Message Passing Interface

From Wikipedia, the free encyclopedia

This article has multiple issues. Please help **improve it** or discuss these issues on the **talk page**. *(Learn how and when to remove these template messages)* [hide]

- This article **possibly contains unsourced predictions, speculative material, or accounts of events that might not occur**. Information must be verifiable and based on reliable published sources. *(June 2010)*
- This article **possibly contains original research**. *(May 2008)*
- This article may need to be **rewritten entirely** to comply with Wikipedia's quality standards. *(August 2011)*

Message Passing Interface (**MPI**) is a standardized and portable message-passing standard designed by a group of researchers from academia and industry to function on a wide variety of parallel computing architectures. The standard defines the syntax and semantics of a core of library routines useful to a wide range of users writing portable message-passing programs in C, C++, and Fortran. There are several well-tested and efficient implementations of MPI, many of which are open-source or in the public domain. These fostered the development of a parallel software industry, and encouraged development of portable and scalable large-scale parallel applications.

Contents [hide]

History [edit]

This section **does not cite any sources**. Please help improve this section by adding citations to reliable sources. Unsourced material may be challenged and removed. *(August 2015) (Learn how and when to remove this template message)*

The message passing interface effort began in the summer of 1991 when a small group of researchers started discussions at a mountain retreat in Austria. Out of that discussion came a Workshop on Standards for Message Passing in a Distributed Memory Environment held on April 29–30, 1992 in Williamsburg, Virginia. Attendees at Williamsburg discussed the basic features essential to a standard message-passing interface and established a working group to continue the standardization process. Jack Dongarra, Tony

WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction

Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools

What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export

Create a book
Download as PDF
Printable version

Languages

Deutsch
Español
Français
Italiano
Nederlands
Polski
Русский
Türkçe
中文
12 more
Edit links

Hey, and David W. Walker put forward a preliminary draft proposal, "MPI1", in November 1992. In November 1992 a meeting of the MPI working group took place in Minneapolis and decided to place the standardization process on a more formal footing. The MPI working group met every 6 weeks throughout the first 9 months of 1993. The draft MPI standard was presented at the Supercomputing '93 conference in November 1993. After a period of public comments, which resulted in some changes in MPI, version 1.0 of MPI was released in June 1994. These meetings and the email discussion together constituted the MPI Forum, membership of which has been open to all members of the high-performance-computing community.

The MPI effort involved about 80 people from 40 organizations, mainly in the United States and Europe. Most of the major vendors of concurrent computers were involved in MPI - along with researchers from universities, government laboratories, and industry.

MPI provides parallel hardware vendors with a clearly defined base set of routines that can be efficiently implemented. As a result, hardware vendors can build upon this collection of standard low-level routines to create higher-level routines for the distributed-memory communication environment supplied with their parallel machines. MPI provides a simple-to-use portable interface for the basic user, yet one powerful enough to allow programmers to use the high-performance message passing operations available on advanced machines.

In an effort to create a universal standard for message passing, researchers incorporated the most useful features of several systems into MPI, rather than choosing one system to adopt as a standard. They adopted features from systems by IBM, Intel, nCUBE, PVM, Express, P4 and PARMACS. The message-passing paradigm is attractive because of wide portability and can be used in communication for distributed-memory and shared-memory multiprocessors, networks of workstations, and a combination of these elements. The paradigm can apply in multiple settings, independent of network speed or of memory architecture.

Support for MPI meetings came in part from DARPA and from the US National Science Foundation under grant ASC-9310330, NSF Science and Technology Center Cooperative agreement number CCR-8809615, and from the Commission of the European Community through Esprit Project P6643. The University of Tennessee also made financial contributions to the MPI Forum.

Overview [edit]

MPI is a communication protocol for programming parallel computers. Both point-to-point and collective communication are supported. MPI "is a message-passing application programmer interface, together with protocol and semantic specifications for how its features must behave in any implementation."[1] MPI's goals are high performance, scalability, and portability. MPI remains the dominant model used in high-performance computing today.[2]

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 supercomputers such as computer clusters often run such programs.

The principal MPI-1 model has no shared memory concept, and MPI-2 has only a limited distributed shared memory concept. Nonetheless, MPI programs are regularly run on shared memory computers, and both MPICH and Open MPI can use shared memory for message transfer if it is available.[3][4] Designing programs around the MPI model (contrary to explicit shared memory models) has advantages over NUMA architectures since MPI encourages memory locality. Explicit shared memory programming was introduced in MPI-3.[5][6][7]

Although MPI belongs in layers 5 and higher of the OSI Reference Model, implementations may cover most layers, with sockets and Transmission Control Protocol (TCP) used in the transport layer.

Most MPI implementations consist of a specific set of routines directly callable from C, C++, Fortran (i.e., an API) and any language able to interface with such libraries, including C#, Java or Python. The advantages of MPI over older message passing libraries are portability (because MPI has been implemented for almost every distributed memory architecture) and speed (because each implementation is in principle optimized for the hardware on which it runs).

MPI uses Language Independent Specifications (LIS) for calls and language bindings. The first MPI standard specified ANSI C and Fortran-77 bindings together with the LIS. The draft was presented at Supercomputing 1994 (November 1994)[8] and finalized soon thereafter. About 128 functions constitute the MPI-1.3 standard which was released as the final end of the MPI-1 series in 2008.[9]

At present, the standard has several versions: version 1.3 (commonly abbreviated *MPI-1*), which emphasizes message passing and has a static runtime environment, MPI-2.2 (MPI-2), which includes new features such as parallel I/O, dynamic process management and remote memory operations,[10] and MPI-3.1 (MPI-3), which includes extensions to the collective operations with non-blocking versions and extensions to the one-sided operations.[11] MPI-2's LIS specifies over 500 functions and provides language bindings for ISO C, ISO C++, and Fortran 90. Object interoperability was also added to allow easier mixed-language message passing programming. A side-effect of standardizing MPI-2, completed in 1996, was clarifying the MPI-1 standard, creating the MPI-1.2.

MPI-2 is mostly a superset of MPI-1, although some functions have been deprecated. MPI-1.3 programs still work under MPI implementations compliant with the MPI-2 standard.

MPI-3 includes new Fortran 2008 bindings, while it removes deprecated C++ bindings as well as many deprecated routines and MPI objects.

MPI is often compared with Parallel Virtual Machine (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 (such as Pthreads and OpenMP) and message passing programming (MPI/PVM) can be considered as complementary programming approaches, and can occasionally be seen together in applications, e.g. in servers with multiple large shared-memory nodes.

Functionality [edit]

The MPI interface is meant to provide essential virtual topology, synchronization, and communication functionality between a set of processes (that have been mapped to nodes/servers/computer instances) in a language-independent way, with language-specific syntax (bindings), plus a few language-specific features. MPI programs always work with processes, but programmers commonly refer to the processes as processors. Typically, for maximum performance, each CPU (or core in a multi-core machine) will be assigned just a single process. This assignment happens at runtime through 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 (send/receive operations), combining partial results of computations (gather and reduce operations), synchronizing nodes (barrier operation) as well as obtaining network-related information such as the number of processes in the computing session, current processor identity that a process is mapped to, neighboring processes 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[*clarification needed*] 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, and some implementations support such code. Multithreaded collective communication is best accomplished with multiple copies of Communicators, as described below.

Concepts [edit]

MPI provides a rich range of abilities. The following concepts help in understanding and providing context for all of those abilities and help the programmer to decide what functionality to use in their application programs. Four of MPI's eight basic concepts are unique to MPI-2.

Communicator [edit]

Communicator objects connect groups of processes in the MPI session. Each communicator gives each contained process an independent identifier and arranges its contained processes in an ordered topology. MPI also has explicit groups, but these are mainly good for organizing and reorganizing groups of processes before another communicator is made. MPI understands single group intracommunicator operations, and bilateral intercommunicator communication. In MPI-1, single group operations are most prevalent. Bilateral operations mostly appear in MPI-2 where they include collective communication and dynamic in-process management.

Communicators can be partitioned using several MPI commands. These commands include `MPI_COMM_SPLIT`, where each process joins one of several colored sub-communicators by declaring itself to have that color.

Point-to-point basics [edit]

A number of important MPI functions involve communication between two specific processes. A popular example is `MPI_Send`, which allows one specified process to send a message to a second specified process. Point-to-point operations, as these are called, are particularly useful in patterned or irregular communication, for example, a data-parallel architecture in which each processor routinely swaps regions of data with specific other processors between calculation steps, or a master-slave architecture in which the master sends new task data to a slave whenever the prior task is completed.

MPI-1 specifies mechanisms for both blocking and non-blocking point-to-point communication mechanisms, as well as the so-called 'ready-send' mechanism whereby a send request can be made only when the matching receive request has already been made.

Collective basics [edit]

Collective functions involve communication among all processes in a process group (which can mean the entire process pool or a program-defined subset). A typical function is the `MPI_Bcast` call (short for "broadcast"). This function takes data from one node and sends it to all processes in the process group. A reverse operation is the `MPI_Reduce` call, which takes data from all processes in a group, performs an operation (such as summing), and stores the results on one node. `MPI_Reduce` is often useful at the start or end of a large distributed calculation, where each processor operates on a part of the data and then combines it into a result.

Other operations perform more sophisticated tasks, such as `MPI_Alltoall` which rearranges *n* items of data such that the *n*th node gets the *n*th item of data from each.

Derived datatypes [edit]

Many MPI functions require that you specify the type of data which is sent between processes. This is because MPI aims to support heterogeneous environments where types might be represented differently on the different nodes[12] (for example they might be running different CPU architectures that have different endianness), in which case MPI implementations can perform *data conversion*.[12] Since the C language does not allow a type itself to be passed as a parameter, MPI predefines the constants `MPI_INT`, `MPI_CHAR`, `MPI_DOUBLE` to correspond with `int`, `char`, `double`, etc.

Here is an example in C that passes arrays of `int`s from all processes to one. The one receiving process is called the "root" process, and it can be any designated process but normally it will be process 0. All the processes ask to send their arrays to the root with `MPI_Gather`, which is equivalent to having each process (including the root itself) call `MPI_Send` and the root make the corresponding number of ordered `MPI_Recv` calls to assemble all of these arrays into a larger one:[13]

```c
int send_array[100];
int root = 0; /* or whatever */
int num_procs, *recv_array;
MPI_Comm_size(comm, &num_procs);
recv_array = malloc(num_procs * sizeof(send_array));
MPI_Gather(send_array, sizeof(send_array) / sizeof(*send_array), MPI_INT,
           recv_array, sizeof(send_array) / sizeof(*send_array), MPI_INT,
           root, comm);
```

However, you may instead wish to send data as one block as opposed to 100 `int`s. To do this define a "contiguous block" derived data type:

```
MPI_Datatype newtype;
MPI_Type_contiguous(100, MPI_INT, &newtype);
MPI_Type_commit(&newtype);
MPI_Gather(array, 1, newtype, receive_array, 1, newtype, root, comm);
```

For passing a class or a data structure, `MPI_Type_create_struct` creates an MPI derived data type from `MPI_predefined` data types, as follows:

```
int MPI_Type_create_struct(int count,
                           int *blocklen,
                           MPI_Aint *disp,
                           MPI_Datatype *type,
                           MPI_Datatype *newtype)
```

where:

- `count` is a number of blocks, and specifies the length (in elements) of the arrays `blocklen`, `disp`, and `type`.
- `blocklen` contains numbers of elements in each block,
- `disp` contains byte displacements of each block,
- `type` contains types of element in each block.
- `newtype` (an output) contains the new derived type created by this function

The `disp` (displacements) array is needed for data structure alignment, since the compiler may pad the variables in a class or data structure. The safest way to find the distance between different fields is by obtaining their addresses in memory. This is done with `MPI_Get_address`, which is normally the same as C's `&` operator but that might not be true when dealing with memory segmentation.[14]

Passing a data structure as one block is significantly faster than passing one item at a time, especially if the operation is to be repeated. This is because fixed-size blocks do not require serialization during transfer.[15]

Given the following data structures:

```
struct A {
    int f;
    short p;
};

struct B {
    struct A a;
    int pp, vp;
};
```

Here's the C code for building an MPI-derived data type:

```
static const int blocklen[] = {1, 1, 1, 1};
static const MPI_Aint disp[] = {
    offsetof(struct B, a) + offsetof(struct A, f),
    offsetof(struct B, a) + offsetof(struct A, p),
    offsetof(struct B, pp),
    offsetof(struct B, vp)
};
static MPI_Datatype type[] = {MPI_INT, MPI_SHORT, MPI_INT, MPI_INT};
MPI_Datatype newtype;
MPI_Type_create_struct(sizeof(type) / sizeof(*type), blocklen, disp, type, &newtype);
MPI_Type_commit(&newtype);
```

MPI-2 concepts [edit]

One-sided communication [edit]

MPI-2 defines three one-sided communications operations, `MPI_Put`, `MPI_Get`, and `MPI_Accumulate`, being a write to remote memory, a read from remote memory, and a reduction operation on the same memory across a number of tasks, respectively. Also defined are three different methods to synchronize this communication (global, pairwise, and remote locks) as the specification does not guarantee that these operations have taken place until a synchronization point.

These types of call can often be useful for algorithms in which synchronization would be inconvenient (e.g. distributed matrix multiplication), or where it is desirable for tasks to be able to balance their load while other processors are operating on data.

Collective extensions [edit]

This section **needs expansion**. You can help by adding to it. *(June 2008)*

This section needs to be developed.

Dynamic process management [edit]

This section **needs expansion**. You can help by adding to it. *(June 2008)*

The key aspect is "the ability of an MPI process to participate in the creation of new MPI processes or to establish communication with MPI processes that have been started separately." The MPI-2 specification describes three main interfaces by which MPI processes can dynamically establish communications, `MPI_Comm_spawn`, `MPI_Comm_accept` / `MPI_Comm_connect` and `MPI_Comm_join`. The `MPI_Comm_spawn` interface allows an MPI process to spawn a number of instances of the named MPI process. The newly spawned set of MPI processes form a new `MPI_COMM_WORLD` intracommunicator but can communicate with the parent and the intercommunicator the function returns. `MPI_Comm_spawn_multiple` is an alternate interface that allows the different instances spawned to be different binaries with different arguments.[16]

I/O [edit]

This section **needs expansion**. You can help by adding to it. *(June 2008)*

The parallel I/O feature is sometimes called MPI-IO,[17] and refers to a set of functions designed to abstract I/O management on distributed systems to MPI, and allow files to be easily accessed in a patterned way using the existing derived datatype functionality.

The little research that has been done on this feature indicates that it may not be trivial to get high performance gains by using MPI-IO. For example, an implementation of sparse matrix-vector multiplications using the MPI I/O library shows a general behavior of negligible performance gain, but these results are inconclusive.[18] It was not until the idea of collective I/O[19] implemented into MPI-IO that MPI-IO started to reach widespread adoption. Collective I/O substantially boosts applications' I/O bandwidth by having processes collectively transform the small and noncontiguous I/O operations into large and contiguous ones, thereby reducing the locking and disk seek overhead. Due to its vast performance benefits, MPI-IO also became the underlying I/O layer for many state-of-the-art I/O libraries, such as HDF5 and Parallel NetCDF. Its popularity also triggered a series of research efforts on collective I/O optimizations, such as layout-aware I/O[20] and cross-file aggregation[21][22].

Official implementations [edit]

- The initial implementation of the MPI 1.x standard was MPICH, from Argonne National Laboratory (ANL) and Mississippi State University. IBM also was an early implementor, and most early 90s supercomputer companies either commercialized MPICH, or built their own implementation. LAM/MPI from Ohio Supercomputer Center was another early open implementation. ANL has continued developing MPICH for over a decade, and now offers MPICH-3.2, implementing the MPI-3.1 standard.
- Open MPI (not to be confused with OpenMP) was formed by the merging FT-MPI, LA-MPI, LAM/MPI, and PACX-MPI, and is found in many TOP-500 supercomputers.

Many other efforts are derivatives of MPICH, LAM, and other works, including, but not limited to, commercial implementations from HP, Intel, and Microsoft.

While the specifications mandate a C and Fortran interface, the language used to implement MPI is not constrained to match the language or languages it seeks to support at runtime. Most implementations combine C, C++ and assembly language, and target C, C++, and Fortran programmers. Bindings are available for many other languages, including Perl, Python, R, Ruby, Java, and CL (see #Bindings).

Hardware implementations [edit]

MPI hardware research focuses on implementing MPI directly in hardware, for example via processor-in-memory, building MPI operations into the microcircuitry of the RAM chips in each node. By implication, this approach is independent of the language, OS or CPU, but cannot be readily updated or removed.

Another approach has been to add hardware acceleration to one or more parts of the operation, including hardware processing of MPI queues and using RDMA to directly transfer data between memory and the network interface without CPU or OS kernel intervention.

Compiler wrappers [edit]

mpicc (and similarly **mpic++**, **mpif90**, etc.) is a program that wraps over an existing compiler to set the necessary command-line flags when compiling code that uses MPI. Typically, it adds a few flags that enable the code to be the compiled and linked against the MPI library.[23]

Language bindings [edit]

Bindings are libraries that extend MPI support to other languages by wrapping an existing MPI implementation such as MPICH or Open MPI.

Common Language Infrastructure [edit]

The two managed Common Language Infrastructure .NET implementations are Pure Mpi.NET[24] and MPI.NET,[25] a research effort at Indiana University licensed under a BSD-style license. It is compatible with Mono, and can make full use of underlying low-latency MPI network fabrics.

Java [edit]

Although Java does not have an official MPI binding, several groups attempt to bridge the two, with different degrees of success and compatibility. One of the first attempts was Bryan Carpenter's mpiJava,[26] essentially a set of Java Native Interface (JNI) wrappers to a local C MPI library, resulting in a hybrid implementation with limited portability, which also has to be compiled against the specific MPI library being used.

However, this original project also defined the mpiJava API[27] (a de facto MPI API for Java that closely followed the equivalent C++ bindings) which other subsequent Java MPI projects adopted. An alternative, less-used API is MPJ API,[28] designed to be more object-oriented and closer to Sun Microsystems' coding conventions. Beyond the API, Java MPI libraries can be either dependent on a local MPI library, or implement the message passing functions in Java, while some like P2P-MPI also provide peer-to-peer functionality and allow mixed platform operation.

Some of the most challenging parts of Java/MPI arise from Java characteristics such as the lack of explicit pointers and the linear memory address space for its objects, which make transferring multidimensional arrays and complex objects inefficient. Workarounds usually involve transferring one line at a time and/or performing explicit de-serialization and casting at both sending and receiving ends, simulating C or Fortran-like arrays by the use of a one-dimensional array, and pointers to primitive types by the use of single-element arrays, thus resulting in programming styles quite far from Java conventions.

Another Java message passing system is MPJ Express.[29] Recent versions can be executed in cluster and multicore configurations. In the cluster configuration, it can execute parallel Java applications on clusters and clouds. Here Java sockets or specialized I/O interconnects like Myrinet can support messaging between MPJ Express processes. It can also utilize native C implementation of MPI using its native device. In the multicore configuration, a parallel Java application is executed on multicore processors. In this mode, MPJ Express processes are represented by Java threads.

MATLAB [edit]

There are a few academic implementations of MPI using MATLAB. MATLAB has its own parallel extension library implemented using MPI and PVM.

OCaml [edit]

The OCamlMPI Module[30] implements a large subset of MPI functions and is in active use in scientific computing. An eleven-thousand-line OCaml program was "MPI-ified" using the module, with an additional 500 lines of code and slight restructuring and ran with excellent results on up to 170 nodes in a supercomputer.[31]

Python [edit]

MPI Python implementations include: pyMPI, mpi4py,[32] pypar,[33] MYMPI,[34] and the MPI submodule in ScientificPython. pyMPI is notable because it is a variant python interpreter, while pypar, MYMPI, and ScientificPython's module are import modules. They make it the coder's job to decide where the call to MPI_Init belongs.

Recently[*when?*] the Boost C++ Libraries acquired Boost:MPI which included the MPI Python Bindings.[35] This is of particular help for mixing C++ and Python. As of October 2016 Boost:MPI's Python bindings still have unfixed packaging bugs in CentOS.[36]

R [edit]

R bindings of MPI include Rmpi[37] and pbdMPI,[38] where Rmpi focuses on manager-workers parallelism while pbdMPI focuses on SPMD parallelism. Both implementations fully support Open MPI or MPICH2.

Example program [edit]

Here is a "Hello World" program in MPI written in C. In this example, we send a "hello" message to each processor, manipulate it trivially, return the results to the main process, and print the messages.

```
/*
  "Hello World" MPI Test Program
*/
#include <assert.h>
#include <stdio.h>
#include <string.h>
#include <mpi.h>

int main(int argc, char **argv)
{
    char buf[256];
    int my_rank, num_procs;

    /* Initialize the infrastructure necessary for communication */
    MPI_Init(&argc, &argv);

    /* Identify this process */
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    /* Find out how many total processes are active */
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

    /* Until this point, all programs have been doing exactly the same.
       Here, we check the rank to distinguish the roles of the programs */
    if (my_rank == 0) {
        int other_rank;
        printf("We have %i processes.\n", num_procs);
        /* Send messages to all other processes */
        for (other_rank = 1; other_rank < num_procs; other_rank++)
        {
            sprintf(buf, "Hello %i!", other_rank);
            MPI_Send(buf, sizeof(buf), MPI_CHAR, other_rank,
                     0, MPI_COMM_WORLD);
        }

        /* Receive messages from all other process */
        for (other_rank = 1; other_rank < num_procs; other_rank++)
        {
            MPI_Recv(buf, sizeof(buf), MPI_CHAR, other_rank,
                     0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            printf("%s\n", buf);
        }

    } else {

        /* Receive message from process #0 */
        MPI_Recv(buf, sizeof(buf), MPI_CHAR, 0,
                 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        assert(memcmp(buf, "Hello ", 6) == 0),

        /* Send message to process #0 */
        sprintf(buf, "Process %i reporting for duty.", my_rank);
        MPI_Send(buf, sizeof(buf), MPI_CHAR, 0,
                 0, MPI_COMM_WORLD);

    }

    /* Tear down the communication infrastructure */
    MPI_Finalize();
    return 0;
}
```

When run with 4 processes, it should produce the following output:[39]

```
$ mpicc example.c && mpiexec -n 4 ./a.out
We have 4 processes.
Process 1 reporting for duty.
Process 2 reporting for duty.
Process 3 reporting for duty.
```

Here, `mpiexec` is a command used to execute the example program with 4 processes, each of which is an independent instance of the program at run time and assigned ranks (i.e. numeric IDs) 0, 1, 2, and 3. The name `mpiexec` is recommended by the MPI standard, although some implementations provide a similar command under the name `mpirun`. The `MPI_COMM_WORLD` is the communicator that consists of all the processes.

A single program, multiple data (SPMD) programming model is thereby facilitated, but not required; many MPI implementations allow multiple, different, executables to be started in the same MPI job. Each process has its own rank, the total number of processes in the world, and the ability to communicate between them either with point-to-point (send/receive) communication, or by collective communication among the group. It is enough for MPI to provide an SPMD-style program with `MPI_COMM_WORLD`, its own rank, and the size of the world to allow algorithms to decide what to do. In more realistic situations, I/O is more carefully managed than in this example. MPI does not stipulate how standard I/O (stdin, stdout, stderr) should work on a given system. It generally works as expected on the rank-0 process, and some implementations also capture and funnel the output from other processes.

MPI uses the notion of process rather than processor. Program copies are *mapped* to processors by the MPI runtime. In that sense, the parallel machine can map to 1 physical processor, or N where N is the total number of processors available, or something in between. For maximum parallel speedup, more physical processors are used. This example adjusts its behavior to the size of the world N, so it also seeks to scale to the runtime configuration without compilation for each size variation, although runtime decisions might vary depending on that absolute amount of concurrency available.

MPI-2 adoption [edit]

Adoption of MPI-1.2 has been universal, particularly in cluster computing, but acceptance of MPI-2.1 has been more limited. Issues include:

1. MPI-2 implementations include I/O and dynamic process management, and the size of the middleware is substantially larger. Most sites that use batch scheduling systems cannot support dynamic process management. MPI-2's parallel I/O is well accepted.[*citation needed*]
2. Many MPI-1.2 programs were developed before MPI-2. Portability concerns initially slowed, although wider support has lessened this.
3. Many MPI-1.2 applications use only a subset of that standard (16-25 functions) with no real need for MPI-2 functionality.

Future [edit]

Some aspects of the MPI's future appear solid; others less so. The MPI Forum reconvened in 2007, to clarify some MPI-2 issues and explore developments for a possible MPI-3, which resulted in versions MPI-3.0 (September 2012) and MPI-3.1 (June 2015).

Architectures are changing, with greater internal concurrency (multi-core), better fine-grain concurrency control (threading, affinity), and more levels of memory hierarchy. Multithreaded programs can take advantage of these developments more easily than single-threaded applications. This has already yielded separate, complementary standards for symmetric multiprocessing, namely OpenMP. MPI-2 defines how standard-conforming implementations should deal with multithreaded issues, but does not require that implementations be multithreaded, or even thread-safe. MPI-3 adds the ability to use shared-memory parallelism within a node. Implementations of MPI such as Adaptive MPI, Hybrid MPI, Fine-Grained MPI, MPC and others offer extensions to the MPI standard that address different challenges in MPI.

Astrophysicist Jonathan Dursi wrote an opinion piece that MPI is obsolescent, pointing to newer technologies like Chapel, Unified Parallel C, Hadoop, Spark and Flink.[40]

See also [edit]

- Actor model
- Adaptive MPI (documentation)
- Bulk synchronous parallel programming
- Calculus of Broadcasting Systems
- Calculus of communicating systems
- Caltech Cosmic Cube
- Chapel (programming language)
- Charm++
- Co-array Fortran
- Global Arrays
- Linda (coordination language)
- Microsoft Messaging Passing Interface
- MPICH
- MVAPICH
- occam (programming language)
- Open MPI
- OpenHMPP (HPC Open Standard for Manycore Programming)
- OpenMP
- Parallel Virtual Machine
- Partitioned global address space
- Unified Parallel C
- X10 (programming language)

References [edit]

1. ^ Gropp, Lusk & Skjellum 1996, p. 3
2. ^ Sur, Sayantan; Koop, Matthew J.; Panda, Dhabaleswar K. (4 August 2017). "High-performance and Scalable MPI over InfiniBand with Reduced Memory Usage: An In-depth Performance Analysis". ACM. doi:10.1145/1188455.1188565 – via ACM Digital Library.
3. ^ KNEM: High-Performance Intra-Node MPI Communication "MPICH2 (since release 1.1.1) uses KNEM in the DMA LMT to improve large message performance within a single node. Open MPI also includes KNEM support in its SM BTL component since release 1.5. Additionally, NetPIPE includes a KNEM backend since version 3.7.2."
4. ^ "FAQ: Tuning the run-time characteristics of MPI sm communications". *www.open-mpi.org*.
5. ^ https://software.intel.com/en-us/articles/an-introduction-to-mpi-3-shared-memory-programming?language=en "The MPI-3 standard introduces another approach to hybrid programming that uses the new MPI Shared Memory (SHM) model"
6. ^ Shared Memory and MPI 3.0 "Various benchmarks can be run to determine which method is best for a particular application, whether using MPI + OpenMP or the MPI SHM extensions. On a fairly simple test case, speedups over a base version that used point to point communication were up to 5X, depending on the message."
7. ^ Using MPI-3 Shared Memory As a Multicore Programming System (PDF presentation slides)
8. ^ Table of Contents — September 1994, 8 (3-4). Hpc.sagepub.com. Retrieved on 2014-03-24.
9. ^ MPI Documents. Mpi-forum.org. Retrieved on 2014-03-24.
10. ^ Gropp, Lusk & Skjellum 1999b, pp. 4–5
11. ^ MPI: A Message-Passing Interface Standard Version 3.1, Message Passing Interface Forum, June 4, 2015. http://www.mpi-forum.org. Retrieved on 2015-06-16.
12. ^ *a b* "Type matching rules". *mpi-forum.org*.
13. ^ "MPI_Gather(3) man page (version 1.8.8)". *www.open-mpi.org*.
14. ^ "MPI_Get_address". *www.mpich.org*.
15. ^ Boost.MPI Skeleton/Content Mechanism rationale (performance comparison graphs were produced using NetPIPE)
16. ^ Gropp, Lusk & Skjelling 1999b, p. 7
17. ^ Gropp, Lusk & Skjelling 1999b, pp. 5–6
18. ^ "Sparse matrix-vector multiplications using the MPI I/O library" (PDF).
19. ^ "Data Sieving and Collective I/O in ROMIO" (PDF). IEEE. Feb 1999.
20. ^ "LACIO: A New Collective I/O Strategy for Parallel I/O Systems" (PDF). IEEE. Sep 2011.
21. ^ Teng Wang; Kevin Vasko; Zhuo Liu; Hui Chen; Weikuan Yu (2016). "Enhance parallel input/output with cross-bundle aggregation". *The International Journal of High Performance Computing Applications*. **30** (2): 241–256.
22. ^ "BPAR: A Bundle-Based Parallel Aggregation Framework for Decoupled I/O Execution" (PDF). IEEE. Nov 2014.
23. ^ mpicc. Mpich.org. Retrieved on 2014-03-24.
24. ^ Pure Mpi.NET
25. ^ "MPI.NET: High-Performance C# Library for Message Passing". *www.osl.iu.edu*.
26. ^ "mpiJava Home Page". *www.hpjava.org*.
27. ^ "Introduction to the mpiJava API". *www.hpjava.org*.
28. ^ "The MPJ API Specification". *www.hpjava.org*.
29. ^ "MPJ Express Project". *mpj-express.org*.
30. ^ "Xavier Leroy - Software". *cristal.inria.fr*.
31. ^ Archives of the Caml mailing list > Message from Yaron M. Minsky. Caml.inria.fr (2003-07-15). Retrieved on 2014-03-24.
32. ^ "Google Code Archive - Long-term storage for Google Code Project Hosting". *code.google.com*.
33. ^ "Google Code Archive - Long-term storage for Google Code Project Hosting". *code.google.com*.
34. ^ Now part of Pydusa
35. ^ "Python Bindings - 1.35.0". *www.boost.org*.
36. ^ "0006498: Package boost-*mpi-python is missing python module - CentOS Bug Tracker". *bugs.centos.org*.
37. ^ Yu, Hao (2002). "Rmpi: Parallel Statistical Computing in R". *R News*.
38. ^ Chen, Wei-Chen; Ostrouchov, George; Schmidt, Drew; Patel, Pragneshkumar; Yu, Hao (2012). "pbdMPI: Programming with Big Data -- Interface to MPI".
39. ^ The output snippet was produced on an ordinary Linux desktop system with Open MPI installed. Distros usually place the mpicc command into an openmpi-devel or libopenmpi-dev package, and sometimes make it necessary to run "module add mpi/openmpi-x86_64" or similar before mpicc and mpiexec are available.
40. ^ "HPC is dying, and MPI is killing it". *www.dursi.ca*.

Further reading [edit]

- This article is based on material taken from the *Free On-line Dictionary of Computing* prior to 1 November 2008 and incorporated under the "relicensing" terms of the GFDL, version 1.3 or later.
- Aoyama, Yukiya; Nakano, Jun (1999) *RS/6000 SP: Practical MPI Programming*, ITSO
- Foster, Ian (1995) *Designing and Building Parallel Programs (Online)* Addison-Wesley ISBN 0-201-57594-9, chapter 8 *Message Passing Interface*
- Wijesuriya, Viraj Brian (2010-12-29) *Daniweb: Sample Code for Matrix Multiplication using MPI Parallel Programming Approach*
- *Using MPI* series:
  - Gropp, William; Lusk, Ewing; Skjellum, Anthony (1994). *Using MPI: portable parallel programming with the message-passing interface*. Cambridge, MA, USA: MIT Press Scientific And Engineering Computation Series. ISBN 0-262-57104-8.
  - Gropp, William; Lusk, Ewing; Skjellum, Anthony (1999a). *Using MPI, 2nd Edition: Portable Parallel Programming with the Message Passing Interface*. Cambridge, MA, USA: MIT Press Scientific And Engineering Computation Series. ISBN 978-0-262-57132-6.
  - Gropp, William; Lusk, Ewing; Skjellum, Anthony (1999b). *Using MPI-2: Advanced Features of the Message Passing Interface*. MIT Press. ISBN 978-0-262-57133-3.
  - Gropp, William; Lusk, Ewing; Skjellum, Anthony (2014). *Using MPI, 3rd edition: Portable Parallel Programming with the Message-Passing Interface*. Cambridge, MA, USA: MIT Press Scientific And Engineering Computation Series. ISBN 978-0-262-52739-2.
- Gropp, William; Lusk, Ewing; Skjellum, Anthony (1996). "A High-Performance, Portable Implementation of the MPI Message Passing Interface". *Parallel Computing*. CiteSeerX 10.1.1.102.9485.
- Pacheco, Peter S. (1997) *Parallel Programming with MPI*.[1] 500 pp. Morgan Kaufmann ISBN 1-55860-339-5.
- *MPI—The Complete Reference* series:
  - Snir, Marc; Otto, Steve W.; Huss-Lederman, Steven; Walker, David W.; Dongarra, Jack J. (1995) *MPI: The Complete Reference*. MIT Press Cambridge, MA, USA. ISBN 0-262-69215-5
  - Snir, Marc; Otto, Steve W.; Huss-Lederman, Steven; Walker, David W.; Dongarra, Jack J. (1998) *MPI—The Complete Reference: Volume 1, The MPI Core*. MIT Press, Cambridge, MA. ISBN 0-262-69215-5
  - Gropp, William; Huss-Lederman, Steven; Lumsdaine, Andrew; Lusk, Ewing; Nitzberg, Bill; Saphir, William; and Snir, Marc (1998) *MPI—The Complete Reference: Volume 2, The MPI-2 Extensions*. MIT Press, Cambridge, MA ISBN 978-0-262-57123-4
- Firuziaan, Mohammad; Nommensen, O. (2002) *Parallel Processing via MPI & OpenMP*, Linux Enterprise, 10/2002
- Vanneschi, Marco (1999) *Parallel paradigms for scientific computing* In Proceedings of the European School on Computational Chemistry (1999, Perugia, Italy), number 75 in *Lecture Notes in Chemistry*, pages 170–183. Springer, 2000
- Bala, Bruck, Cypher, Elustondo, A Ho, CT Ho, Kipnis, Snir (1995) ″A portable and tunable collective communication library for scalable parallel computers" in IEEE Transactions on Parallel and Distributed Systems,″ vol. 6, no. 2, pp. 154-164, Feb 1995.

External links [edit]

- Official website
- Official MPI-3.1 standard (unofficial HTML version)
- Message Passing Interface at Curlie
- Tutorial on MPI: The Message-Passing Interface
- A User's Guide to MPI
- Tutorial: Introduction to MPI (self-paced, includes self-tests and exercises)

Wikibooks has a book on the topic of: ***Message-Passing Interface***

V·T·E **Parallel computing** [show]

Categories: Application programming interfaces | Parallel computing