LAPI on HPS
Evaluating Federation

Adrian Jackson
August 23, 2004

Abstract
LAPI is an IBM-specific communication library that performs single-sided operation. This library was well profiled on Phase 1 of the HPCx system. Phase 2 of HPCx is now in service, using newer hardware and software than Phase 1. This report aims to capture the performance characteristics of LAPI, compare it to MPI’s performance, and investigate whether we can improve performance by tuning the default runtime environment.

1 Introduction
HPCx Phase 1 was a cluster system, composed of a large number of 8-processor shared-memory logical PARtitions (LPARs), connected via an interconnect and associated switch hierarchy (the SP2 switch or “Colony”). It had two communication libraries, MPI and LAPI, both of which were constructed on top of a common low-level communication layer. These libraries have been well profiled.

HPCx Phase 2 comprises a large number of 32-processors shared-memory frames, connected via an interconnect and associated switch hierarchy (the HPS switch or “Federation”). Each frame contains 32 Power4+, 1.7 GHz, processors, 32 GB of memory, and 4 switch links. Again, IBM have implemented both MPI and LAPI for HPS, however the communication libraries have been rearranged so that MPI is constructed from LAPI primitives. This means that whilst LAPI still uses the low-level IBM communication layer to send data, MPI uses LAPI functions to send its messages (MPI uses the new LAPI\_Xfer and MsgPoll functions to send data).

As well as changing the way that MPI and LAPI are implemented, IBM have also extended LAPI with additional functionality. This functionality is discussed in Section 1. The performance of Phase 2 LAPI is profiled, and compared with MPI, in section 2.

2 LAPI
The Communications Low-Level Application Programmers Interface (LAPI) [1] is an IBM library designed to give optimal communication performance on IBM hardware. It provides two communication mechanisms:

- Data Transfer
• Active Message

The data transfer functions implement the RMA Put (LAPI\_Put) and Get (LAPI\_Get) operations, as well as providing global synchronisation and completion checking functionality. As the Put and Get operations are inherently unilateral (i.e. the operation initiated by one process does not require the other process to take some complementary action), completion of a RMA function is signalled using predefined counters. These counters are incremented when the operation has completed, and can be checked using Getcntr (non-blocking) or Waitcntr (blocking) functions.

The active message function (LAPI\_Amsend) relies on user defined handlers, which are invoked and executed in the address space of the target process. As with the data transfer functions, active messages are non-blocking calls which require no complementary action by the target process. When an active message arrives at a target process the user defined header\_handler is executed. This defines where the data associated with the active message should be copied to, and the address of the completion\_handler. This second handler is called after the whole active message has been received, and can be used to perform additional processing on the received data.

All LAPI communication relies upon processes publishing the addresses of memory that can be accessed remotely. This is done using the LAPI\_Address\_init operation (a collective operation) which allows each process to maintain different memory address maps, but still access portions of each other memory (the alternative would be for each process to have the same memory address map, which is restrictive).

This functionality has been extended with the following features:
• UDP/IP as well as US protocols
• Inline completion handlers
• New polling functions
• Doubled Packet size (from 936 bytes to 1960)
• DGSP and associated LAPI Utils
• New data structures
• Statistics

2.1 Message Protocols

The LAPI library now has three potential communication modes (although US and UDP/IP modes are mutually exclusive for any give job):
• User Space (US) using the HPS
• User Datagram Protocol / Internet Protocol (UDP/IP) using the HPS
• Shared memory

The US protocol is an IBM specific communication mode, designed to allow users to fully exploit the performance of IBMs specific switches. UDP/IP protocol is included to allow users to exploit more generic switching networks, such as ethernet. The addition of the UDP/IP communication mode means LAPI and MPI now have access to the same communication modes. User space protocol should have lower latency than UDP/IP for HPCx as we are using the HPS. Bulk transfer is active for both protocols.
2.2 Completion Handlers

Completion handlers are used during active message operations. Use of a completion handler is optional. If specified, LAPI runs the code the user has defined in the completion handler on the target process after the final packet of the active message has been completely received into the target’s buffer.

Under normal execution, completion handlers are placed in a queue by the dispatcher thread, and executed by the completion handler thread. This allows messages to be progressed whilst the completion handler executes. However, this method involves a number of thread context switches that may effect performance, as well as delaying the execution of the completion handler, which will in turn delay the completion of the message as a whole.

This performance impact can be reduced by using inline completion handlers. This new functionality in LAPI allows for the prioritisation of the completion of certain messages, executing the completion handler from the dispatcher thread, reducing the execution time of the completion code and removing the overhead of thread context switching.

2.3 Polling Functions

LAPI can be run in either Polling or Interrupt modes. Interrupt mode allows messages to be progressed without the explicit intervention of the user, but does require an extra thread, the dispatcher thread, to handle the interrupt. This thread is created when LAPI is initialised.

New functionality has been added to facilitate polling mode; LAPI_Msgpoll, has been added to LAPI to allow users to check the progress of communications. The function explicitly runs the LAPI dispatcher, and is intended to be used when interrupts are disabled. This allows the user greater control over when and how often messages are progressed by controlling when the dispatcher thread is executed.

Another related function is the LAPI_Probe function, which is used to execute dispatcher thread to allow it to progress messages. This function is a more restrictive version of the LAPI_Msgpoll function, allowing the dispatcher to execute only once, whereas LAPI_Msgpoll allows the dispatcher to execute as many times as necessary. Again, this is to be used when interrupts are disabled.

2.4 Packet Size

The maximum storage for user data in a single packet has been doubled to 1960 bytes. This means that the LAPI system packet size has doubled from 1k (1024 bytes) to 2k (2048 bytes).

2.5 DGSP

Data gather/scatter programs (DGSPs), along with LAPI vectors, allow for non-contiguous data transfer. LAPI vector transfers are similar to scalar transfers, with vector functions available for Put, Get, and Active Message, designed to move data specified by the repetition of a block size and stride data template.

DGSPs provide more complex behaviour than vectors, allowing LAPI functions to operate directly on non-contiguous user buffers. The LAPI API provides functions and instructions to specify any data layout to be passed to the LAPI_Xfer function for transfer. DGSP functions are well documented in the
2.6 Data Structures

A number of new data structures have been added to the LAPI API to support new functionality. These include data structures for:

- Extended header handler support
- Message polling
- IP/US statistics reporting
- UDP support
- DGSP data transfers

2.7 Statistics

Users are now able to query the library to obtain packet statistics on messages sent by LAPI, for the whole program, or sections of codes. It is also possible to obtain debug information about the setup of the LAPI library and the way messages are sent.

3 Performance Analysis

The performance benchmarks used for this report are an agglomeration of tests previously carried out on the Phase 1 system, see HPCx technical reports HPCxTR0305[2], and HPCxTR0402[3].

The main focus of the performance analysis is the LAPI Put and Get functionality, although some MPI benchmarks were performed to provide comparisons. The LAPI runtime environment was also profiled to analyse the optimal configuration for LAPI programs.

3.1 LAPI Put and LAPI Get

The first benchmarks performed were simple pingpong programs, testing Put and Get over a range of message sizes (from 1 to 1 million integers), using a range of processor numbers. The time to perform a complete pingpong is measured (i.e. time to complete 2 communication operations), and a corresponding bandwidth worked out.

Both on-node, and off-node performance was assessed, with a comparison between Put and Get performance shown in Figures 1 and 2. The graphs show performance figures for 2 process pingpongs with the legend 1N2P standing for 1 node used with 2 processes on that node. Likewise 2N1P stands for 2 nodes used with 1 process per node. Therefore 2N1P shows off-node performance figures, and 1N2P shows on-node only performance figures. We can see from these graphs that Put and Get performance is identical for the current HPCx system (this was not the case for HPCx Phase 1).

More detailed performance characteristics are shown in Figures 3 and 4. These graphs show single processor results for the Get pingpong using 2 to 64 processor, across 2 nodes. As Put and Get have the same performance characteristics, these results are applicable to both. We can see from the graphs that on-node latency is considerably lower than off-node latency, as the on-node communication is
performed through fast shared memory, and not over the (comparatively) slow switch network. The graph also shows that off-node latency scales perfectly up to 4 processors per node (i.e. 2N4P), and almost perfectly up to 2N8P. This is because each compute node in the HPCx system has 4 connections to the switch network, but these cannot be exploited by a single process. Processes are bound to a switch link, and can only use that link for communications. This means that at least 4 processes are needed to use the 4 links, so when additional processes are added to the pingpong benchmark, up to 4 processes latency and bandwidth are unaffected as each process is using its own communication link. Over 4 processes, there is competition for links, so the per process performance is reduced.

Another performance characteristic that is evident from the graphs is the improvement in performance of all off-node communications for messages over 130KB. There is a corresponding drop in on-node performance. This is caused by a change in communication method from normal transfer to bulk transfer, and is further analysis in Section 3.4.

Table 1 compares performance numbers for the Put and Get benchmarks from the current system, and the previous system. Again, these numbers are for a single processor, and show that latency has halved and bandwidth doubled for the new system. From these results we can calculate that the maximum bandwidth available for LAPI off-node communication from a node (i.e. using at least 4 processors) is approximately 2.8 GB/s. The results also suggest that a program may be able to achieve approximately 6 GB/s using on-node communications (i.e. through the memory system), although this is entirely dependent on the level of competition for the shared memory bus.

### 3.2 LAPI Ping

The Put pingpong benchmark was then used to profile the performance of a single message operation, in essence a Ping benchmark. The performance was measured using two benchmarks, one which performed the same test as for the Put/Get comparison (i.e. ranging from 1 to 1 million integers), and the other that looked at the performance of small messages (i.e. 1 to 2000 integers). The results from the
Figure 2: LAPI Put and Get bandwidth comparison

Figure 3: LAPI Get latency

Table 1: Put and Get Comparison: Phase 1 and 2

<table>
<thead>
<tr>
<th></th>
<th>Latency (secs)</th>
<th>Bandwidth (GB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phase 2:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Put and Get</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Off-Node</td>
<td>2.7e^{-05}</td>
<td>0.7</td>
</tr>
<tr>
<td>On-Node</td>
<td>4.6e^{-06}</td>
<td>3.2</td>
</tr>
<tr>
<td>Phase 1:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Put</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Off-Node</td>
<td>5.5e^{-05}</td>
<td>0.23</td>
</tr>
<tr>
<td>On-Node</td>
<td>1.4e^{-05}</td>
<td>2.1</td>
</tr>
<tr>
<td>Phase 1:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Get</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Off-Node</td>
<td>9.8e^{-05}</td>
<td>0.23</td>
</tr>
<tr>
<td>On-Node</td>
<td>2.2e^{-05}</td>
<td>2.8</td>
</tr>
</tbody>
</table>
large message benchmarks are showing in Figures 5, and 6, with the performance of the two processor runs show by themselves in Figures 7, and 8. The results from the small message benchmarks are shown in Figures 9, and 10. The times for sending data in a single packet (i.e. up to 2KB) are shown in Figure 11.

The ping benchmark shows that LAPI put/get has a latency of $1.6e^{-05}$ secs off-node, and $2.32e^{-06}$ secs on-node. The equivalent bandwidth numbers are 2.45 GB/s on-node, and 0.67 GB/s off-node.

However, Figure 6 shows that the performance of LAPI on-node is much less regular that off-node performance, suggesting that LAPI performance is susceptible to contention for the memory system (as on-node communications is performed through shared memory).

The large message benchmark shows the same performance characteristics as previously seen in the Put/Get benchmarks. The small message benchmark shows the impact of packet boundaries on communication time for small messages. The latency for sending an off-node message of 100 bytes is almost the same as 1800 bytes, but the latency for sending a message of 2000 bytes is considerably more than that from sending 1900 bytes. This means that the effective bandwidth of small messages depends upon how much data is being sent in the message packet, and if you are sending 2 messages that contain 500 bytes each, then you would achieve better performance using a single message containing 1000 bytes. This is characterised in the off-node bandwidth graph for small messages (Figure 12). The impact of requiring an additional packet to be sent through the network is approximately $7e - 05$ secs. The on-node performance is not affected in the same way, and scales linearly with message size (at least up to 10KB).

### 3.3 MPI Pingpong

To enable a balanced assessment of LAPI on HPCx, it is necessary to compare it to the other communication library available, MPI. MPI was benchmarked using the same Ping benchmark as used for LAPI. Both **MPI_Send** (Standard send) and **MPI_Ssend** (Synchronous send) where benchmarked, although non-blocking communications were not evaluated.
Figure 5: LAPI Ping latency

Figure 6: LAPI Ping bandwidth

<table>
<thead>
<tr>
<th></th>
<th>Latency (secs)</th>
<th>Bandwidth (GB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send</td>
<td>On-Node</td>
<td>4.0e-06</td>
</tr>
<tr>
<td></td>
<td>Off-Node</td>
<td>1.16e-05</td>
</tr>
<tr>
<td>Ssend</td>
<td>On-Node</td>
<td>8.5e-06</td>
</tr>
<tr>
<td></td>
<td>Off-Node</td>
<td>3.1e-05</td>
</tr>
</tbody>
</table>

Table 2: Performance metrics for MPI_Send and MPI_Ssend on HPCx
Figure 7: LAPI Ping latency using 2 processors

Figure 8: LAPI Ping bandwidth using 2 processors
Figure 9: LAPI PingPong latency for small messages

Figure 10: LAPI PingPong bandwidth for small messages
Figure 11: 1, 2, 4 and 8 processor latency results for small message LAPI PingPong benchmark

Figure 12: Small message bandwidth for off-node LAPI PingPong benchmark
Figure 13: A comparison of MPI Send and Ssend latency for PingPong benchmark

Figure 14: A comparison of MPI Send and Ssend bandwidth for PingPong benchmark
Table 2 shows that Send has half the latency of Ssend for small messages, but ultimately achieves the same bandwidth. This is because Send uses a none synchronous communication method to send data, but this method only works for small messages (i.e. up to the eager limit, which is 64KB for this benchmark). The IBM communication system requires all large messages to be sent synchronously, meaning Send and Ssend are equivalent for large messages. Bandwidth and latency comparisons of Send and Ssend are shown in Figures 13 and 14 respectively. We use \texttt{MPI Send} for the rest of the MPI benchmarks.

Figures 15, and 16, show the performance of MPI for the same range of processor numbers that were using in the LAPI Ping benchmark. We can see from these graphs that MPI has a similar performance profile as LAPI, although LAPI has a lower latency for small on-node communications, and MPI has a lower latency for small off-node communications.

Figures 17, 18, 19, and 20 show the results for the small message benchmarks. It can be seen that, as with LAPI, MPI off-node communications are impacted by the LAPI packet size. However, the impact is not as large as for LAPI, with the difference between the packets being approximately $4.0e-05$ secs, nearly half that of an additional packet using the LAPI benchmark. This means that the bandwidth difference between sending an empty packet and a full packet is not as great for MPI.

MPI also has a different performance profile for small on-node messages. Figure 19 shows a regular bandwidth profile for MPI for all on-node benchmarks (i.e. 1N2P to 1N32P). The corresponding LAPI bandwidth graph (Figure 6) shows that LAPI has a much more chaotic profile for different numbers of processors. This suggests that memory contention within a node has more effect on LAPI communications than it does on MPI.

### 3.4 Altering Bulk Limit

The HPS supports \textit{bulk transfer} communications for large messages, a form of direct memory access which can give improved performance by reducing the number of times the data is copied by the communication system. However, this method of data transfer has a large associated startup cost, meaning it is impractical for messages under a certain size. The default lower limit for bulk transfer on HPCx is
Figure 16: MPI Send PingPong bandwidth

Figure 17: MPI Send PingPong latency for small messages
Figure 18: MPI Send PingPong latency for small messages

Figure 19: MPI Send PingPong bandwidth for small messages
150KB, but this value can be altered when a job is submitted to the system. We ran the LAPI Ping large message benchmark with a range of different lower bulk limits (set through the `MP_BULK_MIN_MSG_SIZE` variable). The results are shown in Figures 21 and 22.

From the graphs we can see that the default lower bulk limit on HPCx is not optimal for this benchmark. In fact, the lowest tested value for this variable, 63KB, gives the best performance.

3.5 Altering the Acknowledgement Threshold

Another variable that can altered in the LAPI runtime environment is `MP_ACK_THRESH`. This sets the number of packets that LAPI receives before it returns acknowledgements to the sending task, and can be any value between 1 and 31.

We testing this variable using the small message Ping benchmark with 2 processor on separate nodes (2N1P), and the results are shown in Figure 23. From this we can see that in general, altering this variable do not significantly change performance. However, it does appear that, for single packet messages, you can improve performance by setting this variable to 1. However, the performance improvement is small, and it may affect over aspects of your programs performance (i.e. reduce the computational performance by causing more interrupts to occur).

3.6 Communication Modes and UDP Packet Size

LAPI can use two different communication modes, User Space (US) and User Datagram Protocol (UDP/IP), although they are mutually exclusive for any given program run. User Space mode is designed specifically for use with the HPS whereas UDP/IP is a general purpose mode for use with any switch network (i.e. ethernet). Therefore, US mode should offer better performance than UDP/IP mode for LAPI on HPCx.

Figures 24, 25, and 26, show the results of running the LAPI Ping benchmark in both IP and US modes. It is obvious that US mode is superior to IP mode, with a much smaller latency, and higher bandwidth, apart from messages around the default bulk limit. Messages around the default bulk
Figure 21: Time to Perform Ping Benchmark for Varied Lower Bulk Limits

Figure 22: Time to Perform Ping Benchmark for Varied Lower Bulk Limits: Close-up of Bulk Region
Figure 23: Time for LAPI small message Ping benchmark when varying Acknowledgement Threshold

Figure 24: Comparison of US and UDP/IP protocol: Latency
Message Size (bytes)

Bandwidth (MB/s)

Figure 25: Comparison of US and UDP/IP protocol: Bandwidth

limit size have similar characteristics for both IP and US modes, although as previously documented, the performance of the US mode can be improved by tuning the bulk limit. Therefore we can confidently say that US mode is the correct choice for LAPI programs.

However, it is possible to tune the IP mode using the MP_UDP_PACKET_SIZE variable, which can be set to any value between 1024 and 65536. The LAPI Ping benchmarks for small and large messages were run in IP mode using different values of this variable, and the results are shown in Figures 27 and 28.

We can see from these graphs that it is possible to tune IP mode for large messages. The default value (8K) can be improved upon by using any value larger than that. However, even with this improvement, IP mode gives considerably poorer performance than US mode.

4 Summary

In the move from Phase 1 to Phase 2 of HPCx, the performance of LAPI (and MPI) has improved. LAPI latency has halved and bandwidth more than doubled. However, whilst MPI is now built on top of LAPI, the performance characteristics of both libraries suggest that the LAPI Xfer function has lower overheads than the traditional LAPI primitives (i.e. Put, Get, and Active Message). In fact, MPI has a smaller latency than LAPI for off-node messages, suggesting that the IBM communication system has been tuned to optimise Xfer performance over traditional LAPI performance. It may be of benefit to construct some benchmarks aimed specifically at the LAPI Xfer function, to investigate this difference.

Both LAPI and MPI have different performance characteristics for small and large messages. This means that the performance of communications in a users’ application depends upon the type of communications they are performing, the size and frequency of messages. If a code generally use only small messages, it should benefit from tuning those messages to the LAPI packet size. However, if an
Figure 26: Comparison of US and UDP/IP protocol: Small Message Latency

Figure 27: Large message benchmark of UDP Packet Size variations
During the benchmarking for this report, we investigated many different settings for LAPI, some of which are not documented here as they revealed nothing of interest. From these investigations we can recommend the default HPCx runtime environment\(^1\) for providing the best performance for codes with regular communication patterns and light computational load, such as those exhibited by our benchmarks. However, this does not mean that the default environment will give the best achievable performance for a users’ code, given that these may not conform to standard communication patterns, and perform intensive computations. What it does suggest is that the current setup of HPCx is a good starting point for programs that use LAPI, but users should consult the documentation[1] to ascertain whether they could improve performance.

\(^1\)With the exception of the bulk limit which can be tuned to improve performance
References


[3] Introducing LAPI and Capturing the Performance of its Point-to-Point Communications on HPCx and the Colony SPswitch2 C. Kartsaklis, EPCC, December 5th, 2004, http://www.hpcx.ac.uk/research/hpc/technical_reports/HPCxTR0402.pdf