Shared Memory Based MPI Broadcast Algorithms for NUMA Systems

Mikhail Kurnosov(✉) and Elizaveta Tokmasheva

Siberian State University of Telecommunications and Information Sciences, Novosibirsk, Russia
{mkurnosov,eliz_tokmasheva}@sibguti.ru

Abstract. `MPI_Bcast` collective communication operation is used by many scientific applications and tend to limit overall parallel application scalability. This paper investigates the design and optimization of broadcast operation for NUMA nodes with GNU/Linux. We describe algorithms for `MPI_Bcast` that take advantage of NUMA-specific placement of queues in a shared memory for message transferring. On a Xeon Nehalem and Xeon Broadwell servers, our implementation achieves on average 20–60% speedup over algorithms of Open MPI `coll/sm` and MVAPICH.

Keywords: MPI · Broadcast · Collectives · NUMA

1 Introduction

High-performance computing systems are growing intensively in two directions: compute node counts and number of cores per node. Many of the supercomputers are built on multi-processor nodes with non-uniform memory architecture (NUMA), it becomes increasingly important for MPI to leverage shared memory for intra-node communication.

Broadcast is an important communication operation in HPC. For a significant number of parallel algorithms and packages of supercomputer simulation, the performance (execution time) of broadcast operation is critical. The MPI standard defines an `MPI_Bcast` routine for single source non-personalized broadcast operation, in which data available at a root process is sent to all other processes. On shared memory systems broadcast can reduce the number of memory transfers with multiple consumers accessing a shared buffer. The most used double-copy (copy-in/copy-out) algorithms involve a shared buffer space used by local processes to exchange messages. The root process copies the content of the message into the shared buffer before the receiver reads from it.

In this paper, we investigate the problem of message broadcasting from the root process to other processes over shared memory of a NUMA machine with GNU/Linux operating system.

This work is supported by Russian Foundation for Basic Research (project 18-07-00624).

V. Voevodin and S. Sobolev (Eds.): RuSCDays 2020, CCIS 1331, pp. 473–485, 2020.
https://doi.org/10.1007/978-3-030-64616-5_41

Main contributions of this paper include: (1) NUMA-aware algorithms for `MPI_Bcast` operation are based on k-ary, k-nomial, chain and flat notification trees. In contrast to other works our algorithms explicitly allocate memory for queues from local NUMA nodes even with active linux page cache readahead subsystem; (2) Optimal values of the size s of buffer and length s of the queue what takes no more than b bytes and provides minimum algorithm time. On NUMA machines with Xeon Nehalem and Xeon Broadwell processors, our implementation based on Open MPI achieves on average 20–60% speedup over algorithms of Open MPI `coll/sm` and MVAPICH (`mv2_shm_bcast`).

The paper is organized as follows. Section 2 discusses related work. Section 3 presents an overview of our approach and describes the shared-memory `MPI_Bcast` for NUMA system implemented within the Open MPI. Analyses and experimental results are presented in Sect. 4. Section 5 summarizes and concludes.

2 Related Work

Modern MPI implementations optimize intra-node collective communication in two different ways: (1) using intra-node point to point communication and minimizing inter-node interactions [1,2,7–9,11,12,15]; (2) allocating a shared memory region that can be used for the communication across processes in the same node [4–6,10,13,14]. The main part of shared memory based `MPI_Bcast` algorithms are based on two step procedure [1–6]. At communicator creation time a set of queues is formed in a shared memory region and a message is transferred over queues at each call of `MPI_Bcast`. The root process copies fragments of the message into the shared queue and the non-root reads from it. This approach is called copy-in/copy-out (CICO, double-copy) and is widely used in practice because it provides portability, and does not require additional libraries and additional permissions from the operating system. Scalability of CICO algorithms are limited by double copying of fragments and waiting for the readiness of the data in the queue.

Along with the CICO method in many MPI implementations a zero-copy approach is used. Zero-copy algorithms perform one copying of each fragment without using of an intermediate buffer. They use special possibilities of operating system to copy of a data from address space of one process into another. Well known examples are KNEM [13], XPMEM and linux Cross Memory Attach. In [6,13,14] a process distance-aware adaptive collective communication framework based on KNEM is proposed. Kernel-assisted collective algorithms do not use intermediate queues in a shared memory segment. This paper addresses problems of CICO algorithms with queues in a shared memory region.

In MVAPICH [3] processes create a shared memory segment with a cyclic queue of $w = 128$ slots for each process. Each slot contains a buffer to store a fragment of $f = 8192$ bytes and an operation number psn. The root process uses flat tree and psn to notify other processes about data readiness. If the queue is full, the root process waits on the barrier until all processes have finished

copying. The total size of the shared memory segment is $O(pwf)$, and an each process requires an $O(pw)$ bytes of memory.

In the paper [4] authors proposed to use p cyclic queues in a shared memory. The queue includes w buffers and is divided into $q = 2$ sets (banks) of memory with each set having several buffers. When the last buffer in the set is used, a non-blocking barrier is initiated. Multiple sets are used to allow the non-blocking barrier to complete while another set is in use reducing the synchronization costs. The root process uses a complete k-ary tree for message transferring and notifications. Algorithm is implemented in `coll/sm` component of the Open MPI. The total size of the shared memory segment is $O(pwf)$ and an each process requires $O(w + pk)$ bytes of memory.

In [5] authors use a single queue divided into $w = 4$ buffers and two synchronization flags per process. An each buffer occupies $f = 8192$ bytes of memory. One of the synchronization flags is used when a process copies its data to the shared buffer, to notify that new data is available. The other flag is used when a process copies the data out of the shared buffer, to signal that it has read the contents of the buffer and the buffer can be reused. A broadcast is implemented using a release followed by a gather step. During the release step, the parent copies the message into the shared queue and updates the children's release flag. Child processes wait on the shared release flag and copy out the data from the buffer. After the release step, in the gather step the children processes signal the parent that they have completed copying the data. Authors use k-ary and k-nomial trees for notifications. The size of the shared memory segment is $O(p + wf)$.

Algorithms in MVAPICH, Open MPI and in [5] allocate memory pages for queues without explicit binding to local NUMA nodes. This can lead to allocating of memory pages for queues from a NUMA node of the master-process which created shared-segment. As a consequence, the amount of inter-socket exchanges can increase. Our approach takes advantage of NUMA-specific placement of queues in a shared memory and tries to minimize a volume of inter-socket traffic.

3 Bcast Algorithms

The developed algorithms include two stages. At communicator creation time they form a set of queues in a shared memory region for inter-process communications. After that, on each call of `MPI_Bcast` a message is transferred from the root process over its queue to others processes.

3.1 Shared Memory Segment Structure

At MPI communicator creation time (including `MPI_COMM_WORLD`) all processes form a shared memory segment. The POSIX-compatible system call `mmap` is used for this purpose. Process 0 allocates memory in shared region and other processes attach it to its address space. The size of the allocated segment and

individual blocks is a multiple of a memory page. The reason is that NUMA memory binding is controlled by linux kernel at the level of memory pages.

Each of the first q memory pages contains two shared counters *shm_op* and *shm_nreaders* (by default $q = 2$). They are used to synchronize access of processes to shared queues during the `MPI_Bcast` operation. The addresses of the counters are aligned to a cache line boundary to reduce possible false sharing. Further, the shared memory region contains for each of p processes a cyclic queue of s buffers ($shm\_queue[rank][s]$) and an array of s control blocks ($shm\_controls[rank][s]$). Each buffer has f bytes length and occupies minimum number of memory pages. A size of control block is one page length. By default we use $s = 8$ and $f = 8192$ bytes. Figure 1 shows an example of shared memory segment structure for $p = 8$ processes running on two NUMA nodes (two quad core processors). In general, the size of a shared segment depends linearly on the number p of processes and queue length s and occupies $O(qw + ps(f + w))$ bytes of memory, where w is the memory page size. In practice, the queue's length s and buffer size f should be chosen taking into account the available memory size. For example, at $p = 64$ processes and $s = 1024$, $f = 8192$ the memory segment will occupy 384 MB.

After calling `mmap`, each process initializes areas of the segment with its data structures: it zeroes control blocks and the first byte of each page of all queue buffers. This ensures that physical memory pages are allocated from its local NUMA node (using the *first touch policy* of the linux kernel). Memory pages with shared counters *shm_op* and *shm_nreaders* are initialized by the process 0. Overall initialization time linearly depends on the number p of processes, queue length s and the number q of sets.

According to default linux memory policy, the first access to any address *addr* on the segment will allocate a physical memory page from the local NUMA node of the process and a certain number of pages for the following addresses will be allocated from the same NUMA node. This is done by page caching subsystem (linux page cache readahead) which speculatively sequentially reading memory-mapped file (shared region) into the page cache. Default behavior of readahead subsystem may cause incorrect allocation of memory pages for queues and control blocks of processes $1, 2, \ldots, p-1$ from NUMA node of the process 0 (it performs a first modification of the shared region). Algorithms of Open MPI `coll/sm` and MVAPICH ignore NUMA topology – pages for shared data structures are allocated from NUMA node of process 0. This increases `MPI_Bcast` operation time due to the increased number of accesses to remote NUMA nodes. In our algorithms to establish correct allocation of memory pages from NUMA nodes we temporarily disable sequential readahead immediately after `mmap` by calling `madvise(seg, segsize, MADV_RANDOM)`. This ensures correct allocation of memory pages for queues and control blocks from local NUMA nodes.

Control blocks are used by the root process to notify other processes about data readiness in the queue. The root copies the fragment i of a message to his queue $shm\_queue[root][i]$ and notifies processes *rank* by writing fragment size to their control blocks $shm\_controls[rank][i]$. Each non-root process spin waits

Process owner	Data Block	Content	Size	NUMA-node
0	*shm_op*	1	4KB	0
	shm_nreaders	0		
0	*shm_op*	1	4KB	0
	shm_nreaders	0		
0	*shm_controls*[*s*]		8 · 4KB	0
	shm_fragments[*s*]		8 · 8KB	
1	*shm_controls*[*s*]		8 · 4KB	1
	shm_fragments[*s*]		8 · 8KB	
2	*shm_controls*[*s*]		8 · 4KB	0
	shm_fragments[*s*]		8 · 8KB	
...				
7	*shm_controls*[*s*]		8 · 4KB	1
	shm_fragments[*s*]		8 · 8KB	

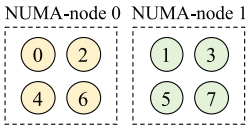

Fig. 1. Shared memory segment structure: $p = 8$ processes on two NUMA nodes; memory page size $w = 4\,\text{KB}$; queue length $s = 8$, number of sets per queue $q = 2$, buffer size $f = 8\,\text{KB}$ (total segment size is 776 KB).

on its own control block until the value becomes positive. We have implemented four algorithms using various trees to propagate notification from the root process to others: completed k-ary tree, k-nomial tree, chain tree and flat (linear) tree (Fig. 2).

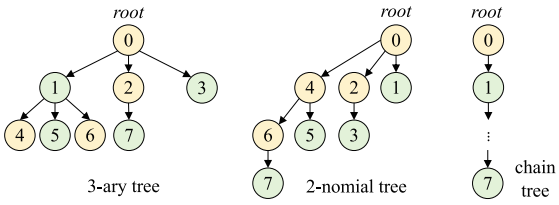

Fig. 2. Notification trees: $p = 8, root = 0$.

At communicator creation time the root process of `MPI_Bcast` operation is unknown. For this reason each process generates a fragment of a tree to all p possible values of root. A process stores information only about his parent and children nodes, it requires $O(p)$ bytes per process.

3.2 MPI_Bcast

The root process implements a pipelined message transferring. It divides the message into $\lceil m/f \rceil$ fragments and copies them through the queue in a shared memory. The root copies the current fragment *index* into the next available buffer in the queue $shm\_queue[root][index]$ and notifies children processes in the tree – updates their control blocks $shm\_controls[rank][index]$ with the current fragment size (Fig. 3). Non-root process *rank* waits on its control block until the value becomes positive, then notifies its children processes (propagates the

```
while sent_size < m do
  set = op % q;  op++
  wait_for(shm_nreaders[set] == 0)
  shm_nreaders[set] = p - 1
  shm_op[set] = op - 1
  i = set * (s / q);
  while i < (set + 1) * (s / q) and
        sent_size < m do
    // Copy to the queue
    frag = get_next_frag(m, sent_size)
    copy(frag, shm_queue[root][i],
         frag_size)
    write_memory_barrier()
    // Notify children
    for each child in children[root] do
      shm_controls[child][i] = frag_size
    end for
    sent_size += frag_size
    i++
  end while
end while
```

Fig. 3. Root process.

```
while sent_size < m do
  set = op % q
  wait_for(shm_op[set] == op)
  op++
  i = set * (s / q);
  while i < (set + 1) * (s / q) and
        sent_size < m do
    // Wait for a data
    wait_for(shm_controls[rank][i] > 0)
    frag_size = shm_controls[rank][i]
    shm_controls[rank][i] = 0
    // Notify children
    for each child in children[root] do
      shm_controls[child][i] = frag_size
    end for
    // Copy data from the queue
    frag = get_next_frag(m, sent_size)
    copy(shm_queue[root][i], frag,
         frag_size)
    sent_size += frag_size
    i++
  end while
  write_memory_barrier()
  atomic_dec(shm_nreaders[set])
end while
```

Fig. 4. Non-root process.

signal down the tree) and copies out the fragment from the root's queue to the output buffer (Fig. 4).

If the queue is full, the root process waits on the barrier until all processes have finished copying from the buffers ($shm\_nreaders = 0$). Non-root process starts to wait on control blocks only when its value of *op* counter is equal to the value of shared counter *shm_op*. The queue is divided into q sets to allow the non-blocking barrier to complete while another set (part of queue) is in use, reducing the synchronization costs [4]. For example, in the case of 12 fragments and the queue of $s = 8$ buffers is divided into $q = 2$ sets, the root process fills the first four buffers (the first set) and without blocking starts to copying data into next four buffers (the second set).

Proposed algorithms are implemented within the Open MPI code base (v4.0.x) as a separate collective component. `Wait_for` operation is implemented by spin waiting with periodic calling of the Open MPI's progress engine. The correctness of the page allocation from NUMA nodes is checked by the `move_pages()` linux system call.

4 Analysis of Algorithms

4.1 Theoretical Analysis

In general, the algorithm execution time is determined by the time of leaf processes in the notification tree. Figure 5 shows time diagrams for the root and leaf process in a flat tree. Let us consider three important cases.

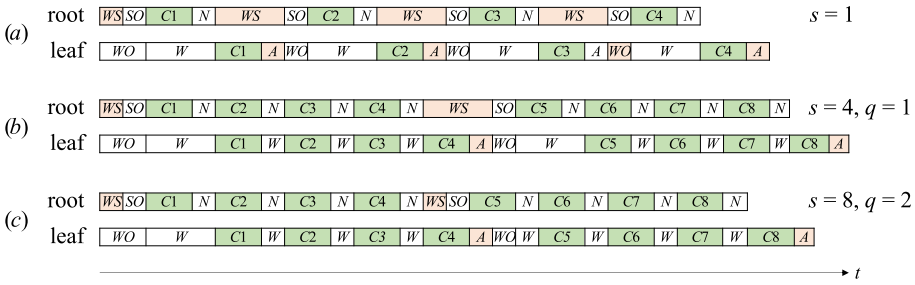

Fig. 5. Time diagrams of the algorithm (WS – waiting for a set in the root, SO – setting the *shm_op* counter, Ck – copying of the fragment k, N – notifying the child process, WO – waiting for a set in the leaf, W – waiting for a notification from the root, A – atomic decrement of *shm_nreaders* counter): *a*) single buffer queue ($s = 1$), $m = 4f$; *b*) queue of $s = 4$ buffers and one set ($q = 1$), $m = 8f$; *c*) queue of $s = 8$ buffers and two sets ($q = 2$), $m = 8f$.

A Single Buffer Queue. In the case of single buffer queue ($s = 1$) a pipelined message transmission is not possible (Fig. 5a). The root and leaf processes perform $\lceil m/f \rceil$ copies of fragments over the single shared buffer. On the first step the root process waits for the readiness of the set (WS) because a buffer may be occupied by non-root processes that complete the previous call of `MPI_Bcast`. After that, the root notifies child about beginning of operation (SO) and starts a loop of fragments copying. A leaf process waits for the readiness of the set (WO) for t_{WO} time and starts copying fragments from the root's queue. The leaf process receives notification for the readiness of the first buffer no sooner then the root copies it ($C1$) for a t_C time. To receive a notification from the root process (W), t_W units of time are required, it depends on the notification tree structure and the number of processes p. Thus, the copying of the first fragment by the leaf process is finished at the time $t_{WO} + t_C + t_W + t_C$. After copying, the leaf notifies the root with an atomic operation (A) for a t_A time about releasing of the set. The root process begins to re-fill the buffers. The time t_C of copying the fragment is mt, where t is the time for reading/writing one byte. Thus, the overall runtime of the algorithm is

$$t(m) = \lceil m/f \rceil (t_{WO} + t_A) + \lceil m/f \rceil t_W + 2mt. \quad (1)$$

A Queue of *s* Buffers. If the message size is larger than the buffer size, then the message is transferred in a pipeline mode ($m > f, s > 1$, Fig. 5b). The presence of s buffers allows the root process to copy s fragments to the queue without waiting. Copying of the fragment k by the root is performed simultaneously with copying of fragment $k-1$ by the leaf (child) process. After filling all s buffers the root process waits for the completion of copying by all processes, which requires at least $t_C + t_A$ time units (second step WS, Fig. 5b). A total number of barrier synchronizations (WO) is $\lceil m/b \rceil$, where $b = fs$ is the total queue size in bytes.

The overall runtime of the algorithm for the case of s buffers is

$$t(m,s) = \lceil m/b \rceil (t_{WO} + t_C + t_A) + \lceil m/f \rceil t_W + mt. \quad (2)$$

As a consequence of the expression (2), the total synchronization cost is s times less than in the case of a single buffer queue (2). Theoretically, at zero costs on waiting (WO, W, A) the queue of s buffers reduces the overall time by a factor of two: $t(m)/t(m,s) < 2$. In practice, the ratio may be greater because the waiting time t_W depends on the notification tree structure and the runtime is influenced by the process placement and copying from local/remote NUMA nodes.

A Queue Divided into q Sets. When the last buffer in the queue is used $(m \geq b)$, a barrier is initiated (WS) and the root process waits for the time $t_C + t_A$ for notifications from the child processes. The waiting cost can be reduced by time t_C if we split s buffers into q sets (Fig. 5c). This allows the root process to start filling the buffers of the next set while the child processes finish copying the fragments from the buffers of the previous set. The runtime of the algorithm:

$$t(m,s,q) = q\lceil m/b \rceil (t_{WO} + t_A) + t_C + \lceil m/f \rceil t_W + mt. \quad (3)$$

4.2 Experimental Results

Experiments were conducted on Intel Xeon Broadwell and Intel Xeon Nehalem dual-processor servers. Intel Xeon Broadwell server has two Intel Xeon E5-2620 v4 processor sockets (8 cores, HyperThreading disabled, L1 cache 32 KB, L2 256 KB, L3 20 MB) and 64 GB of RAM (2 NUMA nodes); linux 4.18.0-80.11.2.el8_0.x86_64, gcc 8.2.1. Intel Xeon Nehalem has two Intel Xeon E5620 processor sockets (4 cores, HyperThreading disabled, L1 256 KB cache, L2 1 MB, L3 12 MB) and 24 GB of RAM (2 NUMA nodes); linux kernel 4.16.3-301.x86_64, gcc 8.2.1.

The performance measurements were taken using Intel MPI Benchmarks (IMB 2019 Update 2). We run one rank per core. For all the figures we use time of the slowest process (`t_max`). An each experiment was run 5 times, discard the slowest and fastest runs, and we average the other 3. The IMB were run with parameters:

```
IMB-MPI1 Bcast -off_cache 20,64 -iter 5000,250 -msglog 6:24
     -sync 1 -imb_barrier 1 -root_shift 0 -zero_size 0
```

In our evaluation we used MVAPICH 2.3.2 and the Open MPI 4.0.x. Both libraries are built with optimizations (`CFLAGS=-O3 CXXFLAGS=-O3`). Figure 6 presents the performance comparison of different queue lengths s on Xeon Broadwell and Xeon Haswell servers. It shows normalized time to the single buffer queue $(s = 1)$. From the formula $t(m,s)$ it follows that the algorithm execution time depends linearly on the message size m, inversely proportional to the queue length s and its size $b = sf$ in bytes. If the messages size m is less then the buffer length f then pipelining is not possible. In Fig. 6 such situation is presented for messages less then buffer size (8 KB).

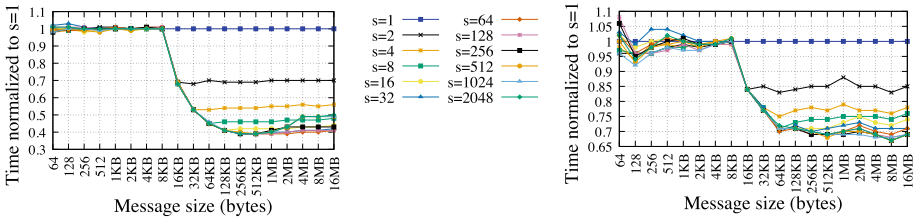

Fig. 6. Latencies of `MPI_Bcast` for different queue lengths s (time normalized to $s = 1$; one set $q = 1$, buffer size $f = 8192$, $p = 8$, one NUMA node, flat notification tree): *a*) Intel Xeon Broadwell server; *b*) Intel Xeon Haswell server.

For messages larger then 8 KB the root process fills s/q buffers without waiting of non-root processes. For example, queue of two buffers ($s = 2$) allows to transfer messages up to 16 KB without blocking of the root and it reduces the runtime by 30% relative to $s = 1$. Similarly, the queue of four buffer reduces the runtime by 40–46%. As noted above, synchronization costs do not allow to reach speedup by a factor of two. Also, the significant size of the shared memory segment limits the use of long queues. Experiments have shown that queues of 32–64 buffers provide good performance. Our results show that a binary or ternary notification trees provide, in most cases, the best performance.

4.3 Optimizing Queue Parameters

Let us find the size s of buffer and length s of the queue, what takes no more than b bytes and provides minimum algorithm time. For example, for a given MPI application it is necessary to determine the optimal configuration of the queue, which fits in 1% of the memory per core. We denote $t_C = tf$ and assume that m is divided by f without remainder. Let find the optimal value of s:

$$t(m,s) = \lceil m/b \rceil (t_{WO} + t_A) + \lceil m/b \rceil ft + m/f \cdot t_W + mt, \tag{4}$$

$$\frac{\partial t}{\partial f} = -mt_W/f^2 + \lceil m/b \rceil t = 0, \tag{5}$$

$$f^* = \sqrt{m/\lceil m/b \rceil \cdot t_W/t} \approx \sqrt{b \cdot t_W/t}, \qquad s^* = b/f^* = \sqrt{b \cdot t/t_W}. \tag{6}$$

In $t(m,s)$ two terms depend on f as f increases, the time $\lceil m/b \rceil ft$ also increases linearly, but the total time $\lceil m/f \rceil t_W$ decreases inversely with f. Figure 7 illustrates minimum point of $t(m,s)$ – intersection point of $\lceil m/b \rceil ft$ and $\lceil m/f \rceil t_W$. Figure 8 shows `MPI_Bcast` latency on Nehalem and Broadwell servers as the function of a fragment size f. The minimum time has been reached at the buffer sizes of 8 KB and 12 KB bytes, which corresponds to the obtained f^* and s^*.

Considering that $t_W > t$, it is practical to use buffers of size $f \geq \sqrt{b}$, rounded up to the nearest multiple of a page size. Consider the choice of the queue parameters for different cases.

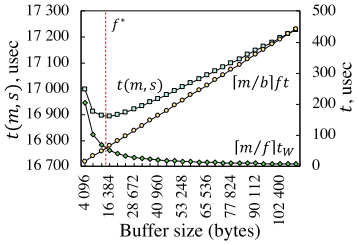

Fig. 7. Algorithm runtime (model, left axis) and terms $\lceil m/b \rceil ft$, $\lceil m/f \rceil t_W$ (right axis): $m = 16\,\text{MB}$, $b = 4\,\text{MB}$, $s = b/f$, $q = 1$, $t = 10^{-9}\,\text{s}$, $t_{WO} = 100t$, $t_A = 10t$, $t_W = 50t$).

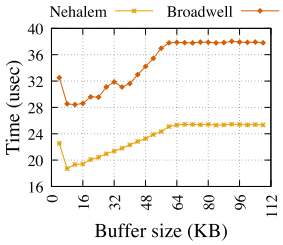

Fig. 8. Latency of `MPI_Bcast`: $m = 64\,\text{KB}$, $b = 4\,\text{MB}$, $s = b/f$, $q = 1$, chain tree, Nehalem ($p = 8$), Broadwell ($p = 16$).

1. The message size m is known or the upper bound for it (for example after application profiling). The best choice is to get f and s such that $f < m \leq fs$. Let be $f = \sqrt{m}$ and rounds up obtained f to the nearest multiply of a page size; $s = m/f$.
2. The buffer size f is given, we need to find the queue length s. Let m_{max} denote the upper bound of message size, then $s = \lceil m_{max}/f \rceil$.
3. The queue length s is given, we need to get the buffer size f. Let $b = m_{max}$ and apply (6): $f = \sqrt{m_{max}}$.

4.4 NUMA-Aware Queues Placement

Much of the algorithms time is spent copying from the input buffer into root's queue. For this reason it is important to store the input buffer and root's queue on a same NUMA node. In our algorithms we temporarily disable sequential readahead by calling `madvise(seg, segsize, MADV_RANDOM)`.

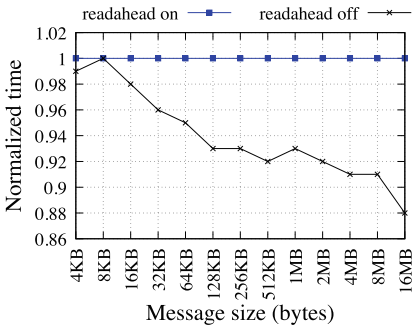

Fig. 9. `MPI_Bcast` normalized time: $s = 64$, $f = 8192$, $q = 1$, binary tree, $p = 16$, two NUMA nodes Xeon Broadwell, IMB `-root_shift on`.

This ensures correct allocation of memory pages from local NUMA nodes within the first touch policy.

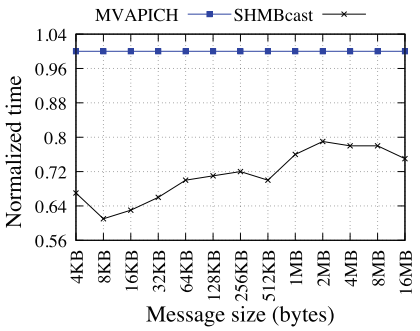

Fig. 10. Normalized time of the developed algorithm (SHMBcast) and MVAPICH ($f = 8192, s = 128$): $s = 64$, $f = 8192$, $q = 1$, binary tree, $p = 8$, two NUMA nodes of Intel Xeon Nehalem, IMB `-root_shift on`.

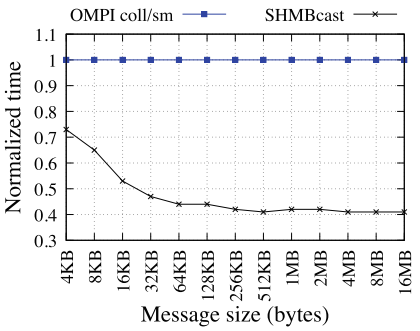

Fig. 11. Normalized time of the developed algorithm (SHMBcast) and Open MPI `coll/sm` ($f = 8192, s = 8$): $s = 64$, $f = 8192$, $q = 1$, binary tree, $p = 16$, two NUMA nodes of Intel Xeon Broadwell, IMB `-root_shift on`.

Figure 9 shows the proposed algorithm's runtime in the "readahead on" mode and without it ("readahead off", `madvise(MADV_RANDOM)`). To estimate overhead due to remote NUMA node access we have run IMB with `-root_shift on` option to cyclically change root on each iteration of measurements. Clearly, as the message size increases, the time for copying fragments to remote NUMA node also increases. Our approach with explicit placement of queues on NUMA nodes (readahead off) allows to reduce inter-socket communications. Similarly, MVAPICH allocates memory for the queues without taking into account a topology of NUMA node (Fig. 10). The Open MPI `coll/sm` algorithm implements a partial binding of buffers to the NUMA nodes, but it is influenced by the readahead subsystem and allocates significant amounts of pages from the NUMA node of the process 0 (Fig. 11). Our algorithms SHMBcast achieve on average 20–40% speedup over Open MPI `coll/sm` and 20–60% over MVAPICH.

5 Conclusion

In this paper we have examined the benefits of NUMA-aware placing of shared queues for optimizing `MPI_Bcast` operation. Proposed algorithms use k-ary, k-nomial, chain and flat trees to propagate notifications from the root process to others. On a Xeon Nehalem and Xeon Broadwell servers, our implementation achieves on average 20–60% speedup over algorithms of Open MPI `coll/sm`

and MVAPICH. We find that a binary or ternary trees provides, in most cases, the best performance.

The same approach could be used to optimize other algorithms of collective operations. Future work will include the use of huge memory pages and optimizing of zero-copy approach to the MPI derived datatypes. Also, we plan to conduct experiments on new platforms and architectures (AMD EPYC, Intel Skylake-SP with UPI and Sub-NUMA Clusters, ARMv8).

References

1. Li, S., Hoefler, T., Snir, M.: NUMA-aware shared memory collective communication for MPI. In: Proceedings of the International Symposium on High-Performance Parallel and Distributed computing, pp. 85–96 (2013)
2. Wu, M., Kendall, R., Aluru, S.: Exploring collective communications on a cluster of SMPs. In: Proceeedings of the HPCAsia, pp. 114–117 (2004)
3. MVAPICH: MPI over InfiniBand, Omni-Path, Ethernet/iWARP, and RoCE. http://mvapich.cse.ohio-state.edu/
4. Graham, R.L., Shipman, G.: MPI support for multi-core architectures: optimized shared memory collectives. In: Lastovetsky, A., Kechadi, T., Dongarra, J. (eds.) EuroPVM/MPI 2008. LNCS, vol. 5205, pp. 130–140. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87475-1_21
5. Jain, S., et al.: Framework for scalable intra-node collective operations using shared memory. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 2018), pp. 374–385 (2018)
6. Ma, T., Herault, T., Bosilca, G., Dongarra, J.J.: Process distance-aware adaptive MPI collective communications. In: Proceedings of the 2011 IEEE International Conference on Cluster Computing, pp. 196–204 (2011)
7. Bienz, A., Olson, L., Gropp, W.: Node-aware improvements to allreduce. In: Proceedings of ExaMPI 2019: Workshop on Exascale MPI (SC 2019), pp. 19–28 (2019)
8. Li, S., Hoefler, T., Hu, C., Snir, M.: Improved MPI collectives for MPI processes in shared address spaces. Cluster Comput. **17**(4), 1139–1155 (2014). https://doi.org/10.1007/s10586-014-0361-4
9. Graham, R., et al.: Cheetah: a framework for scalable hierarchical collective operations. In: Proceedings of IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pp. 73–83 (2011)
10. Chakraborty, S., Subramoni, H., Panda, D.K.: Contention-aware kernel-assisted MPI collectives for multi-/many-core systems. In: Proceedings of IEEE International Conference on Cluster Computing, pp. 13–24 (2017)
11. Luo, X., Wu, W., Bosilca, G., Patinyasakdikul, T., Wang, L., Dongarra, J.J.: ADAPT: an event-based adaptive collective communication framework. In: Proceedings of International Symposium on High-Performance Parallel and Distributed Computing, pp. 118–130 (2018)
12. Träff, J.L., Rougier, A.: MPI collectives and datatypes for hierarchical all-to-all communication. In: Proceedings of EuroMPI/ASIA, pp. 27–32 (2014)
13. Goglin, B., Moreaud, S.: KNEM: a generic and scalable kernel-assisted intra-node MPI communication framework. J. Parallel Distrib. Comput. **73**(2), 176–188 (2013)

14. Ma, T., Bosilca, G., Bouteiller, A., Dongarra, J.: HierKNEM: an adaptive framework for kernel-assisted and topology-aware collective communications on many-core clusters. In: Proceedings of Parallel and Distributed Processing Symposium, pp. 970–982 (2012)
15. Tu, B., Zou, M., Zhan, J., Zhao, X., Fan, J.: Multi-core aware optimization for MPI collectives. In: Proceedings of International Conference on Cluster Computing, pp. 322–325 (2008)