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.

Benefits of the message passing interface

The message passing interface provides the following benefits:

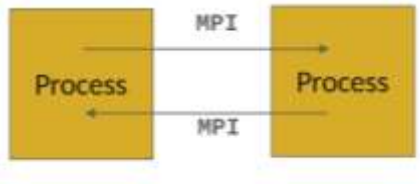
- **Standardization.** MPI has replaced other message passing libraries, becoming a generally accepted industry standard.
- **Developed by a broad committee.** Although MPI may not be an official standard, it's still a general standard created by a committee of vendors, implementors and users.
- **Portability.** 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.

The Message-Passing Model A process is (traditionally) a program counter and address space. Processes may have multiple threads (program counters and associated stacks) sharing a single address space. MPI is for communication among processes, which have separate address spaces.

Inter-process communication consists of...

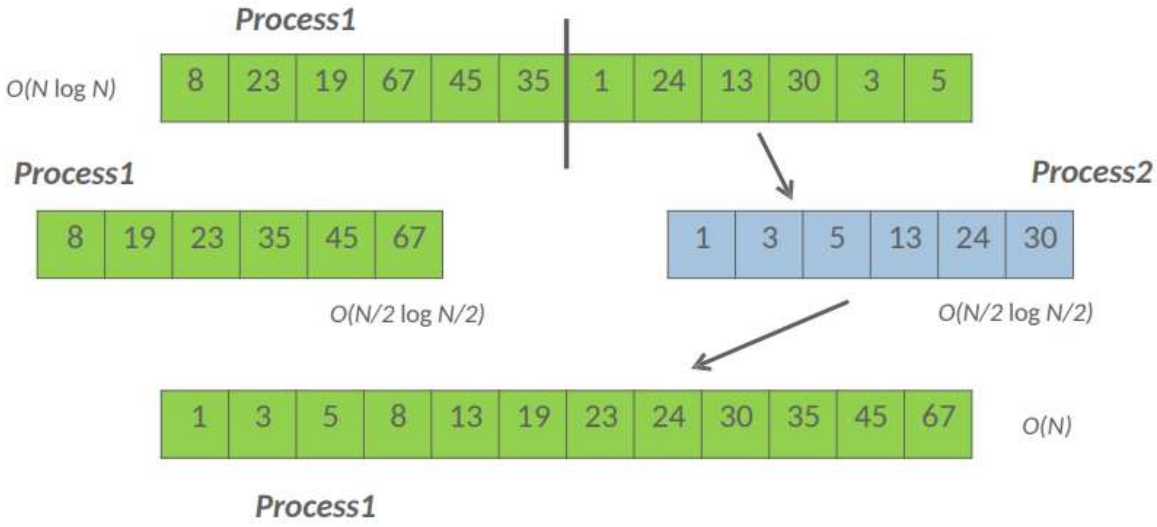
– Synchronization

– Movement of data from one process's address space to another's.



The Message-Passing Model (an example)

- Each process has to send/receive data to/from other processes
- Example: Sorting Integers



Simple MPI Program Identifying Processes

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

int main(int argc, char ** argv)
{
    int rank, size;

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("I am %d of %d\n", rank, size);

    MPI_Finalize();
    return 0;
}
```

Basic requirements for an MPI program

Here's a simple MPI (Message Passing Interface) program in C to print "Hello, world!" using multiple processes:

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char** argv)
 {
   int rank, size;
   MPI_Init(&argc, &argv); // Initialize MPI
   MPI_Comm_rank(MPI_COMM_WORLD, &rank); // Get the rank of the current process
   MPI_Comm_size(MPI_COMM_WORLD, &size); // Get the total number of processes
   printf("Hello world from process %d of %d\n", rank, size);
   MPI_Finalize(); // Finalize MPI
   return 0;
}
```

This code initializes MPI, gets the rank of each process, prints "Hello world" along with the process's rank, and then finalizes MPI.

To compile and run it, you'll need an MPI compiler.

For example, with mpicc:

```
mpicc hello_mpi.c -o hello_mpi
mpiexec -n 4 ./hello_mpi
```

This will run the program with 4 processes, each printing its rank and "Hello world!"

To Install MPI on Ubantu use following Command

```
sudo apt update
sudo apt install lam4-dev
sudo apt install mpich
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#define NUMBERS 100
 int main(int argc, char** argv) {
   int rank, size;
   int numbers[NUMBERS];
   int local_sum = 0, global_sum = 0;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   // Initialize random seed
   srand(rank);
   // Generate local random numbers
   for (int i = 0; i < NUMBERS; i++) {
      numbers[i] = rand() % 100;
      local_sum += numbers[i];
   }
   // Sum local sums across processes
   MPI_Reduce(&local_sum, &global_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
   // Print result from root process
   if (rank == 0) {
      printf("Sum of %d random numbers: %d\n", NUMBERS * size, global_sum);
   }
   MPI_Finalize();
   return 0;
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#define NUMBERS 100
int main(int argc, char** argv) {
   int rank, size;
   int numbers[NUMBERS];
   int local_min = INT_MAX, local_max = INT_MIN;
   int global_min, global_max;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   // Initialize random seed
   srand(rank);
   // Generate local random numbers
   for (int i = 0; i < NUMBERS; i++) {
      numbers[i] = rand() % 100;
      if (numbers[i] < local_min) {
         local_min = numbers[i];
      }
      if (numbers[i] > local_max) {
         local_max = numbers[i];
      }
   }
   // Find local minimum and maximum
   MPI_Allreduce(&local_min, &global_min, 1, MPI_INT, MPI_MIN, MPI_COMM_WORLD);
   MPI_Allreduce(&local_max, &global_max, 1, MPI_INT, MPI_MAX, MPI_COMM_WORLD);
   // Print result from all processes
   printf("Process %d: Local min = %d, Local max = %d\n", rank, local_min, local_max);

   // Print global result from root process
   if (rank == 0) {
      printf("Global min = %d, Global max = %d\n", global_min, global_max);
   }
   MPI_Finalize();
   return 0;   }
```