Parallel Programming with Message Passing Interface

Mr. Nirav K. Shah
Assistant Professor
Department of Computer Science & Engineering
Shree Swaminarayan College of Computer Science, Sardarnagar, Bhavnagar, Affiliated to M. K. Bhavnagar University, India

Abstract— The goal of the Message Passing Interface is to establish a portable, efficient, and flexible standard for message passing that will be widely used for writing message passing programs. This article proposed MPI for MIMD distributed memory concurrent computers. MPI includes point-to-point and collective communication routine.

Key words: Message Passing, Distributed Computing, Parallel Computing, Communication Protocol, Secure Communication

I. INTRODUCTION

MPI is a language independent communication protocol used to program parallel computers. Both point-to-point and collective communication are supported. MPI is a message-passing application programmer interface, together with protocol and semantic specification for how its features must behave in any implementation. "MPI's goals are high performance, scalability and portability. MPI remains the dominant model used in high-performance computing today.

MPI is not sanctioned by any major standards body; nevertheless, it has become a de facto standard for communication among processes that model a parallel program running on a distributed memory system. Actual distributed memory super computers such as computer clusters often run such programs. The principal MPI-1 model has not shared memory concept, and MPI-2 has only a limited distributed share memory concept. None-theless., MPI programs are regularly run on shared memory computers. Designing program around the MPI model has advantages over NUMA architectures since MPI encourages memory locality.

Although MPI belongs in layers 5 and higher of the OSI reference model, implementations may cover most layers, with sockets and TCP used in the transport layer.

Most MPI implementations consist of a specific set of routines (i.e. API) directly callable from FORTRAN, c and C++ and from any language capable of interfacing with such libraries. The advantages of MPI over older message passing libraries are portability and speed.

MPI uses language independent specification for calls and language bindings. The first MPI standard specified ANSI C and Fortran 77 bindings together with the LIS.

MPI is often compared with PVM, which is a popular distributed environment and message passing system developed in 1989, and which was one of the systems that motivated the need for standard parallel message passing. Threaded shared memory programming models and message passing programming can be considered as complementary programming approaches, and can occasionally be seen together in application, e.g. in servers with multiple large shared memory nodes.

II. FUNCTIONALITY

The MPI interface is meant to provide essential virtual topology, synchronization and communication functionality between a set of professes in a language independent way, with language – specific syntax, plus a few language specific features, MPI programs always work with processes as processors. Typically, for maximum performance, each CPU will be assigned just a single process. This assignment happens at runtime though the agent that starts the MPI program, normally called mpirun or mpiexec.

MPI library functions include, but are not limited to point-to-point rendezvous-type send/receive operations, choosing between a Cartesian or graph like logical process topology, exchanging data between process pairs, combining partial results of computations as well as obtaining network-related information such as the number of processes in the computing session, current processor identity that a process accessible in a logical topology, and so on. Point-to-point operations come in synchronous, asynchronous, buffered and ready forms to allow both relatively stronger and weaker semantics for the synchronization aspects of a rendezvous-send. Many outstanding operations are possible in asynchronous mode, in most implementations.

MPI-1 and MPI-2 both enable implementations that overlap communication and computation, but practice and theory differ. MPI also specifies thread safe interfaces, which have cohesion and coupling strategies that help avoid hidden state within the interface. It is relatively easy to write multithreaded point-to-point MPI code. Multithreaded collective communication is best accomplished with multiple copies of communicators.

III. PROGRAMMING MODEL

MPI was designed for distributed memory architectures, which were becoming increasingly popular at that time during 1980 to 1990's

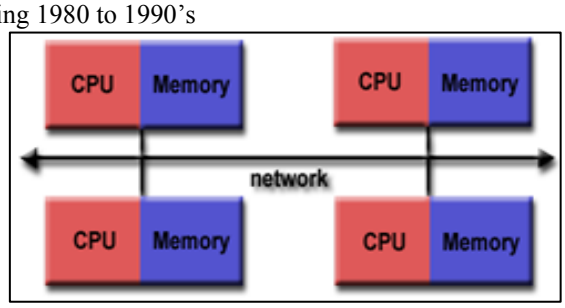

Fig. 1:

As architecture trends changed, shared memory SMPs were combined over networks creating hybrid distributed memory / shared memory systems. MPI implementers adapted their libraries to handle both types of underlying memory architectures seamlessly. They also

adapted/developed ways of handling different interconnect and protocols.

Currently MPI runs on virtually any hardware platform like distributed memory, shared memory and hybrid. The programming model clearly remains a distributed memory model however, regardless of the underlying physical architecture of the machine. All parallelism is explicit: the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs. MPI runs on virtually any hardware platform like distributed memory, shared memory and hybrid.

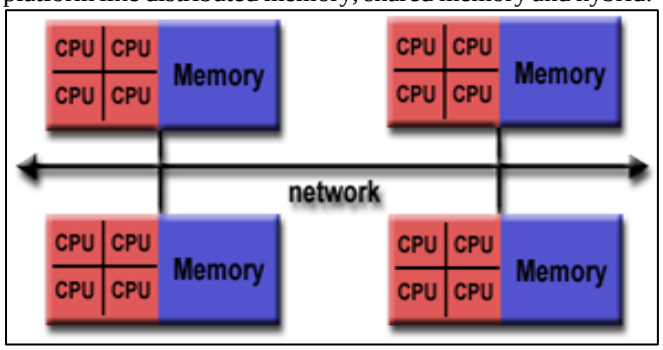

Fig. 2:

IV. MPI PROGRAM STRUCTURE

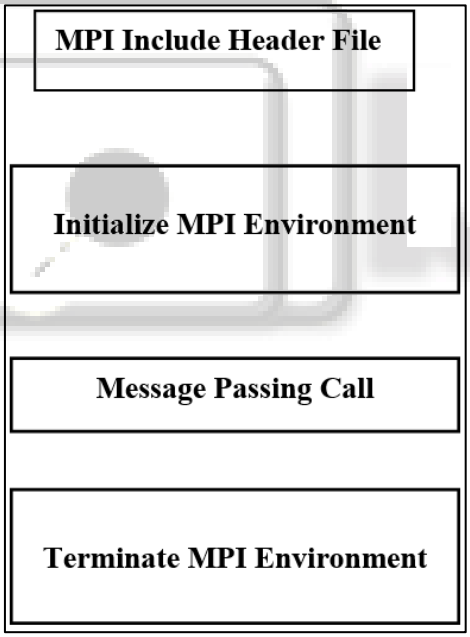

Fig. 3:

V. MPI CALL

1) MPI_init(int *argc,char *argv)
– Initialize the MPI execution environment.
– This call is required in every MPI program and must be the first MPI call. It establishes the environment. Only one invocation of MPI_init can occur in each program execution. It takes the command line argument as parameters.
2) MPI_Comm_rank(MPI_Comm comm., int rank)
– Determine the rank of calling process in the communicator.
– The first argument to the call is a communicator and the rank of the process is returned in the second argument. Essentially a communicator is a collection of process that can send message to each other.
3) MPI_Comm_Size(MPI_comm,intnum_of_processors)
– Determine the size of the group associated with a communicator.
– This function determines the number of processors executing the program. Its first argument is the communicator and it returns the number of processes in the communicator in its second argument.
4) MPI_Send (void *message, int count, MPI_Datatype datatype, int destination, int tag, MPI_comm comm.)
– Basic send(it's a blocking send call)
– The first three arguments describe the message as the address, count and the datatype. The content of the message are stored in the block of memory referenced by the address. The count specifies the number of elements contained in the message. Which are of a MPI type MPI_DATATYPE. The next argument is a destination, an integer specifying the rank of the destination process. The tag argument helps identify message.
5) MPI_Recv(void *message, int count, MPI_Datatype datatype, int source, int tag, MPI_comm comm., MPI_status *status)
– Basic Receive (it's a blocking receive call)
– The first three arguments describe the message as address, count and the datatype. The content of the message are stored in the block of memory referenced by the address. The count specifies the number of elements contained in the message which are of a MPI type MPI_DATATYPE. The next argument is the source which specifies the rank of the sending process. MPI allows the source to be a wild card. This is a predefined constant MPI_ANY_SOURCE that can be used if a process is ready to receive a message from any sending process rather than a particular sending process. The tag argument helps to identify message. That last argument return information on the data that was actually received. It references a record with two field – one for the source and the other for the tag.
6) MPI_Bcast(void *message, int count, MPI_Datatype datatype, int root, MPI_Comm comm.)
– Broadcast a message from the process with rank "root" to all other processes of the group.
– It is a collective communication call in which a single process sends same data to every process. It sends a copy of the data in message on process root to each process in the communicator comm. it should be called by all processors in the communicator with same arguments for root and com.
7) MPI_Reduce(void *operand, void *result, int count, MPI_Datatype datatype, MPI_Operator op, int root, MPI_Com comm.)
– Reduce values on all processes to a single value.
– MPI_Reduce combines the operands stored in *operand using operation op and stores the result on *result on the root. Both operand and result refer cound memory locations with type datatype. MPI_Reduce must be called by all the processor in the communicator comm.,

and count, datatype and op must be same on each processor.

8) MPI_Scatter(void *send_buffer, int send_count, MPI_DATATYPE send_type, void *recv_buffer, int recv_count,MPI_DATATYPE recv_type,int root, MPI_Comm comm.)
- Sends data from one processor to all other processed in a group.
- The process with rank root distributes the contents of send_buffer among the processes. The contents of send_buffer are split into P segments each consisting of send_count elements. The first segment goes to process 0, the second to process 1 etc. The send arguments are significant only on process root.

9) MPI_Gather(void *send_buffer, int send_count,MPI_DATATYPE send_type,void *recv_buffer, int recv_count, MPI_DATATYPE recv_type, int root, MPI_Comm comm.)
- Gathers together value from a group of tasks.
- Each process in comm sends the contents of send_buffer to the process with rank root. The process rank order in recv_buffer. The receive arguments are significant only on the process with rank root. The argument recv_count indicates the number of items received from each process – not the total number received.

10) MPI_Finalize()
- Terminates MPI execution environment
- This call must be made by every process in a MPI computation. It terminate the MPI "environment". , no MPI call maybe made by a process after its call.

11) MPI_Comm_split(MPI_Comm old_comm,int split_key,int rank_key, MPI_comm *new_comm)
- Create a new communicator based on the color and keys.
- The single call to MPI_Comm_split creates a new communicators, all of them having the same name, *new_comm. It creates a new communicator for each value of the split_key. Process with the same value of split_key form a new group. The rank in the new group is determined by the value of rank_key. If process A and process B call MPI_comm split with the same value of split_key and the rank_key argument passed by process A is less than that passed by process B, then the rank of A is underlying group new_comm will be less thank the rank of process B. it is a collective call and it must be called by the processes in old_comm.

12) MPI_Comm_group(MPI_comm comm.,MPI_Group *group)
- Accesses the group associated with the given communicator

13) MPI_Grou_incl(MPI_Group old_group, int new_group_size, int *rank_in_old_group,MPI_Group *new_group)
- Produce a group by reordering an existing group and taking only unlisted members.

VI. CONCLUSION

MPI can implement on all parallel computers even on your future machine. It's easy to learn but something difficulties to implement on real application.

REFERENCES

[1] D.Walker. Standard for Message passing in a distributed memory environment. Technical report TM-12147, Oak Ridge National Laboratory, August 1992.
[2] https://computing.llnl.gov

