Message Passing Interface

The Message Passing Interface (MPI) is an open library standard for distributed memory parallelization. The library API (Application Programmer Interface) specification is available for C and Fortran. There exist unofficial language bindings for many other programming languages, e.g. Python3, or JAVA 1, 2, 3. The first standard document was released in 1994. MPI has become the de-facto standard to program HPC cluster systems and is often the only way available. There exist many implementations, Open source and proprietary. The latest version of the standard is MPI 3.1(released in 2015).

MPI allows to write portable parallel programs for all kinds of parallel systems, from small shared memory nodes to petascale cluster systems. While many criticize its bloated API and complicated function interface no alternative proposal could win a significant share in the HPC application domain so far. There exist optimized implementations, open source and proprietary, for any HPC platform and architecture and a wealth of tools and libraries. Common implementations are OpenMPI, mpich and Intel MPI. Because MPI is available for such a long time and almost any HPC application is implemented using MPI it is the safest bet for a solution that will be supported and stable on mid- to long-term future systems.

Information on how to run an existing MPI program can be found in the How to Use MPI Section.

MPI Program

- Written in C
- Need to add mpi.h header file
- Identifier defined by MPI start with “MPI_”

MPI Component

- MPI_Init
  - Initialize the MPI execution environment.
  - This routine must be called before any other MPI routine.
- MPI_Finalize
  - Terminate MPI execution environment.
  - This routine cleans up all MPI state.
- Communicators
  - A collection of processes that can send message to each other.
  - MPI_Init defines a communicator that consist of all the processes created when the program is started.

- Called MPI_COMM_WORLD

API Overview

The standard specifies interfaces to the following functionality (incomplete list):

- Point-to-point communication,
- Datatypes,
- Collective operations,
- Process groups,
- Process topologies,
- One-sided communication,
- Parallel file I/O.

Point-to-Point communication

Sending and receiving of messages by process is the basics MPI communication mechanism. P2P communication operation are send and receive.

Syntax:

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

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

1. 1st argument - msg_buf_p = It is pointer to the block of memory containing the contents of the message.
2. 2nd & 3rd argument - msg_size & msg_type = The amount of data to be sent. msg_size is no. of characters in the message and msg_type argument is MPI_CHAR.

Data Types

MPI also provides facilities for you to define your own data structures based upon sequences of the MPI primitive data types. Such user defined structures are called derived data types.

Primitive data types are contiguous. Derived data types allow you to specify non-contiguous data in a convenient manner and to treat it as though it was contiguous.

MPI provides several methods for constructing derived data types:

- Contiguous
- Vector

- Indexed
- Struct

Collective Operations

MPI supports collective communication primitives for cases where all ranks are involved in communication.

There exist three types of collective communication:

- Synchronization (barrier)
- Data movement (e.g. gather, scatter, broadcast)
- Collective computation (reduction)

Syntax:

#MPI_Barrier(Comm)

Blocks until all processes in the communicator have reached this routine.

MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm);

Broadcasts a message from the process with rank root to all processes of the communicator, itself included

MPI_Reduce(**const** void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm);

Reduce values on all processes to a single value.

MPI_Allreduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);

Combine values from all processes and distribute the result back in all processes.

SSSMPI_Scatter(void & sendbuf, int sendcnt, MPI_Datatype send type, void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm);

Sends data from one process to all other processes in a communicator.

MPI_Gatter(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, MPI_Datatype recvtype, int root, MPI_Comm comm)

Gathers together values from a group of processes.

MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)

Send data from all to all processes.

MPI_Scan(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);

Computes the scan (partial reductions) of data on a collection of processes.

Process Group

A group is an ordered set of processes. Each process in a group is associated with a unique integer rank. Rank values start at zero and go to N-1, where N is the number of processes in the group. In MPI, a group is represented within system memory as an object. It is accessible to the programmer only by a "handle". A group is always associated with a communicator object.

Primary Purposes of Group and Communicator Objects:

1. Allow you to organize tasks, based upon function, into task groups.
2. Enable Collective Communications operations across a subset of related tasks.
3. Provide basis for implementing user defined virtual topologies.
4. Provide for safe communications.

Virtual Topologies

What are they?

- In terms of MPI, a virtual topology describes a mapping/ordering of MPI processes into a geometric "shape".
- The two main types of topologies supported by MPI are Cartesian (grid) and Graph.
- MPI topologies are virtual - there may be no relation between the physical structure of the parallel machine and the process topology.
- Virtual topologies are built upon MPI communicators and groups.
- They must be "programmed" by the application developer.

Why Use Them?

- **Convenience**

Virtual topologies may be useful for applications with specific communication patterns - patterns that match an MPI topology structure. For example, a Cartesian topology might prove convenient for an application that requires 4-way nearest neighbor communications for grid based data.

- **Communication Efficiency**

Some hardware architectures may impose penalties for communications between successively distant "nodes".

A particular implementation may optimize process mapping based upon the physical characteristics of a given parallel machine.

The mapping of processes into an MPI virtual topology is dependent upon the MPI implementation, and may be totally ignored.

A Brief Word on MPI-2 and MPI-3

MPI-2:

Intentionally, the MPI-1 specification did not address several "difficult" issues. For reasons of expediency, these issues were deferred to a second specification, called MPI-2 in 1998. MPI-2 was a major revision to MPI-1 adding new functionality and corrections.

Key areas of new functionality in MPI-2:

- *Dynamic Processes* - extensions that remove the static process model of MPI. Provides routines to create new processes after job startup.
- *One-Sided Communications* - provides routines for one directional communications. Include shared memory operations (put/get) and remote accumulate operations.
- *Extended Collective Operations* - allows for the application of collective operations to inter-communicators
- *External Interfaces* - defines routines that allow developers to layer on top of MPI, such as for debuggers and profilers.
- *Additional Language Bindings* - describes C++ bindings and discusses Fortran-90 issues.
- *Parallel I/O* - describes MPI support for parallel I/O.

MPI-3:

The MPI-3 standard was adopted in 2012, and contains significant extensions to MPI-1 and MPI-2 functionality including:

- *Nonblocking Collective Operations* - permits tasks in a collective to perform operations without blocking, possibly offering performance improvements.
- *New One-sided Communication Operations* - to better handle different memory models.
- *Neighborhood Collectives* - extends the distributed graph and Cartesian process topologies with additional communication power.
- *Fortran 2008 Bindings* - expanded from Fortran90 bindings
- *MPIT Tool Interface* - allows the MPI implementation to expose certain internal variables, counters, and other states to the user (most likely performance tools).
- *Matched Probe* - fixes an old bug in MPI-2 where one could not probe for messages in a multi-threaded environment.