High-Performance RMA-Based Broadcast on the Intel SCC
(Full Version)*

Darko Petrović, Omid Shahmirzadi, Thomas Ropars, André Schiper
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
firstname.lastname@epfl.ch

Abstract

Many-core chips with more than 1000 cores are expected by the end of the decade. To overcome scalability issues related to cache coherence at such a scale, one of the main research directions is to leverage the message-passing programming model. The Intel Single-Chip Cloud Computer (SCC) is a prototype of a message-passing many-core chip. It offers the ability to move data between on-chip Message Passing Buffers (MPB) using Remote Memory Access (RMA). Performance of message-passing applications is directly affected by efficiency of collective operations, such as broadcast. In this paper, we study how to make use of the MPBs to implement an efficient broadcast algorithm for the SCC. We propose *OC-Bcast* (*On-Chip Broadcast*), a pipelined k-ary tree algorithm tailored to exploit the parallelism provided by on-chip RMA. Using a LogP-based model, we present an analytical evaluation that compares our algorithm with the state-of-the-art broadcast algorithms implemented for the SCC. As predicted by the model, experimental results show that OC-Bcast attains almost three times better throughput, and improves latency by at least 27%. Furthermore, the analytical evaluation highlights the benefits of our approach: OC-Bcast takes direct advantage of RMA, unlike the other considered broadcast algorithms, which are based on a higher-level send/receive interface. This leads us to the conclusion that RMA-based collective operations are needed to take full advantage of hardware features of future message-passing many-core architectures.

Keywords: Broadcast, Message Passing, Many-Core Chips, RMA, HPC

1 Introduction

Studies on future Exascale High-Performance Computing (HPC) systems point out energy efficiency as the main concern [16]. An Exascale system should have the same power consumption as the existing Petascale systems while providing thousand times more computational power. A direct consequence of this observation is that the number of *flops per watt* provided by a single chip should dramatically increase compared to the current situation [27]. The solution is to increase the level of parallelism on a single chip by moving from multi-core to many-core chips [5]. A many-core chip integrates a large number of cores connected using a powerful Network-on-Chip (NoC). Soon, chips with hundreds if not thousands of cores will be available.

Taking the usual shared memory approach for many-core chips raises scalability issues related to the overhead of hardware cache coherence [20]. To avoid relying on hardware cache coherence, two main alternatives are proposed: (i) sticking to the shared memory paradigm, but managing data coherence in software [27], or (ii) adopting message passing as the new communication paradigm [20]. Indeed, a large set of cores connected through a highly efficient NoC can be viewed as a parallel message-passing system.

The Intel Single-Chip Cloud Computer (SCC) is an example of a message-passing many-core chip [14]. The SCC integrates 24 2-core tiles on a single chip connected by a 2D-mesh NoC. It is provided with on-chip low-latency memory buffers, called *Message Passing Buffers* (MPB), physically distributed across the tiles. *Remote Memory Access* (RMA) to these MPBs allows fast inter-core communication.

The natural choice to program a high-performance message-passing system is to use Single Program Multiple Data (SPMD) algorithms. The Message Passing Interface (MPI) [21] is the *de facto* standard for programming SPMD HPC applications. MPI defines a set of primitives for point-to-point communication, and also defines a set of collective operations, i.e., operations involving a group of processes. Several works study the implementation of point-to-point communications on the Intel SCC [30, 23, 22], but only little attention has been paid to the implementation of collective operations. This paper studies the implementation of collective operations for the Intel SCC. It focuses on the broadcast primitive (*one-to-all*), with the aim of understanding how to efficiently leverage on-chip RMA-based communication. Note that the need for efficient collective operations for many-core systems, especially the need for efficient broadcast, goes far beyond the scope of MPI applications, and is of general interest in these systems [27].

1.1 Related work

A message-passing many-core chip, such as the SCC, is very similar to many existing HPC systems since it gathers a large number of processing units connected through a high-performance RMA-based network. Broadcast has been extensively studied in these systems. Algorithms based on a k-ary tree have been proposed [4]. In MPI libraries, binomial trees and *scatter-allgather* [24] algorithms are mainly considered [11, 26]. A binomial tree is usually selected to provide better latency for small messages, while the *scatter-allgather* algorithm is used to optimize throughput for large messages. These solutions are implemented on top of send/receive point-to-point functions and do not take topology issues into account. This is not an issue for small to medium scale systems like the SCC. However, it has been shown that for mesh or torus topologies, these solutions are not optimal at large scale: non-overlapping spanning trees can provide better performance [1].

As already mentioned, MPI libraries usually implement collective operations on top of classical *two-sided* send/receive communication[1]. To take advantage of the RMA capabilities of high-performance network interconnects such as InfiniBand [2], *one-sided put* and *get* operations, have been introduced [21]. In one-sided communication, only one party (sender or receiver) is involved in the data transfer and specifies

[1] In a classical two-sided communication, a matching operation is required by both parties: *send* by the sender, *receive* by the receiver.

the source and destination buffers. One-sided operations increase the design space for communication algorithms, and can provide better performance by overlapping communication and computation. On the SCC, RMA operations on the MPBs allow the implementation of efficient one-sided communication [19].

Two-sided communication can be implemented on top of one-sided communication [18]. This way, collective operations based on two-sided communication can benefit from efficient one-sided communication. Currently available SCC communication libraries adopt this solution. The RCCE library [19] provides efficient one-sided *put*/*get* operations and uses them to implement two-sided send/receive communication. The RCCE_comm library implements collective operations on top of two-sided communication [7]: the RCCE_comm broadcast algorithm is based on a binomial tree or on *scatter-allgather* depending on the message size. The same algorithms are used in the RCKMPI library [28].

Most high-performance networks provide *Remote Direct Memory Access* (RDMA) [1, 2], *i.e.*, the RMA operations are offloaded to the network devices. Some works try to directly take advantage of these RDMA capabilities to improve collective operations [12, 13, 17, 25]. However, it is hard to reuse the results presented in these works in the context of the SCC for two main reasons: (i) they leverage hardware specific features not available on the SCC, i.e., hardware multicast [13, 17], and (ii) they make use of large RDMA buffers [12, 25], whereas the on-chip MPBs have a very limited size (8 KB per core). Note also that accesses to the MPBs are not RDMA operations since message copying is performed by the core issuing the operation.

1.2 Contributions

We are investigating the implementation of an efficient broadcast algorithm for a message-passing many-core chip, such as the Intel SCC. The broadcast operation allows one process to send a message to all processes in the application. As specified by MPI, the collective operation is executed by having all processes in the application call the communication function with matching arguments: the sender calls the *broadcast* function with the message to broadcast, while the receiver processes call it to specify the reception buffer.

To take advantage of on-chip RMA, we propose OC-Bcast (*On-Chip Broadcast*), a pipelined k-ary tree algorithm based on one-sided communication: k processes get the message in parallel from their parent to obtain a high degree of parallelism. The degree of the tree is chosen to avoid contention on the MPBs. To provide efficient synchronization between a process and its children in the tree, we introduce an additional binary notification tree. Double buffering is used to further improve the throughput.

We evaluate OC-Bcast analytically using a LogP-based performance model [9]. The evaluation shows that our algorithm based on one-sided communication outperforms existing broadcast algorithms based on two-sided communication. The main reason is that OC-Bcast reduces the amount of data moved between the off-chip memory and the MPBs on the critical path.

Finally, we confirm the analytical results through experiments. The comparison of OC-Bcast with the RCCE_comm binomial tree and *scatter-allgather* algorithms based on two-sided communication shows that: (i) our algorithm has at least 27% lower latency than the binomial tree algorithm; (ii) it has almost 3 times higher peak throughput than the *scatter-allgather* algorithm. These results clearly show that collective operations for message-passing many-core chips should be based on one-sided communication in order to fully exploit the hardware resources.

The paper is structured as follows. In Section 2 we describe the architecture and the communication features of the Intel SCC. Section 3 presents our inter-core communication model. Section 4 is devoted to our RMA-based broadcast algorithm. Analytical and experimental evaluations are presented in Sections 5 and 6 respectively. Finally, Section 7 concludes the paper.

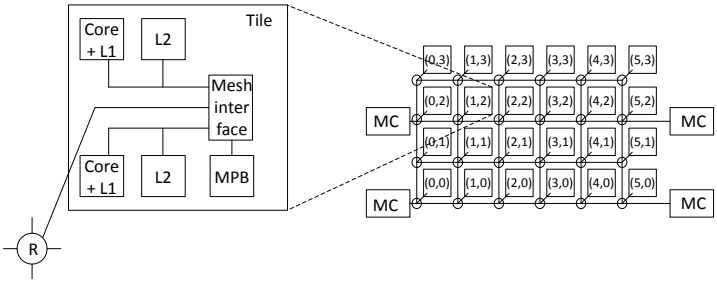

Figure 1: SCC Architecture

2 The Intel SCC

The SCC is a general-purpose many-core prototype developed by Intel Labs. In this section we describe the SCC architecture and inter-core communication.

2.1 Architecture

The cores and the NoC of the SCC are depicted in Figure 1. There are 48 Pentium P54C cores, grouped into 24 tiles (2 cores per tile) and connected through a 2D mesh NoC. Tiles are numbered from (0,0) to (5,3). Each tile is connected to a router. The NoC uses high-throughput, low-latency links and deterministic virtual cut-through X-Y routing [15]. Memory components are divided into (i) message passing buffers (MPB), (ii) L1 and L2 caches, as well as (iii) off-chip private memories. Each tile has a small (16KB) on-chip MPB equally divided between the two cores. The MPBs allow on-chip inter-core communication using RMA: each core is able to read and write in the MPB of all other cores. There is no hardware cache coherence for the L1 and L2 caches. By default, each core has access to a private off-chip memory through one of the four memory controllers, denoted by MC in Figure 1. The off-chip memory is physically shared, so it is possible to provide portions of shared memory by changing the default configuration.

2.2 Inter-core communication

To leverage on-chip RMA, cores can transfer data using the one-sided *put* and *get* primitives provided by the RCCE library [29]. Using *put*, a core (a) reads a certain amount of data from its own MPB or its private off-chip memory and (b) writes it to some MPB. Using *get*, a core (a) reads a certain amount of data from some MPB and (b) writes it to its own MPB or its private off-chip memory. The unit of data transmission is the cache line, equal to 32 bytes. If the data is larger than one cache line, it is sequentially transferred in cache-line-sized packets. During a remote read/write operation, each packet traverses all routers on the way from the source to the destination. The local MPB is accessed directly or through the local router[2].

3 Modeling *put* and *get* Primitives

In this section we propose a model for RMA *put* and *get* primitives. Our model is based on the LogP model [9] and the Intel SCC specifications [14]. We experimentally validate our model and assess its domain of validity.

[2]Direct access to the local MPB is discouraged because of a bug in the SCC hardware.

$$L_w^{mpb}(d) = o^{mpb} + d \cdot L^{hop} \quad (1)$$
$$C_w^{mpb}(d) = o^{mpb} + 2d \cdot L^{hop} \quad (2)$$
$$L_r^{mpb}(d) = C_r^{mpb}(d) = o^{mpb} + 2d \cdot L^{hop} \quad (3)$$
$$L_w^{mem}(d) = o_w^{mem} + d \cdot L^{hop} \quad (4)$$
$$C_w^{mem}(d) = o_w^{mem} + 2d \cdot L^{hop} \quad (5)$$
$$L_r^{mem}(d) = C_r^{mem}(d) = o_r^{mem} + 2d \cdot L^{hop} \quad (6)$$

$$C_{put}^{mpb}(m, d^{dst}) = o_{put}^{mpb} + m \cdot C_r^{mpb}(1) + m \cdot C_w^{mpb}(d^{dst}) \quad (7)$$
$$C_{put}^{mem}(m, d^{src}, d^{dst}) = o_{put}^{mem} + m \cdot C_r^{mem}(d^{src}) + m \cdot C_w^{mpb}(d^{dst}) \quad (8)$$
$$L_{put}^{mpb}(m, d^{dst}) = o_{put}^{mpb} + m \cdot C_r^{mpb}(1) + (m-1) \cdot C_w^{mpb}(d^{dst}) + L_w^{mpb}(d^{dst}) \quad (9)$$
$$L_{put}^{mem}(m, d^{src}, d^{dst}) = o_{put}^{mem} + m \cdot C_r^{mem}(d^{src}) + (m-1) \cdot C_w^{mpb}(d^{dst}) + L_w^{mpb}(d^{dst}) \quad (10)$$
$$L_{get}^{mpb}(m, d^{src}) = C_{get}^{mpb}(m, d^{src}) = o_{get}^{mpb} + m \cdot C_r^{mpb}(d^{src}) + m \cdot C_w^{mpb}(1) \quad (11)$$
$$L_{get}^{mem}(m, d^{src}, d^{dst}) = C_{get}^{mem}(m, d^{src}, d^{dst}) = o_{get}^{mem} + m \cdot C_r^{mpb}(d^{src}) + m \cdot C_w^{mem}(d^{dst}) \quad (12)$$

Figure 2: Communication Model

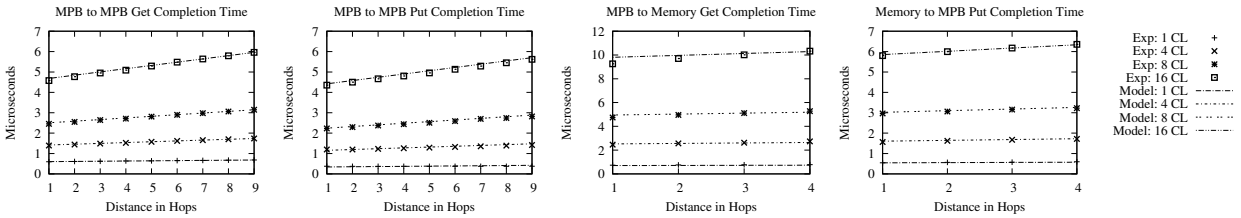

Figure 3: *get* and *put* performance (CL = Cache Line)

3.1 The model

The LogP model [9] characterizes a message passing parallel system using the number of processors (P), the time interval or *gap* between consecutive message transmissions (g), the maximum communication latency of a single-word-sized message (L), and the overhead of sending or receiving a message (o). This basic model assumes small messages. To deal with messages of arbitrary size, it can be extended to express L, o and g as a function of the message size [10].

We adapt the LogP model to the SCC communication characteristics. The LogP model assumes that the latency is the same between all processes. However, the SCC mesh communication latency is a function of the number of routers traversed on the path from the source to the destination. In our model, the number of routers traversed by one packet is defined by the parameter d. Communication on the SCC mesh is done at the packet granularity. A packet can carry one cache line (32 Bytes). We use the number of cache lines (CL) as unit for message size. Note that the SCC cores, network and memory controllers are not required to work at the same frequency. For that reason, time is chosen as the common unit for all model parameters.

For each operation, we model (i) the completion time, i.e., the time for the operation to return, and (ii) the latency, i.e., the time for the message to be available at the destination. We start by modeling read/write on the MPBs and on the off-chip private memory. Then we model put/get operations based on read/write. The read operation, executed by some core c, brings one cache line from an MPB, or from the off-chip

private memory of core c, to its internal registers[3]. The write operation, executed by some core c, copies one cache line from some internal registers of core c to an MPB, or the off-chip private memory of core c. The formulas representing our model are given in Figure 2.

3.1.1 MPB read/write

Any read or write operation of a single cache line includes some core overhead o^{mpb}, as well as some mesh overhead which depends on d (the distance between the core and the MPB). We define L^{hop} as the time needed for one packet to traverse one router; it is independent of the packet size. Therefore, the latency of writing one cache line to an MPB is given by Formula 1 in Figure 2. The write completes when the acknowledgment from the MPB is received, which adds $d \cdot L^{hop}$ (Formula 2).

To read one cache line from an MPB, a request has to be sent to this MPB; the cache line is received as a response. Therefore the latency and the completion time are equal (Formula 3).

3.1.2 Off-chip read/write

By o_r^{mem} and o_w^{mem}, we represent the constant overhead of reading and writing one cache line from/to the off-chip memory. Note that in the LogP model, an overhead o is supposed to represent the time during which the processor is involved in the communication. We choose to include memory read and write overheads in o_r^{mem} and o_w^{mem} for the sake of simplicity. The latency and the completion time of off-chip memory read/write correpond to Formulas 4-6, where d represents the distance between the core that executes the read/write operation and the memory controller.

3.1.3 Operation *put*

To model *put* (and later *get*) from MPB to MPB, we introduce o_{put}^{mpb} (respt. o_{get}^{mpb}) to define the core overhead of the *put* (respt. *get*) function apart from the time spent moving data. The corresponding o_{put}^{mem} and o_{get}^{mem} are used for operations involving private off-chip memory. A *put* operation executed by core c reads data from some source and writes it to some destination: the source is either c's local MPB (Formula 7) or private off-chip memory (Formula 8), and the destination is an MPB. We denote by d^{src} the distance between the data and core c executing the operation, and by d^{dst} the distance between c and the MPB to which the data is written. If c moves data from its local MPB then $d^{src} = 1$. Otherwise, d^{src} is the distance between c and the memory controller. Note also that the P54C cores can only execute one memory transaction at a time: moving a message of m cache lines takes m times the time needed to move one cache line[4]. The latency is a bit lower, since it does not include the acknowledgment of the last cache line written to the remote MPB (Formulas 9 and 10).

3.1.4 Operation *get*

A *get* operation executed by core c reads data from some source and writes it to some destination: the source is an MPB, and destination is c's local MPB (Formula 11) or private off-chip memory (Formula 12). We denote by d^{src} the distance between the data and core c executing the operation, and by d^{dst} the distance between c and the MPB to which the data is written. If c moves data to its local MPB, then $d^{dst} = 1$.

[3]The read operation, as defined here, should not be interpreted as a single instruction. Indeed, it is implemented as a sequence of instructions, which read an aligned cached line word by word. The first instruction causes a cache miss, and the corresponding cache line is moved to the L1 cache of the calling core. The subsequent instructions hit the L1 cache. Analogous holds for write operations, except that L1 prefetching is implemented in software.

[4]For this reason, we do not need to use the parameter g of the LogP model.

parameter	L^{hop}	o^{mpb}	o_w^{mem}	o_r^{mem}	o_{put}^{mpb}	o_{get}^{mpb}	o_{put}^{mem}	o_{get}^{mem}
value	0.005 μs	0.126 μs	0.461 μs	0.208 μs	0.069 μs	0.33 μs	0.19 μs	0.095 μs

Table 1: Parameters of our model

Otherwise, d^{dst} is the distance between c and the memory controller. In the case of a get operation, latency and completion time are equal.

3.2 Model validation

We perform a set of experiments to determine the value of the parameters we introduced and to validate our model. Experimental settings are detailed in Section 6. Figure 3 presents with dots the completion time of *put* and *get* operations from MPB to MPB or to private memory as a function of the distance for different message sizes. The parameter values obtained are presented in Table 1. The performance obtained from the model is represented by lines in Figure 3. It shows that our model precisely estimates the communication performance. Note that, for a given message size, the performance difference between the 1-hop distance (which means accessing the MPB of the other core on the same tile) and the 9-hop distance (maximum distance) is only 30%.

3.3 Contention issues

The proposed model assumes a contention-free execution. Bearing that in mind, we study contention on the SCC, to assess the validity domain of the model. We identify two possible sources of contention related to RMA communication: the NoC mesh and the MPBs. Generally speaking, concurrent accesses to the off-chip private memory could be another source of contention. However, in the configuration without shared memory, assumed throughout this paper, each core has one memory rank for itself and there is no measurable performance degradation even when the 48 cores are accessing their private portion of the off-chip memory at the same time [30].

To understand if the mesh could be subject to contention, we have run an experiment that highly loads one link. We selected the link between tile $(2, 2)$ and tile $(3, 2)$. To put a maximum stress on this link, all cores except the ones located on these two tiles are repeatedly getting 128 cache lines from one core in the third row of the mesh, but on the opposite side of the mesh compared to their own location. For instance, a core located on tile $(5, 1)$ gets data from tile $(0, 2)$. Because of X-Y routing, all data packets go through the link between tile $(2, 2)$ and tile $(3, 2)$. The measurement of a MPB-to-MPB *get* latency between tile $(2, 2)$ and tile $(3, 2)$ with the heavily loaded link did not show any performance drop, compared to the load-free *get* performance. This shows that, at the current scale, the network cannot be a source of contention.

Contention could also arise from multiple cores concurrently accessing the same MPB. To evaluate this, we have run a test where cores are getting data from the MPB of core 0 (on tile $(0, 0)$), and another test where cores are putting data into the MPB of core 0. For these tests, we select two representative scenarios of the access patterns in our broadcast algorithm presented in Section 4: parallel *gets* of 128 cache lines and parallel *puts* of 1 cache line. Note that having parallel *puts* of a large number of cache lines is not a realistic scenario since it would result in several cores writing to the same location. Figure 3.3 shows the impact on latency when increasing the number of cores executing *get* in parallel. Figure 3.3 shows the same results for parallel *put* operations. The x axis represents the number of cores executing *get* or *put* at the same time. The results are the average values over millions of iterations. In addition to the average latency, the performance of each core is displayed to better highlight the impact of contention (small circles in Figure 4). When all 48 cores are executing *get* or *put* in parallel, contention can be clearly noticed. In this case, the slowest

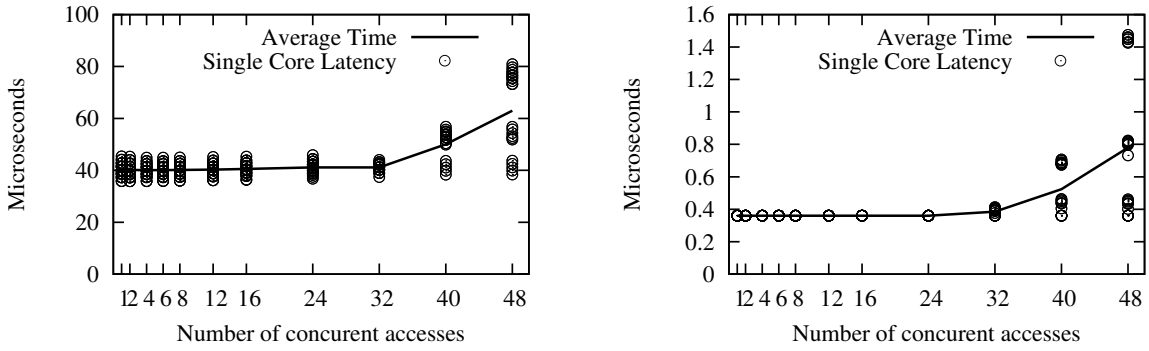

Figure 4: MPB contention evaluation

core is more than two times slower than the fastest one for *get*, and more than four times slower for a *put* operation. Moreover we observed non-deterministic overhead after the contention threshold, by running the same experiment on other cores than core 0. It can be noticed that contention does not equally affect all cores, which makes it hard to model.

These experiments indicate that MPB contention has to be taken into account in the design of algorithms for collective operations. They show that up to 24 cores accessing the same MPB do not create any measurable contention. Next we present a broadcast algorithm that takes advantage of this property.

4 RMA-based Broadcast

This section describes the main principles of OC-Bcast, our algorithm for on-chip broadcast. The full description of the algorithm, including the pseudocode, is provided in Appendix A.

4.1 Principle of the broadcast algorithm

To simplify the presentation, we assume first that messages to be broadcast fit in the MPB. This assumption is later removed. The core idea of the algorithm is to take advantage of the parallelism that can be provided by the RMA operations. When a core c wants to send message msg to a set of cores $cSet$, it *puts* msg in its local MPB, so that all the cores in $cSet$ can *get* the data from there. If all *gets* are issued in parallel, this can dramatically reduce the latency of the operation compared to a solution where, for instance, the sender c would *put* msg sequentially in the MPB of each core in $cSet$. However, having all cores in $cSet$ executing *get* in parallel may lead to contention, as observed in Section 3.3. To avoid contention, we limit the number of parallel *get* operations to some number k, and base our broadcast algorithm on a k-ary tree; the core broadcasting a message is the root of this tree. In the tree, each core is in charge of providing the data to its k children: the k children *get* the data in parallel from the MPB of their parent.

Note that the k children need to be notified that a message is available in their parent's MPB. This is done using a flag in the MPB of each of the k children. The flag, called *notifyFlag*, is set by the parent using *put* once the message is available in the parent's MPB. Setting a flag involves writing a very small amount of data to remote MPBs, but nevertheless sequential notification could impair performance especially if k is large. Thus, instead of having a parent setting the flag of its k children sequentially, we introduce a binary tree for notification to increase the parallelism. This choice is not arbitrary: It can be shown analytically that a binary tree provides the lowest notification latency, when compared to trees of higher output degrees. Figure 5 illustrates the k-ary tree used for message propagation, and the binary trees used for notification.

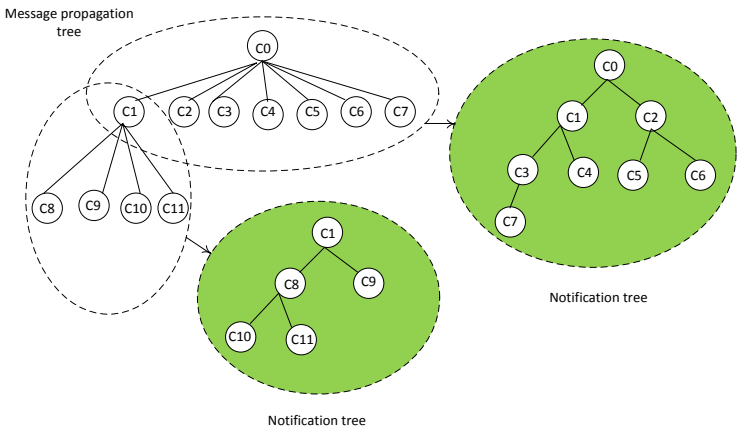

Figure 5: k-ary message propagation tree ($k = 7$) and binary notification trees.

C_0 is the root of the message propagation tree; the subtree with root C_1 is shown. Node C_0 notifies its children using the binary notification tree shown at the right of Figure 5. Node C_1 notifies its children using the binary notification tree, as depicted at the bottom of Figure 5.

Apart from the *notifyFlag* used to inform the children about message availability in their parent's MPB, another flag is needed to notify the parent that the children have got the message (in order to free the MPB). For this we use k flags in the parent MPB, called $doneFlag$, each set by one of the k children.

To summarize, considering the general case of an intermediate core, *i.e.*, the core that is neither the root nor a leaf, a core is performing the following steps. Once it has been notified that a new chunk is available in the MPB of its parent C_s: (i) it notifies its children, if any, in the notification tree of C_s; (ii) it gets the chunk in its own MPB; (iii) it sets its $doneFlag$ in the MPB of C_s; (iv) it notifies its children in its own notification tree, if any; (v) it gets the chunk from its MPB to its off-chip private memory.

Finding an efficient k-ary tree taking into account the topology of the NoC is a complex problem [4] and it is orthogonal to the design of OC-Bcast. It is outside the scope of this paper since our goal is to show the advantage of using RMA to implement broadcast. In the rest of this paper, we assume that the tree is built using a simple algorithm based on the core ids: Assuming that s is the id of the root and P the total number of processes, the children of core i are the cores with ids ranging from $(s + ik + 1) \, mod \, P$ to $(s + (i + 1)k) \, mod \, P$. Figure 5 shows the tree obtained for $s = 0$, $P = 12$ and $k = 7$.

4.2 Handling large messages

Broadcasting a message larger than an MPB can easily be handled by decomposing the large message in chunks of MPB size, and broadcasting these chunks one after the other. This can be done using pipelining along the propagation tree, from the root to the leaves.

We can further improve the efficiency of the algorithm (throughput and latency) by using a double-buffering technique, similar to the one used for point-to-point communication in the iRCCE library [8]. Up to now, we have considered messages split into chunks of MPB size,[5] which allows an MPB buffer to store only one message chunk. With double-buffering, messages are split into chunks of half the MPB size, which allows an MPB buffer to store two message chunks. The benefit of double-buffering is easy to understand. Consider message msg split into chunks ck_1 to ck_n being copied from the MPB buffer of core c to the MPB buffer of core c'. Without double buffering, core c copies ck_i to its MPB in a step r; core c' gets ck_i in step $r+1$; core c copies to its MPB ck_{i+1} in step $r+2$; etc. If each of these steps takes δ time units, the total time

[5] Of course, some MPB space needs to be allocated to the flags.

to transfer the message is roughly $2n\delta$. With double buffering, the message chunks are two times smaller and so, message msg is split into chunks ck_1 to ck_{2n}. In a step r, core c can copy ck_{i+1} to the MPB while core c' gets ck_i. If each of these steps takes $\delta/2$ time units, the total time is roughly only $n\delta$.

5 Analytical Evaluation

We analytically compare OC-Bcast with two state-of-the-art algorithms based on two-sided communication: binomial tree and *scatter-allgather*. We consider their implementations from the RCCE_comm library [7]. RCKMPI [28] uses the same algorithms, but still keeps their original MPICH2 implementation, not optimized for the SCC. Also, our experiments have confirmed that RCCE_comm currently performs better than RCKMPI. Thus, we have chosen to conduct the analysis using RCCE_comm, as the fastest available implementation of collectives on the SCC, to the best of our knowledge.

To highlight the most important properties, we divide the analysis into two parts: latency of small messages (OC-Bcast vs. binomial tree) and throughput for large messages (OC-Bcast vs. *scatter-allgather*). The analysis is based on the model introduced in Section 3. For a better understanding of the presented results, first we give some necessary implementation details.

5.1 Implementation details

Both OC-Bcast and the *RCCE_comm* library use flags allocated in the MPBs to implement synchronization between the cores. SCC guarantees read/write atomicity on 32B cache lines. So, allocating one cache line per flag is enough to ensure atomicity (no additional mechanism such as locks is needed). In the modeling of the algorithms we assume that no time elapses between setting the flag (by one core) and checking that the flag is set (by the other core). OC-Bcast requires $k + 1$ flags per core. The rest of the MPB can be used for the message payload. For this, OC-Bcast uses two buffers of $M_{oc} = 96$ cache lines each. RCCE_comm, which is based on RCCE, uses a payload buffer of $M_{rcce} = 251$ cache lines. Since topology issues are not discussed in the paper, we simply consider an average distance $d^{mpb} = 1$ for accessing remote MPBs, and an average distance $d^{mem} = 1$ for accessing the off-chip memory.

5.2 Latency of short messages

We define the latency of the broadcast primitive as the time elapsed between the call of the broadcast function by the source, and the time at which the message is available at all cores (including the source), i.e., when the last core returns from the function. The analytically computed latency for small messages on the SCC is shown in Figure 6. For OC-Bcast, different values of k are given ($k = 2$, $k = 7$, $k = 47$). A more detailed discussion on the choice of k can be found in Appendix C. Note that OC-Bcast with $k = 7$ provides the best trade-off between latency and throughput according to our analysis. Although the characteristics of the SCC allow us to increase k up to 24 without experiencing measurable contention (as discussed in Section 3), the same tree depth is reached already with $k = 7$. As we can see, OC-Bcast significantly outperforms the binomial tree algorithm. The difference increases as the message size increases.

The improvement of OC-Bcast over the binomial tree algorithm is a direct consequence of using RMA. To clarify this, we now derive the formulas used to obtain the data in Figure 6. For the sake of simplicity, we ignore notification costs here and concentrate only on the critical path of data movement in the algorithms. Figure 7 summarizes the simplified formulas, whereas the complete formulas, which take the notification cost into account, are given in Appendix B.

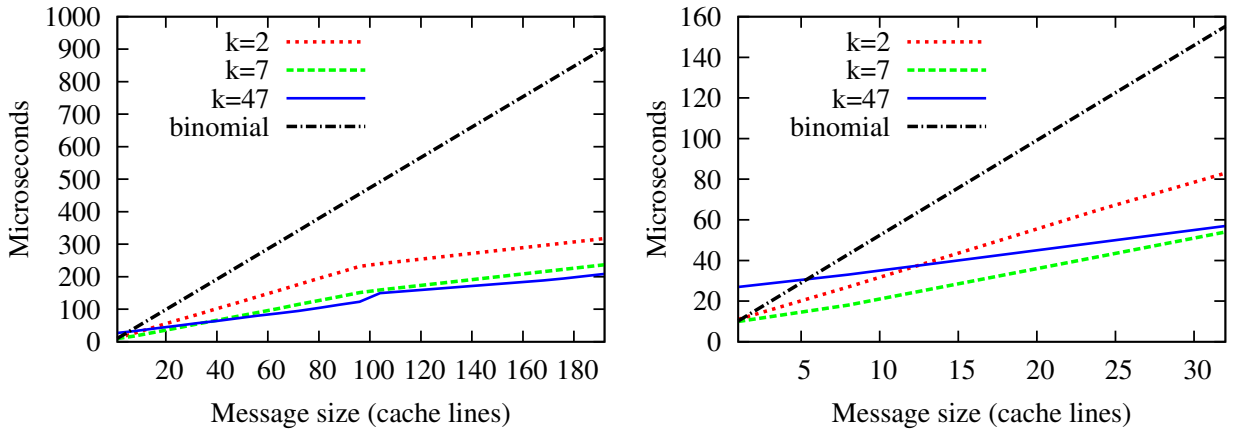

Figure 6: Broadcast algorithms: analytical latency comparison. Legend: k=x, OC-Bcast with the corresponding k; binomial, RCCE_comm binomial.

5.2.1 Latency of OC-Bcast

For OC-Bcast, the critical path of data movement is expressed as follows. Consider a message of size $m \leq M_{oc}$ to be broadcast by some core c. Core c first puts the message in its MPB, which takes $C_{put}^{mem}(m)$ time to complete. Then, depending on k, there might be multiple intermediate nodes before the message reaches the leaves. For P cores and a k-ary tree, there are $O(log_k P)$ levels of intermediate nodes. At each intermediate level, the cores copy the message from their parent's MPB to their own MPB in parallel, which takes $C_{get}^{mpb}(m)$ time to complete. Note that after copying, each node has to get the message to its private memory, but this operation is not on the critical path. Finally, the leaves copy the message, first to their MPB ($C_{get}^{mpb}(m)$) and then to the off-chip private memory ($C_{get}^{mem}(m)$). Therefore, the total latency is given by Formula 13 in Figure 7.

5.2.2 Latency of the two-sided binomial tree

The binomial tree broadcast algorithm is based on a binary recursive tree. The set of nodes is divided into two subsets of $\lfloor \frac{P}{2} \rfloor$ and $\lceil \frac{P}{2} \rceil$ nodes. The root, belonging to one of the subsets, sends the message to one node from the other subset. Then, broadcast is recursively called on both subsets. Obviously, the formed tree has $O(log_2 P)$ levels and in each of them the whole message is sent between the pairs of nodes. A send/receive operation pair involves a put by the sender and a get by the receiver, so the total latency of the algorithm is $O(log_2 P) \cdot (C_{put}^{mem}(m) + C_{get}^{mem}(m))$. However, note that after receiving the broadcast message, a node keeps sending it to other nodes in every subsequent iteration. Therefore, if the message is small, we can assume that it will be available in the core's L1 cache, which reduces the cost of the put operation. We approximate reading from the L1 cache with zero cost. With this, we get Formula 14.

5.2.3 Latency comparison

Now we can directly compare the analytical expressions for the two broadcast algorithms. In Formula 13, which represents the latency of OC-Bcast, there are only two off-chip memory operations ($C_{r/w}^{mem}$) on the critical path for one chunk, regardless of the number of cores P. This is not the case for the binomial algorithm, represented by Formula 14. Moreover, as k increases, the number of MPB-to-MPB copy operations reduces for OC-Bcast.

$$
\begin{aligned}
L^{critical}_{OC-Bcast}(P,m,k) &= C^{mem}_{put}(m) + O(log_k P)\cdot C^{mpb}_{get}(m) + C^{mem}_{get}(m) \\
&= m\cdot\left(O(log_k P)\cdot\left(C^{mpb}_r + C^{mpb}_w\right) + C^{mem}_r + C^{mem}_w\right) && (13)\\
L^{critical}_{binomial}(P,m) &= O(log_2 P)\cdot(m\cdot C^{mpb}_w + C^{mem}_{get}(m)) \\
&= m\cdot\left(O(log_2 P)\cdot\left(C^{mpb}_r + C^{mpb}_w + C^{mem}_w\right) + C^{mem}_r\right) && (14)\\
B_{OC-Bcast} &= \frac{M_{oc}}{C^{mpb}_{get}(M_{oc}) + C^{mem}_{get}(M_{oc})} = \frac{1}{2C^{mpb}_r + C^{mpb}_w + C^{mem}_w} && (15)\\
B_{scatter\_allgather} &= \frac{P\cdot M_{oc}}{P\cdot(C^{mem}_{put}(M_{oc}) + C^{mem}_{get}(M_{oc})) + (2P-3)(M_{oc}\cdot C^{mpb}_w + C^{mem}_{get}(M_{oc}))} \\
&\approx \frac{1}{3C^{mpb}_r + 3C^{mpb}_w + C^{mem}_r + 3C^{mem}_w} && (16)
\end{aligned}
$$

Figure 7: Latency and Throughput Model for Broadcast Operations

The gain of OC-Bcast increases further when increasing the message size because of double buffering and pipelining. It can be observed in Figure 5.2 that the slope changes for messages larger than $M_{OC-Bcast}$ (96 cache lines). In Figure 5.2, we can notice that OC-Bcast-47 is the slowest for very small message in spite of having only two levels in the data propagation tree (the root and its 47 children). The reason is that a large value of k increases the cost of polling. For $k = 47$, the root has 47 flags to poll before it can free its MPB.

5.3 Throughput for large messages

Now we consider messages large enough to fill the propagation tree pipeline used by OC-Bcast. For such messages, every core executes a loop, where one chunk is processed in each iteration. We compare OC-Bcast with the RCCE_comm *scatter-allgather* algorithm.

Table 2 gives the throughput based on the analytical model. The same values of k are considered for OC-Bcast as in the latency analysis. Regardless of the choice of k, the throughput is almost three times better than the one provided by two-sided *scatter-allgather*. To understand this gain, we again compute the critical path of the message payload. As in the latency analysis, we derive simplified formulas (Figure 7), and provide the complete formulas in Appendix B. To simplify the modeling, we assume a message of size $P\cdot M_{oc}$. With OC-Bcast, such a message is transferred in P chunks of size M_{oc}. *Scatter-allgather* transfers the same message by dividing it into P slices of size M_{oc}.

5.3.1 Throughput of OC-Bcast

To express the critical path of data movement of OC-Bcast, we need to distinguish between the root and the other nodes (intermediate nodes and leaves). The root repeatedly moves new chunks from its private off-chip memory to its MPB, which takes $C^{mem}_{put}(M_{oc})$ for each chunk. The other nodes repeat two operations: First, they copy a chunk from the parent's MPB to their own MPB, and then copy the same chunk from the MPB to their private memory, which gives the completion time of $C^{mpb}_{get}(M_{oc}) + C^{mem}_{get}(M_{oc})$. The throughput is determined by the throughput of the slowest node. For the parameter values valid on the SCC, the root is always faster than the other nodes, so the throughput of OC-Bcast (in cache lines per second) is expressed

Algorithm	OC-Bcast, k=2	OC-Bcast, k=7	OC-Bcast, k=47	scatter-allgather
Throughput (MB/s)	35.22	34.30	35.88	13.38

Table 2: Broadcast algorithms: analytical comparison of throughput

by Formula 15. Note that the peak throughput is not a function of k. This is because we assume that the message is large enough to fill the whole pipeline.

5.3.2 Throughput of two-sided scatter-allgether

Scatter-allgather has two phases. During the *scatter* phase, the message is divided into P equal slices of size M_{oc} (recall that the message size is fixed to $P \cdot M_{oc}$). Each core then receives one slice of the original message. The second phase of the algorithm is *allgather*, during which a node should obtain the remaining $P-1$ slices of the message. To implement *allgather*, the Bruck algorithm [6] is used: At each step, core i sends to core $i-1$ the slices it received in the previous step.

Now we consider the completion time of the two phases of the *scatter-allgather* algorithm. The *scatter* phase is done using a binary recursive tree, similar to the one used by the binomial algorithm. The difference is that in this case we transfer only a part of the message in each step. In the end, the root has to send out each of the P slices but its own, so the critical path of this step consists of $P-1$ send/receive operations, which gives the completion time of $(P-1)(C_{put}^{mem}(M_{oc}) + C_{get}^{mem}(M_{oc}))$. The allgather phase consists of $P-1$ exchange rounds. In each round, core i sends one slice to core $i-1$ and receives one slice from core $i+1$. Thus, there are two send/receive operations between pairs of processes in each round, so this phase takes $2(P-1)(C_{put}^{mem}(M_{oc}) + C_{get}^{mem}(M_{oc}))$ to complete. As with the binomial tree, taking the existance of the caches into account gives a more accurate model. Note, however, that this holds only for the allgather phase. Finally, the completion times of the two phases are added up. There is no pipelining in this algorithm, so the throughput can be easily expressed as a reciprocial value of the computed completion time on the root. Formula 16 presents the modeled throughput of the two-sided *scatter-allgather* algorithm (in cache lines per second).

5.3.3 Throughput comparison

The additional terms in Formula 16 compared to Formula 15 explain the performance difference in Table 2, and show the advantage of designing a broadcast protocol based on one-sided operations: The number of write accesses to the MPBs and to the off-chip memory (C_w^{mpb} and C_w^{mem}) with OC-Bcast is three times lower than that of the *scatter-allgather* algorithm based on two-sided communication. The number of read accesses is also reduced.

5.4 Discussion

The presented analysis shows that our broadcast implementation based on one-sided operations brings considerable performance benefits, in terms of both latency and throughput. Note, however, that OC-Bcast is not the only possible design of RMA-based broadcast. Our goal in this paper is not to find the most efficient algorithm and prove its optimality, but to highlight the potential for exploiting parallelism using RMA-based approach. Indeed, a good example of another possible broadcast implementation is adapting the two-sided *scatter-allgather* algorithm to use the one-sided primitives available on the SCC.

Furthermore, some simple, yet effective optimizations can be applied to OC-Bcast to make it even faster. For instance, a leaf in a broadcast tree does not need to copy the data to its MPB, but directly to the off-chip private memory. Similarly, we could take advantage of the fact that there are two cores accessing the

same physical MPB, to have less data copying. However, we have chosen not to include these optimizations because they would result in having to deal with many special cases, which would likely obfuscate the main point of the presented work.

6 Experimental Evaluation

In this section we evaluate the performance of OC-Bcast on Intel SCC and compare it with both the binomial and the *scatter-allgather* broadcast of RCCE_comm [7].

6.1 Setup

The experiments have been done using the default settings for the SCC: 533 MHz tile frequency, 800 MHz mesh and DRAM frequency and the standard LUT entries. We use the sccKit version 1.4.1.3, running a custom version of sccLinux, based on Linux 2.6.32.24-generic. As already mentioned in the previous section, we fix the chunk size used by OC-Bcast to 96 cache lines, which leaves enough space for flags (for any choice of k). The presented experiments use core 0 as the source. Selecting another core as the source gives similar results. A message is broadcast from the private memory of core 0 to the private memory of all other cores. The results are the average values over 10'000 broadcasts, discarding the first 1'000 results. For time measurement, we use global counters accessible by all cores on the SCC, which means that the timestamps obtained by different cores are directly comparable. The latency is defined as in Section 5. To avoid cache effects in repeated broadcasts, we preallocate a large array and in every broadcast we operate on a different (currently uncached) offset inside the array.

6.2 Evaluation results

We have tested the algorithms with message sizes ranging from 1 cache line (32 bytes) to 32'768 cache lines (1 MiB). As in Section 5, we first focus on the latency of short messages, and then analyze the throughput of large messages. Regarding the binomial tree and *scatter-allgather* algorithms, our experiments have confirmed that the former performs better with small messages, whereas the latter is a better fit for large messages. Therefore, we compare OC-Bcast only with the better one for a given message size.

6.2.1 Latency of small messages

Figure 6.2.1 shows the latency of messages of size $m \leq 2M_{oc}$. Comparing the results with Figure 5.2 shows the accuracy of our analytical evaluation, and confirms the performance increase. Even for messages of one cache line, OC-Bcast with $k = 7$ provides 27% improvement compared to the binomial tree ($16.6\mu s$ vs. $21.6\mu s$). As expected, the difference grows with the message size, since a larger message implies more off-chip memory accesses in the RCCE_comm algorithms, but not in OC-Bcast. It can also be noticed that large values of k help improving the latency in OC-Bcast by reducing the depth of the tree. For message size between 96 and 192 cache lines, the latency of OC-Bcast with $k = 7$ is around 25% better than with $k = 2$.

Another result worth mentioning is the relation between the curves representing $k = 7$ and $k = 47$. Namely, we can see that they almost completely overlap in Figure 6.2.1, whereas there is a more significant difference indicated by the analytical evaluation (Figure 5.2). This can be attributed to MPB contention – recall that too many parallel accesses to the same MPB can impair the performance, as pointed out in Section 3.

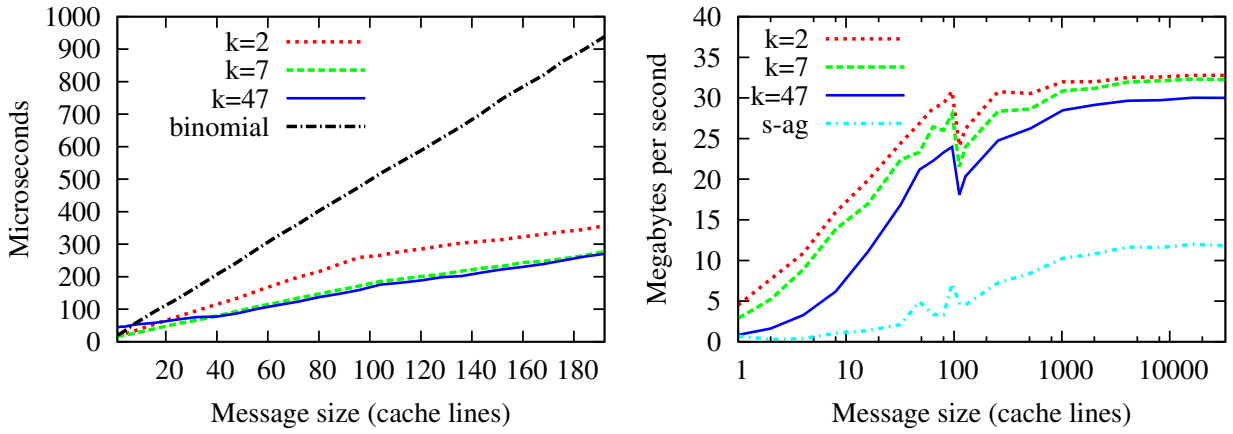

Figure 8: Experimental comparison of broadcast algorithms. Legend: k=x, OC-Bcast with the corresponding value of k; binomial, RCCE_comm binomial; s-ag, RCCE_comm scatter-allgather.

6.2.2 Throughput for large messages

The results of the throughput evaluation are given in Figure 6.2.1 (note that the x-axis is logarithmic). The peak performance is very close to the results presented in Table 2: OC-Bcast gives an almost threefold throughput increase compared to the two-sided *scatter-allgather* algorithm. The OC-Bcast performance drop for a message of 97 cache lines is due to the chunk size. Recall that the size of a chunk in OC-Bcast is 96 cache lines. A message of 97 cache lines is divided into a 96 cache lines chunk and 1 cache line chunk. The second chunk is then limiting the throughput. For large messages, this effect becomes negligible since there is always at most one non-full chunk.

It can be noticed that the only significant difference with respect to the analytical predictions is for OC-Bcast with $k = 47$ (the throughput is about 16% lower than predicted). Once again, MPB contention is one of the sources of the observed performance degradation. This confirms that large values of k might be inappropriate, especially at large scale, since the linear gain in parallelism could be paid by an exponential loss related to contention.

6.3 Discussion

The expected performance based on the model is slightly better than the results we obtain through the experiments. The main reason is that in the analytical evaluation, we assumed a distance of one hop for all *put* and *get* operations: This is physically not possible on the SCC no matter what tree generation strategy is used. However, note that the measured values are still very close to the computed ones.

7 Conclusion

OC-Bcast is a pipelined k-ary tree broadcast algorithm based on one-sided communication. It is designed to leverage the inherent parallelism of on-chip RMA in many-cores. Experiments on the SCC show that it outperforms the state-of-the-art broadcast algorithms on this platform. OC-Bcast provides around 3 times better peak throughput and improves latency by at least 27%. An analysis using a LogP-based model shows that this performance gain is mainly due to a limited number of off-chip data movements on the critical path of the operation: one-sided operations allow to take full advantage of the on-chip MPBs. These results

show that hardware-specific features should be taken into account to design efficient collective operations for message-passing many-core chips, such as the Intel SCC.

The work presented in this paper considers the SPMD programming model. Our ongoing work includes extending OC-Bcast to handle the MPMD programming model by leveraging parallel inter-core interrupts. Many-core operating systems [3] are an interesting use-case for such a primitive. We also plan to extend our approach to other collective operations and integrate them in an MPI library, so we can analyze the overall performance gain in parallel applications.

8 Acknowledgements

We would like to thank Martin Biely, Žarko Milošević and Nuno Santos for their useful comments. Thanks also to Ciprian Seiculescu for helping us understand the behaviour of the SCC NoC. We are also thankful for the help provided by the Intel MARC Community.

References

[1] George Almási, Philip Heidelberger, Charles J. Archer, Xavier Martorell, C. Chris Erway, José E. Moreira, B. Steinmacher-Burow, and Yili 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, 2005.

[2] InfiniBand Trade Association. *InfiniBand Architecture Specification: Release 1.0*. InfiniBand Trade Association, 2000.

[3] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 29–44, 2009.

[4] O. Beaumont, L. Marchal, and Y. Robert. Broadcast Trees for Heterogeneous Platforms. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, IPDPS '05, pages 80–92, 2005.

[5] Shekhar Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, pages 746–749, 2007.

[6] Jehoshua Bruck, Ching-Tien Ho, Eli Upfal, Shlomo Kipnis, and Derrick Weathersby. Efficient Algorithms for All-to-All Communications in Multiport Message-Passing Systems. *IEEE Transactions on Parallel and Distributed Systems*, 8:1143–1156, November 1997.

[7] E. Chan. RCCE comm: A Collective Communication Library for the Intel Single-chip Cloud Computer. http://communities.intel.com/docs/DOC-5663, 2010.

[8] C. Clauss, S. Lankes, J. Galowicz, and T. Bemmerl. iRCCE: a non-blocking communication extension to the RCCE communication library for the Intel Single-chip Cloud Computer. http://communities. intel. com/docs/DOC-6003, 2011.

[9] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '93, pages 1–12, 1993.

[10] David E. Culler, Lok Tin Liu, Richard P. Martin, and Chad O. Yoshikawa. Assessing Fast Network Interfaces. In *IEEE Micro*, pages 35–43, February 1996.

[11] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[12] Rinku Gupta, Pavan Balaji, Dhabaleswar K. Panda, and Jarek Nieplocha. Efficient Collective Operations Using Remote Memory Operations on VIA-Based Clusters. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, IPDPS '03, pages 46–62, 2003.

[13] 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*, IPDPS '07, page 232, 2007.

[14] Jason Howard, Saurabh Dighe, Yatin Hoskote, Sriram Vangal, David Finan, Gregory Ruhl, David Jenkins, Howard Wilson, Nitin Borkar, Gerhard Schrom, and et al. A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS. In *2010 IEEE International SolidState Circuits Conference*, pages 108–109. IEEE, 2010.

[15] P. Kermani and L. Kleinrock. Virtual cut-through: A new computer communication switching technique. *Computer Networks*, 3(4):267–286, 1979.

[16] P. Kogge et al. Exascale Computing Study: Technology Challenges in Achieving Exascale Systems. Technical report, DARPA, 2008.

[17] Jiuxing Liu, Amith R. Mamidala, and Dhabaleswar K. Panda. Fast and Scalable MPI-Level Broadcast Using InfiniBand's Hardware Multicast Supportsch. In *Proceedings of the 18th International Symposium on Parallel and Distributed Processing*, IPDPS '04, page 10, 2004.

[18] Jiuxing Liu, Jiesheng Wu, Sushmitha P. Kini, Pete Wyckoff, and Dhabaleswar K. Panda. High performance RDMA-based MPI implementation over InfiniBand. In *Proceedings of the 17th annual international conference on Supercomputing*, ICS '03, pages 295–304, 2003.

[19] T. Mattson and R. Van Der Wijngaart. RCCE: a Small Library for Many-Core Communication. http://techresearch.intel.com, 2010.

[20] Timothy G. Mattson, Rob Van der Wijngaart, and Michael Frumkin. Programming the Intel 80-core network-on-a-chip terascale processor. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 38:1–38:11, 2008.

[21] MPI Forum. MPI2: Extensions to the Message-Passing Interface. www.mpi-forum.org, 1997.

[22] Thomas Preud'homme, Julien Sopena, Gael Thomas, and Bertil Folliot. BatchQueue: Fast and Memory-Thrifty Core to Core Communication. In *Proceedings of the 2010 22nd International Symposium on Computer Architecture and High Performance Computing*, SBAC-PAD '10, pages 215–222, 2010.

[23] R. Rotta. On efficient message passing on the intel scc. In *Proceedings of the 3rd MARC Symposium*, pages 53–58, 2011.

[24] M. Shroff and R.A. Van De Geijn. CollMark: MPI collective communication benchmark. In *International Conference on Supercomputing 2000*, page 10, 1999.

[25] 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, 2005.

[26] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of Collective Communication Operations in MPICH. *IJHPCA*, 19(1):49–66, 2005.

[27] Josep Torrellas. Architectures for Extreme-Scale Computing. *Computer*, 42(11):28–35, November 2009.

[28] Isaías A. Comprés Ureña, Michael Riepen, and Michael Konow. RCKMPI - lightweight MPI implementation for intel's single-chip cloud computer (SCC). In *Proceedings of the 18th European MPI Users' Group conference on Recent advances in the message passing interface*, EuroMPI'11, pages 208–217, 2011.

[29] Rob F. van der Wijngaart, Timothy G. Mattson, and Werner Haas. Light-weight communications on Intel's single-chip cloud computer processor. *ACM SIGOPS Operating Systems Review*, 45(1):73–83, February 2011.

[30] Michiel W. van Tol, Roy Bakker, Merijn Verstraaten, Clemens Grelck, and Chris R. Jesshope. Efficient Memory Copy Operations on the 48-core Intel SCC Processor. In *Proceedings of the 3rd MARC Symposium*, pages 13–18, 2011.

A Detailed Description of OC-Bcast

This section details our algorithm. The pseudocode for a core c is presented in Algorithm 1. To broadcast a message, all cores invoke the broadcast function (line 20). The input variables are msg, a memory location in the private memory of the core, and $root$, the id of the core broadcasting the message. The broadcast function moves the content of msg on the $root$, to the private memory of all other cores.

The pseudocode assumes that the total number of processes is P and that the degree of the data propagation tree used by OC-Bcast is k. We introduce the following notation for put and get operations: *'put src* $\longrightarrow$ *dest'* and *'get dest* $\longleftarrow$ *src'*.

Each core c has a unique data parent $dataParent_c$ in the data propagation tree, and a set of children $dataChildren_c$. The set $notifyChildren_c$ includes all the cores that core c should notify during the algorithm. Note that a core c can be part of several binary trees used for notifications. In the example of Figure 5, if we consider core $c1$: $dataParent_{c1} = c0$; $dataChildren_{c1} = \{c8, c9, c10, c11\}$; $notifyChildren_{c1} = \{c3, c4, c8, c9\}$. These sets are computed at the beginning of the broadcast (line 15). MPBs are represented by the global variable MPB where $MPB[c]$ is the MPB of core c. A $notifyFlag$ and k $doneFlag$ (one per child) are allocated in each MPB to manage synchronizations between cores. The rest of the MPB space is divided into two buffers to implement double buffering.

The $broadcast\_chunk$ function is used to broadcast a chunk. Each chunk is uniquely identified using a tuple <$bcastID$,$chunkID$>. Chunk ids are used for notifications. To implement double buffering, the two buffers in the MPB are used alternatively: for the chunk <$bcastID$,$chunkID$>, the buffer '$chunkID \mod 2$' is used. By setting the $notifyFlag$ of a core c to <$bcastID$,$chunkID$>, core c is informed that the chunk <$bcastID$,$chunkID$> is available in the MPB of its $dataParent_c$. Notifications are done in two steps. First, if a core is an intermediate node in a binary notification tree, it forwards the notification in this tree as soon as it receives it (line 28): in Figure 5, core $c1$ notifies $c3$ and $c4$ when it gets the notification from core $c0$. Then, after copying the chunk to its own MPB, it can start notifying the nodes that will get the chunk from its MPB (line 32): in Figure 5, core $c1$ then notifies $c8$ and $c9$. When a core finishes getting a chunk, it informs its parent using the corresponding $doneFlag$ (line 30). A core can copy a new chunk $chunkID$ in one of its MPB buffers, when all its children in the message propagation tree got the previous chunk ($chunkID - 2$) that was in the same buffer (line 22). Note that the $bcastID$ is needed to be able to differentiate between chunks of two messages that are broadcast consecutively. The broadcast function on core c returns when c has got the last chunk in its private memory (line 34), and it knows that the data in its MPB buffers is not needed by any other core (line 19).

Algorithm 1 OC-Bcast (code for core c)

Global Variables:

1: P {total number of cores}
2: k {data tree output degree}
3: $MPB[P]$ {$MPB[i]$ is the MPB of core i}
4: $notifyFlag$ {MPB address of the flag, of the form <bcastID,chunkID>, used to notify data availability}
5: $doneFlag[k]$ {MPB address of the flags, of the form <bcastID,chunkID>, used to notify broadcast completion of a chunk}
6: $buffer[2]$ {MPB address of the two buffers used for double buffering}

Local Variables:

7: $bcastID \leftarrow 0$ {current broadcast id}
8: $chunkID$ {current chunk ID}
9: $dataParent_c$ {core from which c should get data}
10: $dataChildren_c$ {set of data children of c}
11: $notifyChildren_c$ {set of notify children of c}

12: **broadcast (**msg**,** $root$**)**
13: $bcastID \leftarrow bcastID + 1$
14: $chunkID \leftarrow 0$
15: $\{dataParent_c, dataChildren_c, notifyChildren_c\} \leftarrow$ prepareTree($root$, k, P)
16: **for all chunks at offset** i **of** msg **do**
17: $chunkID \leftarrow chunkID + 1$
18: **broadcast_chunk(**$msg[i]$**,** $root$**)**
19: **wait until** $\forall child \in dataChildren_c$: $MPB[c].doneFlag[child] = (bcastID, chunkID)$

20: **broadcast_chunk (**$chunk$**,** $root$**)**
21: **if** $chunkID > 2$ **then**
22: **wait until** $\forall child \in dataChildren_c$: $MPB[c].doneFlag[child] \geq (bcastID, chunkID - 2)$
23: **if** $c = root$ **then**
24: *put* $chunk \longrightarrow MPB[c].buffer[chunkID \bmod 2]$
25: **else**
26: **wait until** $MPB[c].notifyFlag \geq (bcastID_c, chunkID_c)$
27: **for all** $child$ **such that** $child \in notifyChildren_c \setminus dataChildren_c$ **do**
28: *put* $(bcastID, chunkID) \longrightarrow MPB[child].notifyFlag$
29: *get* $MPB[c].buffer[chunkID \bmod 2] \longleftarrow MPB[dataParent_c].buffer[chunkID \bmod 2]$
30: *put* $(bcastID, chunkID) \longrightarrow MPB[dataParent_c].doneFlag[c]$
31: **for all** $child$ **such that** $child \in notifyChildren_c \cap dataChildren_c$ **do**
32: *put* $(bcastID, chunkID) \longrightarrow MPB[child].notifyFlag$
33: **if** $c \neq root$ **then**
34: *get* $chunk \longleftarrow MPB[c].buffer[chunkID \bmod 2]$

B Modeling Broadcast Algorithms

This section provides the analytical model of the OC-Bcast algorithm, as well as the *RCCE_comm* broadcast algorithms, based on the material presented in Section 3. It uses the notation introduced in Section 5. Note that the derived formulas are approximative: Our intention is not to fully model all the presented algorithms for every corner case, but to understand their general behavior and characteristics.

B.1 OC-Bcast model

B.1.1 Latency of short messages

Here we consider messages of size $0 < m \leq 2M_{oc}$. We define the broadcast latency as the time elapsed between the call of the broadcast procedure by the root and the time when the last core (including the root) returns from the procedure.

To express the latency of OC-Bcast, we first model the time it takes for the k children of a core to be aware of the availability of a new chunk in the MPB of their parent. Since the children are notified using a binary tree, this time can be expressed as:

$$L_{notify\_children}(k) = \lceil log_2(k) \rceil \cdot (C_{put}^{mpb}(1, d^{mpb}) + L_{put}^{mpb}(1, d^{mpb}) + C_r^{mpb}(1))$$

At each step of the tree it includes the time for completing the notification of the first child ($C_{put}^{mpb}(1, d^{mpb})$), the time to write the flag of the second child ($L_{put}^{mpb}(1, d^{mpb})$), and the time for the child to read it ($C_r^{mpb}(1)$).

Similarly, to find out that all its children copied the data to their MPBs, a core polls k flags, so it must read each of them at least once:

$$L_{polling\_children}(k) = k \cdot C_r^{mpb}(1)$$

Next, we express the depth of the tree D, as a function of k and P (number of cores). In a complete k-ary tree, there are k^i nodes at the i-th level. Therefore, the depth of the tree used by OC-Bcast is the number of levels necessary to "cover" P cores:

$$1 + k + k^2 + ... + k^{d-1} = \frac{k^d - 1}{k - 1} \geq P \tag{17}$$

Now we can express D, as the minimum natural number satisfying the above inequation:

$$D = \lceil log_k(P(k-1) + 1) \rceil \tag{18}$$

Then we model the latency of the *broadcast_chunk()* function for one chunk based on Algorithm 1. As a first step, we model T, the amount of time that the root of the tree, an intermediate node, and a leaf adds to the latency of *broadcast_chunk()*, considering that the message contains at most two chunks. For the root, T_{chunk_root} includes putting the chunk in its MPB ($C_{put}^{mem}(m, d^{mem}, 1)$), and notifying its k children in the message propagation tree:

$$T_{chunk\_root}(m, k) = C_{put}^{mem}(m, d^{mem}, 1) + L_{notify\_children}(k) \tag{19}$$

For an intermediate node, it includes the time to get the chunk from its parent's MPB ($C_{get}^{mpb}(m, d^{mpb})$), to notify its parent that it has the chunk ($C_{put}^{mpb}(1, d^{mpb})$), and to notify its children.

$$T_{chunk\_intermediate}(m, k) = C_{get}^{mpb}(m, d^{mpb}) + C_{put}^{mpb}(1, d^{mpb}) + L_{notify\_children}(k) \tag{20}$$

Finally, for a leaf, it includes the time to get the chunk from its parent's MPB ($C_{get}^{mpb}(m, d^{mpb})$), to notify its parent that it has the chunk, and to copy the leaf to its private memory.

$$T_{chunk\_leaf}(m) = C_{get}^{mpb}(m, d^{mpb}) + C_{put}^{mpb}(1, d^{mpb}) + L_{get}^{mem}(m, 1, d^{mem})$$

Now we can express the latency of transferring one chunk of size $m \leq M_{oc}$ between P cores, using a k-ary tree:

$$\begin{aligned} L_{bcast\_chunk}(P, m, k) = T_{chunk\_root}(m, k) + (D-2) \cdot T_{chunk\_intermediate}(m, k) + \\ + max\big(T_{chunk\_leaf}(m), C_{get}^{mem}(m, 1, d^{mem}) + L_{polling\_children}(k)\big) \end{aligned}$$

Depending on the time needed for the last intermediate node to poll the k flags after copying a chunk to its private memory, it can finish later than the last leaf, as expressed by the last term of the formula.

Finally, the latency of OC-Bcast for a message of size $m \leq 2M_{oc}$ depends on size of the second chunk $m' = max(m - M_{oc}, 0)$:

$$\begin{aligned} L_{OC-Bcast\_short}(P, m, k) = L_{bcast\_chunk}(P, min(m, M_{oc}), k) + \\ + max\big(T_{chunk\_leaf}(m'), C_{get}^{mem}(m', 1, d^{mem}) + L_{polling\_children}(k)\big) \end{aligned} \tag{21}$$

Note that $L_{chunk\_leaf}(0) = 0$ because the function is called only if the second chunk exists. Using the above formula, we compute the latency of OC-Bcast in Figure 6, Section 5.

B.1.2 Throughput for large messages

Now we consider messages large enough to fill the pipeline used by OC-Bcast. For such messages, every node executes a loop, where one chunk is processed in each iteration. We compute the completion time of the *broadcast_chunk()* function for such an iteration considering the root, an intermediate node and a leaf.

$$\begin{aligned} C_{chunk\_root}(k) &= L_{polling\_children}(k) + C_{put}^{mem}(M_{oc}, d^{mem}, 1) + 2 \cdot C_{put}^{mpb}(1, d^{mpb}) \\ C_{chunk\_intermediate}(k) &= L_{polling\_children}(k) + C_{r}^{mpb}(1, 1) + C_{get}^{mpb}(M_{oc}, d^{mpb}) \\ &\quad + 3 \cdot C_{put}^{mpb}(1, d^{mpb}) + C_{get}^{mem}(M_{oc}, 1, d^{mem}) \\ C_{chunk\_leaf} &= C_{r}^{mpb}(1, 1) + C_{get}^{mpb}(M_{oc}, d^{mpb}) \\ &\quad + C_{put}^{mpb}(1, d^{mpb}) + C_{get}^{mem}(M_{oc}, 1, d^{mem}) \end{aligned}$$

The time to fully process a chunk is determined by the slowest node. Therefore, based on the slowest completion time, the throughput is easily expressed as:

$$B(k) = \frac{M_{oc}}{max(C_{chunk\_root}(k), C_{chunk\_intermediate}(k), C_{chunk\_leaf})} \tag{22}$$

We use the above formula to compute the throughput in Table 2, Section 5. Note that no matter what the values of the parameters are, the leaf is always faster than the intermediate node. However, we keep C_{chunk_leaf} in the above formula, because there is a special case with no intermediate nodes, for the maximum value of k (47 for the SCC).

B.2 RCCE two-sided communication model

Before modeling the *RCCE_comm* broadcast algorithms, we need to model the two-sided $send/receive$ primitives from the RCCE library.

The RCCE send/receive functions are implemented on top of the one-sided put and get operations [19]. The RCCE *send* function puts the message payload from the private memory to the MPB of the sender. The RCCE *recv* function gets the message payload from the MPB of the sender to the private memory of the receiver. Both functions are synchronous: the *send* can only terminate when the corresponding *recv* has finished receiving the data. To synchronize between the sender and the receiver, flags are used after putting and after getting the data. Writing or reading a flag is modeled by a single MPB access.

In the broadcast algorithms we consider, each communication step includes only pairs of nodes. In other words, there is no sequence in which more cores send a message to one core, and that core executes $receive$ for each of them, or vice versa. Therefore, we assume that, for each $send/receive$ pair, $send$ and $receive$ are called simultaneously (there is no waiting time). Under that assumption we can directly model the completion time of a $send/receive$ pair. First, we express it for one chunk:

$$\begin{aligned} C_{sr\_chunk}(s) = C_{put}\, mem(s, d^{mem}, d^{mpb}) + C_{get}^{mem}(s, d^{mpb}, d^{mem}) \\ + 2 \cdot \left(L_w^{mpb}(d^{mpb}) + C_r^{mpb}(1) + C_w^{mpb}(1)\right) \end{aligned}$$

The term $\left(L_w^{mpb}(d^{mpb}) + C_r^{mpb}(1) + C_w^{mpb}(1)\right)$ corresponds to the time to synchronize between the sender and the receiver and to reset the corresponding flag.

To transfer a multi-chunk message, the above function is repeatedly called for every chunk, until completion, so the completion time of a $send/receive$ pair of operations is:

$$C_{sr}(s) = \lfloor \frac{s}{M_{rcce}} \rfloor \cdot C_{sr\_chunk}(M_{rcce}) + C_{sr\_chunk}(s \bmod M_{rcce})$$

However, this model is not precise enough for some practical scenarios. Namely, if a core receives a short message and sends it further immediately afterwards, the completion time of the send operation is significantly reduced because the message stays in the on-chip cache (L1 or L2) after the receive operation. To take this into account, we assume negligible cost of reading from on-chip caches and express the completion time as:

$$\begin{aligned} C_{sr\_chunk\_cache}(m) = o_{put}^{mem} + m \cdot C_w^{mpb}(1) + C_{get}^{mem}(m, d^{mpb}, d^{mem}) + \\ + 2 \cdot \left(L_w^{mpb}(d^{mpb}) + C_r^{mpb}(1) + C_w^{mpb}(1)\right) \\ C_{sr\_cache}(m) = \lfloor \frac{m}{S_{max\_rcce}} \rfloor \cdot C_{sr\_chunk\_cache}(M_{rcce}) + C_{sr\_chunk\_cache}(s \bmod M_{rcce}) \end{aligned}$$

When modeling binominal and scatter-allgather algorithms, we assume that the whole message can fit in the on-chip L2 cache (256 KB per tile, i.e. 128 KB per core). Therefore, the presented models give the upper performance bound.

B.3 Binomial broadcast algorithm model

The binominal broadcast algorithm is based on a recursive tree, as already mentioned in Section 5. The set of nodes is divided into two subsets of $\lfloor \frac{P}{2} \rfloor$ and $\lceil \frac{P}{2} \rceil$ nodes. The root, belonging to one of the subsets, sends the message to one node from the other subset. Then, broadcast is recursively called on both subsets.

Obviously, the formed tree has $\lceil log_2(P) \rceil$ levels and in each of them the whole message is sent between the pairs of nodes, so the total latency, ignoring the cache effects discussed above, is:

$$L_{binominal\_nocache}(P, m) = \lceil log_2(P) \rceil \cdot C_{sr}(m)$$

Note that, after receiving the message, a core just sends it to other cores in the subsequent recursive invocations. This means that only the first send-receive pair, which involves the root and another core, should be modeled as C_{sr}, whereas the others are modeled as C_{sr_cache}. When this is taken into account, the latency of the binominal algorithm becomes:

$$L_{binominal}(P, s) = C_{sr}(m) + (\lceil log_2(P) \rceil - 1) \cdot C_{sr\_cache}(m) \tag{23}$$

This formula allows us to compute the latency of the binomial algorithm in Figure 6, Section 5.

B.4 Scatter-allgather broadcast algorithm model

As mentioned in Section 5, the *scatter-allgather* broadcast algorithm has two phases. During the *scatter* phase, the message is divided into P equal slices[6] of size $m_s = m/P$. Each core then receives one slice of the original message. The second phase of the algorithm is *allgather*, during which a node should obtain the remaining $P - 1$ slices of the message. The *allgather* phase implemented in RCCE_comm uses the Bruck algorithm [6]: At each step, core i sends to core $i - 1$ the slices it received in the previous step.

Scatter The *scatter* phase is done using a recursive tree, similar to the one used by the binominal algorithm. However, in this case we do not transfer the whole message in each step, but only a part of it. Thus, expressed recursively, the latency of the scatter part is:

$$\begin{aligned} L_{scat}(P, m_s) &= C_{sr}(\lfloor \frac{P}{2} \rfloor \cdot m_s) + L_{scat}(\lceil \frac{P}{2} \rceil, m_s), P > 2 \\ L_{scat}(2, m_s) &= C_{sr}(m_s) \\ L_{scat}(1, m_s) &= 0 \end{aligned}$$

Overall, the scatter algorithm does not benefit from temporal locality, which can be easily noticed by observing the actions of the root. Namely, the root sends different parts of the message in every recursive call (first, a half of the original message, then, a half of the non-cached half of the original message etc). Although other nodes might reuse data, the critical path is defined by the root which never does so. Therefore, the given formula holds even in the presence of caches.

Allgather In each iteration of the *allgather* algorithm, a core i sends one slice to core $i - 1$ and receives one slice from core $i + 1$. Therefore, the latency of this phase, without considering the caches, is:

$$L_{allgath\_nocache}(P, m_s) = 2 \cdot (P - 1) \cdot C_{sr}(m_s)$$

Note that in each of $P-1$ iterations, except in the first one, a core sends the slice of the message received in the previous iteration. In the first iteration, all cores have the corresponding slice of the message ready in

[6]For simplicity, we assume that $P|m$.

the cache (thanks to the scatter phase), except the root, which has never accessed its slice of the scattered message before. Therefore, in the cache-aware model, each but one occurrence of C_{sr} should be replaced by C_{sr_cache}:

$$L_{allgath}(P, m_s) = C_{sr}(m_s) + (2P - 3) \cdot C_{sr\_cache}(m_s)$$

Scatter-allgather Finally, the latency of the *scatter-allgather* algorithm run on P processes with a message of size m is:

$$L_{scat-allgath}(P, m) = L_{scat}(P, \frac{m}{P}) + L_{allgath}(P, \frac{m}{P}) \quad (24)$$

The throughput of the *scatter-allgather* algorithm in Table 2 (Section 5) is computed using this formula.

C Analysis of Different Tree Degrees for Use with OC-Bcast

We analyze the impact of k (tree degree) on the performance of OC-Bcast. Figure 9 shows the latency of OC-Bcast on the SCC for various values of k, based on Formula 21 in Appendix B.1.1. Note that we represent each tree depth with at least one value of k. For example, $k = 2$ gives a tree with 6 levels, $k = 3$ gives a 5-level tree etc. For the tree levels that correspond to more than one value of k, we represent the lowest and the highest value. Thus, the 3-level data tree is represented by $k = 7$ and $k = 46$.

We can notice that the latency is proportional to the number of levels in the tree, which is expected. Taking only this into account, $k = 47$ is clearly the best choice. Recall, however, that Section 3.3 identifies MPB contention as the limiting factor for using arbitrarily large values of k. More specifically, the contention becomes experimentally visible for $k > 24$. However, for any value of k in interval $[7, 24]$, the depth of the tree remains unchanged, so the latency does not decrease. Moreover, the cost of polling increases with k, which impairs the performance. This is especially obvious for very small messages: $k = 46$, and even $k = 47$ give worse performance than smaller values of k.

When we consider the throughput, it is clear from Formula 22 in Appendix B.1.2 that lower throughput is expected with higher values of k. However, the throughput of OC-Bcast with $k = 7$ is negligibly lower than with $k = 2$. Therefore, $k = 7$ can be chosen as the best fixed solution for the SCC when messages of all sizes are considered.

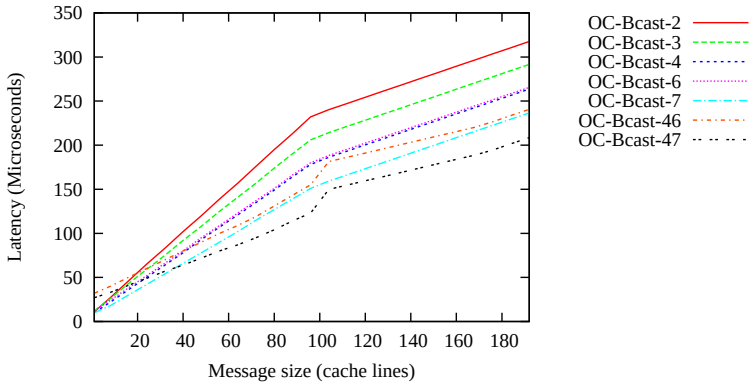

Figure 9: OC-Bcast latency for different values of k