PORTING THE CRI/EPCC T3D/E MPI LIBRARY
TO HPCx UNDER LAPI

Christos Kartsaklis

MSc in High Performance Computing
The University of Edinburgh
Year of Presentation: 2003
Authorship Declaration

I, Christos Kartsaklis, confirm that this dissertation and the work presented in it are my own achievement.

1. Where I have consulted the published work of others this is always clearly attributed;
2. Where I have quoted from the work of others the source is always given. With the exception of such quotations this dissertation is entirely my own work;
3. I have acknowledged all main sources of help;
4. If my research follows on from previous work or is part of a larger collaborative research project I have made clear exactly what was done by others and what I have contributed myself;
5. I have read and understand the penalties associated with plagiarism.

Signed:

Date:

Matriculation no:
Abstract

MPI is a standard for message passing, which since its publication has been implemented for a wide range of HPC hardware and under a number of communication protocols. The CRI/EPCC T3D/E MPI library is one such implementation. It was designed for the Cray T3D and T3E services, it is single-threaded, and used the SHMEM library as the foundation of its inter-process communication. SHMEM has been available for a number of single-spaced and distributed shared-memory systems, and comprises a set of RMC and remote atomic operations.

The HPCx platform, a supercomputer consisting of 40 IBM p690 Regatta nodes (1280 processors), equips the HPC programmer with LAPI (low-level API) – the lowest-level set of user-accessible primitives for inter-processor communications. Our interest is aimed towards the active-messaging facilities provided by LAPI. The concept of active messages has been existent for a while. It suggests intelligent messages, capable of specifying and executing upon their arrival, a piece of computation at the receiver node.

In this document we describe how we ported the T3D/E MPI library to the HPCx supercomputing facility using LAPI. The new library is called HPCxMPI. We focused on re-designing its communications layer entirely in terms of LAPI, in particular its active messages. We present technical difficulties we were faced with while we were moving to this different model of communications.

The latency of 16-byte MPI_Send payloads is 40 µsec and 32 µsec for the interrupts-enabled and polling modes of LAPI respectively, while it is 23 µsec for IBM MPI.
Acknowledgements

Thanks must go to my supervisor, Dr. Stephen Booth, for his invaluable guidance, which allowed me to develop individual thought and helped me to remain within scope. I would also like to thank him for those excellent technical discussions we had. I really do feel that I learned a lot out of working with him.

Finally, it was that discussion I had with Dr. David Henty, which sparked off the idea of porting the library; therefore thanks go to him as well.
# Contents

1. **INTRODUCTION**................................................................. 1  
   1.1. MPI and the Send-Receive Model................................. 1  
   1.2. Active Messages.......................................................... 3  
   1.3. Motivation................................................................. 4  
   1.4. Scope........................................................................ 4  
   1.5. Preliminary Notes...................................................... 4  
   1.6. Outline of the Dissertation......................................... 5  

2. **LAPI, AM INTERFACES AND AM-BASED MPI LIBRARIES**..... 6  
   2.1. The LAPI Library.......................................................... 6  
      2.1.1. What is LAPI and What it Serves.......................... 6  
      2.1.2. LAPI Synchronisation Primitives........................... 7  
      2.1.3. The Principal Mechanism for Active Messages........ 9  
      2.1.4. How LAPI Reacts to Incoming Active Messages....... 10  
      2.1.5. LAPI Handlers, Counters and Threads (Examples).... 11  
      2.1.6. Contrast LAPI to Other AM and AM-Like Libraries... 13  
         2.1.6.1. Controlling the Progress of Active Messages.... 13  
         2.1.6.2. Tracking the Local Completion of AMs............ 14  
         2.1.6.3. Explicitness of the Request-Reply............... 14  
         2.1.6.4. Parameters Passing to Handler Routines........ 15  
   2.2. Existing AM-based MPI Libraries............................... 15  
      2.2.1. MPI-AM............................................................. 15  
      2.2.2. MPI-LAPI.......................................................... 16  
      2.2.3. MPI on the Fujitsu AP1000+................................. 16  

3. **INTRODUCTION TO THE CRI/EPCC T3D/E MPI LIBRARY**...... 18  
   3.1. Facilities Offered by the BASIC COMMS Layer............... 18  
   3.2. Inside the BASIC COMMS Layer.................................. 19  
      3.2.1. The P/Message..................................................... 20  
      3.2.2. The P/Messages Queue......................................... 20  
      3.2.3. The BC_PUTP routine......................................... 22  
      3.2.4. BC_PUTD and BD_GETD....................................... 23  
   3.3. Inside the Protocols.................................................. 23  
      3.3.1. Protocols T(1) and T(N)..................................... 24  
      3.3.2. The RTA Protocol.............................................. 24  
      3.3.3. The RAT Protocol.............................................. 26  

4. **THE DESIGN OF HPCXMPI**............................................ 27  
   4.1. Why not Use TurboSHMEM?........................................ 27  
   4.2. Localisation of Locking............................................ 27  
   4.3. Porting the BC_PUTP Routine.................................... 27  
   4.4. The New Protocol P+D.............................................. 30  
      4.4.1. Why a New Protocol?......................................... 30  
      4.4.2. Why Not Using T(N) Instead?............................. 30
List of Figures

FIGURE 1: THE REQUEST-REPLY ACTIVE MESSAGING MODEL .......................... 3
FIGURE 2: LAPI COUNTERS EXAMPLE A - THE USE OF A LOCAL AND A
    COMPLETION LAPI COUNTER .................................................................. 8
FIGURE 3: LAPI COUNTERS EXAMPLE B - PING-PONG ............................. 9
FIGURE 4: LAPI THREADS MODEL [IBM98] .................................................. 10
FIGURE 5: LAPI HANDLERS, COUNTERS AND THREADS - I ..................... 12
FIGURE 6: LAPI HANDLERS, COUNTERS AND THREADS - II .................... 13
FIGURE 7: THE LAYERS OF CRI/EPCC T3D/E MPI .................................. 18
FIGURE 8: THE STRUCTURE OF A P/MESSAGE ......................................... 20
FIGURE 9: THE BC_PUTP ROUTINE ......................................................... 22
FIGURE 10: THE RTA PROTOCOL .............................................................. 25
FIGURE 11: THE RAT PROTOCOL ............................................................... 26
FIGURE 12: TWO CONSECUTIVE T(1) MPI_SEND .................................. 28
FIGURE 13: HANDLING AND PROCESSING A P/MESSAGE ...................... 29
FIGURE 14: HHBC_BUPDP WITH COUNTERS .......................................... 32
FIGURE 15: P+D PENDING TRANSFERS HANDLED BY BC_ABSORB ........ 33
FIGURE 16: SEQUENCE NUMBERS POOL SYNCHRONISATION ................. 35
FIGURE 17: CLUSTERING CAPABILITIES OF THE OO P/MESSAGES
    ACCUMULATOR .................................................................................. 37
FIGURE 18: THE HHBC_PUTPD WITH OO P/MESSAGES HANDLING .......... 38
FIGURE 19: THE ALGORITHM FOR CLUSTERING OO P/MESSAGES ............ 39
FIGURE 20: LATENCY OF THE BASIC VERSIONS ...................................... 42
FIGURE 21: LATENCIES FOR HPCXMPI (INT)/PROBES (INT) AND IBM MPI ... 44
FIGURE 22: HPCXMPI IN VARIOUS INT MODES ...................................... 44
FIGURE 23: THE PPD – T(1) EMULATOR ............................................... 46
FIGURE 24: BANDWIDTH DELIVERED BY HPCXMPI ............................... 47
List of Tables

TABLE 1: THE BASIC COMMS API.......................................................................................... 19
TABLE 2: AMOUNT OF CONTEXT SWITCHES (FIGURE 22)................................................. 45
1. Introduction

Since the publication of the Message Passing Interface (MPI\textsuperscript{1}), members of the academic community and the HPC hardware vendors have embraced it. MPI has been implemented for a vast range of HPC hardware and for a number of communication protocols. Among those is the CRI/EPCC MPI library \cite{EPCC95} for the Cray T3D/E platforms. The point-to-point part of the MPI specification \cite{MPI95} follows the send-receive model. In this model both the sender and receiver processes actively participate in a data transfer. In fact, reception of data requires the matching of the send operation with a receive. The MPI specification permits sends to progress asynchronously with their matching receives. Hence, developers have to address the issue of offloading a send buffer to the receiver space, especially for the case where the receiver has not already acknowledged the reception.

On the other hand, a concept for achieving communications performance close to the peak of the hardware has been existent for a while – active messages \cite{ECGS92,Eick87}. An active message can specify how the receiver should react prior to the delivery of the message.

1.1. MPI and the Send-Receive Model

Under message-passing send-receive models, communicating parties publish to the communications library their desire to perform a send or receive operation. In addition to this, each party informs its operation about which remote operation it wishes to match and under which context of communications. For instance, to achieve matching, the MPI and FMPL libraries require the user to tag its operations with the same integer value and reference the same communicator. Then, it is the responsibility of the library to achieve offloading the send buffer into a matching receive buffer. In no way does the name of an operation specify the direction of data transfer – pushed to or pulled in, the data eventually reaches the receiver.

Message-passing libraries become really flexible by complementing their send-receive operations with completion semantics. This results in (non)blocking and synchronous/asynchronous operations. The non-blocking ones aim to overlap data transfer with computation, whereas the asynchronous ones aim to initiate the transfer prior to matching. Synchronous communications are dramatically delayed by the time it takes for the receiver to introduce the matching receive. Implementing such flexible communication semantics poses two major problems to the developer of a message-passing library:

- Matching send and receive operations, and
- Dealing with send operations that do not synchronise with the receiver.

Implementations of message-passing send-receive models suffer from the cost of locating the receive buffer, and maintaining the ordering and synchronisation semantics of the library. For this reason, small structures known as control messages or envelopes have been devised. Typically, the sender process populates such a message with a description of the operation (e.g., synchronous send, sender id, tag, etc.) and dispatches it.

\textsuperscript{1} The MPI standard is defined and maintained by the MPI Forum - http://www.mpi-forum.org
to the receiver. At the other end, the receiver maintains a queue where such messages are stored, and uses some way for matching (alien) sends with (local) receive operations.

A number of protocols have been devised to achieve the most efficient transfer of control information and buffers for a particular communications event. The most common protocols are the **eager** and **rendezvous**.

**Eager Protocol:** Under the *eager protocol*, the send buffer is buffered at the remote end (receiver) and at some point the receiver will copy the buffered data in the user-space. This buffering requires the existence of some pre-allocated space, and some logic shared between the two processes to avoid corrupting the space due to multiple sends. Depending on the size of the send buffer, the network characteristics and the availability of remote memory access, a number of variations have emerged: T(1)/short protocol (T3D/E and MPICH), T(N) (T3D/E), persistent buffer association (MPI on IBA\(^1\), [LWKW’03]), postbox queues (Sun MPI for the Sun Fire Link Interconnect, [SiJa02]).

Eager protocols are generally based on remotely manipulating data structures that belong to the receiver process. They are bound to the performance of the library that facilitates remote shared memory access. Typically, each process at bootstrap time allocates a memory space dedicated to eager transfers. When a process wants to eagerly send data, it has to register a partition with the receiver. There are many ways to do this. In SCI-MPICH, each process maintains a ring of pointers to pre-allocated buffers. The state of the ring is placed in a shared SCI memory segment, so that remote processes can read and modify. The sender updates the remote ring, pushes the data into it and issues a 16-byte packet to notify the receiver about changes to the ring. CRI/EPCC T3D follows a similar scheme and uses SHMEM operations exclusively to modify the remote space [EPCCα]. In Sun MPI, instead of having per-process shared memory segments of space dedicated to eager transfers, there is a single large pool of shared memory accessible by all processes within a box [SiJa02].

The responsibility of eager placements is shared between the communicating processes. The sender manipulates remote structures to push its send buffer, while the receiver must identify the placement and shift the send buffer from the eager space to the user-space. Historically, the computational weight of eagerly transferring data is assigned to the sender.

**Rendezvous Protocol:** Under the *rendezvous protocol*, the sender and receiver undergo a handshake session, so that the sender knows where to push the send buffer. The rendezvous protocol has two key features: (i) it maps nicely on synchronous sends, and (ii) it is the choice when either the buffering imposed by the eager protocol becomes an overhead\(^2\), or the eager space is exhausted.

The MPI specification dictates that synchronously sending a buffer will not complete unless the receiver introduces the matching receive operation.

\(^1\) Intel InfiniBand
\(^2\) The overhead is due to shifting buffers from the eager space to the user-space – an extra memory copy.
Therefore, a form of handshake is always required. There are two ways developers usually tackle synchronous transfers:

- The sender handshakes with receiver and then either the sender pushes the send buffer or the receiver pulls it in, or
- The receiver requests the sender to push the send buffer.

Either way, both processes need to exchange some explicit information in order to trigger the synchronous send. Most often, this occurs in the form of eagerly exchanging control messages.

### 1.2. Active Messages

The concept of *active messages* (AM) [Eick87 and ECGS92] was introduced, as a model to highlight the overheads of traditional send-receive models. Active messages where presented as an asynchronous mechanism where a message actively participates in its delivery to a remote node. The participation is defined in terms of a routine executed on the receiving node—a *handler*.

The complexity of the handler routine is dependent on the level of the utilised AM library. In order to match raw hardware performance and cut down wasteful cycles, early active-messaging libraries strived to generalise the networking hardware [Eick87, Mart94]. The handler may have to communicate directly with the network interface (NI) in order to extract the AM payload from it.

![Figure 1: The Request-Reply Active Messaging Model](image)

The most common active-messaging model is the *request-reply* model, which has been employed by a number of AM libraries [CKKL’95, MaCu95, Mart94, Mucc98]. Under this model, the AM transfer operation triggers a *request handler* at the remote node, which can trigger a *reply handler* at the origin (user code can also trigger the reply handler). This forms a handshake between the two communicating nodes. In Figure 1 we attempt to illustrate how the request-reply model works. The sender invokes the request handler remotely ([1] and [2]) to let the receiver prepare itself for the data transfer. Once the receiver is ready, it notifies the sender about itself being in a ready state ([2a] and [4]). Then the sender copies [4a] the data to the space of the receiver. The gray boxes indicate that [2a] and [4a took place from within [2] and [4] respectively.
These handler routines are standard procedures\(^1\) and can therefore be fed with parameters. This permits the communicating parties to exchange information prior to the data transfer event. Typically, the request handler prepares a local placeholder to accept the send buffer and uses the reply handler to communicate the address of the placeholder back to the sender. Ultimately, this eliminates buffering [ECGS92], as the send buffer can be offloaded directly to its destination, rather than temporary space.

Figure 1 demonstrates the data flow and execution of handlers in a rather restricted manner. Nobody dictates [2a] and [4a] to execute outside the scope of the relative handlers. For instance, if the receiver lacks a placeholder, it will perform [2a] in a later phase. How and when handlers are invoked is left entirely to the implementation of the active-messaging library.

Active-messages can complement single-sided communications. In single-sided communications, there is a single party actively transporting data, typically in a push (put) or a pull (get) fashion. Hence, the sender is always required to know the address of the remote memory location, where it is aiming to push data to [LWKW\(^03\)]. To pull data in, a receiver always needs to know the remote memory location where data resides. Therefore, although single-sided communications are very attractive, they require the code to know in advance the address of remote memory locations.

1.3. Motivation
In 1994 Cray Research Inc (CRI) implemented the MPI 1.1 standard for the Cray T3D supercomputing facility, and later on ported it to the T3E platform. Around 1997, the library was optimised by EPCC for the Cray T3E HPC service. Since 1999, the library was abandoned, in the main because of the shutdown of the Cray computer. This project is an opportunity to bring the library back to life on radically different hardware (HPCx). For this reason we will employ a low-level set of communication primitives (LAPI), and particularly its active-messaging interface. Although LAPI was introduced in 1998, very little has been contributed to the literature since then. Part of this effort aims at giving a better insight into the behaviour of the LAPI library on the HPCx platform.

1.4. Scope
The aim of this effort is to port the MPI 1.1 part of the T3D/E library to the HPCx supercomputing service by utilising the IBM LAPI communications library. The focus is placed on achieving this port in terms of the active messaging facilities of the LAPI. The collective communications have not been developed in a portable manner identical to the point-to-point ones, we therefore consider them as potential extra work. Further extra work covers the characterisation of the performance and experimentation with particular tunings of the library.

1.5. Preliminary Notes
The reader is expected to be familiar with the MPI 1.1 specification [MPIF95], although we will always stress semantics that are important in the context of a particular argument. Some familiarity is expected with networking terms such as the LPAR, the US switch,\(^1\)

---
\(^1\) In C for instance, one references these routines using a pointer to function. In Fortan, an integer identifier is more likely to be used.
etc. All referenced acronyms, and what they stand for, can be found at the end of this document (Section 8.1, page 53). In addition to that, some notational conventions are regularly used: text in courier font indicates the name of a subroutine; text in italics either stands for a variable, or indicates emphasis.

1.6. Outline of the Dissertation

In Section 2 we introduce the LAPI communications library, with a particular emphasis on its active-messaging infrastructure and auxiliary primitives. We move on by contrasting LAPI to other AM libraries and finally introduce a series of MPI libraries based on active-messaging.

Section 3 presents the layers of the T3D/E MPI library we are aiming to port. Most importantly, the major communication routines and protocols are reviewed to familiarise the reader with the T3D/E library and design decisions that were considered efficient for the Cray hardware.

Section 4 is dedicated entirely to our contribution. We present how we re-designed the communications layer of the T3D/E MPI library to form HPCxMPI. Emphasis is given on a number of optimisations performed to unleash the capabilities of LAPI.

Section 5 introduces a number of tunings and an analysis we performed in order to speed up the most important components of the communications layer. The purpose of this section is to reveal more about LAPI, show technicalities and explore a number of alternative ideas as well. This section is considered as extra work.

Section 6 is the post-mortem of this document. We summarise our contributions, point out some identified deficiencies and lay out future directions for the developer that is interested in optimising this library.
2. LAPI, AM Interfaces and AM-Based MPI Libraries

2.1. The LAPI Library

2.1.1. What is LAPI and What it Serves

The Low-level API (LAPI) is the lowest level user-accessible communications library supplied by IBM for its series of SP HPC services. It offers a set of active messaging (AM) and RDMA\(^1\) operations (RMC\(^2\), remote atomic and synchronisation). It provides the user with tools that are low-level enough to achieve low-latency communications between the processing elements of an HPC service. LAPI is reliable, discourages the user from interacting with the NI, has a flexible threads-model and is optimised for intra- and inter-node operation.

LAPI is reliable because it guarantees completion of its operations –even under high contention on the NI, LAPI operations will eventually complete. The user does not have to interact directly with the NI: i.e. tasks such as packetisation of outgoing data, reconstruction of data from incoming packets, packet re-transmission, packet extraction, ordering and flushing of NI buffers are all handled by LAPI. LAPI can be instructed to be aware of intra-node and inter-node (via the US switch) communications. It routes traffic through the US switch (NI), and can be configured to use shared-memory within HPCx nodes. This relieves the user from crafting its own LAPI-like API for intra-node communications. LAPI operates in two modes: with interrupts enabled or in polling mode. When the interrupts are enabled, active messages progress automatically –i.e., without the participation of the user. In polling mode, the user explicitly requests LAPI to probe for the active messages and progress them accordingly. For the request-reply model, progression would mean invocation of the request routine.

Although rather small, the LAPI interface is widely configurable and exposes a great degree of flexibility. The main components of this interface are:

**Active Messaging:** LAPI supports non-blocking asynchronous delivery of active messages. The routines `LAPI_Amsend/LAPI_Amsendv`\(^3\) initiate the delivery of active messages.

**Remote Memory Copy (RMC) Operations:** LAPI offers two main routines to facilitate copying of data to/from remote memory locations. The `LAPI_Put/LAPI_Putv` is a non-blocking asynchronous routine for *pushing* data into a remote memory location. The `LAPI_Get/LAPI_Getv` is a non-blocking routine for *pulling* data from a remote memory location into a local one.

**Synchronisation Primitives:** These primitives help LAPI processes to synchronise with the completion of communication events. There are two such primitives: *counters* and *fences*.

---

\(^1\) Remote direct memory access – routines to remotely manipulate data.

\(^2\) Remote Memory Copy – RDMA routines to remotely read/write data.

\(^3\) LAPI functions whose names are postfixed with the symbol ‘v’ indicate manipulation of non-contiguous buffers.
Remote Atomic Operations: The routines LAPI_Rmw and LAPI_Rmw64 are used to perform a number of remote atomic operations in a non-blocking fashion. Such operations are swap, fetch-and-add and fetch-and-or.

2.1.2. LAPI Synchronisation Primitives

LAPI RMC and remote atomic operations are designed to be non-blocking. LAPI_Put is in addition non-blocking and asynchronous. Being non-blocking means that when these routines are invoked, they initiate the operation and return immediately. This leaves the initiated operations in a volatile state until their local completion –i.e. in the meanwhile it is considered unsafe to alter the parameters passed to these operations. Processes must ensure completion of remote writes in their space, before that space is accessed. Also, it may be desirable for the writer process to synchronise with the completion of the write at the remote end.

The simplest form of synchronisation is implemented with local and global fences. Requesting a local fence (LAPI_Fence) blocks execution of the calling thread until all initiated operations complete. With local fencing, one can initiate a group of asynchronous operations, and block execution until the completion of the group. Global fencing (LAPI_Gfence) has a larger scope. It blocks execution of all LAPI processes until all data movement ceases [IBM98]. Fencing eventually forms data movement barrier. Being barriers, these two routines have a very large scope and cover the whole spectrum of initiated operations. This makes them suitable for collective, but very expensive for point-to-point patterns of communication. An alternative synchronisation mechanism uses LAPI counters.

A LAPI counter is an integer counter, such that for each completion of a communication event, e.g. the local completion of a LAPI_Put, the counter is incremented by one. The definition of a communication event is left to the programmer or the LAPI routine that utilises the counter. LAPI offers a set of routines that perform atomic operations on counters. LAPI_Getcntr and LAPI_Setcntr atomically fetch and set the value of a counter. LAPI_Waitcntr blocks execution of the calling thread until the value of the counter becomes greater or equal to a user-specified parameter, say $\beta$. Once LAPI_Waitcntr has unblocked, we know that $\beta$ communication events have completed. We will now describe how LAPI RDMA and remote atomic operations use and classify counters.

The routine LAPI_Put uses three types of counters: the local, target and completion counter –all being optional. The local and completion counters are both owned by the LAPI_Put invoking process, whereas the remote end owns the target one. Typically, referencing a remote counter is done by referencing its address –for C, that would be the value of a pointer. If the employment of a particular counter is undesirable, the value NULL can be used to indicate its absence.

When a local counter is specified, an entity called the LAPI dispatcher will increment it once the send buffer can be safely re-used. Similarly, if a completion counter has been specified, the LAPI dispatcher will increment it once the transfer has completed. Finally, the receiver’s LAPI dispatcher increments its target counter once the transfer has completed.

---

1 A process owns a counter if it is the one that allocates it.
completed. The completion and target counters signal each end upon completion of the event.

To build a blocking LAPI_Put, one specifies the local counter in the call, and waits for a single increment on it immediately after returning from LAPI_Put. To build a synchronous LAPI_Put, one specifies the completion counter and waits for a single increment on it immediately after returning from LAPI_Put. LAPI_Get and remote atomic operations use a single counter – the origin’s counter. A single increment on this counter indicates that the routine has completed as a whole. For a LAPI_Get it implies that the remote buffer has been successfully pulled in. In the next paragraphs we will demonstrate two samples of employing LAPI counters.

Figure 2 illustrates a simple use case of RMC operations with the involvement of a local and completion counter: process A wishes to profile the latency of launching an 8 Mbyte buffer to the US switch, and the achieved bandwidth of a synchronous bulk transfer. In step [1], process A starts timing the transfer. Process A invokes LAPI_Put [2] to initiate the RMC operation, while prior to that operation counters LOCAL and COMPLETION are both 0. In order to measure how long it takes to launch the buffer in the US, we wait until LAPI signals us that it is safe to modify the buffer. This signal occurs as an implicit increment of the LOCAL counter. We invoke the blocking LAPI_Waitcntr [3], which will unblock once the counter has reached the desired value. Before LAPI_Waitcntr unblocks, it decrements the counter by 1 setting it to 0. We store the desired latency [4] and then we wait for transfer to complete [5]. Doing this is similar to [3], but we are using the COMPLETION counter instead. Once the whole buffer has been transferred, the process B’s LAPI dispatcher issues an implicit message that increments process A’s COMPLETION counter. Finally, we unblock from [5], store the timings [6] and repeat the same procedure from [1].

![Figure 2: LAPI Counters Example A - The Use of a Local and a Completion LAPI Counter](image)

Figure 3 illustrates another example of how counters can be used – a ping-pong. Process A initiates the transfer of the ping payload by invoking LAPI_Put [1]. In this case, LAPI_Put is only handed a reference to the TARGET counter that belongs to process B. Once the transfer has initiated, process A blocks by waiting (LAPI_Waitcntr, [4]) for an update of its TARGET counter. Process B begins execution by immediately blocking until its TARGET counter reaches the value of 1 [2]. This may
look like raising the potential for a deadlock case, but is a perfectly valid combination. Although LAPI_Waitcntr blocks the execution of the calling thread, in the meantime it does let the communications and their effects to progress. Hence, process A is not likely to enter a deadlock state just because process B has not initiated the desired transfer. Once LAPI_Waitcntr unblocks, process B initiates its transfer [3] identically to [1], but now references the process A’s TARGET counter. Once the pong payload has been transferred to process A’s space, the LAPI dispatcher that belongs to A updates the local TARGET counter to 1 and finally unblocks the blocked LAPI_Waitcntr [4].

2.1.3. The Principal Mechanism for Active Messages

The LAPI routine LAPI_Amsend is responsible for dispatching an active message to a specified destination. It consists of the header handler specification, the message and a combination of counters with functionality identical to that for LAPI_Put. In fact, LAPI_Amsend appears to consist of an implicit request-reply handshake followed by a LAPI_Put.

A LAPI active message consists of the header handler specification (address of a function and a reference to its parameters), a reference to the send buffer and a combination of counters as seen in LAPI_Put. The header handler specification and the send buffer comprise the AM payload. When LAPI_Amsend is invoked, the LAPI dispatcher splits the AM payload into multiple packets and sends them in an unordered manner to the destination node. The first packet is the one that holds the header handler specification.

When that packet hits the receiver, LAPI grabs the header handler specification and executes its contents. The said execution consists of invoking the handler routine for the specified parameters. The header handler is required\(^1\) to return the address of the placeholder to the LAPI dispatcher. Once returned from the handler, the LAPI dispatcher re-assembles the send buffer by placing its packets in the placeholder. Although the header handler specification is allowed to be quite large (almost 1K), its purpose is not to substitute the send buffer.

There is more to LAPI than this. The header handler is allowed to define a completion handler specification. This is another routine and parameters pair, which the LAPI dispatcher will execute upon completion of the AM transfer. The header handler specifies the parameters to be fed into the completion handler. For instance, the parameters may be the address of the placeholder the header handler returned to LAPI.

\(^1\) Unless the send buffer is empty (NULL). If empty, then the LAPI_Amsend is more like a remote procedure call (RPC).
Hence, an occurrence of the header handler can be associated with a completion handler, but more significantly, a particular transfer can be associated with the completion handler. For the receiver, this offers substantially more flexibility than just waiting for an update on the target counter, as completion handlers can be tied with particular communication events. This is because a counter keeps accumulating completion of communication events in a blind manner—it knows that some events completed, but cannot differentiate between them.

We described what (invocation of handlers) occurs at the destination node due to some AM and when (state of the communication event). We have not mentioned the thread semantics that are associated with them, nor the roles of the NI and inner LAPI entities in the whole process. We will cover these later in this chapter. From now on we will be using a convenient notation to describe the payload of an AM:

\(<\text{header handler routine, parameters}, \text{send buffer}, (L,T,C)\)>

The presence of the characters L,T and C indicates which LAPI counters the LAPI_Amsend is employing (L=local, T=target and C=completion). For instance:

\(<\text{foo,buf1}, \text{buf2}, (L,-,-)\)>

This describes an active message, with the following components: the header handler is the function named foo and its parameters are contained in buf1; buf2 is the send buffer; and only a local counter is used.

### 2.1.4. How LAPI Reacts to Incoming Active Messages

Figure 4 describes the threads model LAPI uses for its active messages and how AMs are delivered and processed, and is heavily based on [IBM98], p. 456. While the authors of [IBM98] document the threads model, their documentation refers to the version of LAPI developed for the RS/6000 platform. Some of the following are mainly based on impressions we received about the behaviour of LAPI throughout the development of HPCxMPI. When LAPI bootstraps, it spawns two threads that remain alive for the whole lifetime of the application\(^1\)—the notification handler thread and the completion handler thread.

\[1\] Unless termination of LAPI is requested explicitly (LAPI_Term).

![Figure 4: LAPI Threads Model [IBM98]](image-url)
The **LAPI dispatcher** is an entity that, in addition to other tasks\(^1\), it implicitly looks for arriving packets that denote initiation of an AM delivery. Its purpose is to extract them from the NI, execute their contents and let them and their effects progress. If such packets are found, the dispatcher extracts their header handler specification and executes it. If the header handler registered a completion handler specification, then the LAPI dispatcher will queue it up for execution once the AM transfer has completed. When the dispatcher is done with the header handlers, it signals ([2a] and [2b]) the completion handler thread. When not executing, the completion handler thread is conditionally waiting [4] for a signal. If a signal is delivered (i.e. transfer is completed and there exists a completion handler), the thread looks for queued completion handler specifications [5], and executes them before entering the wait state again. As the dispatcher is the one who executes the header handlers, we can say that the dispatcher progresses the active messages.

How do the two threads we mentioned fit in this model? The completion handler specifications are always executed under the completion handler thread, while the header handler specifications under the notification handler thread. Consequently, a context switch occurs every time either of the handlers is invoked. Both threads wait when there is nothing to do. On the other hand, if the interrupt is caused while the dispatcher is executing under the scope of the main thread, then it absorbs the interrupt and executes handlers from the main thread instead. Context-switches can be reduced if it happens that at the time the interrupt is caused, the dispatcher happens to be active.

LAPI is designed to execute the dispatcher in almost every LAPI routine, regardless the operating mode—with interrupts enabled, and in polling-mode. When interrupts are enabled, arriving active messages generate interrupts. Such an interrupt forces the kernel to issue a wake-up notification to the notification handler thread. Once it wakes up it executes the LAPI dispatcher, which proceeds as we discussed in the previous paragraph. When operating in polling-mode, an arriving packet causes no such interrupts at all. The user has to manually invoke the dispatcher. LAPI offers the routine `LAPI_Probe` to probe for active messages. In polling mode a context switch cannot occur due to an AM arriving, as we will always discover the active messages using a dispatcher operating from the main thread.

### 2.1.5. LAPI Handlers, Counters and Threads (Examples)

We have described the basic synchronisation primitives, how threads and AMs relate and what the components of an AM are. We would like now to present two examples of combining LAPI handlers and counters. We have the interrupts enabled in both examples.

With respect to Figure 5, the sender process is about to transfer a buffer to the receiver process, via a LAPI AM. Prior to the transfer initiation, the sender’s counter LOCAL is set to 0. The sender passes a reference of the LOCAL counter to `LAPI_Amsend` and invokes it to fire the AM [1]. The LAPI dispatcher segments the send buffer into multiple packets, and sends them to the receiver. Once the first packet hits the receiver’s NI, the NI causes an interrupt. The interrupt leads the LAPI dispatcher to

---

\(^1\) Such as the disassembly of the message into multiple packets.
extract the handler specification from the NI. Once extracted, the dispatcher enforces some form of resource locking and under that scope [3] it invokes the header handler.

The authors of [IBM98] do not specify precisely what is being locked. This protection of resources is necessary to guarantee the threads-safety of the current LAPI context\(^1\). Within the same context, LAPI is threads-safe. While, the scope of the LAPI dispatcher is re-entrant and threads-safe, there are no implied guarantees about the threads-safety of the header-handler(s). That is, the locking inherited from the LAPI dispatcher may lead to header handlers executing one at a time, but not necessarily.

![Figure 5: LAPI Handlers, Counters and Threads - I](image)

The header handler allocates a buffer [4] and returns it to LAPI. Now it is time for LAPI to offload the send buffer into the receive buffer [5]. Mark [6] has been intentionally placed where the arrow of the bulk transfer meets with the column of the main thread. It indicates that the bulk transfer overlaps with any processing performed by the main thread. Once the transfer has completed, the kernel wakes up the completion handler thread and switches context to it. In contrast to the notification handler thread, the LAPI dispatcher does not maintain any resources locked while the completion handler executes. This is considered advantageous, as it allow the completion handler to make LAPI calls, blocking or not.

Similarly to the previous example, in Figure 6 (next page) the sender is about to transfer some buffer via an AM. This time, the sender’s counter COMPLETION and the remote counter TARGET are referenced instead. We do this so that the sender can synchronise with the completion of transfer. The header handler does not specify a completion handler, which implies that the receiver needs to manually track the completion of the communication event. Assume that prior to the execution of the header handler, the sender’s TARGET is set to 0. Once the AM has been delivered, the LAPI dispatcher increments the receiver’s TARGET counter and sets it to 1.

---

\(^1\) The LAPI context is similar to the MPI communicator.
After having that counter incremented, the dispatcher of the receiver sends an implicit message back to the sender in order to increment its COMPLETION counter. Meanwhile, the sender blocked \[8\] at some point, waiting for its COMPLETION counter to reach 1. The sender is free to overlap the transfer of the AM with some computation. In \[7\], the receiver decides to block on its TARGET counter. Although the blocking wait blocks execution of the caller thread, it does not block the progress of the communication event. Hence, equivalent behaviour would occur if \[7\] had taken place before \[2\]. This implies that it is safe to block for transfers before the invocation of the associated header handler.

2.1.6. Contrasting LAPI to Other AM and AM-Like Libraries

We would like to introduce elements of other AM libraries [CKKL’95, LaPC98, PLBK’98], which either exist in or are missing from LAPI. More about these AM libraries and their relation to the underlying hardware, is presented in [Vivo01].

2.1.6.1. Controlling the Progress of Active Messages

LAPI provides no further control on the progress and state of its data transportation facilities (LAPI_Amsend, LAPI_Put and LAPI_Get) –just their initiation. Once such a routine is initiated there is no way the user can exercise any control over the data transfer. This is contrary to the approach followed by FM (Fast Messages 2.0, [LaPC98, PLBK’98]). FM uses the concept of a message stream. The transfer of an AM consists of opening a stream to the destination (FM_begin_message), transferring the AM to it in a multi-part fashion (FM_send_piece), and finally shutting down the stream (FM_end_message).

FM uses FM_send_piece to gradually send a buffer to the remote end. LAPI does send the AM packet in a multi-packet fashion but transparently to the user. Opening the stream permits the execution of a header handler at the remote end. FM header
handlers do have finer access as they can *talk* to the NI. In fact, header handlers are passed a reference to the stream and are allowed to perform data retrieval operations on it. An important feature is that, a FM header handler is allowed to open a new stream. Contrary to that, the LAPI header handler cannot initiate a data transportation routine from within the header handler (e.g. a LAPI_{Put}). On the other hand, the LAPI completion handler is allowed to do so, as it is not executing under the scope of the LAPI dispatcher.

LAPI shares a similarity with the FM library. Both will probe the NI when in polling mode. With FM, the user invokes the FM_{extract}, which reads bytes from the NI and executes the discovered header handlers. When the LAPI library operates in polling mode, the user invokes LAPI_{Probe} to *probe for incoming* AMs, i.e. forces the LAPI dispatcher to examine the NI/its own state and progress active messages.

Streaming capabilities come with an additional advantage. The user is the one who shuts the stream down once all pieces are sent to their destination. Therefore, the user knows throughout all periods of computation, *what* data can be safely reused. Moreover, the user knows exactly which pieces of the buffer can be altered.

### 2.1.6.2. Tracking the Local Completion of AMs

The NX message-passing library follows a different approach. NX MP provides the user with two functions —the hsend and hrecv, which both accept a user handler. The function hsend sends a buffer and executes the user handler when the buffer can be safely reused, whereas hrecv receives non-blockingly a buffer and invokes the handler upon receipt of the whole buffer. Under this model, it is the system that *calls back* the user code upon completion of a *local* communications event—a facility not provided by LAPI. A similar callback facility is offered by GAM (Generic Active Messages, [CKKL’95]) as well, which assists the completion of asynchronous bulk transfers\(^1\) (am_store_async).

LAPI requires the user to explicitly query the local counter in order to find out the local state of some event. The receiver can employ the completion handler as a callback facility upon the completion of an AM. Unfortunately, the completion handler is only available for AMs, and not for the LAPI RMC operations LAPI_{Put} and LAPI_{Get}. Finally, a multiply referenced LAPI counter non-distinguishably accumulates the completion of multiple events, therefore blurring the state tracking of individual events. Surely, the user code could spawn a background thread to handle the task of examining LAPI counters and calling back accordingly. This of course may be undesirable since the user code already has multiple threads interfering with the main one.

### 2.1.6.3. Explicitness of the Request-Reply

Another interesting point is regarding *who* actually initiates the transfer of the AM payload, i.e. how explicit is the *request-reply* handshake. We stressed the semantics of controlling the progress and did mention that LAPI allows the user to initiate the AM delivery only. Other AM libraries, such as GAM, have an explicit request-reply; to copy data without knowing the receiver’s placeholder in advance, the sender invokes the request handler at the receiver (am_request_*). The receiver’s request handled finds a

---

\(^1\) Note that NX MP uses such callback to assist non-blocking sends, not asynchronous sends.
placeholder and the receiver\(^1\) communicates its address back to the sender, by forcing the invocation of the reply routine at the sender’s side (\texttt{am\_reply\_}). Finally and once the sender has collected the address, it pushes the data using any of the RMC routines (\texttt{am\_store\_}).

Contrary to that, LAPI performs the request-reply session internally. Under LAPI this session completes once the header handler routine returns, making it impossible for the receiver to postpone\(^2\) the operation. Computationally intensive header handlers would then slow down the completion of the request-reply – the main reason why LAPI suggests fast handlers. The sender (AM initiator) is only required to fire the AM; LAPI manages the movement of data. It is not specified when the data movement begins. I.e., LAPI does not say whether the packets of the AM payload leave the sender once the placeholder buffer becomes available, or whether some packets are buffered at the NI/LAPI space and copied in the placeholder once it becomes available.

### 2.1.6.4. Parameters Passing to Handler Routines

The * character in the GAM request/reply routines indicates the size/type of arguments passed to the routines. For instance, the routine \texttt{am\_request\_2} indicates that the request handler accepts two integer parameters. In contrast to that, LAPI allows the user to pass almost one LAPI packet (~ 1K) of parameters to the header handler. This indicates that the first packet of the AM payload, the one that triggers the header handler, can pass information to the handler.

### 2.2. Existing AM-based MPI Libraries

#### 2.2.1. MPI-AM

MPI-AM [CCHE96] is an MPI library derived from MPICH and has implemented ADI\(^3\) for the SM AM active messages library (GAM for IBM SP RS/6000). It has been developed for the IBM RS/6000 service. The GAM active messages specification [CKKL\(^4\)95] uses the request-reply active-messaging paradigm (see Section 2.1.6). The request-reply resembles a handshake between the two ends, therefore it can easily form the foundations for a rendezvous style MPI protocol. While using the request-reply for a rendezvous sounds ideal, its major disadvantage is that the data transfer has at least the latency of the handshake session.

The developers of MPI-AM realized this issue and crafted a new protocol to address it. This protocol is called \textit{hybrid buffered/rendezvous}. The idea is simple: part of the outgoing buffer’s payload is eagerly pushed in a remote pre-allocated buffer while the handshake is taking place. Then, once the handshake is over, the rest of the payload is normally transferred to the acknowledged buffer. Finally, the availability of the pre-allocated buffer determines whether the hybrid protocol should degrade to rendezvous or not.

---

\(^1\) This task can be carried out outside the request handler as well.

\(^2\) Possible, of course, under multiple LAPI transactions. For instance, we could build an explicit request-reply using two active messages and a LAPI\_Put.

\(^3\) Abstract Device Interface – an MPICH layer that implements a set of basic communication primitives. MPICH is typically ported to new hardware by just rewriting this layer.
2.2.2. MPI-LAPI

MPI-LAPI was an effort to port the MPL (IBM message-passing library) using the LAPI interface on the RS/6000. The authors of [BBGP01] give an interesting insight into how one goes about implementing the eager and rendezvous protocols for MPI using LAPI active messages. To perform an MPI_Send, the sender constructs an AM and sets its header handler specification to reference the Eager_hdr_hdl routine – i.e. a header handler for eager transfers.

Once the AM hits the receiver, Eager_hdr_hdl is invoked and is required to return the address of a placeholder buffer to LAPI. It moves on by looking up for a receive operation (MPI_Recv/MPI_Irecv) matching this send. If one is found, then the address of the user-space buffer is returned and a completion handler specification is set up (it references Eager_cmpl_hdl and is associated with the user-space buffer). When the AM payload is finally copied in the user-space buffer, the Eager_cmpl_hdl routine is invoked and marks the user-space buffer as having its associated transfer completed. This signal notifies the receive operations about the completion of the data movement. Ignoring the signal, would lead to receive operations returning control on truncated user-space buffers.

On the other hand, a matching receive might not have already been posted at the time Eager_hdr_hdl is invoked. Therefore a user-space buffer is not available and MPI-LAPI grabs an alternative one, called an early-arrival (EA) buffer. This is where LAPI will store the incoming data, and is a pure form of buffering. Then, the EA buffer is being associated with the send operation, which triggered the Eager_hdr_hdl, and both are introduced to the early arrivals queue \(1\) (EA_queue).

Again, the same form of completion signaling (via a completion handler) is used. At a later phase and when the matching receive is fired, it scans the EA_queue for the matching send and waits for the completion signal. Finally, the authors present a set of optimisations the most important being the replacement of completion handler with waits on counters.

2.2.3. MPI on the Fujitsu AP1000+

This [SiHa96] MPI library has been developed for the Fujitsu AP1000+ service – an HPC service where processing elements are called cells. The library uses parts of the MPICH (datatypes and topology), while the rest is architected on the single-sided operations supplied by the CellOS operating system. We are particularly interested in this library for two reasons: (i) CellOS and T3D/E UNICOS only support single-threaded processes only, and (ii) because the library operates in three modes: polling, with interrupts enabled and int-polling.

In polling mode, the library’s non-blocking MPI routines will poll the NI for incoming data, whereas the blocking ones (e.g. MPI_Recv), will block and poll continuously until some condition unblocks the blocked MPI operation [SiHa96]. This mode does work well, because concurrency is naturally avoided. With interrupts enabled, when data is remotely pushed into a cell, an interrupt is generated. The interrupt leads to the invocation of a particular system handler routine, which deals with the data. As the

---

\(1\) This queue maintains unmatched send operations.
authors mention, this is not very desirable because of extra computation required to manage concurrency between the handler routine and the MPI-internal computation.

The *int-polling* mode is based on the availability of mode switching (polling ↔ interrupts) at runtime, which LAPI supports as well\(^1\). User-space computation and non-blocking MPI operations execute with interrupts turned on. Blocking operations will always block interrupts until the operation unblocks. This trick has a particular purpose: if a handler is executed in parallel with a blocking MPI operation, then concurrency emerges. By turning off interrupts, we guarantee that the MPI library can safely alter its structures. Once the blocking MPI operation returns control, all of the accumulated interrupts fire the header handlers.

Contrary to that, LAPI can only enable or disable interrupts. This has a particular disadvantage. Let us assume that we do not know what is arriving at our node and that whenever we turn interrupts off, we probe (*LAPI_Probe*) for AMs. Then, every time we switch to interrupts, we would have to re-probe in order to ensure that no AM was left dangling between the mode-switch interval.

\(^1\) In LAPI, the mode is an environmental variable that can be changed at runtime (*LAPI_Senv*).
3. Introduction to the CRI/EPCC T3D/E MPI Library

In this section we familiarise the reader with the T3D/E MPI library –mostly with the part of the library we will be porting. The library uses the layers cake approach, i.e. multiple layers abstracting a set of functionality and being stacked one on top of the other. The layers have been intuitively separated according to the major sections of the MPI 1.1 specification. Most of the layers have an obvious purpose. The Context Creation layer manages MPI communicators, whereas Generic offers auxiliary routines (e.g. debugging).

As we can see (Figure 7), the point-to-point part of the MPI library is architected over the BASIC COMMS. Basic Communications (BASIC COMMS) exports primitives for the transportation of control information and buffers, and implements them in terms of SHMEM and T3E E-registers. Contrary to point-to-point layer, the collective communications implement a large set of functionality by directly referencing SHMEM in addition to BASIC COMMS. The Environmental Management deals with querying/setting the MPI environment, timers (MPI_Wtick/time), etc. and uses some UNICOS system routines. In the following paragraphs we review how BASIC COMMS works, the major structures and the interactions of Point-to-Point with BASIC COMMS.

![Figure 7: The Layers of CRI/EPCC T3D/E MPI](image)

3.1. Facilities Offered by the BASIC COMMS Layer

The BASIC COMMS layer equips the rest of the layers with a control information and data transportation facility. It is the responsibility of the library to make an efficient use of this layer.

Control information appears in the literature, mostly under the term control/protocol messages or envelopes. In this document we use the term protocol messages and denote it by p/messages. P/Messages can be viewed as placeholders for the information that accompanies MPI operations, such as communicator reference, datatype, etc. In T3D MPI, a p/message will typically hold the classification of the MPI operation, the parameters of that operation and internal information. The central purpose of p/messages is to facilitate matching of MPI operations (e.g. an MPI_Send with a relative MPI_Recv).

A collection of p/messages reflects the order in which a number of MPI operations were initiated. Maintaining the spawned p/messages in-order with their associated operations is required so that the library abides by the synchronisation semantics of the MPI operations. Consequently [CaCS95], p/messages are maintained
and consumed in a FIFO manner in a structure called the P/Messages Queue – PMQ. The routine \(\text{BC PUTP}(\text{trg}, \text{psg}, \text{flag})\) places the p/message \(\text{pmg}\) in the PMQ that belongs to the PE with id \(\text{trg}\) and returns success of the operation in \(\text{flag}\). This routine does not guarantee delivery of a p/message.

The MPI library typically combines a set of p/messages and data transportation routines to build what is called a protocol. A protocol defines when an MPI operation can be safely considered as completed, and is always triggered by an initiator p/message.

Due to the single-sided nature of SHMEM operations, the sender of a p/message cannot notify the receiver upon the placement of the p/message in its queue. The receiver must check that for itself. Hence, the protocol cannot be triggered immediately. For that reason, the whole library is designed to execute in polling mode, i.e. it polls the PMQ for arrived p/messages. The discovered p/messages are removed from the queue and then fed into the handler code for processing (\(\text{BCC HANDLER}()\)). For that reason we say that the handler absorbs p/messages from the PMQ. The routine that absorbs p/messages is called \(\text{BC ABSORB}()\).

Data will most often be either a flat or a packed non-contiguous\(^1\) user-space buffer. User-space buffers are buffers referenced by the user in MPI operations, such as \(\text{MPI Send, MPI Irecv, MPI Scatter, etc.}\) Packing refers to temporarily copying a non-contiguous memory block into a contiguous placeholder, so that it can be transferred as single piece of data. As we have already said (Section 1.1, page 1), transferring a buffer does not necessarily imply sending a buffer. Buffers can be either pushed by the sender to a remote process, or pulled in by the receiver. The routine \(\text{BC PUTPD}(\text{trg}, \text{from}, \text{to}, \text{flag})\) copies the contents of the address \(\text{from}\) in the address \(\text{to}\) of the remote process \(\text{trg}\). Similarly to that, \(\text{BC GETD}(\text{org}, \text{from}, \text{to}, \text{flag})\) copies the contents of process \(\text{org}\)’s address \(\text{from}\) in the local address \(\text{to}\). Both return success of the operation in the parameter \(\text{flag}\). Here is the API of BASIC COMMS:

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{BC INIT}</td>
<td>Bring the BASIC COMMS layer up and running.</td>
</tr>
<tr>
<td>\text{BC_FINALIZE}</td>
<td>Terminate the BASIC COMMS layer.</td>
</tr>
<tr>
<td>\text{BC PUTP}</td>
<td>Place a p/message in a remote PMQ.</td>
</tr>
<tr>
<td>\text{BC_PUTPD}</td>
<td>Push data in a known remote buffer.</td>
</tr>
<tr>
<td>\text{BC GETD}</td>
<td>Pull data from a known remote buffer into a local one.</td>
</tr>
<tr>
<td>\text{BC ABSORB}</td>
<td>Toggle queues and dispatch arrived p/messages for processing.</td>
</tr>
<tr>
<td>\text{BCC HANDLER}(^2)</td>
<td>Handle (high-priority) arrived p/messages.</td>
</tr>
</tbody>
</table>

Table 1: The BASIC COMMS API

### 3.2. Inside the BASIC COMMS Layer

The T3D version of the BASIC COMMS library is entirely crafted in terms of SHMEM operations, whereas the T3E version moves a step forwards and uses the highly optimised e-register operations as well. In the following paragraphs we present the p/message and data transportation routines and their accompanying cost in terms of SHMEM operations.

\(^1\) And/or of a complex data type.

\(^2\) Not part of the BASIC COMMS, but frequently referenced throughout this document.
In addition to that we present the main entities and structure that play a central role in BASIC COMMS. The main characteristics of this layer are:

- P/Messages are maintained in queues.
- The MPI library progresses by polling the queues frequently aiming at finding and processing incoming p/messages.
- RDMA operations are used to place p/messages in a remote queue.
- RDMA operations are used to protect the layer from race issues.
- Both push and pull modes are supported for data transportation.

### 3.2.1. The P/Message

The Cray T3D SHMEM RMC operations are designed not to invalidate the cached memory space, whereas the T3E version does [CRI99a]. This leaves the task of cache coherence maintenance to the programmer [CaCS95]. To ease the invalidation task, the p/message was set to be 64 bytes long—equal to the length of a T3D/E cache line and the T3D network packet. The approach of fixing recycle-able structures to match the size of the cache line size is encountered in other MPI libraries [SiJa02] as well.

![Figure 8: The Structure of a P/Message](image)

### 3.2.2. The P/Messages Queue

The purpose of the p/messages queue (PMQ) is to accumulate the incoming p/messages. The T3D/E version has set the PMQ to contain a total of 64K static FIFO buffers, split into two ends (insertion and process), capable of accommodating 1,024 p/messages. Although there are two FIFOs, only one can be active at a time—the insertion end, while the inactive end is being processed by the handler code. These FIFOs may contain p/messages originating from any process. The active FIFO is left open for the placement of p/messages in it, while the inactive FIFO is being processed by `BCC_HANDLER()`. When `BC_ABSORB()` is called, it toggles FIFO activity and fully processes the inactive end in one pass. `BC_ABSORB()` is a non-reentrant mechanism carefully invoked upon entry or exit from MPI operations.

The PMQ maintains a structure named `mqcw`—the messages queue control word. The `mqcw` encapsulates the id of the active FIFO and its last available slot, where a p/message can be inserted. This entity is atomically manipulated by either the PMQ owner process, or remote processes who place p/messages in the PMQ. More about the concept of toggling routing of incoming messages by switching buffers can be found in [XHLM98 and LuCu98].

Safeguarding access to the PMQ uses SHMEM primitives, therefore we would like to describe how it works. The said safeguarding refers to how processes utilise SHMEM to lock the PMQ of the receiver—a procedure carried out by `BC_PUTP` and `BC_GETP`.

---

1 First-In First-Out – describes how the elements of a queue are consumed.
2 Each end accommodates 512 p/messages, therefore 2 x 512 x 64 = 64K.
The variable _T3DMPI_lock_ref holds the status of the lock. The lock is a simple integer, which is considered acquired whenever its value is equal to the constant QLOCKED. When it is not locked, its value is the value of the remote _mqcw. Then, a process tries to acquire a lock on the remote _T3DMPI_lock_ref by attempting to set it to QLOCKED.

The SHMEM routine shmem_swap performs an atomic swap of a remote variable with a local one. In particular, long shmem_swap(rvar, lvar, trg) sets trg's variable rvar to the value of the local lvar and returns the original rvar. With respect to Figure 9 (next page, for the moment ignore [5]): In order to acquire the lock, one repeatedly swaps the remote _T3DMPI_lock_ref with QLOCKED, until shmem_swap returns a value ≠ QLOCKED ([2] and [3]). The returned value [4] is now our local copy of the remote _T3DMPI_lock_ref, or if you will, our local copy of the remote _mqcw.

Therefore, if somebody has already locked the remote queue, the value of the remote _T3DMPI_lock_ref is equal to QLOCKED and swapping only returns QLOCKED. Then, updating ([6] and [8]) the remote _T3DMPI_local_ref serves two purposes: (i) updates the _mqcw of the remote PMQ, so that it points to the next slot, and (ii) unlocks the queue. This illustrates that, locking a remote queue requires at least one SHMEM operation, whereas unlocking the queue always takes a single one.

A BC_ABSORB that is repeatedly invoked until a condition is met is said to be a blocking BC_ABSORB (b/BC_ABSORB). A blocking BC_ABSORB may give the impression that it slows progression /responsiveness of the MPI library by wasting cycles waiting for a condition to be satisfied. This is not true because there is a single thread of execution, the procedure of discovering p/messages cannot run concurrently with, say, the MPI user code. In a multi-threaded environment for instance, one could have a separate thread examining the PMQ in regular intervals. We should say that blocking the full process does not prohibit other processes from completing SHMEM RMC in its space at all.

It will be shown (Section 3.2.4, page 23) that there are occasions, where a p/message forces BC_ABSORB to blocking-ly push data to a remote process. Note that this is different from b/BC_ABSORB because we have a single invocation of BC_ABSORB, which although temporarily blocked, will eventually return. While BC_ABSORB is blocked waiting for the completion of the push operation, it does not process the rest of p/messages in the inactive queue (if any). Practically, this is not an issue as remote writes in our space still proceed uninterruptedly.

Selectively processing p/messages would leave the array with holes and would require some form of bookkeeping to free-link the unoccupied slots.

To conclude, wherever a blocking BC_ABSORB occurs, it is done so because it maintains semantics and is the only thing left to do. Typically, synchronous MPI operations will block BC_ABSORB-ing at their exit point (e.g. MPI_Recv, MPI_Ssend), whereas the non-blocking ones will invoke BC_ABSORB once either at their entry or exit point (e.g. MPI_Isend). Polling once or repeatedly from within the MPI send/receive routines appears to be common [CCHE96], and sensible action for single-threaded MPI libraries [SiHa96].
3.2.3. The BC_PUTP routine

BC_PUTP (put p/message) is the mechanism that places a p/message in the active end of a remote PMQ. We have already described the action undertaken to remotely lock the PMQ, so that BC_PUTP can place a message in it. We said that this will take a single SHMEM operation, at best. Then, BC_PUTP maintains the queue locked for as long as it takes to remotely copy the p/message in it. The queues must not be toggled prior to p/message copy completion. Such an action would result in truncated p/messages leaking to the handler code. The fact that only a single BC_PUTP can be writing to a remote PMQ at a time implies that p/message transfers are serialised and therefore they cannot overlap. This forces BC_PUTP to be a globally synchronous operation as it forms an implicit global barrier for the rest of the BC_PUTPs.

With respect to Figure 9: once a process locks the remote queue, it uses shmem_put to copy [4] the p/message in its last slot. The address of the last slot is the value of the local copy of the lock (mqcw, _T3DMPI_lock_ref). The shmem_put routine is blocking but not guaranteed to be synchronous. This forces either of the communicating parties to explicitly wait for completion of the operation. The target can invoke the shmem_quiet() to wait for the completion of all shmem_puts aimed to it. The process that shmem_puts can issue the intrinsic _remote_memory_barrier() to wait until the completion. Once the p/message is written in that slot, the _T3DMPI_lock_ref is locally updated to point to the next slot [6]. Finally, BC_PUTP proceeds by unlocking the remote queue. At that point, BC_PUTP is considered to be completed.

Reviewing the rest of the MPI library which uses BC_PUTP, reveals that there are two expectations about the behaviour of BC_PUTP: (i) to be blocking, and (ii) p/messages are placed in the PMQ in-order with their relative BC_PUTP. The second requirement is about what order the handler code is expecting the p/messages to appear in the queue. To conclude:

- The cost of the routine is at least three SHMEM operations, depending on how long until PMQ is unlocked.

![Figure 9: The BC_PUTP routine](image-url)
3.2.4. BC_PUTD and BD_GETD
These two operations facilitate bulk transfer of memory blocks to/from remote processes – BC_PUTD puts data, whereas BC_GETD gets data from a remote process. The SHMEM routines used to push data between processes (shmem_put*) are highly optimised, and are specialised according to the size of the data. For instance, word memory blocks should be transferred by shmem_put32, whereas the double word ones by shmem_put64. Arbitrary-sized blocks can be always transferred by the shmem_put routine. The performance of the shmem_put32 varies according to the alignment of the referenced addresses.

Regardless of the alignment, BC_PUTPD will always pick up the most efficient shmem_put routine. Once a shmem_put is fired, BC_PUTPD waits for the transfer to complete (using _remote_write_barrier) similarly to BC_PUTP. The BC_GETD uses shmem_get* operations according to cache alignment issues, but will not issue a _remote_write_barrier. A major difference between shmem_get* and shmem_put* is that in the T3D service, shmem_get* has almost double the latency of the equivalent shmem_put* ([CRI99a, CaCS95]). This does not hold for the T3E version. In the next paragraphs we will see how this performance characteristic has been tackled by the protocols. To conclude:
- The routine BC_PUTPD uses two SHMEM operations, whereas
- The routine BC_GETD uses a single SHMEM operation.

3.3. Inside the Protocols
The T3D/E MPI library names protocols according to the sequence of operations they perform. There are three types of operations: R – request information, T – transfer, and A – acknowledge. Operation R refers to the exchange of protocol-specific information between the communicating parties in the form of p/messages. Operation T indicates that data transportation takes place, but not necessarily via BC_PUTD/BC_GETD. Operation A indicates either a reply to R, or finalisation of the protocol. These three may sound abstract, but we will be seeing their particular meaning for a number of reviewed protocols.

To differentiate between protocols and the operations of a protocol, p/messages are tagged with a pre-defined identifier. For instance, protocol RTA requests information by tagging a p/message with the MSG_REQ_SEND identifier. The handler code distinguishes between these identifiers and accordingly fires the rest of the protocol’s stages. The purpose of the following paragraphs is to show:
- How p/messages and data transportation are combined to build MPI protocols¹.
- Cost differentiation between eager and rendezvous protocols.
- An introduction to how MPI operations are matched.
- Existing forms of intermediary buffering.

¹ Eager and rendezvous.
3.3.1. Protocols T(1) and T(N)
The naming of these protocols, T(1) and T(N), is not accidental. It indicates that both protocols consists of a single transfer operation only. These two protocols are classified as eager protocols, because they send data that is always buffered at the receiver’s end. Therefore, the send buffer is always buffered in some pre-allocated space residing at the receiver. For the case of the T3D/E library, that space is the slots of the remote PMQ. When the remote active end of the PMQ is not full, BC_PUTP can always store a p/message in the remote queue’s space. Protocol T(1) eagerly transfers a buffer, by encapsulating it in the unused space of a p/message (see Figure 8, page 20). Based on the same principle, protocol T(N) exploits the unused space of the remote active FIFO, to copy a larger buffer in it, in the form of multiple p/messages. Hence T(N), i.e. N*T(1).

Here is what degree of buffering occurs for each protocol at each end:

- T(1) payloads, with a packed or non-packed send buffer, are always buffered locally in the p/message (\_T3D_MPI_pack_small in \_T3D_MPI_msg) and remotely in the space of the receiver’s PMQ, thus making it impossible to have zero-copy transfers.
- Under T(N), the T(N) payload is always buffered at the receiver and non-contiguous and/or complex send buffers are always buffered locally (\_T3D_MPI_pack_buffer in \_T3D_MPI_Tn_buffer).

The T(1) p/message tag is MSG_SEND or MSG_SSEND, for MPI_Send and MPI_Ssend respectively. MPI_Ssend advances the protocol to TA, i.e. the receiver has to acknowledge the sender when the matching receive operation is fired. Acknowledging consists of a p/message tagged MSG_ACK_RECV, which the receiver sends to the sender. Typically, and regardless of the protocol, an MPI_Ssend operation, sends the p/message and then blocks until the MSG_ACK_RECV is collected and processed. Data transportation that cannot be fulfilled by T(1)/T(N) requires a handshake arbitration [EPCC\(\alpha\)]. This handshake refers to how the two ends communicate in order to establish the addresses of their send/receiver buffers. It uses the operations R and A, and is materialised by the protocols RTA and RAT.

3.3.2. The RTA Protocol
RTA (request-transfer-acknowledge) is the original rendezvous protocol developed for the T3D MPI. RTA performs well for Cray architectures (T3E), where the shmem_get operation is not slower than the shmem_put (T3D). The R operation is used to communicate the address of the send buffer to the receiver, so that the receiver can pull the data into its receive buffer. RTA is used by both the MPI_Ssend and MPI_Send routines. The difference lies in that MPI_Ssend blocks for the completion of the protocol, whereas MPI_Send does not. MPI_Send initiates RTA by invoking the post_big_send, whereas MPI_Ssend by invoking the post_ssend. We will see that RTA acknowledging p/message (MSG_ACK_RECV) of the RTA serves two purposes: it unblocks the synchronous send and/or relinquishes resources used for local buffering.
Figure 10: The RTA Protocol

Figure 10 displays the flow of RTA. In step [1a], the sender constructs a p/message, tags it with the RTA initiator tag (MSG_REQ_SEND), and places it in the remote queue [1c]. At some point, the receiver issues a matching MPI_Recv [1b]. The MPI_Recv cannot return immediately, unless the match occurs and buffers are copied. When the MPI_Recv is called, it uses post_recv to introduce a local receive event notification to the events queue—we refer to this operation as posting a (local) receive. When the handler processes a p/message, e.g. a send, it constructs an alien send event notification and introduces it to the queue as well. Therefore, there are two points where the events can be matched (holds for the other protocols as well):

- If the alien send was introduced before MPI_Recv, then it is matched by a later invocation of post_recv.
- If post_recv took place before the arrival of the p/message, then the local receiver will be matched by the handler code.

The entity (post_recv or handler), which achieves the matching progresses the RTA. For our example (Figure 10) we assume that the first case holds.

The MPI_Recv, which is blocking, is implemented as a blocking BC_ABSORB. It absorbs p/messages until the relative RTA initiator is found. A BC_ABSORB that discovers the RTA initiator will initiate the transfer [2] and invoke the _T3D_MPI_pt_get_buffer routine. This one sets up any local buffering required\(^1\) for the receive buffer and pulls the send buffer into it (step [2a], BC_GETD). Once the send buffer has been received, the receiver sends an RTA acknowledging p/message (MSG_ACK_RECV) back to the receiver [2a] as an indication of completion of the protocol. Then, some later invocation of BC_ABSORB will discover the p/message [3]. If the sender had locally buffered\(^1\) its send buffer, then the p/messages will allow it to relinquish its resources. If the RTA was due to an MPI_Ssend, then the MSG_ACK_RECV will also unblock the sender. RTA demonstrates how the rendezvous protocol uses extra communications when the address of the receiver buffer is not known.

---

\(^1\) In case non-contiguous data types are used.
3.3.3. The RAT Protocol

RAT is similar to RTA and is appropriate for architectures where the `shmem_put` operation is faster than the `shmem_get`, such as the T3D platform. RAT is designed for synchronous send operations, which cannot complete unless the receiver invokes its matching receive operations. This requires both ends to synchronise prior to the data transportation. RAT performs the synchronisation in a more explicit manner than RTA.

Similarly to RTA, the RAT initiator (MSG_REQ_SSEND) does not fire any reaction at the receiver’s side immediately, unless the matching local receive is posted. With respect to Figure 11, the RAT initiator is a p/message tagged with MSG_REQ_SSEND—that is, we request a synchronous send. This protocol introduces an additional issue in the flow. In the RTA protocol and for a synchronous send, the sender waits for the final ACK as means for letting it to unblock and relinquish resources. In RAT, the acknowledging p/message has the opposite direction, and acts as a signal for letting the receiver unblock its MPI_Recv.

The code that discovers MSG_REQ_SSEND [2] first reacts by constructing a new p/message; the MSG_CLR_RECV, and sending it to the sender process [2a]. This p/message completes the communication of the address of the receive buffer to the sender. The send buffer can be either the address of the user-space buffer (if simple contiguous type) or a temporarily allocated buffer. This is where the sender will place its send buffer once it receives the MSG_CLR_RECV.

![Figure 11: The RAT Protocol](image)

Once the sender’s BC_ABSORB discovers the MSG_CLR_RECV [3], it extracts the address of the receive buffer from it and pushes its send buffer into it using the _T3DMPI_pt_put_buffer ([3a], BC_PUTPD). Once the send buffer has been safely placed in the remote receive buffer, the sender constructs the RAT acknowledging p/message to complete RAT. This p/message is tagged MSG_ACK_SSEND and placed in the receiver’s PMQ [3b]. Finally, when the receiver processes the RAT acknowledging p/message [4], it releases resources and unblocks control.
4. The Design of HPCxMPI

In the previous section, we described the components of the BASIC COMMS layer, and how as a whole they comprise the communications core for the rest of the T3D/E MPI library. In this section we document how we ported BASIC COMMS using LAPI, what the driving forces were and how we dealt with the single-sided assumptions of SHMEM. Porting the Environmental Management layer was trivial, so we do not present it here. For our convenience, we have ported the MPI_Barrier routine without emphasising its performance at all, but will not cover it here either.

4.1. Why not Use TurboSHMEM?
The TurboSHMEM [Klep98 and Klep02] is a library developed by IBM to support legacy SHMEM applications on the IBM hardware. It implements SHMEM on top of LAPI and MPI operations. Technically, one could port the T3D/E MPI library by just linking it with TurboSHMEM. This is undesirable for two reasons mainly:

- SHMEM is ideal for shared-memory systems (e.g. T3D, p690 intra-LPAR), where accessing remote memory can be very fast. For inter-LPAR operation, the said memory access has to be routed through the US switch, which makes it considerably slow. Given for instance that the core routine BC_PUTP takes at least three SHMEM operations, using TurboSHMEM would clearly overkill performance.
- It is rather awkward to implement HPCxMPI on top of IBM MPI.

4.2. Localisation of Locking
In Section 3.2.3 (page22), we showed that the process of placing a p/message in a remote PMQ (BC_PUTP) requires at best three SHMEM operations. We also showed how a stream of BC_PUTP operations leads to serialisation of p/message placements and can delay the lock acquisition. Delayed lock acquisition leads to extra SHMEM operations, which can be only tolerated for intra-node operation. This is undesirable for inter-node communication though, where performance principles suggest minimising it as much as possible.

Therefore, by lock localisation we mean that the owner of a PMQ is its exclusive manipulator. The manipulation of the lock only occurs at two very specific points (BC_PUTP and BC_ABSORB), and is transparent to the rest of the layers. This is very convenient for the transition to local locking. We decided to use the pthread mutex mechanism, rather than building our own locking facility. Then, pthread_mutex_lock and pthread_mutex_unlock, lock and unlock the PMQ. In later sections we will see when and how locking is performed.

4.3. Porting the BC_PUTP Routine
The BC_PUTP routine facilitates insertion of a p/message in a remote PMQ, so that the p/message is inserted in-order with its relative BC_PUTP. BC_PUTP is a very important mechanism, as it underlies T(1), which is used by all of the MPI send operations. In the
T3D version, the sender process was assigned the full responsibility of correctly inserting the p/message in the remote queue.

In section 2.1.3 (page 9), we covered what the header handler specification stands for—a header handler routine, and a set of parameters passed to it. The process that fires an AM results in the remote LAPI dispatcher invoking the handler routine [at the remote end] and feeding it with the parameters. The purpose of the handler routine is to find a placeholder and return its address to the LAPI dispatcher. LAPI permits the header handler to do LAPI-unrelated computation prior to returning that address, but encourages such computations to be of low-latency. We can therefore define a header handler, say $HH_{BC\_PUTP}$, whose purpose is to return a placeholder for the incoming p/message to the LAPI dispatcher. Ideally, we would like that placeholder to be the next available slot in the PMQ, and we could equip the handler with localised locking to protect access. Our AM would then look like:

$$<(HH_{BC\_PUTP}, -), p/message, (L,T,C)>$$

![Figure 12: Two Consecutive T(1) MPI_Send](image)

With respect to Figure 12, the purpose of $HH_{BC\_PUTP}$ [2] is to allocate [2a] a 64-byte placeholder for storing the p/message and to return [2c] the address of the buffer to the LAPI dispatcher. In addition to that, it sets up a completion handler specification [2b] ($CH_{BC\_PUTP}$, allocated buffer), to be executed upon completion of the data transfer [3]. The completion handler $CH_{BC\_PUTP}$ [4] will then lock the PMQ [4a], copy the p/message from the allocated buffer in its last slot [4b], unlock the queue [4c] and return. $BC\_PUTP$ employs the completion counter in order to form a synchronous $BC\_PUTP$. Otherwise, $BC\_PUTP$ could race ahead and lead to the completion handler introducing p/messages to
the PMQ without respecting the order of their initiation. We form a synchronous
BC_PUTP by blocking for an update on completion counter C [1b] immediately after
sending the p/message. This counter will not be updated, unless the CH_BC_PUTP returns.
Once it returns, an implicit message updates the remote completion counter [5], and
allows [1b] to unblock.

Clearly, this flow is very expensive for both the sender and the receiver. The
receiver experiences double the amount of context switches due to the invocation of both
handlers –header and completion. In addition to that, an extra implicit communication
event occurs to unblock the sender. In the case where T(1) can be employed,
asynchronous communications (MPI_Send and MPI_Isend) degrade to synchronous
ones, due to the synchronisation imposed by BC_PUTP. The only good advantage of this
approach is that the handler routines are lightweight and it could be argued that they
approximate the cost of a minimal handler.

| HH_BC_PUTP  |
| 01: claim an OON placeholder |
| 02: populate OON(PR) given the PR |
| 03: lock the PMQ |
| 04: handle the OON(PR) |
| 05: unlock the PMQ |
| 06: return nil |

| BC_ABSORB  |
| 07: lock the PMQ |
| 08: toggle FIFO activity |
| 09: unlock the PMQ |
| 10: process the inactive FIFO (BCC_HANDLER) |

Figure 13: Handling and Processing a P/Message

Having shown the sources of overheads (synchronisation and context switching),
we would like to introduce an alternative and more lightweight approach. When we
contrasted LAPI to GAM (Section 2.1.6, page 13) we said that LAPI is capable of
delivering almost 1K of parameters to the header handler. We can encapsulate the 64-
byte p/message payload in the parameters space, and achieve passing of a completed
p/message to the HH_BC_PUTP routine. This assigns the responsibility of introducing the
p/message to the PMQ, to the HH_BC_PUTP. On the other hand, we are still faced with how
we can get rid of the implicit message. We do this by permitting out-of-order p/messages
to occur and deal non-natively\(^1\) with their re-ordering. This will be discussed in a later
section (4.5, page 33). Right now the reader should view the BC_PUTP AM payload as follows:

\[
< (HH_BC_PUTP, p/message), -, (L, -,-)> \]

Figure 13 lays out the action undertaken for the receipt and processing of a p/message,
say PR. The HH_BC_PUTP is specified by the origin at the AM construction time. The OON
entity is a placeholder for p/messages that may arrive out-of-order and its handling will

---

\(^1\) Native handling refers to synchronous delivery of p/messages.
be discussed in later sections (Section 4.5). Assume that step 04 will place the p/message in the PMQ. At a later time, and when \texttt{BC\_ABSORB} is called, it will atomically swap queues (toggle FIFOs, steps 07-09), and pass the p/message to the \texttt{BCC\_HANDLER()}. The body of \texttt{BC\_ABSORB()} we presented here is rather simplistic. Later on we will see that 07–09 have a degree of complexity associated with them. To conclude:

- \texttt{BC\_PUTP} uses a blocking asynchronous LAPI active message.
- \texttt{BC\_PUTP} inserts the p/message in the remote PMQ in a single LAPI transaction.
- \texttt{BC\_PUTP} only uses a header handler.
- P/Message movement completes at header handler invocation time.

4.4. The New Protocol P+D

4.4.1. Why a New Protocol?

T(1)/T(N) transfers and the flexibility of \texttt{LAPI\_Amsend} lead to the inspiration of a new protocol, the P+D –i.e. p/message and data. It achieves movement of control information (p/message) and send buffer in a single LAPI transaction, and exploits the capability of \texttt{LAPI\_Amsend}, which permits the transfer of arbitrary-sized buffers without knowing the destination receive buffer. P+D will always buffer the send buffer classifying this protocol as an \textit{eager protocol}. P+D is a mechanism which allows the point-to-point core (\texttt{MPI\_point.c}) to initiate transfers in a manner identical to T(1) (see Section 3.3.1, page 24).

4.4.2. Why Not Using T(N) Instead?

The logic underlying the T(1) protocol has the advantage of data placement in a single LAPI transaction. We have mentioned (Section 3.3.1, page 24) how T(N) eagerly transfer payloads larger than the T(1) payload, by exploiting the existing space of the remote queue. For the T3D version of the library, T(N) implementation did share a great degree of similarity with T(1). The most natural way to port T(N) would be to use the active messages of the form 4.3-[1]. On the other hand, there is no reason to consume FIFO space for eager transfers when we can allocate a required receive buffer in the header handler. In addition to that, T(N) assumes that the queues of the PMQ are contiguous blocks of memory (array of p/message placeholders) –something we plan to change. There is a more complicated issue here. The \texttt{BC\_ABSORB} routine expects the contents of the queues to be complete when it decides to process them. Populating the PMQ with pending transfers and devising an algorithm to skip them, leads us to the overheads mentioned in the PMQ structure discussion (Section 3.2.2, page 20).

4.4.3. The Design of P+D

While there is not much to say about how the sender initiates a P+D transfer (similar to \texttt{BC\_PUTP}), there is a significant level of complication at the receiver’s side. We should start by laying out the P+D AM payload:

\begin{verbatim}
)<(HH_{BC\_PUTPD}, p/message), send buffer, (L_{BC\_PUTPD}, T_{BC\_PUTPD},−)> [1]
\end{verbatim}

The transfer is initiated identically to T(1), and the routine responsible for doing so is called \texttt{BC\_PUTP}. The HH_{BC\_PUTPD} is the header handler, which will be triggered
once the first packet of the P+D payload hits the receiver. This protocol uses two counters: the local \( L_{BC\_PUTPD} \) and a remote target one, the \( T_{BC\_PUTPD} \). P+D receives and manages the p/message identically to T(1). It then claims a receive buffer and returns it to LAPI, while it leaves the data transfer ongoing in the background. The basic property of P+D is to overlap data movement with as much of the computation as possible – regardless if that is library-internal or user-space computation.

From [1], one derives that the only way the receiver can track completion of the transfer is by accessing its \( T_{BC\_PUTPD} \) target counter\(^1\). In Section 2.1.2 (page 7), we described the LAPI counter as a \emph{virtual device}, which accumulates the completion of communication events. Therefore, the receiver’s \( T_{BC\_PUTPD} \) accumulates completed P+D transfers. We also said that, event accumulation is performed in a blindly manner. Hence, \( T_{BC\_PUTPD} \) just knows \emph{how many} P+D transfers completed, and not \emph{which ones}. P+D manipulates the p/message identically to T(1) but the p/message has a pending P+D transfer associated with it.

The real issue lies in \( BC\_ABSORB() \). \( BC\_ABSORB() \) needs to ensure that all p/messages passed to the handler for processing have completed their pending transfers. This was one of the reasons for avoiding T(N). Let us look at a scenario that raises some complications: It is possible that the 2 P+D p/messages are distributed between the two queues because \( BC\_ABSORB() \) managed to toggle queues before the injection of the 2\textsuperscript{nd} p/message into the active end. Then, \( BC\_ABSORB \) has to wait for the completion of both transfers, i.e. wait for \( T_{BC\_PUTPD} \) to reach the value of 2. \( T_{BC\_PUTPD} \) reaching the value of 1 indicates that either of the transfers has completed –not necessarily the one associated with the p/message in the inactive queue.

One would ask, why there is not a separate \( T_{BC\_PUTPD} \) counter for each end of the PMQ, so that \( BC\_ABSORB \) only has to wait for transfers associated with the contents of the inactive queue? Then, toggling FIFO activity implies toggling counters. This is complex and expensive for the following reasons:

- The queues of the PMQ accumulate p/messages from any process, therefore changes on the counter should be broadcast to all –a collective operation.
- This collective operation expands to a global barrier because the owner of the target counters needs to ensure that, at the time counters are toggled, nobody is planning to initiate transfers that reference the old counter. I.e., we must cater for outdated references about the active counter.

### 4.4.4. Pending Transfers Routed Through the PMQ

We have decided to settle with a single \( T_{BC\_PUTPD} \) target counter, and introduce p/messages in the PMQ as early as they arrive\(^2\) in a T(1) fashion. To record the number of pending P+D transfers, we introduce a new non-LAPI counter \( C_{PENDING} \). \( C_{PENDING} \) is atomically incremented for each P+D \( HH_{BC\_PUTPD} \) invocation, i.e. for every arriving P+D AM. We use the lock of the PMQ to ensure atomic manipulation of the counter.

Figure 14 displays the \( HH_{BC\_PUTPD} \) –the header handler for P+D active messages. Step 03 claims a placeholder for the incoming data, whereas step 04 associates the claimed buffer with the p/message. This allows any part of the MPI library that accesses

\(^1\) Or via a completion handler specification, although this is undesirable, due to context switching.

\(^2\) And are in-order of course.
the p/message to access the buffer as well. Step 06 constitutes an atomic update of the \( C_{\text{PENDING}} \) counter. It is atomic because it is protected by the lock of the PMQ (step 05 and 08). Contrary to the BC_PUTP header handler, this one has to return the address of the placeholder [9]. The send buffer, which P+D delivers, will be offloaded to the user-space in a manner identical to T(1).

```
01: claim an OON placeholder
02: populate OON\((P_R)\) given the \( P_R \)
03: claim a buffer
04: \( OON(P_R).buffer\_address \leftarrow \text{address(buffer)} \)
05: lock the PMQ
06: \( C_{\text{PENDING}}.value \leftarrow C_{\text{PENDING}}.value + 1 \)
07: handle the OON\((P_R)\) \{identically to HHBC_PUTP\}
08: unlock the PMQ
09: return the address\((\text{buffer})\)
```

**Figure 14: HH\( _{\text{BC_PUTPD}} \) with Counters**

The header handler forms a natural barrier for newly appearing P+D transfers. This holds due to the PMQ lock acquisition the BC_PUTPD handlers must do. If the lock is already acquired, the handlers cannot return the address and let LAPI progress with the new transfers. So, all critical sections guarded by the PMQ’s lock, are sections where we can safely tell two things: (i) there are \( C_{\text{PENDING}} \) transfers in an unknown state, and (ii) there is a total of \( C_{\text{PENDING}} \) P+D p/messages distributed in the queues. Such a critical section is employed by the BC_ABSORB routine when it toggles queues. We still cannot use the blocking \text{LAPI}\_Waitcntr synchronisation primitive to wait for \( T_{\text{BC_PUTPD}} \) to reach the value of \( C_{\text{PENDING}} \) inside a critical section.

When we were describing the LAPI threads model and how the execution of header handlers fits in that model (Sections 2.1.4, page 10 and 2.1.5, page 11), we mentioned the semantics of the LAPI dispatcher. The LAPI dispatcher first locks a number of resources associated with the referenced LAPI context, and then it executes the header handler under that scope. LAPI prohibits users from performing a number of LAPI operations from within header handlers, one of which is \text{LAPI}\_Waitcntr. \text{LAPI}\_Waitcntr is blocking, which implies that it does not return control until a number of transfers complete. There are two sources of falling in a deadlock when calling the \text{LAPI}\_Waitcntr:

- \text{LAPI}\_Waitcntr is called from within the header handler.
- The header handler has blocked for some reason.

The latter may occur if we attempt to call \text{LAPI}\_Waitcntr from within a PMQ critical region and at the same time a P+D header handler has blocked waiting to enter its PMQ critical region as well.

To support deadlock-free \text{LAPI}\_Waitcntr and protect BC_ABSORB from toggling queues prior to the completion of data movement, we have developed a new lock acquisition policy. The idea is the following: we wait (\text{LAPI}\_Waitcntr) outside the PMQ critical section and enter in it hoping no new transfers have appeared. In other words, BC_ABSORB favors pending transfers by not swapping queues unless communications become quiet.
Figure 15: P+D Pending Transfers Handled by BC_ABSORB

Figure 15 lays out the said policy. Step 02 fetches the value of $C_{PENDING}$ in variable $done$. The obtained value gives the number of known pending transfers at the time of the fetch. Then we wait for $C_{PENDING}$ value transfers to complete ($\text{LAPI\_Waitcntr}$, step 03). In step 04 we lock the PMQ and thus block progression of P+D handlers. Since 02, it is entirely possible for $C_{PENDING}$ to have changed due to newly appearing transfers. So, once transfers are blocked, we can check whether $done$ is equal to $C_{PENDING}$ value –i.e., whether all transfers have completed (step 05 and 06). If there are still pending transfers, we unblock progression of the communications (step 07) and restart (step 08 and 01). Otherwise, we swap queues (step 09), unblock communications (step 10) and let the handler code to process the p/messages in the inactive end of the queue (Step 11).

So, to summarise the characteristics of P+D:

- P+D uses a blocking asynchronous LAPI active message.
- P/message and sender’s buffer are moved in a single LAPI transaction.
- The p/message becomes available prior to the completion of the data transfer.
- BC_ABSORB tries to block P+D transfers and guarantee no pending P+D transfers.

4.5. Out-of-order P/Messages Detection and Handling

4.5.1. Objectives

This section aims to explain how p/messages are detected to be out-of-order and what is done about this. A major requirement of the point-to-point communications is that sends arrive at the receiver in the exact order they were initiated –and this is regardless if sends are non-blocking or asynchronous. The satisfaction of this requirement guarantees that send operations do not overtake each other [MPIF95]. Let us see a case where, without ordering maintained, MPI messages may overtake:

<table>
<thead>
<tr>
<th>Sender</th>
<th>Receiver</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Send(buf1, 1, MPI_INT, receiver, 999, …);</td>
<td>MPI_Recv(st1, 1, MPI_INT, sender, 999, …);</td>
</tr>
<tr>
<td>MPI_Send(buf2, 1, MPI_INT, receiver, 999, …);</td>
<td>MPI_Recv(st2, 1, MPI_INT, sender, 999, …);</td>
</tr>
</tbody>
</table>
Assume that the sender issues two blocking asynchronous\(^\text{1}\) sends with identical control information, i.e. same communicator, tag, receiver, etc. The receiver issues two matching blocking receive operations, which match both sends. According to the standard, the first receive should match with the first send and the second receive with the second send.

We said that \texttt{BC\_PUTP} and \texttt{BC\_PUTPD} never synchronise with the sender, i.e. multiple p/messages may arrive in any order at the receiver’s end. Without ordering maintained, the first \texttt{MPI\_Recv} may match any of the \texttt{MPI\_Sends}. Re-arranging the \texttt{MPI\_Recvs} does not solve the problem as the sends may still arrive in any order. Using different tags for the \texttt{MPI\_Send} operations is a possible workaround, but this leaves the library non-conforming to the standard MPI behaviour. On the other hand, we would like both \texttt{MPI\_Send} operations to overlap so that more network capacity is utilised (pipelining).

4.5.2. The Required Framework

To handle out-of-order p/messages (OO p/messages), one needs some form of sequencing and some algorithm to restore ordering. The required framework consists of two components: a sequence numbers pool (SNP) and an OO p/messages re-ordering facility. As ordering is a property that must hold just for a pair of endpoints, it is meaningful for the two components to be allocated in a per-pair fashion, i.e. given \(P\) processes, each process should have SNPs for \(P\) targets, and re-ordering facilities for \(P\) origins. The OO handling should be such that, when we introduce a potentially-OO p/message \(P_M\), we would like to achieve the following performance goals:

- The approach is faster than using a synchronous \texttt{BC\_PUTP/BC\_PUTPD}.
- Deduce quickly whether \(P_M\) is out-of-order or not.
- Restore ordering as soon as possible.
- The approach should not add overheads to the PMQ.
- If header handlers are participating, then their latency should be kept low.

4.5.3. The Sequence Numbers Pool

The purpose of the sequence numbers pool (SNP) is to generate sequence numbers. The SNP issues incremental sequence numbers, which are used by the \texttt{BC\_PUTP/BC\_PUTPD} components to tag outgoing p/messages. We decided to use a simple 64-bit integer counter for generating sequence numbers. Incremental integer sequence numbers possess a very lightweight structure, which makes out-of-order exceptions easily detectable. Two p/messages \(P_A\) and \(P_B\), where \(P_A\) precedes \(P_B\), are considered to be in-order with each other if:

\[
\text{SEQ}(P_B) - \text{SEQ}(P_A) = 1 \quad [1]
\]

Although integer sequence numbers are easily manipulated, they are constrained by the natural limits of the underlying integer counter. The SNP will have to reset (wrap-around), when its counter reaches the maximum value. Because such resets are infrequent, we can settle with an expensive approach. Here is the concept: the sender process blocks the generation of sequence numbers until the receiver restores fully.

---

\(^{1}\) Asynchronous in the sense that the sender does not have to synchronize with the receiver in the way \texttt{MPI\_Send} does.
Hence, unless the ordering is achieved at the receiver end, no BC_PUTP/BC_PUTPD may progress. We do this in the form of a handshake between the two ends.

```
Sender : Main Thread
01: next ← SNP(receiver).nextSEQ
02: if next = max SEQ
03: SNP(receiver).status ← not ready
04: transfer: AMWA_REQUEST → receiver
05: busy-wait until SNP(receiver).status = ready
06: return initial SEQ
07: else return next

Sender: AMWA_REPLY header handler
08: SNP(sender) ← ready

Receiver: AMWA_REQUEST header handler
09: busy-wait until ordering is achieved for source(AMWA_REQUEST)
10: completion handler spec ← {AMWA_REQUEST CC, sender}

Receiver: AMWA_REQUEST completion handler
11: transfer: AMWA_REPLY → sender(AMWA_REQUEST)
```

Figure 16: Sequence Numbers Pool Synchronisation

The BC_PUTP/BC_PUTPD queries the SNP for a sequence number. With respect to Figure 16, the SNP generates the sequence number next (step 01). If the pool needs not to reset, then next is returned to the caller routine (step 07). If a reset needs to take place, then we follow a synchronisation procedure entirely crafted in terms of active messages. The SNP maintains a state flag, SNP(receiver).status, which indicates whether the pool is ready to continue. Prior to the synchronisation the pool brings itself offline by setting that flag to not ready (step 03). It then constructs an AM with the following payload:

```
<AMWA_REQUEST, sender id>,-,(-,-,-)
```

That is—this AM forces a non-blocking asynchronous remote invocation of the AMWA_REQUEST header handler routine. When the AMWA_REQUEST header handler gets invoked, it blocks until ordering is achieved (step 09). This is admittedly dangerous. According to [IBM98], header handlers are not guaranteed to execute one at a time. Throughout the development of the HPCxMPI library we could observe header handlers executing concurrently, so we took this behaviour into account. If header handlers where indeed executing one at a time, then this model would deadlock the notification handler thread. Another interesting remark is that the duration of blocking depends on the performance of ordering restoration.

Once ordering has been restored, the AMWA_REQUEST header handler constructs a completion handler specification (step 10). The purpose of the completion handler is to unblock the remote SNP and finalise the synchronisation. This uses an active message with the following payload:

```
<AMWA_REPLY, receiver>,-,(-,-,-)
```

Again, this forces a non-blocking asynchronous remote invocation of the AMWA_REPLY header handler routine at the sender end. Once invoked, it sets the status of the SNP to true and unblocks the SNP (step 08). The reader may wonder why we do not use a synchronous AMWA_REQUEST and block on a completion counter instead of waiting for an
AM to unblock us. We just wanted to reserve execution space for action prior to unblocking of the pool. This has not been investigated more because this handshake occurs rarely. To conclude:

- The SNP is an integer counter, which needs to reset when it becomes saturated.
- Resetting the SNP resets requires an expensive AM-based handshake.
- Resetting the SNP occurs very rarely.

4.5.4. OO P/Messages Detection and Organisation

In the previous paragraph we discussed how the SNP generates sequence numbers and how it resets. Probably, the most important is how ordering is restored. P/Messages should finally end in the PMQ, in the same way if the responsible BC_PUTP/BC_PUTPD were executing synchronously. Assume that the last p/message that was found in order was PM, and had a sequence number equal to X. We consider a stream of p/messages, to be in-order with PM when:

- the p/messages are sorted incrementally according to their sequence number,
- each one is in-order with its predecessor, and
- the one with the smallest sequence number is in-order with PM.

For a stream of unordered p/messages, we do not want to sort it once all p/messages have arrived in. This is because we cannot know in advance how many p/messages are expected. Most importantly, we do not desire to postpone the processing of p/messages, for which some ordering has been achieved. I.e., if part of the stream achieves ordering with PM, then the p/messages that belong to the set should be dispatched for processing by the handler code.

We can maintain a list of sorted p/messages, which may not be necessarily in-order. Whenever a new p/message arrives in the system, we sort-insert it in the pool so that the pool is continuously kept sorted. A list is not enough. Every time we wish to identify p/messages being in-order with the last in-order one, we would have to scan the list forwards to identify the subset. In addition to that, sort-inserting in a continuously growing list would be very expensive.

We should now discuss the structure of an efficient OO p/messages pool. The concept of our approach is to capture sequences of in-order p/messages, and keep them separated from those that are out-of-order. Each potentially-OO p/message PR is encapsulated in an OO node OON(PR) and is introduced to the pool. There are two classes of OONs — single and clusters:

- A single OO node is one that encapsulates a single p/message in it.
- A cluster OO node contains a sequence of single OONs, whose encapsulated p/messages are in-order with each other. Each node maintains two sequence numbers, OON(PR).from and OON(PR).to, to indicate the beginning and end of the sequence.

All OO nodes are doubly linked with their closest neighbours, forming a chain of OO nodes. Because sequences of in-order p/messages are always encapsulated in a cluster OON, the chain only consists of out-of-order p/messages. In other words, sort-inserting a new node in the chain, ignores sequences of in-order p/messages. This has an interesting advantage: the more ordering is achieved, the faster sort-insertion becomes.
Figure 17 attempts to clarify our approach with an example. Assume that the last \textit{p/message} passed to the PMQ had a sequence number of 8. In (a), OO nodes with \textit{p/message} who have sequence numbers 10, 11 and 12 are all grouped into a cluster (displayed with gray background). The cluster (10~12) is doubly linked with the single nodes (23) and (25). All of them are out-of-order with respect to each other. Sort-inserting requires searches in the vertical direction, and as it can be seen, (11) and (12) are ignored. In (b), two OO nodes are introduced, with a sequence numbers equal to 13 and 24. Because (24), (25) and (26) are all in order, they advance into the cluster (25~26). Then, the tail of the cluster (10~12) is in-order with 13, so the cluster absorbs it and advances to (10~13). Now sort-insertions only see two nodes, and not seven. Finally in (c), an OO node with a sequence number equal to 9 is introduced. Because 9 is in-order with the head of the cluster (10~13), the new cluster (9~13) is formed. Moreover to that, because 9 is in-order with the last \textit{p/message} (had a SEQ equal to 8), the cluster (9~13) advances to (8~13), gets detached from the pool and is introduced to the PMQ as a whole. We discuss how the procedure works in the next paragraph.

4.5.5. OO P/Messages Handling

The ordering restoration facility breaks down into the OO \textit{p/messages} detection and accumulation service (OODetector), and interaction between service and PMQ. We have decided to perform this task from within the \texttt{BC\_PUTP}/\texttt{BC\_PUTPD} header handler, thus slightly increasing its latency.

Figure 18 displays how \texttt{BC\_PUTP} and \texttt{BC\_PUTPD} handle a potentially-OO \textit{p/message} \texttt{P}_R. When the LAPI dispatcher invokes the header handler, the handler claims an OO node (step 01), within which the \textit{p/message} will be encapsulated (step 02). It then locks the PMQ and the OODetector instance that is responsible for the sender of the \textit{p/message} (steps 03 and 04). Then, the OO node is sort-inserted in the pool (step 05). Introducing the node to the pool will is always necessary so that ordering is achieved. In other words, ordering gets restored gradually and only when a new \textit{p/message} arrives at the node.

After that, a request is made to detach the head node of the pool (step 06). A detachment occurs whenever the head of the pool (clustered or not) is in-order with the last \textit{p/message} found in-order. So for instance, if our \texttt{BC\_PUTP}\* was synchronous, every incoming \textit{p/message} would lead to the detachment of itself from the pool and the pool
would be left constantly empty. If no detachment can take place, nothing is returned. Otherwise, the detached node is appended to the end of the active queue (step 08). Finally, locks are released (steps 09 and 10). Whenever a detachment occurs, the OODetector caters to update the record it keeps about the last p/message that was found in order. If the detached node was a cluster then its upper sequence \( \text{OON(\ldots).to} \) is used as the sequence of the last in-order p/message.

```
01 claim an OON placeholder
02 populate OON given the \( P_R \) \{forms OON(\( P_R \))\}
03 lock the PMQ
04   lock the OODetector(source(\( P_R \)))
05   sort-insert the OON(\( P_R \)) in the OODetector(source(\( P_R \)))
06   OONDETACHED \leftarrow detach head(OODetector(source(\( P_R \))))
07 if OONDETACHED \neq \text{nil}
08    append OONDETACHED to the end of PMQ’s active FIFO
09 unlock the OODetector(source(\( P_R \)))
10 unlock the PMQ
```

**Figure 18: The HH\textsubscript{BC\textunderscore PUTPD} with OOP/Messages Handling**

There is a reason behind performing the detachment of an in-order OON and its attachment to the PMQ while having both PMQ and OODetector locked. Assume that two p/messages \( P_A \) and \( P_B \) are received, which do form the in-order sequence \( \{P_A, P_B\} \) and \( P_A \) is in-order with the last p/message. If we introduce \( P_B \) first, it will be the introduction of \( P_A \) that will detach both. If \( P_A \) is introduced first and \( P_B \) later, then we have two detachments, i.e. we have two handler occurrences holding a separate node each. Without having the PMQ locked, there is no guarantee that the handler that holds OON(\( P_A \)), will manage to introduce it to the PMQ before the one that holds OON(\( P_B \)). Without the PMQ locked we would break the achieved ordering.

Figure 19 (next page) displays the algorithm employed to introduce an OON to the OODetector. The variable root is the head of the list (see Figure 17, page37), while \( A \) is the OON we introduce. Understanding the flow of this algorithm is not necessary for following the rest of the report. It is placed here as a high-level hint for the developer that desires to experiment with the library. Please take into account the following predicates:

\[
\text{after}(S, T) :\neg \text{SEQ}(S).\text{from} - \text{SEQ}(T).\text{to} > 1 \quad \text{iafter}(S, T) :\neg \text{SEQ}(S).\text{from} - \text{SEQ}(T).\text{to} = 1
\]

### 4.6. Porting BC\_PUTPD and BC\_GETD

The routines that facilitate bulk transfers (BC\_PUT and BC\_GET) are actually wrappers for the shmem\_put and shmem\_get respectively. Their porting has been straightforward. The BC\_GET has been ported by using a blocking LAPI\_Get operation, i.e. by supplying LAPI\_Get with a local counter\(^1\). The LAPI\_Get is known to be slower than the LAPI\_Put as it is has been implemented on top of LAPI\_Put. The authors of [Klep98] describe the LAPI\_Get flow in terms of two components: the Get\_req, initiated by the source, aimed at the target; and the Put\_response initiated by the target due to the received Get\_req. To pull data from remote address \( X \) into the local address \( Y \), the target sends both addresses implicitly back to the source, and the source issues a LAPI\_Put to

\(^1\) Contrary to LAPI\_Put, LAPI\_Get does not use a completion counter.
facilitate the X→Y transfer. Given the confidence built from the stability of LAPI_Get operations on unpublished addresses\(^1\), BC_GETD is configured to accept any remote address passed to it. The BC_PUTPD routine has been implemented as a synchronous LAPI_Put. To conclude:

- BC_GETD is a blocking LAPI_Get.
- BC_PUTPD is a synchronous LAPI_Put.

\[\text{insert}(A,\text{root}):\]
\begin{align*}
01: & \quad \text{if root} = \text{nil} \\
02: & \quad \text{root} \leftarrow A \\
03: & \quad \text{else} \\
04: & \quad \quad C \leftarrow \text{root} \\
05: & \quad \quad \text{if after}(C,A) \\
06: & \quad \quad \quad \text{root} \leftarrow A, \ \text{link}(A,C), \ \text{exit} \\
07: & \quad \quad \text{if iafter}(C,A) \\
08: & \quad \quad \quad \text{root} \leftarrow A, \ \text{link}(A,C.next), \ \text{accumulate}(A,C), \ \text{exit} \\
09: & \quad \quad \text{while} \ \text{current} \neq \text{nil} \\
10: & \quad \quad \quad N \leftarrow C.next, \ P \leftarrow C.previous \\
11: & \quad \quad \quad \text{if after}(A,C) \ \text{AND} \ (N = \text{nil} \ \text{OR} \ (N \neq \text{nil} \ \text{AND} \ \text{after}(N,A))) \\
12: & \quad \quad \quad \quad \text{link}(C,A), \ \text{link}(A.N), \ \text{exit} \\
13: & \quad \quad \quad \text{if after}(A,C) \ \text{AND} \ N \neq \text{nil} \ \text{AND} \ \text{iafter}(N,A) \\
14: & \quad \quad \quad \quad \text{link}(A,N.next), \ \text{link}(C,A), \ \text{accumulate}(A,N), \ \text{exit} \\
15: & \quad \quad \quad \text{if iafter}(A,C) \ \text{AND} \ (N = \text{nil} \ \text{OR} \ (N \neq \text{nil} \ \text{AND} \ \text{NOT} \ \text{iafter}(N,A))) \\
16: & \quad \quad \quad \quad \text{accumulate}(C,A), \ \text{exit} \\
17: & \quad \quad \quad \text{if iafter}(A,C) \ \text{AND} \ N \neq \text{nil} \ \text{AND} \ \text{iafter}(N,A) \\
18: & \quad \quad \quad \quad \text{link}(C,N.next), \ \text{accumulate}(C,A), \ \text{accumulate}(C,N), \ \text{exit} \\
19: & \quad \quad \quad C \leftarrow N \\
\end{align*}

\[\text{Figure 19: The Algorithm for Clustering OOP/Messages}\]

---

\(^1\) Addresses not initialized by LAPI_Address_init.
5. Understanding the Performance

5.1. Experimental Tunings
Once a stable version of the library became available, we began evaluating the library against a number of alternative design ideas (potential tunings) in order to understand performance. The tools used for collecting performance metrics are our own MPI and LAPI benchmarks, the IBM Hpmcount tool and a standard UNIX profiler. We use two acronyms to distinguish between LAPI operational mode: INT – interrupts are enabled, and POL – in polling mode. While most of the tunings are discussed, some are referenced just to make the potential developer aware of their existence.

Probes: In this tuning we explore means for efficiently blocking the MPI user-space code, until a condition is met. A blocking BC_ABSORB (b/BC_ABSORB) is a mechanism, which blocks progress until a certain condition is satisfied. Most times, it will be the outcome of processing some p/message(s) that leads to the satisfaction of the condition. For instance, MPI_Recv absorbs p/messages until a matching send occurs. While b/BC_ABSORB does block the execution of the MPI user-space code, it does not block the processing of p/messages. If the library is executing in INT mode, such busy-waiting works. Contrary to that, if the library is in POL mode, then the receiver has to explicitly check for the arrival of p/messages, i.e. probe (LAPI_Probe). This occurs because in HPCxMPI, it is header handlers that introduce p/messages to the PMQ. So unless they are executed, no progression can take place. Moreover, any sort of blocking behaviour, which assumes progression of communications, should probe. We have named this blocking behaviour spin-probing. The BASIC COMMS layer exports two new macros: _block_while(condition), and BC_ABSORB_until(condition). Both encapsulate probing and are used transparently by the INT and POL versions.

Speculative Couple-Add (SCA): This is an experimental P+D mode, under which the BC_PUTPD handler attempts to find the matching receive and use its referenced buffer instead of allocating a new one. I.e., we are trying to achieve zero copying using an eager protocol. In general, this mode is unsafe as it accesses the threads-unsafe events manipulation layer of the library (point.c, not seen –part of Point-to-Point).

Packed P+D (PPD): Allows single-packet P+D transfers to skip the T_BC_PUTPD counter, and make BC_PUTPD behave as the BC_PUTP header handler. It has the disadvantage of locally buffering the p/message and the send-buffer. Essentially, it attempts to emulate T(1).

Assume-in-order (AINO): This mode forces the BC_PUTPD/BC_PUTP handlers to assume that the p/messages always arrive in order and thus skip the detection for out-of-order exceptions. This allows the handlers to introduce the p/message directly in the PMQ. The purpose of this mode is to demonstrate
the overhead of OO p-messages detection under scenarios where ordering is naturally guaranteed. We will not discuss AINO in the following paragraphs.

**PMQ has large scope (PLS):** The detection of OO p/messages and the ordering restoration process always takes place under the scope of the PMQ lock. This mode removes the redundancy to relax locking acquisition on OO handler structures. We will not discuss PLS in the following paragraphs.

**Use Completion Handler (UCH):** P+D transfers skip $T_{BC\_PUTPD}$ entirely and use a completion handler instead. This is implemented similarly to MPI-LAPI (see 2.2.2, page16).

### 5.2. The Benchmark

We decided to investigate performance issues for the unidirectional ping-pong benchmark. A unidirectional ping-pong is the classic ping-pong, where pings and pongs do not overlap. It requires two processes, where process 0 issues an `MPI_Send` followed by an `MPI_Recv`, and process 1 issues an `MPI_Recv` followed by an `MPI_Send`. We repeat the ping-pong for a number of times and according to the size of the payload. For instance, we have 200,000 runs for payloads less that 8K. We record how much time it takes ($T_{TOTAL}$ in seconds) to complete all of the runs ($R$) for a particular payload and calculate the following metrics:

\[
\text{latency} = \frac{1}{2} \times \left( \frac{T_{TOTAL}}{R} \right) \times 10^6 \quad \text{Msec}
\]

\[
\text{bandwidth} = 10^6 \times 2 \times R \times \text{size(payload)} / T_{TOTAL} \quad \text{Mbytes/sec}
\]

### 5.3. The Basic Versions

We should start by characterising the unoptimised version of HPCxMPI, [2]. This version operates in INT mode and uses busy-waiting to implement the blocking $BC\_ABSORB$, i.e. \texttt{do}\{\texttt{BC\_ABSORB}()\}\texttt{while}(...). For the ping-pong test, the only existent blocking behaviour is due to `MPI_Recv`. `MPI_Recv` invokes post_recv to post its local receive, and then blocks until the receive event is matched with an alien send. Please note that $BC\_ABSORB$ is a macro to `_BC\_ABSORB`. The routine `_BC\_ABSORB` is not invoked unless the PMQ becomes non-empty –a sensible thing to do in order to avoid unnecessary procedural calls. With respect to Figure 20 (next page), version [2] delivers an almost uniform latency (59-62 μsec) for payloads in the range 8~512 bytes. The first jump can be observed during the transition from 512 bytes → 1K. Performance degrades for the IBM library [1] as well, but in a smoother manner. Degradation is expected indeed. The length of a LAPI packet is almost 1K, implying that above 512 bytes we are experiencing multi-packet transfers.
All HPCxMPI versions, have specified T(1) to apply for payloads of sizes up to 24 bytes—thereon, P+D is employed. While both protocols couple p/message and send buffer in a single active message payload, they have a major difference:

- T(1) encapsulates the coupled data in the handler parameters, therefore send buffer and p/message are both placed in the PMQ upon return from the header handler.
- In P+D, the header handler places the p/message in the PMQ and returns the address of a receive buffer to the LAPI dispatcher. At a later time, the LAPI dispatcher copies the send buffer in that address and increments the target counter.

For P+D transfers, _BC_ABSORB has to wait for completion of transfers. Given this principal difference, we consider P+D to be more expensive than T(1). Interestingly, both protocols behave identically for single-packet transfers. When we enter multi-packet sessions, performance is degraded dramatically. More noticeably, [2] is almost three times slower than [3].

Version [3], is the HPCxMPI library operating in polling mode (POL). In POL mode, it is the user that probes (LAPI_Probe) for incoming active messages. Header handlers are always executed from the main thread, and therefore there are no context-switches due to arriving active messages. We can see that not only [3] has half the latency of [2], but it also displays a curve with a shape very close to that for [1]. In POL HPCxMPI we have to find all places where blocking BC_ABSORBs exist, and introduce probing (LAPI_Probe). If we do not probe, no active messages will be discovered and the process will remain blocked forever.
We calculated the latency as half the time it takes for an MPI_Send/MPI_Recv pair to complete. So, inherent to the latency are the costs of:

- Local completion of the blocking LAPI_Amsend the MPI_Send issues, and
- Any delays introduced prior to the processing of the send’s p/message.

Single-packet payloads are probably buffered in the NI, so that the LAPI_Amsend can unblock immediately. Multi-packet transfers (>1K) are bound to the time it takes for the header handler to return the address of a receive buffer to LAPI —for instance, a LAPI_Amsend cannot complete locally if the (receiver’s) header handler never returns. Having said this, [2] is at least 30 µsec slower than [3] for either of the following reasons:

- Excessive context switching due to header handler invocations from the notifications handler thread, or
- The manner at which interrupts are delivered. The LAPI manual specifies that in INT mode, active messages cause an interrupt. It is not clear whether the interrupt is delivered immediately.
- In addition to that, it is not known what happens while the header handler is running —i.e. are packets of the send buffer being buffered in the space of NI, or they do not leave the sender unless the handler returns. In either case, header handlers of high latency do harm performance.

We could go on making more speculations, but only good knowledge about the SP2 switch can justify the seen behaviour.

5.4. Bringing the Dispatcher In-Code

When we described how LAPI reacts to AM packets, we stressed a particular feature: if the interrupt is caused while the LAPI dispatcher is active, the header handler is then executed from within the thread that the dispatcher was called from. So, if we can keep the dispatcher active from within the main thread (in-core), we can avoid context switching to the notifications handler thread. One possible way to do so, it is to use the Probes tuning in INT mode. This tuning suggests that, in addition to busy-waiting by repeatedly absorbing, we probe for active messages as well. This tuning advances the blocking BC_ABSORB to the following:

```c
bool LAPI_Probe();
BC_ABSORB( )
```

We exploit the fact that most LAPI operations (including LAPI_Probe) implicitly invoke the dispatcher. By continuously probing, we therefore increase the likelihood for an active message to hit our node while the dispatcher is running from the main thread and avoid a context-switch.

With respect to Figure 21 (next page), the new version is [2] —HPCxMPI in INT mode using the Probes tuning. The improvement in performance is obvious: for an 8 bytes payload, [2] delivers 41 µsec —less than 10 µsec slower than the library in POL mode [3]. We believe that [1] is slower than [3] due to context-switching as their only difference is reduced context-switches. Finally, we reach the following conclusion:

- Bringing the LAPI dispatcher in-core at the right time (when an AM hits the node) can lead to significant performance improvements.

In practice, this works for any blocking MPI routine. Non-blocking ones can hardly exploit this feature.
Figure 21: Latencies for HPCxMPI (INT)/Probes (INT) and IBM MPI.

Figure 22: HPCxMPI in Various INT Modes.
5.5. HPCxMPI with the UCH Feature

The purpose of the UCH tuning is to relief \_BC\_ABSORB from having to call \texttt{LAPI\_Waitcntr} in order to wait for pending transfers to complete. Moreover it permits \_BC\_ABSORB to progress and just block for particular transfers. We employ a completion handler to do that \textit{but} at the cost of extra context switches.

Figure 22 (previous page) displays four versions of the HPCxMPI library. The new versions are [2] and [4]; both employ the completion handler. We can see that T(1) transfers (payload $\leq 24$ bytes) are unaffected by UCH as UCH only applies to P+D. For single-packet P+D transfers, UCH adds at least 5 µsec to both [2] and [4]. Most remarkable is the behaviour of [4]. [4] \textit{explodes} for multi-packet transfers above 1K, bringing its performance close to [1] and [2].

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>256 bytes</td>
<td>1.32</td>
<td>0.74</td>
<td>1.79</td>
</tr>
<tr>
<td>2K</td>
<td>1.77</td>
<td>0.81</td>
<td>2.54</td>
</tr>
</tbody>
</table>

\textbf{Table 2: Amount of Context Switches (Figure 22)}

Table 2 displays the amount of context switching, which versions [1], [3] and [4] cause for the reception of 256 byte and 2K payloads. We ensured accuracy of the figures by repeating ping-pongs 200,000 times and collecting the average values. With respect to the 2K payloads case (latencies are not listed in the table):

- [4] causes 1.4 times more context switches than [1], while its latency (98 µsec) is 1.08 times that for [1].
- [4] causes 3.1 times more context switches than [3], while its latency is 1.6 times higher that for [3].

Finally, for the 256 bytes payload [4] causes 2.4 more context-switches than [3], while it delivers 1.1 times higher latency than [3] (45 µsec). The figures we collected lead us to two conclusions:

- Context-switching \textit{does degrade} the performance of small payloads delivery (up to 4K).
- This degradation is more evident in multi-packet transfers.

5.6. T(1) Emulation

The tuning PPD (packed P+D) has been developed mostly to demonstrate whether P+D incurs an overhead for single-packet transfers. Contrary to T(1), which encapsulates the send buffer in the handler’s parameters space, P+D cannot make the send buffer \textit{available} at handler invocation time. PPD emulates T(1) by locally merging the p/message and send buffer into a separate buffer, and dispatching it through the parameters space.

Figure 23 (next page) displays the latency achieved by PPD-enabled ([2], [4] and [6]) and plain INT/POL versions ([1], [3] and [5]). It is obvious that [2], [4] and [6] have just worsened things. This demonstrates how much expensive local buffering is. What is worth mentioning though, is that PPD transfers (>16 bytes) are quite degrade smoothly, in contrast to their non-PPD equivalents. The conclusion:

- Allowing T(1) to support bigger payloads (>24 bytes), is a \textbf{bad idea}, due to the enforced local buffering (send buffer $\rightarrow$ p/message envelope). More importantly, we should have better replaced T(1) by P+D.
5.7. Zero-Copying

P+D is an eager protocol as it always buffers the send buffer into the receiver’s space. Under the SCA tuning, the P+D header handler attempts to couple the send operation with a posted receive operation. If a match occurs then P+D can offload the send buffer directly in the user-space, thus achieving a zero-copy.

Alternatively, zero-copying can be achieved by RTA, as it always pulls the send buffer into the user-space. We can see (Figure 24, next page) that from 16K and onwards, versions [2] – [4] behave similarly, while SCA [4] is slightly faster for payloads in the range 64K – 256K. Of course SCA is an experimental tuning, which does not guarantee zero-copying. Contrary to this, RTA always zero-copies buffers (unless non-contiguous). RTA is very expensive due to its handshake, but behaves well above 256K. The cost of buffering becomes overhead for payloads larger than 256K. To conclude:

- We should employ a protocol threshold to switch from P+D to RTA for payloads greater or equal to 256K.

Figure 23: The PPD – T(1) Emulator.
Figure 24: Bandwidth Delivered by HPCxMPI
6. Conclusions

6.1. Summary of our Contributions

As a matter of fact, the original developers of T3D/E library have architected it with a long-lasting perspective about portability. Given that it was developed in 1994, for an infrequent OS and in C, it was surprising indeed to find out that the library was highly portable. Actually, except for non-collective communications, just the BASIC COMMS layer needed porting.

To port the BASIC COMMS layer we completely rebuilt it from scratch and keeping in mind the active-messaging capabilities of LAPI. Because active messages can specify how the receiver deals with them, we looked towards how we could assign previously remotely dealt tasks, to the receiver.

In particular and more noticeably, BC_PUTP absorbed most of our changes and interest, more because it is the exclusive mechanism for sending short messages and control information. In the way BC_PUTP was designed, it only allowed nodes to receive a single BC_PUTP payload at a time (Section 3.2.3, page 22). We turned the BC_PUTP payload into an active-message and attacked this issue from two perspectives:

- **PMQ lock localisation:** we localised the manipulation of the PMQ, by making the lock acquisition/release and PMQ manipulation operations part of the BC_PUTP AM header handler. This permits a node to receive BC_PUTP payloads from more than one distinct node. More importantly, the old BC_PUTP, which required at least 3 SHMEM transactions, now takes a single LAPI transaction instead.

- **Asynchronous BC_PUTP:** we permitted BC_PUTP to be asynchronous so that a node could receive multiple BC_PUTP payloads from a single node. This makes better use of the pipelining capabilities of the network, but leads to unordered delivery of payloads (and thus unordered invocation of handlers). To deal with that we had to write an efficient ordering restoration mechanism, which required some considerable testing.

The ordering restoration scheme features three properties:

- **Per-origin scope:** Restoration overheads have a per-origin scope, implying that we do not have to wait for all p/messages to come in-order.

- **Gradual reordering:** Only arriving p/messages can trigger ordering restoration for a particular origin. Sequences of in-order p/messages are detected immediately.

- **Efficiency:** An efficient algorithm for sort-insertions has been designed to let the ordering restoration scheme to deal with large volumes of p/messages.

Then, we decided to make use of the real power of LAPI active messages: the transfer of arbitrarily sized buffers without prior knowledge of the reception buffer. We crafted a new protocol, P+D, i.e. p/message plus data in a single LAPI transaction. This came with two advantages:

- **Eager transfers:** Bulk transfers can be initiated without handshaking with the receiver as in RTA. The header handler of the P+D AM would then find a buffer to let LAPI deposit the P+D payload in. Because P+D payloads are always buffered at the receiver, P+D transfers data eagerly.

48
• Coupled control information and data: As in T(1), P+D couples control information (p/message) and data in a single active message, and dispatches both to the receiver as a whole.

Finally, we developed a number of experimental modes (Probes, UCH, PPD, SCA, etc.) in order to lower latencies and justify the seen behaviour.

6.2. Limitations and Future Directions

When LAPI executes with interrupts enabled, context switching has a high performance cost. For single-packet payloads, we measured the cost to be at least 10 µsec. From our experimentation with LAPI in interrupts mode, we came to the conclusion that it is possible to reduce context-switching. Typically, an arriving AM causes an interrupt, which executes the dispatcher from within the notification handler thread. If the LAPI dispatcher is executing at the time the interrupt is caused, then the header handler executes from the running thread of the dispatcher. Of course, our conclusions applied to the unidirectional ping-pong, where it was very likely for the dispatcher to catch the interrupt. In reality, that might not be the case, unless the dispatcher is set to execute in fixed intervals (possibly under a SIGALRM).

The scheme we use to wait for pending transfers to complete has some serious disadvantages, which can become evident in scenarios with more than one communicating processes. The handler always waits for the total number of pending transfers to complete, and not just for the one associated with the p/message it currently processes. Furthermore, this model harms progression with the rest of the accumulated p/messages:

• p/messages that do not have any pending transfers associated with them are not processed until all pending transfers complete. This applies to p/messages originating from any process.

• The handler cannot just wait for the completion of those pending transfers associated with the p/messages in the inactive end of the PMQ.

The T3D/E MPI library was designed for a single thread of execution. Concurrency was due to remote access and did not span outside the BASIC COMMS layer. There were cases (SCA) where we required threads-safety of layers outside BASIC COMMS. Without guaranteeing exclusive access to parts of the other layers, we are dragging ourselves in a dangerous situation.

P+D has been tested on MPI_Send operation only, implying that the rest of the send operations still use the old RTA/RAT for bulk transfers (verified). This is mainly because the protocol bootstrap process is hardcoded in each send operation separately. In an upcoming version we would like to allow P+D to be non-blocking and make it available for the rest of the send operations.

Currently, p/messages are dispatched in single-packet payloads, implying that approximately 90% of the packet is left unused. There are two possible ways to make some use of that space. We could pack multiple p/messages in a single LAPI packet, which would give approximately 15 p/messages per packet.

6.3. A Final Word

This effort aimed to make the T3D/E library available on the HPCx platform, using the low-level LAPI. We concluded this document with a number of speculations that need to
be examined thoroughly under a number of benchmarks/tests. Probably the next milestone should focus on collective communications.
7. References


[CRI99a] Cray T3E\textsuperscript{TM} and Cray T3D\textsuperscript{TM} Programming Environment Differences 3.0.1, Cray Research Inc, SR-2199 3.0.1.


8. Appendix

8.1. List of Acronyms

AM Active Message
b/BC_ABSORB Blocking BC_ABSORB
BCOMMS Basic Communications
EA Early-Arrival Buffer
FM Fast Messages
GAM Generic Active Messages
HH Header Handler
HPC High Performance Computing
INT HPCxMPI Operating with Interrupts Enabled
LAPI Low-Level API
LD LAPI Dispatcher
MPI Message Passing Interface
MPL Message Passing Library
OO Out-of-Order Protocol Message
OON Out-of-Order Protocol Message Place-holding Node
P+D Protocol Message and Data
P/Message Protocol Message
POL HPCxMPI Operating in Polling Mode
PMQ Protocol Messages Queue
RAT Request-Acknowledge-Transfer
RDMA Remote Direct Memory Access
RMC Remote Memory Copy
RTA Request-Transfer-Acknowledge
SEQ Sequence Number
SNP Sequence Numbers Pool
T(1) Single Protocol Message Transfer
T(N) Multi Protocol Messages Transfer
UNICOS Unix Cray Operating System