Energy, Memory, and Runtime Tradeoffs for Implementing Collective Communication Operations

Torsten Hoefler and Dmitry Moor, ETH Zürich

1. Introduction

Collective operations are among the most important communication operations in shared- and distributed-memory parallel applications. In this paper, we analyze the tradeoffs between energy, memory, and runtime of different algorithms that implement such operations. We show that existing algorithms have varying behavior and that no known algorithm is optimal in all three regards. We also demonstrate examples where of three different algorithms solving the same problem, each algorithm is best in a different metric. We conclude by posing the challenge to explore the resulting tradeoffs in a more structured manner.

The performance of collective operations often directly affects the performance of parallel applications significantly. Thus, many researchers designed fast algorithms and optimized implementations for various collective communication operations. The newest version of the Message Passing Interface (MPI) standard [30], the de-facto standard for distributed-memory parallel programming, offers a set of commonly-used collective communications. These operations cover most use-cases discovered in the last two centuries and we thus use them as a representative sample for our analyses. In general, collective patterns reflect key characteristics of parallel algorithms at large numbers of processing elements, for example, parallel reductions are used to implement parallel summation and alltoall is a key part of many parallel sorting algorithms and linear transformations.

Recent hardware developments in large-scale computing increase the relative importance of other features besides pure execution time: energy and memory consumption may soon be key characteristics. Minimizing *energy consumption* is especially important in the context of large-scale systems or small battery-powered devices. *Memory consumption* is important in systems that offer hardware to support the execution of collective operations. Here, we assume state-of-the-art *offloaded* execution models (e.g., [23]) where communication schedules are downloaded into the network device that operates with severely limited resources. The increasing availability of such offload architectures motivates us to model the memory consumption of offloaded collective communications.

In this work, we provide an overview and a classification of state-of-the-art algorithms for various collective operations. Our report is not meant to cover all possible algorithms for implementing collective operations of which there are far too many to fit in the space limitations of this short article. Instead, our classification and analysis shall establish a discussion basis for the fundamental tradeoffs between runtime, energy, and memory consumption. For each algorithm, we derive analytic models for all three key metrics. Our theoretical study shows, for example, that reducing the number of messages sent may reduce the performance but, at the same time, decrease energy and memory consumption. Furthermore, our analysis of existing algorithms allows us to point out gaps and define future research topics. In general, we argue for a more general design mechanism that considers the multi-objective optimization problem for time, energy, and memory.

2. Architectural Energy, Runtime, and Memory Models

Good architectural models strike the balance between minimizing the number of parameters and modeling the architecture's main effects accurately. A small number of parameters facilitates reasoning about

algorithms and algorithm design and simplifies the optimization problems in the context of real applications. However, models need to capture the main parameters that determine the performance of the implementation on the target architecture. Several such models for the performance of communication algorithms have been designed. The most prominent ones belong to the LogP family while many other models can either be expressed as subsets of LogP (e.g., alpha-beta) or have a similar character but increase the complexity of the parameters, (e.g., PlogP [25]). For the purpose of this paper, we use LogGP [1] as a model for the execution time because we believe that it expresses the most relevant architecture parameters while still allowing elegant formulations of optimization problems.
We now proceed to discuss several communication technologies and mechanisms in the context of collective algorithms and the LogGP model.

Message Passing Message Passing is the basis of the design of LogGP. Here, **L** denotes the maximum communication latency between two endpoints. The parameter **o** represents the constant CPU overhead for sending or receiving a single message, e.g., the call to the message passing library. The parameter **g** is the equivalent overhead for sending or receiving a message caused by the network interface. The maximum of o and g limits the small-message injection rate, an important parameter of current interconnection networks. The model also implies that only L/g messages can be in flight between two processes at any time. The parameter **G** models the cost per injected Byte at the network interface, this is the reciprocal bandwidth. Finally, the number of processes is represented by **P**.

Noncoherent Shared Memory Noncoherent shared memory systems as used in remote direct memory access (RDMA) communications or for the data transfer between CPUs and GPUs are similar to message passing systems. The typical programming interface to such systems are put and get operations that store into or load from remote memory. The main difference to message passing is that the receiver is not explicitly involved and thus o is not charged at the destination. However, all other parameters remain. For the purpose of this article, we ignore this discrepancy with the traditional LogGP model.

Coherent Shared Memory Coherent memory systems are slightly more complex. Coherence between multiple caches is often guaranteed by a cache coherence protocol operating on blocks of memory (e.g., cache lines). The protocol ensures that each block always holds exactly one value in the whole system. Such protocols often allow for multiple readers (i.e., multiple identical copies of the block) but each write access requires exclusive ownership. Since all communication is implicitly performed during standard load/store accesses, performance characteristics are more complex and LogGP is only an approximate model for such transfers in the general case. Yet, if the amount of sharing is low (i.e., data is transferred from each writer to a single reader), then LogGP can model the performance characteristics accurately. Ramos and Hoefler [35] provide a detailed explanation of the intricacies of modeling for cache-coherent systems and related work.

Network Offload Architectures Some newer network architectures such as Portals IV [7] or CORE-Direct [14] allow to offload collective operations to the network device. This enables faster execution (messages do not need to travel to the CPU) and isolation (computations on the CPU and collective communications do not interfere and can progress independently). This reduces the impact of small delays on the CPU, often called system noise [19, 47] and allows asynchronous execution of nonblocking collective operations [17]. Communications are performed using messages and can thus be modeled using the LogGP model. Offload devices have limited resources to store communication schedules and we model the memory consumption of each algorithm in such devices.

Runtime Models We will use LogGP to model the approximate runtime of the algorithms on all target systems. Furthermore, in order to keep the models interpretable, we set $o > g$ and assume that the LogGP CPU overhead o is also charged in offloading devices so that we never need to charge g (o for offloading devices is most likely much smaller than o on a general-purpose CPU). We also assume that the cost to transmit a message of size s is $T_{\text{msg}} = L + 2o + sG$. We report the maximum finishing time that any process needs.

Energy Models Energy consumption can generally be split into two components: dynamic and static energy [28, 29]. The static energy is the leakage energy during the operation of an electronic device, regardless of the device's activity. Dynamic energy represents the energy that is consumed by activities such as computation, sending and receiving messages, or memory accesses. For the purpose of our analysis, we assume that computation and local memory operations (e.g., shuffling data) are free. These assumptions are similar to the LogGP model which also only considers network transactions. To model the energy for communication, we assume that each message consumes a fixed energy **e**. This represents the setup cost to send a zero-byte message and is similar to o and g in the LogP model, we do not separate CPU and network costs because energy consumption is additive and can thus be captured by a single parameter. Furthermore, we denote the energy required to transport each byte from the source's memory to the destination's memory as **E**, similar to LogGP's G parameter. This model assumes a fully connected network such that the energy consumption does not depend on the location of the source and destination. Thus, ignoring local computations, the total energy consumption of a collective operation is $L = T \cdot P + D$ where T is the runtime (e.g., modeled by LogGP), P is the leakage power, and D is the dynamic energy model. In our analysis, we derive dynamic energy models for the overall operation (the sum of all dynamic energies consumed at each process).

Memory Models Similarly, we derive a simple model for capturing memory overheads for offloading devices. To offload a collective operation to a network device, one copies some state (e.g., a set of triggers [7] or a set of management queue entries [14]) that models the execution schedule to the device. The device then generates messages based on arriving messages from other processes and the local state without CPU involvement. Here, we assume that each sent message has to be represented explicitly as a descriptor in the offloaded operation. We assume that these descriptors have the constant size **d**. This descriptor size does not depend on the size of the actual message to be sent or received. We report the maximum memory needed by any process.

3. Implementation Strategies for Collective Operations

Instead of describing algorithms for specific collectives, we discuss common algorithms to implement collective operations. For each of these algorithms, we develop runtime, energy, and memory overhead models. We then proceed to briefly describe each of MPI's collective operations and discuss how the algorithms can be used to implement it. This method reflects the state-of-the-art in which collective libraries often implement a set of algorithm skeletons and match them to particular collective implementations [12].

3.1. Existing Collective Algorithms

Each collective algorithm exploits a particular *virtual topology*, i.e., a directed graph representing message propagation between processes. We distinguish between three classes of collective algorithms: (1) trees in various shapes and forms, (2) distribution algorithms, and (3) specialized algorithms.

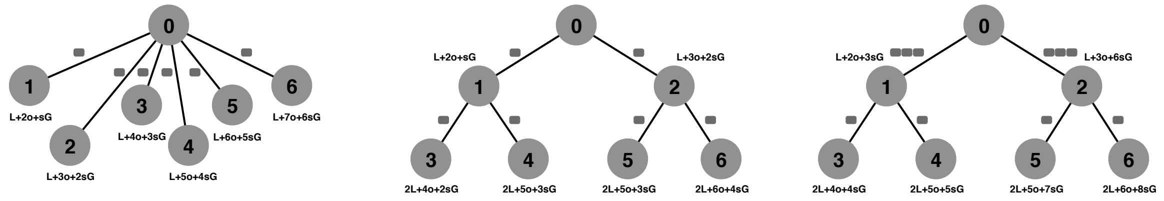
Trees can be used to implement any collective communication. In these algorithms, processes are arranged in a tree shape and messages are flowing from parents to children or vice versa, depending on the collective operation. Some collectives require personalized data (e.g., scatter/gather) such that the messages grow or shrink as they are sent along the tree while other operations either replicate or reduce the data (e.g., reduce, broadcast) leading to constant-size messages. Trees are often used for communicating small messages because in most cases, leave processes only receive messages and are thus not able to use their own send bandwidth. Simple pipelines (i.e., degenerated regular trees) that minimize the number of leaves often provide excellent and simple solutions for very large message sizes. We will also discuss double-tree algorithms that improve the latency over such simple pipelines.

While trees can be used to implement any collective, they may incur a higher cost if they need to be combined. For example, unrooted collectives where all processes receive the result (e.g., allreduce) require communication up and down a tree. These communications can be efficiently implemented using distribution patterns that can also be seen as intertwined trees rooted at each process. A third class of specialized algorithms takes advantage of either specific hardware properties such as topology or multicast semantics or specific semantics of the collective problem.

We now proceed to describe existing tree algorithms followed by distribution patterns. We conclude this subsection by referencing several specialized algorithms. A simple lower bound for the runtime of all algorithms is $\Omega(o \log P) + sG$ because data needs to reach all processes and data must be sent at least once. Similarly, a lower bound to the energy consumption is $(P-1)(e+sE)$ and a lower bound for the memory consumption is d because each process must receive the data once. We will provide exact and simplified models for each algorithm; the simplified models use mixed asymptotic notation for $s \to \infty$ and $P \to \infty$ to facilitate highest intuition.

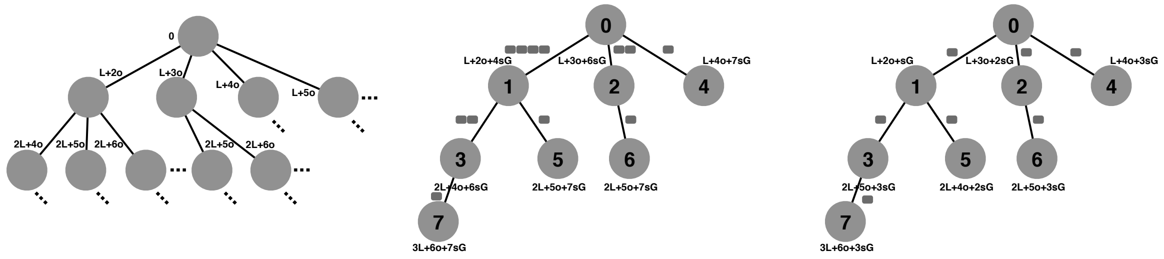
3.1.1. Flat Tree Algorithms

We start with the simplest algorithm for collective operations—a flat tree (FT) [25] in which a single processor sends messages to all destinations directly. Figure 1a provides an example of such a tree for a non-personalized or personalized operation. The gray squares at communication edges denote the communicated data of size s. The annotations in this and the following figures denote the finishing times of the processes in the example. In all figures, we assume that data is sent to the children of a process in the order drawn, beginning with the leftmost. Though simplicity of the algorithm is a clear advantage, its sequential communication limits performance. The time to perform such an operation (personalized or not) is $T_{\text{FT}} = L + oP + sG(P-1) = (o+sG)P - \mathcal{O}(s)$ in the LogGP model. The dynamic energy consumption of such a communication can be estimated as $D_{\text{FT}} = (P-1)(e+sE) = P(e+sE) - \mathcal{O}(s)$. The maximally needed storage at the root of the tree is $M_{\text{FT}} = d(P-1)$.



a) Flat tree (both). b) Binary tree (non-personal). c) Binary tree (personal).

Figure 1. Flat and binary trees ($k = 2$) with seven processes ($P = 7$) in personal and non-personal configurations.



a) Optimal tree (non-personal). b) Binomial tree (personal). c) Binomial tree (non-personal).

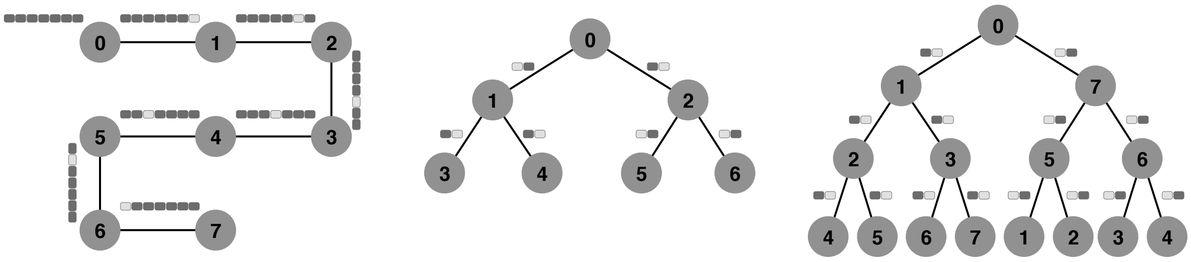
Figure 2. Optimal Fibonacci trees and binomial trees with eight processes ($P = 8$) in personal and non-personal configurations.

3.1.2. Regular Trees

A widely used topology for rooted collective operations is based on regular trees. In such trees, processes perform communications concurrently and thus achieve better performance than flat trees. Trees are called regular when each inner process has the same number of child nodes. We call trees with k such children per process k-ary trees; in this sense, flat trees can be seen as regular trees with $k = (P-1)$.
To illustrate the concept, Figures 1b and 1c show non-personalized and personalized communications along a binary tree, respectively. General k-ary trees (KT) require $\log_k(P)$ total parallel communication steps. In particular, the time of a k-ary tree algorithm for a non-personalized operation is $T_{\text{KT}} = (L + k(o+sG)+o)\lfloor \log_k P \rfloor = (L+ko+ksG)\log_k P - \mathcal{O}(s) + \log_k P \cdot \mathcal{O}(1)$. The dynamic energy model for the same algorithm is $D_{\text{KT}} = (P-1)(e+sE) = P(e+sE) - \mathcal{O}(s)$. The storage requirements for k-ary trees are $M_{\text{KT}} = kd$ because each process sends to at most k children.
For personalized communications on full trees (which we mark with a tilde above the virtual topology type, e.g., $\widetilde{\text{KT}}$), the communication time can be modeled with $T_{\widetilde{\text{KT}}} = \lfloor \log_k P \rfloor (L + o(k+1)) + sG \sum_{i=0}^{\lfloor \log_k P \rfloor} (\lfloor \log_k P \rfloor - i) k^i = (L+ko)\log_k P + sGP \cdot \mathcal{O}(1) + \mathcal{O}(\log P - s)$. Here, one can simply count the packet along the rightmost path assuming that messages are sent to each left child first. The dynamic energy consumption is $D_{\widetilde{\text{KT}}} = e(P-1) + sE \cdot k^{\lfloor \log_k P \rfloor} \sum_{i=0}^{\lfloor \log_k P \rfloor - 1} (\lfloor \log_k P \rfloor - i) \frac{1}{k^i} \approx P(e + sE \log_k P) + \mathcal{O}(sP)$ (for large k) and the memory consumption is $M_{\widetilde{\text{KT}}} = kd$ as in the non-personalized case.
Pjesivac-Grbovic et al. [33] use splitted binary trees (SB) to accelerate non-personalized communications. They use a normal binary tree but instead of distributing the whole message along each tree edge, the message is divided into two parts. The first part is sent to the nodes of the left subtree of the root, while the second part is distributed among nodes of the right subtree of the root. Once a node received the data and sent it on to its children, it also sends it to its own counterpart in the other subtree. The approximate time of the splitted binary tree algorithm is a combination of the normal binary tree non-personalized algorithm with $\frac{s}{2}$ data and a full exchange: $T_{\text{SB}} = (L + 2(o + \frac{s}{2}G) + o)\lfloor \log_2 P \rfloor + 2o + L + \frac{s}{2}G = \log_2 P(L + 3o + sG) + s\log_2 P \cdot \mathcal{O}(1)$. The estimated dynamic energy for this algorithm is $D_{\text{SB}} = 2(e + \frac{s}{2}E)(P-1) = P(2e+sE) - \mathcal{O}(s)$ while the memory model is $M_{\text{SB}} = 3d$.

3.1.3. Irregular Trees

While simplicity of regular tree algorithms is a strong advantage and they are asymptotically optimal for small messages, they are generally not strictly optimal. For example, Karp et al. [24] demonstrate that Fibonacci trees are optimal for single-item broadcasts and thus non-personalized tree communication in the LogP model. Figure 2a shows the optimal tree construction, each node is labeled with its arrival time



a) Linear pipeline ($P=8$). b) Tree pipeline ($P=7$). c) Double tree ($P=8$).

Figure 3. Non-personalized pipelined trees and double trees with seven or eight processes.

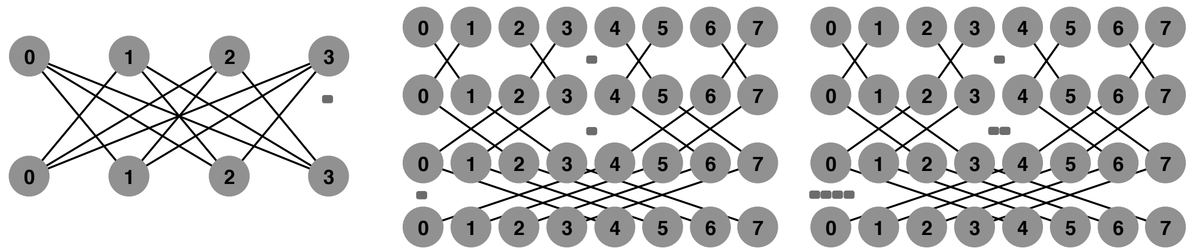
and the best broadcast tree for P processes is constructed from the P nodes with the smallest labels. Karp et al. also state that, if f_n and f_{n+1} are the consecutive members of the generalized Fibonacci sequence s.t. $f_n < P-1 < f_{n+1}$, the lower bound for broadcasting s items is $n+1+L+(s-1)-\lfloor\sum_t \frac{f_t}{P-1}\rfloor$ [24] (assuming $g=1$, $o=0$, $G=0$). For personalized tree communication, Alexandrov et al. [1] as well as Iannello [22] show that in the LogGP model the usage of irregular trees for virtual topologies allows to achieve better performance. Both algorithms are hard to derive and have not been used in practice to the best of the authors knowledge.

A much simpler class of irregular trees that improves over regular trees are k-nomial trees. Here, we discuss the most-used binomial tree (BT) ($k=2$) as example and we assume that P is a power of two. The runtime of non-personalized binomial trees is $T_{\text{BT}} = (L+2o+sG)\log_2 P$, their dynamic energy consumption is $D_{\text{BT}} = (P-1)(e+sE) = P(e+sE)-\mathcal{O}(s)$, and their memory use is $M_{\text{BT}} = d\log_2 P$ at the root process. The runtime of personalized binomial trees is $T_{\widetilde{\text{BT}}} = (2o+L)\log_2 P + sG(P-1) = (2o+L)\log_2 P + sGP - \mathcal{O}(s)$, their dynamic energy consumption is $D_{\widetilde{\text{BT}}} = e(P-1)+sE\frac{P}{2}\log_2 P = Pe+sE\frac{P}{2}\log_2 P-\mathcal{O}(1)$, and their memory consumption is $M_{\widetilde{\text{BT}}} = d\log_2 P$. Figures 2b and 2c show examples for personalized and non-personalized binomial trees.

Binomial tree algorithms are commonly used for small messages; for larger messages, more complex algorithms provide better results (see, for example, various algorithms proposed by Van de Geijn et al. [5, 41, 44]). We will now discuss pipelined trees that have a similar goal to improve bandwidth.

3.1.4. Pipelined Tree Algorithms

Pipeline algorithms are based on the idea to divide a large message into multiple small pieces and to distribute these pieces among processors in a pipeline fashion [33, 38]. Here, different virtual topologies can be utilized for transmitting the data. Linear pipelines as illustrated in Figure 3a are simplest while tree pipelines as illustrated in Figure 3b allow to reduce latencies. As before, our models assume that data is sent down the left pipe first and then alternating. We also assume in this case that the send and receive overheads (o) can be charged simultaneously (e.g., in a multicore environment). Pipelines are often used as building blocks for more complex algorithms [40]. For example, in a non-personalized setting, the runtime of a pipelined binary tree (PBT) algorithm can be estimated as $T_{\text{PBT}} = 2(o+\frac{s}{N}G)N+L+o+(o+2(o+\frac{s}{N}G)+L)(\lfloor\log_2 P\rfloor-1) = \log_2 P(L+3o+\frac{sG}{N})+\mathcal{O}(N+s)$, where N is the number of pieces into which the message is divided and it typically depends on s. The corresponding dynamic energy model is $D_{\text{PBT}} = (P-1)(e+\frac{s}{N}E)N = P(Ne+sE)-\mathcal{O}(s)$ and the storage requirement is $M_{\text{PBT}} = 2dN$. For personalized communications, the runtime is $T_{\widetilde{\text{PBT}}} = o+(o+(2^{\lfloor\log_2 P\rfloor}-1)\frac{s}{N}G)2N+L+(o+L)(\lfloor\log_2 P\rfloor-1)+2(o(\lfloor\log_2 P\rfloor-1)+\frac{s}{N}G(2(2^{\lfloor\log_2 P\rfloor-1}-1)-(\lfloor\log_2 P\rfloor-1))) = (L+3o)\log_2 P+sG(P\mathcal{O}(1)-\frac{2}{N}\log_2 P)+N\mathcal{O}(1)$, the dynamic energy



a) Direct sends (both).

b) Butterfly (non-personal).

c) Butterfly (personal).

Figure 4. Different distribution algorithms for unrooted collectives. Only one data packet is shown at each stage for readability.

consumption is $D_{\widetilde{\text{PBT}}} = N(2e(2^{\lfloor \log_2 P \rfloor} - 1) + \frac{sE}{N} 2^{\lfloor \log_2 P \rfloor}(1 + 2(\lfloor \log_2 P \rfloor - 1) + (2^{1-\lfloor \log_2 P \rfloor} - 1))) = NP\mathcal{O}(1)e + sEP\mathcal{O}(1)(\log_2 P + \frac{1}{P\mathcal{O}(1)} - \mathcal{O}(1))$, and the memory overhead is $M_{\widetilde{\text{PBT}}} = 2Nd$.

3.1.5. Double Trees

While pipelined trees improve the overall bandwidth utilization, they are still not optimal. The reason for this is that the leaves in the tree never transmit messages and thus do not contribute their bandwidths. To use the leaves' bandwidth, one can employ two trees with different structure (leave nodes) such that each node sends eventually. Sanders and Träff [39, 40] demonstrate such a two-tree virtual topology that achieves full bandwidth, extending and simplifying an earlier algorithm [45]. The authors utilize two trees so that the interior nodes of the first tree correspond to the leaf nodes of the second tree and vice versa (see Figure 3c). They also describe a scheduling algorithm to define from which parent node the data should be received at the current step and to which child node the data should be forwarded. The approach only applies to non-personal communications. The runtime of this double tree (DT) algorithm is $T_{\text{DT}} = (L{+}3o{+}sG){+}T_{\text{PBT}}\left(\frac{s}{2}\right)$. Its dynamic energy consumption is $D_{\text{DT}} = 2(e{+}\frac{s}{2}E){+}2D_{\text{PBT}}\left(\frac{s}{2}\right)$ and the memory consumption for this approach is $M_{\text{DT}} = 2dN$.

This algorithm concludes our treatment of successively more complex algorithms for rooted collective communications. We now proceed to discuss distribution patterns such as direct send, dissemination, and butterfly algorithms for unrooted collective communications.

3.1.6. Direct Sends

In unrooted collectives, typically all processes receive some data from every other process, either personalized or reduced. This can be achieved by a direct send (DS) topology among all processes. This is similar to a flat tree rooted at each process. The runtime for the personalized as well as the non-personalized variant is $T_{\text{DS}} = L + (P-1)(o + sG) = P(o + sG) - \mathcal{O}(s)$, the energy consumption is $D_{\text{DS}} = P(P-1)(e + sE) = P^2(e + sE) - \mathcal{O}(Ps)$, and the memory consumption at each process is $M_{\text{DS}} = (P-1)d$. Figure 4a illustrates the DS scheme.

3.1.7. Dissemination and Butterfly Algorithms

The well-known Butterfly (BF) graph [8] implements a binary scheme to quickly exchange data among all processes which can be applied if P is a power of two. The dissemination approach [15] generalizes this scheme to arbitrary numbers of processes. Here, we limit ourselves to the simpler case where P is a power of two. In the Butterfly pattern, data is communicated between processes with exponentially growing distances, i.e., in the k-th step, nodes at distance 2^k from each other exchange data. Thus, $\log_2 P$ steps are required to complete the communication.

The non-personalized version of butterfly executes in time $T_{\mathrm{BF}} = (2o + sG + L)\log_2 P$, with a dynamic energy consumption of $D_{\mathrm{BF}} = (e + sE)P\log_2 P$, and with a memory consumption of $M_{\mathrm{BF}} = d\log_2 P$. The well-known recursive doubling algorithm [44] as well as the Bruck algorithm [9] implement a personalized variant of the Butterfly pattern. If we ignore local data shuffles, then the runtime of this personalized algorithm is $T_{\widetilde{\mathrm{BF}}} = (2o + L)\log_2 P + Gs(P-1) = (2o + L)\log_2 P + sGP - \mathcal{O}(s)$. Its energy consumption can be modeled as $D_{\widetilde{\mathrm{BF}}} = eP\log_2 P + sE(P-1)P = P(e\log_2 P + sEP) - \mathcal{O}(sP)$ and its memory requirement is $M_{\widetilde{\mathrm{BF}}} = d\log_2 P$. Each model increases with a multiplicative constant if the number of processes is not equal to a power of two [44]. Figures 4b and 4c illustrate the Butterfly pattern with eight processes in non-personalized and personalized configurations, respectively.

3.1.8. More Specific Algorithms

Several researchers developed algorithms that are tuned to particular properties of the machine. For example, several algorithms that specialize to the network topology exist. Some others utilize special hardware features. We provide some examples here but this list is not meant to be complete.

Hardware-specific algorithms Ali et al. [2] provide algorithms for collective communications on the Cell B.E. chip, Panda et al. demonstrate a series of algorithms tuned to InfiniBand networks and RDMA systems [27, 42], and Almasi et al. [3] show optimization techniques for the BlueGene/L Torus network.

Topology-aware algorithms There is a class of algorithms that take the network topology and congestion into account. For example, Sack and Gropp [36, 37] introduce a congestion-aware model for network communication. In the same articles they propose a recursive-doubling distance-halving algorithms for the allgather and reduce scatter collectives for Clos and Torus networks. Payne et al. [32] describe several algorithms on how to implement some reduction operations on a 2-dimensional mesh and Barnett et al. [6] develop a broadcasting algorithm for the mesh topology. Watts and Van de Geijn [48] show a pipelined broadcast for mesh architectures and Chan et al. [10] show how to utilize all available links in Torus networks.

Using Unreliable Multicast Hardware Other algorithms base on special hardware features such as multicast [11]. Multicast packets can be lost and in order to guarantee reliable transmission, recovery algorithms are necessary. One such recovery protocol is presented by Hoefler et al. [20]. Their protocol combines InfiniBand (or Ethernet) unreliable multicast with reliable point-to-point messages to achieve a with high probability constant-time ($\mathcal{O}(1)$ complexity) broadcast operation. Using these special hardware features allows us to circumvent the logarithmic lower bound.

3.2. Implementing Collective Operations

We now briefly discuss how the modeled algorithms can be combined to implement collective operations. We follow our previous categorization into rooted collectives implemented by personalized or non-personalized trees and unrooted collectives implemented by personalized or non-personalized distribution algorithms.

3.2.1. Rooted Collectives

Table 1 shows an overview of the tradeoffs in various personalized and non-personalized tree algorithms. We use the previously introduced subscripts as abbreviation: FT for flat trees, KT for k-ary regular trees, BT for binomial trees, PBT for pipelined binary trees, and DT for double trees. Abbreviations with a tilde on top, e.g., $\widetilde{\mathrm{FT}}$, denote personalized versions of the algorithms.

	FT, $\widetilde{\text{FT}}$	KT	$\widetilde{\text{KT}}$	BT	$\widetilde{\text{BT}}$	PBT	$\widetilde{\text{PBT}}$	DT
T	$P(o+sG)$	$(L+ko+ksG)\log_k P$	$(L+ko)\log_k P +sGP$	$(L+2o +sG)\lg P$	$(L+2o)\lg P +sGP$	$(L+3o+\frac{sG}{N})\lg P$	$sG(P-\frac{2}{N}\lg P)+(L+3o)\lg P$	$(L+3o+sG) +T_{PBT}\left(\frac{s}{2}\right)$
D	$P(e+sE)$	$P(e+sE)$	$P(e+sE\log_k P)$	$P(e+sE)$	$P(e+\frac{sE}{2}\lg P)$	$P(Ne+sE)$	$P(Ne+sE\lg P)$	$2(e+\frac{s}{2}E) +2D_{PBT}\left(\frac{s}{2}\right)$
M	Pd	kd	kd	$d\lg P$	$d\lg P$	$2dN$	$2dN$	$2dN$

Table 1. Overview of tree algorithms for rooted collectives
(minor terms are dropped, $\lg$ stands for $\log_2$).

Broadcast/Reduce Broadcast and reduce are structurally similar but very different in their semantics. In a broadcast, a single message of size s is distributed (copied) from a designated root process to all other $P-1$ processes. In a reduction, each process contributes a message of size s. The associative (and often commutative) operator $\oplus$ combines all P messages into a single result of size s at a designated root process: $r = m_1 \oplus m_2 \oplus m_3 \oplus \cdots \oplus m_P$.
Both collectives can be implemented with non-personalized tree algorithms. Binomial and binary trees are commonly used for implementations of small-message broadcast and reduction [43, 44]. Large-message operations can be implemented with double trees. Our models in Table 1 show that, for non-personalized communications, double-trees are the best contenders in terms of runtime (for all s and P). However, they require more dynamic energy and memory due to the pipelining of messages. The exact number of additional messages sent depends on the number of pipeline segments N, which in turn is chosen based on the LogGP parameters and s. If the memory is constrained, then pipelining would be limited, possibly leading to suboptimal performance. All non-pipelined algorithms are work-optimal and thus consume the minimal energy. Regular k-ary trees have only constant memory overhead and are thus best for execution in very limited offload settings.

Scatter/Gather In a scatter, a designated process (root) sends personalized messages, each of size s, to $P-1$ other processes. In a gather, the root process receives different messages, each of size s, from $P-1$ processes and stores them locally. Both collectives can be implemented using personalized tree algorithms. For example, Binomial trees have been used to perform both, scatter and gather [4].
Our models in Table 1 show that, for personalized communications with small P, flat trees are best. Other regular and irregular trees reduce the latency to a logarithmic term and thus benefit large P but they are not work-optimal and send multiple messages multiple times and thus harm large s. For large s and small P one can use linear pipelines to utilize the bandwidth of all processes as discussed before. Alexandrov et al. [1] formulate the condition for an optimal gather tree in LogGP but to the best of the authors' knowledge, no practical algorithm is known that achieves this bound. In terms of energy, we remark that all tree algorithms increase dynamic energy consumption significantly in comparison to a flat tree. Memory consumption is similar to the non-personalized algorithms where the pipelining versions may dominate and k-ary regular trees are minimal for small k.

3.2.2. Unrooted Collectives

Table 2 shows an overview of various distribution algorithms and trees that can be used for unrooted collectives. We use the previously defined abbreviations for distribution algorithms: DS for direct send and BF for Butterfly. We compare these to implementations with two combined trees, such as a k-ary tree to reduce data towards a root followed by a second k-ary tree to broadcast data to all processes, which we denote as 2xKT. We only combine trees of similar nature and show some select examples even though combinations of any two trees can be used in practice.

	DS, $\widetilde{\text{DS}}$	BF	$\widetilde{\text{BF}}$	2xKT	2x$\widetilde{\text{KT}}$	2xPBT	2x$\widetilde{\text{PBT}}$
T	$P(o+sG)$	$(L+2o+sG)\log_2 P$	$\log_2 P(2o+L)$ $+sGP$	$2(L+ko+$ $ksG)\log_k P$	$2(L+ko)\log_k P$ $+2sGP$	$2(L+3o$ $+\frac{sG}{N})\lg P$	$2sG(P-\frac{2}{N}\lg P)+$ $2(L+3o)\lg P$
D	$P^2(e+sE)$	$P\log_2 P(e+sE)$	$eP\log_2 P+$ $P^2 sE$	$2P(e+sE)$	$2P(e+sE\log_k P)$	$2P(Ne+sE)$	$2P(Ne+$ $sE\lg P)$
M	Pd	$d\log_2 P$	$d\log_2 P$	$2kd$	$2kd$	$4dN$	$4dN$

Table 2. Overview of algorithms for unrooted collectives (minor terms are dropped, $\alpha = L + o + sG$).

Allreduce/Barrier Allreduce is similar to reduce in that all processes contribute a message of size s and $r = m_1 \oplus m_2 \oplus m_3 \oplus \cdots \oplus m_P$ is computed. However, as opposed to reduce, the final r will be distributed to all processes. The Barrier collective guarantees that no process completes the operation before all processes called it. It is similar to allreduce with a zero-sized message and is commonly implemented using the same algorithms. Both collectives can be implemented using two trees, a reduction to a root followed by a broadcast to all processes as in [21]. However, a more time-efficient implementation would be non-personalized distribution such as the Butterfly pattern [31, 34, 49].

The models in Table 2 suggest that, for non-personalized communication, Butterfly patterns are fastest for all s and P. However, their dynamic energy consumption is asymptotically higher than the combination of two trees. Combining two pipelined trees can improve tree performance for large messages. Butterfly consumes logarithmically growing memory at each node, two k-ary trees could reduce this memory consumption to a constant.

Allgather/Alltoall Allgather is similar to a gather but the result is distributed to all processes. A simple but slow implementation would be a gather followed by a broadcast. In alltoall, each process has P messages of size s. Each of these messages is sent to another target process, so that each process sends and receives $P-1$ messages (and an implicit message to itself). Direct send or Bruck's algorithm (using a personalized Butterfly communication) can be used to implement such collective operations. In addition, these operations can be implemented using personalized trees that gather the result to a single node and broadcast it to all nodes.

The models in Table 2 suggest that, for personalized communication, Butterfly patterns are fastest for all small s and large P but quickly become inefficient with growing s. Direct sends are most efficient for large s and small P. Tree patterns are always more expensive in terms of runtime and energy consumption than distribution patterns. However, tree patterns can provide a constant memory consumption while other patterns have linear or logarithmic memory requirements in P.

3.2.3. Other Collectives

Scans/Reduce Scatter In prefix scan operations, each process specifies a message of size s and received the partial sum of all messages specified by processes with a lower id than itself. I.e., the process with id k receives $r_k = m_1 \oplus m_2 \oplus m_3 \oplus \cdots \oplus m_k$ (assuming $k > 3$). A reduce scatter performs a reduction of a message of size Ps specified at each process. Then, messages of size s are scattered to each P process. Both steps are performed together so that algorithms can optimize them as a single step. Reduce scatter can be implemented by a simple reduce followed by a scatter and scans can be implemented by rooting a different reduction tree at each process. However, merging the trees can lead to substantial performance improvements for reduce scatter [22] as well as scans.

Neighborhood Collectives MPI-3 introduces neighborhood collective operations [18] where the programmer can specify any communication pattern and in this way build his own collective communication operation. For example, one can express all non-reduction collective operations as neighborhood collectives. However, the expressiveness of this operation comes at the cost of optimizability. Thus, there are no generic optimization algorithms for these operations yet.
For the purpose of the analyses in this paper, we ignore irregular/vector collective operations.

4. Discussion and Open Problems

We now conclude our theoretical analyses with a brief summary of the lessons learned followed by an outlook to important open problems and future research directions in the area of optimizing collective communications.

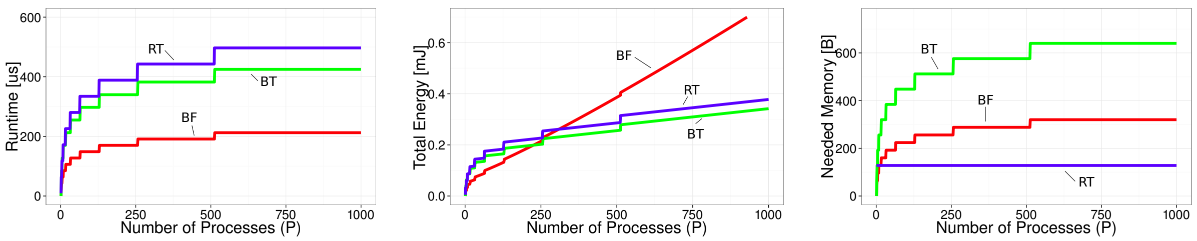
4.1. Approaching the Optimal

Some systems combine existing algorithms using an auto-tuning approach for algorithm selection [46]. Pjesivac-Grbovic et al. [33] for example utilize decision trees to select the best algorithm at runtime while Faraj and Yuan [13] use collective building blocks to tune them to a particular network topology. Yet, all these approaches are not strictly optimal. Selecting different algorithms and parameters for them automatically may yield significant speedups over any single algorithm. However, the problem of attaining the best bounds in terms of latency and bandwidth in the full spectrum of possible datasizes s and process numbers P remains open for many personalized communication algorithms.

Problem 1: Runtime-optimal collective algorithms We identified four essential classes of algorithms that need to be developed to attack this problem: trees with personalized and non-personalized data and dissemination mechanisms with personalized and non-personalized data. While several discrete algorithms exist for both, we expect that a general latency- and bandwidth-optimal solution will significantly improve upon the state-of-the-art.

4.2. Energy, Memory, and Runtime Tradeoffs

In our analysis, we identified several problems where algorithms with a smaller runtime consume more energy than algorithms with a larger runtime and vice-versa. In addition, we found that the best algorithms are generally not space optimal. This means that offloading devices with strictly limited resources may not be able to use the best known algorithms. To illustrate the tradeoff, we plot our models for a set of parameters chosen to represent an InfiniBand network architecture. These parameters are approximate and vary across installations, however, they provide insight into the tradeoffs between energy consumption and runtime.
As LogGP parameters, we use previously reported values measured for InfiniBand using MPI: $L = 6\ \mu$s, $o = 4.7\ \mu$s, $G = 0.73$ ns/B [16]. Kim et al. [26] model the memory read and write power consumption per MTU packet (2048 B) per switch as 8.1 pJ. We use this data to approximate the NIC power consumption assuming that each Byte in a packet is read and written once and a single packet is needed to send a 0-Byte messages. Thus, we assume $e = 16.5$ pJ, $E = 8.1$ nJ/B, and a static NIC chip power of $P = 0.5$ W for our model. For the memory overhead, we assume that each descriptor



a) Runtime. b) Energy. c) Memory.

Figure 5. Example for the tradeoff between runtime, energy, and memory for non-personalized distribution (e.g., allreduce) of 8 Bytes for 2-ary regular trees (RT), binomial trees (BT), and Butterfly (BF).

stores a pointer, an offset, a trigger counter, and a target address. We assume that each of these fields is represented by a 64-Bit number, thus $d = 32$ B.

Figure 5 shows one particular example for a non-personal distribution communication that could be used to implement allreduce. We compare only three different options: two 2-ary trees, two binary trees, and Butterfly to instantiate the intuition from Table 2 with real-world parameters. The runtime model shows that the Butterfly algorithm is by far the best option followed by the binomial tree and the binary tree. However, in the energy model, Butterfly is far worse than both, binomial and binary trees for large numbers of processes. In fact, its dynamic energy consumption is always higher than the trees but for small process counts, the performance advantage reduces the static energy consumption in comparison to the trees. The memory model shows that the regular binary tree has the lowest, even constant memory consumption per process followed by Butterfly and binary tree. We observe that depending on the target metric, each of the three algorithms can perform best: Butterfly has the best performance, binomial trees use the least energy, and binary trees require the least memory in the network interface.

Problem 2: Energy-optimal collective algorithms Finding the energy-optimal algorithm for a given set of parameters (the dynamic energy consumption with e and E and the static power consumption P) for each collective operation remains an open and challenging topic as it requires to optimize time to minimize static energy in combination with the dynamic energy consumption. The optimal algorithm in terms of dynamic energy is often the simple linear algorithm that would result in excessive static energy consumption. The exact tradeoff between these algorithms is determined by the energy and runtime models as well as the energy and runtime parameters.

Problem 3: Pareto-optimum for energy and runtime If both previous problems are attained, one could phrase the Pareto-optimal region for the energy consumption versus the runtime. This allows to optimize the runtime in energy-constrained systems as well as the energy consumption in real-time systems. In power-constrained settings, one could also limit the dynamic energy consumption to stay within certain limits.

Problem 4: Optimal neighborhood collective operations The problem of optimizing neighborhood collectives is not well understood. Since they can represent any arbitrary collective operation, an optimal solution (in terms of energy consumption or runtime) would also yield optimal solutions for all MPI collectives.

4.3. Tradeoffs for Offload Architectures

Collective offload architectures often offer limited space on the device. The optimization problem (in terms of power and energy) can now be formulated under the restriction of limited space on the device.

Our models show that each algorithm can be implemented with constant space per device. However, we also show that the necessary algorithms are slower than the best known algorithms. Interestingly, the slowdown of the constant-space algorithms seems to be limited to a factor of two compared to the best known practical algorithm. The difference may be higher when compared to close-to-optimal solutions such as Fibonacci trees and optimal personalized schedules.

We also found that many best known algorithms utilize pipelining, a technique where the memory consumption grows with the size of the sent data. Designers of offload architectures may consider to support pipelining of N messages with a constant-size operation. In addition, one could allow to offload simple programs to the network card that generate sends on the fly without pre-programming everything at initialization time.

Problem 5: Optimal memory-constrained collectives The problem to determine the runtime- or energy-optimal schedule under the constraint of space on the offloading device may be important to support future collective offload architectures.

5. Conclusions

This study provides an overview of existing collective algorithms and implementations. We describe the most common algorithms for implementing collective operations in practice. However, our list is not meant to be exhaustive. We classify these algorithms into three groups: tree-shaped algorithms, distribution algorithms, and optimized schedules. The first two groups base on virtual topologies which can be used in a personalized and non-personalized setting. The last group includes optimized and specialized messaging schedules for particular cases.

We derive runtime, energy, and memory consumption models for each algorithm and compare the algorithms within each group. Our models and comparisons provide fundamental insights into the nature of these algorithms and various tradeoffs involved. For example, we show that runtime-optimal algorithms always exhibit non-optimal dynamic energy consumption. In the case of non-personalized distribution, the energy consumption of the fastest algorithm is asymptotically higher than the consumption of an algorithm that is only a slower by a constant. We also show that optimal algorithms always require more memory in offload devices than other algorithms that are only slower by a constant. This provides interesting optimization problems to find the best tradeoffs between runtime, energy, and memory consumption in offload devices.

In our theoretical study, we identified several research problems and open questions. We believe that it is most important to understand the tradeoff between energy and runtime and possibly memory consumption in offload devices. It is also interesting to design offloading protocols and devices that require minimal storage in the network architecture. In addition, a generic framework to design close-to-optimal schedules for predefined as well as neighborhood collective operations would be a valuable contribution to the state of the art.

References

1. A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating Long Messages into the LogP Model—One Step Closer Towards a Realistic Model for Parallel Computation. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '95, pages 95–105, New York, NY, USA, 1995. ACM.

2. Q. Ali, S. P. Midkiff, and V. S. Pai. Efficient High Performance Collective Communication for the Cell Blade. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 193–203, New York, NY, USA, 2009. ACM.
3. G. Almási, P. Heidelberger, C. J. Archer, X. Martorell, C. C. Erway, J. E. Moreira, B. Steinmacher-Burow, and Y. Zheng. Optimization of MPI Collective Communication on BlueGene/L Systems. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, pages 253–262, New York, NY, USA, 2005. ACM.
4. M. Banikazemi and D. K. Panda. Efficient scatter communication in wormhole k-ary n-cubes with multidestination message passing, 1996.
5. M. Barnett, S. Gupta, D. G. Payne, L. Shuler, R. Geijn, and J. Watts. Interprocessor Collective Communication Library (InterCom). In *Proceedings of the Scalable High Performance Computing Conference*, pages 357–364. IEEE Computer Society Press, 1994.
6. M. Barnett, D. G. Payne, and R. V. D. Geijn. Optimal broadcasting in mesh-connected architectures. Technical report, 1991.
7. Barrett, B.W. and Brightwell, R. and Hemmert, S. and Pedretti, K. and Wheeler K. and Underwood, K.D. and Reisen, R. and Maccabe, A.B., and Hudson, T. *The Portals 4.0 network programming interface*. Sandia National Laboratories, November 2012. Technical Report SAND2012-10087.
8. E. D. Brooks, III. The butterfly barrier. *Int. J. Parallel Program.*, 15(4):295–307, Oct. 1986.
9. J. Bruck, C.-T. Ho, S. Kipnis, E. Upfal, and D. Weathersby. Efficient algorithms for all-to-all communications in multiport message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 8:1143 – 1156, 1997.
10. E. Chan, R. van de Geijn, W. Gropp, and R. Thakur. Collective communication on architectures that support simultaneous communication over multiple links. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 2–11, New York, NY, USA, 2006. ACM.
11. H. A. Chen, Y. O. Carrasco, and A. W. Apon. MPI Collective Operations over IP Multicast. In *Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, IPDPS '00, pages 51–60, London, UK, UK, 2000. Springer-Verlag.
12. G. Fagg, G. Bosilca, J. Pjeivac-Grbovic, T. Angskun, and J. Dongarra. Tuned: An Open MPI Collective Communications Component. In *Distributed and Parallel Systems*, pages 65–72. Springer US, 2007.
13. A. Faraj and X. Yuan. Automatic generation and tuning of mpi collective communication routines. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, pages 393–402, New York, NY, USA, 2005. ACM.
14. R. Graham, S. Poole, P. Shamis, G. Bloch, N. Bloch, H. Chapman, M. Kagan, A. Shahar, I. Rabinovitz, and G. Shainer. ConnectX-2 InfiniBand management queues: First investigation of the new support for network offloaded collective operations. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 53–62, 2010.
15. D. Hensgen, R. Finkel, and U. Manber. Two algorithms for barrier synchronization. *Int. J. Parallel Program.*, 17(1):1–17, Feb. 1988.
16. T. Hoefler, A. Lichei, and W. Rehm. Low-Overhead LogGP Parameter Assessment for Modern Interconnection Networks. In *Proceedings of the 21st IEEE International Parallel & Distributed Processing Symposium, PMEO'07 Workshop*. IEEE Computer Society, Mar. 2007.
17. T. Hoefler, A. Lumsdaine, and W. Rehm. Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI. In *Proceedings of the 2007 International Conference*

on High Performance Computing, Networking, Storage and Analysis, SC07. IEEE Computer Society/ACM, Nov. 2007.

18. T. Hoefler and T. Schneider. Optimization Principles for Collective Neighborhood Communications. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 98:1–98:10. IEEE Computer Society Press, Nov. 2012.
19. T. Hoefler, T. Schneider, and A. Lumsdaine. Characterizing the Influence of System Noise on Large-Scale Applications by Simulation. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*, Nov. 2010.
20. T. Hoefler, C. Siebert, and W. Rehm. A practically constant-time MPI Broadcast Algorithm for large-scale InfiniBand Clusters with Multicast. In *Proceedings of the 21st IEEE International Parallel & Distributed Processing Symposium (CAC'07 Workshop)*, page 232, Mar. 2007.
21. L. P. Huse. Collective communication on dedicated clusters of workstations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 469–476. Springer-Verlag, 1999.
22. G. Iannello. Efficient Algorithms for the Reduce-Scatter Operation in LogGP. *IEEE Trans. Parallel Distrib. Syst.*, 8(9):970–982, Sept. 1997.
23. K. Kandalla, H. Subramoni, J. Vienne, S. Raikar, K. Tomko, S. Sur, and D. Panda. Designing Non-blocking Broadcast with Collective Offload on InfiniBand Clusters: A Case Study with HPL. In *High Performance Interconnects, 2011 IEEE 19th Annual Symposium on*, pages 27–34, Aug 2011.
24. R. M. Karp, A. Sahay, E. E. Santos, and K. E. Schauser. Optimal Broadcast and Summation in the LogP Model. In *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '93, pages 142–153, New York, NY, USA, 1993. ACM.
25. T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang. MagPIe: MPI's Collective Communication Operations for Clustered Wide Area Systems. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '99, pages 131–140, New York, NY, USA, 1999. ACM.
26. E. J. Kim, G. Link, K. H. Yum, N. Vijaykrishnan, M. Kandemir, M. Irwin, and C. Das. A holistic approach to designing energy-efficient cluster interconnects. *Computers, IEEE Transactions on*, 54(6):660–671, Jun 2005.
27. S. Kini, J. Liu, J. Wu, P. Wyckoff, and D. Panda. Fast and scalable barrier using rdma and multicast mechanisms for infiniband-based clusters. In J. Dongarra, D. Laforenza, and S. Orlando, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 2840 of *Lecture Notes in Computer Science*, pages 369–378. Springer Berlin Heidelberg, 2003.
28. V. A. Korthikanti and G. Agha. Towards optimizing energy costs of algorithms for shared memory architectures. In F. M. auf der Heide and C. A. Phillips, editors, *SPAA*, pages 157–165. ACM, 2010.
29. V. A. Korthikanti and G. Agha. Energy-performance trade-off analysis of parallel algorithms for shared memory architectures. In *Sustainable Computing: Informatics and Systems, In Press*, 2011.
30. MPI Forum. *MPI: A Message-Passing Interface standard. Version 3.0*, 2012.
31. P. Patarasuk and X. Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *J. Parallel Distrib. Comput.*, 69(2):117–124, Feb. 2009.
32. B. L. Payne, M. Barnett, R. Littlefield, D. G. Payne, and R. V. D. Geijn. Global combine on mesh architectures with wormhole routing. In *Proc. of 7 th Int. Parallel Proc. Symp*, 1993.
33. J. Pjesivac-Grbovic, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra. Performance analysis of mpi collective operations. In *19th IEEE International Parallel and Distributed Processing Symposium, 2005. Proceedings.*, 2005.

34. R. Rabenseifner. Optimization of collective reduction operations. In *Proceedings of the International Conference on Computational Science*, ICCS 2004, pages 1–9, 2004.
35. S. Ramos and T. Hoefler. Modeling Communication in Cache-Coherent SMP Systems - A Case-Study with Xeon Phi. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, pages 97–108. ACM, Jun. 2013.
36. P. Sack. Scalable collective message-passing algorithms. In *PhD Thesis*. University of Illinois at Urbana-Champain, 2011.
37. P. Sack and W. Gropp. Faster topology-aware collective algorithms through non-minimal communication. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 45–54, New York, NY, USA, 2012. ACM.
38. P. Sanders and J. F. Sibeyn. A bandwidth latency tradeoff for broadcast and reduction. *Inf. Process. Lett.*, 86(1):33–38, Apr. 2003.
39. P. Sanders, J. Speck, and J. Traff. Full bandwidth broadcast, reduction and scan with only two trees. In *Proceedings of the 14th European conference on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 2007.
40. P. Sanders, J. Speck, and J. L. Träff. Two-tree algorithms for full bandwidth broadcast, reduction and scan. *Parallel Comput.*, 35(12):581–594, Dec. 2009.
41. M. Shroff and R. A. V. D. Geijn. CollMark: MPI Collective Communication Benchmark. Technical report, 2000.
42. S. Sur, U. K. R. Bondhugula, A. Mamidala, H. W. Jin, and D. K. Panda. High performance rdma based all-to-all broadcast for infiniband clusters. In *Proceedings of the 12th International Conference on High Performance Computing*, HiPC'05, pages 148–157, Berlin, Heidelberg, 2005. Springer-Verlag.
43. R. Thakur and W. Gropp. Improving the performance of collective operations in MPICH. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. Number 2840 in LNCS, Springer Verlag (2003) 257267 10th European PVM/MPI Users Group Meeting*, pages 257–267. Springer Verlag, 2003.
44. R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of Collective communication operations in MPICH. *International Journal of High Performance Computing Applications*, 19:49–66, 2005.
45. J. L. Träff and A. Ripke. Optimal broadcast for fully connected networks. In *Proceedings of the First International Conference on High Performance Computing and Communications*, HPCC'05, pages 45–56, Berlin, Heidelberg, 2005. Springer-Verlag.
46. S. S. Vadhiyar, G. E. Fagg, and J. Dongarra. Automatically tuned collective communications. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, SC '00, Washington, DC, USA, 2000. IEEE Computer Society.
47. A. Wagner, D. Buntinas, D. Panda, and R. Brightwell. Application-bypass reduction for large-scale clusters. In *Cluster Computing, 2003. Proceedings. 2003 IEEE International Conference on*, pages 404–411, Dec 2003.
48. J. Watts and R. Van De Geijn. A pipelined broadcast for multidimensional meshes. *Parallel Processing Letters*, 5(02):281–292, 1995.
49. W. Yu, D. K. Panda, and D. Buntinas. Scalable, High-performance NIC-based All-to-all Broadcast over Myrinet/GM. In *Proceedings of the 2004 IEEE International Conference on Cluster Computing*, pages 125–134, Washington, DC, USA, 2004. IEEE Computer Society.