Synchronization

- Synchronization is the capability to control the access of multiple threads to any shared resource. synchronization is better option where we want to allow only one thread to access the shared resource.
- Managing the sequence of work and the tasks performing it is a critical design consideration for most parallel programs.
- Can be a significant factor in program performance (or lack of it)
- Often requires "serialization" of segments of the program.

Why use Synchronization?

The synchronization is mainly used to

1. To prevent thread interference.
2. To prevent consistency problem.

Synchronization Categories:
Barrier

- Usually implies that all tasks are involved
- Each task performs its work until it reaches the barrier. It then stops, or "blocks".
- When the last task reaches the barrier, all tasks are synchronized.
- What happens from here varies. Often, a serial section of work must be done. In other cases, the tasks are automatically released to continue their work.

Lock / semaphore

- Can involve any number of tasks
- Typically used to serialize (protect) access to global data or a section of code. Only one task at a time may use (own) the lock / semaphore / flag.
- The first task to acquire the lock "sets" it. This task can then safely (serially) access the protected data or code.
- Other tasks can attempt to acquire the lock but must wait until the task that owns the lock releases it.
- Can be blocking or non-blocking.

Types of Synchronization

There are two types of synchronization

1. Process Synchronization
2. Thread Synchronization

Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusive
    1. Synchronized method.
    2. Synchronized block.
    3. Static synchronization.
2. Cooperation (Inter-thread communication)

Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. It can be achieved by using the following three ways:

1. By Using Synchronized Method
2. By Using Synchronized Block
3. By Using Static Synchronization

Concept of Lock in Java

Synchronization is built around an internal entity known as the lock or monitor. Every object has a lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.

From Java 5 the package java.util.concurrent.locks contains several lock implementations.

What is parallel storage?

A parallel storage file system is a sort of clustered file system. A clustered file system is a storage system shared by multiple devices simultaneously.

The data is spread amongst several storage nodes for redundancy and performance in a parallel file system. In this system, the physical storage device is built using the storage devices of multiple servers. When the file system receives data, it distributes it across several storage nodes after breaking it into data blocks.

Parallel file storages duplicate the data on the physically distinct nodes. This lets the system be fault-tolerant and permits data redundancy. The data distribution improves the system's performance and makes it faster.

In other words, the parallel file system breaks data into blocks and distributes the blocks to multiple storage servers. It uses a global namespace to enable data access. The data is written/ read using different input/ output (I/O) paths. Power and energy consumption for these systems is highly required.

Message Passing Interface (MPI)

What is the message passing interface (MPI)?

The message passing interface (MPI) is a standardized means of exchanging messages between multiple computers running a parallel program across distributed memory.

In parallel computing, multiple computers – or even multiple processor cores within the same computer – are called nodes. Each node in the parallel arrangement typically works on a portion of the overall computing problem. The challenge then is to synchronize the actions of each parallel node, exchange data between nodes, and provide command and control over the entire parallel cluster.

The message passing interface defines a **standard suite of functions** for these tasks. The term *message passing* itself typically refers to the sending of a message to an object, parallel process, subroutine, function or thread, which is then used to start another process.

MPI isn't endorsed as an official standard by any standards organization, such as the Institute of Electrical and Electronics Engineers (IEEE) or the International Organization for Standardization (ISO), but it's generally considered to be the industry standard, and it forms the basis for most communication interfaces adopted by parallel computing programmers. Various implementations of MPI have been created by developers as well.

MPI defines useful syntax for routines and libraries in programming languages including Fortran, C, C++ and Java.

Benefits of the message passing interface

The message passing interface provides the following benefits:

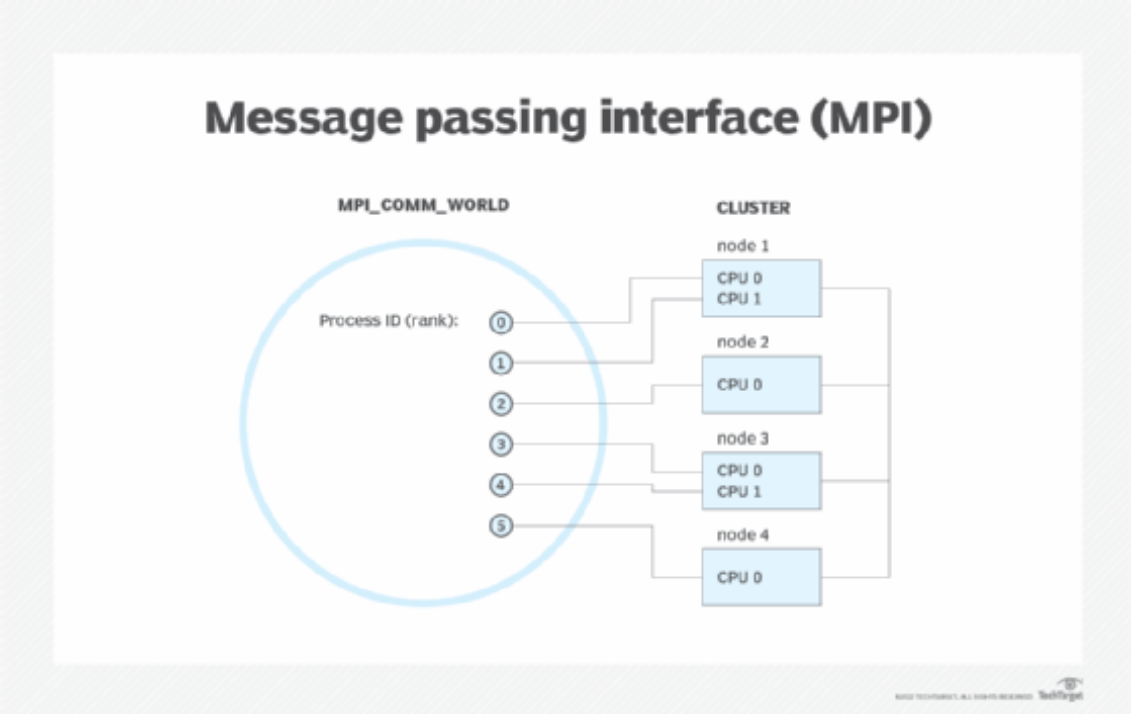
- **Standardization.** MPI has replaced other message passing libraries, becoming a generally accepted industry standard.
- **Developed by a broad committee.** Although MPI may not be an official standard, it's still a general standard created by a committee of vendors, implementers and users.
- **Portability.** MPI has been implemented for many distributed memory architectures, meaning users don't need to modify source code when porting applications over to different platforms that are supported by the MPI standard.
- **Speed.** Implementation is typically optimized for the hardware the MPI runs on. Vendor implementations may also be optimized for native hardware features.
- **Functionality.** MPI is designed for high performance on massively parallel machines and clusters. The basic MPI-1 implementation has more than 100 defined routines.

MPI terminology: Key concepts and commands

The following list includes some basic key MPI concepts and commands:

- **Comm.** These are communicator objects that connect groups of processes in MPI. Communicator commands give a contained process an independent identifier, arranging it as an ordered topology. For example, a command for a base communicator includes *MPI_COMM_WORLD*.
- **Color.** This assigns a color to a process, and all processes with the same color are located in the same communicator. A command related to color includes *MPE_Make_color_array*, which changes the available colors.
- **Key.** The rank or order of a process in the communicator is based on a key. If two processes are given the same key, the order is based on the process's rank in the communicator.
- **Newcomm.** This is a command for creating a new communicator. *MPI_COMM_DUP* is an example command to create a duplicate of a comm with the same fixed attributes.
- **Derived data types.** MPI functions need a specification to what type of data is sent between processes. *MPI_INT*, *MPI_CHAR* and *MPI_DOUBLE* aid in predefining the constants.

- **Point-to-point.** This sends a message between two specific processes. *MPI_Send* and *MPI_Recv* are two common blocking methods for point-to-point messages. Blocking refers to having the sending and receiving processes wait until a complete message has been correctly sent and received to send and complete a message.
- **Collective basics.** These are collective functions that need communication among all processes in a process group. *MPI_Bcast* is an example of such, which sends data from one node to all processes in a process group.
- **One-sided.** This term is typically used referring to a form of communications operations, including *MPI_Put*, *MPI_Get* and *MPI_Accumulate*. They refer specifically to being a writing to memory, reading from memory and reducing operation on the same memory across tasks.



An MPI COMM process containing multiple nodes in four clusters shows how a rank is given to each CPU.

History and versions of MPI

A small group of researchers in Austria began discussing the concept of a message passing interface in 1991. A Workshop on Standards for Message Passing in a Distributed Memory Environment, sponsored by the Center for Research on Parallel Computing, was held a year later in Williamsburg, Va. A working group was established to create the standardization process.

In November 1992, a draft for MPI-1 was created and in 1993 the standard was presented at the Supercomputing '93 conference. With additional feedback and changes, MPI version 1.0 was released in 1994. Since then, MPI has been open to all members of the high-performance computing community, including more than 40 participating organizations.

The older MPI 1.3 standard, dubbed MPI-1, provides over 115 functions. The later MPI 2.2 standard, or MPI-2, offers over 500 functions and is largely backward compatible with MPI-1. However, not all MPI libraries provide a full implementation of MPI-2. MPI-2 included new parallel I/O, dynamic process management as well as remote memory operations. The MPI-3 standard released in November 2014 improves scalability, enhances performance, includes multicore and cluster support and interoperates with more applications. In 2021, The MPI Forum released MPI 4.0. It introduced partitioned communications, a new tool interface, Persistent Collectives and other new additions. MPI 5.0 is currently under development.