A multi-threaded Message Passing Interface (MPI) architecture: performance and program issues[*]

Boris V. Protopopov[$]

Department of Computer Science
High-Performance Computing Laboratory
Mississippi State University
e-mail: boris@cs.msstate.edu

Anthony Skjellum[$$]
Department of Computer Science and NSF Engineering Research Center
Mississippi State University
P.O. Box 9637
Mississippi State, MS 39762
e-mail: tony@cs.msstate.edu

[*]Support provided by DARPA under order D350 is acknowledged, through contract from the US Air Force, Research Laboratory, F30602-95-1-0036. Additional support is acknowledged from the National Science Foundation, under the Career Award Program, ASC-9501917. Partial support by the MSU Office of Research is also gratefully acknowledged.

Submitted to Journal of Parallel and Distributed Computing, July 30, 1997.

Resubmitted with corrections on June 19, 1998.

[$]Presently at Mercury Computer Systems, Inc., Chelmsford, MA

[$$]Corresponding author

A multi-threaded Message Passing Interface architecture

Address correspondence to:

Anthony Skjellum
Mississippi State University, Engineering Research Center
P.O. Box 9637
Mississippi State, MS 39762, USA
Tel: (601)-325-8453.
e-mail: tony@cs.msstate.edu.

Abstract

This paper discusses a multi-threaded software architecture for Message-Passing Interface (MPI) software specification. The architecture is thread-safe, allows for concurrent communication over several communications media (multi-fabric communication), efficiently utilizes available hardware concurrency over a wide range of target platforms, and allows for concurrent communication and computation within the limits imposed by the hardware.

The architecture is developed in the framework of the MPICH software architecture, a well-known MPI implementation used worldwide. The proposed architecture adopts wide portability of the MPICH design and remedies some of its deficiencies such as inefficient multi-fabric communication and non-thread-safety. The paper also considers the issues concerning development of high-performance portable message-passing systems for general-purpose architectures.

The contributions of the paper are: improving architecture and addressing thread safety of modern reliable messaging software, as well as identifying and taking advantage of inherent concurrency in the message-passing software itself.

Key words: *MPI, thread-safety, multi-threaded design, multi-fabric communication, concurrency in message-passing*

1. Introduction

MPI is a de-facto standard for message-passing software [17] used for developing high-performance portable parallel applications [4]. The MPI specification has been implemented for a wide range of computer systems from clusters of networked workstations running general-purpose operating systems (UNIX, Windows NT) to high-performance computer systems such as CRI T3E, CRI C90, SGI Power Challenge, Intel Paragon, IBM SP1, SP2, and so on [3]. Among various MPI implementations, the MPICH implementation developed by Argonne National Laboratory and Mississippi State University [3] is known as the most portable implementation that allows one to achieve reasonably good message-passing performance without extensive porting efforts and platform-specific tuning. Because of this *performance portability*, this implementation has served as a basis for many of the above-mentioned implementations.

The MPICH layered software architecture and clean abstracted interfaces between the layers [5, 6, 7, 8, 9] are the features that ensure *performance portability*. We use MPICH as a framework for developing our multi-threaded architecture and as a reference point for evaluating our design in order to preserve these valuable design features.

The presented architecture meets the following design criteria. First, the architecture is generic (it targets a general-purpose time-sharing computer system), and it provides reliable and ordered message-passing with uniform communications progress. Second, the architecture allows for concurrent multi-fabric communication. Third, the architecture allows for concurrent computation and communication. Therefore, the architecture is capable of utilizing available hardware concurrency over a wide range of target platforms. Finally, the architecture is thread-safe, and, therefore, allows multi-threaded applications to take advantage of the MPI message-passing services.

We believe that these criteria can be met by making our architecture multi-threaded. Running computation and communication over each fabric in separate threads allows us to satisfy the second and third design criteria. Since the architecture itself is multi-threaded, it is supposed to be safe for the threads implementing the architecture. It is also not difficult to make it thread-safe for applications and, thereby, to satisfy the fourth criterion. On the other hand, as we show later in the paper with MPICH as an example, single-threaded design forces round-robin switching between different communication fabrics in order to ensure uniform communication progress of all communication requests, which has affect of averaging communication latency over all the fabrics. Also, overlapping communication and computation and making the design thread-safe requires more efforts in singe-threaded case.

We also discuss the issues that arise while designing and implementing high-performance message-passing systems for general-purpose architectures. We propose to rely on widely available standard features for utilizing fine-grain parallelism and achieving overlapping of communication and computation, such as POSIX-compliant thread libraries with their means of thread scheduling and synchronization [10, 12]. Our design actively takes advantage of these standard features and, hence, remains portable, while potentially delivering better communication performance.

We expect that readers are familiar with message-passing programming model [4] and the MPI message-passing standard [17]. Later in the paper, we provide a brief review of the overall MPICH design and direct interested readers to [6, 7, 8, 9, 14] for more information about the MPICH software architecture. We refer to the MPICH design and implementation at the time of writing the paper, and improvements are forthcoming over time[1].

The rest of the paper is structured as follows. In Section 2, we briefly review the MPICH design and consider the capabilities and deficiencies of the MPICH software architecture. Then, in Section 3, we discuss the proposed multi-device thread-safe MPICH design and consider how it overcomes the disadvantages of the original design. In Section 4, we discuss problems arising while implementing high-performance message-passing systems for general-purpose architectures. Finally, we summarize the results discussed in the paper and outline further interesting directions of MPI design development in Sections 5 and 6.

2. Capabilities and deficiencies of the MPICH software architecture

The MPICH implementation is a portable and efficient implementation of the MPI standard, used worldwide. MPICH was developed by Argonne National Laboratory and Mississippi State University [3]. Having passed through several revisions and architectural updates, it delivers good message-passing performance. MPICH has layered software architecture (Figure 1), which is the foundation of its portability [3, 6, 7]. The first (highest) layer incorporates the MPI API, high-level message-passing logic, and implements user-level MPI abstractions such as topologies, data types, and communicators. The second layer, Abstract Device Interface (ADI), includes a generic message-passing engine and defines an abstract set of middle-ware services that are required in order to support the upper-level's functionality. The third layer, Device, incorporates communication protocol modules. It implements the ADI services for a particular platform. A part of the Device layer, Channel Device Interface (CDI), includes a small set of basic message-passing routines that further abstract low-level communication services [6, 7]. Any of the architectural layers can be chosen as a portability

layer. However, implementing CDI for a target platform is the most time-efficient approach that often proves to have competitive message-passing performance.

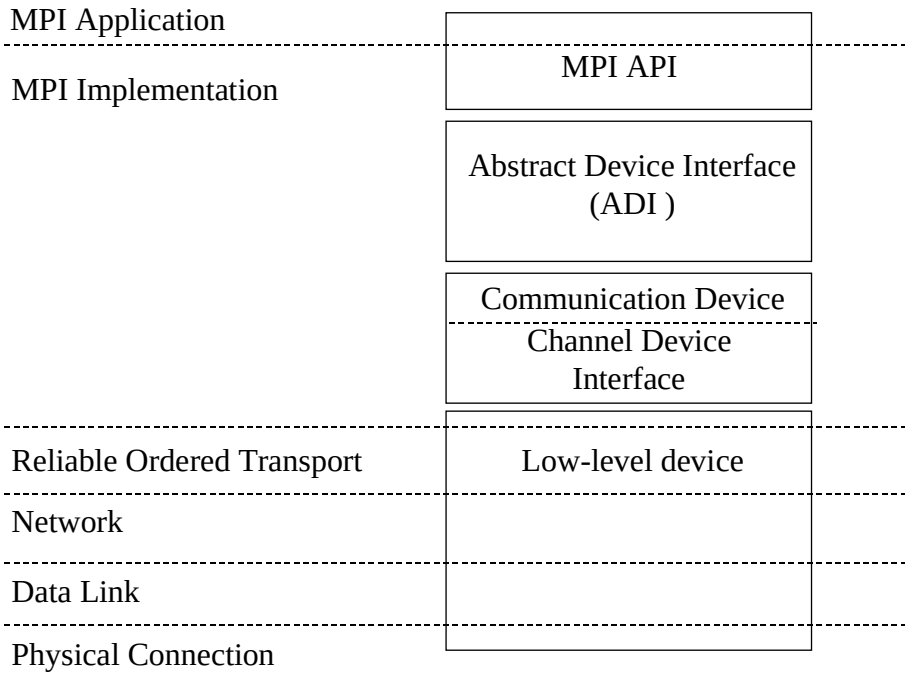

Figure 1. MPICH layered software architecture

The deficiencies of the MPICH architecture, such as inefficient multi-fabric communication and non-thread-safety are rooted in the ADI and Device layers. In order to make further discussion specific, we present the ADI and CDI in more detail below.

2.1 Abstract Device Interface (ADI)

Message passing in MPI is organized in the following way. When users call *MPI_Send* or *MPI_Recv*, a data structure that contains an instantaneous description of the data transfer operation is created (we refer to it as *transfer description* later in the paper). The data structure is reference by an opaque pointer called *handle*. The handles to transfers are placed in *send* and *receive queues* that are used in order to ensure the correct order of transfer completion. The

handles are removed from the queues and passed to the appropriate communication devices for further processing. Each communication device accepts a handle, performs the transfers described by the handle, and reports completion of the transfer by setting a special flag associated with the transfer.

This general idea is captured in the ADI design. Conceptually, ADI contains a queue for outgoing messages, and two queues for incoming messages – *posted-* and *unexpected- receive queues* (Figure 2). If a receive request is posted (the user receive buffer is made available by the *MPI_Recv* call) earlier than the corresponding message actually arrives, the transfer description is created and its handle is placed into the posted-receive queue. When the message is received by some communication device, it is matched with the description, and the data is copied into the user buffer. Accordingly, if a message has been received but has not been posted yet, a transfer

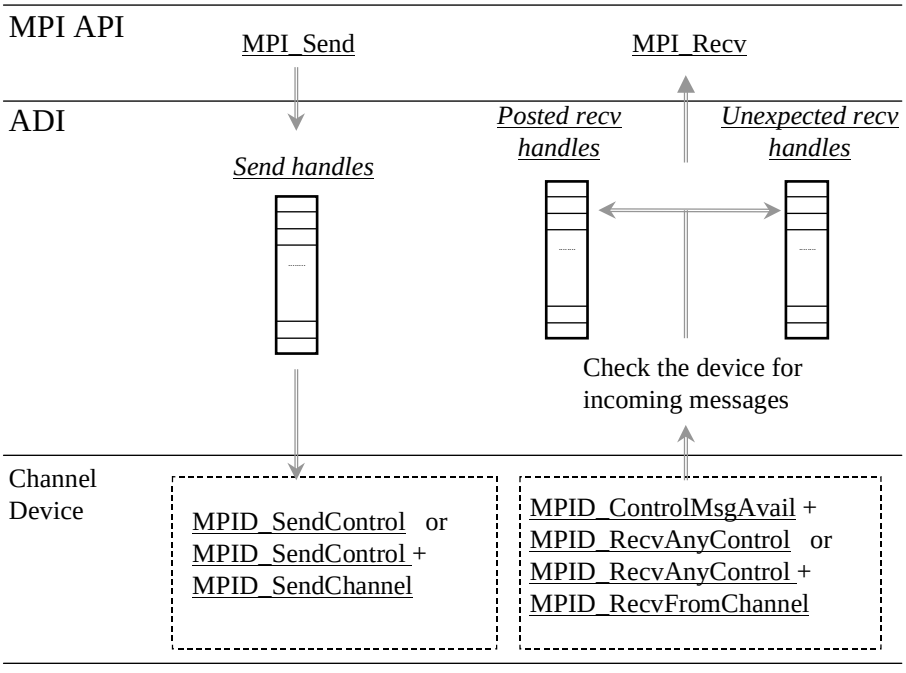

Figure 2. MPICH message-passing engine

description for this message is created, the message data is buffered, and the handle is placed in the unexpected-receive queue. (We have described the *standard* transfer mode; *ready* and *synchronous* modes allow one to avoid intermediate data buffering.) When the corresponding receive operation is posted, the MPICH runtime system matches it with the handle in the unexpected-receive queue, copies the data into the user buffer and completes the transfer.

In order to unify interface with communication devices, ADI maintains a list of available devices and sets of process ranks that are handled by each devices. After a handle for an outgoing transfer is created, ADI determines which device handles messages to the destination specified in the transfer, and passes the handle to this device. While receiving messages, ADI treats all available communication devices uniformly (fairly). ADI queries every device in a round-robin fashion and matches incoming messages with posted handles. If a match is found, incoming data is copied in the user buffer; otherwise, ADI handles the message as unexpected.

2.2 Device layer

The device layer contains a set of communication protocol modules (devices) that allow for communication over a specific fabric such as a network or shared memory. In MPICH [8, 9], several communication protocols may be encapsulated in a device object. In accordance with some device-specific criteria, a protocol is chosen every time a message needs to be transferred.

Different communication protocols are necessary in order to address various system limitations (communication media, buffering space) and for performance optimization. MPICH devices generally use three protocols depending on the message size: short, eager (long), and rendezvouz (very long) [3, 5].

2.3 Channel Device Interface

All communication modes in MPICH can be portably implemented atop of a small set of communication primitives that perform simple functions such as sending/receiving control information, sending/receiving user data, and querying communication media for incoming messages [6, 7]. The corresponding five function calls (Figure 3) constitute the minimal interface with the low-level system communication services in MPICH that is called Channel Device Interface (CDI) [6].

```
int    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* buf, int maxsize, int from)
void MPID_SendChannel (void* buf, int size, int dest)
```

Figure 3. The MPICH Channel Device Interface Calls.

There is also a set of auxiliary Channel Device Interface calls that can (and sometimes should) be implemented [6]. These calls are used with simple low-level devices to ensure progress of communications, as well as to extend the set of low-level device services accessible for the upper layers to allow for performance optimizations. A detailed discussion of these extensions is beyond the scope of this paper.

2.4 Deficiencies of the MPICH design

There are several deficiencies of the current MPICH design that deserve detailed consideration. These deficiencies have common reasons: non-thread-safety of the design and, as a consequence, running all communication activities in a single thread (that also performs computation in a given process). We will discuss them in the logical order.

2.4.1 Non-thread-safe features of the MPICH design

The MPICH design presented above is not thread-safe, and hence allows one neither to implement multi-threaded MPICH, nor to use MPICH in multi-threaded applications. In order to make MPICH thread-safe, it is certainly necessary to lock the send and receive queues to ensure atomic access by different threads. Note that both unexpected- and posted-receive queues should be locked simultaneously to ensure proper operation. The device code also should be locked because the results of executing CDI calls for the same device concurrently in several threads are ill-defined (two threads should not be able to receive the same message while executing CDI calls for the same device). However, this is not the end of all potential problems.

Consider what happens if an application thread is doing a blocking receive. The message is associated with the transfer handle created in ADI and posted in the appropriate queue (let us assume that the message has not arrived yet). The next thing for the thread to do is to execute a blocking receive in the device that is handling communication with the expected source of the message. However, it is entirely possible that another thread is already in the process of receiving the message under discussion. This second thread gets the data, finds posted transfer description, and completes the transfer. However, the first thread will never find out that the transfer is completed. This thread will be receiving incoming messages forever and will never return from the blocking receive call unless it is somehow notified that the transfer is already completed.

The above example shows that there is a race condition between two activities: extracting messages from the device and managing the receive queues. It is necessary to lock both the receive queues and the device code simultaneously in order to avoid this race condition. However, since ADI contains a single pair of the receive queues that are shared by all the

devices, locking the receiving code along with the queues practically means that only one user threads at a time can make progress in any of the devices while receiving data.

A possible solution to the problem is to maintain a pair of receive queues per device. However, this complicates processing of wildcard receives (receives from *any* source). Since the source of a wildcard receive is not known in advance, one cannot determine which receive queue pair should be used for the transfer. A possible approach is to enqueue handles to a wildcard transfer into all the devices' posted-receive queues. When one device receives the matching message, the transfer is marked completed, so that the rest of the devices can later discard the handles to this transfer. In this case, it is necessary to lock the transfer data structure when some device starts receiving a matching message until the transfer is marked completed. This prevents receiving (by different devices) more than one message that matches the wildcard transfer.

This approach requires extra functionality in the devices in order to retire handles pointing to a completed wildcard transfer. While retiring the handles, it is necessary to make sure that the transfer structure is deleted when the last handle is retired (otherwise, some devices would wind up with handles pointing to unavailable memory). Therefore, some sort of reference counting is required. The reference count updates need to be protected by a lock. Consequently, processing wildcard transfer with this approach introduces extra synchronization overhead proportional to the number of the devices, as well as introduces some dependence between the devices (they compete for the lock protecting a wildcard transfer).

Another possible approach is to allow the race condition to exist but to keep track of which messages are received in which threads or to arrange periodic checking whether a given message was completed from the thread that started executing the corresponding receive operation. This approach has an advantage of avoiding re-engineering of MPICH. However, our

experience of making single-threaded MPICH thread-safe without significant changes in the design showed notably lower message-passing performance of the thread-safe MPICH version as compared to the non-thread-safe one [14].

2.4.2 Serialization of communication and computation

The fact that all message-passing activities in MPICH occur in the same thread with user computation also leads to the following undesirable system characteristics. This prevents one from actually overlapping communication and computation unless asynchronous communication hardware is available. For portable implementations that do not rely on the availability of any special hardware, this is a significant drawback. One can argue that it might be possible to run MPI communication in one thread while performing computation in the other(s). However, since the MPI-1 standard [17] does not address this scenario, the current MPICH implementation does not guarantee that this would work. Moreover, this imposes restrictions on the application design, which is not desirable.

Also, if all communication takes place in one execution context, it is not possible to take advantage of multiple available processors for carrying out independent communicational and computational operations concurrently within the same process, thereby increasing performance. This is a significant disadvantage because computer architecture is developing towards multiprocessor platforms and some of the emerging architectures have hardware thread support [16]. With decreasing per- processor prices and increasing number of processors per computer system, the ability of software to take advantage of multiple processors while avoiding costly inter-process context switch overhead becomes increasingly important.

2.4.3 Averaging of communication latency

Round-robin querying of the devices for incoming messages designed to service all communication requests in a fair fashion has affect of averaging communication latency in the system. This makes system communication latency of the same order as the latency of the slowest device. Indeed, since the receiver side has no knowledge of which communication fabric has a pending message, it has to query all communication devices in accordance with some strategy that ensures that messages from all devices will be eventually received. A desired property of a communication system is fairness of service meaning that the system pays equal attention to all communication devices and ensures that incoming messages are processed "as soon as they are available". Otherwise, the communication progress in the system is ill-defined[2].

If one considers usual round-robin technique for ensuring fairness of service, which is used in the current MPICH design, the average communication latency will be equal to the half the sum of the device latencies, which is of the order of the largest value. For a system with a shared-memory device (characteristic latency of the order of 10^{-5} seconds) and a TCP network device (usual latency of the order of 10^{-3} seconds), a common choice for clusters of workstations, one is bound to have a millisecond-range latency even though the shared-memory device can do much better. On the other hand, if communication in each device is allowed to make progress in separate threads, the need in round-robin querying the devices disappears, and the above mentioned dependence between receive operations in the devices is broken.

Other single-threaded design options that alleviate this problem are definitely possible. For instance, one can poll faster devices more. However, this approach requires special tuning for a specific combination of the devices; this diminishes the design portability. The approach also

increases overall querying overhead (fraction of application execution time spent while querying the devices), which is not desirable.

The above mentioned deficiencies can be avoided by introducing multiple threads in the MPICH design. As a matter of fact, communication software naturally lends itself to a multi-threaded design, since it incorporates two asynchronous parallel activities, sending and receiving data, that are independent in the majority of cases (unless chosen communication protocols require certain feedback between data receiving and sending, such an acknowledgments). That is why we reconsider the design of the ADI and Device layers, and propose a refined multi-threaded design. The CDI is also reconsidered and reduced to two simple thread-safe calls.

3. Multi-fabric thread-safe MPI design

Our general goal is to design a portable and reasonably efficient thread-safe MPI implementation for general-purpose computer systems with two devices: a shared-memory device and a generic "network" device. Nothing in the design limits the number of the devices used, and two devices are considered only in order to make the discussion specific. The design ensures fair communications progress for all devices.

We assume that every ordered pair of processes uses one device for all data transfers between them. This allows us to determine which device should be used for a given data transfer using the IDs of the processes. We use the framework of the MPICH MPI implementation and ideas concerning the multi-protocol device design to take advantage of the overall design portability. Our contributions are addressing thread-safety and efficiency of multi-fabric communication in MPI.

We are using preemptive threads in our design and rely on the fair threads scheduling provided by the operating system through the thread package. The reason for this choice is simple design strategy that allows us to manage threads easily and to ensure communications progress. Every implementation thread does its work until it is done with all available service requests, then it goes to sleep and gets awaken if there are new requests pending. We consider other options in the last section of the paper.

3.1 ADI

The message-passing engine included in ADI is simple and generic. It multiplexes outgoing messages between the devices and de-multiplexes incoming messages received by the communication devices. The send queues are moved from ADI down to the devices, effectively parallelizing outgoing data flows trough different devices. There is a single pair of receive queues maintained in the ADI and used by all the devices. Using a single pair of queues allows us to treat wildcard receives in the same way as ordinary receives and avoid complicating the design. However, in our multi-threaded design, the receive queues need to be locked only for the duration of queue management operations but not for the duration of the receive in the device, as in case of single-threaded design (see Section 2.4.1). The later statement can be verified by inspection of the multi-threaded design presented below. It is also possible to use a pair of receive queues per device, and the choice between these two options can be determined by experimentation, as well as by amount of time available for implementing and debugging the required extra functionality discussed above.

The ADI accepts requests for data transfer, creates transfer descriptions, and puts user threads to sleep while waiting for completion of blocking operations. User threads are awaken by

the appropriate device threads upon completion of communication operations. The interface between the MPI API layer and ADI consists of routines requesting send or receive operation of a given type. The interface between ADI and Device consists entirely of thread-safe routines for managing the queues of handles for send and receive operations that are maintained in every device object (see discussion below).

3.2 Device

The Device layer contains several device objects. A device object is a generic abstraction of a communicating entity that is instantiated for a target platform either directly, or through CDI. The design introduced here emphasizes that a device is an asynchronous multi-threaded object (collection of routines and data structures). Every device has two special threads: a Sender thread and a Receiver thread. This allows the device to process unrelated sends and receives in the same device concurrently, as well as to perform all communications in different devices in parallel.

The Device layer has two well-defined interfaces: one with the ADI code and the other with the low-level device code. The former interface consists entirely of the thread-safe routines for handling queues of handles. The latter one is either a low-level device API (such as a network driver's API, for instance), or the Channel Device Interface.

A device object contains a queue of handles to send transfers that are carried out by the device. Handles to all outgoing transfers are enqueued into the send queue by the ADI code, which is executed in user threads. These handles are dequeued by the device code while performing sends.

All devices share two queues of handles for posted and unexpected incoming transfers. Handles to all posted transfers are enqueued into the posted-receive queue by the ADI code.

These handles are dequeued by the Device code. Analogously, all handles to unexpected transfers are enqueued into the unexpected-receive queue by the Device code and are dequeued by the ADI code executed in the user threads.

In addition, every device contains a *control-packets queue* for handling control packets that are placed there by device's Receiver threads. The packets contain control information necessary to carry out some communication protocols that require exchange of requests and acknowledgments prior to or after the actual data transfer.

Every device object incorporates (possibly multiple) communication protocols specific to the device and a decision-making mechanism to choose between these protocols while transferring the data. An example of such communication protocols is a set of protocols used in the MPICH ADI: short, eager, and rendezvouz [5].

3.3 The lower-level communication service

Low-level device code provides device objects of the Device layer (perhaps through CDI) with basic message-passing capabilities, such as ordered reliable transfer of messages up to a certain size. While considering this part of the MPICH design, a generic low-level device that provides ordered reliable delivery of packets up to the fixed length across some sort of communication fabric was targeted. This type of service has less software overhead than the stream-oriented services that incorporate intermediate data buffering and other overhead in order to provide a data stream abstraction. So, the "network" device mentioned above can be based on some packet-oriented communication service provided by the operating system on target platforms, such as RDP protocol stack [18]. This type of service is also identical to the services provided by the shared memory low-level device with static memory management strategy [14].

3.3.1 The Channel Device Interface

The following two functions constitute the interface between low-level device layer and the Device layer (Figure 4). These functions send and receive packets up to a maximal size. The t_packet is a data type that describes a packet used by the Device layer. The *SendPacket* function is non-blocking, and *RecvPacket* blocks for incoming packets. The functions return one if data transfer is successful, zero if the call would block (*SendPacket* only), and an error code otherwise. The *SendPacket* function sends a specified packet *pkt* of a given *size* to a destination

```
int SendPacket(t_packet *pkt, int size, int dest)
int RecvPacket(t_packet *pkt, int *size, int dest)
```

Figure 4. Revised Channel Device Interface.

process with the rank *dest*. The *RecvPacket* call receives any packet of maximal size **size* into a buffer pointed by *pkt*; the actual size of the packet is returned in the **size* in-out parameter, and the source rank is returned in the **source* in-out parameter. As shown below, MPI extended communication services can be built atop of the functionality provided by these two calls.

3.3.2 The design of channel devices

The design of devices that use CDI (channel devices) is based on the above mentioned CDI functions. The following design ensures fairness of services, as well as concurrent progress of independent data transfer (Figure 5).

Each of the Sender and Receiver threads maintains an array of handles (pointers to transfer description structures). The size of every array is equal to the number of processes serviced by this device and each handle points to the description of a transfer that is currently in

progress between the local process and the process whose rank equals to the array index of this handle. The Sender thread steps through its array of transfers in a round-robin fashion and performs non-blocking *SendPacket* calls for every operation in progress. If an operation is completed, the corresponding handle array entry is reset, and a new handle from the *Send Queue* is processed. This allows one to parallelize sending messages to different destination without violating MPI ordering rule. If a send to some destination is blocked, for instance, because of lack of buffering space on that destination, data transfer to other destinations can still make progress in the device.

Analogously, the Receiver thread extracts incoming packets from the network by calling the *RecvPacket*, checks the destination, and appends the data (if any) from the packet to the

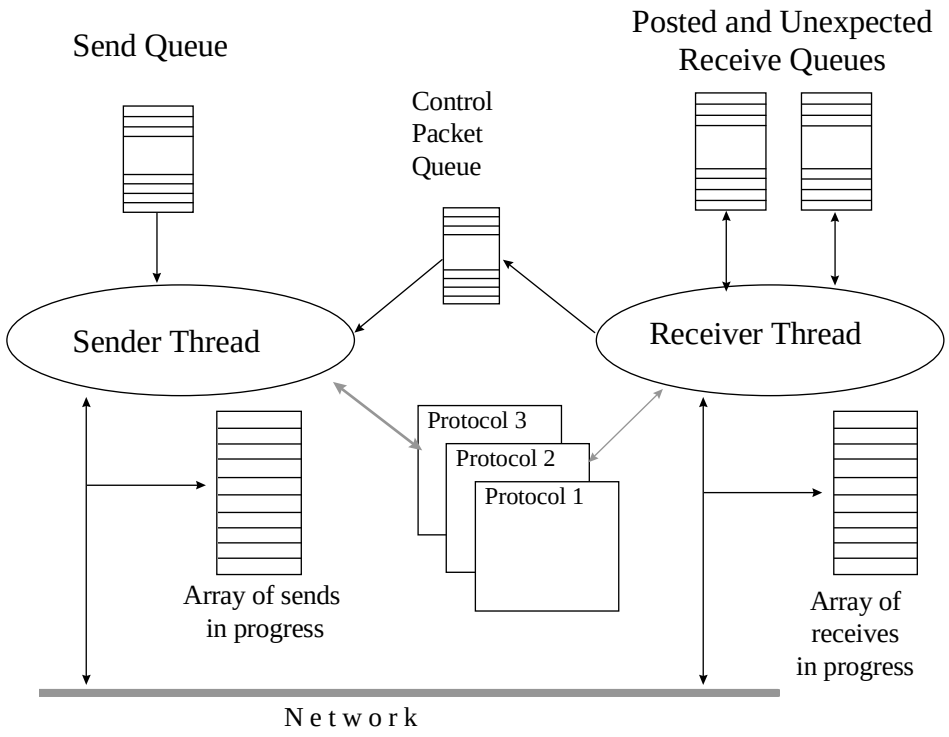

Figure 5. The proposed channel device design.

appropriate data buffer. If a receive operation is completed, the corresponding entry in the receiver array of handles is reset, otherwise, it is properly updated. In the former case, the transfer description for the message is matched with posted transfers in the posted-receive queue and is placed in the unexpected-receive queue if the match is not found.

One can see that the only functions that need to be implemented for a target platform are *SendPacket()* and *RecvPacket().* This brings the amount of porting efforts to a minimum, which was one of the design goals. However, it might be more efficient to implement communication protocols of some devices directly in terms of the low-level device API calls.

It should be mentioned that the Sender's and Receiver's arrays of transfer structures, along with the send and receive queues, are considered only as an example. In practice, a variety of approaches to monitoring progress of service requests can be used. The described design emphasizes that the message transfer between any pair of processes should be carried out in the temporal order of the communication calls made in the source process, so that the messages are received in the same order they are sent.

3.4 The design summary

Let us summarize the MPICH design presented (Figure 6). As can be seen, user communication calls (sends and receives) are initiated on the MPI API level. Then send requests are multiplexed between the devices described above and handles for these operations are enqueued into the send queue of the appropriate devices. The Sender threads dequeue the handles and carry out corresponding send operations in accordance with some communication protocol. Accordingly, the Receiver threads in all devices check if there are some pending messages in

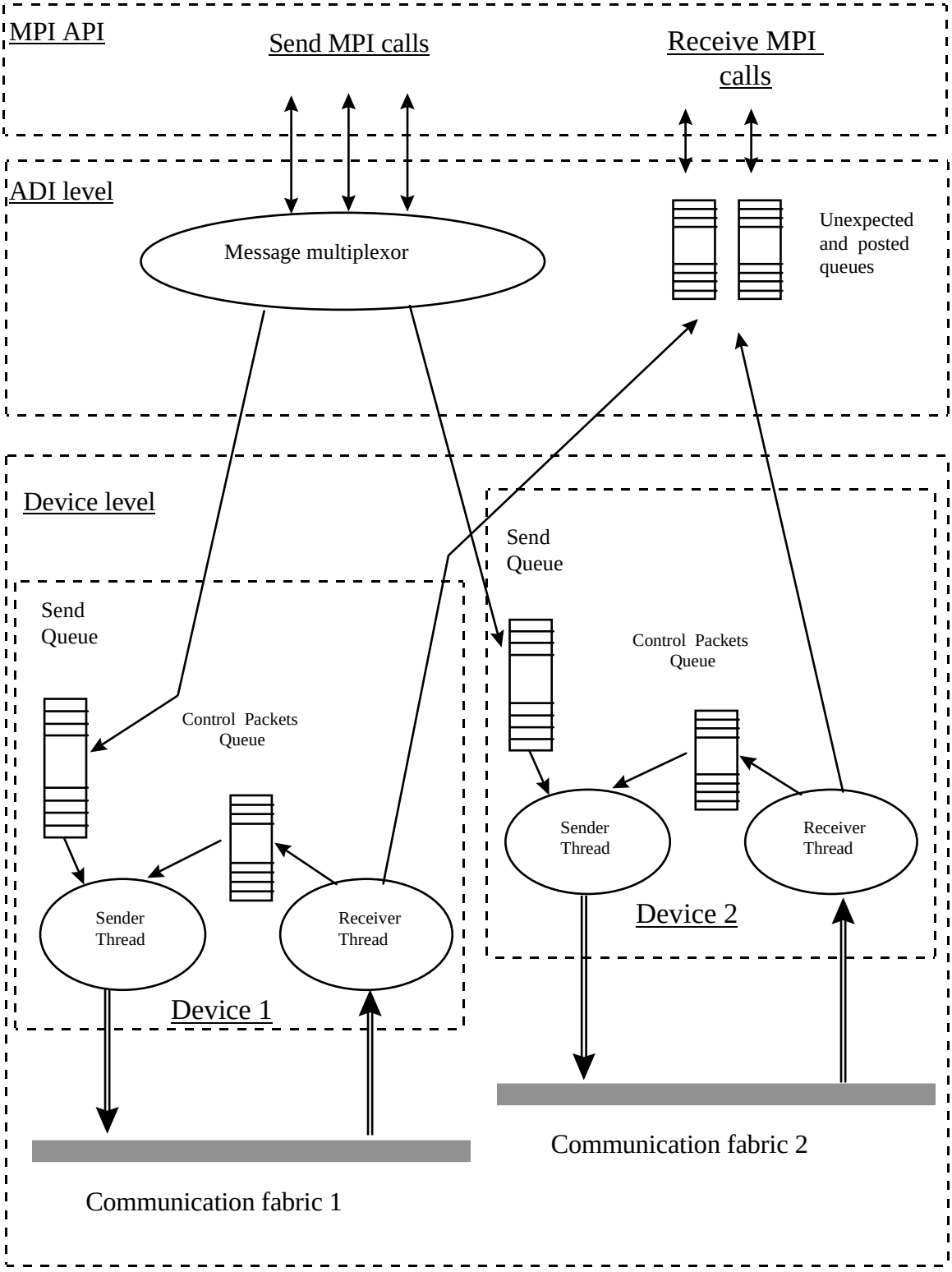

Figure 6. The proposed multi-threaded design

their communication media and extract these messages. If these are control packets that require acknowledgment, the Receiver threads place them into the control packet queue, and the Sender threads will process them later. Otherwise, the Receiver threads place the message data in the buffers associated with transfer structured in the posted- or unexpected-receive queues, depending on whether they have been posted by the user threads.

3.5 General design comments

While implementing the design discussed above, one should use synchronization and scheduling mechanisms provided by a POSIX-compliant thread package in order to ensure portability. It is also important to make sure that neither user, nor device threads spend an unnecessary amount of CPU time busy-waiting for any events associated with communications. For instance, device receiver threads should not poll for incoming messages. If they do, one has to balance their CPU use by assigning appropriate priority to the threads and/or by using some platform-dependent polling strategy. In the design presented above, it is assumed that the "network" device is based on some communication service provided by the target platform operating system that provides asynchronous network event notifications. In case of the shared memory device, one can use POSIX condition variables to notify a target process of an incoming message.

As mentioned, the locking and scheduling mechanisms used in the design are provided by a POSIX-compliant threads library available on a target platform. In this respect, the efficiency of an MPI implementation, which follows the design being described, is affected by the quality of a POSIX-compliant threads library for a target platform. POSIX mutexes, semaphores, and condition variables provide enough functionality for the design to be successfully implemented.

It is important to note that the transfer description structures, which is designed to keep track of communication operations in progress, should be amended with a condition variable object. User threads can enter an efficient waiting state on this object while waiting for completion of an operation associated with the handle. In order to wake such a user thread up, the device code can signal the condition. Since handles are created dynamically, one has to dynamically create condition variables for them. In order to make this process less time consuming, one can cache and reuse handles that are freed by user threads.

4. Implementing high-performance message-passing systems for general-purpose architectures

Achieving high message-passing performance on general-purpose platforms is complicated by operating-system-relates overheads embedded in traditional communication protocol stacks. Sub-optimal communication latency and bandwidth are mostly due to the swapping, interrupt handling, and system call overhead, as well as excessive intermediate data buffering. These problems are widely recognized and are being addressed in [1, 2]. However, some of these problems can be alleviated with the use of latency hiding in the application domain.

Latency hiding is present on all levels of computer system architecture, from hardware processor design with several hardware contexts and memory hierarchies to CPU and I/O scheduling by operating systems. With wide acceptance of multi-threaded programming model, the technique becomes available for application programmers.

POSIX threads standard [10, 12] is an official standard that establishes a foundation for portable multi-threaded programming. Although POSIX weak requirements for support of thread

scheduling policies restrict POSIX threads portability, POSIX thread packages for widely-used general-purpose operating systems (Solaris, Windows NT) provide consistent support for preemptive thread scheduling that can be used to resolve various design and performance issues.

In addition to latency hiding, threads also allow for taking advantage of multiple CPUs and utilizing fine-grain parallelism in applications (the inter-thread communication and synchronization overhead is low in comparison with the inter-process overhead). The detailed discussion of multi-threaded programming models is beyond the scope of this paper and can be found elsewhere [11, 12, 13]. The point is that threads are a generic means of communication latency hiding, achieving fine-grain parallelism, and better utilization of available multi-processor hardware. Since multiprocessor platforms will most probably dominate on the market in the near future, this issue is to be addresses in the design of parallel software (message-passing libraries are parallel software too!).

All these factors make multi-threaded programming models a friendly environment for implementation of high-performance communication software. The fact that communication software itself reveals certain parallelism that should be utilized, makes usage of multiple threads for communication software implementation even more attractive.

It should be mentioned that switching between threads and using synchronization primitives also incurs a finite overhead. Hence, communication benchmark results obtained with a multi-threaded communication software will probably not be as good as the results of a single-threaded implementation that burns CPU cycles in busy waiting for incoming data and delivers low communication latency. However, this latency can be hidden with the overlap of communication and computation, so in a real-life situation, well-designed applications that use

multi-threaded communication software will reveal better overall performance that their single-threaded counterparts even though the communication benchmarks show the opposite.

5. Conclusions

In this paper, we have discussed the design of a high-performance thread-safe multi-fabric implementation of the MPI standard. We showed that using multiple threads inside the implementation supplies the system with a set of important properties, such as utilizing available parallelism in computation and communication, simple and effective multi-fabric thread-safe design, and easy use of multi-processor hardware. Finally, we briefly reviewed the obstacles to implementation of high-performance message-passing software for general-purpose platforms and showed that threads can be portably used to deal with these obstacles. Consequently, using threads is an efficient and portable way to solve many problems encountered while implementing message-passing software that are not easy to address with single-threaded design options.

6. Future work

We envision other design opportunities for increasing message-passing performance of MPI implementations. According to the semantics of communications in MPI, ordered message delivery should be enforced only within a communicator. Hence, total message ordering can be relaxed, such that ongoing communication is processed on per-communicator basis (a set of the send/receive queues is maintained per communicator). Another advantage of this option is decreased locking granularity inside the implementation. Indeed, queue management overhead in one communicator can be hidden by communication in others. This design approach requires reconsidering thread scheduling discipline because of finer-grain parallelism that is targeted. We

are considering using cooperative scheduling techniques and light-weight thread packages in order to take advantage of the above mentioned design option.

7. References

1. Compaq Computer Corp., Intel Corporation and Microsoft Corp. VI Architecture. http://www.viarch.org, May 1998.
2. Basu, A., Welsh, M., and Thorsten von Eicken. Incorporating Memory Management into User-Level Network Interfaces. Department of Computer Science, Cornell University, Technical Report TR97-1620.
3. Gropp, W., Lusk, E., Skjellum, A., and Doss, N. MPICH: A High-Performance, Portable Implementation for the MPI Message-Passing Interface. Parallel Computing, 22, 1996, pp. 789-828.
4. Gropp, W., Lusk, E., and Skjellum, A. Using MPI: portable parallel programming with the Message-Passing Interface. MIT Press, Cambridge, MA, 1994.
5. Gropp, W. and Lusk, E. A high-performance MPI implementation on a shared-memory vector supercomputer. Parallel Computing, 22, 1997, pp. 1513-1526.
6. Gropp, W. and Lusk, E. MPICH ADI Implementation Reference Manual. ftp://info.mcs.anl.gov/ pub/mpi/adiman.ps, Dec. 1995.
7. Gropp, W. and Lusk, E. MPICH working note: creating a new MPICH device using the Channel Interface. ttp://www.mcs.anl.gov/home/lusk/mpich/workingnote/newadi/ note.html, Dec. 1995.
8. Gropp, W. and Lusk, E. MPICH working note: the implementation of the second-generation MPICH ADI. ftp://info.mcs.anl.gov/pub/mpi/ workingnote/adi2imp.ps, April, 1996.
9. Gropp, W. and Lusk, E. MPICH working note: the second-generation ADI for MPICH Implementation of MPI. ftp://info.mcs.anl.gov/pub/mpi/workingnote/nextgen.ps, April, 1996.

10. Institute of Electrical and Electronic Engineers. Information Technology - Portable Operating Systems Interface - Part 1: System Application Program Interface (API) - Amendment 2: Threads Extensions [C Language]. IEEE, New York, NY.
11. Lewis, B. and Berg, D. J. Threads Primer: a guide to multithreaded programming. SunSoft Press, Mountain View, CA, 1996
12. Nichols, B., Buttlar, D., and Farrell, P. Pthreads programming. O'Reilly & Associates, Sebastopol, CA, 1996.
13. Powell, M.L., Kleiman, S.R., Barton, S., Shah, D., Stein, D., and Weeks, M. SunOS multi-thread architecture. http://www.sun.com/sunsoft/Developer-products/sig/threads/papers/ sunos_mt_arch.ps, Oct. 1996.
14. Protopopov, B.V. Concurrency, multi-threading, and message passing. M.S. thesis, Department of Computer Science, Mississippi State University, 1996.
15. Skjellum, A., Protopopov, B., and Hebert, S. A thread taxonomy for MPI. Proceedings of the Second MPI Developer's Conference in University ofNotre Dame, Indiana, 1-2 July 1996, Lumsdaine, A. and Skjellum, A. (Eds.), IEEE Press, Los Alamos, CA, 1996, pp. 50-57.
16. Tera Computer Company. The Tera multi-threaded architecture (MTA). http://www.tera.com/web/mta.html, May, 1998.
17. The MPI Forum. The MPI Message-Passing Interface Standard. http://www.mcs.anl.gov/mpi /mpi-report/mpi-report.html, Dec. 1995.
18. Velten, D., Hinden, R., and Sax, J. Reliable Data Protocol. http://www.fagg.uni-lj.si/MIRROR/ rfc/rfc/rfc908.html, Sept. 1996

Footnotes

1. The MPICH design and implementation is being improved and corrected. In this paper, we refer to the MPICH v.1.0.13 that implements the revised ADI design (ADI-2) [8, 9]. This version of MPICH has refined code structure and addresses use of multiple communication devices and different communication protocols on the ADI level. According to our experiments, there is no substantial performance difference between the new MPICH version and the earlier MPICH 1.0.12 with ADI-1.
2. We consider a general-purpose computer system with time-sharing controlled by the operating system's kernel. In such systems, application designers do not have direct control on the process's scheduling. The only way to ensure deterministic progress of a message-passing application is to require uniform communications progress. Real-time systems, on the contrary, schedule all process's activities directly and do not depend on uniformity of the communications progress.

Biographies

Boris V. Protopopov

Boris V. Protopopov holds MS in Information and Systems Science from Moscow Institute of Physics and Technology [1993] and MS in Computer Science from Mississippi State University [1996]. He is a Ph.D. candidate at the Department of Computer Science, Mississippi State University and currently holds a position of Senior Software Engineer, Operating Systems Division, Mercury Computer Systems, Inc. His research interests include operating systems design, high-performance computing, message passing, scheduling, and optimal control.

Anthony Skjellum

Anthony Skjellum holds the BS (Physics) [1984], MS (Chemical Engineering) [1985], and PhD (Chemical Engineering plus Computer Science Minor) [1990] from the California Institute of Technology. From 1990 to 1993, he undertook research in computer science at the Lawrence Livermore National Laboratory in message-passing systems and scalable libraries. Since 1993, Skjellum has undertaken research in message passing systems, has contributed heavily to the MPI standards, and is a co-developer of the widely used MPICH library. He has co-authored a book, and has several publications relating to message passing and parallel libraries.