Scalable PGAS Collective Operations in NUMA Clusters

Damián A. Mallón · Guillermo L. Taboada · Carlos Teijeiro · Jorge González-Domínguez · Andrés Gómez · Brian Wibecan

Abstract The increasing number of cores per processor is turning manycore-based systems in pervasive. This involves dealing with multiple levels of memory in NUMA (Non Uniform Memory Access) systems and processor cores hierarchies, accessible via complex interconnects in order to dispatch the increasing amount of data required by the processing elements. The key for efficient and scalable provision of data is the use of collective communication operations that minimize the impact of bottlenecks. Leveraging one sided communications becomes more important in these systems, to avoid unnecessary synchronization between pairs of processes in collective operations implemented in terms of two sided point to point functions. This work proposes a series of algorithms that provide a good performance and scalability in collective operations, based on the use of hierarchical trees, overlapping one-sided communications, message pipelining and the available NUMA binding features. An implementation has been developed for Unified Parallel C (UPC), a PGAS (Partitioned Global Address Space) language, which presents a shared memory view across the nodes for programmability, while keeping private memory regions for performance. The performance evaluation of the proposed implementation, conducted on five representative systems (JuRoPA, JUDGE, Finis Terrae, SVG and Superdome), has shown generally good performance and scalability, even outperforming MPI in some cases, which confirms the suitability of the developed algorithms for manycore architectures.

Keywords Manycore Architectures · Collective Operations · NUMA · UPC · PGAS · MPI · High Performance Computing · Communication Algorithms

Damián A. Mallón
Forschungzentrum Jülich. E-mail: d.alvarez.mallon@fz-juelich.de

Guillermo L. Taboada, Carlos Teijeiro, Jorge González-Domínguez
University of A Coruña. E-mail: {taboada,cteijeiro,jgonzalezd}@udc.es

Andrés Gómez
Galicia Supercomputing Center. E-mail: agomez@cesga.es

Brian Wibecan
Hewlett-Packard. E-mail: brian.wibecan@hp.com

1 Introduction

Current computer systems are based on multicore chips, which are constantly increasing their number of cores. This scenario, and particularly the new manycore processor architectures, heightens the importance of memory performance and scalability. The inclusion of the memory controller inside the processor's chip helps to minimize the issues associated with the access to shared memory from manycore chips. As a result, currently most systems, both single socket and multi-socket, have NUMA (Non Uniform Memory Access) architectures, as now processors package several memory controllers in a single chip. NUMA architectures provide scalability through the replication of paths to main memory, reducing the memory bus congestion as long as the accesses are evenly distributed among all the memory modules. This scalability is key for applications running on thousands of cores. The contributions presented in this paper, a series of new collective operations algorithms for NUMA architectures, target effectively the need for hardware awareness to extract the maximum performance.

Previous works on optimizing collective communication operations for clusters of nodes with multicore processors and SMP systems focused on taking advantage of intra-node shared memory transfers. Additionally, scalable collective algorithms are usually based on the use of process-based trees, usually disregarding their particular location in the system. Therefore, there is still room for locality exploitation. Moreover, the trees used for transferring data are usually computed every time the collective routine is invoked, as the source of the data can vary from one call to another. While the overhead of this tree computation is reduced for small setups, for large systems can limit the scalability of the collective operations. Furthermore, the presence of multiple NUMA regions and multicore/manycore processors within each cluster node poses new challenges, such as taking advantage of data locality and implementing efficient data transfers in these multilevel, hierarchical and complex architectures.

This work presents a new, scalable and efficient algorithm which computes statical trees at initialization time, thus improving scalability, especially on large systems. These trees consist of subtrees that map the different hierarchical levels present in clusters of NUMA nodes: (1) cluster of network interconnected nodes, (2) NUMA regions within a cluster node, and (3) cores within NUMA regions. This algorithm has been implemented in Unified Parallel C (UPC), a representative PGAS (Partitioned Global Address Space) language, due to its suitability for programming clusters of manycore nodes. In fact, PGAS UPC is increasing its popularity as it provides a shared memory view of all resources for programmability while exploiting local memory for performance. Four representative collective operations (broadcast, reduce, gather and scatter) have been compared in terms of performance against current collectives libraries, both UPC and MPI.

The rest of this paper is organized as follows. Section 2 discusses the related work. Section 3 presents the proposed algorithm and its realization for broadcast, reduce, gather and scatter operations on clusters of NUMA nodes. Section 4 analyzes the performance evaluation of the developed collective operations on five representative systems, assessing their performance against its industry leaders counterparts. Finally, Section 5 concludes the paper.

2 Related Work

The optimization of middleware for High Performance Computing is a complex task. It might involve all the layers of the runtime, and it should evolve with new versions of the language and library APIs [11, 35]. Within the runtime optimization, one of the most important points is the optimization of the collective operations, as most applications rely on them, both for programmability, as they implement popular operations, relieving the programmer from its error prone implementation, and also for performance, as they generally implement optimized and refined algorithms. Collective operations are usually key to achieve a good scalability. There are basically two approaches for the optimization of collective operations: the algorithmic and the system approach.

The algorithmic approach focuses on how the data is transferred and how the processes are organized. Not all algorithms can be suitable or implemented for all systems. Previous works on this field can be split between two main groups: distributed memory algorithms and shared memory algorithms, which target the main current architectures.

Typically, distributed memory algorithms for clusters of multicore nodes do not take into account the potential performance benefits of NUMA awareness. Kandalla et al. [13] developed and tested a topology-aware algorithm that builds the interconnect tree taking into account the process placement in relation to the switches, avoiding unnecessary switch hops. Bibo Tu et al. [31] described a new broadcast algorithm for multicore clusters. In this algorithm two sets of communicators were used. The first one for intra-node communications, where binding is used to improve locality within a node. The second one is for inter-node communications. This way a broadcast is performed in two steps, inter and intra-node transfer steps, avoiding the network interface congestion. Kumar et al. [15] designed and evaluated an all-to-all algorithm for multicore clusters. This algorithm is similar to the Bibo Tu's broadcast algorithm, in the sense that is performed in two steps. While these approaches yield important benefits for clusters of nodes with a small number of cores per processor, they present some limitations for clusters with tens of cores per node, ignoring the current NUMA architectures. Chan et al. [4] proposed an algorithm that takes advantage of architectures with multiple links, where messages can be sent simultaneously over different links in systems with N-dimensional meshes/tori.

Other works focused on optimizations for shared memory. Nishtala et al. [22] conducted a series of experiments in three shared memory systems, based on multicore processors, using k-nomial trees for representing the virtual topology of the processes. These experiments demonstrated that for each architecture and message size the optimal radix of the k-nomial trees is different. Graham et al. [7] designed and tested a series of algorithms for shared memory, each one appropriate for a set of functions and message sizes. The described algorithms are basically fan-in or fan-out trees of variable radix; reduce-scatter (each process reduces its data) followed by a gather or all-gather; and a recursive doubling algorithm. As expected, every algorithm is the best performer for some setups, whereas not optimal for others.

Additionally, there are some works that aimed to optimize both shared and distributed memory architectures, such as the work of Mamidala et al. [19], which implements and evaluates similar algorithms to the previous works. A work more closer to ours is that of Kandalla et al. [12], which proposed a multi-leader algorithm. This proposal is similar to Bibo Tu's broadcast algorithm, except for using more than one leader per node, initially considering only the allgather operation. Nishtala et al. [23] leveraged shared memory and trees to optimize collectives and explore their autotuning possibilities. Qian [24] followed a similar path to Kandalla et al. and proposed a series of algorithms mainly focused on all-to-all and allgather, targeting multicore systems with multiple connections per node, as well as optimizing the algorithms for cases where different processes arrive at the collective at a different time.

Other optimizations can be categorized in the algorithmic approach, but without targeting specifically distributed or shared memory systems. Rabenseifner [25] proposed an algorithm for reduce operations in MPI that is widely used in many implementations. It is performed in two steps: a reduce-scatter and a gather, and yields increased performance in cases where the size of the arrays to be reduced are big enough.

The algorithms considered in the related work are usually independent of their actual implementation in a particular language. However, they have been generally developed using MPI or UPC, two of the most popular choices nowadays. They are representative of message-passing and PGAS solutions, respectively. However, usually there is no algorithm that always outperforms the others. In fact, the performance of an algorithm depends on three factors: (1) message size, (2) number of processes involved, and (3) the hardware. Providing the best algorithm for each setup and message size is the optimum approach, as demonstrated in [28]. However, selecting among the algorithms entails a significant effort, as they are highly dependent on the system. The solution typically relies on autotuning [32], generally based on an automatic performance analysis of each algorithm for a wide range of setups.

Furthermore, it is possible to adapt the runtimes to the underlying hardware. Two main approaches can be easily distinguished: the first one adds software features in order to achieve a better usage of the underlying hardware; whereas the second one adapts the corresponding layer to a given architecture.

Following the first approach, Miao et al. [20] proposed a single copying method to take advantage of shared memory architectures, avoiding the system buffer. The proposal of Trahay et al. relies on a multithreaded communication engine to offload small messages [30]. Brightwell et al. [3] propose the sharing of page tables between processes, speeding up applications performance. Hoefler et al. [8] proposed the use of multicast in networks, resulting in highly scalable operations, but just valid for very small messages. More recently, Li et al. [16] have proposed a NUMA-aware multithreaded MPI runtime, where MPI ranks are implemented as threads as opposed to processes, and they have implemented and evaluated algorithms for allreduce in this runtime.

The second approach generally consists of the design of custom algorithms for a given architecture. Following this approach, Velamati et al. [33] designed a set of algorithms for MPI collective operations for the heterogeneous Cell processor.

Previous works adapted collective operations algorithms and runtimes to several architectures. However, few of them took into account clusters of nodes with multicore processors with NUMA features. These features are increasingly important, as even some single-socket servers based on recent processors are NUMA systems [1, 5]. Therefore, thread/process binding is increasingly important, as well as a key feature for maximizing performance [2]. The work described in the present paper optimizes collective operations for NUMA clusters in three ways: (1) the use of one-sided communications to leverage asynchronicity and pipelining with full-bidirectional bandwidth; (2) the use of NUMA binding to take advantage of the locality; (3) the use of hierarchically-fixed trees, computed at initialization time. Pipelining is a well-known technique implemented in many MPI libraries, and has been proposed before for UPC [26]. However, to the best of our knowledge, it has never been implemented and tested.

3 Scalable Algorithm for Collective Operations on NUMA Clusters

Scalability is a pervasive and complex problem in HPC. Sometimes an algorithm presents easy development and good performance but it might not scale well with hundreds or thousands of cores, hindering the use of today's HPC systems at their full power. The key issue is the use of highly scalable methods, even though they might be more complex. This kind of methods should be the preferred choice to face up large scale problems, as demonstrated in [21], whereas less complex algorithms with good performance are acceptable for small or medium scale setups. This principle is valid both for applications and collective operations libraries. The algorithm presented in this work aims at scalable performance on hundreds or thousands of cores on NUMA clusters rather than providing efficiency on small/medium scale setups.

One of the design principles for scalability is to avoid the use of dynamic structures whose size and build time overhead increases with the number of processes. Thus, in the proposed algorithm the first call to a collective function creates a persistent and fixed (invariable) process tree structure, which can be reused in a future collective call. If the root of the collective operation and the root of the tree are not the same, then a copy of the message into the tree root is required, in top-down operations like scatter or broadcast, or the copying from the tree root to the operation root, in bottom-up operations like reduce or gather. This approach has as main benefit the reuse of the tree structures through all the run time. However, for a reduced number of processes it is still faster building a custom tree than reusing a structure, if the root of the collective operation is not the root of the precomputed tree.

Traditionally, most efficient collective implementations use trees of processes to distribute or gather the data, although generally regardless the processes placement. Only some advanced solutions implement topology or multicore aware trees [12, 13, 15, 31]. The algorithm presented in this paper extends these approaches to NUMA clusters, taking into account the NUMA topology. Therefore, the trees used will be decomposed in three levels of subtrees: (1) the cluster level, (2) the node level, and (3) the NUMA region level.

In the cluster level the nodes are interconnected through a network . For each node one process is selected as a node leader, in charge of communicating its node with the other nodes. Under certain circumstances it would be desirable to have multiple node leaders (for instance, in systems with more than one network interface per node). However, given the characteristics of the machines analyzed in this work, the number of node leaders is one. For efficiency and scalability purposes, a binomial tree of node leaders will be built.

The node level comprises the NUMA regions available in a node. The consideration of this level is one of the contributions of this work. For each NUMA region one process is selected as NUMA region leader, and a binomial tree of NUMA region leaders will be built, with the node leader as root of its node tree. This level leader will be responsible for the communication of its children and its father, that could be the node leader or another NUMA region leader in systems with multiple NUMA regions. The tree used for this level is also a binomial tree.

The NUMA region level connects its elements through shared memory. In this level a new tree will be built, with the NUMA region leader as root. Here processes are attached to the NUMA region through binding, thus avoiding process migration to another NUMA region. Using NUMA region binding rather core binding is advisable, as it allows the operating system to move processes within the NUMA region if necessary. However, in some architectures the cache or bus sharing can have a significant impact [17]. Therefore, in these architectures a per core binding could be a better choice. In this level two types of trees have been implemented and tested: binomial and flat trees. In the proposed algorithm there is no reason to avoid a process from being leader at several levels. In fact, it is advisable. Therefore, the node leader has been made also its NUMA region leader.

Figure 1 illustrates the proposed structure for the algorithm using an example which consists of 8 nodes, each one of them with 24 cores distributed between 4 NUMA regions. The proposed algorithm will be able to distribute effectively and efficiently the data transfers among processes, taking advantage of the increased locality at the same time, as it minimizes the usage of the most costly links, using the fastest data channels whenever is possible, taking the most out of runtimes' shared memory optimizations. In fact, even runtimes without shared memory optimizations can get an extra benefit. For instance a UPC runtime without shared memory optimizations but support for privatizability functions [34] can map page tables from other processes into its own memory space, allowing the use of the much faster `memcpy` system call.

The reason for using binomial trees instead of binary trees is their reduced number of steps needed to traverse them in setups with large number of nodes. Binomial trees will complete a 1-to-N or N-to-1 operation (broadcast, gather, scatter, reduce) in a $\lceil log_2(N) \rceil$ number of steps, for an N number of nodes. That is considering that the communication starts towards the deepest branch, and that the communication is done with one connected node at a time (sibling nodes can not communicate at the same time with their parent). Binary trees on the other hand, will complete the operation in $(\lceil log_2(N) \rceil - 1) * 2 - 1$ number of steps in the best case, or $(\lceil log_2(N) \rceil - 1) * 2$ in

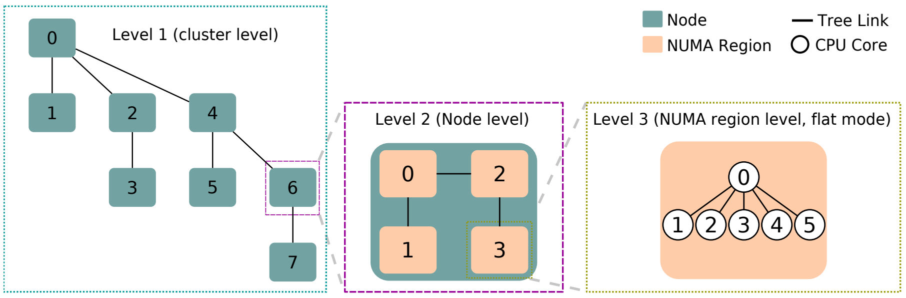


Fig. 1 General overview of the scalable algorithm for collective operations on NUMA clusters

the worst case, for $N > 2$. For 16 processes the binomial tree will finish in 4 steps, whereas the binary tree will finish in 6. For 4096 nodes the difference is 12 vs. 22.

Therefore, binomial trees are a better choice for scalable communications. However, it shall be noted that binary trees can be also a valid option when communication between nodes in the tree can be non-blocking and/or one-sided. In these cases/scenarios communications can be overlapped, making the time required to communicate with all the children nodes in a lower level close to the time required to communicate with just one node. In that case the operation will be completed in a maximum of $(\lceil log_2(N)\rceil - 1)$ number of steps. This is true when both transfers can be done simultaneously without mutual interference, which is usually not possible, and is highly dependent on the bandwidth and the message rate that the network adapter can handle. Moreover, if that scenario is possible, binomial trees will also finish in a $(\lceil log_2(N)\rceil - 1)$ number of steps, which makes considering binary trees impractical in most situations.

Flat trees do not scale, as they saturate the sender or receiver (depending on the operation) easily. However, for a small number of nodes in the tree, a flat tree avoids intermediate steps, reducing the synchronization overhead. Since the number of cores per NUMA region is not likely to increase significantly, to avoid the memory bus saturation, the library presented in this work also evaluates the use of flat trees in the intra NUMA level (the core level).

Another feature present in high speed network fabrics is the presence of separate links for upload and download data. Bearing that in mind, it is possible to pipeline communications, overlapping send and receive operations to reduce latency. This library implements two modes of message fragmentation for pipelining: fixed and dynamic. In the fixed mode the message is fragmented into chunks of a given size. This way, when one chunk is received, the destination process is able to send that data while it is receiving the next chunk. This operation goes on until the complete message has been delivered. The dynamic mode is similar to the fixed mode, except that it splits the message in two halves, instead of $\lceil message\_size/chunk\_size\rceil$ messages. The selected chunk size for the fixed mode is 32768 bytes. The dynamic mode will start fragmenting the messages when they are larger than 8192 bytes.

This library also takes advantage of one-sided memory copies, implementing most functions in two approaches: *push* and *pull*. In the push approach the source process puts the data in the destination process, whereas in the pull mode it is the destination process the one which gets the data. This way it is possible to achieve a higher degree of communication overlapping, since data is streamed to/from different sources at the same time.

Finally, the correctness of the developed functions has been assessed with GUTS (GWU Unified Testing Suite) [6].

3.1 Scalable Reduce Operations

Reduce in UPC is different than reduce in MPI. In UPC, all the values of a shared array are reduced to a single element, as opposed to MPI, where the result is an array, with reduced values for every array position. Therefore, the developed algorithms do not conform the definition of the reduce operation in MPI, and no comparison between the two will be made.

The developed reduce function is based on several design principles looking for scalability. The first one is that each process performs the reduction of its own data. Therefore, the data communication is restricted to communicating a single element of a primitive data type, that is to say, from a minimum size of `char` and a maximum size of `long double`. Due to this, no fragmentation occurs.

The second consideration is motivated by the fact that a process might not know if its children participate in the reduction. To solve this issue, each process with a passive participation will contribute to the operation with a neutral operand value (e.g., 0 for add operations and 1 for multiplications). In case the user defines its own operation then this value must be adapted.

The third reduce design principle is a consequence of the tree used. Thus, for noncommutative operations (such as noncommutative operations defined by the user, `UPC_NONCOMM_ FUNC`), the operations must take the order into account, otherwise they will provide an erroneous result.

The use of binomial trees not mapped into the hardware supports the two first design principles. However, this would neglect the benefits of hierarchical trees.

3.2 Scalable Scatter and Gather Operations

Scatter and gather operations have a particularity. In these functions the source or destination of the data, respectively, is a single process, while the other processes receive or send, respectively, their specific chunk of data. Therefore, the data movement can not be optimized in the same way as for broadcast, since each process holds a different chunk of data. For this reason, trees are not an appropriate structure for data distribution in these functions. However, a collective using trees avoids the overhead of each process copying data separately, since less copies from/to source/destination will be done. Such collectives seem interesting in scenarios where the data hold by each process is not excessive. Besides this, having a scatter or gather function that uses trees has an additional benefit. Since just a few processes will communicate with the root of the operation, the memory footprint will be smaller in some systems, leading to a higher scalability. In high-speed cluster networks, such as InfiniBand, a small buffer is used for each peer connection. In jobs with thousands of processes this becomes a big problem as it has been pointed out before in several works [14, 27]. Mitigating this effect usually involves deep changes in the communication layer of the runtime or the transport layer. Shared Receive Queues (SRQ) and eXtended Reliable Connection (XRC) are recent InfiniBand features that allow to minimize the memory overhead in large setups.The library presented in this paper and a runtime/driver with on-demand approach, where buffers are allocated as needed instead of at initialization, help to solve this problem for scatter and gather at a higher level than runtime or transport layer modifications.

The trees aforementioned take advantage of the underlying hardware and memory hierarchy. In scatter and gather operations in order to move data efficiently downward or upward the tree, the processes have to be contiguous within a given branch. This cannot be guaranteed, as the user can choose a cyclic process distribution among nodes. One possible workaround is having each level root being aware of all the processes (and their order) hanging in all their branches. However, this workaround has two major issues that leads it to do not scale. The first issue is that the tree root would need to store too much information about the tree. The second and more important issue is that root processes would have to perform multiple out-of-order memory copies, instead of a single big memory copy. The overheads of these two issues make this workaround impractical.

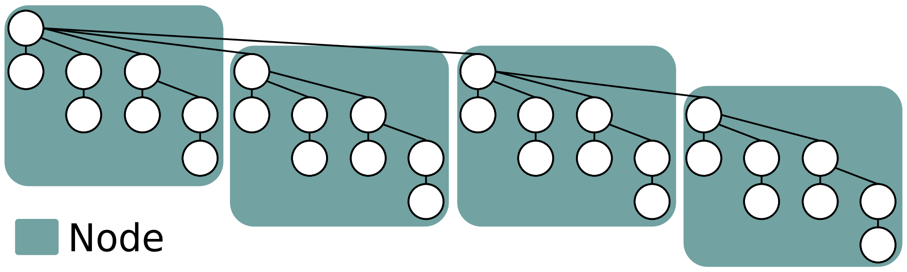


Fig. 2 Example of a binomial tree structure for a 32 process configuration, distributed among 4 nodes in a block fashion. This tree structure is used in Scatter and Gather operations, and ignores the process distribution and memory hierarchies.

A trade-off solution is building a binomial tree with all the processes, ignoring their distribution among nodes and NUMA regions. Figure 2 presents this approach in an ideal case, with a block distribution and a power of 2 number of processes in the nodes and in the NUMA regions. In this case the processes will map perfectly onto the hardware, minimizing the use of the links with more latency.

Scatter and gather functions with trees have also some other particularities. They use intermediate buffers to copy data. The buffer management code is performed before the initial barriers (if `UPC_IN_ALLSYNC` or `UPC_IN_MYSYNC` are set) and the ending barriers (if `UPC_OUT_ALLSYNC` or `UPC_OUT_MYSYNC` are set). The buffers are not freed if the new call has the same or fewer buffer requirements than previous calls. However, if a certain call needs a buffer size of more than a certain threshold (currently $16MB$), the buffer will be freed at the end of the function to avoid excessive memory usage.

Another particularity is that process 0's buffers will be the source or the destination (in gather or scatter, respectively), if the process 0 is the root of the operation.

The last particularity is that, even though these functions do not take advantage of the processes' distribution and trees do not map onto the hardware, they are bound to the corresponding NUMA region, since this step is performed when the library is initialized, at the beginning of each runtime execution.

The described particularities up to now are for both scatter and gather. Gather has an extra particularity. It does not have a dynamic fragmentation version. The reason for this is that, since the data flows upwards, copying the first half do not make sense in most situations. The parent process could not take advantage of it, since its own first half will be larger than any of its sons' first half. Therefore, data can not flow in halves because parent processes would have to wait for the second half anyway before sending the first half.

Lastly, an additional algorithm has been implemented for scatter and gather. The tree structure is not appropriate when the chunks of data are too big. A token-passing algorithm has been developed for those cases. The token is passed to the next process, in a ring fashion. The process with the token starts copying data. This way the algorithm prevents that all process access at the same time, saturating the network. The token is passed to the next process when one of the following conditions are met: (1) the current process is in the same node as the source/destination process, before start copying, to allow overlapping using the fast memory subsystem; (2) the data to be copied is smaller than a given threshold, to avoid the following processes wait unnecessarily; (3) when the remaining data to be transferred is smaller than the previous threshold. When the data is bigger than this threshold the copy is performed in two phases, the first one with a size $N - Threshold$ and the second one with a size $Threshold$. Since the bottleneck of scatter and gather operations is the outbound link from the source thread, the presented implementation operates with a single token, assuming that a single thread can saturate the network or, when this condition is not met, passing the token before initiating the copy, to allow overlapping of copies to/from different threads. Operations with very small messages and a large number of threads can benefit from the tree algorithm or use a ring algorithm with multiple tokens. Evaluating the use of multiple tokens has not considered in this work, as the evaluated systems are not big enough for taking advantage of it.

3.3 Summary of the Implemented Algorithms

The number of variations of the developed algorithms is a result of combining different orthogonal optimizations suited for the operations. The Table 1 presents an overview of the developed algorithms. It should be noted that, even though these algorithms have been implemented in UPC, the underlying principle and optimizations are also valid for MPI. However, UPC, and more generally any PGAS language, allows to implement them in a more natural way, since one-sided communication is a

Table 1 Algorithms implemented.

			Operations			
			Broadcast	Reduce	Scatter	Gather
Push	Ring				✓	✓
	Hierarchical binomial	Standard	✓	✓	✓	✓
		Dynamic fragmentation	✓			
		Static fragmentation	✓			✓
	Hierarchical binomial+flat	Standard	✓	✓		
		Dynamic fragmentation	✓			
		Static fragmentation	✓			
Pull	Ring				✓	✓
	Hierarchical binomial	Standard	✓	✓	✓	✓
		Dynamic fragmentation	✓		✓	
		Static fragmentation	✓		✓	
	Hierarchical binomial+flat	Standard	✓	✓		
		Dynamic fragmentation	✓			
		Static fragmentation	✓			

main feature of the language. In MPI, put and get operations, and their non-blocking counterparts, require explicit memory and window management, as opposed to UPC.

4 Performance Evaluation

This section presents a performance evaluation of the proposed algorithms implemented in UPC. This language implements the PGAS programming model, quite suitable for current and upcoming hybrid shared/distributed memory architectures, especially its one-sided features. This development is a portable library as it is based on standard UPC operations, using `upc_mem{cpy,put,get}` operations. However, some algorithms require three extended UPC features: (1) UPC semaphores, (2) privatizability functions, and (3) asynchronous and non-blocking functions. Even though this features are not part of the standard, implementors have expressed they willingness to add then, which makes this approach the most portable (as opposed to implement it directly in GASNet).

4.1 Experimental Configuration

The performance evaluation of this work has been carried out on five representative systems. The first one is the Finis Terrae supercomputer (at Galicia Supercomputing Center, CESGA) [29], composed of 142 HP Integrity rx7640 nodes, each one with 8 Montvale Itanium 2 dual-core processors (16 cores per node) at 1.6 GHz and 128 GB of memory. The processors are distributed between 2 cells, each one with 4 processors (8 cores) and its own I/O subsystem. Each cell is an independent NUMA region. The interconnection network is InfiniBand 4X DDR (16 Gbps of theoretical effective bandwidth), with Mellanox InfiniHost III Ex HCAs and a Voltaire Grid Director ISR 2012 switch. The HCAs are plugged in the cell 0. The node root thread is bound to this NUMA region (cell 0 NUMA region) to that NUMA region. The library has been tested with up to 1024 cores in this system. The number of nodes used in the performance evaluation is $\lceil n/16 \rceil$, being n the total number of cores used. The UPC compiler and runtime is Berkeley UPC 2.12.1, relaying on the effective the InfiniBand Verbs library for distributed memory communication. The GASNet PSHM (GASNet inter-Process SHared Memory) optimization has been enabled. The backend C compiler available in the system is the Intel 11.1.

The second system is an HP Integrity Superdome at CESGA, with 64 Montvale Itanium 2 dual-core processors (128 cores total) at 1.6 GHz and 1 TB of memory. The processors are distributed between 16 NUMA regions, each one with 4 processors (8 cores). The library has been tested with up to 128 cores in this system. The UPC compiler and runtime is Berkeley UPC 2.12.1, with the SMP conduit, which uses shared memory constructs for communications. The backend C compiler is the Intel 11.1.

The third system is the SVG 2011 (Galician Virtual Supercomputer) at CESGA, composed of 46 HP ProLiant SL 165z G7 nodes, each one with two 12-core AMD Opteron 6174 Magny-Cours processors (hence 24 cores per node) at 2.2 GHz, and 32 or 64 GB of memory. Each processor has 2 memory controllers. Therefore each node has 4 NUMA regions, connected through high-speed HyperTransport links. The interconnection network is Gigabit Ethernet, with Intel 82576 cards. There are two interfaces bonded together. The bonding mode is 0 (round-robin balancing). The ethernet switches are HP ProCurve 2910al. The library has been tested with up to 192 cores in this system. The number of nodes used in the performance evaluation is $\lceil n/24 \rceil$, being n the total number of cores used. The UPC compiler and runtime is Berkeley UPC 2.12.1, relying on the MPI conduit for distributed memory communication. Therefore the remote memory operations are built on top of MPI. In this particular testbed the implementation used is MPICH 1.3.2. The GASNet PSHM optimization has been also enabled in this system. The backend C compiler available in the system is the Open64 4.2.4.

The fourth system used in the evaluation is the JUDGE (Jülich Dedicated GPU Environment) supercomputer, comprised of 206 IBM System x iDataPlex dx360 M3 nodes, each one with two 6-core Intel Xeon X5650 Westmere processors (hence 12 cores per node) at 2.66 GHz, and 96 GB of memory. Each processor has its own memory controller. Therefore each node has 2 NUMA regions, connected through a high-speed Intel QPI (Quick Path Interconnect) links. The interconnection network is InfiniBand 4X QDR (32 Gbps of theoretical effective bandwidth), with Mellanox ConnectX HCAs. The InfiniBand switches are Voltaire Grid Director 4036. The library has been tested with up to 648 cores in this system. The number of nodes used in the performance evaluation is $\lceil n/12 \rceil$, being n the total number of cores used. The UPC compiler and runtime is Berkeley UPC 2.12.2, a minor release fixing some bugs on 2.12.1, and the communication layer uses the InfiniBand Verbs library for distributed memory communication. The GASNet PSHM optimization has been also enabled in this system. The backend C compiler available in the system is the Intel 11.1.

The fifth and last system is the JuRoPA (Jülich Research on Petaflop Architectures) supercomputer, comprised of 2208 Sun Blade 6048 nodes, each one with 2 quad-core Intel Xeon X5570 Nehalem-EP processors (hence 8 cores per node) at

2.93 GHz, and 24 GB of memory. Each processor has its own memory controller. Therefore each node has 2 NUMA regions, connected through a high-speed Intel QPI links. The interconnection network is InfiniBand 4X QDR (32 Gbps of theoretical effective bandwidth), with Mellanox ConnectX HCAs and a Sun Data Center Switch 648. The library has been tested with up to 4096 cores in this system. The number of nodes used in the performance evaluation is $\lceil n/8 \rceil$, being n the total number of cores used. The OS is SUSE Linux Enterprise Server 11, with kernel 2.6.32. The UPC compiler is Berkeley UPC 2.12.2, relying on the InfiniBand Verbs library for distributed memory communication. The GASNet PSHM optimization has been also enabled. The backend C compiler available in the system is the Intel 11.1.

The use of five different systems allows for a broader study. Finis Terrae and Superdome are basically two different implementations of the same node architecture, allowing to extrapolate results in a system with a large number of NUMA regions to future systems with multiple nodes. The architecture and network of SVG is present in many deployed clusters, and even though it does not have RDMA capable hardware that can fully benefit from one-sided communications, results in this system are valuable to a large number of researchers. JUDGE allows to compare the benefits of hierarchical trees, that can be hidden in systems with a number of cores per node that is a power of two, due to the fact that an hierarchical binomial-based tree in those systems is equivalent to a normal binomial tree. Lastly, JuRoPA allows to study the behavior of the algorithms in a large scale system.

The implementation of the proposed algorithm (from now on we will refer to it as NUMACol) presented in this paper has been tested against the Berkeley UPC collectives (from now on BerkeleyCol), based on the high performance layer GASNet which implements an optimized binomial tree scheme; and also against the Michigan Technological University (MTU) reference implementation of the collective operations (from now on MTUCol), based on flat trees on top of `upc_memcpy` operations. The reference implementation has, like the algorithm here presented, two approaches: pull and push, where data is either pulled from the destination thread or pushed from the source thread.

The software used for the performance evaluation is the UPC Operations Microbenchmarking Suite (UOMS) [18], version 1.1. For each particular test, given a number of cores and message block sizes, it performs several iterations, from 1000 iterations to 20 iterations depending on the number of cores being used and the message size, to ensure representativeness and significance of the measures. All the tests have been performed in the same batch job, one after the other, to try to guarantee fairness in the comparison. Finally, the metric shown is the best result for each setup. By showing the minimum runtime the paper presents the performance of the algorithm without the influence of external factors (e.g., network contention/congestion), allowing to focus on the scalability of the operations that implement the proposed algorithm.

The UPC threads distribution has been performed in a block fashion. That is, consecutive thread ranks will be in the same node until the node is fully populated. This benefits algorithms that use trees but are not topology aware.

4.2 Analysis of Performance Results

Table 2 Bandwith (in MB/s) obtained by the basic pull approach algorithm (labelled Pull) and the pull algorithm with fragmentation of a fixed size and flat trees at the intra NUMA level of this library (labelled Pull-f-f). The data displayed has been obtained with the maximum number of processes tested in each system. That is: 1024 cores in Finis Terrae, 128 cores in Superdome, 192 cores in SVG, 648 cores in JUDGE and 4096 cores in JuRoPA.

		Message Size				
		256B	4KB	64KB	1MB	16MB
FT	Pull	665.2	9706	63465	48409	48647
	Pull-f-f	650.5	9865	84944	162280	167094
SD	Pull	172.2	2501	7284	7764	10750
	Pull-f-f	172.6	2610	13003	26265	24031
SVG	Pull	11.86	188.0	1900	4094	4096
	Pull-f-f	11.83	184.5	1824	3194	3322
JUDGE	Pull	1550	22493	124537	108300	83310
	Pull-f-f	1521	22881	157871	248346	270849
JuRoPA	Pull	4369	68478	476794	320855	275361
	Pull-f-f	4387	66841	582289	1155182	1222592

This section presents the performance results of four representative collectives, broadcast, reduce, scatter and gather in Figures 3, 4, 5 and 6, respectively. Their results have been also analyzed comparatively against MPI collectives. All figures present the performance of a representative medium-size message (16KB) on the left and the performance of a representative large-size message on the right. The size of the large message is 1MB for broadcast and reduce, and 64KB for scatter and gather, due to their higher memory requirements in the root process, which is the result of multiplying the message size by the number of processes. The *y* axis represents latency in microseconds in the graphs on the left (medium-size message case), whereas the *y* axis represents bandwidth in GB/s in the graphs on the right (large-size message case), except for broadcast on the SVG, which shows MB/s, and the reduce operation, which always shows latencies.

Variations in the same basic algorithm can lead to some dramatic performance differences, as shown in Table 2. The graphs display only the most relevant algorithms for each combination of system, function and message size.

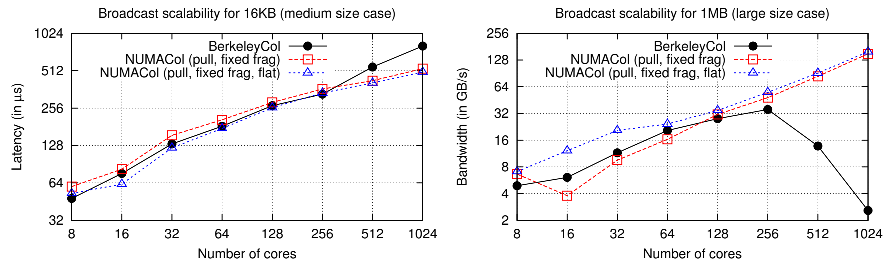
4.2.1 Scalability and Performance of UPC Broadcast Collective

Figure 3(a) shows the performance obtained for the different implementations of the broadcast operation in the Finis Terrae supercomputer. The most relevant algorithms for this system are the variations of the pull approach with static fragmentation, both with flat tree and binomial trees in the intra NUMA level, NUMACol (pull, fixed frag, flat) and NUMACol (pull, fixed frag), respectively. The proposed algorithms present the highest benefit, both for 16KB and 1MB messages, on 16 cores, and also for 512 and 1024 cores. The efficient handling of intranode transfers is key for achieving a good 16-core performance result whereas the scalability on 512 and 1024 cores is key to outperform BerkeleyCol, which suffer performance drops for the largest core counts. These graphs confirm that it is possible to improve the performance through NUMA-awareness. 16KB message performance (left graphs) is dominated by start-up latency and synchronization whereas 1MB message performance is dominated by the ability to overlap communications and harness data locality. Furthermore, scalable algorithms do not show performance degradation as the number of cores increases. In fact, the proposed NUMACol algorithms scale almost linearly, whereas BerkeleyCol suffers from poor scalability. Thus, the bandwidth obtained by NUMACol (pull, fixed frag, flat) is more than 61 times the bandwidth of BerkeleyCol on 1024 cores.

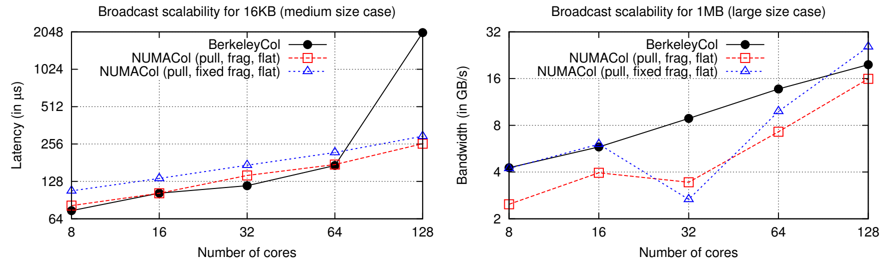
Figure 3(b) presents the performance in the Superdome system. The best performer algorithms in this system are again variations of the pull approach using flat trees in the intra NUMA level, both with static and dynamic fragmentation. In this system BerkeleyCol does not scale for 16 KB. Here the OS scheduler has a major importance since it is a shared memory machine with 16 NUMA regions. Different core mappings might yield significantly different results. This is specially important in the medium message case, since it is latency bounded rather than bandwidth bounded. This is why the NUMACol algorithms present more stable results, especially for 16 KB messages. Furthermore, it is remarkable the performance benefits with 128 cores, where the NUMACol algorithms obtain almost 8 times the BerkeleyCol performance. However, different runs can yield different results for BerkeleyCol, depending on the OS scheduler decisions, resulting in a non-predictable performance. Regarding 1 MB communication scalability, NUMACol suffer when using 32 or 64 cores (2 or 4 processes per cell). However, its better scalability allows performance benefits of around 30% over BerkeleyCol.

The results for the SVG system can be observed in Figure 3(c). The best NUMACol performance results have been obtained with the pull version with flat trees in the intra NUMA level, and the pull version with dynamic fragmentation. The performance drops significantly for 16 KB messages when more than one node is in use. Additionally, there is an increasing difference between the NUMACol algorithms and BerkeleyCol when fully populating the first node. In systems using Magny-Cours processors or similar architectures with many NUMA regions, the NUMA awareness becomes more important. With more than one node the NUMACol algorithms are heavily penalized. The network is Gigabit Ethernet, with high latency. Since the NUMACol algorithms rely on semaphores for synchronizing, which are basically very short messages, they will suffer in latency bounded scenarios, like the one depicted on the left plot. The right plot is bandwidth bounded, and therefore the use of semaphores does not hurt the performance as much as in latency bounded scenarios. The tree topology yields major gainings in this scenario. The pull approach with dynamic fragmentation performs almost 18 times better than BerkeleyCol, with 192 processes. However, despite its good performance compared with BerkeleyCol, the network prevents to achieve a good scaling. The bandwidth for 192 processes is just 28% higher than with 96 processes.

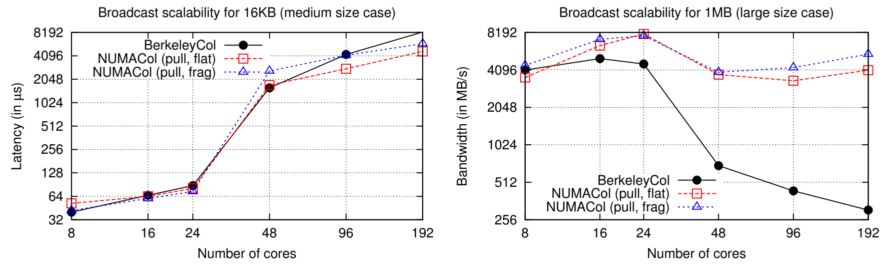
The performance numbers obtained in the JUDGE supercomputer are shown in Figure 3(d). This system shares some common features with the Finis Terrae supercomputer. Namely the network, even though JUDGE is equipped with a later generation of the InfiniBand standard (InfiniBand QDR 32 Gbps vs InfiniBand DDR 16



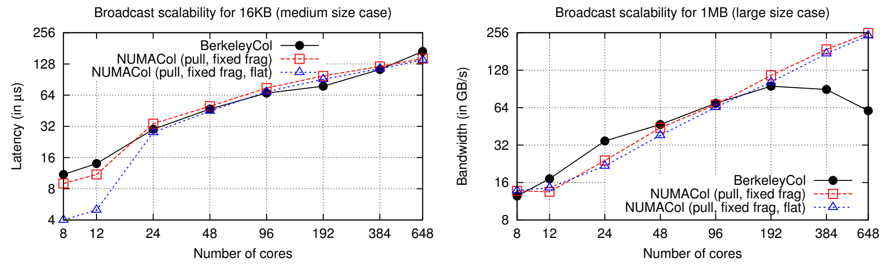
(a) Finis Terrae



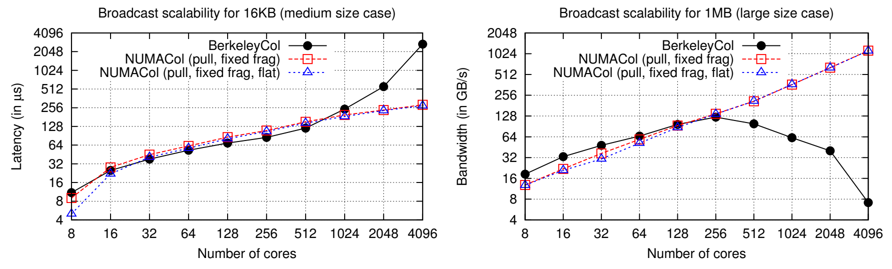
(b) Superdome



(c) SVG



(d) JUDGE



(e) JuRoPA

Fig. 3 Broadcast performance and scalability

Gbps). The behavior of the different collective operations evaluated is also similar. The major difference is the gap between BerkeleyCol and the NUMACol algorithms in the medium message case, when a single node is in use. This gap is the result of the good latency of the QPI bus and the tree topology mapped to the hardware, allowing the algorithm to achieve a good performance. The results for the large message case show the same tendency than in the Finis Terrae system. The BerkeleyCol collective does not scale beyond 192 processes, while the NUMACol algorithms keep scaling. For 648 processes the performance of the pull approach with fixed fragmentation is more than 420% of the performance of the BerkeleyCol collective.

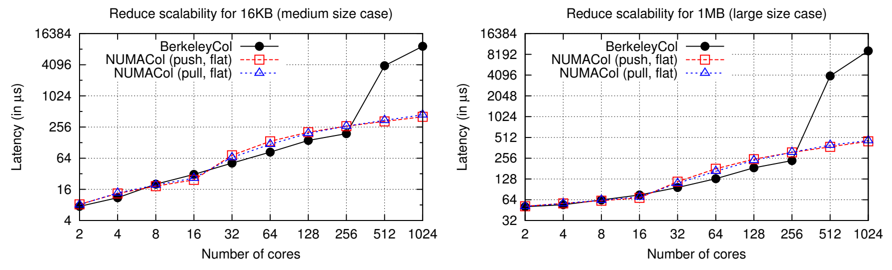
Lastly, the Figure 3(e) depicts the results for the JuRoPA supercomputer. This supercomputer also shares some architectural features with JUDGE and Finis Terrae, and the algorithms showed are the same as in these systems. The results are also similar to the results observed in JUDGE. However, there is one remarkable difference in the medium message case. Even though the performance of BerkeleyCol is slightly better than the NUMACol algorithms' performance in setups with a few nodes, with 1024 processes this gap disappears, showing the superior scalability of the NUMACol algorithms. In the large message case stands out the fact that the NUMACol algorithms keep scaling without hesitation up to 4096 processes. The NUMACol algorithm with pull approach and static fragmentation achieves almost 160 times the bandwidth obtained by BerkeleyCol, with 4096 processes.

The conclusions that can be extracted about this operation are: (1) the scalability of the developed algorithms is outstanding, especially for large messages; (2) for medium messages its performance is good within one node if the internal node buses are latency optimized; and (3) for medium messages BerkeleyCol usually performs better when more than one node is in use. Optimizations in the network access can boost further the performance of the NUMACol algorithms.

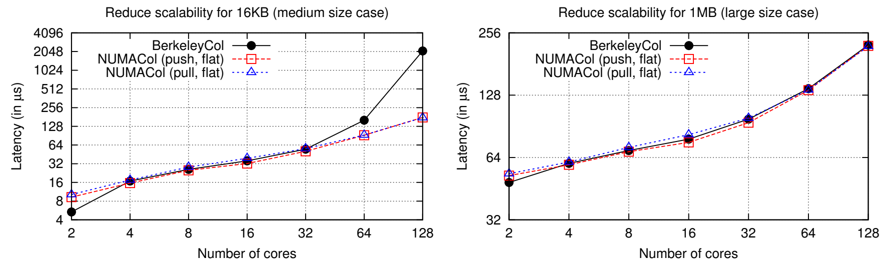
4.2.2 Scalability and Performance of UPC Reduce Collective

There are no major differences between algorithms for the reduce operation. Therefore, all graphs will show the same two algorithms: Push and pull with flat trees in the intra NUMA level. The reduce operation is addition, and the data type double. The data size is per process. Therefore 2048 elements per process for 16KB message size, and 131072 for 1MB.

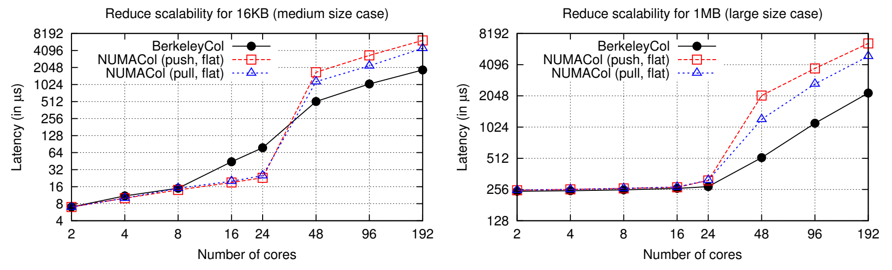
Figure 4(a) represents the performance for the reduce operation in the Finis Terrae supercomputer. The medium message case shows that the algorithms scale steadily. This performance data is the most important one from the communication scalability point of view, since the large message case will be computational power bounded. The data moved between processes will be the same, even though each process will have to spend more time computing the reduction of its own data in the large case. When using more than one node the performance is worse than BerkeleyCol by a narrow margin. Since BerkeleyCol has its collectives implemented in GASNet, rather than in UPC, its network access is slightly faster than directly from the UPC layer, causing this performance difference. Despite its slightly better performance, BerkeleyCol performs much worse for 512 processes or more. This fact is also present in the other



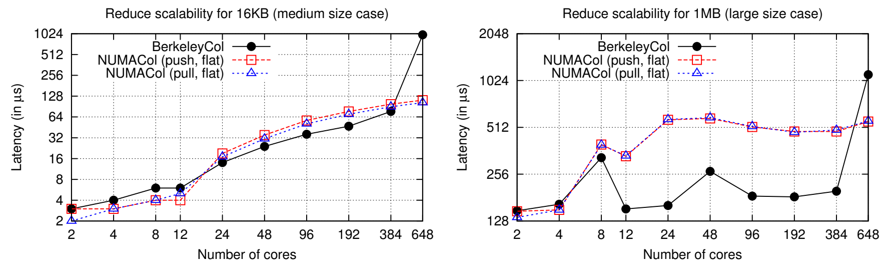
(a) Finis Terrae



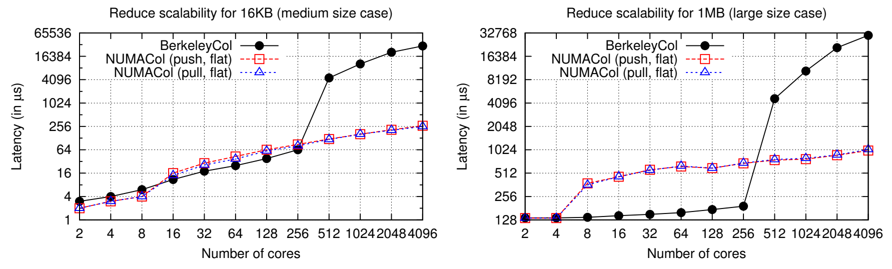
(b) Superdome



(c) SVG



(d) JUDGE



(e) JuRoPA

Fig. 4 Reduce performance and scalability

InfiniBand systems. The issue can be attributed to the InfiniBand conduit or the reduce algorithm in BerkeleyCol. The large message case is not large enough on this system to be computational power bounded. Therefore, when more than one node is in use the time to synchronize the processes is larger than the time spent computing, and left and right graphs are quite similar.

The Figure 4(b) shows the performance obtained in the Superdome system. In this shared memory system the results for the medium message case show that the BerkeleyCol reduce performs better than the NUMACol algorithms just with 2 processes. Up to 32 processes both algorithms performs at the same level. However, for 64 and, especially, 128 processes, the NUMACol algorithms keep scaling, while the performance of BerkeleyCol degrades. This effect is observable for data sizes from 8 bytes to 16KB, not being present in the large message case. In the right graph the NUMACol algorithms outperform BerkeleyCol except for 4 processes. However, with the system fully populated the differences are not appreciable, since the limit is imposed by the caches and memory buses performance, and the operation is computational power bounded.

The results for the SVG system are presented in Figure 4(c). In the medium message case the NUMACol algorithms outperform BerkeleyCol when just one node is in use (up to 24 cores). This is especially true when the node is fully populated (using 24 cores), due to the NUMA awareness. However, and as seen before, when more than one node is used, the NUMACol algorithms do not perform better than the BerkeleyCol counterpart. The large message case shows a scenario very similar to the medium message case when using more than one node, since the Gigabit Ethernet interconnect becomes a major bottleneck and the benefits reducing the computational times are neglected by the high latency of the network (as it can be seen for the medium message case, the left graph). When using a single node BerkeleyCol and the NUMACol algorithms perform at the same level, except for the case with 24 processes. The reduce computational task is not optimized on this system, and therefore, as the data sets to be reduced get larger, the importance of the computing time increases, and the benefits of the NUMACol algorithms are neglected.

Figure 4(d) shows the results in JUDGE. The general shape of the plot in the medium message case is similar to the results for the Finis Terrae system. However, in JUDGE the NUMACol algorithms are able to better exploit the NUMA hardware than in the Finis Terrae. This is due to the fact that in JUDGE the caches are shared in the same NUMA region, since there is a single NUMA region per socket. However, in Finis Terrae there are 4 different processors per NUMA region. Therefore communication will be significantly faster in JUDGE between neighbouring processes. In the Finis Terrae the ratio between speed communicating processes in different processors, but same NUMA region, and speed communicating processes in different NUMA regions is much lower. However, as for the previously analyzed systems, when more than a single node is being used, this advantage is lost due to the high network latency overhead which hides the differences between algorithms in the shared memory scenario. Regarding the 1MB performance results, BerkeleyCol is generally the best performer. This is because of the better implementation of the reduce computations in BerkeleyCol collective together with the fact that the QDR InfiniBand interconnection network is fast enough to make this setup computational power bounded. Therefore, NUMACol collectives are only able to outperform BerkeleyCol when using 648 cores, as for more than 384 processes the performance of BerkeleyCol degrades sharply.

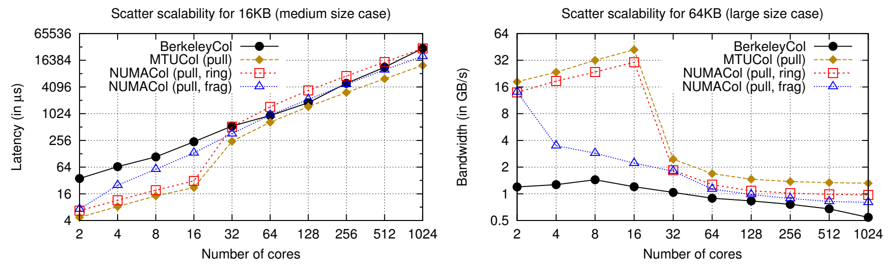
Lastly, Figure 4(e) displays the performance obtained in JuRoPA, a system which is similar to JUDGE in terms of architecture. The biggest difference, besides the size of the system, is the type of processors, with different number of cores (JUDGE has hexa-core Xeon Westmere processors whereas JuRoPA has quad-core Xeon Nehalem processors). Thus, in JuRoPA when using more than 8 cores (more than a single node), BerkeleyCol outperforms the NUMACol algorithms in both the medium and the large message scenarios, but only up to 256 cores, as for 512 cores BerkeleyCol's performance degrades, whereas the NUMACol algorithms keep scaling steadily.

The analysis of the performance of the reduce implementations have allowed to draw the following conclusions: (1) the NUMACol algorithms can effectively outperform BerkeleyCol in modern NUMA hardware; (2) the performance of NUMACol reduce algorithms is latency sensitive, due to the synchronization and copy of single elements between processes, so therefore reducing network latency yields significant improvements, as observed when comparing systems with low latency networks (Finis Terrae, JUDGE or JuRoPA) with systems with high latency networks (SVG); and finally (3) BerkeleyCol presents a much more efficient implementation of the arithmetic operations supported in the reduce operation, which means that NUMACol reduce implementations have still room for improving its performance.

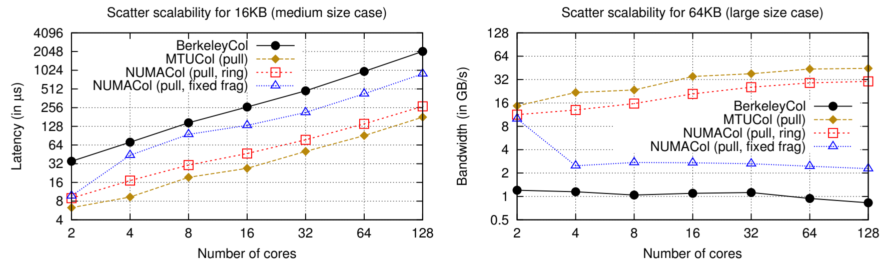
4.2.3 Scalability and Performance of UPC Scatter Collective

The MTU reference implementation of the scatter operation, unlike the broadcast and the reduce reference implementations (whose results were not shown for clarity purposes), presents a quite competitive performance despite its simplicity (it implements a flat tree). In the scatter operation the data from a root process has to be distributed (scattered) among all processes participating in the collective operation. The bottlenecks are, therefore, the outbound bandwidth of the root process and the start-up network latency. The simple algorithm implemented in the reference library is a good alternative due to that, since the use of the bandwidth of the root process is maximized without additional synchronization and copying overhead. As for previous collectives, for clarity purposes only the two best performer NUMACol algorithms are shown. Additionally, for scatter and gather the MTU reference library is considered and every graph will show its best performer algorithm (either the pull or the push version). Finally, it has to be noted that for scatter and gather the amount of data to be scattered/gathered increases with the number of cores. In this performance evaluation the selected message sizes are 16 KB an 64 KB. Therefore, by selecting 64 KB messages the root process will be handling a 1 MB message when communicating 16 cores (16 $\times$ 64 KB), or handling a 256 MB message when communicating 4096 cores (4096 $\times$ 64 KB).

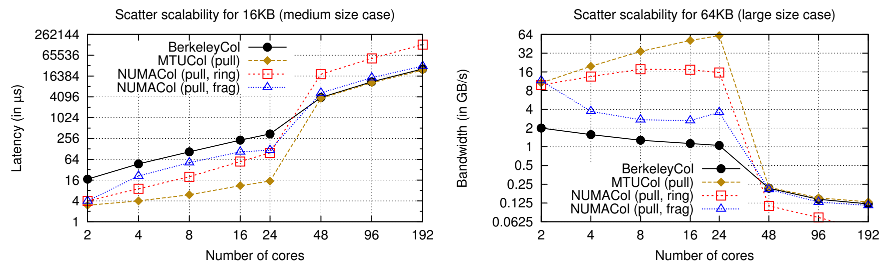
Figure 5(a) displays the results obtained from the benchmarking of the scatter operation in the Finis Terrae supercomputer. The relevant algorithms for this supercomputer are the pull versions of the MTUCol library, of the NUMACol ring algorithm and the NUMACol tree with dynamic fragmentation. In the 16KB case, in the



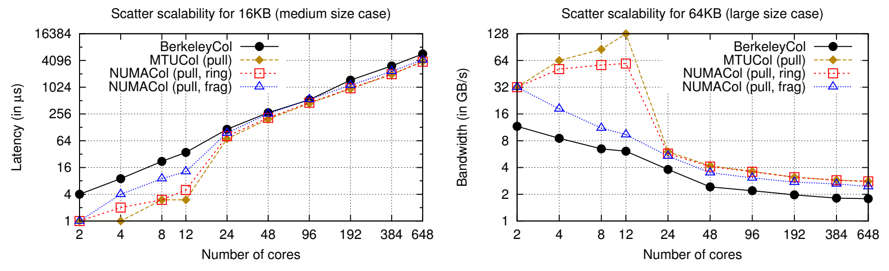
(a) Finis Terrae



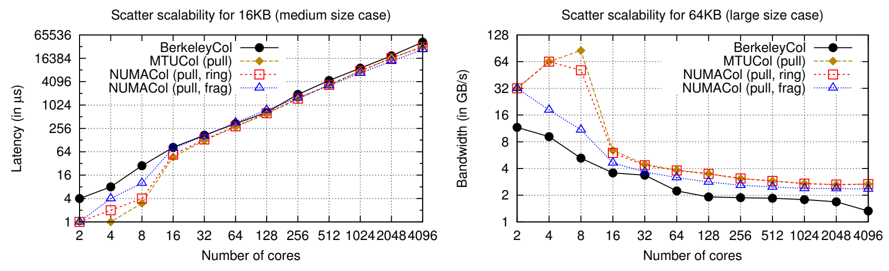
(b) Superdome



(c) SVG



(d) JUDGE



(e) JuRoPA

Fig. 5 Scatter performance and scalability

one hand the best performer is the reference implementation, for the whole range of number of cores evaluated (2-1024). On the other hand, BerkeleyCol is the worst performer in shared memory (up to 16 cores), whereas it performs slightly better when using two or more nodes (from 32 cores), except when using 1024 cores (64 nodes). Here the ring algorithm presents the opposite behavior, as it performs well in shared memory (close to MTUCol performance), but it is the worst performer in the internode case. Finally, the pull version of the NUMACol tree with dynamic fragmentation has balanced performance, between the best and the worst case. The conclusions derived from the analysis of the performance results using 64 KB messages are different. Thus, BerkeleyCol is always the worst performer. Here MTUCol is the best performer, but in this case closely followed by the NUMACol ring algorithm. Once again the pull version of the NUMACol tree algorithm with dynamic fragmentation is not able to take advantages of its features because here the bottleneck is the outbound bandwidth of the root process. However, as for the 16 KB case, it presents performance results between MTUCol and the best performer NUMACol algorithm. When using 1024 processes the performance gap between the best performer and the worst performer is almost 1 GB/s, which in relatively terms means that the best performer, MTUCol, presents 3 times higher performance than the worst performer, BerkeleyCol, which is not able to provide scalable bandwidth as the number of cores increases.

In Figure 5(b) the results measured in the Superdome system are showed. The best performer algorithms are the same as for the Finis Terrae, except for the NUMACol tree algorithm, which presents its optimal performance with static fragmentation. Here the differences between algorithms are much higher than in the Finis Terrae system, in both cases (16KB and 64KB). The reason is that the Superdome is a large NUMA server with lower communication latency than an interconnection network such as InfiniBand (the network in Finis Terrae). Moreover, in this shared memory system it is possible to access directly to the source data, minimizing problems such as congestion/contention like in a networked environment. Therefore, removing the interconnection network limitations (latency overhead, network congestion and contention) the differences between algorithms are more noticeable. In fact, the performance gap between the best (MTUCol) and the worst performer (BerkeleyCol) can be as high as 54 times, as for 64KB message size and 128 cores. As before, the best performer is the reference implementation, whereas the performance of BerkeleyCol falls behind all the other evaluated options, for 16KB and 64KB. The ring algorithm shows performance results around 30% lower than MTUCol, but following the same trend line, as both algorithms show very similar scalability. Finally, the NUMACol tree algorithm presents performance results quite close to the worst performer, BerkeleyCol, since the multiple levels the data has to go through, plus additional memory requirements and synchronizations, do not compensate. As can be derived from observing Figure 5, the scatter operation is only able to scale on the Superdome, where BerkeleyCol is outperformed for 64KB messages on 128 cores by the NUMACol tree algorithm (2.75 times higher performance), the NUMACol ring algorithm (37 times higher performance) and MTUCol (54 times higher performance).

In the SVG the best performer algorithms are the same as for the Finis Terrae supercomputer, namely the pull versions of MTUCol, NUMACol ring and NUMACol tree with fixed fragmentation. The results can be seen in Figure 5(c). Here the results in shared memory (intra-node, up to 24 cores) are similar to the Superdome system, although the MTUCol bandwidth is higher for 64 KB messages. However, for internode results (from 48 cores) the network latency overhead is a major issue, since the network available in this system –Gigabit Ethernet– presents a very high start-up latency. Thus, algorithms such as NUMACol ring especially suffers this high start-up overhead as it relies on semaphores, which are implemented using very short messages. Therefore, its performance on inter-node setups (from 48 cores) falls behind the remaining algorithms which are less sensitive to start-up network latency. BerkeleyCol, MTUCol and the NUMACol tree algorithm are able to take more advantage of the network, despite their limitations, avoiding synchronization overhead.

The results measured in JUDGE can be seen in Figure 5(d), where the best performer algorithms are the same as for the SVG and Finis Terrae. Moreover, the performance of these algorithms is quite similar to previous results on shared memory (intra-node case, using up to 12 cores). However, when using multiple nodes –24 or more cores– the performance drops significantly, showing higher latency (left graph) or lower bandwidth (right graph). Thus, in this case the MTUCol and NUMACol ring algorithms show quite similar results, within 1% of performance gap for the 648 core setup, for both 16KB and 64KB. However, the use of NUMACol ring is recommended as communications are done one by one, coordinated by semaphores, which presents lower risk than the MTUCol implementation where all cores communicate to the root process, which could be potentially an important bottleneck. These two algorithms –the MTUCol and NUMACol ring– perform up to 60% better than BerkeleyCol on 648 cores.

Finally, the last system, JuRoPA, has an architecture similar to JUDGE, so it seems reasonable that the best performer algorithms are the same as for JUDGE, and that their performance results present similar behavior (they can be seen in Figure 5(e)), so they share most of the analysis of the JUDGE results. Regarding JuRoPA benchmarking, the most important contribution is the analysis of the selected algorithms using up to 4096 cores. Thus, one of the conclusions of the analysis of the results is that MTUCol (pull), which implements a flat tree, is able to cope with up to 4096 simulatenous messages, even without degrading too much the performance, thanks to the InfiniBand network. However, BerkeleyCol can not avoid a significant performance drop for the 64KB test case using 4096 cores, falling in this case below half of the performance of MTUCol and NUMACol ring.

The conclusions that can be derived from the analysis of the performance results of the scatter operation are: (1) tree-based NUMACol algorithms, despite their scalability, are never the best option, due to the extra data that has to be handled; (2) the scatter operation is seriously limited by the outbound performance at the root process, which explains why quite simple algorithms, such as the flat tree implemented by MTUCol, are able to achieve the best performance although they might be disregarding the scalability of the data transfers; and finally (3) the MTUCol implementation has shown the best performance results and it has been able to deal with up to 4096 simultaneous communications, without saturating the interconnection network (in the evaluated system an InfiniBand network) and without requiring the implementation of any synchronization mechanism to support the scalability of the operation.

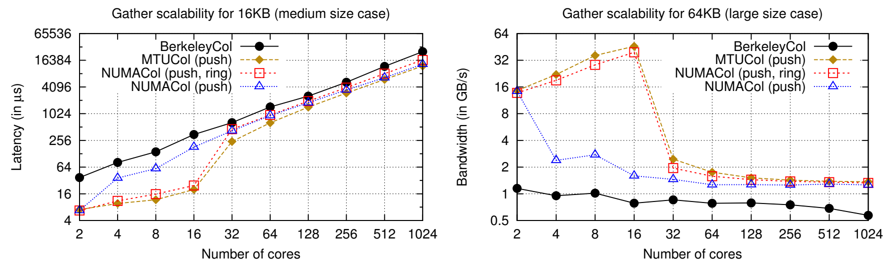
4.2.4 Scalability and Performance of UPC Gather Collective

Figure 6 presents the performance results of the microbenchmarking of the gather operation on the 5 representative systems considered in this work. As for scatter, MTUCol, implementing a quite simple flat tree algorithm, has an outstanding performance. In this case the data is collected from all the processes and has to be written in the root process, so it is the reverse operation of the scatter and the analysis could be the same as for the scatter, just considering the reverse operation. Thus, the bottleneck is the inbound bandwidth and latency. Apart from the considerations about the direction of the communications, the gather operation presents performance results very similar to those of the scatter collective for all the systems and messages sizes. Thus, the analysis and conclusions for the scatter results are perfectly valid for gather. However, it shall be noted that whereas the best performer algorithms for scatter are those which implement pull-based approaches, for gather the best option is push. The reason behind that is that communications are initiated by all the participants, rather than just one. Therefore, the cost of setting up the communication is partly distributed, avoiding jitter and providing better overlapping.

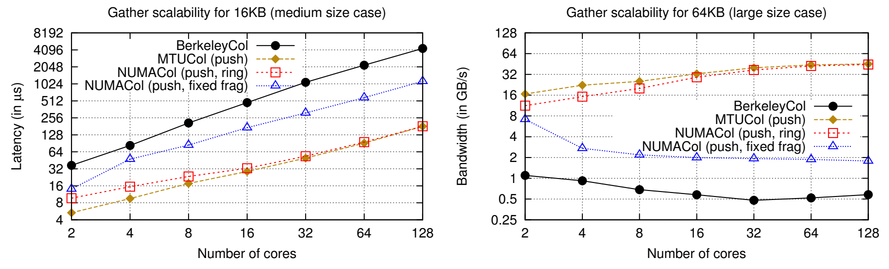
4.2.5 Comparative Performance Analysis of NUMA Algorithms against MPI

This section presents a comparative evaluation of the proposed NUMACol algorithms against state-of-the-art collective algorithms, such as those available for MPI, which has been carried out in JuRoPA using the MPI implementation ParaStationMPI 5.0.27, based on MPICH2 1.4.1p1. Even though ParaStationMPI is not as widely spread as other MPICH2 derivatives, the fact that it is based on MPICH2 makes suitable for a reasonable comparison. Moreover, this is the MPI implementation installed and supported on JuRoPA, and therefore results with it are more significant for users of this system.

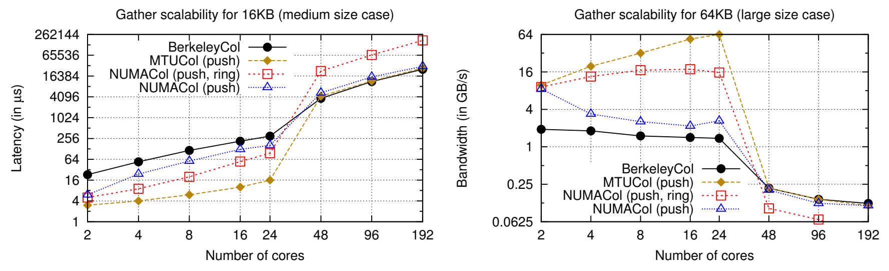
The software used for the performance evaluation is the Intel MPI Benchmarks (IMB) suite [9] version 3.2.4, and the UPC Operations Microbenchmarking Suite (UOMS) [18] version 1.1. These two tools have small differences in how they measure performance. IMB reports minimum, maximum and average latency. However, this data is the average per message size per process. The formulas of the reported date are described in Equation 1, where p is the number of processes and n is the number of iterations for a given message size. UOMS also reports minimum, maximum and average latencies. However, these latencies are considering iterations, not processes, as UOMS considers one operation finished just when all the processes involved are done, using `UPC_IN_ALLSYNC|UPC_OUT_ALLSYNC` as synchronisation mode. The formulas for the reported data in UOMS are described in Equation 2. In order to allow comparisons as fair as possible the reported values for IMB are the maximum, i.e. the highest average time among processes, to guarantee a state where all the processes have finished the operation. The reported values for UOMS are the average, i.e. the average time per iteration needed to guarantee that all the processes have finished the operation. Both reflect the average time needed to allow the operation to be completed by all the processes. UOMS also reports the bandwidth, but on this case based on the minimum latency. Due to that, the reported bandwidth on



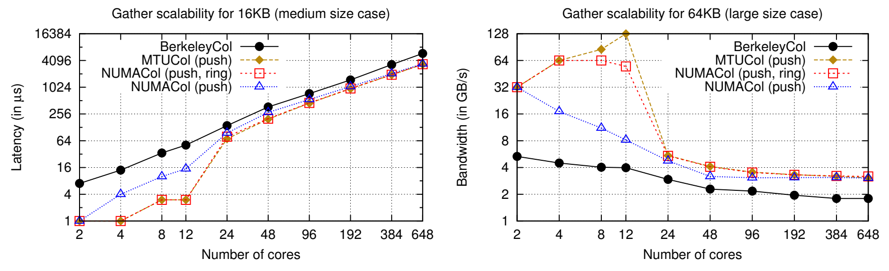
(a) Finis Terrae



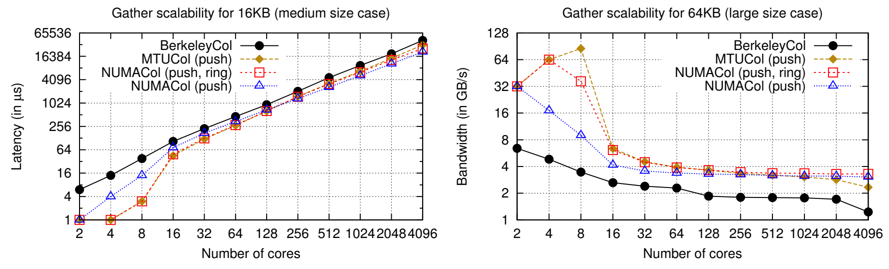
(b) Superdome



(c) SVG



(d) JUDGE



(e) JuRoPA

Fig. 6 Gather performance and scalability

this paper is not the one reported from the output of UOMS, but the one calculated using the average latency. Another difference that requires attention is that the root of each collective in IMB changes every iteration, whereas UOMS keeps the root static. However, the impact of this is minimum, specially in large experiments.

$$\min_{i=1}^{p}\left(\frac{\sum_{j=1}^{n} l_j}{n}\right)_i \qquad \max_{i=1}^{p}\left(\frac{\sum_{j=1}^{n} l_j}{n}\right)_i \qquad \frac{\sum_{i=1}^{p}\left(\frac{\sum_{j=1}^{n} l_j}{n}\right)_i}{p} \tag{1}$$

$$\min_{i=1}^{n}\left(\max_{j=1}^{p} l_j\right)_i \qquad \max_{i=1}^{n}\left(\max_{j=1}^{p} l_j\right)_i \qquad \frac{\sum_{i=1}^{n}\left(\max_{j=1}^{p} l_j\right)_i}{n} \tag{2}$$

MPICH2 implements three broadcast algorithms, selected at runtime depending on message size. These message size thresholds are configurable, but this evaluation uses the default thresholds. Thus, for messages up to 12KB the algorithm is based on binomial trees. For sizes between 12KB and 512KB the algorithm performs a scatter using a binomial tree and followed by an allgather implemented with a recursive doubling algorithm. For messages larger than 512KB the algorithm is similar to the previous one, except for the allgather phase, which is performed with a ring algorithm.

Regarding the scatter and gather operations, MPICH2 implements these collectives using an algorithm based on binomial trees, with intermediary buffers in non-leaf processes, in a similar way as the NUMA implementation proposed in this paper.

Finally, the reduce operation has been also included in this comparison. The reduce operation in UPC and MPI have significant differences. In UPC this collective is done on a shared array and produces a single value, whereas the outcome of the MPI reduce is an array result of reducing elements per position, using private arrays as source. However, when the number of elements per rank or UPC thread is 1, both operations are comparable. MPICH2 implements reduce using two algorithms: Rabenseifner's algorithm, for messages larger than 2KB, and a binomial algorithm for shorter messages. Since our comparison is limited to one element per rank, the Rabenseifner's algorithm is not used.

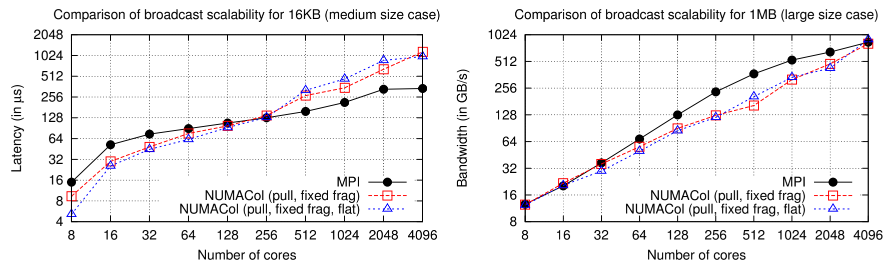
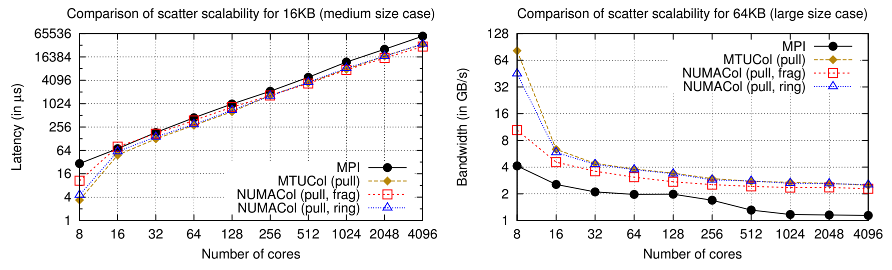


Fig. 7 Comparison of broadcast scalability of NUMA-based algorithms against MPI on JuRoPA

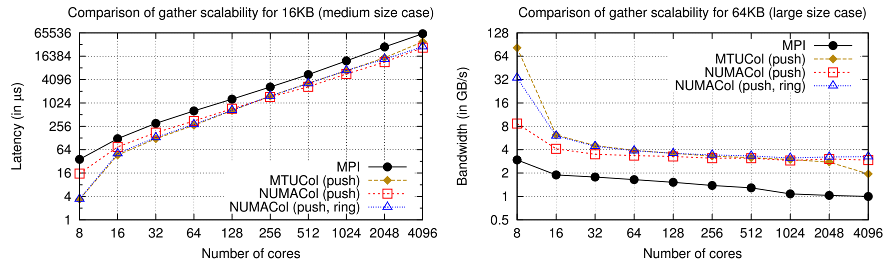
Figure 7 presents the comparison of NUMACol with MPI for the broadcast using two representative message sizes, 16KB representative of medium size messages, and 1 MB, representative of large messages. For short messages (<12KB) the considered algorithms have similar scalability, whereas the performance is highly dependent on the start-up latency achieved by MPI and UPC communications. Both graphs show a good performance and scalability of the NUMACol algorithms, although it is noticeable that MPI achieves the highest performance for 16KB when using more than 256 cores, and for 1MB from 64 up to 2048 cores. However, for 4096 cores MPI performance is overcome by the better scalability of the NUMACol algorithms. Nevertheless, it shall be noted that the UPC experimental results present much more variability than the MPI ones, both for the NUMACol algorithms and for the BerkeleyCol broadcast. This variability is already present in BerkeleyCol point-to-point communication, whose performance is much more variable than MPI point-to-point primitives, showing occasionally differences of an order of magnitude between measures for short messages. Nevertheless, when considering large messages these peaks are not present.

Figure 8 shows the comparative performance results for scatter and gather, respectively, with a format similar to the layout previously presented. Thus, the selected message size for evaluation are 16KB and 64KB. Regarding the performance results, generally MPI is the worst performer and the NUMACol ring algorithm the best performer (up to 3 times more performance than MPI), especially for 64KB. MTUCol also outperforms MPI. Here the NUMACol tree algorithm is basically the same as the MPI algorithm. There are only two major differences: the NUMA affinity support, not present in MPI, and the fragmentation of the messages. These two differences explain the better performance of the NUMACol algorithms. Moreover, as for gather the fragmentation does not add any benefit and the best algorithm is the one that does not use fragmentation. Therefore, in this case the key aspect for achieving more performance in UPC operations is the efficiency achieved in shared memory thanks to the NUMA binding. In gather, the data flows upwards, causing the algorithm to be more sensitive to jitter and accumulating the penalty of ignoring optimizations of the memory subsystem. In scatter this is less important, since the data is transmitted to the root, and immediately pulled by other root processes from other nodes, which minimizes the penalty of not optimizing the memory subsystem, instead of adding additional overhead.

In Figure 9 the results for reduce can be observed. Both plots contain the data for 8 bytes (a double per MPI rank or UPC thread). The plot on the left represents latency, whereas the plot on the right represents MFLOP/s. In this range, with a message size of just 8 bytes, the best algorithms are both pulling algorithms. However, despite their good scalability, their performance is worse than for MPI, and in some cases worse than BerkeleyCol. With this setup, all the algorithms (except NUMACol with flat tree at the NUMA level) are algorithms based on binomial trees. The number of cores per node and per NUMA region is power of 2, and therefore the shape of the trees and the cost of the operation is the same between them. However, MPI outperforms all the UPC implementations, due to its lower start-up latency, that is specially important in this case due to the fact that this operation is largely dominated by the network



(a) Scatter



(b) Gather

Fig. 8 Comparison of scatter and gather scalability of NUMA-based algorithms against MPI on JuRoPA

latency. This fact is also the root cause for the low number of MFLOP/s, due to the low computation/communication ratio.

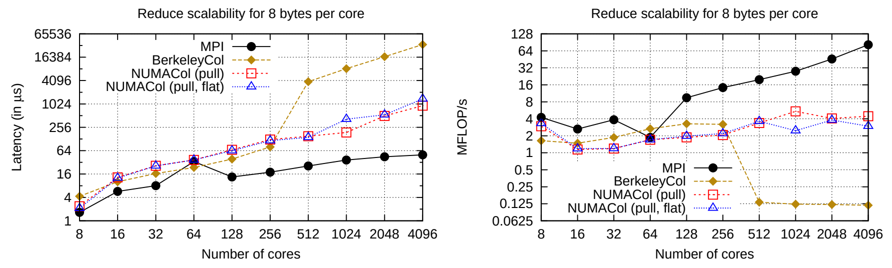


Fig. 9 Comparison of reduce scalability of NUMA-based algorithms against MPI on JuRoPA

4.2.6 Impact on Performance of Different Optimizations at High Core-Counts

The basic algorithm has been optimized using different techniques. However, up to now, the contribution of each optimization to the overall collective performance has not been assessed. This section analyzes the influence of several optimizations for broadcast, due to its importance in the context of this work. The analysis has been focused on the impact of these techniques on scalability. Therefore JuRoPA has been selected for this analysis due to its higher number of available cores (it has been used up to 4096 cores).

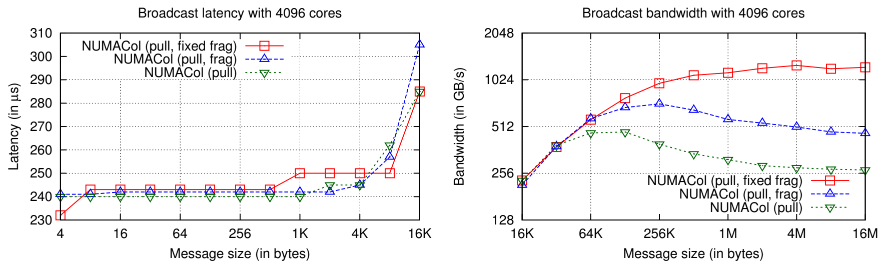


Fig. 10 Impact of message pipelining in broadcast performance

Figure 10 shows the contribution of the message pipelining to the overall performance, for a setup of 4096 cores. In short message communication, with messages from 4 bytes to 8KB, the performance of the different variations of the algorithm shows the same performance results. In fact, they are using the same algorithm since the dynamic fragmentation algorithm processes messages larger than 8KB, and the fixed fragmentation algorithm starts processing messages larger than 32KB. It is from this point, messages larger than 32KB, that each algorithm presents a different performance. Thus, the pull algorithm without fragmentation increases performance slightly for 64KB and 128KB, achieving at this latter point its peak performance, degrading performance from that point on. The pull algorithm with dynamic fragmentation performs twice as good as the pull algorithm. However, its performance also degrades for messages larger than 256KB. The usage of the fixed fragmentation pull algorithm achieves even higher performance, reaching its maximum at 4MB. At this point its performance is more than 4 times as good as the initially considered pull algorithm, showing the importance of message pipelining.

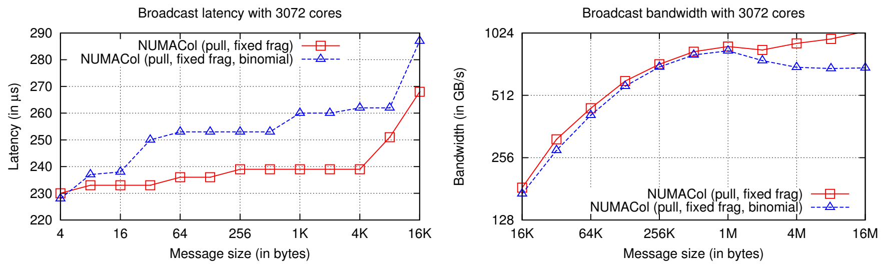


Fig. 11 Impact of multilevel trees in broadcast performance

Figure 11 presents the impact on performance of the usage of multilevel trees. This experiment has been conducted with 3072 processes, with 512 nodes and 6 processes per node. A multilevel tree assigning 4096 processes, with 512 nodes and

8 processes per node, is equal to a standard binomial tree, due to the usage of a number of processes per node that is a power of 2. However, nowadays is increasingly common to find systems with a number of cores per node that is not a power of 2. It is in these scenarios where the usage of multilevel trees become important and where they are different from binomial trees. In scenarios where the short messages latency dominates the overall performance, the importance of having a multilevel tree is noticeable for messages larger than 16 bytes. The difference between both approaches is small up to 1MB. At that point the benefits of using the most efficient multilevel tree become more apparent as the message size increases, and for 16MB the use of a multilevel tree performs 1.5 times better than using a binomial tree.

The benefits of NUMA affinity to control the mapping of processes to the underlying hardware are negligible in setups with a high number of nodes where the effects of network latency and bandwidth have much more impact on performance than the small benefit obtained from NUMA binding control. Nevertheless NUMA affinity has shown its importance in shared memory scenarios. Moreover, a few facts suggest that NUMA affinity control has room for improving collective operations performance over the coming years. Thus, (1) the latest processor models are directly connected to network interfaces, typically one per node. In this case the relevance of having the node root process in the processor with direct connection to the network increases. Moreover, (2) the increasing number of NUMA regions per socket is forcing the consideration of new algorithms that are able to minimize jitter. This has been demonstrated through Section 4.2. The NUMA aware algorithms have outperformed other approaches in single node setups, with fully populated nodes. Finally, (3) as interconnection networks become faster, supercomputers with a high number of nodes turn out to be more sensitive to jitter. These facts suggest that NUMA affinity can have a major impact in collective performance in future systems. Moreover, affinity should be carefully evaluated for every application, as show in [10]. Correct affinity can have a significant impact on the performance of an application. The optimal affinity setup for any application will not interfere with the performance of NUMACol, as long as the trees are set up according to the process mapping.

5 Conclusions

This work has presented a new series of algorithms for collective operations for NUMA multicore and manycore architectures. Its main contributions are: (1) the use of correct one-sided point-to-point communications (pull vs. push) to leverage communication overlapping; (2) message fragmentation to allow communication pipelining on one-sided communications; (3) resorting to multilevel trees to minimize the use of the slowest data paths; (4) NUMA region binding as a core feature of the collective library; and finally, (5) the use of fixed trees, to avoid tree computation at function init. It worths it when the initial copy to the tree root is faster than the tree computation.

The analysis of the implementation of these algorithms (NUMACol) has shown: (1) the implementation of these algorithms is able to equal and even outperform an evolved and more mature UPC library (BerkeleyCol); (2) NUMACol can outperform in some scenarios the state-of-the-art implementation of their equivalent functions in MPI, bringing another algorithm to the mix, allowing more possibilities for autotuning and choosing the most appropriate algorithm in each situation; (3) major contributor factors to performance are a tree mapped to the underlying hardware considering all levels, message pipelining, communications overlapping with adequate (pull vs. push) one-sided point-to-point transfers. Furthermore, (4) it is hard to determine which is the optimal tree shape for each level, as it depends on the architecture and message size; and (5) tree-based collectives are often outperformed by ring algorithms with communication overlapping, in operations where data have to be scattered/gathered from a single point. Finally, (6) NUMA binding does not improve significantly the performance in nowadays clusters, as the main performance bottleneck is the network overhead. However, due to its highly scalable design, it is expected that the performance benefits of the developed library will be higher in future systems with tens of NUMA regions.

Acknowledgements This work was funded by Hewlett-Packard and partially supported by the Ministry of Science and Innovation of Spain under Project TIN2010-16735 and by the Galician Government (Xunta de Galicia, Spain) under the Consolidation Program of Competitive Research Groups and under CN2012/211 grant, cofunded with FEDER. We gratefully thank Jim Bovay at HP for his valuable support, CESGA for providing access to the Finis Terrae, Superdome and SVG supercomputers, and Forschungzentrum Jülich for providing access to the JuRoPA and JUDGE supercomputers.

References

1. Advanced Micro Devices, Inc (2009) Magny-Cours and Direct Connect Architecture 2.0
2. Antony J, Janes PP, Rendell AP (2006) Exploring Thread and Memory Placement on NUMA Architectures: Solaris and Linux, UltraSPARC/FirePlane and Opteron/HyperTransport. In: Proc. 13th IEEE International Conference on High Performance Computing (HiPC'06), Bangalore (India), pp 338–352
3. Brightwell R, Pedretti KT (2009) Optimizing Multi-core MPI Collectives with SMARTMAP. In: Proc. 3rd International Workshop on Advanced Distributed and Parallel Network Applications (ADPNA 2009), Vienna (Austria), pp 370–377
4. Chan E, van de Geijn R, Gropp W, Thakur R (2006) Collective communication on architectures that support simultaneous communication over multiple links. In: Proc. 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'06), Manhattan (NY), pp 2–11
5. Conway P, Kalyanasundharam N, Donley G, Lepak K, Hughes B (2010) Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor. IEEE Micro 30:16–29
6. George Washington University (2010) GWU Unified Testing Suite (GUTS). http://threads.hpcl.gwu.edu/sites/guts [Last visited: March 2014]
7. Graham RL, Shipman GM (2008) MPI Support for Multi-core Architectures: Optimized Shared Memory Collectives. In: Proc. 15th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'08), Dublin (Ireland), pp 130–140

8. Hoefler T, Siebert C, Rehm W (2007) A Practically Constant-time MPI Broadcast Algorithm for Large-scale InfiniBand Clusters with Multicast. In: Proc. 7th Workshop on Communication Architecture for Clusters (CAC'07), Long Beach (CA), pp 1–8
9. Intel Corporation, "Intel MPI Benchmarks," http://software.intel.com/en-us/articles/intel-mpi-benchmarks [Last visited: March 2014].
10. Jeannot E, Mercier G (2010) Near-Optimal Placement of MPI Processes on Hierarchical NUMA Architectures. In: Proc. 16th International European Conference on Parallel and Distributed Computing (Euro-Par'10), Ischia (Italy), pp 199–210
11. Jiang W, Liu J, wook Jin H, Panda DK, Gropp W, Thakur R (2004) High Performance MPI-2 One-Sided Communication over InfiniBand. In: Proc. 4th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid'04), Chicago (IL), pp 531–538
12. Kandalla KC, Subramoni H, Santhanaraman G, Koop M, Panda DK (2009) Designing Multi-Leader-Based Allgather Algorithms for Multi-Core Clusters. In: Proc. 9th Workshop on Communication Architecture for Clusters (CAC'09), Rome (Italy), pp 1–8
13. Kandalla KC, Subramoni H, Vishnu A, Panda DK (2010) Designing Topology-Aware Collective Communication Algorithms for Large Scale Infiniband Clusters: Case Studies with Scatter and Gather. In: Proc. 10th Workshop on Communication Architecture for Clusters (CAC'10), Atlanta (GA), pp 1–8
14. Koop MJ, Sridhar JK, Panda DK (2008) Scalable MPI Design over InfiniBand using eXtended Reliable Connection. In: Proc. 10th IEEE International Conference on Cluster Computing (Cluster'08), Tsukuba (Japan), pp 203–212
15. Kumar R, Mamidala AR, Panda DK (2008) Scaling Alltoall Collective on Multi-core Systems. In: Proc. 8th Workshop on Communication Architecture for Clusters (CAC'08), Miami (FL), pp 1–8
16. Li S, Hoefler T, Snir M (2013) NUMA-Aware Shared Memory Collective Communication for MPI. In: Proc. 22nd International ACM Symposium on High Performance Parallel and Distributed Computing (HPDC'13), New York (NY), pp 85–96
17. Lorenzo JA, Rivera FF, Tuma P, Pichel JC (2009) On the Influence of Thread Allocation for Irregular Codes in NUMA Systems. In: Proc. 10th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'09), Hiroshima, (Japan), pp 146–153
18. Mallón DA, Mouriño JC, Gómez A, Taboada GL, Teijeiro C, Touriño J, Fraguela BB, Doallo R, Wibecan B (2010) UPC Operations Microbenchmarking Suite. In: Proc. 25th International Supercomputing Conference (ISC'10), Research Poster, Hamburg (Germany), URL: http://upc.cesga.es [Last visited: March 2014]
19. Mamidala AR, Kumar R, De D, Panda DK (2008) MPI Collectives on Modern Multicore Clusters: Performance Optimizations and Communication Characteristics. In: Proc. 8th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid'08), Lyon (France), pp 130–137
20. Miao Q, Sun G, Shan J, Chen G (2007) Single Data Copying for MPI Communication Optimization on Shared Memory System. In: Proc. 7th International Conference on Computational Science (ICCS'07), Beijing (China), pp 700–707

21. Mouriño JC, Gómez A, Taboada JM, Landesa L, Bértolo JM, Obelleiro F, Rodríguez JL (2009) High Scalability Multipole Method. Solving Half Billion of Unknowns. Computer Science - R&D 23(3–4):169–175
22. Nishtala R, Yelick KA (2009) Optimizing Collective Communication on Multicores. In: Proc. 1st USENIX Workshop on Hot Topics in Parallelism (HotPar'09), Berkeley (CA), pp 1–6
23. Nishtala R, Zheng Y, Hargrove PH, Yelick KA (2011) Tuning Collective Communication for Partitioned Global Address Space Programming Models. Journal of Parallel Computing 37(9):576–591
24. Qian Y (2010) Design and Evaluation of Efficient Collective Communications on Modern Interconnects and Multi-core Clusters. Thesis (Ph.D, Electrical & Computer Engineering) – Queen's University. http://hdl.handle.net/1974/5383 [Last visited: March 2014]
25. Rabenseifner R (2007) A New Optimized MPI Reduce Algorithm. http://www.hlrs.de/mpi/myreduce.html [Last visited: March 2014]
26. Salama RA, Sameh A (2007) Potential Performance Improvement of Collective Operations in UPC. In: Proc. 12th International Conference on Parallel Computing (ParCo'07), Jülich and Aachen (Germany), pp 413–422
27. Shipman GM, Poole S, Shamis P, Rabinovitz I (2008) X-SRQ - Improving Scalability and Performance of Multi-core InfiniBand Clusters. In: Proc. 15th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'08), Dublin (Ireland), pp 33–42
28. Thakur R, Rabenseifner R, Gropp W (2005) Optimization of Collective Communication Operations in MPICH. International Journal of High Performance Computing Applications 19(1):49–66
29. Top 500 (2008) Finis Terrae Supercomputer. http://www.top500.org/system/9500 [Last visited: March 2014]
30. Trahay F, Brunet E, Denis A, Namyst R (2008) A Multithreaded Communication Engine for Multicore Architectures. In: Proc. 8th Workshop on Communication Architecture for Clusters (CAC'08), Miami (FL), pp 1–7
31. Tu B, Fan J, Zhan J, Zhao X (2012) Performance Analysis and Optimization of MPI Collective Operations on Multi-core Clusters. Journal of Supercomputing 60(1):141–162
32. Vadhiyar SS, Fagg GE, Dongarra JJ (2000) Automatically Tuned Collective Communications. In: Proc. 2000 ACM/IEEE Conference on Supercomputing (SC'00), Dallas (TX), pp 1–11
33. Velamati MK, Kumar A, Jayam N, Senthilkumar G, Baruah PK, Sharma R, Kapoor S, Srinivasan A (2007) Optimization of Collective Communication in Intra-cell MPI. In: Proc. 14th International Conference on High Performance Computing (HiPC'07), Goa (India), pp 488–499
34. Wibecan B (2009) Proposal for Privatizability Functions. http://www2.hpcl.gwu.edu/pgas09/ HP_UPC_Proposal.pdf [Last visited: March 2014]
35. Z Ryne and S Seidel (2005) Ideas and Specifications for the new One-sided Collective Operations in UPC. http://www.upc.mtu.edu/papers/OnesidedColl.pdf [Last visited: March 2014]