Introducing LAPI and Capturing the Performance of its Point-to-Point Communications on HPCx and the Colony SPswitch2

Christos Kartsaklis
HPCx Software Engineering Team
EPCC
The University of Edinburgh

December 5, 2003

Abstract
LAPI (Low-Level API) is a user-accessible set of single-sided and active-messaging routines, which are backed up by a collection of synchronisation primitives. IBM supplies its SP line of products with LAPI in a similar way Cray did with SHMEM and its T3D/E hardware. We introduce LAPI and its active-messaging capabilities and draw a correspondence between MPI and LAPI’s workings. We then move on to capture the performance of LAPI versus MPI in the department of point-to-point communications. We show where LAPI becomes the choice over MPI; that there do exist commonalities regarding how these libraries utilise the networking layer; and issues that are particular to LAPI only. The target hardware is HPCx, which uses the Colony SPswitch2 as its interconnection fabric.

1 Introduction
Traditionally, HPC vendors supply their hardware with low-level libraries designed to facilitate inter-processor communication – SHMEM (Cray); NX (Intel); XPMEM.NET (SGI) to name a few. Among those is LAPI (Low-Level API), which IBM has traditionally supplied its SP-based line of HPC services with –such as HPCx (a cluster of 40 p690T “Regatta H” nodes). The HPCx system uses SPswitch2 with Colony adapters, which is shortly to be replaced. This document has two targets: (i) to familiarise the programmer with LAPI and expose its capabilities (Section 1); and (ii) to capture the performance of LAPI point-to-point communications over the Colony SPswitch2 (Section 2).
1.1 The IBM Low-Level API (LAPI)

The Low-level API (LAPI) is the lowest level user-accessible\(^1\) communications library supplied by IBM for its series of SP-based HPC services. It offers a set of active messaging (AM) and RDMA\(^2\) operations (RMC\(^3\), remote atomic and synchronisation). It provides the user with tools that are at a low enough level to lower the latency of communications between the processing elements of an HPC service. LAPI is reliable, discourages the user from interacting with the NI\(^4\), 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 (shared memory) and inter-node (US switch) 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.

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 \texttt{LAPI\_Amsend/LAPI\_Amsendv}\(^5\) 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 \texttt{LAPI\_Put/LAPI\_Putv} is a non-blocking asynchronous routine for *pushing* data into a remote memory location. The \texttt{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*.

**Remote Atomic Operations:** The routines \texttt{LAPI\_Rmw} and \texttt{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*.

---

\(^1\) With documentation available to the public.
\(^2\) Remote direct memory access – routines to remotely manipulate data.
\(^3\) Remote Memory Copy – RDMA routines to remotely read/write data.
\(^4\) Network Interface.
\(^5\) LAPI functions whose names are postfixed with the symbol ‘v’ indicate manipulation of non-contiguous buffers.
1.2 LAPI Synchronisation Primitives

LAPI RMC and remote atomic operations are designed to be non-blocking. LAPI_Put and LAPI_Amsend are asynchronous as well as non-blocking. 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. until the operation has completed locally it is considered unsafe to alter the parameters passed to these operations. Processes must also ensure completion of remote writes in their memory space, before that space is accessed. Furthermore 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 the remote completion of all operations initiated by that thread’s process. With local fencing, one can initiate a group of 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 a data movement barrier. Being barriers, these two routines have a very large scope and cover the whole spectrum of initiated operations. This makes fences suitable for controlling groups of operations and not just single ones.

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 dependant on the code that utilises the counter. LAPI offers a set of routines that perform atomic operations on such 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\(^1\) by the LAPI_Put invoking process, whereas the remote process 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 in its place.

When a local counter is specified, LAPI will increment it once the send buffer can be safely re-used. Similarly, if a completion counter has been specified, LAPI will increment it once the transfer has completed. At the remote end, LAPI increments the target counter once the transfer has completed. I.e., the completion and target counters signal respective processes upon completion of the event.

Contrary to the MPI API, which offers native (non)-blocking and (a)-synchronous variations of send routines, LAPI needs the user to construct them. 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

\(^1\) A process owns a counter if it is the one that allocated it.
specifies the completion counter and waits for a single increment on that counter immediately after returning from LAPI_Put. LAPI_Get is a non-blocking routine and uses two counters, with most important being the origin counter. A single increment on this counter indicates that the routine has completed locally –i.e. the remote buffer has been successfully pulled in. To summarise blocking-ness and synchronicity in terms of LAPI and MPI:

- The concept of a data push operation being blocking applies to both MPI and LAPI –unless the blocking operation has completed locally, it is considered dangerous to alter the specified buffer.
- A synchronous MPI send (MPI_Send, MPI_Isend) function only returns when it has synchronised with the target’s matching receive. A synchronous LAPI_Put|Amsend is one, where we synchronise its completion at the remote end with the return from its invocation.

### 1.3 LAPI 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. In a nutshell, LAPI_Amsend signals the target process about a transfer, and assigns the responsibility of buffer acceptance to that process.

A LAPI active message (AM) 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. This execution consists of invoking the handler routine using the specified parameters. The header handler is required 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 (944 bytes), its purpose is not to substitute the send buffer.

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. 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

---

1 Unless the send buffer is empty (NULL). If empty, then the LAPI_Amsend is more like a remote procedure call (RPC).
2 A placeholder is the target memory space for the incoming data.
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 useful as a counter keeps accumulating completion of communication events in a blind manner—it knows that some events completed, but not exactly which ones.

We have 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 in the next session.

### 1.4 The LAPI Threads Model

Figure 1 (p. 5) describes the threads model LAPI uses for its active messages and how AMs are delivered and processed. It 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. 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.

![Figure 1: The LAPI Threads Model [IBM98]. Note that in polling mode (POL) the notification handler thread remains in the wait state regardless of the arrival of active messages – i.e. LAPI allocates it in case we need it in the future (e.g. if we decide to turn interrupts on).](image)

The LAPI dispatcher is the entity that, in addition to other tasks\(^2\), looks for arriving packets that denote initiation of an AM delivery. Its purpose is to extract them from the NI, execute their contents allowing them and their effects to progress. If such packets are found, the dispatcher extracts their header handler specification and executes it. There are two ways to have the header handler specification executed. LAPI is designed to execute the dispatcher in almost every LAPI routine, regardless the operating mode—with interrupts enabled or 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 ([1], Figure 1) it executes

---

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

\(^2\) Such as the disassembly of the message into multiple packets.
the LAPI dispatcher, which proceeds as previously discussed. When operating in polling-mode, an arriving packet causes no such interrupts at all. The user has to manually *probe* for active messages (*LAPI_Probe*). Probing for messages leads to the dispatcher looking for incoming header handler specifications. Given that the dispatcher executes the header handlers, we can say that the dispatcher *progresses* the active messages.

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.

The completion handler specifications are always executed under the completion handler thread, while the header handler specifications under the notification handler thread or the main thread. Context-switches can be reduced if at the time the interrupt is caused, the dispatcher *happens* to be active. The lack of the interrupts facility in polling mode implies that we always have to *probe*, therefore context switching due to the execution of header handlers *does not occur*, as the dispatcher is always operating from the main thread.

2 The Performance of LAPI Point-to-Point Communications

2.1 Introduction

Before we begin evaluating the performance of point-to-point communications, we should highlight those LAPI data movement features lacking from MPI. Note that both the send-receive *and* the single-sided models can form patterns of point-to-point communications. Because a comparison between LAPI and the MPI-2 single-sided communications has been already covered in [Jack03], we will focus our comparison against the send-receive set of MPI routines. With respect to the interface level, these are the features where we believe LAPI is advantageous to MPI:

- Traceable single-sided communications via the utilisation of target counters.
- Synchronisation with the completion of remote events via the utilisation of completion counters.
- Callback (handler invocation) functionality upon initiation or completion of a data transfer at the remote end.
- Local and global data movement barriers (local and global fences).

The reader should take into account some overall facts that govern the content of the next paragraphs: by default, all benchmarks refer to a 2x8LPAR arrangement, where each process is sitting in a different LPAR —i.e., communications are routed through the user-space (US) switch; unless specified, LAPI figures refer to LAPI operating with
interrupts-enabled (INT) rather than in polling mode (POL); the terms dual-plane (DP) / single-plane (SP) imply that the load-leveler script uses csss/css0, and unless indicated so, both planes are available; timings are obtained using a timer of microsecond accuracy (gettimeofday); the compiler used is mpcc_r as it comes in its original configuration; and LAPI error-handling/reporting is disabled.

A LAPI packet is 968 bytes long. A single-packet transfer refers to the transfer of a ≤944 bytes payload. The missing 24 bytes are reserved by LAPI for internal information –such as references to LAPI handle, header handler address, counters, etc. Therefore a full X-packets transfer should consume 944+ (X-1)x968 bytes.

In the next paragraphs we cover the performance of uni- and bi-directional transfers (latency/bandwidth) for both LAPI and MPI_Send; common message ranges that affect both libraries; remarks about context-switching and techniques to avoid them; various configurations for single-packet transfers; demonstration of computation / communication overlapping capabilities for data transfers of 1Mbyte; synchronous LAPI transfers versus MPI_Ssend and the behaviour of manual LAPI_Get; multi-messaging and multi-processor point-to-point communications.

2.2 Unidirectional Data Transfers - Latency

A unidirectional ping-pong is a non-overlapping exchange of messages between two processes –say 0 and 1. Process/PE 0 sends a message to process 1, and process 1 collects it and sends it back to process 0. Figure 2 shows the flow of data movement under LAPI: PE 0 pushes data to PE 1 and then waits for PE 1’s push to complete. When PE 1 pushes data, it specifies PE 0’s target counter (T0) as the target counter in the LAPI_Put|Amsend. The latency is reported as:

\[
\text{latency}(\text{payload}) = \frac{\text{best}(T1 - T0)}{2} \quad (\mu \text{sec})
\]

Figure 2: Schematic for a LAPI Unidirectional Ping-Pong. Gray areas represent the time it takes for the routine (LAPI_Waitcnt here) to return from its invocation.

Note that we will focus on LAPI_Put primarily, and that we have set the eager limit\(^1\) to 65536 for MPI (maximum). The LAPI_Put start-up cost (latency for a 0-byte payload) is 22µsec and 22.4µsec for single-plane (SP, [1]) and dual-plane (DP, [2]) mode respectively, which compares to 2.4µsec in shared-memory mode. SP/DP single-packet transfers perform comparatively to MPI_Send. With respect to Figure 3 (p. 9), it can be clearly noticed that up to ~1K, increments in latency are shaped uniformly for [1]-[4].

\(^1\)MP_EAGER_LIMIT
This is an indication of both libraries utilising the networking layer similarly for single packet transfers.

The availability of two planes is a HAL\textsuperscript{1} feature previously seen to be exploitable by IBM MPI [HeBB03]. For up to 8-packet payloads (~8K\textsuperscript{2}), LAPI and MPI appear not to make use of both of planes. Single-packet DP LAPI transfers are slightly slower (~3μsec) than their SP equivalents. Note that this behaviour is not observed under MPI, where [3] and [4] match perfectly. Also for LAPI, 4K messages behave differently (Figure 3), as no run has been recorded showing DP 4K puts being faster than SP ones.

In fact, 4K+ is where MPI begins to significantly outperform LAPI. Characteristically, LAPI delivers 8K messages ~20μsec slower than MPI regardless the mode. For 9-packet payloads and onwards, we can observe the gains obtained from utilising both planes. In DP mode, it takes 89.5μsec and 98.5 μsec for LAPI to deliver messages of 8K and 16K respectably as opposed to 82/130μsec for SP mode—a gain of around 30μsec. Optimally, one would expect 8K and 16K messages to take the same time. It is evident that DP transfers have a significant overhead (9μsec for LAPI and 15μsec for MPI). To summarise our impressions about the latencies achieved:

- LAPI is aware of both planes and exploits them similarly to MPI.
- The availability of two planes triggers some extra overhead in LAPI, which affects the unidirectional transfer of ≤8-packet messages.
- The performance of single-packet transfers is shaped identically for both LAPI and MPI, regardless the plane availability.
- LAPI unidirectional transfers up to 16K are slower than MPI, regardless the plane availability.

\textsuperscript{1} Hardware Access Layer
\textsuperscript{2} Actually, ≤ 7720 bytes
2.3 Unidirectional Data Transfers – Common Degradation

In Figure 3 (p. 9) we can clearly notice a *spike*, which occurs at 128 bytes. According to Figure 4 (p. 10) though, there is a range of message sizes (72-200 bytes) where the latency increments sharply, reaching ~33µsec. The latency appears to increment by 1µsec approximately every 32 bytes, which appears as 32byte-wide *stages*. These stages can be easily spotted in the less-noisy LAPI. After 200 bytes, latencies re-stabilise back to that for 64bytes, i.e. 26µsec. The same holds for MPI where latencies increase up to 28µsec and then drop to 22µsec.

The figure displays start-up costs for both communication libraries. Starting at 0bytes does not imply that no data is exchanged between the communicating parties, as a message consists of both a header and user-supplied information (typically a buffer). Admittedly, LAPI latencies are almost a blueprint of MPI latencies, but are shifted upwards by a few microseconds. One could assume that LAPI is slower than MPI just because of bigger internal LAPI header information associated with data transfers. If this did hold, then the LAPI diagram should have been shifted rightwards and not just upwards. The fact that both lines are almost identical suggests that the culprit of performance degradation lies in layers common to both libraries –presumably HAL. The ~2µsec upwards shift can be attributed to software overheads (LAPI) and/or extra (implicit) bi-directional traffic. We can reach the following conclusion:
• Both LAPI and MPI appear to accompany their data transfers with the same volume of internal header information.
• The programmer should typically avoid payloads in the range of 72..200 bytes.

Figure 4: Unidirectional ping-pong; latency for single-packet transfers with a payload in the range of 0..256bytes; measurements have 4 byte precision.

2.4 Unidirectional And Bidirectional Bandwidth

In the bi-directional ping-pong (ping-ping), the pong operation is the same as the ping. We have set the ping to consist of the \{MPI_Isend(R), MPI_Irecv(R), MPI_Waitall\} fragment for MPI; where \(R\) is the id of the remote PE –the one pinged. For LAPI, the ping contains the \{LAPI_Put(R), MPI_Wait(T_L)\}; \(T_L\) is the local target counter, i.e. \(T_0\) for PE 0 and \(T_1\) for PE 1. Both versions are best illustrated in Figure 5.

The following table displays how we calculate bandwidth for the two data transfer modes. Note that \(ToV\) stands for the total volume (Mbytes) launched into the network by each side, while \(ToT\) is the total duration (sec) of the overall ping-ping session.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Formula</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>unidirectional ping-pong</td>
<td>(10^6 \times \text{payload} / \text{latency(payload)})</td>
<td>Mbytes/sec</td>
</tr>
<tr>
<td>bidirectional ping-pong</td>
<td>(2 \times \text{ToV} / \text{ToT})</td>
<td>Mbytes/sec</td>
</tr>
</tbody>
</table>
According to Figure 6, both MPI ([3]) and LAPI ([1]) achieve 160Mbytes/sec for 1Mbyte bulk SP transfers. Up to 64K, MPI SP transfers have been observed to be occasionally at most 20Mbytes/sec faster than their LAPI equivalents. Ideally, with two planes one would expect to double bandwidth. In DP mode, MPI ([4]) reaches 300Mbyte/sec –just 6.25% less than the expectation (2x160Mbytes/sec). This does not appear to be the case for LAPI ([2]) though, as it only delivers 243.5Mbytes/sec. That is 24% less than the optimal expectation (320Mbytes/sec), and 20% less than what MPI achieves. Note, though, that for 128K messages, LAPI can achieve ~255Mbytes. LAPI matches MPI at the point where MPI transfers switch to the rendezvous protocol\(^1\) (64K) and regardless the number of planes (more about this later).

In Figure 7 (p. 12) we display the bandwidth delivered by the bi-directional pattern of buffers exchange. This reveals some interesting results. We can see that LAPI achieves a ~300Mbytes/sec peak in DP mode ([2]), while 180Mbytes/sec in SP mode ([1]). This is a 55 and 20Mbytes/sec of speed-up compared to the relative unidirectional transfers. Similar speedups apply to MPI as well. MPI reached 340Mbytes/sec in DP mode ([4]) and 207Mbytes/sec in SP mode ([3]). This is a 20 and 47 Mbytes/sec of speedups. So for bi-directional transfers, LAPI has ~12% less bandwidth than MPI regardless the plane availability (DP or SP).

---

\(^1\) I.e. when MPI\_Send becomes synchronous.
Figure 6: Unidirectional Bandwidth; LAPI_Put versus MPI_Send; single and dual plane versions.

Figure 7: Bidirectional Bandwidth; LAPI_Put versus LAPI_Send; single and dual plane versions.
In Figure 6 we captured the delivered bandwidth for (common) message sizes, which are a power of two. Widely separated payloads cannot interpret the actual performance of course. For instance, MPI transfers take place eagerly for payloads \( \leq 64K \). [4] (MPI, 2 planes) displays constant performance from 64K..128K, while [2] (LAPI) appears to be worse than [4], in particular above 64K. We have decided to capture the unidirectional bandwidth for payloads that are a perfect multiple of packets. In particular, we examine multiples in the range 0..136, which correspond to message sizes in the range 0bytes..128K.

Figure 8 (p. 14) shows that between 64K and 128K, LAPI achieves better bandwidth than MPI, whose latencies should have been increased by \( \sim 40\mu\text{sec} \) at that point. This is because the rendezvous handshake, which requires two non-overlapping messages –i.e., 2x20\mu\text{sec}, becomes active above 64K. The results show that this is indeed true. The delivery of \( \sim 63K \) (67 packets) costs 243\mu\text{sec}, while 283\mu\text{sec} for \( \sim 64K \) (68 packets). It then takes more than 128K for MPI to hide the extra 40\mu\text{sec}. Throughout 64K..128K, LAPI can be at most 30Mbytes faster than MPI. On the other hand, LAPI appears to be very noisy throughout 16K..64K. It appears that just one or two packets are enough to make a difference of 10..30Mbytes. Please note that these spikes and knees are reproducible for the particular number of packets, with only difference the peak values they reach. Therefore:

- Under unidirectional patterns of communication, MPI outperforms LAPI for DP transfers outside the 64K..128K range, and for \( \leq 32K \) SP transfers.
- Payloads, which consume more than 16 packets and are an exact multiple of 8-packets, e.g. 40 packets, behave badly.
- Under bi-directional patterns of communication, LAPI speedups noticeably, regardless the plane availability.
- Under bi-directional patterns of communication, MPI is 12% faster than LAPI regardless the availability of planes.
Figure 8: Unidirectional bandwidth. The payload is an exact multiple of packets, where packet multiples are in the range 0..136. There is single packet precision and two planes available.

2.5 Active-Messages and Context-Switching

The standard point-to-point benchmarks developed for testing communication libraries have the advantage of being built for a known pattern of communications. In a ping-pong, for instance, each node expects to receive. In LAPI, whenever one is about to ping us, we block using LAPI_Waitcntr to wait for the ping to complete; then we ping-back.

Because there is a high degree of commonality between LAPI_Put and LAPI_Amsend, we can easily repeat the ping-pong benchmark for the later. The LAPI_Amsend-based ping-pong benchmark displays the exact same behaviour with the LAPI_Put-based one, regardless the costly expectations raised from the LAPI threads model (see Sections 1.3 and 1.4, p. 5). One would expect LAPI_Amsend to be slower because of switching contexts to execute the header handler specification. Such context-switches occur minimally, as the LAPI_Waitcntr routine maintains the dispatcher under the scope of the main thread – i.e. no need to switch to the notification handler thread.

In practice though, it is possible that the target may not wish to block for the completion of an active message. This would be the case of communication/computation overlapping arrangements, or active messages with the completion of which the target
does not wish to synchronise. The later applies to the case where completion is traced through a completion handler rather than a counter.

To assess the impact of excessive context-switching, one could engineer an equivalent to LAPI_Waitcntr without the capability of bringing the dispatcher in-core. Such an alternative is the busy-waiting approach—we busy-wait until the counter reaches the desired value. Instead of querying the counter using the threads-safe LAPI_Getcntr, we obtain its value by reading its structure directly. So, an alternative LAPI_Waitcntr looks this way:

\[
\text{busy-wait until } \text{counter} \geq \text{value to reach}; \text{LAPI_Waitcntr(...)}
\]

The LAPI_Waitcntr call does not alter the semantics we are trying to achieve, but is necessary to ensure threads-safe modifications\(^1\) to the counter. We can do the following in the absence of a counter are while in need for minimizing context-switching:

\[
\text{while (counter}<\text{value to reach}) \text{LAPI_Probe(...); LAPI_Waitcntr(...)}
\]

In other words, we spin around the counter and probe for incoming in order to keep the dispatcher active in the main thread. As we will show, such recommendations are occasionally helpful and only apply to known and well-understood patterns of communications. In the next paragraph we discuss the cost of such waiting alternatives in an AM-based version of the ping-pong code.

Figure 9 (p. 16) displays the latency for the non-overlapping delivery of messages in the range of 8bytes-16K. The latency is obtained identically to that for LAPI_Put. Version [1] uses the busy-waiting approach, therefore increasing the occurrence of context switches. We can see that [1] is about 15\(\mu\)sec slower than [3] for payloads up to 2K. From thereon, [1] and [3] behave similarly. Version [2] replaces LAPI_Waitcntr using the spin-probing approach. We can clearly observe that for single packet transfers (<1K) [2] is just 5\(\mu\)sec slower than [3]. Above 1K performance degraded –15\(\mu\)sec slower than [1] and [3] at 8K.

- Attempting to thread-position the dispatcher can make a difference for particular patterns of communication (e.g. ping-pong) and payloads (e.g. single-packet).
- Such attempts are of interest when active-messages are not accompanied with references to a counter.

---

\(^1\) Assume that we busy-wait until the counter reaches the desired value, say X. It is possible that the counter gets updated to Y after we exit the busy-wait. To match the semantics of LAPI_Waitcntr, we have to set the counter to Y-X—i.e. a LAPI_Getcntr (to fetch Y) and a LAPI_Setcntr (to set to X-Y). But we cannot guarantee that Y has not changed between the LAPI_Getcntr and the LAPI_Setcntr calls. We would then need a fetch-and-add routine, to add –X to the most current value of the counter. Although a fetch-and-set is available in AIX, the LAPI_Waitcntr knows exactly how to update a counter.
2.6 Overlapping Communication with Computation

A common practice for accelerating performance breaks down to exploiting the capability of the communications library to overlap communication(s) with computation (workload). Assume that the communication event costs $T_{\text{COMM}}$ and the workload $T_{\text{LOAD}}$. It is known that if $T_{\text{COMM}}=T_{\text{LOAD}}$, a fully overlapping code is at best two times faster than the non-overlapping one.

One needs to guarantee that $T_{\text{LOAD}}$ remains reasonably stable while communications are taking place. In our experiment we have sketched a simple workload, with $T_{\text{LOAD}} \approx 450\mu\text{sec}$. To ensure stability, we calculate $T_{\text{LOAD}}$ prior to benchmarking and force the benchmarking code to monitor $T_{\text{LOAD}}$ and report differentiations. The benchmarking code is a bi-directional ping-pong (ping-ping), which forces global synchronisation after every ping-ping pair. The message size is 1Mbyte ($T_{\text{COMM}} \approx 6,800\mu\text{sec}$), which remains fixed while the workload increases progressively ($T_{\text{COMP}} \approx \text{multiples of } 450\mu\text{sec}$). Please note that $T_{\text{COMM}}$ is the average cost of a ping-ping pair, i.e. two 1Mbyte crossing LAPI_Put calls.

Our workload routine performs the $a[i] \ast=b[i]$ computation for all the elements of two ~370Kbyte arrays $a$ and $b$. We repeat the workload routine, in order to achieve multiples of $450\mu\text{sec}$. Our monitoring indicates that $T_{\text{LOAD}}$ increases slightly.

Figure 9: The cost of context-switching. Displayed versions have replaced LAPI_Waitcntr with an alternative, but semantically equal, waiting policy. Figures have been obtained from the unidirectional ping-pong and for a dual plane arrangement.
due to communication events (up to 150μsec for \( T_{\text{WLOAD}} \approx 9\text{msec} \))—a reminder about possible implications, when \( T_{\text{WLOAD}} \approx T_{\text{COMP}} \). The 150μsec penalty is particular to our workload and by no means should be viewed as a general figure.

A typical ping-pong consists of each side invoking a number of \((\text{LAPI}_\text{Put}, \text{LAPI}_\text{Waitcntr})\) fragments. One could simply inject the workload into this fragment to form \((\text{MPI}_\text{Put}, \text{work}, \text{MPI}_\text{Waitcntr})\) —a ping-pong ready to reflect overlapping capabilities. To ensure that two such fragments cross each other, we have to synchronise both upon their return from the \text{LAPI}_\text{Waitcntr} calls. MPI provides the programmer with a synchronisation barrier (\text{MPI}_\text{Barrier}). LAPI offers its global fencing routine (g/fence), which is an operations and data movement barrier. We reform the fragment from \((\text{MPI}_\text{Put}, \text{work}, \text{MPI}_\text{Waitcntr})\) to \((\text{LAPI}_\text{Gfence}, \text{LAPI}_\text{Put}, \text{work}, \text{LAPI}_\text{Waitcntr}, \text{LAPI}_\text{Gfence})\). Global fencing is known to be an expensive operation (~20μsec for a pair of processes), but negligible for a large \( T_{\text{COMM}} \).

Table 1 displays the four versions we sketched out for our experiment. Versions [1] and [2] do not use any form of a barrier at all and therefore are prone to run ahead and partially overlap incoming/outgoing data with the workload.

<table>
<thead>
<tr>
<th>Version and Description of the Ping</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1] ((\text{MPI}<em>\text{Put}, \text{work}, \text{MPI}</em>\text{Waitcntr}))</td>
<td>partially (non-)overlapping</td>
</tr>
<tr>
<td>[2] ((\text{MPI}<em>\text{Put}, \text{MPI}</em>\text{Waitcntr}, \text{work}))</td>
<td>partially (non-)overlapping</td>
</tr>
<tr>
<td>[3] ((\text{LAPI}<em>\text{Gfence}, \text{LAPI}</em>\text{Put}, \text{work}, \text{LAPI}<em>\text{Waitcntr}, \text{LAPI}</em>\text{Gfence}))</td>
<td>overlaps events</td>
</tr>
<tr>
<td>[4] ((\text{LAPI}<em>\text{Gfence}, \text{LAPI}</em>\text{Put}, \text{LAPI}<em>\text{Waitcntr}, \text{LAPI}</em>\text{Gfence}))</td>
<td>enforces separation of events</td>
</tr>
<tr>
<td>[5] ((\text{MPI}<em>\text{Barrier}, \text{MPI}</em>\text{Isend}, \text{MPI}<em>\text{Irecv}, \text{work}, \text{MPI}</em>\text{Waitall}, \text{MPI}_\text{Barrier}))</td>
<td>overlaps events</td>
</tr>
</tbody>
</table>

Table 1: Versions of (Non-)Overlapping Ping-Ping Codes

Figure 10 displays a schematic of the flow of computation and communication(s) developed by versions [3] (left) and [4] (right).

Figure 10: G/fenced overlapping (left) and non-overlapping (right) versions of the bidirectional ping-pong.
Figure 11: Demonstration of overlapping the bi-directional transfer of 1Mbyte with varying computation (workload). OV stands for overlapping, while NOV for the non-overlapping version of the particular version.

Version [3] enforces overlapping by using 2 global fences. We can see that for $T_{WLOAD}$ up to 6ms (where $T_{COMM} > T_{WLOAD}$), [3] achieves full overlapping. Once $T_{WLOAD} > T_{COMM}$, the ping-pong cost is increased by $T_{WLOAD} - T_{COMM}$. Version [4] uses three $g/fences$ to separate the ping-pong session from the following workload — i.e. forced avoidance of overlapping. The cost of ping-pong for [4] is $T_{COMM} + T_{WLOAD}$. Version [5] is the MPI equivalent of [4], where barriers replace global fences. On the other hand, it is known that MPI overlaps when transfers occur *eagerly* (i.e. for messages ≤64K). So we expect [5] to be almost parallel to [4] and shifted rightwards by $T_{COMM}(MPI) - T_{COMM}(LAPI)$ msec. Versions [1] and [2] were designed without securing their desired policy (overlapping or non-overlapping), leading themselves to experience partial overlaps. To conclude:

- Codes designed (not) to overlap, may overlap partially due to the likelihood of the two ends getting out-of-sync with each other.
- LAPI is capable of overlapping fully, when computation and communications are stressed to execute overlapping-ly.
- The programmer should always evaluate how communications affect computation.
2.7 Synchronous Data Transfers

We have already described (Section 1.2, p. 3) how one uses counters to advance LAPI data pusher operations to synchronous. In this section we are focusing on how synchronous LAPI operations compare to MPI_Ssend, characteristic results about context-switching and synchronous groups of routines.

![Figure 12: Synchronous Transfers in MPI and LAPI.](image)

By definition, synchronous LAPI and MPI transfers are not comparable as they differ semantically. In MPI, the return from the procedural invocation of an MPI_Issend is synchronised with the remotely matching receive; in LAPI, we force synchronisation with the completion of the transfer at the remote end. What both libraries have in common though, is the existence of at least two network transactions: data transfer plus response (ack for MPI and c/update for LAPI). In Figure 12 we show three different patterns of communications where synchronous routines are employed. Note that we indicate implicit messages (ack and c/update) by arrows. Here are the latencies for the versions appearing in Figure 12 (and later in Figure 15): for the “Synchronous MPI Ping-Pong” ([4] in Figure 15) latency is equal to \( T_2 - T_0 / 2 \); for the “Synchronous MPI Pings” ([3] in Figure 15) latency is equal to \( T_1 - T_0 \); and for “Synchronous LAPI Pings” ([1] and [2] in Figure 15) the latency is \( T_1 - T_0 \) (gray area indicates overlapping).

With respect to Figure 13, a synchronous LAPI_Put (routine [1]), i.e. a put immediately followed by a wait for its remote completion, has a start-up cost of 40µsec – an additional ~20µsec. 20µsec is what it takes for the receiver node to figure out the completion of the transfer and backfire an update for the sender’s completion counter. This figure is not surprising at all, given that a 0byte synchronous put resembles a two process roundtrip and should be twice the latency of the Colony switch, i.e. ~40µsec at best. LAPI_Amsend (routine [3]) delivers performance identical to [1], which enhances our belief that LAPI_Put is built on top of active-messaging as well.

---

1 This breaks down to how synchronous sends are implemented. If for instance a 2-messages handshake precedes the data transfer then we have three operations.
Then version [3] of MPI_Ssend delivers the same performance with [1] and [2] for single-packet transfers. Routines [1] and [2] are faster than [3] for payloads up to 4K. For 4K+ messages we experience known behaviour (see Section 2.2, p. 7) – latencies increase sharply, thus rendering MPI_Ssend faster. Again, we observe that, although LAPI handles unidirectional dual-plane transfers inefficiently, in the transition from 8K to 16K, LAPI’s extra overhead (10μsec) is smaller compared to that incurred by MPI (15μsec).

![Figure 13: Synchronous LAPI (LAPI_Put) and MPI (MPI_Send) transfers. Note that [3] displays partial overlapping, while [4] does not. Both [1] and [2] are non-overlapping and synchronous.](image)

### 2.8 Synchronous LAPI_Put versus Blocking LAPI_Get

The LAPI_Get routine ([2]) can only be blocking, but due to its nature (reverse of LAPI_Put) it is presented here as well. Routine [2] is expected to be at best as fast as the synchronous LAPI_Put ([1]) and LAPI_Amsend ([3]). This should hold because all three of them use (at least) two network transactions. In [3] for instance, the sender delivers the data, and the receiver signals him about the completion of the transfer. In [2], the getter requests data and the remote end sends them. So, [1]-[3] should take at least 40μsec. The blocking LAPI_Get has a start-up cost of 44μsec. Note that, the blocking LAPI_Get compares only to the synchronous LAPI_Put and not to the blocking asynchronous LAPI_Put, which is much faster (22.4μsec).
We then decided to implement our own LAPI_Get routine—namely manual LAPI_Get. The manual get ([3], in Figure 15, p. 23) is implemented as follows: when we request a get, an active-message (AM_REQGET) is constructed and dispatched to the remote node. Its header handler specification contains the handler address and all of the parameters we pass to the native LAPI_Get (source/target address, number of bytes, counters, etc.). A LAPI_Get can accept two counters at most, with most important being the origin counter. An increment on it indicates local completion of the get operation. The AM_REQGET’s handler constructs a completion handler specification and populates it with the parameters destined for the header handler. At a later point, the completion handler specification is executed. It uses the LAPI_Get parameters and LAPI_Puts the requested data to the manual get invoker. The LAPI_Get’s origin counter becomes the LAPI_Put’s target counter. Routine [4] is a variation of [3], which makes this LAPI_Put synchronous. Hence both ends achieve mutual synchronisation with the completion of the manual get.

Figure 14 illustrates the two versions of the manual LAPI_Get (appearing as m/LAPI_Get and sm/LAPI_Get). We pick up the smallest (T₁ - T₀) to stand for the latency of the operations. Note that the picture showing the flow of sm/LAPI_Get contains two invocations of sm/LAPI_Get. Please focus on the second one, which will be discussed thoroughly in the next paragraphs.

Routine [3] (manual LAPI_Get) costs ~53 μsec, i.e. ~15 μsec slower than [1] and [2] for single-packet transfers. Of course, [3] experiences more context-switching compared to [2] as a result of feeding LAPI with completion handler specifications. In addition to that, the latency is increased by the time it takes to transit from the completion of the AM transfer to the execution of the completion handler specification. Interestingly enough, 15 μsec is the cost of the non-overlapping version of LAPI_Amsend designed to context-switch to the notifications handler thread in order to execute the header handler specifications (see Figure 9, p. 16; latency of [1] – latency of [3] = 15 μsec). I.e. two different figures suggest that context-switching costs ~15 μsec.

Routine [4] (synchronous manual get) is expected to be slower than [3], as it forces the remote end to synchronise with the completion of the manual get. The question to be answered is how much slower should it be, given the extra message. It appears that [4] costs 67.5 μsec, i.e. ~15 additional μsec to [3]. There can be two explanations to that:

- The implicit message which updates 1’s completion counter (used in the LAPI_Put), and which is sent by 0 causes an overhead of ~15 μsec. It appears that by specifying a completion counter in the LAPI_Put without waiting on it though, delivers the performance seen for [3].
- The completion handler specifications are executed serially (one at a time). The completion handler invokes the LAPI_Put and then waits for its completion. Assuming that there is a blocked completion handler, none of the queued completion handlers will be executed unless the blocked one unblocks. Process 0 does not know about the state of the completion handler and it thus keeps firing an sm/LAPI_Get once the previous one has completed. Performance remains the same with [3] as long as at the time an AM_REQGET arrives, the previous AM_REQGET’s completion handler has been already unblocked. If that is not the
case, then sm/LAPI_Get will be delayed for an additional \((T_3 - T_2) - (T_1 - T_0)\) \(\mu\)sec (see Figure 14, right picture).

![Figure 14: The Manual and Synchronous Manual LAPI_Get. Note that the later forces the completion handler to synchronise with its LAPI_Put. This blocking behaviour prohibits the next sm/LAPI_Get from progressing unless the LAPI_Put completes.](image)

Given that it takes 42\(\mu\)sec for a \((\text{LAPI}_\text{Put}, \text{LAPI}_\text{Waitcntr})\) pair, we investigate the cost of a synchronous group of \text{LAPI}_\text{Puts} –denoted by \((\{\text{LAPI}_\text{Put}_1, \ldots, \text{LAPI}_\text{Put}_n\}, \text{LAPI}_\text{Waitcntr})\), for a fixed payload of 64 bytes. Then we noticed that for each \text{LAPI}_\text{Put}_i\ we add to the group, latency increases by 6-10\(\mu\)sec indeed. Note though, that synchronisation at the group level displays three levels of overlapping: overlapping of data transfers; overlapping of data transfers and procedural costs of successive members of the group, i.e. previous initiated transfers with next put; and overlapping at the counter updates communication level. With this in mind we can justify why [4] is just \(~10\mu\)sec slower than [3]. To summarise our impressions about synchronous data transfers:

- If not equal, single-packet synchronous \text{LAPI}_\text{Put}|\text{Amsend} transfers are faster than MPI_Ssend.
- Context-switching due to the execution of a completion/header handler specification costs \(~15\mu\)sec for each of them.
- \text{LAPI_Get} is slightly slower than \text{LAPI_Put} and \text{LAPI_Amsend}.
- The dispatcher appears to execute the completion handler specifications in a serial fashion.
- Overlapping appears to cost \(~10\mu\)sec.
- Manual implementation(s) of \text{LAPI_Get} turned out to be very expensive as they are prone to doubling context-switches.
Figure 15: Variations of the blocking LAPI_Get (including our manual versions) versus the synchronous LAPI_Put. Figures have been obtained from the single pings benchmark.

2.9 Multi-messaging

In this section we give an answer to whether multi-messaged point-to-point communications can achieve better performance than the single-messaged ones.

In multi-messaging, a fixed payload is delivered in an overlapping multi-part fashion. For instance, 8K can be delivered as 2x4K or 8x1K parts. Of course, inherent to multi-messaging are: (i) the start-up costs for each of the multiple data-transfer operations; and (ii) the context-switching on the other end. These issues though, become negligible for payloads whose cost is much greater than (i) and/or (ii). Also it is considered meaningful for each part to fill in at least one LAPI packet –i.e., each part should be ≥944 bytes. Finally, we want to know if, for a fixed payload there is a particular multi-messaging configuration, which outperforms the single messaging one.
Figure 16: Multi-messaged ping-pong. This benchmark uses the LAPI_Put routine, a dual plane arrangement and with interrupts enabled. The size of each messages is payload/NoM.

Figure 16 displays what sort of bandwidth is achieved by the multi-messaged version of the unidirectional ping-pong benchmark. Although pings and pongs are not crossing each other, the pinged payload is delivered as overlapping multi-parts (in the form of sequential LAPI_Puts). Each line refers to a particular ping payload, while the y-axis indicates in how many overlapping parts/messages (NoM) the payload is split into. For instance, when the payload is 512K and NoM=8, the ping consists of 8x64K overlapping LAPI_Puts. We can see that in general, single-messaged deliveries are always faster than the multi-messaged ones. Also, the larger the size of each message, the lesser visible is the effect of per-message overheads. So, regardless NoM, performance stabilizes as we move towards to a volume of data enough to let LAPI hit its peak (e.g. 512K). The only exception is 8K –a problematic payload, known to behave badly (Sections 2.2, p. 7 and 2.7, p. 19). When 8K are delivered as a 2x4K multi-message, there is a 20Mbytes/sec gain in bandwidth. There are such big overheads involved in the delivery of 1x8K messages that even the 4x2K multi-message is faster.

- Delivering some payload as multiple overlapping messages of smaller sizes results in no particular gain(s).
- Exceptional behaviour regards 8K messages, where the 2x4K multi-messaged form is significantly less expensive than 1x8K.

An alternative multi-messaging scenario suggests that instead of having message parts being fired by a single process we could have multiple processes pushing each part.
We examine whether the inefficiency encountered in unidirectional double-plane transfers (see Section 2.2, p. 7) is due to per-process limits. Typically, a user-space-based ping-pong requires two communicating processes, each sitting on a different LPAR. In our multi-process pings, we have $P$ processes per LPAR, such that process $N$ from the 1st LPAR ($N=0..7$) pings process $N+P/2$ from the other LPAR. Once PE 0 is done with its set of pings, it waits for the rest to complete and reports achieved bandwidth. This implies that reported figures are bound to the last-finishing process. Note that pairs execute with no awareness of other pairs’ progress. We found out that such type of transfers do not result in any performance advantages at all.

### 2.10 Configurations for Single-Packet Transfers

Active-messaging is an option of interest whenever some computation is required prior to the specification of a placeholder for the incoming data by the remote end. A single-packet AM can hold at most 944 bytes of parameters. We investigate whether there do exist faster configurations for single-packet transfers. For active-messages we use the notation $X/Y$, where $X$ is the size of the header handler parameters and $Y$ the size of the referenced buffer, such that $X+Y=944$. The programmer should note that:

- A 944 bytes LAPI\_Put; and a 944/0, 0/944 and 472/472 LAPI\_Amsend have exactly the same cost; 35.5\,µsec (non-overlapping) and 55\,µsec for the synchronous versions.
- No implicit prepackaging takes place for a set of sequential non-blocking asynchronous data pushes, even if the pushed payloads consume a total of a single packet. In other words, data transfers begin as soon as the LAPI\_Put|Amsend is fired.

### 3 Conclusions

In this document we introduced LAPI—the IBM Low-Level API. We described its core data transfer routines with a particular emphasis on its threads-model and associated issues (context switching). We captured the performance of point-to-point communications in a number of scenarios and against the IBM MPI.

LAPI does make use of both of the switch planes, but appears not to cope well under the presence of both. LAPI will stripe DP-ready payloads ($>8$ packets) across the two planes and almost double bandwidth for 16-packet payloads. From thereon, perfect DP-ready payloads (perfect multiples of 8 packets, and $>16$ packets) experience sharp overheads. On the other hand, DP-unready payloads ($\leq8$ packets) are faster in SP mode rather than DP mode, hardening the evidence of overheads, which are trigger-able in DP mode only. In general, LAPI is slower than MPI regardless the directionality of transfers.

LAPI is advantageous to MPI, when MPI transfers are or become synchronous. Contrary to LAPI whose data transfers are always asynchronous, the synchronicity of MPI transfers is constrained by the availability of eager space (which reduces according to the number of utilised processors and is controlled by the library). If not the same, LAPI asynchronous transfers are always faster than synchronous MPI transfers for unidirectional transfer of payloads $\leq128K$. 

25
But the real advantage of LAPI over MPI is in that LAPI easily overlaps computation with communication. This is contrary to MPI, which can overlap computation with eager transfers only. The means we used to ensure overlapping (LAPI_Gfence) are expensive though and apply when the cost of g/fencing is negligible compared to the cost of communications. Synchronous LAPI transfers present a new message to the network at the cost of 20µsec. The separation of this message from the data transfer is well defined (after completion) and controlled (manipulation of a completion handler). This allows the programmer to easily maneuver its overlapping with other data transfers –if not computation.

Although LAPI is significantly advantageous to synchronous MPI transfers, it is prone to excessive context-switching. Our figures displayed performance in INT mode only (with interrupts enabled) and not in POL mode (polling). The reason is that INT matches POL for our very benchmarks, which deploy blocking behaviour (LAPI_Waitcntr). In non-blocking scenarios though, INT can only accidentally match POL. Throughout two different tests (replacement of LAPI_Waitcntr and fully-synchronous manual LAPI_Get), we determined that the cost of context switching in LAPI is 15µsec -although this overhead impacts small payloads only.

Therefore LAPI leaves us with two options: either block to avoid the context switch, or attempt as much overlapping as possible.

4 References

