Overview of the MPI Standard and Implementations

Christina Zeeh

May 21, 2004

The Message Passing Interface (MPI) has become a standard for message passing parallel applications. This report first introduces the underlying paradigm, message passing, and explores some of the challenges explicit message passing poses for developing parallel programs. We then take a closer look at the MPI standardization effort, its goals, and its results to see what features the current version of the MPI standard provides and how MPI became what it is today. Several implementations of the MPI standard, including the open source LAM/MPI and MPICH implementations, and Sun MPI, as an example of a vendor-supplied MPI implementation, are being presented. A comparison is attempted regarding various aspects, such as supported MPI features, system architecture, network hardware, and operating system. Graph-oriented programming (GOP), a high-level abstraction for message passing applications based on MPI, is included as an example of ongoing research that aims to help the developer overcome some of the challenges the low-level approach taken by MPI poses. The report concludes with an outlook on the future of the MPI standard and its implementations, and how they are influenced by current trends in cluster computing.

1 Introduction

MPI stands for *Message Passing Interface*. Message passing is one of the oldest and most widely used paradigms for programming parallel computers, in particular for distributed memory machines/multicomputers. Communication happens exclusively through messages, which by itself implies some degree of synchronization. In contrast, when programming a shared memory multiprocessor, synchronization always has to be explicitly taken care of, for example through semaphores (Fig. 1).

	Multiprocessors	Multicomputers
Memory	shared memory	distributed memory
Communication	data in shared memory	message-passing
Synchronization	explicit	implicit

Figure 1: Multiprocessors and Multicomputers

1.1 Message Passing

Message passing is based on two primitives, *send* and *receive*. Issues such as buffering, blocking, and reliable communication introduce different message passing semantics:

- A message can be buffered by the message passing subsystem at the sender, the receiver, or both. This determines whether a send operation returns immediately, or needs to wait until the message is buffered, or even received at the other end.
- Operations can be blocking, meaning the operation will not complete until some condition is satisfied, for example, until the message is copied to a buffer, or received at the other side, etc.
- Reliability determines whether messages are guaranteed to make it to their destination, whether the sending order is preserved, and whether there are provisions against message corruption.

Characteristics

Developing applications using the basic message passing paradigm requires a low-level approach: Work and data must be explicitly distributed to processes, and communication happens exclusively through messages. All interactions between two processes therefore require both processes to actively participate – one sends, the other receives. Also, blocking communication comes with a danger of deadlocks. All of these characteristics lead to a widespread perception that writing message passing programs is hard. But those characteristics pose not only drawbacks: A low-level approach leaves room for optimization, and having to explicitly pass messages to communicate makes the programmer realize the cost of communication and leads to good locality.

2 The MPI Standard

2.1 History

In the early 1990s, message passing was already an established paradigm. Though some consensus had been reached on what functionality a message passing library is to provide, most of the available message passing libraries – usually vendor-supplied – were mutually incompatible. Applications developed for one message passing library could not easily be ported to a different one, leading to investments in software development being lost when moving to a different system. First attempts at developing portable message passing libraries were successful, and it was felt that the time was right to start developing a standard for message passing libraries.

The MPI Forum, a group of over 60 people from 40 organizations – hardware and software vendors, as well as research institutions – formed to develop this standard. An initial meeting at the Workshop on Standards for Message Passing in a Distributed Memory Environment in April 1992 was followed by preliminary draft proposals for the standard in November 1992 and February 1993. The draft of the standard was presented at the Supercomputing '93 conference, and version 1.0 of the **Message Passing Interface (MPI)** standard was released May 5, 1994 [6].

2.2 MPI-1

What is known today as MPI-1 has evolved over three releases of the standard: version 1.0, the original standard, released in April 1994, version 1.1, released in 1995, containing clarifications, corrections and additional examples, and version 1.2, which is incorporated in the MPI-2 standard, containing further clarifications, corrections, and a routine to obtain the standard version number.

The MPI-1 standard specifies routines for dealing with

- point-to-point communication
- collective communication
- communicators, groups, contexts and
- datatypes.

We will cover each of these topics of the MPI-1 standard in the following sections. Additional details and the exact syntax for the MPI-1 routines can be found in the MPI standard documents [6], [7].

2.3 Programming with MPI-1

MPI-1 specifies language bindings for C and Fortran 77, and – though not part of the standard – mentions the possibility for providing C++ and Fortran 90 bindings, which are expected to "(...) use the Fortran 77 and ANSI C bindings, respectively." [7].

The following is a simple "Hello World" program in C to illustrate some of the basic concepts of MPI programs:

```
 0 #include <string.h>
 1 #include <stdio.h>
 2 #include "mpi.h"
 3
 4 int main(int argc, char* argv[]) {
 5  char msg[15];
 6  int myrank;
 7  MPI_Status status;
 8  MPI_Init( &argc, &argv );
 9  MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
10  if (myrank == 0) {                  /* process 0 */
11   strcpy(msg,"Hello World");
12   MPI_Send(msg, strlen(msg), MPI_CHAR, 1, 99,
              MPI_COMM_WORLD);
13  } else {                            /* process 1 */
14   MPI_Recv(msg, 15, MPI_CHAR, 0, 99, MPI_COMM_WORLD,
              &status);
15   printf("Message: %s\n", msg);
16  }
17  MPI_Finalize(); 14
18  return 0;
19 }
```

The first lines deal with some formalia: MPI is a library, so its headers must be included into the program (line 2). In line 8, the MPI environment is initialized and in line 17 it is terminated. Before and after those lines, none of the MPI-related functions may be used.

The "Hello World" program is designed to run as two processes, process 0 and process 1. Each process uses the same program, which then takes different paths according to the rank of the process (*rank* will be explained shortly, for now just think of it as a process ID). The rank is determined in line 9, after that, the if-statement makes the two processes take different execution paths: Process 0 sends a message "Hello World" to process 1 (line 12), while process 1 receives this message (line 14) and prints it on standard out.

Compiling an MPI Program

MPI implementations are provided through means of a library. Therefore, MPI programs are compiled using a standard compiler.

Running an MPI Program

Most MPI implementations provide a command `mpirun` to run MPI programs. Often, a parameter to `mpirun` specifies the number of parallel processes that will run the MPI program. It is the MPI implementation's job to deal with distribution of the program to multiple nodes, this can be done using one of the remote login services (RSH, SSH) and/or daemons running on each node. On heterogeneous clusters and grids, an MPI implementation might also provide means to specify the nodes the program should run on, enabling users to select the computational power they require.

While the `mpirun` command is only a de-facto standard among MPI implementations, the MPI-2 standard specifies a command to run MPI implementations, `mpiexec`.

2.4 Ranks and Communicators

Every process in MPI belongs to one or more *communicators*. A communicator defines the scope of communication with the help of *groups* and *contexts*:

- **Groups** are ordered collections of processes. Each process in a group has a rank unique in the group, ranks range from 0 to the size of the group - 1.
- **Contexts** provide means to isolate one message space from another. It is similar to an additional tag for messages.

In addition to its group and its context, a communicator also provides means to store additional attributes, such as virtual topology information (see section 2.8).

There exist two types of communicators: intra-communicators and inter-communicators:

- An **intra-communicator** is used for communication between a group of processes.
- An **inter-communicator** is used for communication between two groups of processes. We will encounter this type of communicator later when talking about the MPI-2 standard (section 3.2).

2.5 Datatypes

MPI messages contain a fixed number of elements of a datatype. MPI distinguishes between *basic datatypes* like MPI_INT, MPI_FLOAT, MPI_CHAR, as well as the uninterpreted MPI_BYTE and MPI_PACKED, and *derived datatypes* that are build from basic datatypes or recursively from other derived datatypes. Except for MPI_PACKED and MPI_BYTE, the datatype specified in the send and receive calls must be identical.

Derived Datatypes allow for messages with mixed datatypes and for sending non-contiguous data without the overhead incurred by manually packing/unpacking data (memory-to-memory copies). A *type map* specifies the layout of such a derived datatype and consists of the datatypes that make up the derived datatype, and their respective displacements from the "start" of the datatype. There are four principal derived datatypes with corresponding constructors:

- `MPI_TYPE_CONTIGUOUS` is the most simple constructor, it simply repeats one datatype contiguously a certain number of times.

- `MPI_TYPE_VECTOR` provides for block-wise replication of a a datatype with equal spacing between each block.

- `MPI_TYPED_INDEXED` allows a different number of repetitions in each block, and also a different displacement for each block

- `MPI_TYPE_STRUCT` is the most powerful constructor, it adds the possibility to use different datatypes in each block

2.6 Point-to-Point Communication

MPI provides routines for basic message passing functionality between two processes using the send and receive primitives. For sending, blocking and non-blocking operations are provided, while for receiving there is only a blocking operation.

The basic form of an MPI is passed six parameters:

```
MPI_SEND(buf, count, datatype, dest, tag, comm)
```

- `buf` – buffer containing the data to be sent
- `count` – number of elements in buffer
- `datatype` – datatype of the buffer elements
- `dest` – destination's rank
- `tag` – message tag
- `comm` – communicator

The last three items, together with the message source, constitute the envelope of the MPI message.

2.6.1 Blocking Send

The blocking send operations of MPI block until some condition, according to the selected communication mode, is fulfilled. There are four communication modes:

- standard
- buffered
- synchronous
- ready

Standard Mode

In this mode, it is up to MPI to decide whether to buffer outgoing messages or not. If buffering is used, the call can complete before a matching receive is posted at the other side. If no buffering is used, the call will not complete before a matching receive is posted. Since this mode may depend on an action at the receiver, it is non-local.

Buffered Mode

In this mode, buffering is always used at the sender. Therefore, this call does not depend on a matching receive being issued at the other side and is local. If the sender runs out of buffer space, an error occurs.

Synchronous Mode

A call in this mode will only complete when a matching receive has been posted at the other side. Therefore, this call is non-local. In combination with a blocking receive operation it provides truly synchronous communication.

Ready Mode

In this mode, send can only be started if a matching receive has been posted at the other side, though no statement is being made about the state of such a receive operation. It is non-local and completes according to the semantics of the standard mode. The reason for the existence of this mode is that it saves some overhead on some systems.

2.6.2 Non-Blocking Send

MPI's non-blocking send operations use the same four communication modes as the blocking operations. The difference is that the non-blocking versions are split up into an *initiation* and a *completion* part. This allows the application to do computations while waiting for a communication to complete.

Communication is initiated using one of the `MPI_ISEND` operations, and completed using either `MPI_TEST` or `MPI_WAIT`. `MPI_TEST` checks if the send operation has completed and returns immediately, while `MPI_WAIT` is a blocking call that will return only when the send has completed. Thus, when using one of the `MPI_ISEND` functions immediately followed by an `MPI_WAIT`, it is equivalent to a blocking send.

2.6.3 Receive

The receive operation in MPI is always blocking. The message to be received is selected according to the parameters of the receive operation, it must match tag, source, and communicator of the message. It is possible to use wildcards (`MPI_ANY_SOURCE` and `MPI_ANY_TAG`) to match arbitrary messages.

2.7 Collective Communication

Collective communication operations, such as barrier, broadcast, scatter/gather, and reduction, allow for communication among all processes of a communicator. The functions for collective communication are *collective*, meaning they have to be called by all processes in the communicator before they return.

Barrier

A barrier is the simplest collective communication operation, its purpose is to allow all processes to rendezvous at a point in execution. The barrier function returns only when all processes in the communicator have called

it as shown in Fig. 2 – metaphorically speaking, the barrier won't open until all processes have arrived there. Traditionally, barriers are used to separate phases of computation, for example when intermediate results have to be exchanged. Since MPI's more sophisticated collective communication operations provide rendezvous functionality, as well as data transfer, basic barrier synchronization is primarily used for debugging purposes.

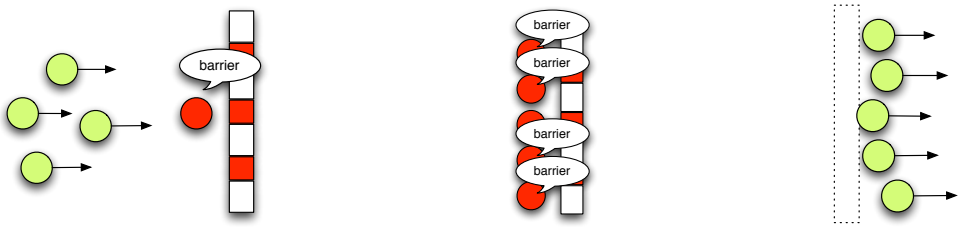

Figure 2: Barrier

Broadcast

The broadcast operation broadcasts a message from one process (the *root* process) to all processes in the communicator. Like every collective function, it returns once it has been called by all processes in the communicator, on return, all the processes have root's message in the buffer specified in their call of the broadcast function (Fig. 3).

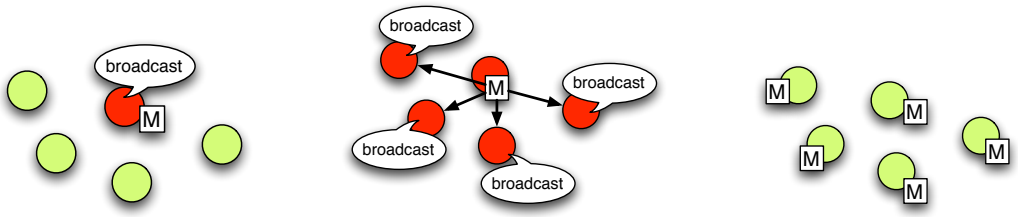

Figure 3: Broadcast

Scatter/Gather

The scatter/gather operations are used to distribute data to processes (*scatter*) and collect data – usually results – from processes (*gather*). The basic operations scatter data from the root process buffer to all the processes, and gather data from all processes to a buffer in the root process. MPI also provides more sophisticated scatter/gather operations: Gather-to-all being similar to gather, but distributing the buffer with the gathered data to all processes, and an all-to-all gather/scatter operation that allows each process to send and receive distinct data to and from each other process.

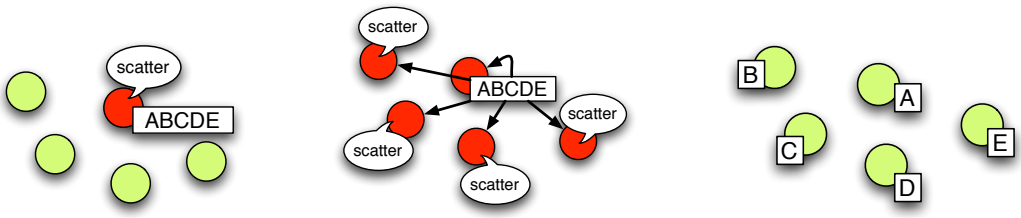

Figure 4: Scatter

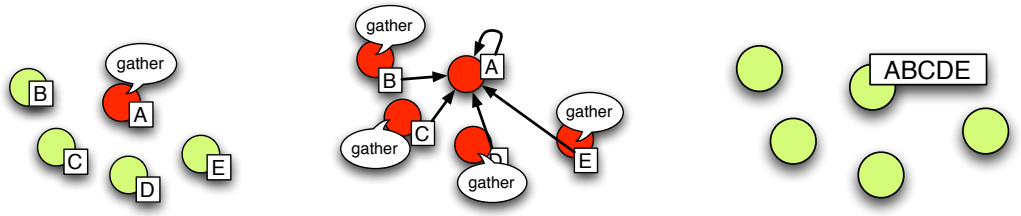

Figure 5: Gather

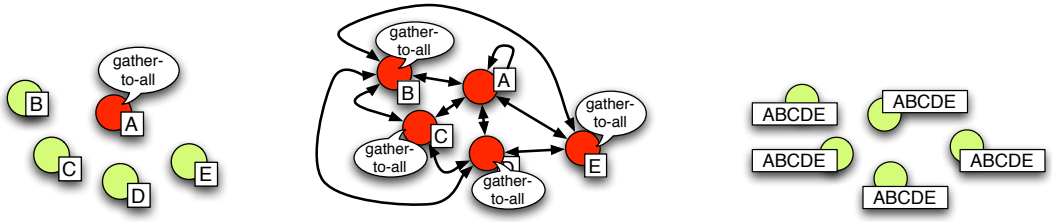

Figure 6: Gather-to-all

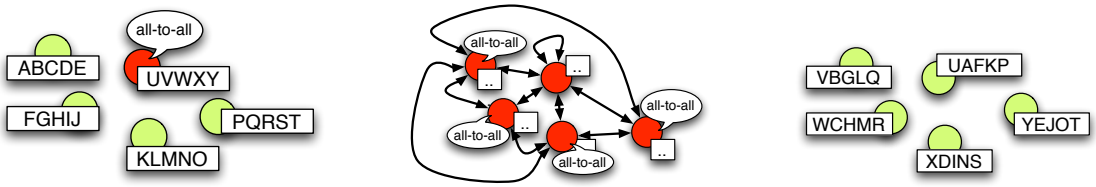

Figure 7: All-to-all

Reduction

Reduce operations apply a function to values gathered from processes. The function can be one of the built-in functions such as MIN, MAX, and SUM, or a user-defined function.

2.8 Virtual Topologies

Since most parallel programming problems are inherently multi-dimensional, it would be convenient to be able to reflect this multi-dimensionality in the process naming, for example using coordinates in a multi-dimensional grid, instead of sequential ranks. Also, if we had a way to reflect the topology of our problem in our MPI program, sophisticated MPI implementations could use this information to optimize the assignment of processes to physical nodes.

The MPI standard specifies a way to store this *virtual topology* information through additional communicator attributes, as mentioned in section 2.4. There are two kinds of virtual topologies defined by MPI:

Graph Topologies assume communication to be between neighboring nodes in the graph, MPI provides convenience functions to determine the neighbors of a process.

Cartesian Topologies allow for cartesian structures of any dimension. Convenience functions for converting between cartesian coordinates and ranks are provided.

3 MPI-2

MPI-2, the second major version of the MPI standard, was released in 1997 [8]. It incorporates a new version of the MPI-1 standard (1.2) containing additional clarifications, error corrections, and also a function for retrieving the version number of the MPI standard used. It is the first MPI version that defines official bindings for Fortran-90 and C++. New functionality provided by MPI-2 includes one-sided communication (remote memory access), dynamic creation and termination of processes, and provisions for parallel I/O. Each of those newly included topics will be presented in this chapter.

3.1 One-Sided Communication

Until MPI-2, all communication in MPI required at least two processes to actively participate in the communication. One-sided communication in MPI-2 allows processes to access each others memory using the two primitives `MPI_PUT` and `MPI_GET` without active participation from the process whose memory is being accessed (*target* process). As a prerequisite for one-sided communication, all processes in a communicator must

make a part of their memory (a *window*) available for access by other processes. This is done through a collective call, on return, all processes have made a window available and know about the other processes' windows.

Since one-sided communication is a deviation from the core message passing paradigm, explicit synchronization is needed. MPI-2 distinguishes between *active* and *passive target* communication, which use different synchronization mechanisms.

Active Target Communication

- The first form of active target communication allows a process (*origin*) to access the memory of another process (*target*). The target process is actively involved in the remote memory access. To gain access, the origin process calls `MPI_WIN_START`, which blocks until the target process allows access using `MPI_WIN_POST`. The target process ends the *access epoch* using `MPI_WIN_WAIT`, which will block until the origin process voluntarily gives up access through `MPI_WIN_COMPLETE` (Fig. 8).

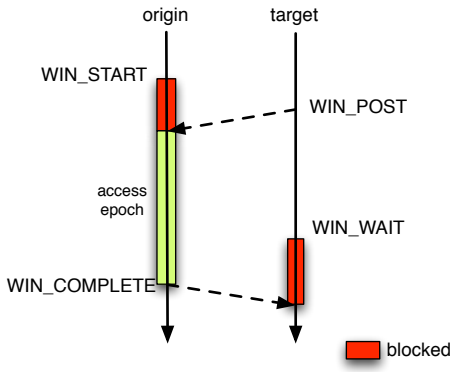

Figure 8: Active Target Communication

- There is also `MPI_WIN_FENCE`, a collective call that allows all processes in the communicator to access each other's memory between two calls of `MPI_WIN_FENCE`.

Passive Target Communication

In passive target communication, the target process is not involved at all in the synchronization. Processes that want to access the memory of another process gain access using `MPI_WIN_LOCK` and give up the lock using `MPI_WIN_UNLOCK`. If a process already holds a lock for a window, another process' attempt to acquire a lock for that window will either block

(as shown in Fig. 9), or a subsequent `MPI_PUT` will block until the lock can be obtained – it is up to MPI implementors to decide which method to use.

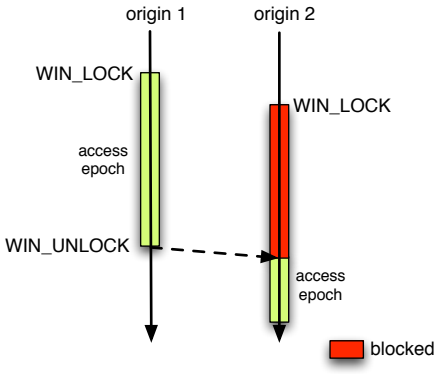

Figure 9: Passive Target Communication

3.2 Dynamic Processes

So far, the number of processes of an MPI implementation had to be specified on startup and remained static until the application terminated. MPI-2 provides ways for a running MPI application to create and terminate processes, and to establish communication to non-related processes. For these tasks, the underlying process management system is being used. It is even possible to access specific properties of the process management system using a special argument (*info*), though this opportunity should be used with caution since it compromises portability of the MPI application.

Dynamic processes are an important application for inter-communicators, all process creation operation return a new inter-communicator to establish communication to the newly created process(es). The new process(es) can obtain an inter-communicator to communicate to the processes that created it/them using `MPI_COMM_GET_PARENT`.

3.3 Parallel I/O

In parallel applications files are accessed concurrently, and the data in one file may be shared by many processes that need to read and write non-contiguous pieces of the file – coordination of that access is something demanded of a parallel programming library. Also, parallel applications should be able to influence lower-level parallel I/O mechanisms such as RAID striping.

A part of MPI-2 also known as "MPI-IO" makes provision for both of these demands by means of *views* and *hints* (Fig. 10). Hints are similar to the "info argument" with dynamic processes, they allow applications to influence, for example, how a file is striped on a RAID system. Views enable parallel processes to concurrently read and write non-contiguous pieces of a file, while each having their own, contiguous view of that file.

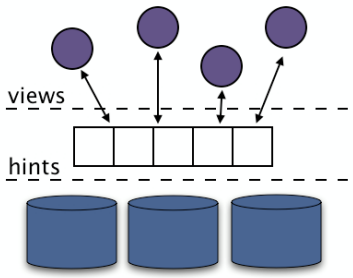

Figure 10: Parallel I/O and MPI-2

Using the same mechanisms known from derived datatypes, MPI-2 lets each process have its own, contiguous view of non-contiguous pieces of a file. An *etype* defines the datatype of entries in a file, while the *filetype* defines a "filter" on the sequence of etypes as illustrated in figure 11.

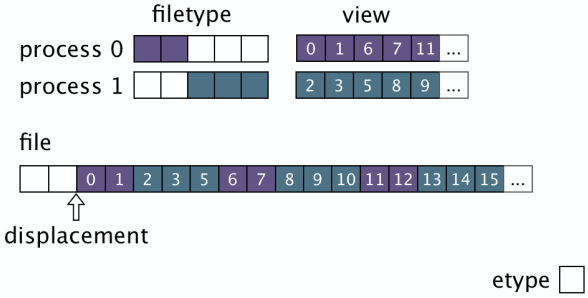

Figure 11: Views

In addition to views and hints, MPI-IO also provides support for shared and individual file pointers, blocking and non-blocking/split-collective synchronization, as well as individual and collective file access operations.

3.4 Other Developements

Part of the MPI-2 standardization process was the development of a standard for MPI in realtime environments. This standard, MPI/RT, ultimately

was not included in the MPI-2 standard, but is now a separate standard currently available in version 1.1 [11]. Also not included were features like split collective operations (similar to non-blocking point-to-point routines) and dynamic processes creation without establishing communication [9].

4 MPI Implementations

The MPI standardization process was characterized by close and productive cooperation of vendors and researchers, leading to a widespread acceptance and numerous available implementations. There are vendor-supplied MPI implementations, for example from Sun, SGI, HP, and NEC, as well as commercial implementations like ChaMPIon/Pro and WMPI. Universities and government laboratories also have developed open source MPI implementations, such as MPICH and LAM/MPI.

All of the available MPI implementations implement all or almost all of MPI-1. In contrast, complete MPI-2 implementations are still hard to find, even though MPI-2 has been standardized seven years ago.

In the following sections, we look at four major MPI implementations, and attempt a comparison of features: MPICH, LAM/MPI, ChaMPIon/Pro and Sun MPI.

4.1 LAM/MPI

LAM/MPI [5] is an open source MPI implementation that was first developed at the Ohio Supercomputing Center, maintained by the University of Notre Dame and now by Indiana University. It implements all of MPI-1 and almost all of the MPI-2 standard. Most POSIX platforms are supported.

LAM/MPI uses daemons for the runtime environment, and authentication/remote execution is based on the well-established RSH and SSH programs. Since the LAM runtime environment is "booted" independently from the start of an MPI application using `mpirun`, the actual startup of an MPI application is faster than on other platforms.

Newer versions of LAM/MPI offer a modular architecture through the System Services Interface (SSI). This mechanism currently allows for replacement of the LAM daemons (boot), collective operations (coll) and low-level point-to-point communication (rpi), an important prerequisite to allow customization for particular networking hardware. Also, it is possible to add checkpoint/restart (cr) functionality through SSI.

4.2 MPICH

MPICH [1] is another open source MPI implementation. It was developed alongside the standard to allow the MPI Forum to evaluate the viability of their ideas. The most current version provides a complete MPI-1 implementation and the MPI-IO portion of MPI-2. Supported platforms include most Unix flavors and also Windows NT.

MPICH's architecture (Fig. 12) is divided into two parts, a platform independent MPI implementation, the MPICH layer, and the platform-dependent Abstract Device Interface (ADI) implementation. To port MPICH to a new platform, only the ADI implementation needs to be exchanged, the MPICH layer depends only on the ADI and is completely independent of its implementation. This architecture has lead to the development of numerous successful MPICH derivates, such as

- MPICH-V, a fault-tolerant MPI implementation
- MP-MPICH, an MPI implementation for heterogeneous clusters
- MVAPICH, an implementation with support for the native Verbs Level Interface (VAPI) of InfiniBand
- MPICH-G2, a grid-based MPI implementation

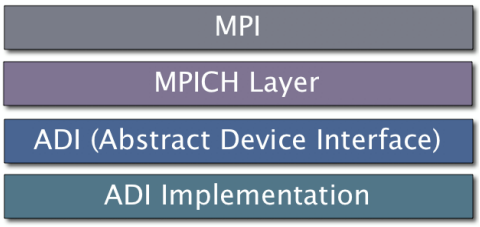

Figure 12: MPICH Architecture

MPICH-G2

Some of the properties of grids, like widely varying hardware and software platforms (with different process management, filesystems, etc.), diverse network conditions and the need for cross-site authentication pose new challenges to an MPI implementation. MPICH-G2 [4] faces those challenges through an ADI implementation that is based on the Globus Toolkit, a widely used toolkit for grids, providing ressource allocation (Globus Ressource Allocation Manager, GRAM), authentication (Grid Security Infrastructure, GSI), node and service discovery (Monitoring and Discovery Service, MDS), and I/O capabilities (Global Access to Secondary Storage, GASS).

Since the physical topology of the grid may have a significant impact on performance (for example, depending on whether communication happens in a LAN or a WAN), MPICH-G2 uses topology information in communicator attributes to optimize collective operations, and uses vendor-supplied MPI (vMPI) when possible, instead of Globus Communication for TCP.

Future MPICH Development – MPICH2

All new development effort for MPICH focuses on a complete re-implementation, MPICH2 [2]. Currently available as a beta, MPICH2 already provides a complete MPI-1 implementation and some parts of MPI-2 (all of MPI-IO and preliminary one-sided communication), as well as limited device support (TCP sockets and shared memory).

4.3 Sun MPI

Sun Microsystems provide their own open-source MPI implementation [12] for clusters running Solaris 8 and 9. It implements MPI-1 and almost all of MPI-2 and comes with an extensive software environment:

- Sun Parallel File System
- Prism, a debugger and performance analyzer
- S3L, the Scalable Scientific Subroutine Library
- Sun Cluster Runtime Environment
- Cluster Console Manager

Sun's MPI implementation is heavily tuned for use on Sun systems.

4.4 ChaMPIon/Pro

ChaMPIon/Pro is a commercial MPI implementation that lists full MPI-2 support as one of its features. According to material on the product website [10], it is currently supported on Red Hat Linux on the IA32 architecture, and SUSE Linux on AMD Opteron systems, with support for InfiniBand and Myrinet networking hardware provided on both platforms. Support for additional platforms seems to be planned.

A closed-source commercial implementation, ChaMPIon/Pro is priced at about 300-400 USD per CPU, and requires the purchase of a one-year support and maintenance contract. The same company that develops ChaMPIon/Pro also offers an MPI-1 only implementation with fairly extensive operating system support (Linux, Windows, Mac OS X).

4.5 Comparison

Figure 13 shows an overview of the MPI implementations presented in this report. Full support is indicated by a check mark, partial support by a dot, and no support by a cross.

	MPICH	LAM/MPI	Sun MPI	ChaMPIon/ Pro
MPI-1	✔	✔	✔	✔
MPI-2	•	•	•	✔
- one-sided comm.	✘	•	•	✔
- parallel I/O	•	✔	✔	✔
- dynamic processes	✘	✔	•	✔
- C++ / Fortran 90	•/•	✔/✘	✔/✔	✔/✔
IMPI Support	✘	✔	✘	✘
Grid Capabilities	✔	✔*	✔	✘
Debugging Facilites	✔	✔	✔	✔

* beta

Figure 13: MPI Implementations – Comparison

In addition to the support for features of the MPI standard (for details see the previous sections), this table indicates the support of implementations for Interoperable MPI (IMPI), a standard that enables MPI applications to run across different MPI implementations simultaneously, support for grid computing, and for parallel debuggers.

5 Outlook

5.1 Graph-Oriented Programming

In a recent paper [3], graph-oriented programming is presented as a way to alleviate some of the difficulties in developing message passing application by providing a high-level abstraction (Fig. 14). A complete development environment (Visual GOP) supplements the proposed approach (Fig. 15).

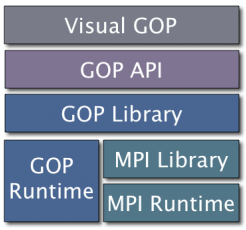

Figure 14: GOP Architecture

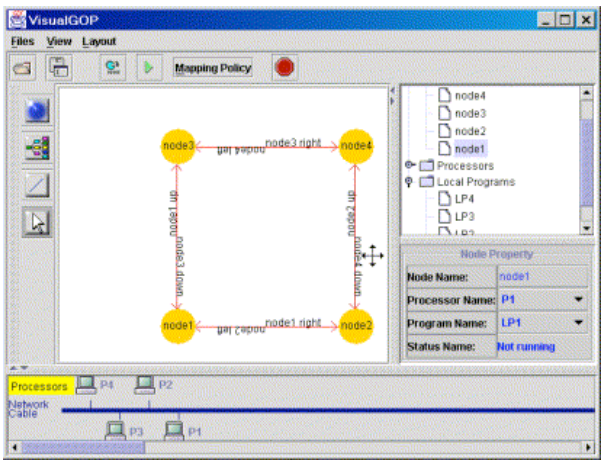

Figure 15: Visual GOP – screenshot taken from [3]

According to this approach, a program consist of a graph construct describing the relationships between local programs, a mapping of local programs to graph nodes, and an optional node-to-processor mapping. The authors define their own GOP API with the aim of simplifying the MPI interface by reducing the number of parameters, and replacing communicators with node groups, which identify processes through node IDs instead of rank and communicator. GOP makes active use of graph topology information and allows for modification of the graph topology at runtime.

5.2 The Future of MPI

Currently, there is no visible effort towards a new version of the MPI standard. MPI-2's functionality is not yet exhausted, and available implementations are not yet as widely available as one would expect for a standard that has been available for seven years. Also, most MPI developers are not programmers, but scientists in an application field, so their interest is not in a highly sophisticated programming environment, but in one that works and that they are accustomed to.

Nevertheless, there are significant developments surrounding the MPI standard: The increasing popularity of grid computing poses new challenges implementations like MPICH-G2 are trying to meet. Large clusters require excellent scalability from MPI impementations. Faster networking hardware demands highly customized versions of the lower layers of MPI implementations, there is, for example, not yet an MPI implementation that completely exhausts the InfiniBand VAPI. And of course there is need for more efforts like GOP, aiming to provide better development environments and abstractions for MPI programming.

References

[1] ARGONNE NATIONAL LABORATORY. MPICH website. http://www-unix.mcs.anl.gov/mpi/mpich/.

[2] ARGONNE NATIONAL LABORATORY. MPICH2 website. http://www-unix.mcs.anl.gov/mpi/mpich2/.

[3] CHAN, F., CAO, J., AND SUN, Y. High-level abstractions for message-passing parallel programming. *Parallel Computing 29* (November 2003).

[4] KARONIS, N., TOONEN, B., AND FOSTER, I. MPICH-G2: A grid-enabled implementation of the message passing interface. *Journal of Parallel and Distributed Computing 63* (May 2003).

[5] LAM TEAM. LAM/MP website. http://www.lam-mpi.org.

[6] MESSAGE PASSING INTERFACE FORUM. MPI: A message-passing interface standard. http://www.mpi-forum.org/docs/mpi-10.ps, 1994.

[7] MESSAGE PASSING INTERFACE FORUM. MPI: A message-passing interface standard. http://www.mpi-forum.org/docs/mpi-11.ps, 1995.

[8] MESSAGE PASSING INTERFACE FORUM. MPI-2: Extensions to the message-passing interface. http://www.mpi-forum.org/docs/mpi-20.ps, 1997.

[9] MESSAGE PASSING INTERFACE FORUM. MPI: Journal of development. http://www.mpi-forum.org/docs/mpi-20-jod.ps, 1997.

[10] MPI SOFTWARE TECHNOLOGY. ChaMPIon/Pro website. http://www.mpi-softtech.com/products/cluster/champion_pro/.

[11] MPI/RT FORUM. MPI/RT website. http://www.mpirt.org.

[12] SUN MICROSYSTEMS. Sun HPC ClusterTools website. http://www.sun.com/software/hpc/overview.html.