MPI+Parallel Computing An Introduction

Rushikesh K. Joshi
Computer Science & Engineering
IIT Bombay

Purpose

- Parallel and Distributed Computing
- Inter-Process Communication/IPC Communication Protocol
- Abstractions for communication
- Standardized/Portable way through libraries,
- Multi-language implementation:
  - C, C++, Fortran as well in early days

Message Passing Interface

- Point to point communication between processes
- Collective communication, synchronization over a group of processes
- Process Topologies – such as making grids
- Communicators: A namespace for processes
- Runtime profiling/performance measurement interface
- Environment related functions, error handling etc.

The Evolution of MPI

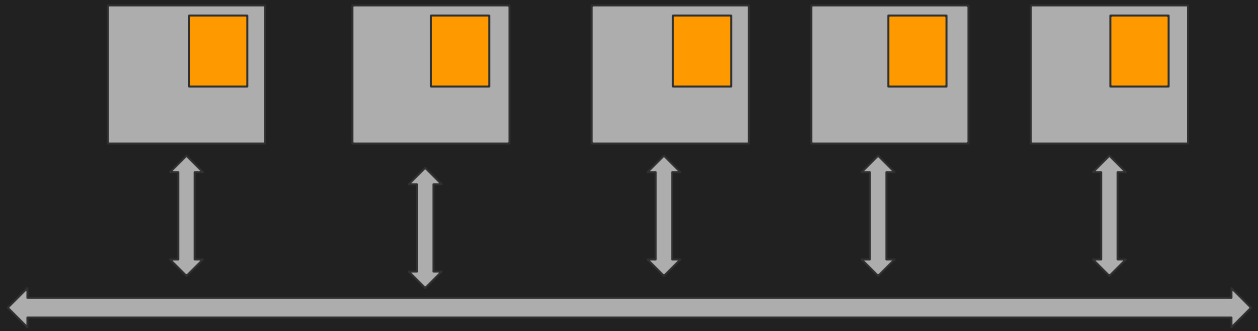
1991	First proposal	Beginning of MPI Ideas (PVM- 1989 was prior to MPI)
1994	MPI 1 standard	no shared memory
1996	MPI 2 standard	limited distributed ShM
2008	MPI 1.3 (now called MPI 1)	128 functions supported, message passing, static runtime
2009	MPI 2.2 (now called MPI 2)	500+ functions, parallel I/O, dynamic process management, remote memory; C,C++,Fortran 90 bindings
2012	MPI 3 standard	explicit shared memory
2015	MPI 3.1 (now called MPI 3)	non-blocking extensions to collective operations, one sided operations; adds Fortran 2008 bindings
2021, 2024	MPI 4, MPI 4.1	more refined features

The Original MPI Concepts

Processes

- A process is an independent execution unit
- MIMD style processes
- Typically, each process has its own address space; (SHM implementations of MPI are possible)
- Processes communicate via MPI communication primitives
- A process can be sequential, or internally multi-threaded, MPI does not assume anything, and MPI is meant to be Thread-SAFE.
  - It means, even if there are multiple threads, it won't change the meaning.
- Blocking MPI call blocks only the invoking thread.
- Each process is labeled unique RANK. It can be discovered by a call MPI_COMM_RANK().
  - If there are 4 processes, ranks can be 0..3.
    - int my_rank;
    - MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

Distributed Memory Architectures/workstations
Each Process has its own memory, they are connected via network; Each process has a different rank



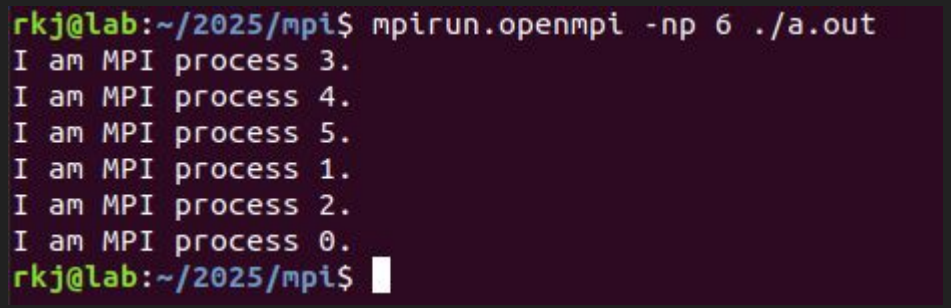
MPIRUN

mpirun executes 4 processes by default; it runs its internal scheduler

```
rkj@lab:~/2025/mpi$ mpiCC 0msg.cpp
rkj@lab:~/2025/mpi$ mpiCC 1getrank.cpp
rkj@lab:~/2025/mpi$ mpirun.openmpi ./a.out
I am MPI process 2.
I am MPI process 0.
I am MPI process 3.
I am MPI process 1.
rkj@lab:~/2025/mpi$
```

MPIRUN

mpirun executes n copies given the argument to it



The test program

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
int main(int argc, char* argv[])
{
    MPI_Init(&argc, &argv);
    int my_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    printf("I am MPI process %d.\n", my_rank);
    MPI_Finalize();
    return EXIT_SUCCESS;
}
```

SPMD Idea

Each program does something different!

```
rkj@lab:~/2025/mpi$ mpirun.openmpi ./a.out
my rank:3
15 16 17 18 19
my rank:0
0 1 2 3 4
my rank:2
10 11 12 13 14
my rank:1
5 6 7 8 9
rkj@lab:~/2025/mpi$
```

```
#include <iostream>
#include <mpi.h>
using namespace std;
int main(int argc, char* argv[]){
int my_rank;
int A[20];
      MPI_Init(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
      cout << "my rank:" << my_rank << endl;
      for (int i=0; i<5;i++) A[i]=i+my_rank*5;
      for (int i=0; i<5;i++) cout << A[i] << " "; cout << endl;
      MPI_Finalize();
      return EXIT_SUCCESS;
}
```

Knowing the number of Processes

```cpp
#include <iostream>
#include <mpi.h>
using namespace std;
int main(int argc, char* argv[])
{
int my_rank;
int A[20];
int N;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &N);
    cout << N << " Processes running in the systen\n";
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    cout << "my rank:" << my_rank << endl;
    for (int i=0; i<5;i++) A[i]=i+my_rank*5;
    for (int i=0; i<5;i++) cout << A[i] << " "; cout << endl;
    MPI_Finalize();
    return EXIT_SUCCESS;
}
```

```
rkj@lab:~/2025/mpi$ mpiCC 3mpicommsize.cpp
rkj@lab:~/2025/mpi$ mpirun.openmpi -np 5 ./a.out
5 Processes running in the systen
my rank:0
0 1 2 3 4
5 Processes running in the systen
my rank:2
10 11 12 13 14
5 Processes running in the systen
my rank:3
15 16 17 18 19
5 Processes running in the systen
5 Processes running in the systen
my rank:4
20 21 22 23 24
my rank:1
5 6 7 8 9
rkj@lab:~/2025/mpi$
```

```
rkj@lab:~/2025/mpi$ mpirun.openmpi -np 2 ./a.out
2 Processes running in the systen
my rank:1
10 11 12 13 14 15 16 17 18 19
2 Processes running in the systen
my rank:0
0 1 2 3 4 5 6 7 8 9
rkj@lab:~/2025/mpi$ mpirun.openmpi -np 1 ./a.out
1 Processes running in the systen
my rank:0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
rkj@lab:~/2025/mpi$ mpirun.openmpi -np 4 ./a.out
4 Processes running in the systen
my rank:0
0 1 2 3 4
4 Processes running in the systen
my rank:1
5 6 7 8 9
4 Processes running in the systen
my rank:2
4 Processes running in the systen
my rank:3
15 16 17 18 19
10 11 12 13 14
rkj@lab:~/2025/mpi$ mpirun.openmpi -np 5 ./a.out
5 Processes running in the systen
my rank:0
0 1 2 3
5 Processes running in the systen
my rank:2
8 9 10 11
5 Processes running in the systen
my rank:3
12 13 14 15
5 Processes running in the systen
my rank:1
4 5 6 7
5 Processes running in the systen
my rank:4
16 17 18 19
```

```
rkj@lab:~/2025/mpi$ mpirun.openmpi -np 5 ./a.out
5 Processes running in the systen
my rank:0
0 1 2 3
5 Processes running in the systen
my rank:2
8 9 10 11
5 Processes running in the systen
my rank:3
12 13 14 15
5 Processes running in the systen
my rank:1
4 5 6 7
5 Processes running in the systen
my rank:4
16 17 18 19
rkj@lab:~/2025/mpi$ mpirun.openmpi -np 10 ./a.out
10 Processes running in the systen
my rank:5
10 11
10 Processes running in the systen
my rank:0
0 1
10 Processes running in the systen
my rank:2
10 Processes running in the systen
my rank:3
6 7
10 Processes running in the systen
my rank:4
8 9
10 Processes running in the systen
my rank:8
16 17
10 Processes running in the systen
my rank:9
18 19
10 Processes running in the systen
my rank:7
14 15
4 5
10 Processes running in the systen
my rank:6
12 13
10 Processes running in the systen
my rank:1
2 3
```

Dynamic Work Sharing (on one's own)

```
#include <iostream>
#include <mpi.h>
using namespace std;
int main(int argc, char* argv[])
{
int my_rank;
int A[20];
int N;
int W;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &N);
    cout << N << " Processes running in the systen\n";
    W = 20/N; // work size
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    cout << "my rank:" << my_rank << endl;
    for (int i=0; i<W;i++) A[i]=i+my_rank*20/N;
    for (int i=0; i<W;i++) cout << A[i] << " "; cout << endl;
    MPI_Finalize();
    return EXIT_SUCCESS;
}
```

Rank, and N are used to divide work by oneself

MPI_COMM_WORLD represents the global communicator which is the system of all processes.

One can define new communicators to partition the system

```
#include <iostream>
#include <mpi.h>
using namespace std;
int main(int argc, char* argv[]){
int my_rank;
int N, S;
int tag=99;
MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    if (my_rank==0) {
        N=100;
        MPI_Send(&N,1,MPI_INT,1,tag,MPI_COMM_WORLD);
        MPI_Recv(&S,1,MPI_INT,1,tag,MPI_COMM_WORLD,&status);
        cout << S << endl;
    }
    else {
        MPI_Recv(&N,1,MPI_INT,0,tag,MPI_COMM_WORLD,&status);
        S = N*N;
        MPI_Send(&S,1,MPI_INT,0,tag,MPI_COMM_WORLD);
    }
    MPI_Finalize();
    return EXIT_SUCCESS;
```

C++ ▾ Tab Width: 8

Send and Receive to Communicate Work and Receive Results

Send to Communicate Work and Results

IN	buf	initial address of send buffer
IN	count	number of entries to send
IN	datatype	datatype of each entry
IN	dest	rank of destination
IN	tag	message tag
IN	comm	communicator

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

Receive to Receive Work and Results

OUT	buf	initial address of receive buffer
IN	count	max number of entries to receive
IN	datatype	datatype of each entry
IN	source	rank of source
IN	tag	message tag
IN	comm	communicator
OUT	status	return status

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

Typical MPI Standard Data Types defined (Portable)

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

```cpp
using namespace std;
int main(int argc, char* argv[]){
int my_rank;
int N, S;
int tag=99;
MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    if (my_rank==0) {
        N=100;
        MPI_Send(&N,1,MPI_INT,1,tag,MPI_COMM_WORLD);
        MPI_Recv(&S,1,MPI_INT,1,tag,MPI_COMM_WORLD,&status);
        cout << S << endl;
    }
    else {
        MPI_Recv(&N,1,MPI_INT,MPI_ANY_SOURCE,tag,MPI_COMM_WORLD,&status);
        S = N*N;
        MPI_Send(&S,1,MPI_INT,status.MPI_SOURCE,status.MPI_TAG,MPI_COMM_WORLD);
    }
    MPI_Finalize();
    return EXIT_SUCCESS;
}
```

The ANY Source: Receive from any source

```
rkj@lab:~/2025/mpi/sndrcv$ mpiCC 6anyrcv.cpp
rkj@lab:~/2025/mpi/sndrcv$ mpirun.openmpi -np 2 ./a.out
10000
rkj@lab:~/2025/mpi/sndrcv$
```

Broadcast

4.5.1 Example Using MPI_BCAST

Example 4.1 Broadcast 100 ints from process 0 to every process in the group.

```
MPI_Comm comm;
int array[100];
int root=0;
...
MPI_Bcast( array, 100, MPI_INT, root, comm);
```

Some More Primitives

Non-blocking operation-non blocking send, receive (ireceive, isend)

Scatter – broadcast sends same value, scatter cuts the array sends chunks

Gather – takes elements from many processes and gathers them into one

wait - wait for non-blocking calls to complete– if they are being used

Parallel Computing Performance

Speedup – how fast it computes

Utilization – how much of idle cycles

Anonymous Remote Computing: A Paradigm for Parallel Programming on Interconnected Workstations

Rushikesh K. Joshi and D. Janaki Ram, *Member*, *IEEE*

Abstract—Parallel computing on interconnected workstations is becoming a viable and attractive proposition due to the rapid growth in speeds of interconnection networks and processors. In the case of workstation clusters, there is always a considerable amount of unused computing capacity available in the network. However, heterogeneity in architectures and operating systems, load variations on machines, variations in machine availability, and failure susceptibility of networks and workstations complicate the situation for the programmer. In this context, new programming paradigms that reduce the burden involved in programming for distribution, load adaptability, heterogeneity, and fault tolerance gain importance. This paper identifies the issues involved in parallel computing on a network of workstations. The Anonymous Remote Computing (ARC) paradigm is proposed to address the issues specific to parallel programming on workstation systems. ARC differs from the conventional communicating process model by treating a program as one single entity consisting of several loosely coupled remote instruction blocks instead of treating it as a collection of processes. The ARC approach results in distribution transparency and heterogeneity transparency. At the same time, it provides fault tolerance and load adaptability to parallel programs on workstations. ARC is developed in a two-tiered architecture consisting of high level language constructs and low level ARC primitives. The paper describes an implementation of the ARC kernel supporting ARC primitives.

Index Terms—Anonymous remote computing, cluster computing, remote instruction block, parallel programming.

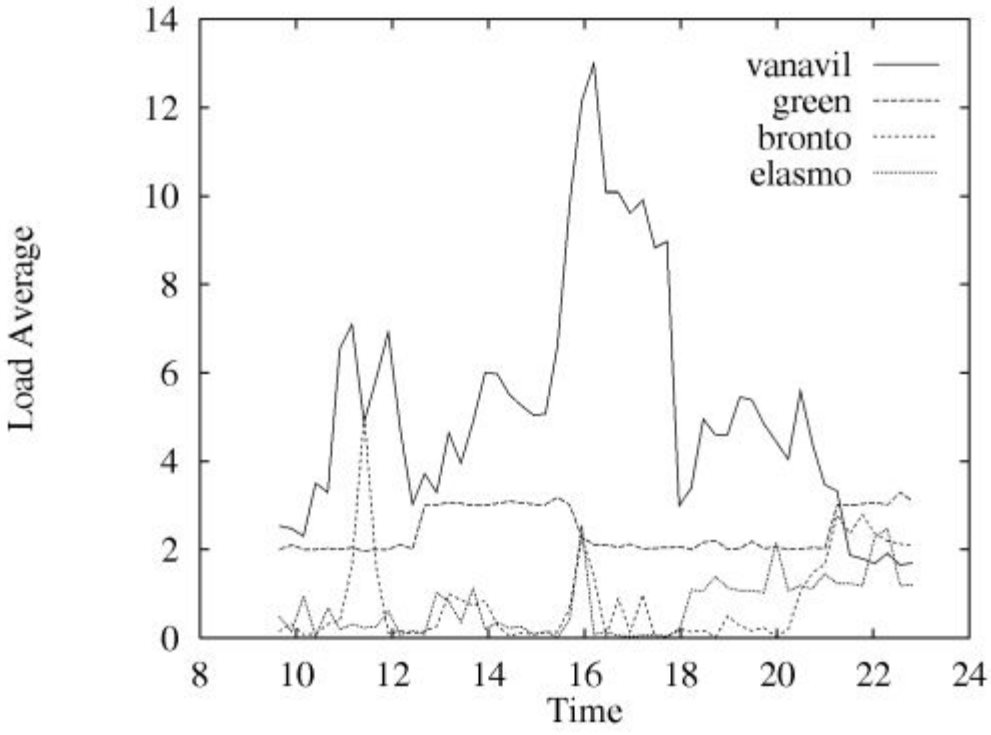


Fig. 1. Load variation on four machines in an academic environment.

TABLE 3
THE HETEROGENEITY LOAD TEST, 16 PROCESSORS

Task: A dynamically generated tree of total 3100 nodes with 2000 iterations per node					
Machine	No. of Hops Taken	Average HPF per Hop	No. of Tree Nodes Processed	Compilation Time (sec)	Processing Time (sec)
IBM RS/6000	33	9209	831	1.31	514.89
IBM RS/6000	18	5088	441	waiting time	592.65
IBM RS/6000	19	5040	441	waiting time	581.80
IBM RS/6000	18	4916	440	waiting time	584.21
Sun Sparc	16	1127	286	20.55	571.44
Sun Sparc	14	959	220	waiting time	563.60
Sun 3/60	40	54	70	nil	540.92
Sun 3/60	39	53	67	nil	546.70
Sun 3/50	35	36	45	nil	574.08
Sun 3/50	34	36	44	nil	556.98
Sun 3/50	36	36	47	nil	565.92
Sun 3/50	30	35	39	nil	566.92
Sun 3/50	29	32	37	nil	557.10
Sun 3/50	30	32	38	nil	562.56
Sun 3/50	29	31	35	nil	554.16
Sun 3/50	17	24	19	nil	580.66
Total no. of locks obtained (total no. of hops)					437
Time spent in obtaining locks					135.24
Total Processing Time					661.84
HP Utilization					85.34%
Task Imbalance					0.15%

TABLE 4
LOAD ADAPTABILITY TEST

No. of Processors	Machine Architectures	Tree Size in No. of Nodes	No. of Iterations per Tree Node	Average HPF per Hop	Load Imbalance (%)	HPU (%)
4	1 IBM RS/6000, 2 Sun Sparcs 1 Sun 3/60	465	1500	1208	1.13	85.85
8	4 IBM RS/6000s, 2 Sun Sparcs, 2 Sun 3/60s	1550	2000	5429	0.43	86.40
12	4 IBM RS/6000s, 2 Sun Sparcs, 2 Sun 3/60s, 4 Sun 3/50s	2480	2000	2216	0.31	84.65
16	4 IBM RS/6000s, 2 Sun Sparcs, 2 Sun 3/60s, 8 Sun 3/50s	3100	2000	1427	0.15	85.34

TABLE 6
NO LOAD PARALLELISM TEST

Task size	Nodes	Speedup for PVM	Speedup for ARC
50 × 50	2	1.81	1.46
100 × 100	2	1.97	1.75
	4	3.72	3.2
	5	4.30	3.8
150 × 150	3	2.96	2.63
	6	5.55	4.75
200 × 200	4	3.84	3.48
	5	4.80	3.48
	8	7.20	6.24

TABLE 7
NO LOAD INLINING TEST

Task Size	Pure Sequential (sec)	ARC on Same Node by Inlining (sec)	Slowdown (%)
25 × 25	2.00	2.06	3.00
50 × 50	16.08	16.28	1.25
75 × 75	54.88	55.18	0.55
100 × 100	129.44	130.4	0.75

Prior Work at C-DAC (1992)

Parallel IDA* on 4-32 Processors, PARAM Supercomputer

Ring Architecture

Dynamic Load Balancing

The Same program executes on all nodes, but works on different workloads

References

MPI papers, docs

ARC papers