Message Passing Interface

MPI

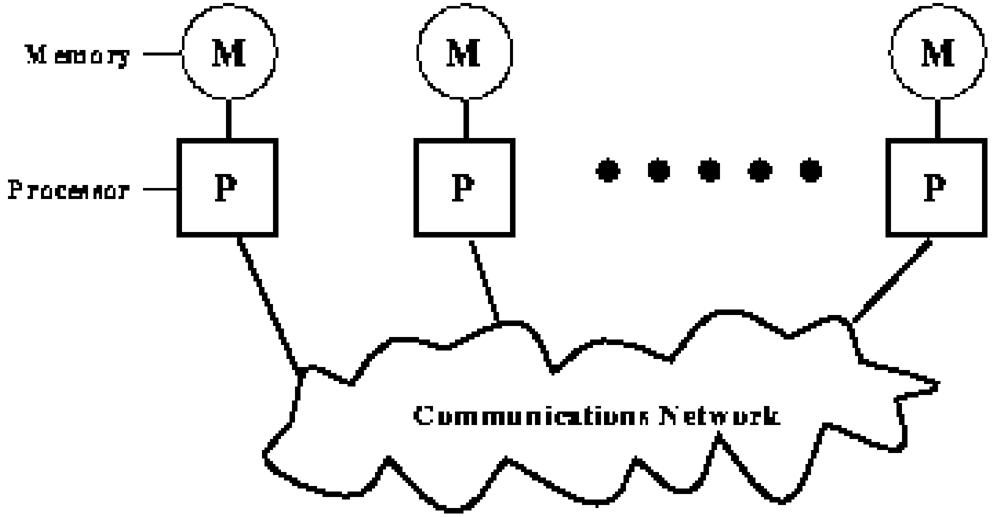
Message Passing Interface - MPI

What is the message passing model?

An application passes messages among processes in order to perform a task.

This model works out quite well in practice for parallel applications

Message Passing Programming Paradigm



Message Passing Interface - MPI

- The goal of the Message Passing Interface is
  - To establish a portable, efficient, and flexible standard for message passing
  - To provide source-code portability
  - To allow efficient implementation across a range of architectures

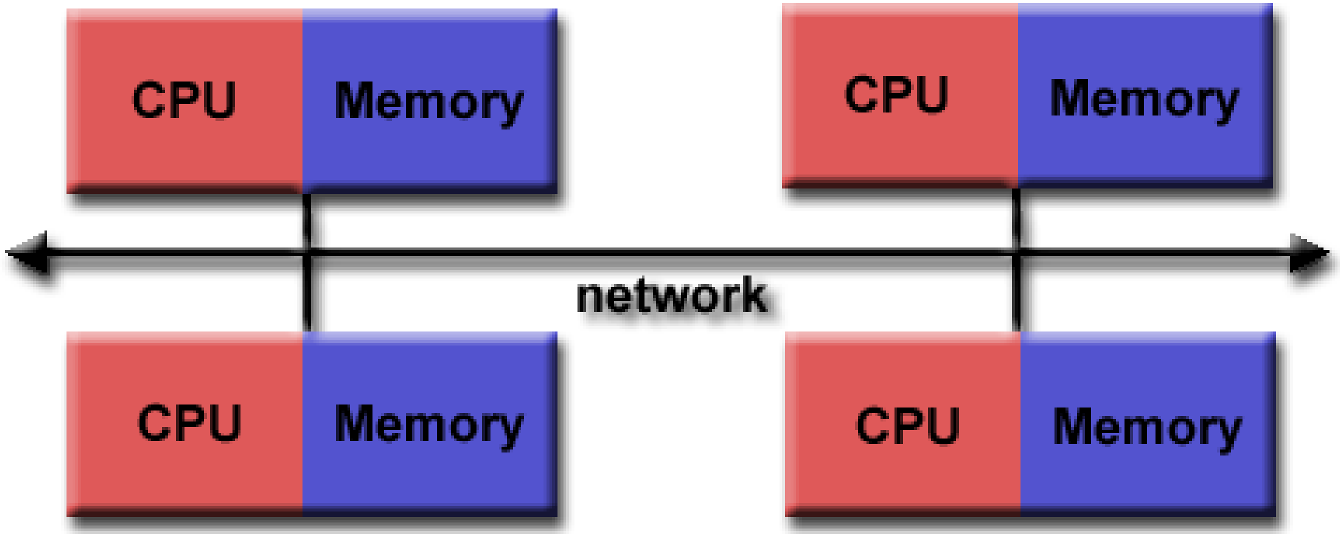
- It is a message passing library standard based on the consensus of the MPI Forum
- It becomes the "industry standard" for writing message passing programs on HPC platforms
- It addresses the ***message-passing parallel programming model:*** data is moved from the address space of one process to that of another process
- An MPI message is an array of elements of a particular MPI datatype.

History

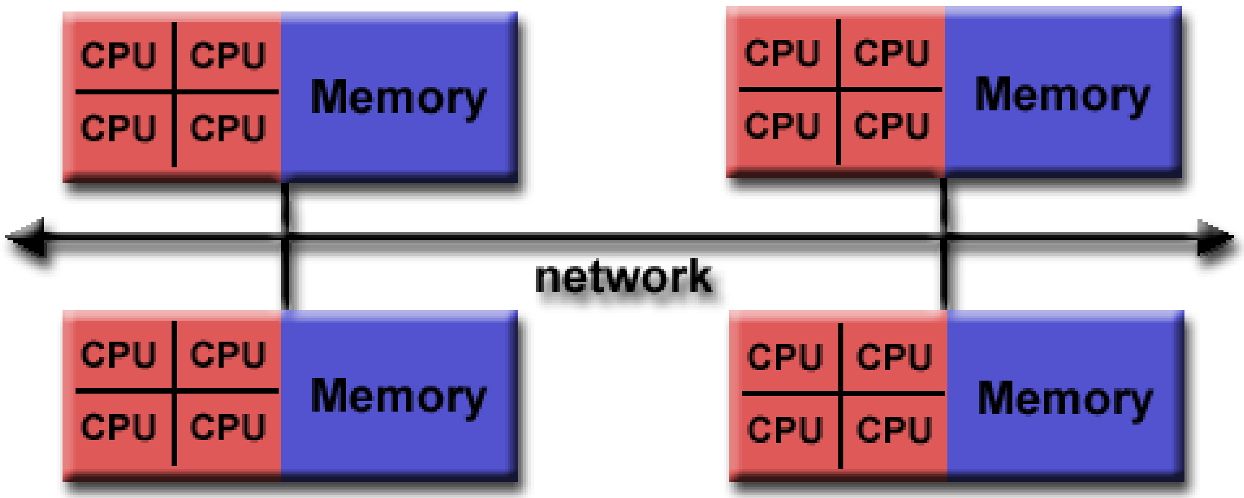
- **1982-1992** – Parallel computing developers uses no. of incompatible tools for writing programs
- **April 1992** – Workshop on standards for message passing - features essential to a standard message passing interface were discussed
- **Nov 1992:** Working group meets in Minneapolis with MPI draft proposal
- **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), MPI-2.2 (Sep 2009)
- **Sep 2012:** The MPI-3.0 standard was approved.
  - MPI-3.1 (Jun 2015)
- **Current:** The MPI-4.0 standard is under development

Programming Model:

- Originally, MPI was designed for distributed memory architectures (1980 – 1990’s)



- 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.



Today, MPI runs on virtually any hardware platform:

- Distributed Memory, Shared Memory, Hybrid

Reasons for Using MPI

- **Standardization**
- **Portability**
- **Performance**
- **Functionality**
- **Availability**

MPI's design for the message passing model

Communicator - A communicator defines a group of processes that have the ability to communicate with one another.

- In this group of processes, each process is assigned a unique *rank*, and they explicitly communicate with one another by their ranks.
- A process may send a message to another process by providing the rank of the process and a unique *tag* to identify the message.
- The receiver can then post a receive for a message with a given tag and then handle the data accordingly.

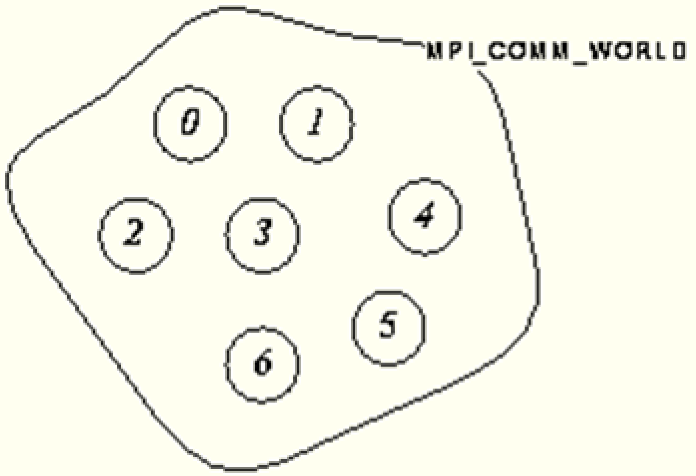


Figure : The predefined communicator MPI_COMM_WORLD for seven processes. The numbers indicate the ranks of each process.

MPI’s design for the message passing model

- Communications such as this which involve one sender and receiver are known as ***point-to-point*** **communications**.

- ***Collective*** **communications -** There are many cases where processes may need to communicate with everyone

- For example, when a master process needs to broadcast information to all of its worker processes

- Mixtures of point-to-point and collective communications can be used to create highly complex parallel programs.

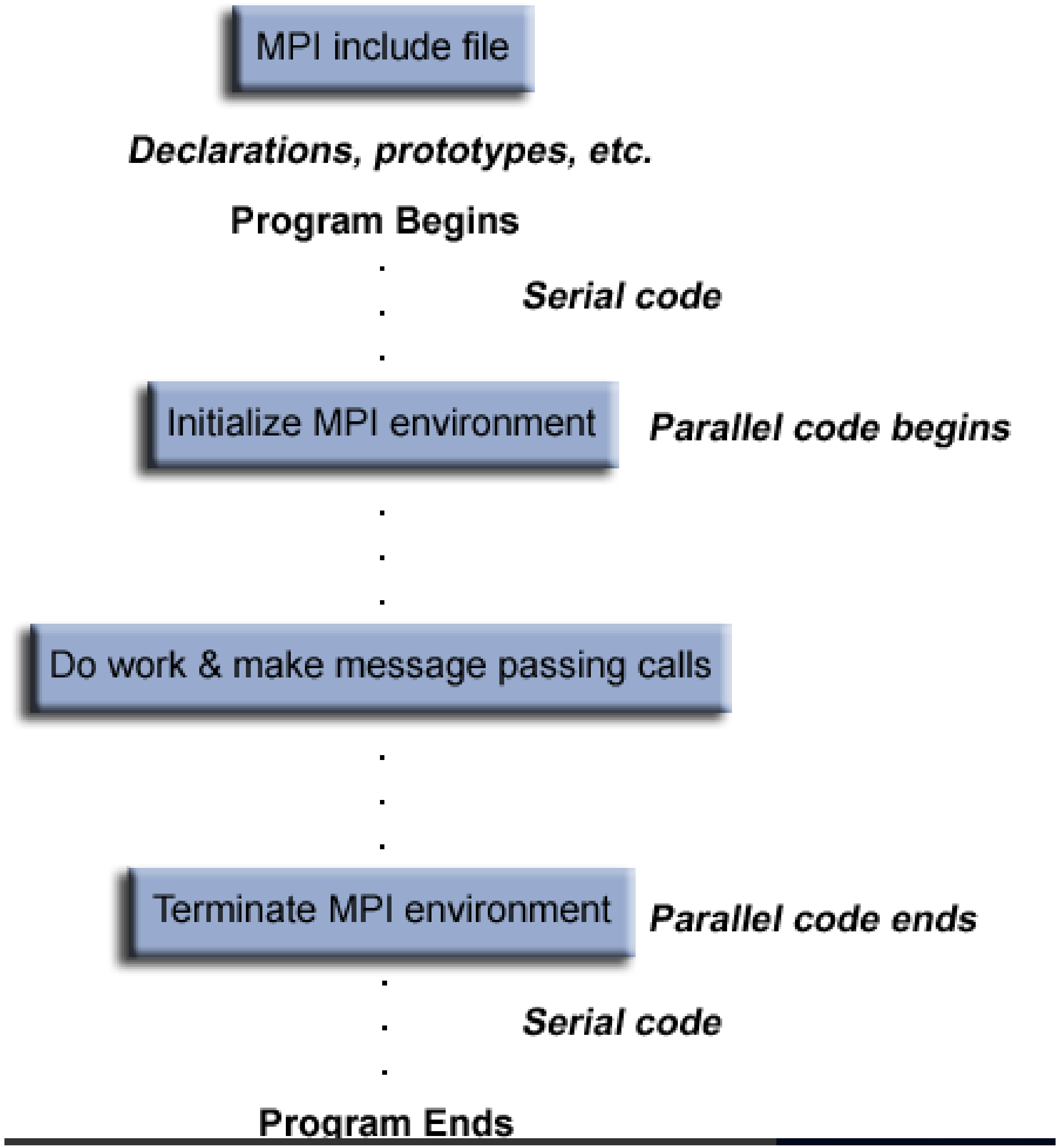
MPI Library

- 125 functions

Full list available @ https://www.mpich.org/static/docs/latest/

- 6 functions (commonly used)
  - MPI_Init
  - MPI_Finalize
  - MPI_Comm_size
  - MPI_Comm_rank
  - MPI_Send
  - MPI_Recv

MPI Program Structure



Example Program 1

```
#include “mpi.h”
#include <stdio.h>
int main(int argc, char *argv[])
{
  MPI_Init(&argc, &argv);

printf(“Hello, world!\n”);

  MPI_Finalize();
  return 0;
}
```

Example Program 2

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

int main(int argc, char** argv) {
    // Initialize the MPI environment
    MPI_Init(NULL, NULL);

    // Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
```

Example Program

```
    // Get the name of the processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    // Print off a hello world message
    printf("Hello world from processor %s, rank %d out of %d processor
s\n",
           processor_name, world_rank, world_size);

    // Finalize the MPI environment.
    MPI_Finalize();
}
```

Program execution (UNIX)

Using MPICH Compiler

To create a file, gedit filename.c

To compile, mpicc filename.c –o name

To run, mpirun –np 4 ./name