MPI Implementation for Windows NT

Using the Channel Device Interface for reasonably efficient WinNT device implementation

by

Boris V. Protopopov.

Outline

- *Introduction*
- *Insight into the architecture of MPI software package*
- *MPID Channel Interface as a starting point of MPI implementation for Windows NT*
- *Possible extensions and improvements of currently used MPI software*
- *Conclusions*

Introduction

- Background info about MPI and message passing
- Computational model used by MPI
- Basic assumptions and abstractions provided for the programmer in message-passing model of IPC

Mississippi State University / ICDCR Lab 5/22/96 3

Message passing is one of the two conceptual models of inter-process communications (IPC). Communicating processes are identified by their global rank (usually an integer ID); this allows each process to send some data to any other process using this identifier. Each processor has its own address space and can communicate only by explicitly sending and receiving data. No assumptions can be made about the relative speed of execution of different processes, that is why some means of global synchronization is required. A global barrier function ensures that all processes have completed some phase of their calculations at some particular moment in time (specifically, when all the processes execute the barrier function). It is also important to make sure that a message passing system guarantees reliable data transfer (which means error-free, in-order data transfer without data loss or duplication) because one could not guarantee correct operation of message passing system otherwise.

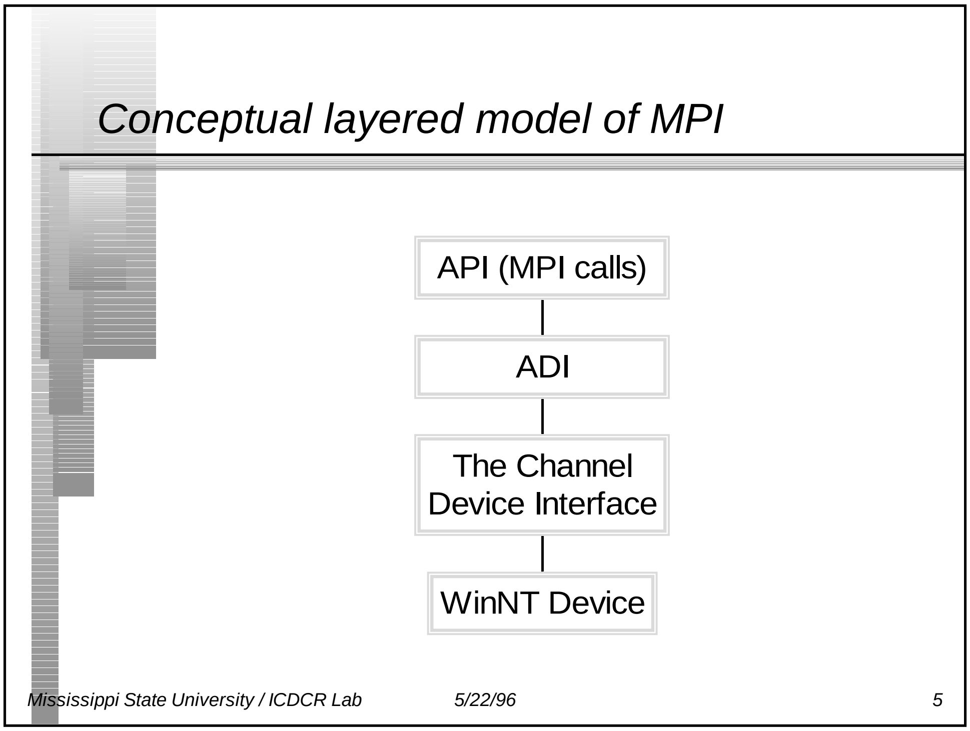
MPI is a message-passing library specification which provides portable and reasonably efficient message-passing API. MPI does not make any restrictive assumptions about the underlying device architecture provided that some means of remote process execution and IPC are available. Application programmers can send and receive data between processes (point-to-point operations) synchronize them (barrier), and perform collective operations (reduce, allreduce, gather, scatter, etc. functions); reliable, in-order data transfer is guaranteed.

Insight into the architecture of the MPI software package

- *Conceptual layered model of MPI*
- *Hierarchy of the existing MPI implementations*
- *Tradeoffs of implementing different layers (performance vs. portability and time investments)*

Mississippi State University / ICDCR Lab *5/22/96* *4*

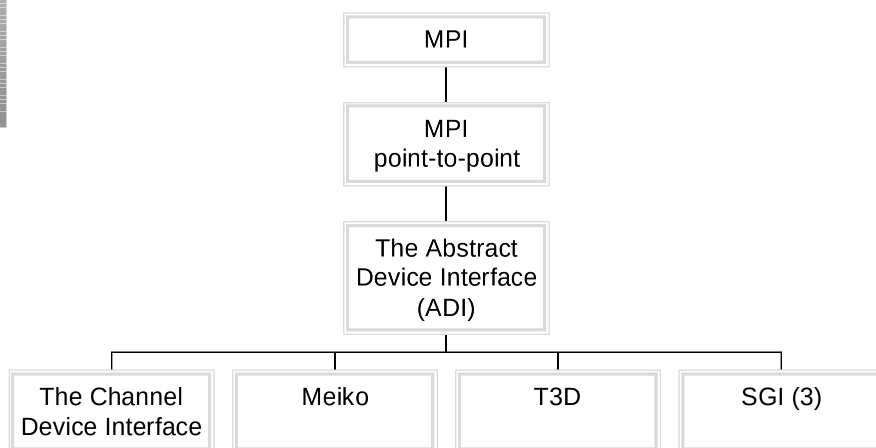
MPI has layered architecture. In particular, it has three layers: API (application programmer interface), ADI (abstract device interface), and device (the implementation of the set of ADI calls on a particular platform) itself. All these layers communicate with each others through appropriate service entry point to access some data or perform some operations. As for the device layer, there is an abstraction called Channel Device which conceptualizes a set of services which a device should provide to the upper ADI layer. Channel interface, in turn, has a set of lower level calls which should be implemented on a particular architecture.



The Channel device interface makes some specific assumptions about the nature of basic communication operations because of requirements of the ADI. In particular, all messages are sent in one or several packets of constant predefined length. Every packet consists of the header containing control information (source/destination address, packet type, communication mode, current communicator, packet tag, length of the data portion, etc), and the data part of the packet. Depending on the length of the data, there are “short” and “long” messages. “Short” messages are being sent in one packet which includes both control and data, and “long” messages are sent in a series of packets. The first packet of the series contains control information, and all others contain data along with some small amount of identification information.

Hierarchy of the existing MPI implementations

Upper layers of MPICH (redrawn from Gropp et al. 1995).



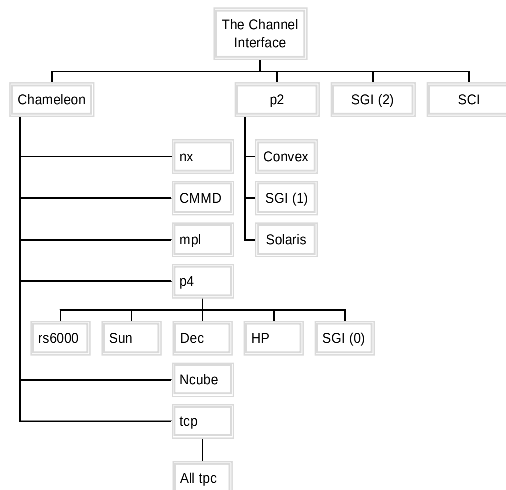
Mississippi State University / ICDCR Lab *5/22/96* *6*

Due to the MPI architecture, a programmer who wants to implement MPI for a particular platform can start at any layer of the architecture and either directly implement services which ADI provides for API, or write his own device, or chose to take advantage of existing Channel Device and implement a small set of very basic communications primitives, probably trading some efficiency for the time required to implement and test the new device.

The example is SGI implementations which appear on different layers of the MPI implementations hierarchy.

Hierarchy of the existing MPI implementations

Lower layers of MPICH (redrawn from Gropp et al. 1995).



MPID Channel Interface as a starting point of MPI implementation for Windows NT

- *Time constraints; good point to start without major performance degradation*
- *Major channel operations*
- *Auxiliary operations*
- *Multithreaded implementation of channel operations:*

  -separation of channel operations and direct shared memory handling (issues involved: additional buffering vs. possible overlaps of computations and communication operations and more fair receive policy)

  -synchronization problems with multiple writers / readers (dynamic variation of numbers of the communicating threads)
- *Basic channel operations in WinNT device*

Mississippi State University / ICDCR Lab 5/22/96 8

Application programmers use send - receive operations;

send (buffer, count, type, tag, context, dest);

recv (buffer, count, type, tag, context, source, &status);

These operations can be of different types:standard, non-blocking, synchronous, ready.

The device is supposed to implement them with the use of :

copy a buffer in the other process’s address space;

copy into a buffer from the other process’s address space;

receive an acknowledgment about the completion of the operation or an error message.

There should be some transport mechanism to communicate between processes. In case of distributed system it is a communication protocol (TCP/IP for example); in case of shared memory we use it to communicate. It is also very beneficial if the device can perform asynchronous communications operations such as asynchronous sends and receives.

It allows an application programmer to overlap communications and computations where possible. Naturally, if the device itself cannot support such operations, there is nothing we can do to help it. In particular, interrupt-driven sends/receives supported on some platforms, asynchronous sends/receives on sockets provided by some operating systems, or interprocess syncronization promitives (signals in Unix, events in WindowsNT) can be used to allow for asynchronous communications.

Major channel operations

- BOOL MPID_ControlMsgAvail (void)
- void MPID_RecvAnyControl (MPID_PKT_T* pkt, int size, int* from)
- void MPID_SendControl (MPID_PKT_T* pkt, int size, int dest)
- void MPID_RecvFromChannel (void* buff, int maxsize, int from)
- void MPID_SendChannel (void* buff, int size, int dest)

Mississippi State University / ICDCR Lab 5/22/96 9

The MPID_ControlMsgAvail (void) function returns a boolean value indicating whether a control packet is available from any source. It scans local pool of buffers in hopes to find some data.

The MPID_RecvAnyControl (MPID_PKT_T* pkt, int size, int* from) function receives any control packet. The arguments are as follows:

pkt - the address of the buffer in the local memory where the process will copy the contents of a shared memory buffer before releasing it;

size - the size of the data in the packet;

from - an output parameter indicating the rank of the source of the packet.

It is important to notice that this function blocks until it receives something.

The MPID_SendControl (MPID_PKT_T* pkt, int size, int dest) function sends a control packet. The arguments are:

pkt - the address of the packet to send;

size - size of the data in the packet;

dest - the rank of the destination process.

It is also important that this function is non-blocking by its nature. Indeed, it is supposed to be used for every send - blocking or non-blocking.

Auxiliary operations.

- *void MPID_IRecvFromChannel (void* buff, int size, int partner, int request)*
- *void MPID_WRecvFromChannel (void* buff, int size, int partner, int request)*
- *void MPID_RecvStatus (int request)*
- *void MPID_ISendChannel (void* buff, int size, int partner, int request)*
- *void MPID_WSendChannel (void* buff, int size, int partner, int request)*
- *void MPID_TSendChannel (int request)*

Mississippi State University / ICDCR Lab 5/22/96 10

So far we dealt with “short” packets which included their data. But, as it was mentioned above, there are “long” packets which also should be processed.

The MPID_RecvFromChannel (void* buff, int maxsize, int from) receives “long” data portion of the long message which control packet was previously received. The arguments are:

buff - the pointer where to put the long data;

maxsize - the maximum size of the data which can be received without truncation;

from - the rank of source process.

This function also blocks until the message is received entirely.

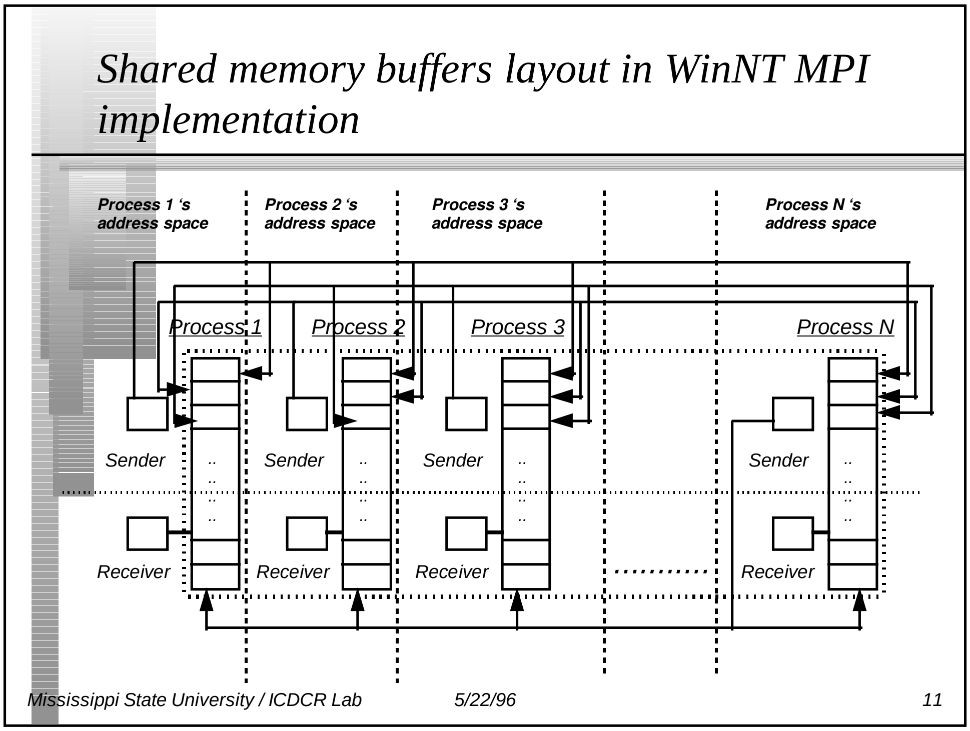
The MPID_SendChannel (void* buff, int size, int dest) function send long data to the other process. The arguments are:

buff - the buffer in local memory to send;

size - the size of the buffer to send;

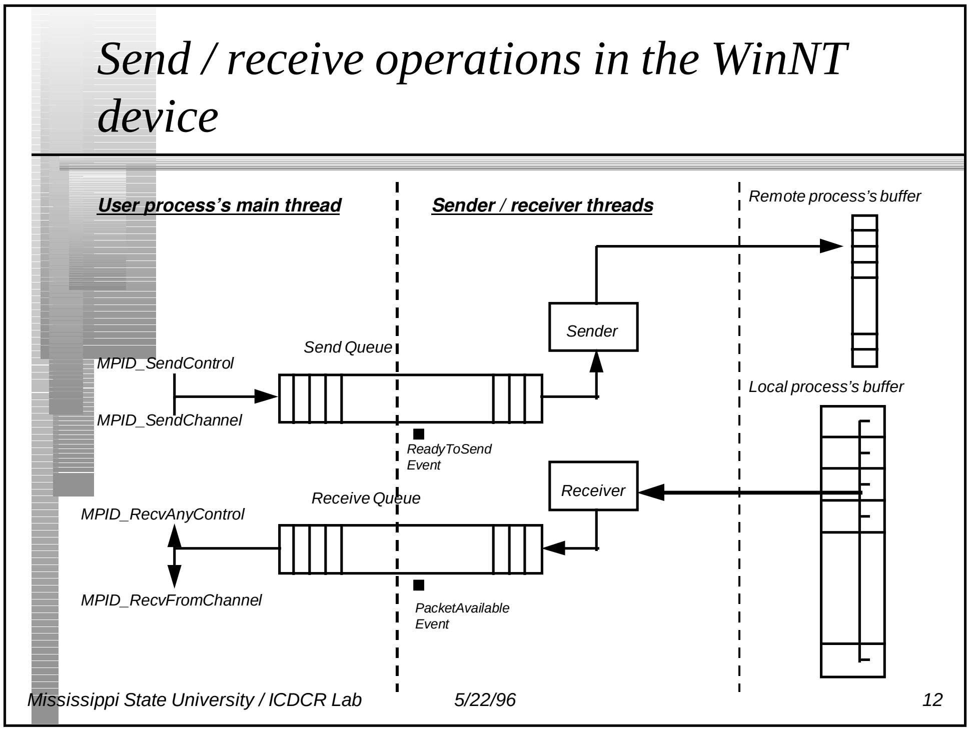
dest - the rank of the destination process.

These are the functions which provide a “bare minimum” for the Channel device to use to provide all unrestricted set of services for the ADI and API to operate correctly.



As for the MPI implementation for Windows NT 3.5* which is now in progress in Mississippi State University, ICDC Research Lab, by Shane Herbert and Boris Protopopov, the current choice is to start with Channel device implementation because of time constraints and some software engineering considerations. It should be mentioned that we are currently finished our work on the Channel device for one box case and are performing last tests of the TCP device for inter-box implementation. The one-box case is a shared memory implementation which makes use of memory-mapped files provided by the operating system. Every process obtains a pool of shared memory buffers for every other process to be able to send some data into these buffers. All buffers are currently visible to all processes. So, if process i wants to send something to the process j, it takes j-th buffer pool and puts its data into the i-th buffer. Such a simple but useful setup provides full-duplex point-to-point communications.

The Channel device Interface imposes some restrictions on the design of the communication subsystem and synchronization scheme of the device. Indeed, every send operation is implemented as either one MPID_SendControl call (short send) or MPID_SendControl followed by MPID_SendChannel call (long send). Long sends cannot be interrupted by any other operation which potentially can get in between or



MPID_SendControl and MPID_SendChannel calls. On the other hand, if the receiving process is completely occupied by servicing a communication channel for the time of the long operation, several other short sends can wait for the service on the other connections, and it will be unfair to serve the long operation and place the rest of the communication requests on hold.

To overcome these problems, one can add a thin additional layer below the Channel device which would accept / deliver packets from / to the Channel device, queue them and asynchronously send / receive to /from their destinations. This design is currently implemented for Windows NT MPI device.

In particular, there are two separate threads acting on behalf of every communicating process (in addition to the main, user, thread); one receiver and one sender. There are also two queues (send queue and receive queue accordingly) where the threads mentioned above place messages which or should be sent or retrieve messages which are received. The operations over these queues are thread safe, so any number of parallel threads can enqueue or dequeue messages which is equivalent to sending or receiving them form the point of view of the Channel device. The sender and receiver threads mentioned above actually work with shared memory buffers and ensure all desired properties of asynchronous communication operations, such as fair service and in-order communication.

Possible extensions and improvements of currently used MPI software

- *Collective operations*
- *Platform-independent object-oriented MPI implementation (MPI++, Java)*
- *Load-balancing for applications using MPI; real-time MPI*

Mississippi State University / ICDCR Lab   5/22/96   13

To eliminate busy waiting, the events mechanism provided by the Windows NT operating system is used.

A number of possible extensions and improvements of currently used MPI software are being considered. One can implement collective operations on the device level and make them more efficient in comparison with current point-to-point implementation. One can also write MPI completely (or almost) on C++ and then use polymorphism to provide run-time support for communications in heterogeneous environment (virtual methods, dlls).

It is very interesting to augment current MPI software with load-balancing capabilities and consider the issues involved in transformation of the currently used MPI into the real-time software.

Most of these extensions are complex tasks and can easily be cast into independent research projects. As for the collective operations, this is, probably, the closest step toward improving efficiency of MPI software we are going to address along with implementing multithreaded ADI.

Conclusions

- MPI is a message-passing library specification which provides portable and reasonably efficient message-passing API
- Multithreaded, platform-independent object-oriented MPI implementations, device support for collective operations, dynamic process management are some of many extensions which are currently under the consideration

Selected Bibliography:

Custer, Helen. 1993. Inside Windows NT. Washington: Microsoft Press.

Foster, Ian. 1995. Designing and building parallel programs. Addison-Wesley Publishing Company.

Gropp, William, Ewing Lusk, and Anthony Skjellum. 1994. Using MPI: portable parallel programming with the Message Passing Interface. Cambridge, Massachusetts: MIT Press.

Gropp, William, and Ewing Lusk. MPICH ADI implementation reference manual. ftp://info.mcs.anl.gov /pub/mpi/ adiman.ps (18 Dec.1995).

Gropp, William, and Ewing Lusk. MPICH working note: creating a new MPICH device using the Channel Interface. http://www.mcs.anl.gov /home/lusk/mpich/workingnote/newadi/ note.html (18 Dec.1995).

Gropp, William, Ewing Lusk, Anthony Skjellum, and Nathan Doss. MPICH - a high-performance, portable implementation for the MPI Message-Passing Interface. http://www.mcs.anl.gov /home/lusk/slides/mpich/ talk.html (19 Dec.1995).

Richter, Jeffrey. 1995. Advanced Windows: the developer's guide to the Win32 API for Windows NT and Windows 95. Washington: Microsoft Press.

The MPI Forum. The MPI Message Passing Interface standard. http://www.mcs.anl.gov /mpi/mpi-report/ mpi-report.html (20 Nov.1995).