Network Endpoints for Clusters of SMPs

Gabriel Tanase∗, Gheorghe Almási∗, Hanhong Xue† and Charles Archer†
∗IBM T. J. Watson Research Center
Email: {igtanase, gheorghe}@us.ibm.com
†IBM Systems and Technology Group
Email: {hanhong, archerc}@us.ibm.com

Abstract—Modern large scale parallel machines feature an increasingly deep hierarchy of interconnections. Individual processing cores employ simultaneous multithreading (SMT) to better exploit functional units; multiple coherent processors are collocated in a node to better exploit links to cache, memory and network (SMP); and multiple nodes are interconnected by specialized low latency/high speed networks. Current trends indicate ever wider SMP nodes in the future. To service these nodes, modern high performance network devices (including Infiniband and all of IBM’s recent offerings) offer the ability to sub-divide the network devices’ resources among the processing threads. System software, however, lags in exploiting these capabilities, leaving users of e.g., MPI[14], UPC[19] in a bind, requiring complex and fragile workarounds in user programs.

In this paper we discuss our implementation of endpoints, the software paradigm central to the IBM PAMI messaging library [3]. A PAMI endpoint is an expression in software of a slice of the network device. System software can service endpoints without serializing the many threads on an SMP by forcing them through a critical section. In the paper we describe the basic guarantees offered by PAMI to the programmer, and how these can be used to enable efficient implementations of high level libraries and programming languages like UPC. We evaluate the efficiency of our implementation on a novel P7IH system with up to 4096 cores, running microbenchmarks designed to find performance deficiencies in the endpoints implementation of both point-to-point and collective functions.

I. INTRODUCTION

Most modern parallel machines are organized as clusters of Shared Memory Nodes (SMP) connected by a low latency/high bandwidth network with specialized accelerators for point to point and collective communication. An SMP in turn contains multiple processors, cores and hardware threads. IBM’s P7IH ([4], [5], [15]) compute node (“octant”) is generally configured to run up to 128 hardware threads in a single O/S image.

The Message Passing Interface (MPI) is the traditional way to program clusters of SMPs. Although the MPI standard does not decree that MPI tasks have their own address spaces, this is generally the approach taken by today’s implementations, and indeed expected by programmers. On a cluster of SMPs this means that every hardware thread gets to run its own O/S process, forcing a one-to-one mapping between MPI tasks, O/S processes and hardware threads.

Users of clusters of SMPs have been quick to realize that this one-to-one mapping of tasks to hardware threads has major disadvantages. Specifically:

- MPI tasks co-located on the same SMP talk to each other through MPI instead of writing into each others’ memory directly. Custom MPI implementations mitigate this problem to some extent using for example System V shared memory regions, but the overhead of MPI message matching etc. still exists.
- Running many O/S processes means giving up control to the O/S scheduler about resource allocation (processing, network, memory and disk resources). This can cause a number of performance issues, from uneven file I/O performance to O/S jitter.
- While MPI applications are carefully written to scale to millions of processes, the underlying system (file I/O, job management software) often has trouble coping with extreme scaling requirements. In practice, this means long startup and shutdown times, sub-par I/O performance and a general increase in the brittleness of the system.

These problems are not limited to MPI. Alternative programming models, such as UPC and other partitioned global address space (PGAS) languages suffer from the same problems. PGAS languages in particular are amenable to performance increase when tasks are co-located into the same process and share memory directly.

To work around the scalability issue, many MPI applications today are written by mixing MPI with a shared memory paradigm, i.e. OpenMP or POSIX threads. These are used within each MPI task, reducing the total number of tasks and therefore mitigating scalability problems: the overhead of process management and file I/O can be reduced by an amount proportional to the number of cores per SMP. There remains one issue, however. Programmers of OpenMP+MPI code often invoke MPI primitives within the parallel OpenMP section. Newer versions of MPI allow this, but at the cost of major performance degradation due to the fact that the threads in the parallel section all use the same network resource. We illustrate this problem in the evaluation section of this paper.

In this paper we describe an approach that allows users to explicitly manage network resources in a hybrid (multi-threaded + networked) environment. We call these network resources endpoints: they represent the smallest individually addressable network resource that can be used for communication. Each O/S process in a parallel job is assigned a number of network endpoints, which then can be assigned by the user to individual threads. Based on our experience it is preferable although not necessary - a one-to-one mapping between endpoints and threads, so as to make locking superfluous. This approach allows the user to explicitly manage network
resources, thus avoiding network contention and ensuring scalability.

Our paper’s specific contributions are as follows. We describe a novel endpoint implementation completely integrated into the PAMI low level messaging library. This is different than previous work on endpoints where the proposed interfaces are discussed in the context of MPI. Our description includes the detailed semantics of endpoints and the proposed PAMI API. We also describe how the endpoint-based library is transparently used by the xIUPC [8] compiler. Finally, to support our case, we present a detailed experimental evaluation of the endpoint features of the PAMI library. We examine the performance of endpoint-based point-to-point and collective operations, operating PAMI directly as well as through UPC.

II. RELATED WORK

The concept of endpoints has been introduced in [10], [17]. The discussion there is mainly targeted towards MPI implementations as a mechanism to exploit multithreading within an MPI application. In [10] the authors describe the semantics of endpoints, specifying the message delivery between endpoints and endpoint based communicators. Our work introduces the endpoint as a basic low-level concept that can be used to implement higher level libraries or parallel programming languages. In Section IV we show how UPC language exploits the endpoints provided by PAMI for efficiently supporting the shared memory mode of execution (see Section IV). Also, [10], [17] is mainly a proposal; our paper describes a finished and functional implementation of endpoints, validated by experimental results.

While there is little related work on the endpoints as introduced in [10], a large body of work exists on the exploitation of shared memory nodes within MPI or other parallel programming languages like UPC [7], [9], [13]. Many of these papers focus on optimizing various communication primitives by means of a shared memory region. Optimizations are possible for both point to point messages and collectives. Very good performance can often be attained using this approach [18]. This, of course, still leaves the system with a very large number of processes running, which can cause bottlenecks. With endpoint support we envision that only one process per shared memory node will be needed.

Other messaging libraries provide concepts similar to the endpoints we discuss in this paper. The Common Communication Interface (CCI)[6] endpoint is very similar with the PAMI endpoint. A major difference is the completion notification using call backs in PAMI versus events in CCI. In a related paper [18] we discuss the importance of callbacks for non blocking primitives (P2P, collective) composition. Another important difference relative to the results reported in [6] is that this paper focuses especially on a multi-threaded deployment while the CCI paper only present result for the distributed memory case. Portals [11] is another messaging interface that supports asynchronous communication and user notifications using an event mechanism. Again this is different than PAMI’s call back mechanism. Additionally portals as in version 3.3 don’t support collective operations.

Infiniband[1] and Myricom provide lower level API-s. PAMI for example builds on top of Infiniband APIs for x86 clusters and the PAMI implementation in this case provides efficient support for the endpoint concept described in this paper.

III. PROGRAMMING WITH ENDPOINTS: CONCEPTUAL FRAMEWORK

In this section we introduce the concepts for understanding the PAMI endpoints based programming. Consider a parallel application consisting of a set of $P$ processes, each uniquely identified by a task ID in the range $0..P − 1$. Our central addition to this model is that the network hardware exposes multiple distinctly addressable and independently operated components in the same process. We call these contexts, and we number them in a linear fashion (just like task IDs). For the remainder of this paper we assume (for simplicity’s sake) that each task has access to the same number of network contexts $C$.

Endpoints. Our execution model thus has a two-dimensional task addressing scheme: a pair of numbers in the form $\{taskID, ctxt\} \in [0..P − 1] \times [0..C − 1]$. This pair of numbers is called an endpoint. The higher level programming model can choose to linearize this addressing scheme (the IBM UPC implementation chooses to do it), hide the existence of contexts from the programmer (as the Blue Gene/Q MPI implementation does), or expose two-dimensional addressing directly to the programmer.

In our terminology we use the context as a piece of network hardware, and the endpoint as a globally recognizable name for it. Thus, at program startup time we initialize contexts; but we send messages to endpoints.

Addressability. In order for our system to be functional, every context has to be capable of sending messages to any other context in the same job. One might argue that the ability to send a message to an arbitrary context of any node should be sufficient. However, this would imply that messages arrived on the destination node would have to be forwarded to the destination by software, raising the specter of locked delivery FIFOs contended by threads - in short, the very problem we are trying to avoid by introducing contexts in the first place.

Reentrancy of contexts. Freedom from locks comes with limitations. We do not require individual contexts to be re-entrant. More precisely, we allow two different threads to service two different contexts without a provision for locking, but the same is not true for the same context. Same-context reentrancy may be provided by the implementation, but is not required conceptually.

Progress guarantees. Our execution model requires all contexts in a parallel program to be serviced by at least one thread. Since contexts are usually associated with finite network resources (such as FIFO memory and so on), if any one context is not serviced, incoming messages can accumulate.
and cause the rest of the system to stop making forward progress. Any context, once created, has to be serviced.

**Endpoints vs. threads.** Other than the basic progress guarantee (all contexts must be serviced) we do not require any particular thread-to-context mapping. At a minimum, all existing contexts in a process could be serviced by just one thread; on the other hand, multiple threads can service a context with proper mutual exclusion (although that rather defeats the purpose of endpoints).

### A. Interface

IBM’s Parallel Active Message Interface (PAMI) library is the successor of IBM’s previous messaging libraries, LAPI [2] and DCMF [12]. One of PAMI’s primary distinguishing features is that endpoints are integrated into its core design as opposed to tacked onto it. It is not our goal to present the PAMI API in its entirety in this section; instead we focus on how endpoints integrate into the general design. For the complete specification of PAMI we recommend that the reader consult the manual [3].

In broad lines, PAMI is a non-blocking one-sided messaging library. The primary functions of PAMI are active messages, remote put and get operations, synchronization and atomic functions, and a set of collective communication primitives. Every PAMI function call is non-blocking and takes a completion handler argument. PAMI functions with non-local semantics have both local and remote completion handlers (i.e. functions that run when the PAMI function is complete locally vs. on the remote node). Next we introduce the relevant functionality of the PAMI library for endpoint support.

**PAMI clients and contexts:** After initializing the messaging library itself - creating a messaging client object - one must specifically create the contexts in use by the client. This is where the users asks for a number of network resources (e.g., windows), and this functionality is provided by the `PAMI_Context_createv` call. Conversely, at the end of the program contexts must be destroyed, which is what `PAMI_Context_destroyv` does.

**Creating endpoints:** As discussed before, endpoints are no more than a stylized pair of numbers, pointing to a context on a (potentially remote) task. PAMI provides a utility function called `PAMI_Endpoint_create` that builds an endpoint based on a target task ID/target context ID pair. This is an extremely low overhead function, allowing users to create endpoints on demand rather than store them for reuse.

**Sending messages and data:** Most PAMI functions differ slightly from the usual expected signature in two ways. First, every PAMI function needs to specify the actual context it is being submitted to. This needs to be explicit since the context is not tied in any way to the identity of the submitting thread; PAMI has no knowledge of what threads the user may or may not have started. Second, the destination of a PAMI communication is specified by an endpoint as opposed to just a task ID.

**Polling the network:** Most messaging library has a function that polls the network, making explicit progress, running active message handlers and so on. So does PAMI, but with one difference: network progress is made on a per-context basis. Since all created contexts are required to make progress, this means that the `PAMI_Context_advance` function needs to be called for every context - although, again, PAMI does not specify what user threads do this.

**Event handlers** are function pointers submitted by the programmer to be run when an asynchronous event happens (e.g. when a PAMI message arrives or is complete). Since PAMI events are specific to the contexts on which they happen, so are event handlers: the completion of a send on context 3 will never be announced on any other context than 3. The first parameter of every event handler is the context in which it is running.

**Asynchronous progress:** The PAMI library provides both asynchronous and synchronous progress. In synchronous mode the user advances the network by explicitly invoking `PAMI_Advance`. Asynchronous mode will use an extra thread to call PAMI event handlers on several contexts. In this situation, however, locking of contexts may be required, especially if event handlers call back into PAMI.

**Collective communication:** The PAMI collective routines are very close to (and inspired by) MPI-3 non-blocking collectives. The only major difference is that PAMI geometries - the equivalent of MPI communicators - are built not as collections of tasks, but in terms of collections of endpoints. All communicator arithmetic - familiar to MPI programmers - is in fact expressed in terms of endpoints. Participants of a collective routine are contexts, not tasks; thus, a PAMI collective can have multiple participants in the same task.

### B. PAMI endpoints: an example

Figure 1 shows a simple PAMI program written in C like pseudo code that illustrates endpoint programming: specifically, the startup phase, the method of sending a message and how to write a callback. The main program (Line 3) creates the PAMI client and a number of contexts. Our simple example creates a thread to handle each context (lines 11-13). Each thread receives the context ID as an argument (not shown for brevity). When the context handlers are all done, the threads are joined (not shown) and the main thread destroys the contexts and the client (Lines 15-16).

Lines 19-33 shows the multithreaded code that sends an active message using the `PAMI_Send` function. In line 22 we declare a data structure defined by the PAMI specification, and we fill in its members: first, a pre-registered dispatch ID that identifies a function to run at the destination (registered on Line 8 in the main program). Next, in lines 24 and 25 we specify the data buffer to send. Line 26 specifies a function to execute when the send is locally complete on the sender (i.e., the send buffers can be reused). Note that the user also has the option of specifying a remote completion callback (similar to specifying a local completion, but not shown in the example). The destination of the send object is an endpoint. This endpoint is created on the fly in Line 28 by specifying
```c
#define DISPATCH_ID 111

void main() { 
  /* create PAMI client and 'NC' contexts */
PAMI_Client_create ("my_client", client,);
PAMI_Context_createv(client,...,contexts,NC);
  /* register callbacks for an active message */
PAMI_Dispatch_set (... , DISPATCH_ID, HeaderHandler);
  /* create N threads, one for each context */
  /* and run those threads. */
  for (i=0;i<NC;i++) {
    pthread_create...,thread_main,...);
  }
  /* end of program: destroy contexts, client */
  PAMI_Context_destroyv(contexts, num_contexts);
PAMI_Client_destroy(client);
}

****** PROGRAM FOR EACH THREAD **************
void thread_main(td) {
  /* create and populate a send object */
pami_send_t s;
s.send.data.iov_base = /* data we send */
s.send.data.iov_len = /* how much data */
psend.events.local_fn=send_cb_done;
  /* build an endpoint */
PAMI_Endpoint_create (client,
  dest_task,
  dest_context,
  &s.dest);
PAMI_Send (myContext, &s);"/
}

****** PROGRAM FOR EACH THREAD **************
void HeaderHandler {
  /*IN: origin context */
pami_context_t context,...
  /*IN: endpoint that originated the send*/
pami_endpoint_t origin,
  /*OUT: receive message structure */
pami_recv_t * recv) {
  /* This handler has two roles:
 1) prepare a data structure to store incoming data
 2) register a function to run when incoming data transfer is complete */
  recv->addr = <some buffer to store data>;
  recv->local_fn = <some function pointer>;
}

Fig. 1. A simple endpoint based PAMI program

the destination's task ID and context. Once all the parameters are ready, the PAMI_Send in Line 32 delivers the message.

Lines 35 to 48 show a user-defined PAMI callback mechanism to handle the first phase of the reception of a message. PAMI has a two-phase message reception mechanism, consisting of two callbacks. The first callback, called the header handler (shown here) notifies the user of the pending reception of a message and provides for an opportunity to have a message buffer reserved/allocated for the incoming data (Line 46). The user also has to provide the header handler with the address of a second function to call - the completion handler (shown in Line 47). The completion handler is called when the data transfer is finished.

C. Endpoint Geometries

The MPI standard allows tasks to be arbitrarily grouped in collections called communicators. The set of all MPI processes created at application startup is referred as MPI_COMM_WORLD; other communicators can be specified as a collection of tasks. Every collective communication primitive is associated with a communicator; all members of the communicator are required to participate.

PAMI geometries are very similar in nature to MPI communicators, except PAMI allows the creation of geometries that are collections of endpoints. Driven in part by requirements of backwards compatibility, there are two flavors of geometry creation primitives that we discuss next.

Task based geometry creation: For applications who do not want to use endpoints. These primitives create geometries with a single participant in each task. A natural question would be “which of the task’s contexts will participate in a geometry?” PAMI resolves this question by allowing any context to represent a task in a geometry, but requiring that the representative’s context number should be the same in every task.

Task based geometry creation can happen in any thread of a multithreaded executable; it is not required that the owner of a context call the function. Note, however, that PAMI does not guarantee reentrancy of any PAMI function except when different contexts are targeted. Thus, task based geometry creation should be done from a master or any other dedicated thread.

Endpoint based geometry creation: The PAMI library also provides equivalent functions to instantiate endpoint based geometries. The current API has a slight inconsistency in that only one invocation of the geometry creation function is required on each task, instead of each participating context having to invoke it. This is due to backwards compatibility issues, and may be fixed in future versions of PAMI.

Once created, geometries are used to invoke collective operations. A collective on an endpoint based geometry has to be invoked by every participating context. One should not forget, however, that the basic tenet of endpoint based programming - every context needs to make progress - has to be respected. A barrier on a sub-geometry may appear to hang if an unrelated context is not making progress on the network.

D. Hardware and system software support for endpoints

Most modern network interconnects designed to operate on clusters of SMP machines - the IBM HFI [4] used by PERCS, the Blue Gene networks, Infiniband™ - provide multiple independent network windows on each physical device. In a traditional setting network windows allow multiplexing the network device across multiple processes running on the same SMP node. Typically the operating system partitions up the memory or I/O space provided by the device and maps individual network windows to each process that requests one.

All that is really required, then, to implement our concept of a network endpoint is a slight change in the allocation policy: allowing multiple hardware windows to be mapped into the
same process. On P7IH each shared memory node has at least one network device and each network device provides a number of hardware windows. The PAMI context creation call (PAMI_Context_createv) amounts to mapping a number of network windows into user space and returning their addresses to the user. The hardware guarantees that the network windows can be accessed independently by the threads of the application; but of course if two threads want access to the same window (context) they need to synchronize.

IV. PAMI ENDPOINTS IN THE XL UPC RUNTIME

In this section we describe the way in which the IBM xlUPC compiler and runtime use PAMI endpoints to increase the efficiency of shared memory execution. UPC programs are executed in an SPMD execution model similar to MPI. The UPC construct corresponding to an MPI task is a “thread”, not to be confused with a POSIX thread - in most UPC implementations UPC threads are actually mapped into processes. With IBM’s xlUPC compiler we also provide an additional shared memory mode of execution where a slightly different approach is used. UPC threads map into POSIX threads. UPC shared memory accesses devolve into loads and stores when executed on the same SMP node. UPC shared memory is managed by libc calls like malloc() and free().

When using the shared memory mode of execution, an xlUPC program starts up with one process on each SMP node. This process starts the POSIX threads that will act as UPC threads, as well as creates the necessary number of PAMI contexts. We assign threads to contexts on a one-to-one basis.

The xlUPC compiler is standards compliant: contexts and endpoints are invisible to the UPC programmer. Launching an xlUPC program behaves slightly differently, however. The job launcher (POE) is instructed to launch only one process per node, and the UPC runtime has to figure out the number of threads to be launched by dividing the desired number of UPC threads (specified, as per the standard, in the UPC_NTHREADS environment variable) with the number of processes running. In Section V-D we analyze the performance of UPC collectives and a simple application under the shared memory model of execution.

V. PERFORMANCE EVALUATION

In this section we evaluate the performance of the endpoint implementation currently supported by PAMI. As a reminder to the reader, we are not arguing that endpoint based systems provide inherently better performance. Instead, our goal in this section is to show that endpoints can provide similar performance when used in a nested multithreaded code as in a non multi threaded code. Provided that we can support an efficient endpoint implementation then users interested in their expressivity can employ them without worrying about performance loss.

In Section III-D we have shown that only minimal hardware support is necessary to enable endpoints. In our experience with a P7IH system, software and not hardware, is the major contributing factor to loss of performance in the messaging library. A fairly trivial example of such an inefficiency may be simply an overabundance of malloc() calls, which itself becomes a bottleneck. Removing such sources of inefficiency is difficult and painstaking work, and this is what we are evaluating in this Section.

Section V-A describes the system on which we run our test. Section V-B looks at several point-to-point messaging scenarios using targeted microbenchmarks. Section V-C explores collective performance. Section V-D shows the performance of simple UPC programs when running with endpoint support.

A. System setup

We used a P7IH system for our experiments. The basic compute node of the P7IH consists of four Power7 [16] CPUs and a HUB chip, all managed by a single OS instance. The HUB provides the connectivity between the four P7 Chips participating in the cache coherence protocol. Additionally the HUB acts as a switch supporting communication with other HUBs in the system, doing away with the requirement for any network switches. The resulting system is close to fully connected, sporting an extremely high cross-section.

Each Power7 CPU has 8 cores running at 3.8 GHz, and each core can run four hardware threads (SMT). This results in each O/S image running 4CPUs × 8cores × 4SMT = 128 hardware threads. The node is equipped with 256 GB or 512 GB of RAM, depending on the setup.

We should mention that P7IH systems are further organized into drawers of 8 nodes (“octants”) each, and supernodes consisting of 4 drawers. A drawer consists of 8 × 4CPUs × 8cores = 256 cores, each 4 way SMT; a supernode has 1024 cores. Additional hardware description and key parameters are included elsewhere [16], [5], [15].

B. Point to point endpoint performance

Benchmark setup: Our goal was to evaluate the performance of endpoint mode relative to the more traditional one-process-per-task mode. All plots included in this section compare performance in the following two scenarios: A first one referred as the distributed (Dist) case where we have one endpoint per task (e.g., process) and a second one (MultiEP) where we have multiple endpoints per task and within each task, multiple threads take exclusive ownership of particular endpoints and place communication requests to them.

```c
int main(int argc, char **argv) {
    int i=0;i<num_iterations;++i) {
        if (my_id< P/2) {
            flag = 1;
            PAMI_send (myContext, &psend);
            /* the flag will be set to zero using a callback invoked when the message is received remotely */
            while (flag == 1) PAMI Advance (myContext);
        } /* stop timer and report average per iteration */
    }
    return 0;
}
```
Pairwise split processes: Our first experiment splits the $P$ available endpoints into two sets $L = [0, P/2)$ and $R = [P/2, P)$, with endpoints $EP_i \in L$ sending messages to $EP_i + P/2 \in R$. We measure roundtrip latency and per-thread bandwidth as a function of data buffer size. Figure 2 shows the code executed by each thread in our experiment. We measure roundtrip latency by waiting for the remote completion callback after the send. The purpose of the experiment is to highlight any performance degradation due to imperfect implementation (e.g. contention of endpoints to network resources).

Latency comparison: Figure 3(a) compares roundtrip latency $\mu s$ as a function of the message size. The distributed base case (Dist) is run on two nodes with 32 tasks per node; the multiple endpoints (MultiEP) case is run on two nodes, with one task per node and 32 threads per task, with one endpoint per thread.

The network smallest data packet sent by a P7IH is a 128 byte flit, causing a latency “plateau” on message sizes up to 128 bytes, distinctly visible in the Figure 3(a). The Figure 3(a) also shows a small performance difference; 2.1 $\mu s$ round trip for Dist vs. 2.3 $\mu s$ for MultiEP. The difference disappears with larger message sizes; what we see, then, is likely an overhead problem in the multi-endpoint implementation of PAMI.

Bandwidth comparison: Figure 3(b), compares the achievable bandwidth in the same setup. The two nodes in the setup are connected by a single link, shared by 32 pairs of communicating threads. The Figure shows 0.55 GBytes/s/link peak bandwidth, corresponding to about 80% of $\frac{1}{2^2}$ of the total available link bandwidth (20 GBytes/s).

Multiple nodes: Figure 4(a) extends our experiment to up to a drawer (256 cores, one half of which send messages to the other half). As message size increases, we notice some performance degradation with increasing number of participating cores (plainly visible on the rightmost group of bars of the Figure, with 8KByte message sizes). The MultiEP runs all have slightly lower performance than the Dist runs; the difference is in the 2% to 8% range. We attribute this difference, again, to higher messaging overhead in the runtime. However, the overhead does not increase with the number of participating cores.

Bipartite alltoall pattern: To increase the amount of stress placed on the networking subsystem relative to the pairwise send pattern, we extended our measurements to an alltoall like pattern: Figure 4(b) compares MultiEP vs. Dist message send overheads when half the threads in the system send messages to each of the threads in the other half. This results in $\frac{P^2}{2^2}$ messages sent, and puts pressure on both the sender side (to get
those messages out) and the receiver side (to handle incoming messages quickly). Note that in this experiment we measure message send overheads, not roundtrip latency. The shortest message send overhead seems to be in the 1 µs range. There is no visible performance degradation for MultiEP mode, which indicates that if there is a problem with the PAMI implementation, it is not on the sender side.

**Conclusion:** We put the endpoint-based PAMI library under pressure we concluded that while minor performance degradation exists, there are no major scalability issues with endpoints.

**C. Endpoint based collective performance**

In this section we compare the performance of a set of PAMI non blocking collectives running in Dist vs. MultiEP mode. These are collectives written using point-to-point messaging primitives; we expect equivalent performance in the two modes. In Dist mode, there is a single endpoint per process; in MultiEP mode we create up to 32 endpoints per process (running in SMT-1 mode). We maintain a consistent 1 thread/endpoint mapping. We considered the following collectives: Barrier, Broadcast and AlltoAll. Details about the non blocking implementations of these algorithms can be found in [18].

**Barrier:** Figure 5(a) compares the scaling of Barrier execution. Barrier is implemented as a recursive doubling type algorithm with a \( \log(P) \) number of distinct phases; a lot of the initial phases involve communication among processes/endpoints located on the same node. Thus MultiEP has a natural advantage here, and this shows up on the left side of the graph. As we scale to larger systems the advantage is supposed to disappear, since communication is no longer mostly local. However, we did not expect the dramatic performance loss of MultiEP visible at 2K and 4K threads. The root cause of this is under investigation.

**Broadcast:** Figure 5(b) compares broadcast latencies for 8 byte messages in Dist vs. MultiEP modes. The MultiEP implementation again has a natural advantage - the binomial tree algorithm used in broadcast has good locality in the early stages; so MultiEP is 7% faster on average.

**Conclusion:** We put the endpoint-based PAMI library under pressure we concluded that while minor performance degradation exists, there are no major scalability issues with endpoints.

![Fig. 5. PAMI point to point barrier and broadcast collective. Execution time in microseconds for various number of hardware threads](image.png)

**Fig. 6. Sobel performance for multiple endpoints and distributed case. Results in megapixels per second for two input image sizes: 1k×1k and 8k×8k.**

<table>
<thead>
<tr>
<th>P</th>
<th>MultiEP(1k)</th>
<th>Dist(1k)</th>
<th>MultiEP(8k)</th>
<th>Dist(8k)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>444.76</td>
<td>167.82</td>
<td>461.13</td>
<td>379.69</td>
</tr>
<tr>
<td>64</td>
<td>204.55</td>
<td>176.52</td>
<td>642.94</td>
<td>605.48</td>
</tr>
<tr>
<td>128</td>
<td>952.48</td>
<td>902.20</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**D. Applications written in UPC**

In Section IV we have shown that UPC can exploit the multiple endpoints feature to provide better shared memory support. In this section we look at the performance of a simple Sobel benchmark discussing the performance when running distributed or shared memory mode. The `UPC_NTHREADS` environment variable specify to the UPC runtime the total number of threads to be used, and this in conjunction with the total number of processes to use allows users to specify the number of internal threads per process. In the experiments of this section when using `P` cores, the distributed mode employs one process per UPC thread per core while the shared memory mode employs 32 threads per process and one process per shared memory node.

**UPC implementation of Sobel microbenchmark:** We evaluate the performance of a simple Sobel convolution benchmark implemented in UPC and we discuss the performance benefits of using the MultiEP setup. The Sobel benchmark computes an approximation of the gradient of the image intensity function, performing a nine-point stencil operation. An image is represented as a two dimensional shared array were the first dimension is block distributed across all computational threads.

The UPC shared memory model of execution, discussed in Section IV, is greatly facilitated by the endpoint support in PAMI. In the shared memory mode of execution, the UPC compiler and runtime are aware of collocated threads and can optimize accesses. The compiler can generate simple memory loads for shared accesses that can be proven node-local at compile time; the runtime generates calls to `memcpy` instead of invoking PAMI when accesses turn out to be node-local.

In Figure 6 we include experimental results where for a
We have demonstrated, by experiments, that endpoints benefit users who want to take advantage of the locality of threads on shared memory nodes. In particular, parallel programming languages like UPC can transparently take advantage of the multiple endpoint support in PAMI. We have argued that the PAMI library is well prepared to accommodate future MPI standard changes requiring endpoints if they are approved.

fixed image size we show the performance in mega pixels per second as we increase the number of threads (cores) used. We show data for two image sizes: 1K×1K and 8K×8K. The advantage of shared memory access is most noticeable when running within a node (32 cores). For the 1K×1K case the performance is more than double relative to the distributed case (444 MPixels/s vs. 176 MPixels/s). We notice a much smaller difference in the 8K×8K case. For big input sizes the computation on local data dominates the total runtime, and the performance difference due to shared memory access disappears. Additionally the benefits diminish as we increase the number of threads. In this case, even though some of the 32 processes within a shared memory node will finish faster they are still waiting for the threads that communicate across SMPs before reporting the execution time. Hierarchical algorithms that minimize communication across SMPs can observe improved performance due to endpoints even for larger number of threads.

E. MPI and OpenMP limitations

As discussed in the introduction, we include in this section a simple experiment to quantify the limitations of current MPI implementations when mixed with threads or OpenMP. We wrote a simple OpenMP+MPI program that re-creates the unidirectional send microbenchmark we used in Section V-B. The program consists of two MPI tasks, each creating a parallel region using OpenMP and then one of the nodes sending messages to its counterpart from each of the OpenMP threads.

The table in Figure 7 shows the execution times when using one, two and four threads per task for 3 different message sizes. The problem here is twofold: (a) MPI’s message queuing mechanism has to serialize message reception order, even when this order is not required by the programmer, and (b) all OpenMP threads are attempting to use the same slice of network hardware, with MPI arbitrating the access. Predictably, access time explodes as contention grows.

Our paper shows that the tools to address this shortcoming of MPI are implementable, and IBM’s PAMI library provides them as a matter of course. The MPI standards committee has a chance to address this problem with [17].

VI. CONCLUSION

In this paper we described endpoints as a way to improve scalability of messaging libraries on clusters of SMP nodes. We have described the way in which the IBM PAMI library implements endpoints. We have shown that with a little additional work endpoints can be retrofitted onto a “standard” messaging library, without compromising efficiency and scalability.

Fig. 7. MPI and OpenMP Send latency (µs) for three different message sizes (8, 16 and 32 bytes) and three communication patterns.

### References