RC25259 (W1201-025) January 16, 2012
Computer Science

IBM Research Report

Hybrid Collective Operations on Power7 IH

Gabriel Tanase, Gheorghe Almasi, Charles Archer, Hanhong Xue
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Hybrid Collective Operations on Power7 IH

Gabriel Tanase Gheorghe Almasi Charles Archer Hanhong Xue
IBM TJ Watson Research Center, Yorktown Heights, NY
igtanase@us.ibm.com, gheorghe@us.ibm.com, archerc@us.ibm.com, hanhong@us.ibm.com

Abstract

The Power7 IH (P7IH) is one of IBM's latest generation of supercomputers. Like most modern leadership class parallel machines, it has a hierarchical organization consisting of simultaneous multithreading (SMT) within a core, multiple cores per processor, multiple processors per node (SMP), and multiple SMPs per cluster. A low latency/high bandwidth network with specialized accelerators is used to interconnect the SMP nodes. System software is tuned to exploit the hierarchical organization of the machine.

In this paper we present a novel set of collective operations that take advantage of the P7IH hardware. We discuss non blocking collective operations implemented using point to point messages, shared memory and accelerator hardware. We show how collectives can be composed to exploit the hierarchical organization of the P7IH for providing low latency, high bandwidth operations. We demonstrate the scalability of the collectives we designed by including experimental results on a P7IH system with up to 4096 cores.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Parallel Programming

General Terms Languages, Design, Performance

Keywords Hybrid, Collectives, Parallel, Programming, Languages, Libraries, Messaging

1. Introduction

The need for ever increasing computational power leads major computer vendors, national labs and academia to continuously innovate in the area of high performance computer design and large scale parallel programming. The Defense Advanced Research Projects Agency (DARPA) recognized the importance of using massive computational power for accelerating the process of scientific discovery and in 2001 it launched the High-Productivity Computing Systems (HPCSs) initiative [12]. In response to this IBM created in 2002 the Productive, Easy-to-use, Reliable Computing System (PERCS) program. Among the notable outcomes of the project is the novel Power7-IH supercomputer architecture [6, 23, 24], a powerful programming environment consisting of the novel X10 language [7, 10] and optimized compilers and runtime systems for existing parallel programming languages like Unified Parallel C (UPC) [5, 28] and CoArray Fortran [11].

In this paper we describe a design for providing scalable collective communication primitives on the P7IH system. We argue that system software must follow the hierarchical organization of the hardware; efficient parallel algorithms must employ a compositional approach. Thus in our design, algorithms optimized for various levels of the hierarchy are *chained together* to obtain scalable parallel operations.

Another important aspect of a parallel collective operation that we argue for in this paper is the ability to not block threads upon invocation but allow them to further proceed until the result (e.g., reduce, broadcast) or effect (e.g, barrier) of the invocation is required. This has been recognized [5, 17] as an important feature for designing scalable applications on large HPC systems.

We make two different contributions in this paper. First, in Section 3 we describe and analyze three different types of collectives optimized for various components of the P7IH hardware hierarchy: a novel generic design for achieving non blocking collectives using only *point to point* message exchange, *shared memory* non blocking collectives that can achieve very low latency and high bandwidth on the large SMP nodes of the P7IH and collectives that use the novel P7IH *accelerated collective unit* (CAU). We describe for the first time the performance of these collectives on the P7IH architecture.

Our second contribution is in Section 4, where we describe *a novel design for composing existing collectives* to exploit the hierarchical organization of the P7IH. We formalize the process of collective composition and evaluate the performance of a set of hybrid collectives developed in our framework, on a large P7IH system with up to 4096 processor cores.

2. Hardware and Software Overview

P7IH [23] systems employ a hierarchical design allowing highly scalable deployments of up to 512K processor cores. The basic compute node of the P7IH consists of four Power7 (P7) [24] CPUs and a HUB chip, all managed by a single OS instance. The HUB provides the network 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. There is no additional communication hardware present in the system (no switches). As shown in Figure 1, each P7 CPU has 8 cores and each core can run four hardware threads (SMT) thus leading to a 32 core, 128 threads, and up to 512GB memory compute *nodes*. A large P7IH system is further up organized in *drawers* consisting of 8 compute nodes (256 cores) connected in an all2all fashion (see Figure 1), *supernodes* consisting of four drawers (1024 cores) and full system consisting of up to 512 super nodes (524288 cores). Additional hardware description and key parameters are included elsewhere [6, 19, 23, 24].

In this paper we focus on PAMI [2], the IBM parallel messaging library. PAMI is a component of the IBM core system software stack for parallel programming that includes products like PESSL [3], the engineering and scientific subroutine library),

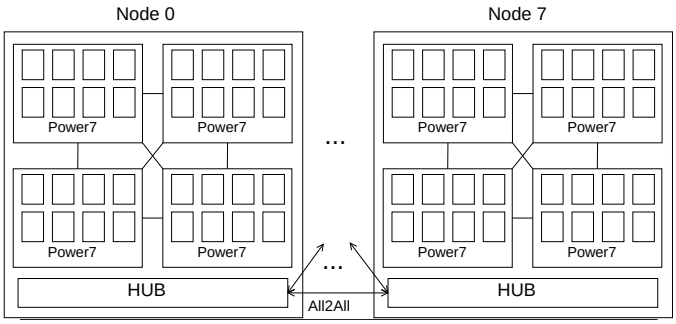


Figure 1. P7IH drawer. The Power7 is an 8 core, 32 threads CPU. Four CPUs and one HUB makes a shared memory compute node (SMP node). Eight compute nodes are connected in an all2all fashion in a drawer.

```
1  void cb_done (void * ctxt,
2                void * clientdata,
3                pami_result_t err) {
4   int * flag = (int *) clientdata;
5   *flag = 1;
6  }
7  ...
8  broadcast.cb_done = cb_done;
9  broadcast.algorithm = algo_option;
10 broadcast.cmd.xfer_broadcast.buf=buf;
11 broadcast.cmd.xfer_broadcast.root=root_ep;
12 flag = 0;
13 result=PAMI_Collective(context,broadcast);
14 assert(result == PAMI_SUCCESS);
15  ...
16 while (flag)
17   PAMI_Context_advance (context, 1);
```

Figure 2. PAMI non blocking invocation

GPFS the general parallel file system, the MPI [22] library as well as the UPC[28] and X10[10] compilers.

The current release of PAMI unifies the functionality of its predecessors, LAPI[1] and DCMF[20] respectively. Thus PAMI provides support for *non blocking* point to point active messages and collectives. The library is not targeted at end users, but is rather intended for system programmers; developers of runtime systems for programing languages and paradigms like MPI, OpenMP, X10, UPC, CoArray Fortran, etc. In terms of collectives the PAMI library supports multiple algorithmic options for a given operation and higher level run time systems are left with the decision of selecting the best choice.

3. Collectives

Collectives are parallel algorithms performing certain well defined operations like barrier, reduce, broadcast, scatter, gather, prefix sums, all to all exchange, etc., and they are often used as building blocks in users applications. PAMI library provides an extensive collection of such operations tuned for different IBM architectures like BlueGene, Power, x86 clusters. In this section we discuss requirements for the PAMI collectives and the different strategies employed to satisfy these requirements.

- **Portability**: PAMI provides a set of collectives that have minimal assumptions about the underlying architecture, thus ensuring they work on a large spectrum of machines supported by IBM. It is often the case that portability comes at the cost of performance as we will see in this paper.
- **Efficiency and scalability**: collectives have to take advantage of the underlying hardware to provide the best performance for a given architecture. Tuning efforts performed on a given platform not always can be ported to a different one.
- **Non blocking**: Non blocking collectives have the potential to improve user applications as they allow for better overlap of communication and computation. Supporting this feature has important implications on the collective design as discussed next.

3.1 The PAMI Non Blocking Collectives Interface

In PAMI, both collectives and point to point operations are non blocking. This generally translates into an interface that allows users to start the operation and later on to check for its completion. Multiple solutions are proposed in the literature to achieve this functionality:

- **handles:** One simple solution starts the operation and returns a handle that the user subsequently checks for completion. Examples of libraries employing this approach are the libNBC [15] discussed more in the related work section, the STAPL library [26] where data structures have non blocking methods and others [5, 17]. The problem with this solution is one of scalability - the number of handles to check grows linearly with the number of simultaneously outstanding operations.
- **counters:** Another solution, employed by LAPI, is to increment designated counters upon completion of non blocking operations. This solution has the problem that counters have to be incremented atomically, causing a very high locking overhead.
- **callbacks:** In this case a callback function is invoked by the library upon completion. This approach is favored by PAMI because of its relative flexibility and scalability. In fact all other solutions can be trivially implemented with appropriately chosen callbacks. Within the callback function, flags can be set to mark the completion of the operation or as we will see later on in Section 4, other non blocking operations can be started to create composed operations.

Figure 2 shows a simple example of a broadcast invocation. We notice the callback (Line 1) as a standalone function where the second argument is the user data that can be modified inside the call. Lines 8-11 prepare the arguments for a broadcast operation, including the callback function, the broadcast buffer, the root, and a particular broadcast algorithm to be used. Line 12 marks a flag variable with zero. The variable subsequently will be assigned a value of one by the callback. Line 13 starts the broadcast and returns immediately. The callback will be invoked when the operation locally finishes. Lines 16, 17 show a possible way to wait for the broadcast's completion. This example is important for understanding hybrid collections described in Section 4.

3.2 Execution Model

The execution model under which a PAMI program executes is similar to MPI with the following differences. A program consist of a set of *tasks* and not threads or processes. A task can be mapped on a process or a thread and multiple tasks can be advanced by a thread or process. In the example included in Figure 2 the *context* variable is used to distinguish among various tasks of a computation. In this paper a collective operation is invoked from a set of tasks for which the collective is defined. Similar to MPI ranks or thread identifiers a task has an unique identifier and communication or data transfer happens between tasks. Another important aspect about the PAMI library is the progress model. Because collectives are non blocking it is user responsability to ensure progress is made by calling PAMI_Context_Advance from all the tasks of a computation [2].

```
1  struct p2p_state {
2   int    dest;
3   void* send_buf;
4   int    source;
5   void* recv_buf;
6   ...
7  }
8  class p2p_collective{
9   p2p_state* phases;
10  int phase;
11  int num_phases;
12  callback function *cb;
13 public:
14  void reset(data buffers, callback function);
15  void start();
16 };
```

Figure 3. P2P State machine and non blocking interface

3.3 Generic Non-blocking Collectives

In this section we describe a framework for generic and portable non-blocking collective communication primitives that rely solely on the point-to-point active message primitive [13] provided by PAMI.

One of the key requirements of the collective subsystem is scalability, both in number of participating tasks and in number of simultaneously executing collectives. Any single executing collective cannot tie down or block execution threads or processes. This imposes a state machine based implementation. There is a separate state machine for every instance of an executing collective operation on each executing task, and it runs from the starting state to the final state during the execution of the collective. Since the collective state machine cannot block or reserve an execution thread, state changes are always caused by callbacks: completion callbacks from sent messages as well as callbacks for incoming messages. For more details on active message programing models please consult [1, 2, 13].

All our state machines are organized around a number of phases. In each phase the state machine sends and/or receives one active message. When these are complete, the state machine advances to the next phase. The number of required phases for typical algorithms like barriers, reductions, broadcasts, prefix sums is logarithmic in the number of tasks, which ensures that our state machines are scalable with respect to memory use.

Figure 3 shows the main data structure we maintain for each of the phases of a collective for the current implementation. This structure, at minimum, contains the destination where to send the data for a phase, the send data buffer, the source from where it will receive data and the receive buffer. We employ a simple convention that requires data buffers to be NULL when no message is sent or expected in any one phase.

We implement all collectives as individual classes, each class holding variables describing their internal state. The interface of such a class is shown in Figure 3. It includes two methods: `reset()` and `start()`, and a completion callback.

- The `reset()` method sets the state machine into the start state and initializes input and output buffers for the collective. This method is completely local (that is, it is not allowed to communicate).
- The state machine starts executing when the `start()` method is invoked. The `start()` method is guaranteed by the system to be non blocking: it returns to the user within a finite amount of time, independent of other tasks' progress. The method is allowed to start an arbitrary number of communication operations.

```
1 nphases = ceil(log(T));
2 for (int i=0; i<nphases; i++) {
3    int  dist     = 1<<(nhases-1-i);
4    int  sendmask = (1<<(nphases-i))-1;
5    destrank[i]   = myrank + dist;
6    if ((myrank&sendmask) ||
7        destrank[i]>=nprocs) destrank[i]=NULL;
8 }
```

Figure 4. Initializing state machine for binomial tree broadcast

- The completion callback (which is passed in as an argument to the reset() method) is invoked when the state machine reaches the final state (i.e. when the collective is complete). The API contract requires the completion callback to be non blocking as well, but does not prevent it from starting other communication primitives.

Note that completion semantics are *local*, i.e. completion of the state machine on one task does not intrinsically mean that the same instance has completed on any other task. This is completely in line with expected non-blocking collective semantics, e.g. a non-blocking barrier may *complete* on any task as soon as all other tasks have *started* executing it.

Figure 4 illustrates the calculations for building a state machine implementing a binomial tree broadcast from rank 0 to T tasks. In a binomial tree the senders are 0 in the first phase, 0 and T/2 in the second, 0, T/4, T/2 and 3T/4 in the third phase and so on; the destination ranks are T/2 in the first phase, T/4 and 3T/4 in the second phase, and so on. This simple rule is encoded by the `dist` and `sendmask` variables, which are calculated with simple shifts and masks. Similar calculations can be made for most well known generic collective algorithms, including butterflies, Bruck's algorithm [9] and others, giving us sufficient coverage for single-ported network algorithms.

3.4 Shared Memory Collectives

Modern processors employ an increasing number of cores (SMP) and support for simultaneous multithreading (SMT). Additionally, multiple processors are often packed in a shared memory node. In this section we describe the methodology employed by PAMI that allows collectives to exploit the shared memory address space.

For higher level languages and libraries like MPI, UPC, X10, etc., tasks running in a shared memory node may or may not be in the same process. In order to facilitate communication between tasks sharing a node, but not in the same address space, the PAMI library reserves a shared memory region that is accessible from every task on the node. This is done by using standard UNIX primitives such as the `shmctl()` or `mmap()` functions. In the remainder of this section we describe reduction and broadcast algorithms based on the assumption that a pre-allocated shared memory region is shared by all tasks running in an SMP node.

Shared Memory Reduction: We sketch out the code for reduction in Figure 5. Note that the code is incomplete due to space restrictions. The P tasks invoking the collective are logically organized as an n-ary tree, $\{2 \leq n < P\}$. Every node in the tree reads and reduces values from its children, and then posts its own value. The data dependency between children and parents imposes two synchronization operations: (a) children's contributions should not be read before they are posted and (b) a node's calculated value should not be updated before the parent has read it. The dependency is implemented by monotonically incremented counters, one per task, that are being read by other tasks to assess whether data is available.

Each task busy-waits for data to become available from children.This is accomplished using the WAIT macro in line 18 which blocks the incoming thread until the associated condition becomes true. Once the data arrives, the children's values are reduced (Line

```
1  // State machines (one per task)
2  struct {
3    int ctr;
4    TYPE buf;
5  }         * state;      // state machines
6
7  // Static setup (local to task)
8  int          me;         // my index
9  int          nchildren; // # of my children
10 int          chld[];     // list of my children
11 int          parent;     // idx of my parent
12
13 reduce (me, op) {
14  // 1. wait for children data & reduce
15  val = state[me].buf;
16  for (int c=0; c<nchildren; c++) {
17   WAIT (state[chld[c]].ctr>state[me].ctr);
18   reduce_op(val,state[chld[c]].buf,op);
19  }
20  // 2. wait parent to have read prev value
21  WAIT (state[parent].ctr>=state[me].ctr);
22  // 3. Post my value
23  state[me].buf= val;
24  __lwsync(); // write barrier
25  // 4. update my state counter
26  state[me].ctr++;
27  return val;
28 }
```

Figure 5. Shared memory blocking reduce

```
1  bool reduce_worker (me, op) {
2    // 1. test children data readiness.
3    // Bail if not ready.
4    for (int c=0; c<nchildren && finished; c++)
5      if (WAIT_NB(state[chld[c]].ctr<=state[me].ctr))
6        return false;
7    // 2. test parent status.
8    // Bail if not ready.
9    if (WAIT_NB(state[parent].ctr<state[me].ctr)))
10     return false;
11   // 3. perform reduction
12   val = state[me].buf;
13   for (int c=0; c<nchildren; c++)
14    reduce_op(val, state[chld[c]].buf, op);
15   // 4. post result, notify.
16   state[me].buf=val;
17   __lwsync();
18   state[me].ctr++;
19   return true;
20 }
```

Figure 6. Non blocking reduce

19). Next, the task waits for the parent to announce having read its *previous* posted value (Line 23). The *new* result can now be posted (Line 26) and announced (Line 30) to parent and children alike. Note the memory sync instruction in line 27. The instruction ensures that data is visible to other tasks before the "data ready" notification.

The algorithm, as depicted here, is blocking. Tasks will not exit the collective until their are done with their contribution. While this in general leads to good performance for the collective considered in isolation, it blocks the task from doing other available work.

We mentioned in Section 3 that all PAMI collectives are non blocking. To transform the algorithm in Figure 5 into a non blocking one we need to replace the busy-waiting-based dependency mechanism.

In one potential solution children notify their parents of data readiness by sending an active message. Upon receiving the message the parent increments a (local) counter and when all children are accounted for it performs the reduction and in turn notifies its own parent.

In the second approach, depicted in Figure 6, notification is, as before, by means of incrementing a variable. However, we do not busy-wait indefinitely on these variables. The `WAIT_NB` macro waits a predefined amount of time for its associated condition to become true. If the condition is still false when the allowed time expires, then `WAIT_NB` returns false. The reduction operation only takes place if all preconditions are met within a set amount of time. Otherwise the function execution is simply aborted and retried later by the underlying layers of PAMI. In this situation the operation is retried several times until the preconditions are met. This is accomplished by employing the PAMI work queue feature: Arbitrary functions can be queued up here for subsequent execution. A simple mechanism allows us to re-queue the reduction function as long as it keeps returning FALSE [2].

We found the second approach to have *much* higher performance than the first, due to the lower overhead of dependency mechanism. Active messages, even in shared memory, tend to have high overhead, whereas the PAMI task queue is relatively fast. By suitably adjusting the busy-wait time we can ensure that many non-blocking operations can be outstanding at the same time without impacting system performance. For this second approach, individual reduce performance is on par with the blocking version.

Shared Memory Broadcast: Similar with the reduce algorithm presented above we implemented various broadcast algorithms to accommodate both short and large messages. As before, dependencies are handled by monotonically increasing counters. We implemented a binary tree broadcast for small messages and a flat algorithm based on a star topology for large messages. In the flat algorithm the root writes the data and increments a counter while everybody else waits for the counter to be incremented before they start copying data. The advantage of the flat algorithm is that we can accommodate arbitrary large buffers with a minimum number of memory copy operations. A third algorithm we deployed is a variant of the flat broadcast where two buffers are used in an alternated manner. This improves overall performance by a factor of two due to pipelining reads and writes.

Like the reduce algorithm, broadcasts are made non blocking by allowing synchronization phases to abort the operation after a certain amount of busy-waiting; the broadcast is then posted into the PAMI task queue for subsequent retry.

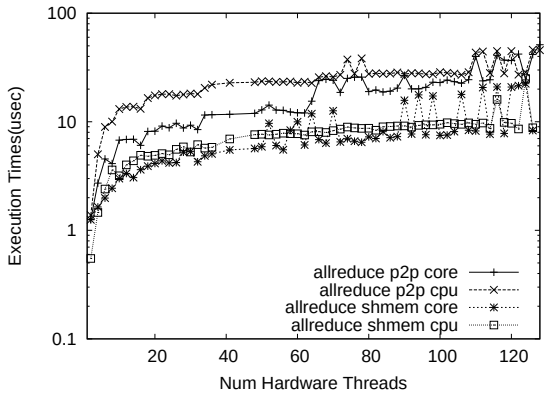
Shared Memory Allreduce: A simple allreduce operation can be implemented as a reduction followed by a broadcast. This can be easily achieved using the callback mechanism supported by our collectives. The allreduce will instantiate first a reduce operation whose callback function invokes the broadcast operation. More details on composition are provided in Section 4 as we describe hybrid collectives.

3.5 Evaluation of shared memory algorithms

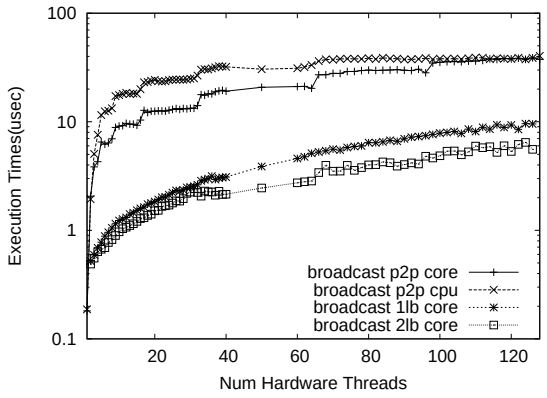
In this section we analyze the performance of the non blocking shared memory collectives on the P7IH system and compare their performance relative to the generic P2P implementation as described in Section 3.3. We focus the analysis on allreduce and broadcast for which experimental results are included in Figure 7.

To evaluate the performance we employed a simple kernel that repeats the same operation in a loop. A task does not advance to the next iteration until the current one is locally completed. The corresponding code is a while loop around the code included in Figure 2, Lines 12-17. This same kernel has been used for all experiments included in this paper. The PAMI library code is compiled with gcc 4.1.2 and O3 optimization level. Individual kernels for performance evaluation were compiled with xlC 11.1 and O3 optimization level.

The performance of shared memory collectives in the context of a multithreaded machine is highly dependent on the mapping

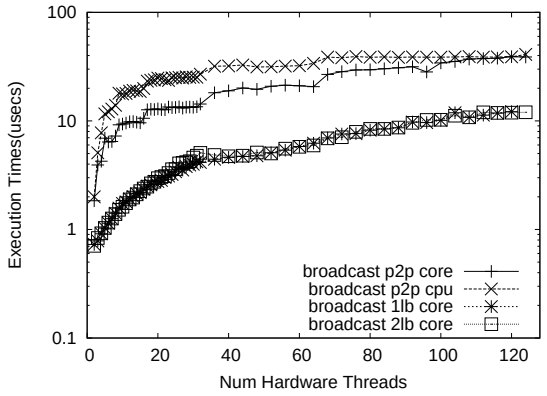


(a) Allreduce (integer, SUM)

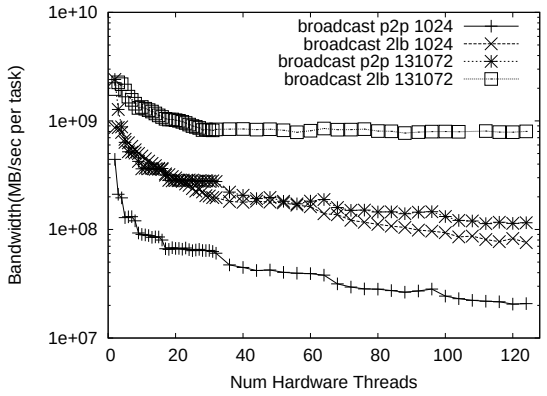


(b) Broadcast Latency (64bit data).1lb - one buffer, 2lb-two alternated buffers

Figure 7. Execution times for shared memory allreduce and broadcast for two different mappings on the machine. Execution time in microseconds for various number of hardware threads



(a) Broadcast Latency (64 bit data)



(b) Broadcast Bandwidth. Two buffer sizes

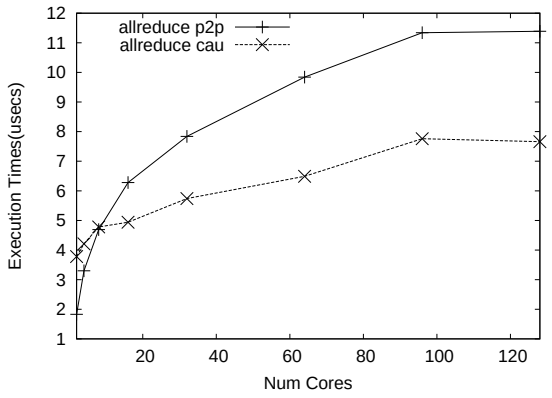
Figure 8. Broadcast execution time and bandwidth when different roots are used. Each iteration uses as root $iteration\ id\%num\ tasks$ The reported bandwidth is per task.

between software tasks and hardware threads. On P7IH, for both Linux and AIX, system software provides support to explicitly perform this mapping. In the experiments included in Figures 7, 8, we consider a P7IH shared memory node with 32 cores configured in a 4-way SMT; thus, 128 hardware threads.

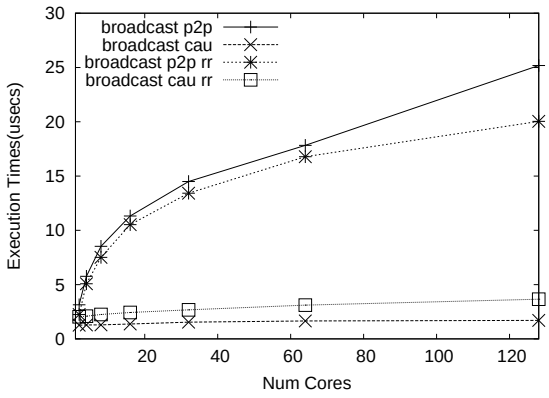
We evaluate two different mappings, called *core* and *CPU* respectively. The *core* mapping allocates first 32 application tasks one per core, on physical thread zero of each of the 32 cores. The next 32 tasks are mapped on physical thread 1 and so on. The *CPU* mapping maps the first four tasks on core zero, the next four tasks on core one and so on.

Figure 7(a) shows the execution time for the shared memory allreduce operation described in the previous section and the corresponding point to point implementation, for two different mappings of the tasks on the machine. We observe that both collectives scale well across a large number of tasks with the CPU mapping being less susceptible to noise due to cache sharing. Please note that the shared memory allreduce is an operation that performs mainly synchronizations with minimal computation. For compute intensive application running a task per core may be the preferred way. With 128 tasks the shared memory achieves 9 μs in average while the P2P performance is around 12 μs. As a reference point the POSIX threads barrier is 169 μs on 32 threads.

In Figure 7 (b) and 8 (a) we show the broadcast performance on 8 bytes long messages thus describing the latency characteristics of our algorithm. For P2P we use a binomial tree while for shared memory we use a flat tree using one (`broadcast_1lb`) or two alternate buffers (`broadcast_2lb`). We considered too strategies for selecting the root. In Figure 7 (b) we average the execution times for 100000 iterations using the same root zero. The shared memory algorithm with two buffers and core mapping has the lowest latency with 2.23 μs when using 32 tasks. For core mapping the single buffer option achieves 2.65 μs and the P2P is 14 μs, a much higher overhead in this case relative to the allreduce we previously analyzed. We notice the two buffers algorithm benefiting from the pipelining effects that takes place when multiple broadcasts are invoked in succession. In Figure 8 (a), every new iteration uses as root $iteration\ id\%number\_of\_tasks$ to avoid the pipelining effects. In this situation the two buffers version of broadcast doesn't have additional benefits over the single buffer. On 32 cores, using the core mapping the P2P achieves 14.3 μs, the shared memory version using one buffer 4.14 μs and the two buffers version 5.06 μs. The impact of the core mapping versus CPU is included only for the P2P broadcast the other two algorithms considered being less impacted by the mapping. For example on 32 threads the P2P core is 14.3 μs while the cpu is 26.9 μs, for one buffer the core mapping is 4.14 μs and cpu is 3.58 μs, and for two buffers version the core mapping is 5.06 μs while cpu is 3.41 μs. The shared memory version performs better with cpu mapping as the tasks are mapped on fewer cores thus benefiting from the L2 shared caches.



(a) Allreduce CAU versus P2P



(b) CAU Broadcast 64 bit data (rr - root rotated every iteration)

Figure 9. Execution times for CAU allreduce and broadcast. Execution time in microseconds for various number of nodes; One task per core per node

In Figure 8 (b) we look at the bandwidth performance of the broadcast for two buffer sizes in the situation where every iteration uses a different root. The performance of the shared memory version outperforms the P2P in terms of bandwidth for all range of buffer sizes considered. In Figure 8 we exemplify for two buffer sizes but additional results are available in Section 5. The interesting aspect of this experiment is to point out the good scalability of the algorithms on P7IH when scaling across all 128 hardware threads in a node. For a message size of 130K the bandwidth per core with 32 tasks is 1GB/sec and with 128 tasks is 800MB/sec.

3.6 Collective Accelerated Unit (CAU)

The CAU unit integrated in the IBM P7IH HUB provides offload and acceleration for broadcast and reduce up to 64 bytes of data. For reduce, it supports NO-OP, SUM, MIN, MAX, AND, OR and XOR operations with 32-bit/64-bit signed/unsigned fixed-point operands or single/double precision floating-point operands. The CAU device must be correctly initialized before deployment. Once a collective topology (set of nodes) is decided on, PAMI builds a tree of CAU devices. The maximum fan-out of the tree is 8. CAU hardware provides one unit per SMP node. That is, when multiple tasks on a node are in the topology they need to elect a leader task to operate the CAU. Up to 64 different CAU trees can co-exist in the same device, and they can be used simultaneously. Each CAU packet carries an identifier of the virtual topology it belongs to. Once a CAU tree is established, any of the tasks attached to the tree can issue a broadcast as the root of the operation. Other tasks in the tree will receive the data without any other software involvement. Reductions operate in exactly the opposite manner: all tasks but the root send in data which is combined by the CAU tree and delivered to the root. Obviously, all participants have to agree on the identity of the root.

To implement operations like barriers and all-reduce the CAU tree must be deployed twice in succession, i.e. a reduction followed by a broadcast. This doubles the operation's latency, but as we will see this method still handily outperforms any software-based algorithm we are capable of devising.

In Figure 9 we show the performance for allreduce (a) and broadcast (b) and compare them with P2P implementations. Results are included for up to 128 P7IH nodes, using only one task per node. We observe overall good scalability for both P2P and the CAU collectives, but the CAU outperforms the P2P solution even at this (relatively small) node count. For P2P allreduce the execution time increases from 1.83 μs on two nodes to 11.39 μs on 128, while for CAU allreduce the time increases from 3.78 μs on 2 to 7.66 μs on 128. For all but the smallest number of nodes, the factor of 2 loss in performance of the CAU (reduce followed by broadcast) is more than compensated for by the lower node-to-node latency.

Figure 9 (b) evaluates short broadcast performance for P2P versus CAU. Note that since CAU broadcasts do not require the two deployments of the CAU per operation, results are much better relative to P2P. We evaluate short broadcasts in two situations: using the same root (task=0) in every iteration vs. using rotating roots (a different broadcast root in every iteration). In the case of the CAU algorithm the fixed root test yields better latency than the rotating root test (2.24 vs. 3.6 μs latency on 128 nodes). The reason for this is very simple - in the CAU broadcast packets get pipelined. The P2P algorithm cannot benefit from pipelining because in this case software, not hardware, has to guarantee correct ordering of packets. Thus the P2P algorithm actually includes a set of message exchanges with barrier semantics, preventing any sort of pipelining. The scaling of the P2P algorithm is still logarithmic, but with a much higher constant factor, 20 μs to 24 μs per broadcast on 128 nodes.

4. Hybrid Collectives

In this section we present a *framework* that allows us to compose highly tuned hardware-specific algorithms to obtain proper general purpose collective implementations and yet retain the performance characteristics of their components.

We have so far presented three sets of collective implementations on IBM P7IH: point-to-point messages, shared memory and CAU based. P2P collectives are general and are useful on almost any system. However, they come with built-in efficiency compromises. Shared memory and CAU collectives are designed for proper subsets of the P7IH system hierarchy. In PAMI terminology these are called *local topologies* (for shared memory) and *leader topologies* (one representative task per node), respectively. Topology specific collectives are very efficient, but do not provide a general purpose solution. [1]

To illustrate our approach, consider Figure 10, in which we use a composition of local and leader topologies to execute broadcast across a hybrid system. An initial (shared memory) broadcast within the local topology containing the root is followed by a broadcast across leaders in each node and concluded by a sec-

[1] In this paper we restrict ourselves to local and leader topologies. However, PAMI represents many other topologies that carry built-in advantages: for example, on the Blue Gene machines, with its various torus shaped networks, multidimensional rectangular shapes are considered advantageous because of the "broadcast bit" feature of the network [20].

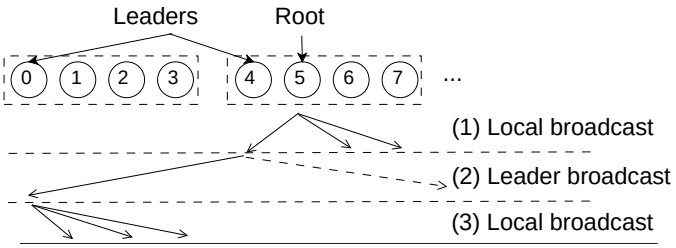


Figure 10. Hybrid broadcast as a three phase composition of local and leader broadcasts

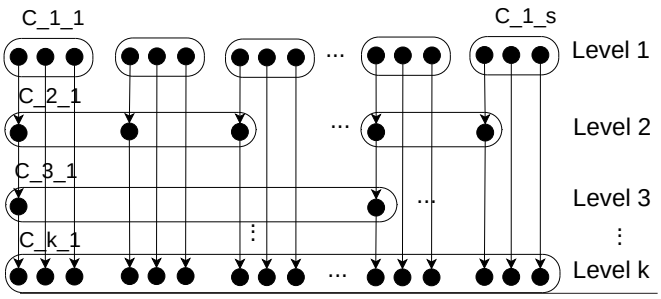


Figure 11. Collective composition

ond shared memory broadcast within the nodes whose leaders just received the data.

Our compositional approach rests on the following design features:

- **Optimized components:** Optimized implementations for various subsets of the system hierarchy where high performance can be obtained via hardware support and/or optimized algorithms. In this paper we will consider the shared memory and CAU collectives described in previous sections.
- **Chaining:** The ability to seamlessly compose collective components without sacrificing performance characteristics.
- **Composition:** The ability to decompose a collective to execute on a chained sequence of optimized topologies defined by hardware constraints. This allows us to break down the general problem into pieces that already have optimized solutions.

4.1 Chaining non blocking collectives

The Figure 3 in Section 3.3 shows the proposed application programming interface (API) for non blocking collectives in our framework. The interface includes the `reset` and `start` methods, as well as a callback to be invoked when the collective has terminated. To summarize: `reset()` is guaranteed to be local, `start()` is guaranteed to be non blocking, and the completion callback, provided by the user, is *required* by the API to be non blocking.

The API is carefully crafted to stay invariant under chaining. Our idea of chaining collectives is to use the completion callback of an already executing *primary* collective to start one or more *secondary* collectives. We call such completion callback *continuation functions*, since they don't actually complete the chained collective. Starting a secondary collective involves calls to its `start()` and `reset()` methods. Since these are guaranteed to be non blocking and local respectively, we can satisfy the API contract requirement that our primary completion callback be non-blocking.

4.2 Collective Decomposition Formalism

In this Section we formalize the compositional process to show its further generality as a methodology for deploying efficient parallel algorithms. As we mentioned already a collective is invoked simultaneously by a set of tasks that we refer in this paper as topology and we use the following notation $T = \{t_1, ..., t_n\}$.

For a given topology T we can compute the local topologies $Local(T) = \{Local_1(T), ..., Local_s(T)\}$ where s is the number of SMP nodes in the system and $Local_j(T) = \{t | t \in T \; and \; t \; mapped \; on \; the \; same \; SMP \; node \; j\}$ (e.g., the subset of tasks mapped on the same SMP node).

The leader topology of a given topology T, $Leader(T)$ is similarly defined as a set of leader tasks from T, one from each SMP node. Yet more topologies can be defined on other architectures.

A collective C then will be instantiated with a given topology T and has an associated callback function to be invoked on all members of T when the collective locally completes. We denote this as $C(T, CB)$.

A composed collective then $C_{Composed}(T, CB)$ is defined based on a set of existing collectives and rules on how the collectives are chained together using continuation functions (intermediary callbacks).

Hierarchical chaining: One particularly useful approach to tackle physical hierarchies in the P7IH system is the hierarchical decomposition pattern. In this approach (shown in Figure 11) a composed collective is conceptually organized as a number of levels $\{L_1, L_2, ...L_l\}$. One or more collective instances, executing on different tasks, exist in each level; they are all instances of the same collective, although operating on different topologies and possibly with different callback functions. Thus a level L_k of the composition, $L_k = \{C_{k_1}(T_{k_1}, CB_{k_1}), C_{k_2}(T_{k_2}, CB_{k_2})...\}$ consists of a set of collectives with their associated topologies and completion callbacks.

Successive levels in the hierarchy are started by continuations of previous levels. In the example included in Figure 11 we have collectives at level 1 employing different continuation functions for different tasks. We see that upon local completion of the collective at level one certain tasks will start the collectives at level two while other tasks may start collectives at level K. The only requirement is that lower levels can not be re-invoked. The last continuation function invoked by each task invokes the user completion callback.

The performance of such a hierarchically decomposed collective is ultimately determined by its *critical path*, i.e. the maximum number of distinct levels traversed by any of the tasks involved in the collective.

Pipeline chaining: another useful pattern that emerged during our experiments is pipeline composition. Pipelining allows us to chain together multiple instances of a collective.

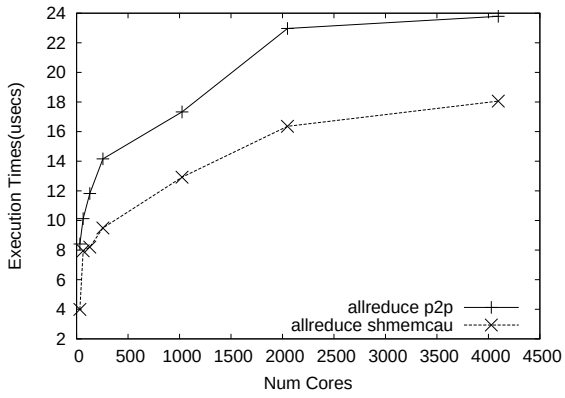
Specifically, System V shared memory is a precious resource. We cannot internally allocate shared memory chunks to accommodate e.g. arbitrarily large broadcasts. Pipeline chaining allows us to allocate a single instance of broadcast with a limited amount of shared memory, and allow it to restart itself in the continuation function, jumping forward in the user buffer with every new started instance. This can go on until there is no more user buffer to send, in which case the continuation function can revert to calling the next level in a hierarchy.

4.3 Composing hybrid collectives for the P7IH

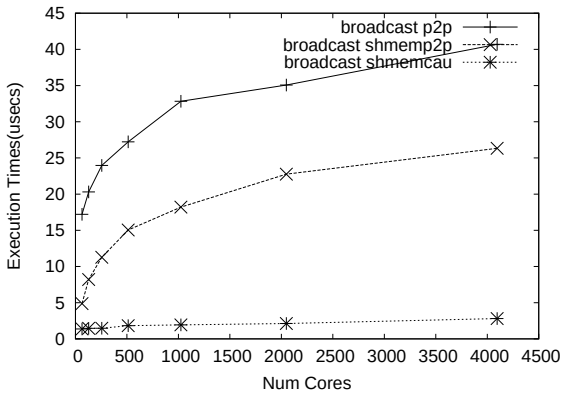
In this section we present the composition of the P7IH hybrid collectives we actually implemented and tested. We start off with a **low latency allreduce** for short messages on P7IH consisting of four levels:

$ShortAllReduce(T, CB) = \{L_1, L_2, L_3, L_4\}$, where:

- **L1 = Initial shared memory reduction**:
  $L_1 = \{SharedMemReduce(LocalTopology_j(T), CF_1\}$, where j scans all local topologies (shared memory nodes) for this collective.
  - CF_1 starts L_2 on all tasks of $LeaderTopology(T)$



(a) Allreduce



(b) Broadcast 64 bit data

Figure 12. Execution times for hybrid allreduce and broadcast. Execution time in microseconds for various number of cores

- CF_1 starts L_4 on all tasks of $T \setminus LeaderTopology(T)$,

- **L2 = Leader CAU reduction:**
  $L_2 = \{CAUReduce(LeaderTopology(T), CF_2\}$
  - CF_2 starts L_3 on $LeaderTopology(T)$
- **L3 = Leader result broadcast:**
  $L_3 = \{CAUBroadcast(LeaderTopology(T), CF_3\}$
  - CF_3 starts L_4 on $LeaderTopology(T)$
- **L4 = Final shared memory broadcast:**
  $L_4 = \{SharedMemBroadcast(LocalTopology_j(T), CB\}$
  - CB is the completion callback provided by the user.

In a similar vein, our **low latency broadcast** for P7IH consists of three levels: a shared memory broadcast on the local topology containing the root, followed by a CAU accelerated broadcast across leaders, concluded by shared memory broadcasts on the local topologies not owning the root.

For **high bandwidth hybrid broadcast** optimized for medium to large messages we use the same three phases as for short broadcast, but using a pipelined shared memory broadcast for the first and third phase, and a point to point broadcast across leaders (CAU is primarily useful for short messages).

5. Performance evaluation

In this section we evaluate the performance of our composed hybrid collectives. We compare the latency and bandwidth of our hybrid allreduce and broadcast against more traditional implementations.

For the experiments in this section we used a different machine than the one introduced in Section 3.5. The P7IH system we employed here has four supernodes, in other words 128 nodes of 32 cores each for a total of $32 \times 128 = 4096$ cores. For the shared memory experiment in Section 3.5 we used a single node machine configured in SMT 4 (each core had all four threads enabled) and using an older network hardware (HFI 2.0). For this section the machine used was configured in SMT 2 (only two hardware threads enabled) and the latest network interconnect (HFI 2.1).

For all experiments on hybrid collective performance we map one application task per process and use the core mapping as described in Section 3.5. All times are averaged over 100000 iterations. For all broadcast experiments included in this section we use as rotating roots (every iteration has a different root) in order to guarantee that latency is measured correctly without pipelining broadcasts.

5.1 Latency study

In Figure 12 we show the performance of the allreduce (a) and short broadcast (b). The short allreduce is of a particular importance to us because it is used by the implementation of the barrier primitive in the UPC language. [2]

Figure 12 (a) shows the scalability of our composed hybrid allreduce. As explained in Section 4.3 the composed allreduce has two shared memory and CAU components each, and we expect to measure the four components' latencies. Thus we expect

$T \approx T_{CAUreduce} + T_{CAUbcast} + T_{shmreduce} + T_{shmbcast}$

Drawing on numbers presented earlier in the paper - shared memory reduction on 32 threads is in the 6 μs range; shared memory broadcast in the 2 μs range; and CAU allreduce on 128 nodes is in the 7-8 μs range. Thus we expect 16 μs or so on 2048 tasks. We are actually measuring 18 μs close to the expected value. Thus the effect of factors like implementation overhead, system noise etc. can be capped at 15% or less. The point to point alternative of short allreduce (a non-hybrid, standard butterfly based algorithm) also scales well, but is about 50% behind the optimized algorithm, 24 μs on 4096 cores.

Results for short (8 byte word) broadcast are in Figure 12 (b). Here we compare three versions against each other: a standard binomial algorithm, a composite algorithm with optimized shared memory communication (shmem+P2P) and finally a shmem+CAU algorithm as described in Section 4.3.

All three algorithms show logarithmic scaling behavior. Predictably, the shmem+P2P algorithm has better latency than pure P2P (26 μs vs 40 μs on 4096 cores). Just as unsurprisingly the shmem+CAU combination beats them both (2.8 μs on 4096 cores).

5.2 Bandwidth study

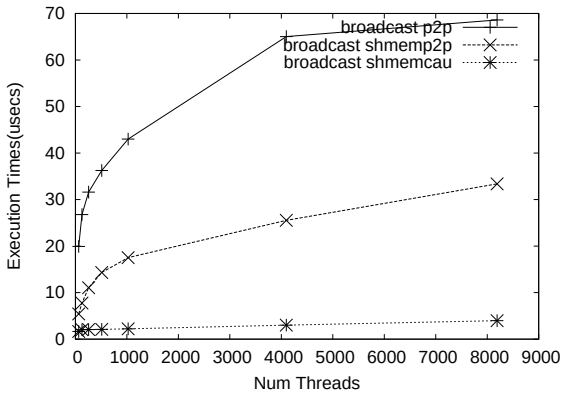
Figure 13 compares the pure P2P binomial tree broadcast with a hybrid shmem+P2P algorithm. We consider both scaling behavior (number of cores from 32 to 4096) and message size (1 KByte to 8 MBytes). We report broadcast bandwidth in MBytes/s/task (not the aggregate number).

We observe that the hybrid algorithm (gray background columns) performs better than its P2P counterpart for all buffer sizes and all number of cores considered. However, the difference varies with scale and message size. With 32 tasks the hybrid algorithm is reduced to shared memory only, and hence outperforms the P2P ver-

[2] UPC barriers have an optional integer argument. The UPC runtime checks expressions supplied by various UPC participants in the same barrier for equality. This helps UPC programmers to ensure code correctness. A UPC barrier is therefore actually implemented as a PAMI allreduce with a MAX operator rather than an actual barrier.

Algo	P2P	Hyb	P2P	Hyb	P2P	Hyb	P2P	Hyb	P2P	Hyb	P2P	Hyb	P2P	Hyb	P2P	Hyb
Size/Cores	32	32	64	64	128	128	256	256	512	512	1024	1024	2048	2048	4096	4096
1KB	60	210	47	164	39	97	34	71	25	54	24	43	22	32	19	31
16KB	278	637	158	620	139	486	122	347	112	249	102	208	98	183	80	143
65KB	237	766	164	712	140	643	121	545	107	427	95	372	127	315	79	275
128KB	281	1573	219	1209	191	727	172	575	154	491	140	412	143	359	117	315
512KB	431	1791	392	995	348	675	313	561	284	488	257	413	190	370	210	330
1MB	421	1860	400	1146	360	719	329	602	302	520	278	445	208	391	228	344
2MB	490	1756	470	1162	418	726	382	599	348	516	321	450	284	396	260	348
4MB	628	1612	567	1049	503	696	455	577	411	504	374	440	323	386	298	343
8MB	734	1507	642	1008	564	691	507	574	455	499	418	434	360	382	321	340

Figure 13. Broadcast bandwidth for various buffer sizes and number of cores. The reported bandwidth is in MB/second per task



(a) Broadcast latency

Algo	P2P	Hyb	P2P	Hyb	P2P	Hyb	P2P	Hyb
Size/HThreads	64	64	128	128	4096	4096	8192	8192
1KB	37	152	29	106	12	30	12	23
65KB	151	922	86	800	39	262	38	227
128KB	184	1223	114	657	61	155	58	157
512KB	202	1554	192	815	125	290	114	238
1MB	236	1537	255	914	160	359	146	319
2MB	272	1180	312	937	189	372	167	330
4MB	301	663	340	589	211	374	177	317
8MB	315	619	359	615	212	355	201	271

(b) bandwidth for various buffer sizes and number of hardware threads

Figure 14. Hybrid broadcast performance for SMT2: Latency and Bandwidth

sion by a factor between 2 and 5. As the number of nodes increases the relative advantage of the hybrid algorithm diminishes, since the shared memory broadcast contributes a declining proportion of the total execution time as the number of cores is scaled up.

We observe a similar effect for message sizes. Since the chief advantage of the shared memory implementation is lower overhead, larger message sizes tend to wash out the difference between hybrid and P2P only; on 4096 cores with 8MB buffers, the hybrid algorithm only holds a 5% advantage.

5.3 SMT impact on hybrid collectives

In this section we analyze the latency and bandwidth for broadcast operation when using two hardware threads per core. We include in Figure 14 results for the broadcast algorithms discussed previously in this section. In the absence of other user computation we observe very good scalability for both latency and bandwidth. In terms of latency (Figure 14(a)), when using 4096 cores and 8192 hardware threads the binomial P2P achieves 68 μs, the hybrid (Shared Memory local and P2P on leaders) is 33 μs and the hybrid version employing CAU is 3.9 μs.

For the broadcast bandwidth, we include experimental results for various buffer sizes and number of cores in the table in Figure 14 (b). The core mapping has been used for the included results. Similar with the single thread per core results, we observe good scalability for both algorithms, with the hybrid algorithm leading the P2P on all buffer sizes considered. For 64 tasks mapped on 64 hardware threads the hybrid algorithm employs only the shared memory part. In this situation the hybrid algorithm is between 7 (512K buffer) and 2 (8MB) times faster than the P2P version. When using 8192 tasks mapped on as many hardware threads the hybrid algorithm provides 34% more bandwidth relative to P2P. Recall that when not using SMT the hybrid version had only 5% advantage over P2P. Thus when using multiple hardware threads the advantages of shared memory are even bigger.

6. Related Work

Specializing collectives to take advantage of the underlying hardware and specialized networks is a common approach for obtaining the best performance on modern high performance systems [4, 18, 21, 25, 27]. In [4] the authors describe optimization for BlueGene/L. Other major vendors like Cray, Infiniband also provide support for collective acceleration and various academic papers describe optimizations of the MPI library to obtain good performance. For example, in [21, 25] the authors describe a hybrid solution for broadcast using shared memory and Infiniband multicast support. Similar the authors of [16] show how Infiniband can be used to accelerate non blocking collectives.

In [15, 17] the authors present the Non Blocking Collective library (libNBC) as an extension to MPI collectives to provide non blocking functionality. In their model a collective maintains an internal schedule consisting of rounds with operations in a round proceeding only when the previous round completed. Our multi phase mechanism described in Section 3.3 is similar with this approach. The main difference between our approach and this one is the fact that we provide the call back mechanism to signal completion. In libNBC, upon a collective invocation, a handle is initialized and later on the handle can be queried if the collective has finished. We showed in Section 4 that the callback mechanism is essential for us to efficiently compose non blocking collectives to achieve new ones. We can't envision an elegant composition mechanism using the libNBC approach.

Existing parallel programming languages and libraries [8, 14] already exploits shared memory nodes to improve the performance

of both point to point and collective operations. In [8] the authors describe performance improvements for point to point operations for the UPC language when optimizing for shared memory intra node communication. However they have only a minimal section on collectives optimizations.

In [19] the authors perform an in depth performance study of the P7IH various intra and inter node communication links describing achievable latencies and bandwidths. They measure the performance using simple point to point kernels and discuss it relative to the maximum specified values. In this paper we discuss for the first time the performance of the collective operations on P7IH, including the novel CAU not presented elsewhere.

In [25], the authors discuss composing collective communication algorithms from primitives. The authors focus there on specific optimization enabled by Infiniband hardware. Our description of collective composition however is more general and establishes the bases under which arbitrary algorithms can be combined to exploit the various levels of a hardware architecture

7. Conclusion and Future Work

In this paper we have presented an experimental evaluation of collective communication primitives on the IBM P7IH architecture. We presented initial measurements of the collective acceleration unit (CAU) running reduction and broadcast collectives. We also presented collective communication algorithms optimized for the rather large SMP nodes of the P7IH.

Our second contribution in this paper is a novel framework and a formalism that allows us to chain together non blocking collectives optimized for different subsets of the system hierarchy. We demonstrated that the combined hybrid algorithms in our framework offer the best chance to achieve good performance and scalability on the IBM P7IH system.

In the future we will extend our work to other modern leadership class systems. Since large scale SMPs and deeply hierarchical systems are here to stay we expect that our work will be relevant on many other architectures.

References

[1] *Parallel Environment Runtime Edition for AIX, LAPI Programming Guide, Version 1 Release 1.0, IBM*.

[2] *Parallel Environment Runtime Edition for AIX, PAMI Programming Guide, Version 1 Release 1.0, IBM*.

[3] *Parallel ESSL for AIX V4.1 Guide and Reference, IBM*.

[4] G. Almási and all. Optimization of mpi collective communication on bluegene/l systems. In *Proceedings of the 19th annual international conference on Supercomputing*, ICS '05, pages 253–262, New York, NY, USA, 2005. ACM.

[5] G. Almasi, P. Hargrove, I. G. Tanase, and Y. Zheng. UPC Collectives Library 2.0. In *Proceedings of the Fifth Conference on Partitioned Global Address Space Programming Models*, Oct. 2010.

[6] R. X. Arroyo, R. J. Harrington, S. P. Hartman, and T. Nguyen. Ibm power7 systems. *IBM Journal of Research and Development*, 55(3):2:1 –2:13, may-june 2011.

[7] G. Bikshandi, J. G. Castanos, S. B. Kodali, V. K. Nandivada, I. Peshansky, V. A. Saraswat, S. Sur, P. Varma, and T. Wen. Efficient, portable implementation of asynchronous multi-place programs. *SIGPLAN Not.*, 44:271–282, February 2009.

[8] F. Blagojević, P. Hargrove, C. Iancu, and K. Yelick. Hybrid pgas runtime support for multicore nodes. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, pages 3:1–3:10, New York, NY, USA, 2010. ACM.

[9] J. Bruck, C.-T. Ho, S. Kipnis, E. Upfal, and D. Weathersby. Efficient algorithms for all-to-all communications in multiport message-passing systems. *Parallel and Distributed Systems, IEEE Transactions on*, 8(11):1143 –1156, nov 1997.

[10] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40:519–538, October 2005.

[11] C. Coarfa, Y. Dotsenko, J. Eckhardt, and J. Mellor-Crummey. Co-array fortran performance and potential: An npb experimental study*.

[12] DARPA High Productivity Computing Systems. http://www.darpa.mil/ ipto/programs/hpcs.

[13] T. Eicken, D. Culler, S. Goldstein, and K. Schauser. Active messages: A mechanism for integrated communication and computation. In *Computer Architecture, 1992. Proceedings., The 19th Annual International Symposium on*, pages 256 –266, 1992.

[14] E. Gabriel and all. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[15] T. Hoefler and A. Lumsdaine. Design, Implementation, and Usage of LibNBC. Technical report, Open Systems Lab, Indiana University, Aug. 2006.

[16] T. Hoefler and A. Lumsdaine. Optimizing non-blocking Collective Operations for InfiniBand. In *Proceedings of the 22nd IEEE International Parallel & Distributed Processing Symposium, CAC'08 Workshop*, Apr. 2008.

[17] T. Hoefler, A. Lumsdaine, and W. Rehm. Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI. In *Proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC07*. IEEE Computer Society/ACM, Nov. 2007.

[18] N. Karonis, B. de Supinski, I. Foster, W. Gropp, E. Lusk, and J. Bresnahan. Exploiting hierarchy in parallel computer networks to optimize collective operation performance. In *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, pages 377 –384, 2000.

[19] D. Kerbyson and K. Barker. Analyzing the performance bottlenecks of the power7-ih network. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 244 –252, sept. 2011.

[20] S. Kumar and all. The deep computing messaging framework: generalized scalable message passing on the blue gene/p supercomputer. In *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, pages 94–103, New York, NY, USA, 2008. ACM.

[21] A. Mamidala, L. Chai, H.-W. Jin, and D. Panda. Efficient smp-aware mpi-level broadcast over infiniband's hardware multicast. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, page 8 pp., april 2006.

[22] MPI Forum. Mpi:a message-passing interface standard (version 1.1). technical report (june 1995), January 2012. available at: http://www.mpi-forum.org (Jan. 2012).

[23] R. Rajamony, L. B. Arimilli, and K. Gildea. Percs: The ibm power7-ih high-performance computing system. *IBM Journal of Research and Development*, 55(3):3:1 –3:12, may-june 2011.

[24] B. Sinharoy, R. Kalla, W. J. Starke, H. Q. Le, R. Cargnoni, J. A. Van Norstrand, B. J. Ronchetti, J. Stuecheli, J. Leenstra, G. L. Guthrie, D. Q. Nguyen, B. Blaner, C. F. Marino, E. Retter, and P. Williams. Ibm power7 multicore server processor. *IBM Journal of Research and Development*, 55(3):1:1 –1:29, may-june 2011.

[25] H. Subramoni, K. Kandalla, S. Sur, and D. Panda. Design and evaluation of generalized collective communication primitives with overlap using connectx-2 offload engine. In *High Performance Interconnects (HOTI), 2010 IEEE 18th Annual Symposium on*, pages 40 –49, aug. 2010.

[26] G. Tanase and all. The stapl parallel container framework. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 235–246, New York, NY, USA, 2011. ACM.

[27] V. Tipparaju, J. Nieplocha, and D. Panda. Fast collective operations using shared and remote memory access protocols on clusters. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, IPDPS '03, pages 84.1–, Washington, DC, USA, 2003. IEEE Computer Society.

[28] UPC language specifications, v1.2. Technical Report LBNL-59208, Lawrence Berkeley National Lab, 2005.