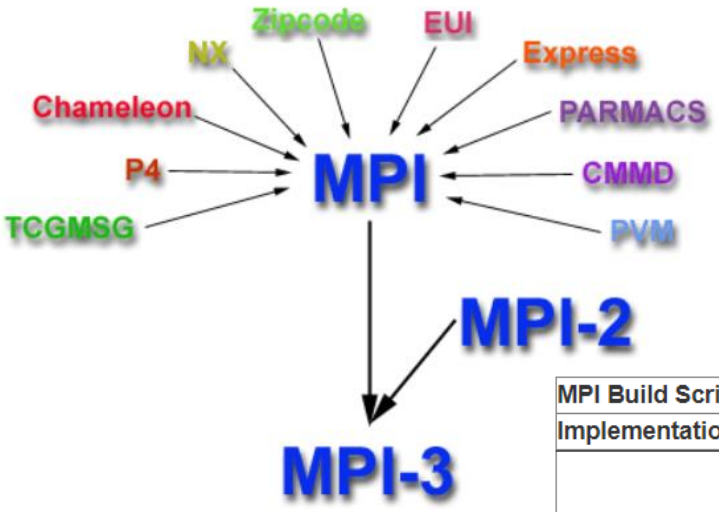
History and Evolution: (for those interested)

MPI has resulted from the efforts of numerous individuals and groups that began in 1992. Some history:

- **1980s - early 1990s**: Distributed memory, parallel computing develops, as do a number of incompatible software tools for writing such programs - usually with tradeoffs between portability, performance, functionality and price. Recognition of the need for a standard arose.
- **Apr 1992**: Workshop on Standards for Message Passing in a Distributed Memory Environment, sponsored by the Center for Research on Parallel Computing, Williamsburg, Virginia. The basic features essential to a standard message passing interface were discussed, and a working group established to continue the standardization process. Preliminary draft proposal developed subsequently.
- **Nov 1992**: Working group meets in Minneapolis. MPI draft proposal (MPI1) from ORNL presented. Group adopts procedures and organization to form the MPI Forum. It eventually comprised of about 175 individuals from 40 organizations including parallel computer vendors, software writers, academia and application scientists.
- **Nov 1993**: Supercomputing 93 conference - draft MPI standard presented.
- **May 1994**: Final version of MPI-1.0 released
  - MPI-1.1 (Jun 1995)
  - MPI-1.2 (Jul 1997)
  - MPI-1.3 (May 2008).
- **1998**: MPI-2 picked up where the first MPI specification left off, and addressed topics which went far beyond the MPI-1 specification.
  - MPI-2.1 (Sep 2008)

Reasons for Using MPI

- **Sep 2012**: The MPI-3.0 standard was approved.
  - MPI-3.1 (Jun 2015)
- **Current**: The MPI-4.0 standard is under development.



Tor Vergata centro di calcolo

MPI Build Scripts - Linux Clusters			
Implementation	**Language**	**Script Name**	**Underlying Compiler**
MVAPCH2	C	`mpicc`	C compiler for loaded compiler package
	C++	`mpicxx` `mpic++`	C++ compiler for loaded compiler package
	Fortran	`mpif77`	Fortran77 compiler for loaded compiler package. Points to mpifort.
		`mpif90`	Fortran90 compiler for loaded compiler package. Points to mpifort.
		`mpifort`	Fortran 77/90 compiler for loaded compiler package.
Open MPI	C	`mpicc`	C compiler for loaded compiler package

	C++	`mpiCC`
		`mpic++`
		`mpicxx`
			C++ compiler for loaded compiler package

	Fortran	`mpif77`	Fortran77 compiler for loaded compiler package. Points to mpifort.
		`mpif90`	Fortran90 compiler for loaded compiler package. Points to mpifort.
		`mpifort`	Fortran 77/90 compiler for loaded compiler package.

OPENMPI

MPI program must **include** the **MPI header file**

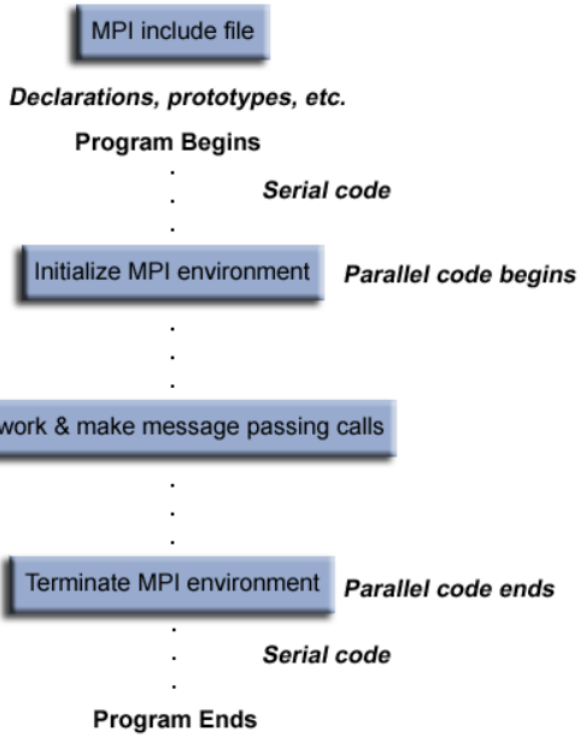
MPI PARALLEL REGION BEGINS

COMPUTATIONS AND COMMUNICATIONS

MPI PARALLEL REGION ENDS

Simile a quanto visto con OpenMP / SMP

General MPI Program Structure:



Header File:

Required for all programs that make MPI library calls.

C include file	Fortran include file
#include "mpi.h"	include 'mpif.h'

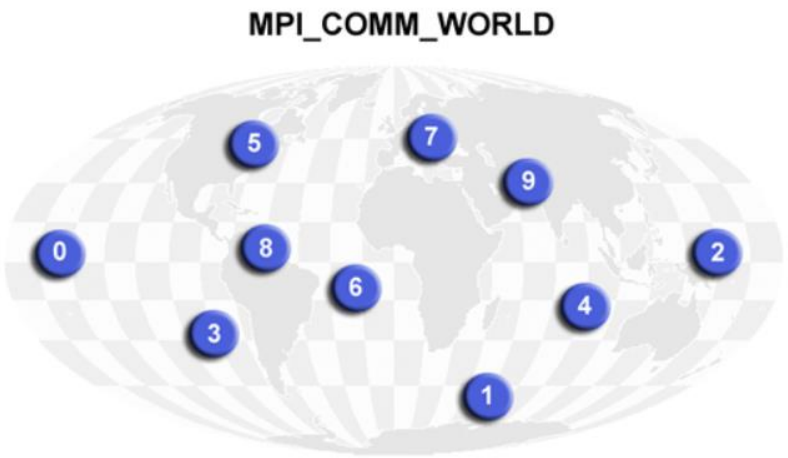
OPENMPI- FORMAT

Format of MPI Calls:

C names are case sensitive; Fortran names are not.

Programs must not declare variables or functions with names beginning with the prefix `MPI_` or `PMPI_` (profiling interface).

C Binding	
Format:	**rc = MPI_Xxxxx(parameter, ...)**
Example:	**rc = MPI_Bsend(&buf,count,type,dest,tag,comm)**
Error code:	Returned as "rc". MPI_SUCCESS if successful
Fortran Binding	
Format:	**CALL MPI_XXXXX(parameter,..., ierr)** **call mpi_xxxxx(parameter,..., ierr)**
Example:	**CALL MPI_BSEND(buf,count,type,dest,tag,comm,ierr)**
Error code:	Returned as "ierr" parameter. MPI_SUCCESS if successful



Communicators and Groups:

MPI uses objects called communicators and groups to define which collection of processes may communicate with each other.

Most MPI routines require you to specify a communicator as an argument.

Communicators and groups will be covered in more detail later. For now, simply use `MPI_COMM_WORLD` whenever a communicator is required - it is the predefined communicator th
includes all of your MPI processes.

OPENMPI- Environment Management Routines

MPI_Init

Initializes the MPI execution environment. This function must be called in every MPI program, must be called before any other MPI functions and must be called only once in an MPI program. For C programs, MPI_Init may be used to pass the command line arguments to all processes, although this is not required by the standard and is implementation dependent.

```
MPI_Init (&argc,&argv)
MPI_INIT (ierr)
```

MPI_Comm_size

Returns the total number of MPI processes in the specified communicator, such as MPI_COMM_WORLD. If the communicator is MPI_COMM_WORLD, then it represents the number of MPI tasks available to your application.

```
MPI_Comm_size (comm,&size)
MPI_COMM_SIZE (comm,size,ierr)
```

MPI_Comm_rank

Returns the rank of the calling MPI process within the specified communicator. Initially, each process will be assigned a unique integer rank between 0 and number of tasks - 1 within the communicator MPI_COMM_WORLD. This rank is often referred to as a task ID. If a process becomes associated with other communicators, it will have a unique rank within each of these as well.

```
MPI_Comm_rank (comm,&rank)
MPI_COMM_RANK (comm,rank,ierr)
```

MPI_Abort

Terminates all MPI processes associated with the communicator. In most MPI implementations it terminates ALL processes regardless of the communicator specified.

```
MPI_Abort (comm,errorcode)
MPI_ABORT (comm,errorcode,ierr)
```

Environment Management Routines

MPI_Get_processor_name

Returns the processor name. Also returns the length of the name. The buffer for "name" must be at least MPI_MAX_PROCESSOR_NAME characters in size. What is returned into "name" is implementation dependent - may not be the same as the output of the "hostname" or "host" shell commands.

```
MPI_Get_processor_name (&name,&resultlength)
MPI_GET_PROCESSOR_NAME (name,resultlength,ierr)
```

MPI_Get_version

Returns the version and subversion of the MPI standard that's implemented by the library.

```
MPI_Get_version (&version,&subversion)
MPI_GET_VERSION (version,subversion,ierr)
```

MPI_Initialized

Indicates whether MPI_Init has been called - returns flag as either logical true (1) or false(0). MPI requires that MPI_Init be called once and only once by each process. This may pose a problem for modules that want to use MPI and are prepared to call MPI_Init if necessary. MPI_Initialized solves this problem.

```
MPI_Initialized (&flag)
MPI_INITIALIZED (flag,ierr)
```

MPI_Wtime

Returns an elapsed wall clock time in seconds (double precision) on the calling processor.

```
MPI_Wtime ()
MPI_WTIME ()
```

MPI_Wtick

Returns the resolution in seconds (double precision) of MPI_Wtime.

```
MPI_Wtick ()
MPI_WTICK ()
```

MPI_Finalize

Terminates the MPI execution environment. This function should be the last MPI routine called in every MPI program - no other MPI routines may be called after it.

```
MPI_Finalize ()
MPI_FINALIZE (ierr)
```

Overview of sending and receiving with MPI

MPI's send and receive calls operate in the following manner:

First, process *A* decides a message needs to be sent to process *B*.
Process A then packs up all of its necessary data into a buffer for process B.
These buffers are often referred to as *envelopes* since the data is being packed into a single message before transmission.

After the data is packed into a buffer, the communication device (which is often a network) is responsible for routing the message to the proper location. The location of the message is defined by the process's rank.

Overview of sending and receiving with MPI

```
MPI_Send(
    void* data,
    int count,
    MPI_Datatype datatype,
    int destination,
    int tag,
    MPI_Comm communicator)
```

A

```
MPI_Recv(
    void* data,
    int count,
    MPI_Datatype datatype,
    int source,
    int tag,
    MPI_Comm communicator,
    MPI_Status* status)
```

B

MPI datatype	C equivalent
MPI_SHORT	short int
MPI_INT	int
MPI_LONG	long int
MPI_LONG_LONG	long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	char

MPI ping pong program

```
MPI_Send(
    void* data,
    int count,
    MPI_Datatype datatype,
    int destination,
    int tag,
    MPI_Comm communicator)
```

A

```
MPI_Recv(
    void* data,
    int count,
    MPI_Datatype datatype,
    int source,
    int tag,
    MPI_Comm communicator,
    MPI_Status* status)
```

B

```
int ping_pong_count = 0;
int partner_rank = (world_rank + 1) % 2;
while (ping_pong_count < PING_PONG_LIMIT) {
    if (world_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", world_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",
               world_rank, ping_pong_count, partner_rank);
    }
}
```

MPI send / recv program

A

```
MPI_Send(
    void* data,
    int count,
    MPI_Datatype datatype,
    int destination,
    int tag,
    MPI_Comm communicator)
```

B

```
MPI_Recv(
    void* data,
    int count,
    MPI_Datatype datatype,
    int source,
    int tag,
    MPI_Comm communicator,
    MPI_Status* status)
```

```
// Find out rank, size
int world_rank;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
int world_size;
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

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

MPI send / recv program

A

```
MPI_Send(
    void* data,
    int count,
    MPI_Datatype datatype,
    int destination,
    int tag,
    MPI_Comm communicator)
```

B

```
MPI_Recv(
    void* data,
    int count,
    MPI_Datatype datatype,
    int source,
    int tag,
    MPI_Comm communicator,
    MPI_Status* status)
```

```
// Find out rank, size
int world_rank;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
int world_size;
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

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

MPI Reduce and Allreduce

Reduce is a classic concept from functional programming.

Data reduction involves reducing a set of numbers into a smaller set of numbers via a function.

For example, let's say we have a list of numbers [1, 2, 3, 4, 5]. Reducing this list of numbers with the sum function would produce sum([1, 2, 3, 4, 5]) = 15.

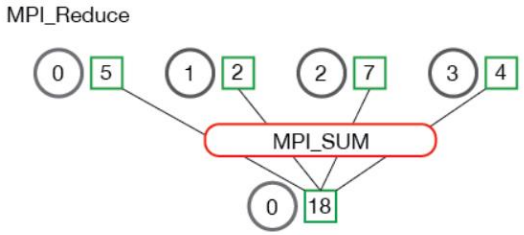
Similarly, the multiplication reduction would yield multiply([1, 2, 3, 4, 5]) = 120.

MPI_REDUCE

```
MPI_Reduce(
    void* send_data,
    void* recv_data,
    int count,
    MPI_Datatype datatype,
    MPI_Op op,
    int root,
    MPI_Comm communicator)
```

The send_data parameter is an array of elements of type datatype that each process wants to reduce. The recv_data is only relevant on the process with a rank of root. The recv_data array contains the reduced result and has a size of sizeof(datatype) * count.

The op parameter is the operation that you wish to apply to your data. MPI contains a set of common reduction operations that can be used.



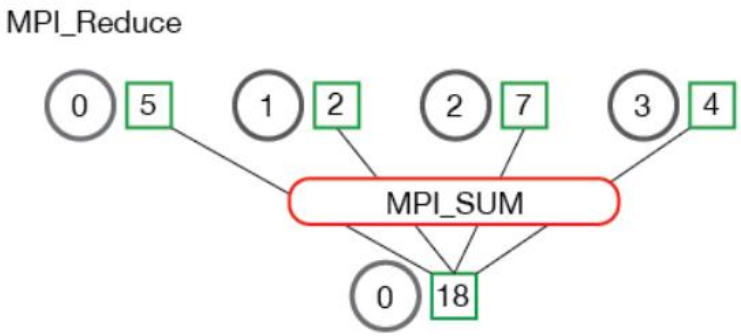
MPI Reduce and Allreduce

The reduction operations defined by MPI include:

MPI_REDUCE

- MPI_MAX - Returns the maximum element.
- MPI_MIN - Returns the minimum element.
- MPI_SUM - Sums the elements.
- MPI_PROD - Multiplies all elements.
- MPI_LAND - Performs a logical *and* across the elements.
- MPI_LOR - Performs a logical *or* across the elements.
- MPI_BAND - Performs a bitwise *and* across the bits of the elements.
- MPI_BOR - Performs a bitwise *or* across the bits of the elements.
- MPI_MAXLOC - Returns the maximum value and the rank of the process that owns it.
- MPI_MINLOC - Returns the minimum value and the rank of the process that owns it.

```
MPI_Reduce(
    void* send_data,
    void* recv_data,
    int count,
    MPI_Datatype datatype,
    MPI_Op op,
    int root,
    MPI_Comm communicator)
```



Computing average of numbers with MPI_Reduce

MPI_REDUCE

```
float *rand_nums = NULL;
rand_nums = create_rand_nums(num_elements_per_proc);

// Sum the numbers locally
float local_sum = 0;
int i;
for (i = 0; i < num_elements_per_proc; i++) {
  local_sum += rand_nums[i];
}

// Print the random numbers on each process
printf("Local sum for process %d - %f, avg = %f\n",
       world_rank, local_sum, local_sum / num_elements_per_proc);

// Reduce all of the local sums into the global sum
float global_sum;
MPI_Reduce(&local_sum, &global_sum, 1, MPI_FLOAT, MPI_SUM, 0,
           MPI_COMM_WORLD);

// Print the result
if (world_rank == 0) {
  printf("Total sum = %f, avg = %f\n", global_sum,
         global_sum / (world_size * num_elements_per_proc));
}
```

each process creates random numbers and makes a local_sum calculation.

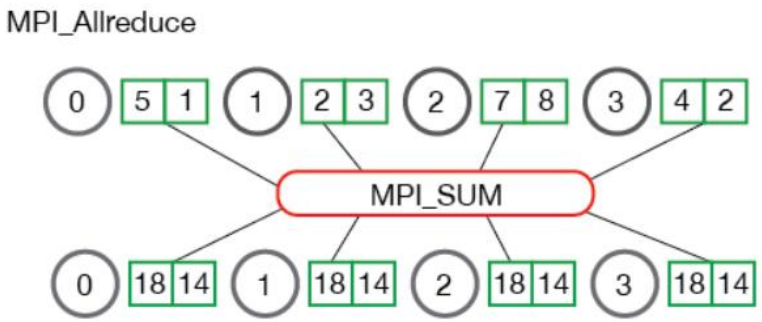
The local_sum is then reduced to the root process using MPI_SUM.

The global average is then global_sum / (world_size * num_elements_per_proc)

MPI_AllReduce

Many parallel applications will require accessing the reduced results across all processes rather than the root process. The function prototype is the following:

```
MPI_Allreduce(
    void* send_data,
    void* recv_data,
    int count,
    MPI_Datatype datatype,
    MPI_Op op,
    MPI_Comm communicator)
```



MPI_Allreduce is identical to MPI_Reduce with the exception that it does not need a root process id (since the results are distributed to all processes).

Examples (C, FORTRAN)

C Language

```c
// required MPI include file
#include "mpi.h"
#include <stdio.h>

int main(int argc, char *argv[]) {
int  numtasks, rank, len, rc;
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);


    // do some work with message passing


// done with MPI
MPI_Finalize();
}
```

FORTRAN Language

```fortran
program simple

! required MPI include file
include 'mpif.h'

integer numtasks, rank, len, ierr
character(MPI_MAX_PROCESSOR_NAME) hostname

! initialize MPI
call MPI_INIT(ierr)

! get number of tasks
call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)

! get my rank
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)

! this one is obvious
call MPI_GET_PROCESSOR_NAME(hostname, len, ierr)
print *, 'Number of tasks=',numtasks,' My rank=',rank,' Running on=',hostname


    ! do some work with message passing


! done with MPI
call MPI_FINALIZE(ierr)

end
```

Examples (C, FORTRAN)

C Language

```c
/******************************************************************************
* FILE: mpi_hello.c
* DESCRIPTION:
*   MPI tutorial example code: Simple hello world program


#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#define  MASTER         0

int main (int argc, char *argv[])
{
int   numtasks, taskid, len;
char hostname[MPI_MAX_PROCESSOR_NAME];

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
MPI_Get_processor_name(hostname, &len);
printf ("Hello from task %d on %s!\n", taskid, hostname);
if (taskid == MASTER)
   printf("MASTER: Number of MPI tasks is: %d\n",numtasks);
MPI_Finalize();

}
```

FORTRAN Language

```
C ******************************************************************************
C FILE: mpi_hello.f
C DESCRIPTION:
C   MPI tutorial example code: Simple hello world program


      program hello
      include 'mpif.h'
      parameter (MASTER = 0)

      integer numtasks, taskid, len, ierr
      character(MPI_MAX_PROCESSOR_NAME) hostname

      call MPI_INIT(ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD, taskid, ierr)

      call MPI_GET_PROCESSOR_NAME(hostname, len, ierr)
      write(*,20) taskid, hostname
      if (taskid .eq. MASTER) then
        write(*,30) numtasks
      end if

      call MPI_FINALIZE(ierr)

   20 format('Hello from task ',I2,' on ',A48)
   30 format('MASTER: Number of MPI tasks is: ',I2)

      end
```

Compile: Use a C or Fortran MPI compiler command. For example:

```
mpicc -w -o hello myhello.c
mpif77 -w -o hello myhello.f
mpif90 -w -o hello myhello.f
```