Message Passing Interface

Message Passing Interface (**MPI**) is a standardized and portable message-passing system designed by a group of researchers from academia and industry to function on a wide variety of parallel computers. 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 different computer programming languages such as Fortran, C, C++ and Java. There are several well-tested and efficient implementations of MPI, including some that are free or in the public domain. These fostered the development of a parallel software industry, and there encouraged development of portable and scalable large-scale parallel applications.

History

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. At this workshop the basic features essential to a standard message-passing interface were discussed, and a working group established to continue the standardization process. Jack Dongarra, Rolf Hempel, Tony Hey, and David W. Walker put forward a preliminary draft proposal in November 1992, this was known as MPI1. In November 1992, a meeting of the MPI working group was held in Minneapolis, at which it was 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.

The MPI 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 Fortran and C.

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 powerful enough to allow programmers to use the high-performance message passing operations available on advanced machines.

As an effort to create a “true” standard for message passing, researchers incorporated the most useful features of several systems into MPI, rather than choose one system to adopt as a standard. Features were used 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 is applicable in multiple settings, independent of network speed or memory architecture.

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

Overview

MPI is a language-independent communications 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 specifications 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.[1]

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. Designing programs around the MPI model (contrary to explicit shared memory models) 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 Transmission Control Protocol (TCP) used in the transport layer.

Most MPI implementations consist of a specific set of routines (i.e., an API) directly callable from C, C++, Fortran 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)[2] 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.[3]

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, and MPI-3.0 (MPI-3), which includes extensions to the collective operations with nonblocking versions and extensions to the one-sided operations.[4] MPI-2's LIS specifies over 500 functions and provides language bindings for ANSI C, ANSI C++, and ANSI Fortran (Fortran90). 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

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 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

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

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-communicator by declaring itself to have that color.

Point-to-point basics

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

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. 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 processor such that the *n*th node gets the *n*th item of data from each.

Derived datatypes

Many MPI functions require that you specify the type of data which is sent between processors. This is because these functions pass variables, not defined types. If the data type is a standard one, such as int, char, double, etc., you can use predefined MPI datatypes such as MPI_INT, MPI_CHAR, MPI_DOUBLE.

Here is an example in C that passes an array of ints and all the processors want to send their arrays to the root with MPI_Gather:

```
int array[100];
int root, total_p, *receive_array;

MPI_Comm_size(comm, &total_p);
receive_array=malloc(total_p*100*sizeof(*receive_array));
MPI_Gather(array, 100, MPI_INT, receive_array, 100, MPI_INT, root,
comm);
```

However, you may instead wish to send data as one block as opposed to 100 ints. 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);
```

Passing a class or a data structure cannot use a predefined data type. 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, also number of entries in blocklen[], disp[], and type[]:

- blocklen[] — number of elements in each block (array of integer)
- disp[] — byte displacement of each block (array of integer)
- type[] — type of elements in each block (array of handles to datatype objects).

The disp[] array is needed because processors require the variables to be aligned a specific way on the memory. For example, Char is one byte and can go anywhere on the memory. Short is 2 bytes, so it goes to even memory addresses. Long is 4 bytes, it goes on locations divisible by 4 and so on. The compiler tries to accommodate this architecture in a class or data structure by padding the variables. The safest way to find the distance between

different variables in a data structure is by obtaining their addresses with MPI_Get_address. This function calculates the displacement of all the structure's elements from the start of the data structure.

Given the following data structures:

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

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

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

```
void define_MPI_datatype(){

  //The first and last elements mark the beg and end of data structure
  int blocklen[6]={1,1,1,1,1,1};
  MPI_Aint disp[6];
  MPI_Datatype newtype;
  MPI_Datatype type[6]={MPI_LB, MPI_INT, MPI_SHORT, MPI_INT, MPI_INT,
MPI_UB};
  //You need an array to establish the upper bound of the data
structure
  B findsize[2];
  MPI_Aint findsize_addr, a_addr, f_addr, p_addr, pp_addr, vp_addr,
UB_addr;
  int error;

  MPI_Get_address(&findsize[0], &findsize_addr);
  MPI_Get_address(&(findsize[0]).a, &a_addr);
  MPI_Get_address(&((findsize[0]).a).f, &f_addr);
  MPI_Get_address(&((findsize[0]).a).p, &p_addr);
  MPI_Get_address(&(findsize[0]).pp, &pp_addr);
  MPI_Get_address(&(findsize[0]).vp, &vp_addr);
  MPI_Get_address(&findsize[1],&UB_addr);

  disp[0]=a_addr-findsize_addr;
  disp[1]=f_addr-findsize_addr;
  disp[2]=p_addr-findsize_addr;
  disp[3]=pp_addr-findsize_addr;
  disp[4]=vp_addr-findsize_addr;
  disp[5]=UB_addr-findsize_addr;

  error=MPI_Type_create_struct(6, blocklen, disp, type, &newtype);
  MPI_Type_commit(&newtype);
}
```

MPI-2 concepts

One-sided communication

MPI-2 defines three one-sided communications operations, Put, Get, and 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

This section needs to be developed.

Dynamic process management

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.

I/O

The parallel I/O feature is sometimes called MPI-IO, 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 the difficulty for good performance. For example, some implementations of sparse matrix-vector multiplications using the MPI I/O library are disastrously inefficient.[5]

Implementations

'Classical' cluster and supercomputer implementations

The MPI implementation language 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, CL.

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 2, implementing the MPI-2.1 standard. LAM/MPI and a number of other MPI efforts recently merged to form Open MPI. Many other efforts are derivatives of MPICH, LAM, and other works, including, but not limited to,

commercial implementations from HP, Intel, and Microsoft.

Python

MPI Python implementations include: pyMPI, mpi4py,[6] pypar,[7] MYMPI,[8] 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 the well known Boost C++ Libraries acquired Boost:MPI which included the MPI Python Bindings.[9] This is of particular help for mixing C++ and Python.

OCaml

The OCamlMPI Module[10] 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.[11]

Java

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,[12] 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[13] (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,[14] 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.[15] 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

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

R

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

Common Language Infrastructure

The two managed Common Language Infrastructure (CLI) .NET implementations are Pure Mpi.NET[16] and MPI.NET,[17] 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.

Hardware implementations

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.

mpicc

mpicc is a program which helps the programmer to use a standard C programming language compiler together with the Message Passing Interface (MPI) libraries, most commonly the OpenMPI implementation which is found in many TOP-500 supercomputers, for the purpose of producing parallel processing programs to run over computer clusters (often Beowulf clusters). The mpicc program uses a programmer's preferred C compiler and takes care of linking it with the MPI libraries.[18][19]

Example program

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.

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

#define BUFSIZE 128
#define TAG 0

int main(int argc, char *argv[])
{
  char idstr[32];
  char buff[BUFSIZE];
```

```
   int numprocs;
   int myid;
   int i;
   MPI_Status stat;
   /* MPI programs start with MPI_Init; all 'N' processes exist
thereafter */
   MPI_Init(&argc,&argv);
   /* find out how big the SPMD world is */
   MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
   /* and this processes' rank is */
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);

   /* At this point, all programs are running equivalently, the rank
      distinguishes the roles of the programs in the SPMD model, with
      rank 0 often used specially... */
   if(myid == 0)
   {
     printf("%d: We have %d processors\n", myid, numprocs);
     for(i=1;i<numprocs;i++)
     {
       sprintf(buff, "Hello %d! ", i);
       MPI_Send(buff, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_WORLD);
     }
     for(i=1;i<numprocs;i++)
     {
       MPI_Recv(buff, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_WORLD,
&stat);
       printf("%d: %s\n", myid, buff);
     }
   }
   else
   {
     /* receive from rank 0: */
     MPI_Recv(buff, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD, &stat);
     sprintf(idstr, "Processor %d ", myid);
     strncat(buff, idstr, BUFSIZE-1);
     strncat(buff, "reporting for duty", BUFSIZE-1);
     /* send to rank 0: */
     MPI_Send(buff, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD);
   }

   /* MPI programs end with MPI Finalize; this is a weak
synchronization point */
   MPI_Finalize();
   return 0;
 }
```

When run with two processors this gives the following output.[20]

```
0: We have 2 processors
0: Hello 1! Processor 1 reporting for duty
```

The runtime environment for the MPI implementation used (often called mpirun or mpiexec) spawns multiple copies of the program, with the total number of copies determining the number of process *ranks* in MPI_COMM_WORLD, which is an opaque descriptor for communication between the set of processes. A single process, 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 guarantee how POSIX I/O would actually work on a given system, but it commonly does work, at least from rank 0.

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

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.Wikipedia: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

Some aspects of 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.

Like Fortran, MPI is ubiquitous in technical computing, and it is taught and used widely.Wikipedia:Citation needed

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. Few multithreaded-capable MPI implementations exist. Multi-level concurrency completely within MPI is an opportunity for the standard.

References

[1] High-performance and scalable MPI over InfiniBand with reduced memory usage (http://portal.acm.org/citation.cfm?id=1188565)

[2] Table of Contents — September 1994, 8 (3-4) (http://hpc.sagepub.com/content/8/3-4.toc). Hpc.sagepub.com. Retrieved on 2014-03-24.

[3] MPI Documents (http://www.mpi-forum.org/docs/). Mpi-forum.org. Retrieved on 2014-03-24.

[4] MPI: A Message-Passing Interface Standard
Version 3.0, Message Passing Interface Forum, September 21, 2012 (http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf). http://www.mpi-forum.org.Retrieved on 2014-06-28.

[5] Sparse matrix-vector multiplications using the MPI I/O library (http://www.phys.uu.nl/~hulten/mod3a/report.ps)

[6] mpi4py (http://code.google.com/p/mpi4py/)

[7] pypar (http://code.google.com/p/pypar/)

[8] Now part of Pydusa (http://sourceforge.net/projects/pydusa/)

[9] Boost:MPI Python Bindings (http://www.boost.org/doc/libs/1_35_0/doc/html/mpi/python.html)

[10] OCamlMPI Module (http://cristal.inria.fr/~xleroy/software.html#ocamlmpi)

[11] Archives of the Caml mailing list > Message from Yaron M. Minsky (http://caml.inria.fr/pub/ml-archives/caml-list/2003/07/155910c4eeb09e684f02ea4ae342873b.en.html). Caml.inria.fr (2003-07-15). Retrieved on 2014-03-24.

[12] mpiJava (http://www.hpjava.org/mpiJava.html)

[13] mpiJava API (http://www.hpjava.org/theses/shko/thesis_paper/node33.html)

[14] MPJ API (http://www.hpjava.org/papers/MPJ-CPE/cpempi/node6.html)

[15] MPJ Express (http://mpj-express.org/)

[16] Pure Mpi.NET (http://www.purempi.net)

[17] MPI.NET (http://www.osl.iu.edu/research/mpi.net/)

[18] Woodman, Lawrence. (2009-12-02) Setting up a Beowulf Cluster Using Open MPI on Linux (http://techtinkering.com/2009/12/02/setting-up-a-beowulf-cluster-using-open-mpi-on-linux/). Techtinkering.com. Retrieved on 2014-03-24.

[19] mpicc (http://www.mpich.org/static/docs/latest/www1/mpicc.html). Mpich.org. Retrieved on 2014-03-24.

[20] Using OpenMPI, compiled with `gcc -g -v -I/usr/lib/openmpi/include/ -L/usr/lib/openmpi/include/ wiki_mpi_example.c -lmpi` and run with `mpirun -np 2 ./a.out`.

Further reading

- 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* (http://www.redbooks.ibm.com/abstracts/sg245380.html), ITSO
- Foster, Ian (1995) *Designing and Building Parallel Programs (Online)* Addison-Wesley ISBN 0-201-57594-9, chapter 8 *Message Passing Interface* (http://www-unix.mcs.anl.gov/dbpp/text/node94.html#SECTION03500000000000000000)
- Viraj B., Wijesuriya 2010-12-29 *Daniweb: Sample Code for Matrix Multiplication using MPI Parallel Programming Approach* (http://www.daniweb.com/forums/post1428830.html#post1428830)
- *Using MPI* series:
  - Gropp, William; Lusk, Ewing; Skjellum, Anthony (1994). *Using MPI: portable parallel programming with the message-passing interface* (http://www-unix.mcs.anl.gov/mpi/usingmpi/usingmpi-1st/index.html). 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* (http://mitpress.mit.edu/book-home.tcl?isbn=0262571323). 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 0-262-57133-1.
- 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 (http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.102.9485).

- Pacheco, Peter S. (1997) *Parallel Programming with MPI* (http://books.google.it/books?&id=tCVkM1z2aOoC). (http://www.cs.usfca.edu/mpi/) 500 pp. Morgan Kaufmann ISBN 1-55860-339-5.
- *MPI—The Complete Reference* series:
  - Snir, Marc; Otto, Steve; Huss-Lederman, Steven; Walker, David; Dongarra, Jack (1995) *MPI: The Complete Reference* (http://www.netlib.org/utk/papers/mpi-book/mpi-book.html). MIT Press Cambridge, MA, USA. ISBN 0-262-69215-5
  - M Snir, SW Otto, S Huss-Lederman, DW Walker, J (1998) *MPI—The Complete Reference: Volume 1, The MPI Core*. MIT Press, Cambridge, MA. ISBN 0-262-69215-5
  - Gropp, William; Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir (1998) *MPI—The Complete Reference: Volume 2, The MPI-2 Extensions* (http://mitpress.mit.edu/book-home.tcl?isbn=0262571234). MIT Press, Cambridge, MA ISBN 978-0-262-57123-4
- Parallel Processing via MPI & OpenMP, M. Firuziaan, O. Nommensen. Linux Enterprise, 10/2002
- Vanneschi, Marco (1999) *Parallel paradigms for scientific computing* In Proc. of the European School on Computational Chemistry (1999, Perugia, Italy), number 75 in *Lecture Notes in Chemistry* (http://books.google.com/books?&id=zMqVdFgVnrgC), pages 170–183. Springer, 2000.

External links

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

- "MPI Examples - Message Passing Interface" (http://www.haberdar.org/#tutorials). Hakan Haberdar, University of Houston. Retrieved October 2012.
- Message Passing Interface (http://www.dmoz.org/Computers/Parallel_Computing/Programming/Libraries/MPI) at DMOZ
- Tutorial on MPI: The Message-Passing Interface (http://polaris.cs.uiuc.edu/~padua/cs320/mpi/tutorial.pdf) (PDF)
- A User's Guide to MPI (http://moss.csc.ncsu.edu/~mueller/cluster/mpi.guide.pdf) (PDF)

Article Sources and Contributors

Message Passing Interface *Source*: http://en.wikipedia.org/w/index.php?oldid=621974825 *Contributors*: 2, A3 nm, Abs0ft2781, Adamantios, Alerante, AlexChurchill, AliveFreeHappy, Andrewhayes, Aquaeolian, ArglebargleIV, Avenged Eightfold, AxelBoldt, BagpipingScotsman, BarretB, BenFrantzDale, Binary Runner, Blaxthos, Blelbach, Bluemoose, Boggie, Bovineone, Boy1jhn, BradfordBaze, BrianDominy, Bsilverthorn, Bsod2, Byornski, CALR, CarlHewitt, Cdc, Codingking, Compfreak7, Cswierkowski, Cybercobra, DanKidger, Danger, Dave1g, Davepape, DavidBiesack, Db1618, DenisKrivosheev, Dgies, Drbreznjev, Drrngrvy, Dsimic, Earin, Edward, Egil, El Cubano, Emeraldemon, EmmetCaulfield, Emperorbma, EpiVictor, Erget2005, Erik Garrison, Flies 1, Flooey, Forderud, Foxj, Fprincipe, Frap, Fwyzard, GPHemsley, Ghettoblaster, Glennklockwood, Gloomy Coder, GoingBatty, Grendelkhan, Griggy1, Gronky, Hebrides, Hellisp, Hklimach, Hrafnkell.palsson, Hulten, Immunize, Ipsign, Iridescent, JJL, Jacob grace, JamesBWatson, Japs 88, Jarble, Jazzydee, JenniferForUnity, Jerryobject, Jesse V., Jim1138, Jin, Jmath666, Jnc, Joelmoniz, John of Reading, Juedsivi, Kaell, Katalaveno, Katmairock, Keithathaide, Ketil, Ketiltrout, Khazadum, Kirachinmoku, Kula85, Lebenworld, Leonard^Bloom, Les boys, Lfstevens, Liao, Locus99, LokiClock, Lordofcode, Lucadjtoni, M-le-mot-dit, Magioladitis, Marie Poise, Michael Hardy, Michael Suess, Mirv, Modamoda, Modster, MrOllie, Mreftel, MusicScience, Mwtoews, Nealmcb, Niceguyedc, Nnh, Nonugoel, Nuno Tavares, Oleszkie, Omicronpersei8, Paul Foxworthy, Phatom87, Phil Boswell, Phuzion, Qwertyus, Qxz, R'n'B, Raul654, Raysonho, Rege, Rich Farmbrough, Rjwilmsi, Rodrigo.toro, Romanc19s, Sheepe2004, Sigma 7, Skappes, Sligocki, Sliwers, Sofia Koutsouveli, Sspecter, Stardust85, Stillnotelf, SummerWithMorons, Superm401, Suruena, SvenDowideit, Syed Zafar Gilani, Tempodivalse, Thaddeusw, The Anome, TheAMmollusc, Thv, Tonyskjellum, Tyrantbrian, Uday, Un brice, Uniomni, Unknown, VanishedUserABC, Vetter, Vicarage, W Nowicki, Warren, Wbm1058, Webelity, Wenzeslaus, Windharp, Windwisp, Winterschlaefer, Wizard191, Zwilson14, 289 anonymous edits

Image Sources, Licenses and Contributors

Image:Wikibooks-logo-en-noslogan.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:Wikibooks-logo-en-noslogan.svg *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: User:Bastique, User:Ramac et al.

License