Performance and Experience with LAPI – a New High-Performance Communication Library for the IBM RS/6000 SP

Gautam Shah† Jarek Nieplocha✣ Jamshed Mirza† Chulho Kim† Robert Harrison✣
Rama K. Govindaraju† Kevin Gildea† Paul DiNicola† Carl Bender†

†IBM Power Parallel Systems
Poughkeepsie, NY 12601
<gautam@vnet.ibm.com>

✣Pacific Northwest National Laboratory
Richland, WA 99352
<j_nieplocha@pnl.gov>

Abstract

LAPI is a low-level, high-performance communication interface available on the IBM RS/6000 SP system. It provides an active-message-like interface along with remote memory copy and synchronization functionality. It is designed primarily for use by experienced programmers in developing parallel subsystems, libraries and tools, but we also expect power programmers to use it in end-user applications. IBM developed LAPI as a part of a project with Pacific Northwest National Laboratory (PNNL) to optimize the performance of the Global Arrays (GA) toolkit and its applications on the IBM RS/6000 SP. We provide an overview of LAPI characteristics and discuss its differences from other models such as MPI-2. We present some base performance parameters of LAPI including latency and bandwidth and compare it with performance of the MPI/MPL. The Global Arrays library from PNNL was ported to LAPI to exploit the performance benefits of this new interface. Experience using LAPI to implement GA and the performance of the resulting library are presented.

1 Introduction

The IBM RS/6000 SP [1] is a general-purpose scalable parallel system based on a distributed-memory message-passing architecture. Generally available systems range from 2 nodes to 128 nodes. The uniprocessor nodes are available with the latest Power2-Super (P2SC) microprocessors‡. The nodes are interconnected via an adapter to a high-performance, multistage, packet-switched network for interprocessor communication capable of delivering bi-directional data-transfer rate of up to 110 MB/s between each node pair. Each node contains its own copy of the standard AIX operating system and other standard RS/6000 system software. Many communication APIs that utilize the high performance switch are available on the SP system including MPI, MPL and PVM [2,3]. All of them provide a message-passing interface. The MPL *receive-and-call (rcvncall)* operation allows implementation of one-sided access to memory of a remote process. There are implementations of MPI and MPL that exploit user space communication on the SP. User space communication improves performance by avoiding the expensive system calls, mode switch and extra copy overheads associated with communication interfaces that have a path through the kernel. However, there are other overheads associated with user space message passing implementation. For instance, in order to satisfy the MPI/MPL semantics, the implementation often needs to keep multiple copies of the data. Further, the cost of interrupts is fairly high in the implementation of the above libraries. LAPI (Low-level Applications Programming Interface), a new communications library available as part of the SP software, was designed with the following primary objectives in mind: *Performance:* The primary design consideration for LAPI was to define syntax and semantics that would allow efficient implementation on the underlying hardware and software communications infrastructure of the SP. We believe that we have succeeded in eliminating much of the protocol overheads discussed above. *Flexibility:* LAPI is based on the remote memory copy (RMC) model. RMC is a one-sided programming paradigm similar to the load/store model of shared memory programming. The one-sided model eases some of the difficulties of the send/receive model, which uses two-sided protocols. The send/receive paradigm is not very easy to use when the communication pattern between processes cannot be easily determined a priori; this includes applications that use sparse matrices, adaptive grids, any kind of indirect array references, or dynamic load balancing, for example, in the electronic structure calculations. *Extensibility:* In addition to a set of basic functions, LAPI also provides the active message style interface [4]; with this interface, users can add additional communications functions that are customized for their specific application or environment. Note that the choice of using LAPI or MPI/MPL depends on the application requirements. IBM offers the use of both MPI and LAPI in the same application.

The material presented in this paper is based on results of a project involving IBM and PNNL. The venture was initiated to optimize performance of the Global Arrays [5,6] on the SP system. The GA provides portable shared-memory style access to distributed data structures. The implementation uses system specific features to provide the best possible performance to the application. In Section 2, we provide an overview of LAPI and compare it with other approaches. In Section 3, we briefly discuss related work in one sided communication and active messages, and contrast it with LAPI. In Section 4, we discuss the base performance parameters of LAPI and compare it with the MPI/MPL implementation on the SP. In Section 5, our experience in optimizing GA is discussed along with performance of the toolkit and its applications. We conclude with some remarks on the project accomplishments and future work in Section 6.

2 Overview of LAPI

LAPI is an asynchronous communication mechanism intended to provide users the flexibility to write parallel programs with dynamic and unpredictable communication patterns. LAPI is architected to be an efficient (low latency, high bandwidth) interface. In order to keep the LAPI interface as simple as possible it is designed with a small set of primitives. However the limited set does not compromise on functionality expected from a communication API. LAPI functionality includes data communication as well as synchronization and ordering primitives. Further, by providing the active message function as part of the interface the

‡. The SP nodes may also be SMPs based on the PowerPC processors; here we deal with the type of nodes more commonly used in technical computations.

LAPI design allows users to expand the functionality to suit their application needs. The set of LAPI functions are shown in Table 1 and are discussed briefly below. For a detailed description of the LAPI functions please refer to [7].

Operations	Functions
Setup	*LAPI_Init, LAPI_Term*
Active Message	*LAPI_Amsend*
Data Transfer	*LAPI_Put, LAPI_Get*
Mutual Exclusion	*LAPI_Rmw*
Signaling Communication Progress	*LAPI_Setcntr,LAPI_Waitcntr, LAPI_Getcntr*
Ordering	*LAPI_Fence, LAPI_Gfence*
Address Exchange	*LAPI_Address_init*
Environment Query/Setup	*LAPI_Qenv, LAPI_Senv*

Table 1: LAPI Functionality

2.1 Active Message Infrastructure

Active Message (AM) was selected as the underlying infrastructure for LAPI. We use the term *origin* to denote the task (or process or processor) that initiates a LAPI operation, and the term *target* to denote the other task whose address space is accessed by the LAPI operation. The active message includes the address of a user-specified handler. When the active message arrives at the target process, the specified handler is invoked and executes in the address space of that process. Optionally, the active message may also bring with it a user header and data from the originating process. The user header contains parameters for use by the header handler in the target process. The data is the actual message the user intends to transmit from the origin to the target. The operation is unilateral in the sense that the target process does not have to take explicit action for the active message to complete. Buffering (beyond what is required for network transport) is not required because storage for arriving data (if any) is specified in the active message, or is provided by the invoked handler. The ability for users to write their own handlers provides a generalized yet efficient mechanism for customizing the interface to one's specific requirements. LAPI supports messages that can be larger than the size supported by the underlying network layer. This implies that data sent using an active message call will arrive in multiple packets; further these packets can arrive out of order. This places some requirements on how the handler is written. When the active message brings with it data from the originating process, LAPI requires that the "handler" be written as two separate routines:

- A *header_handler* function: This is the function specified in the active message call. It is called when the first packet of the message arrives at the target, and it provides the LAPI dispatcher (a part of the LAPI layer that deals with the arrival of messages and invocation of handlers) with: a) an address where the arriving data of the message must be copied, and b) the address of the optional *completion* handler; and
- A *completion_handler* which will be called after the whole message has been received (i.e. all the packets of the message have reached the target process).

The decoupling of the handler into a header handler/completion handler in the active message infrastructure allows multiple independent streams of messages to be sent and received simultaneously within a LAPI context. At any given instance LAPI ensures that only one header handler per LAPI context is allowed to execute. The rationale for this decision is that the header handler is just expected to return a buffer pointer for the incoming message and locking overheads might be comparatively expensive. Further, while the header handler executes, no progress can be made on the network interface. Multiple completion handlers are allowed to execute concurrently per LAPI context (the user is responsible for any synchronization among the completion handlers).

Figure 1 illustrates the flow of data and control in a LAPI active message. A process on the origin makes the LAPI_Amsend call. The call initiates a transfer of the header *uhdr* and data *udata* at the origin process to the target process specified in the LAPI active message call. As soon as the user is allowed to reuse *uhdr* and *udata*, an indication is provided via *org_cntr* at the origin process. At some point (Step 1) the header and data arrive at the target. On arrival at the target, an interrupt is generated which results in the invocation of the LAPI dispatcher. The LAPI dispatcher identifies the incoming message as a new message and calls the *hdr_hndlr* specified by the user (Step 2) in the LAPI active message call. The handler returns a buffer pointer where the incoming data is to be copied (Step 3). The header handler also provides LAPI with an indication of the completion handler that must be executed when the entire message is copied into the target buffer specified by the header handler. The LAPI library moves the data (which may be transferred as multiple network packets) into the specified *buffer*. On completion of the data transfer the user-specified completion routine is invoked (Step 4). After the completion routine finishes execution, the *tgt_cntr* at the target process and *cmpl_cntr* at the origin process are updated indicating that the LAPI active message call is now complete. LAPI also provides a set of defined functions built on top of the active message infrastructure. These defined functions provide basic data transfer, synchronization, signaling, and ordering functions. LAPI can be used in either interrupt or polling mode. The typical mode of operation is expected to be interrupt mode. In the interrupt mode, a target process does not have to make any LAPI calls to assure communication progress. Polling mode may be used to provide better performance by avoiding the cost of interrupts. However a user of polling mode should be aware that in the absence of appropriate polling, the performance may substantially degrade or may even result in deadlock. The active message infrastructure and most LAPI functions built on top of that are non-blocking. The non-blocking nature allows the LAPI user to have a task initiate several concurrent operations to one or more target tasks. In our implementation, these concurrent calls return as soon as the messages has been queued at the network, and do not have to wait for the communication event to actually complete. This "unordered pipelining" architecture results in significantly reducing (hiding) the per-operation latency.

2.2 Data Transfer Operations

LAPI provides get and put operations to allow basic data copy from the address space of one process to that of another. They are sometimes referred to as remote memory copy (RMC) operations. Put copies data from the address space of the origin process to the address space of the target process; Get pulls data from the target process and copies it into the origin process. These operations are semantically unilateral or one-sided. The get or put is initiated by the origin process, and no complementary action by the target process is necessary for the call to complete. This is unlike traditional send/receive semantics, where a send has to be matched at the other end with a corresponding receive being posted with matching parameters before the data transfer operation can complete. Since get and put are unilateral operations, non-blocking, and not guaranteed to complete in order, the user is responsible for explicit process synchronization when necessary for program correctness.

```
LAPI_Get(handle, target, len, tgt_addr, org_addr, tgt_cntr, org_cntr)
LAPI_Put(handle, target, len, tgt_addr, org_addr, tgt_cntr, org_cntr, cmpl_cntr)
LAPI_Amsend(handle, target, hdr_hdl, uhdr, uhdr_len, udata, udata_len,
            tgt_cntr, org_cntr, cmpl_cntr)
```

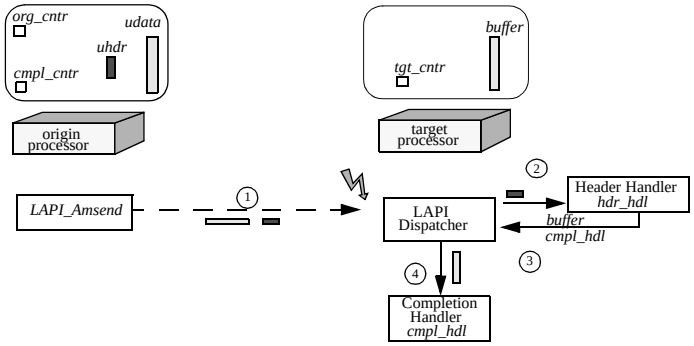

Figure 1: Interface to LAPI communication operations (left) and flow of data in the LAPI_Amsend operation (right)

2.3 Signaling completion of communication events

Put, *Get* and *AM* are unilateral communication operations that are initiated by one task (the origin), but an indication of the completion of a communication operation is provided at both ends. The definition of when a *put* or *get* operation is complete needs some discussion. Intuitively, the origin may consider a *put* as complete when the data has been moved from the origin to the target, (i.e., The data is available at the target, and the origin data may be changed). However, another equally valid interpretation is one where the origin task considers the operation to be complete when the data has been copied out of its buffer and either the data is safely stored away or is on their way to the target. The target task would consider the *put* complete, when the data has arrived into the target buffer. Similarly, for a *get*, the target task may consider the operation to be complete when the data has been copied out of the target buffer, but has not yet been sent to the origin task. In order to provide the ability to exploit these different intuitive notions, LAPI has a completion notification mechanism via the use of counters. The user is responsible for associating counters with events related to message progress. However, the counter structure is an opaque object internally defined by LAPI and the user is expected to access the counter using only the appropriate interfaces provided in LAPI. The user may use the same counter across multiple messages. This gives the user the freedom to group different communication calls with the same counter and check their completion as a group. The LAPI library updates the user specified counters when a particular event (or one of the events) with which the counter was associated has occurred. The user can either periodically check the counter value (using the non-blocking polling LAPI function *Getcntr*) or can wait until the counter reaches a specified value (using the blocking LAPI *Waitcntr* function). On return from the *Waitcntr* call, the counter value is automatically decremented by the value specified in the *Waitcntr* call.

2.4 Synchronization

LAPI operations decouple synchronization from data movement and there is no need for bilateral coordination of data transfers between the origin and target. For maximum performance concurrent operations may complete out of order. As a result, data dependencies between the source and the destination must be enforced using explicit synchronization as is the case in the shared memory programming style. However, in many cases the program structure makes it unnecessary to synchronize on each data transfer. LAPI provides atomic operations for synchronization.

2.5 Message Ordering & Atomicity

Two LAPI operations that have the same origin task, are considered to be ordered with respect to the origin if one of the operations starts after the other has completed at the origin task. Similarly, two LAPI operations that have the same target task, are considered to be ordered with respect to that target, if one of the operations starts after the other has completed at the target task. If two operations are not ordered they are *concurrent*. LAPI provides no guarantees of ordering for concurrent communication operations. For example, consider the case where a node issues two non-blocking *puts* to the same target node, where the target buffers overlap. These two operations may complete in any order, including the possibility of the first *put* partially overlapping the second, in time. Therefore, the contents of the overlapping region will be undefined, even after both the *puts* complete. Waiting for the first to complete (for instance using the completion counter) before starting the second, will ensure that the overlapping region contains the result of the second, after both *puts* have completed. Alternatively, a *fence* call can be used to enforce order.

3 Related Work

The MPI-2 one sided communication (MPI1S) [8] provides a standard interface for one sided communication operations. It was designed to be portable across multiple platforms but not necessarily provide optimum performance for any particular platform. LAPI differs from MPI1S in many ways, for example:

- MPI1S communication operations are restricted to windows of address space marked for MPI1S operations. No concurrent accesses to the same window by multiple processes are allowed. LAPI has no such restrictions.
- LAPI progress rules are less restrictive and ambiguous than those in MPI1S. For example, to avoid uncertain interpretation of the progress rules in different MPI1S implementations (p.142 in [8]), the remote side has to post a synchronization (MPI_WIN_FENCE or MPI_WIN_POST).
- LAPI provides a simple RMW (read-modify-write) mechanism with four atomic primitives for *Swap*, *Compare_and_Swap*, *Fetch_and_Add*, *Fetch_and_Or* for synchronization whereas MPI1S has three different synchronization mechanisms which are quite complex: (a) MPI_Lock, MPI_Unlock, (b) MPI_Start, MPI_Complete, MPI_Post, MPI_Wait and (c) MPI_Fence.
- Signaling completion of a communication operation is quite different. LAPI provides 3 counters (2 at the origin and 1 at the target) which are incremented to signal completion of communication operations, whereas in MPI1S completion is indicated when the control returns from the epoch end indicated by one of three synchronization mechanisms.
- LAPI unlike MPI, does not have any support for different data-types. Also, LAPI does not have any concept of grouping tasks into a communicator or process groups. All LAPI communication calls (except for *LAPI_Address_init* and *LAPI_Gfence*) are point to point communication calls.

Overall, LAPI provides a much simpler programming model than MPI1S (in terms of progress rules, ease of implementation and ease of use).

LAPI active messages differ from the other active message interfaces and implementations in terms of API and semantics. The progress rules do not require remote processor polling. However, LAPI performance in polling mode can benefit from remote process probing for incoming messages. Other unique features of LAPI active messages include decoupled header and completion handlers and a very flexible mechanism for signaling communication events using three counters (see Figure 1). The decoupled header and completion handlers provide several advantages over other interfaces for large data transfers. The application is not required to specify the address of the remote buffer like *am_store* in [9]. The address, if available, can be transferred in the header for use by the header handler; if the address is not available it can also be determined on the remote process inside the header handler. The FM *FMf_send* interface in [10] does not require the remote buffer address but it allocates the buffer upon message arrival. In LAPI, by allowing the user to manage the buffer allocation, it may be possible to avoid redundant memory copies and better deal with limited memory resources. The origin, target, and completion counters can be used to signal to the application the important events associated with the progress of communication initiated by LAPI *Amsend, Put* and *Get* operations. Although the LAPI communication calls are non-blocking, the blocking version is a simple extension by immediately waiting on the appropriate counter after issuing the non-blocking call. The sender can detect when the local buffer can be reused and when the data transfer completed. The remote process using a single target counter can be notified about the completion status of a single or multiple messages targeting its address space. Other AM interfaces use handler functions to signal such events. LAPI provides a simple mechanism to combine multiple event signaling and allows the application to avoid the overhead associated with message handlers when their sole role is event signaling.

4 LAPI Performance Study

In this section we discuss the performance of LAPI implementation on the SP with respect to latency, bandwidth, etc. in polling and interrupt modes. We also compare the performances of LAPI and MPI/MPL. The measurements were made in user space mode on an SP with 120MHz P2SC nodes, SP switch and adapter. The latency experiments for LAPI and MPI were performed in polling and interrupt mode using 4-byte messages and results are shown in Table 2. The MPI polling measurements were done using the latest MPI (threaded) library. The round-trip interrupt measurement was done using MPL *rcvncall* mechanism with target task sending back message to the origin from the interrupt handler. We note that only the non-threaded version of MPL library is available and therefore the polling and interrupt measurements for MPI/MPL are done using different libraries.

Measurement	LAPI [µs]	MPI/MPL[µs]
polling	34	43
polling round-trip	60	86
interrupt round-trip	89	200

Table 2: Latency Measurements

The non-blocking nature of LAPI communication calls (LAPI Put, Get, etc.) allow multiple communication calls to be pipelined and communication/computation to be overlapped. An important performance metric is the amount of time it takes LAPI to issue a communication call which we call the pipeline

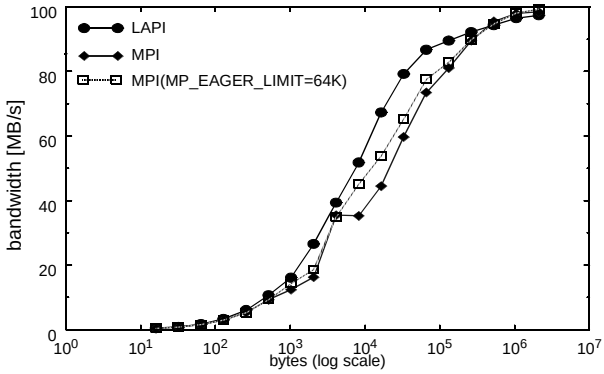

Figure 2: LAPI and MPI bandwidth

latency. It is measured by computing the amount of time it takes for a LAPI *Put/Get* call to return control to the user program. The pipeline latency for *Put* is 16µs and for *Get* is 19µs. It includes the time for a Put message or a Get request to be injected into the network.

The bandwidth benchmarks involved two tasks and measured the one way bandwidth for message sizes varying from 16 bytes to 2MB. The LAPI one-way bandwidth was measured by having one task make a LAPI_Put call to the other task and waiting for it to complete. The MPI one-way bandwidth was measured by running the same kind of experiment using message passing. MPI bandwidth is reported using the default settings as well as by setting the MP_EAGER_LIMIT environment variable to 65536. By choosing the MP_EAGER_LIMIT we are changing the point where the MPI implementation uses a rendezvous protocol. The results of the experiment are plotted in Figure 2. The asymptotic one-way bandwidth in LAPI is approximately 97MB/s, whereas MPI achieves around 98MB/s. The message size at which the transfer rate is half the asymptotic rate, is approximately 8Kbytes in LAPI and 23Kbytes in MPI. This shows that the LAPI bandwidth rises much faster than the MPI bandwidth as is clear from Figure 2. LAPI's better performance can be attributed to the following reasons, including:

- The one sided nature of LAPI helps in the implementation of simpler communication protocols avoiding complex matching and buffering overheads. The MPI standard on the other hand imposes complex semantics of ordering, matching, grouping and buffering which can lead to higher implementation overheads.
- LAPI has no ordering requirements and hence the amount of state that needs to be maintained is less.
- LAPI has a small but powerful set of functions which help in easier and more efficient implementation.

For medium sized messages (256 - 64K)bytes most commonly used in applications bandwidth in LAPI) is considerably greater than in MPI. The peak bandwidth in MPI is slightly greater than in LAPI because the LAPI packet header size (48 bytes) is larger than the MPI packet header size (16 bytes). Hence the LAPI payload per packet is smaller. The reason for the larger header size stems from the one sided nature of LAPI; where the origin side making the LAPI call must specify the parameters to be used on the target side as well and all these parameters need to be packed in the packet header. Reducing the packet header size in large message is a subject of future work. The slight flattening of the bandwidth curve for MPI for message size greater than 4Kbytes occurs because for message sizes greater than 4K MPI switches from *eager* protocol to *rendezvous* protocol resulting in an extra round-trip cost. This flattening can be avoided by setting the

MP_EAGER_LIMIT environment variable to 64K (the maximum value). The difference between the default MPI bandwidth curve and the MPI curve with the modified setting is caused by the extra round-trip in the rendezvous protocol. The difference between the MPI bandwidth with the MP_EAGER_LIMIT setting and the LAPI bandwidth caused by an extra copy in MPI.

5 Example User Library: Global Arrays

In this section we discuss implementation issues and performance of the Global Array toolkit [5,6] as an example of a user-level library that uses LAPI. We had earlier mentioned that LAPI was designed to aid development of libraries, tools and compiler run-time systems rather than general-purpose applications. Although LAPI can be used by "power users" to develop complete applications, it requires experience using concurrent programming concepts such as threads, synchronization, memory consistency, and asynchronous handlers. The GA portable shared-memory-programming model has been adopted in a variety of applications ranging from financial security forecasting, parallel rendering, molecular dynamics to numerous electronic-structure quantum chemistry calculations. Some of them contain hundreds of thousands lines of code. All applications that we and others tried ran unchanged on top of LAPI. In the following subsections, we give a brief description of the GA model, its implementation, and performance. We address differences between GA and LAPI memory models, describe techniques used to achieve optimum performance for variety of message requests sizes and access patterns, manage the resources, implement mutual exclusion, and assure robust and deadlock-free execution.

5.1 Characteristics of the Global Array Model

GA supports an abstraction of shared objects (dense 2-dimensional arrays) for message-passing applications. GA operations such as *put*, *get*, *scatter* and *gather* transfer data between local and global arrays in a shared-memory style. Synchronization operations such as *locks*, atomic *read-and-increment*, and *barrier* are provided. An atomic *accumulate* (reduction) operation can be used to combine local and remote data values. GA operations have a 2-dimensional array API motivated by the HPF notation. Many shared-memory programming facilities [11,12,13,14] hide from the user performance differences in accessing remote vs. local data and present a flat view of the memory hierarchy. GA makes users aware of the memory hierarchy of the current MPP systems. It recognizes variable costs of accessing remote and local data, furnishes to the user control over data distribution, and provides full locality information and control. These features have been essential for achieving good performance and scalability of massively parallel applications on distributed-memory systems [15,16,17]. GA operations are unilateral, just as is the case with LAPI operations. The progress of operations does not depend on the target process taking any action (such as polling) for requests to be serviced in the target process address space. While polling can be effective in the compiler run-time systems of languages used to develop complete applications, it is not practical for many applications that use large standard libraries. Use of polling in such applications would require inserting polling calls to *all* of the application code including the large standard libraries [18]. There are similarities and differences between memory models in GA and LAPI. In both models, remote store operations do not complete upon return from the library call. Both GA and LAPI provide fence calls to enforce completeness and ordering of remote store operations. LAPI operations (*puts* and *gets*) can complete out of order. GA allows out-of-order completion only for operations that reference non-overlapping array sections. For example, two consecutive *put* calls targeting array section A(1:100,2:2) and A(2:2,300:400) can complete in arbitrary order. This would not be the case if second *put* targeted A(2:2,1:400), for example. The *accumulate* operation is commutative and thus the completion order is arbitrary.

5.2 MPL implementation

A previous implementation of GA on the RS/6000 SP-1/2 used the MPL interrupt-receive *rcvncall* functionality [6]. Access to the remote data was accomplished by sending an MPL request message that interrupted the target process and invoked a message-handler. The handler copied the data from the message buffer to local memory of the target process (*put/scatter*) or copied data from the local memory of the target process to another message buffer and sent the requested data back to the origin process in the case of *get/gather* operation. The atomicity of *accumulate* or *read-and-increment* operations was accomplished by disabling and enabling interrupts (with *lockrnc*) and taking advantage of the single-threaded execution of the handler or the application code. This mechanism worked reliably despite performance problems. First, high latency (>300 μS for the get operation on the *previous* generation of the SP) was caused by the AIX overhead in creating the handler context. Second, it was not possible to utilize the network bandwidth because of the necessary additional copies to and from message buffers on both sides. The alternative to MPL solutions such as a dedicated "hidden-agent" thread to service remote requests or polling-based implementation of Active Message from Cornell [19] have been considered. However, they either appeared inefficient (a message-passing library integrated with a preemptively scheduled thread-package would be needed in "hidden-agent" approach) or not consistent with the GA progress rules (because of polling).

5.3 LAPI implementation

The key considerations in designing the LAPI-based protocols for GA implementation were:

1. need for good performance profile for a wide range of GA message sizes and different array section access patterns which is expected by users from a general purpose library
2. out-of-order delivery of messages in LAPI,
3. efficient management of the AM buffer memory, and
4. robustness (ability to handle high-levels of contention, deadlock-free execution etc.).

LAPI provides only contiguous interface to remote memory copy operations. Noncontiguous data can transferred in multiple contiguous chunks in separate put/get messages. The significant message overhead and poor utilization of the message packet space makes this solution not effective for small and medium chunks. Alternatively, the AM interface can to be used to send all noncontiguous data in a single message. Similarly to the MPL *rcvncall* implementation two extra memory copies are required for noncontiguous data in this approach. Since remote memory copy operations inherently have better performance (no user handlers are executed or intermediate buffering is required) than active message interface, GA uses hybrid protocols that switch between remote memory copy operations and active messages for implementing get, put, scatter and gather operations. Active messages have been used before in the GA implementations under SUNMOS [20], on the Intel Paragon, and related Fast Messages [21] have been used on the clusters of PCs [22]. The hybrid protocols have been designed to achieve optimal transfer rate for a wide range of message sizes and array access patterns. The thresholds used for switching between different protocols are selected empirically to maximize the performance.

5.3.1 Management of AM buffers

The application is responsible for management of the LAPI AM buffer memory. LAPI requires that the application specify in the AM header handler a pointer to receive buffer for every incoming

active message that contains user data. Since LAPI uses this buffer to copy data from the network, the header handler cannot block or return a NULL pointer. The data can be consumed by the ongoing computation inside completion handler and then the buffer space can be released. The completion and header handlers are executed independently by potentially separate threads. There can be many messages received and header handlers executed before the completion handler for the first message is invoked. Unlike the send/receive operations in the message-passing model, the GA remote store and accumulate operations are one-sided, and there can be multiple outstanding operations issued by one node to another. In case of a contention in the GA application, the rate of data arrival can be higher than the rate at which the data is consumed in the completion handler which could quickly deplete memory available at the node. These considerations make dynamic memory allocation for active-message buffer not practical for GA. Common solutions to similar problems in message-passing libraries use rendezvous protocols to negotiate buffer space before sending the data or preallocate multiple buffers for incoming messages. Due to higher start-up costs, such protocols are suitable for larger messages. Protocols that use preallocated buffers offer lower latency. Hybrid protocols combine preallocated message buffers for short messages with a round-trip protocol for larger messages. Preallocating one buffer for every remote process is not an optimal solution for the GA flow control. The model does not impose a limit on the number of outstanding store operations (targeting non-overlapping memory locations) that a process can send to another process without blocking. Fortunately, the LAPI Active Messages offer two important features that allow avoid preallocating multiple buffers: 1) a substantial room for user data in the AM header and 2) pipelining. Active Messages can transfer a substantial amount of user data(the packet size less LAPI AM header which on the current SP switch leaves around 900 bytes to the application). The exact amount is implementation specific and can be obtained through the *LAPI_Qenv* operation. GA exploits the pipelining by splitting medium-size requests into multiple active messages that each carry up to a 900-byte payload. They are processed by LAPI with reduced overhead compared to the cost of processing a single message. Pipelining works well (see Section 4) because on the sender side LAPI internally copies smaller messages (since retransmissions might be required in a case of switch failures) into its internal buffers, sends the message, and returns immediately without waiting for the acknowledgment. It allows a process to prepare and send the next active message in a very short time. The remote node can receive it before the processing of a previous message is complete thus avoiding an interrupt.

5.3.2 Ordering and Completion of Operations

LAPI fence operations can be used to enforce order and completion of data transfers through the network in the LAPI *put*, *get* and active message interfaces. They do not ensure the operational completion. When a fence operation returns, for the outstanding active messages this event indicates that data has been copied out from the network to the remote user buffers but the status of corresponding completion handlers is not known. In order to enforce completion or ordering of operations implemented on top of active messages one can use completion counters in *LAPI_Amsend* interface. In particular, an array of generalized counters (one per remote node) is employed in GA. A generalized counter structure contains a LAPI counter (used as completion counter for both *LAPI_Amsend* and *LAPI_Put*), a GA operation code (*put/scatter* etc.) for the most recent operation that used AM, and the number of requests issued. This number is passed to *LAPI_Waitcntr* to wait for completion of the outstanding active messages targeting a particular node when necessary (GA *fence* or *barrier*). The GA operation code is used to recognize operations that do not require ordering (for instance, accumulate is commutative) and to avoid redundant fencing.

5.3.3 Mutual exclusion and related considerations

Data transfer in the *put* and *accumulate* operation (that combines origin and target data in a manner similar to DAXPY) is almost identical. However, with accumulate the target data has to be updated atomically. Even on a single-processor SP node, there can be up to three threads executing the accumulate operation: the main application thread, completion handler and header handler (for short messages) threads. Mutual exclusion is implemented with the Pthread mutexes. Some care is needed to avoid de-scheduling of the LAPI thread that runs the header handler, when for short requests the mutex protecting a critical section has already been acquired by another thread. This could stall the network adapter, possibly cause packet loss, and consequently require data retransmission.

5.4 Performance

We present performance of the LAPI version of GA and compare it with the MPL implementation. The performance studies were performed on the 512-node IBM RS/6000 SP at PNNL. This system runs AIX 4.2.1 with PSSP 2.3 parallel software environment which includes LAPI. We used "thin" nodes with the 120MHz P2SC processors. Our synthetic benchmark runs on four nodes and measures performance of *get* and *put* operations that reference data in global array sections located on remote nodes. We timed a series of operations with the series length decreasing as the request size increases. Every request issued by node 0 accesses other nodes in a round-robin fashion. To avoid caching effects, each time a different array patch is referenced. The benchmark measures performance for both square 2-D as well as 1-D array sections. Since the leading dimension of the 2-D array does not match patch dimension, non-contiguous (strided) data is referenced. The latency measured for transfer of a single element (8-bytes) of a double-precision array is 94.2µs in GA *get* and 49.6µs for *put* in the LAPI implementation. In the MPL implementation, the corresponding numbers are 221µs for GA *get* and 54.6µs for *put*. The GA bandwidth profile in the data transfers ranging from 8-bytes to 2MB is shown in Figures 3 and 4.

Performance of the GA implementation using LAPI for GA *put* for large and small requests is better than when using MPL. Since the operation is non-blocking the much larger buffer space in MPL/MPI allows the send operation to return to the application sooner for messages larger than 1KB and smaller than 20KB. For larger messages, buffering of all the data is not possible on the sender side and LAPI implementation is faster. For the 1-D requests, GA uses *LAPI_Put* directly and avoids the copy overhead required in the AM-based protocols. This allows GA *put* to achieve bandwidth within 6% of *LAPI_Put* for larger mes-

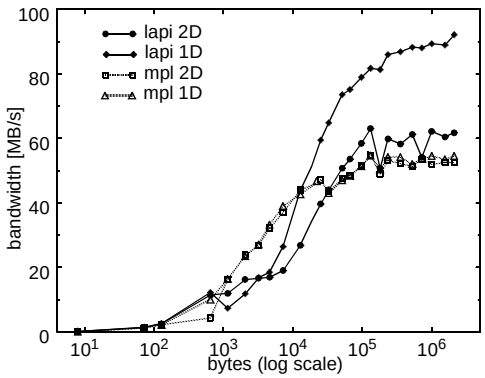

Figure 3: Performance of GA put under LAPI and MPL

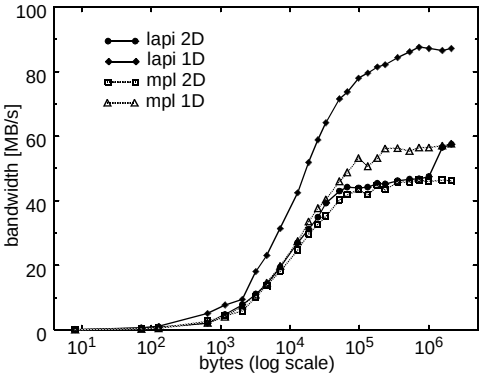

Figure 4: Performance of GA get under LAPI and MPL

sages. For larger 2D requests (0.5MB in the figure), GA switches to *LAPI_Put* protocol to send individual columns of a 2-D patch but their size is not large enough to exploit the available network bandwidth. For very large square 2-D patches (18MB) the asymptotic bandwidth of *LAPI_Put* can be achieved. The MPL implementation of GA performs identically for the 1-D and 2-D requests. The extra memory copy on the sender side cannot be avoided even for 1-D requests because of the MPL progress rules (in-order message delivery) that prevent separation of the GA request header and the data into separate messages. Since the GA get operation is blocking, the measured performance of LAPI and MPL implementations is easier to explain than for the GA *put*. Figure 4 shows that LAPI outperforms MPL for all the cases. Both MPL and LAPI versions perform better for 1-D than 2-D requests. The LAPI version uses *LAPI_Get* operation directly and avoid two memory copies for entire message range and MPL implementation is able to avoid one memory copy. With 2-D requests, the implementation switches to *LAPI_Get* protocol for request size approx. 0.5 MB similarly to the *put* operation.

To date several GA applications from the electronic structure computation and molecular dynamics area have been used under LAPI. The particular algorithms include self-consistent field (SCF), density functional theory (DFT), second-order Moller-Plesset (MP-2) and multi-reference configuration. The performance improvement over MPL-versions vary from 10 to 50% depending on the problem size, ratio of communication and calculations, and physical properties of the problems. The most performance improvement can be obtained in codes that mostly rely on 1-D array communication since it allows to avoid redundant memory copies in the AM-based protocols.

6 Concluding Remarks

LAPI is a new low-level communication library on the IBM RS/6000 SP system. It provides one-sided communication capabilities and, as we just demonstrated, it delivers excellent performance competitive with that of the MPI. The Active Message infrastructure chosen for LAPI makes it both flexible and extensible. The Global Array library is a higher-level shared-memory communication library implemented on top of LAPI and is currently used by many applications that together contain millions of lines of source code. We presented our experience in optimizing performance of GA on top of LAPI as an example to illustrate how features of LAPI can be exploited by the applications. Both performance of the GA library and its applications improved considerably as compared to the older implementation based on the MPL message-passing library. There are several ways in which LAPI architecture and implementation can be enhanced, for example:1) Providing a non-contiguous interface to *LAPI_Put* and *LAPI_Get* to help applications like GA which require non-contiguous data transfer by removing the overhead associated with multiple requests or the copy overhead in the AM-based implementations. 2) Providing multiple completion handler and multiple message-passing threads which will be important for SMP nodes. 3) Extending LAPI for use by kernel subsystems by avoiding copies into the DMA buffer for improved performance.

Acknowledgments

This research was partially performed at and supported by the William R. Wiley Environmental Molecular Sciences Laboratory at the Pacific Northwest National Laboratory. The EMSL is a national scientific user facility funded by the Office of Biological and Environmental Research in the U.S. Department of Energy. PNNL is operated by Battelle Memorial Institute for the U.S. Department of Energy under Contract DE-AC06-76RLO 1830.

References

[1] T.Agerwala, J.L.Martin, J.H.Mirza, D.C.Sadler, D.M.Dias, and M.Snir. SP-2 system architecture. *IBM Systems J.*, 34(2):152–184, 1995.
[2] M. P. I. Forum. MPI: A message-passing interface standard. Technical Report CS-94-230, Computer Science Dept., University of Tennessee, Knoxville, TN, April 1994.
[3] B. K. Grant and A. Skjellum. The PVM systems: An in-depth analysis and documenting study: Technical Report UCRL-JC-112016, Lawrence Livermore National Laboratory, August 1992.
[4] T. von Eicken. *Active Messages: an Efficient Communication Architecture for Multiprocessors*. PhD thesis, University of California at Berkeley, November 1993.
[5] J. Nieplocha, R. Harrison, and R. Littlefield. Global Arrays: A portable 'shared-memory" programming model for distributed memory computers. In *Proc. Supercomputing 1994*, pages 340–349. IEEE Computer Society Press, 1994.
[6] J. Nieplocha, R. Harrison, and R. Littlefield. Global Arrays: A non-uniform memory access programming model for high-performance computers. *Journal of Supercomputing*, 10:197–220, 1996.
[7] IBM Corporation. *IBM Parallel System Support Programs for AIX Administration Guide*, rel. 2.3, document GC23-3897-03, 1997.
[8] MPI. Forum. MPI-2: Extension to message passing interface, U. Tennessee, July 18, 1997.
[9] D. Culler, K. Keeton, C. Krumbein, L. T. Liu, A. Mainwaring, R. Martin, S. Rodrigues, K. Wright, and C. Yoshikawa. Generic active message interface specification. Technical report, University of California at Berkeley, November 1994.
[10] V. Karamcheti and A. Chien. Active messages on the Cray T3D. Technical report, University of Illinois, 1995.
[11] B. Bershad, M. Zekauskas, and W. Sawdon. The Midway distributed shared memory system. In *Proc. '93 CompCon Conference*, 1993.
[12] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.
[13] H. Bal, M. Kaashoek, and A. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Trans. Software Eng.*, 18(3):190–205, 1992.
[14] C.Amza, A.Cox, S.Dwarkadas, P.Keleher, H.Lu, R.Rajamony, W.Yu, and W.Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, 1996.
[15] J. Nieplocha, R. Harrison, and I. Foster. Explicit management of memory hierarchy. In L. Grandinetti, J. Kowalik, and M. Vajtersic, editors, *Advances in High Performance Computing*, Kluwer, 1997.
[16] H. Dachsel, H. Lischka, R. Shepard, J. Nieplocha, and R. J. Harrison. A massively parallel multireference configuration interaction program - the parallel COLUMBUS program. *J. Chem. Phys*, 18:430, 1997.
[17] A. T. Wong and R. J. Harrison. Approaches to large-scale parallel self-consistent field calculations. *J. Comp. Chem.*, 16:1291, 1995.
[18] K. Langendoen, J. Romein, R. Bhoedjang, and H. Bal. Integrating polling, interrupts and thread management. In *Proc. Frontiers 96*, 1996.
[19] C.-C. Chang, G. Czajkowski, C. Hawblitzel, and T. von Eicken. Low-latency communication on the IBM RISC System/6000 SP. In *ACM/IEEE Supercomputing '96*, November 1996.
[20] R. Riesen, A. B. Maccabe, and S. R. Wheat. Active messages versus explicit message passing under SUNMOS. In *Proceedings of the Intel Supercomputer Users' Group.*, pages 297–303, 1994.
[21] S. Pakin, M. Lauria, and A. Chien. High performance messaging on workstations: Illinois Fast Messages (FM) for Myrinet. In *Proc. Supercomputing'95*, 1995.
[22] A. Chien, S. Pakin, M. Lauria, M. Buchanan, K. Hane, L. Giannini, and J. Prusakova. High Performance Virtual Machines HPVM: Clusters with supercomputing APIs and performance. In *Proc. Eighth SIAM Conference on Parallel Processing for Scientific Computing*, 1997.