Efficient and Predictable Group Communication for Manycore NoCs ⋆

Karthik Yagna, Onkar Patil, Frank Mueller

North Carolina State University

Abstract. Massive manycore embedded processors with network-on-chip (NoC) architectures are becoming common. These architectures provide higher processing capability due to an abundance of cores. They provide native core-to-core communication that can be exploited via message passing to provide system scalability. Despite these advantages, manycores pose predictability challenges that can affect both performance and real-time capabilities.
In this work, we develop efficient and predictable group communication using message passing specifically designed for large core counts in 2D mesh NoC architectures. We have implemented the most commonly used collectives in such a way that they incur low latency and high timing predictability making them suitable for balanced parallelization of scalable high-performance and embedded/real-time systems alike. Experimental results on a single-die 64 core hardware platform show that our collectives can significantly reduce communication times by up to 95% for single packet messages and up to 98% for longer messages with superior performance for sometimes all message sizes and sometimes only small message sizes depending on the group primitive. In addition, our communication primitives have significantly lower variance than prior approaches, thereby providing more balanced parallel execution progress and better real-time predictability.

1 Introduction

The future of computing is rapidly changing as manycore processors are becoming ubiquitous. Massive manycore platforms with NoC architectures are starting to penetrate high-performance systems, three-tier servers, network processing and embedded/real-time systems. These architectures provide a significant advancement due to an abundance of cores, which requires mesh-based NoCs for scalability. This allows a large number of cooperating tasks to be scheduled together. Tasks can employ group communication via messages over the NoC to achieve scalability and reduced latency. Their on-chip mesh NoC speeds are at par with processor clock frequencies as a hop between neighboring tiles/cores takes 1-2 cycles, which is a game changer in terms of communication. But such meshes are not without challenges as link contention and flow control become potential bottlenecks.

Poor group communication implementations can result in increased and highly variant latency due to NoC contention resulting in loss of predictability and imbalance in execution progress across cores. When multiple pairs of cores communicate, they may

⋆ This work was supported in part by NSF grants 0905181 and 1239246.

experience contention due to wormhole routing: After opening a source-destination path along a route, any other communication trying to use links on this path remains blocked until the former connection is closed. Such situations can be avoided using intelligent scheduling of each round of message exchanges.

For example, consider 9 cores taking part in all-to-all communication as in Figure 1. The task on core 3 is trying to send to the task on core 8, and the task on core 4 is trying to send to the task on core 2. This results in 2 messages, one from $3 \rightarrow 8$ and another from $4 \rightarrow 2$. When sent at the same time, contention on link $4 \rightarrow 5$ due to wormhole routing results in a delay for one of these messages as they are arbitrated within the NoC hardware routers. Thus, sending tasks experience highly variable latencies. The effect shown in this example is amplified with increasing NoC mesh sizes. Such situations can be avoided using intelligent scheduling of each round of message exchanges.

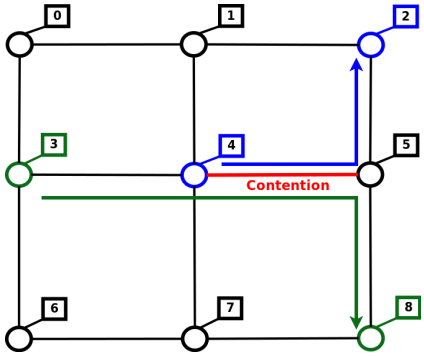

Fig. 1: NoC Contention

Furthermore, NoC architectures provide multiple message queues and networks [1–4]. On the TilePro64 [2], the User Dynamic Network (UDN) uses dynamic routing to forward messages from a source core to a destination core on Manhattan path following first the X- and then the Y-direction (X/Y-dimension ordered routing). The Static Network (SN) uses statically configured routes to forward packets received on each link. SN is faster than UDN in terms of packet forwarding speed (1 vs. 2 cycles), but is difficult to program and has route setup overhead.

This work contributes the design and implementation of group communication for large core counts utilizing 2D mesh NoC architectures. We employ efficient algorithms to reduce communication latency and exploit advanced NoC hardware features to provide better performance. We also ensure that communication uses contention-free paths and that no deadlock may occur. Deadlocks may occur due to head-of-line and path-based blocking, they are avoided by credit-based backpressure monitoring [5, 6] or by ensuring absence of link contention (this work, for collectives, only). We have implemented five commonly used group communication primitives [7].

Our Barrier, Broadcast and Reduce use a communication tree in which the cores are arranged as nodes and share a parent-child relationship. The communication tree is used to send messages to/from the root. The Barrier and Reduce implementations utilize the UDN, whereas Broadcast uses the SN. Our implementation of Alltoall exploits simple pattern-based communication, common in MPI [7] runtime system implementations, to

send messages concurrently, yet without contention, to reduce communication latency. This neither requires dynamic computation of a routing schedule nor incurs scheduling overhead or memoization of large routing tables. Our implementation uses message passing over the NoC of a TilePro64 and Intel SCC but is generic enough to be adopted to any 2D mesh NoC.

Experimental results on the TilePro hardware platform show that our implementation has lower latencies and less timing variability (lower variance) than prior work. We compared the performance of our implementation in micro-benchmarks against OperaMPI [8], a reference MPI implementation for the Tilera platform. Performance improvements of up to 95% are observed in communication for single packet messages with significantly higher timing predictability (lower variance), which supports more balanced execution progress for high-performance computing (HPC) and helps meet deadlines in embedded/real-time scenarios. Our port to the Intel SCC achieves similar results compared to the vendor libraries [9].

2 Design and Implementation

Our work assumes a generic, generalized 2D mesh NoC switching architecture similar to existing fabrics with high core counts [1–3, 9]. Each core is composed of a compute core, network switch, and local caches.

NoC Message Layer (NoCMsg): Our implementation provides an MPI-style message passing interface for NoCs. This facilitates basic point-to-point communication and supports our group communication. The NoC message layer implementation optionally provides flow control support. In our design, we turn off flow control when not required by program logic to further improve performance.

Group Communication Primitives: The key ideas behind our design of group communication primitives are to (1) reduce contention in the NoC; (2) exploit pattern-based communication to exchange messages concurrently; (3) reduce the number of messages by aggregation; and (4) leverage hardware features to improve performance. Due to these objectives, it is not feasible to simply resort to binomial trees for most collectives or other algorithms such as recursive doubling for allreduce since these algorithms are contention agnostic and will result in reduced performance over contention-sensitive NoCs.

We implemented the group communication on the Tilera TilePro64 and ported it to the Intel SCC [9] to demonstrate that our implementation is generic and can be extended to any 2D mesh NoC architectures.

2.1 Alltoall and Alltoallv

Alltoall/Alltoallv employ pattern-based communication, which allows several sets of tasks to exchange messages concurrently without contention as many exchanges split into multiple rounds.

The rounds are comprised of (1) direct (2) left and (3) right rounds. The direct round is further split into two subrounds. In subrounds, each task sends messages only along a straight path to its partner task. Tasks exchange messages along the X direction in direct subround 1 and along the Y direction in direct subround 2. In left rounds, each task sends messages along the X direction followed by the Y direction such that their path

follows a counter-clockwise direction. In right rounds, each task sends messages along the X direction followed by the Y direction such that their paths follow a clockwise direction. These cases are depicted in Figure 2. The XY dimension routing ensures that these directions are maintained consistently.

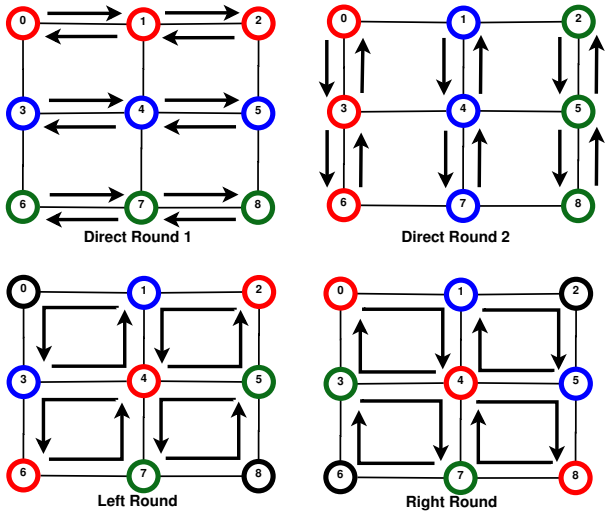

Fig. 2: Alltoall Rounds

The implementation details are sketched in Algorithm 1. In each round, the number of hops the message is forwarded is incremented until all tasks are covered. To begin, each task starts the direct subround one with one hop. The current column is selected by function Select-col. Tasks exchange messages with their neighbors one hop away along the X direction. This is done to ensure that the exchange is free of contention. The select functions operate as follows: Given a horizontal distance (e.g., two), determine the subset of cores via modulo arithmetic (example: every even core / rank modulo 2) that are active, followed by others round (example: every odd core / rank+1 modulo 2). Once the round has been completed, a barrier is forced between all tasks. The barrier ensures absence of contention across rounds. Subsequent rounds for other directions follow. The algorithm is unique in its absence of contention, which is key to the performance improvements we observed compared to naive exchange sequences with contention.

2.2 Barriers

We utilize modified 3-ary tree-based barriers that distribute the work evenly among nodes to minimize cycle differences upon barrier completion. In a 2D mesh, nodes have a most 4 neighbors so that in a barrier tree, any interior node receives a message from one neighbor and relays this message to 3 others. This provides maximal link coverage with minimal tree height (which is optimal). Hence, the tree is 3-ary on the interior, 4-ary for the root (to be precise) and of lower degree (2/1/0) for nodes close to the leaves and leaves themselves. The root of this tree is placed in the center of the NoCMsg grid to minimize latency (hops). The tree is constructed as part of the initialization process. Children notifying their parents when they have entered the barrier, up to the root.

Algorithm 1 Alltoall: x,y are task ranks; myrow,mycol are Cartesian coordinates of x.

```
1: function NoCMsg-Alltoall
2:     Xmax ← gridwidth; Ymax ← gridheight
3:     for xhops ← 1,Xmax do // Direct subround 1 (DR1)
4:         currcol = Select-col(DR1, xhops) // select column for this round
5:         if mycol == currcol then                                  ▷ my column's turn
6:             UDN-xchg(x+xhops, y); UDN-xchg(x-xhops, y)
7:         end if
8:         Barrier()
9:     end for
10:    for yhops ← 1,Ymax do // Direct subround 2 (DR2)
11:        currrow = Select-row(DR2, yhops) // select row for this round
12:        if myrow == currrow then                                  ▷ my row's turn
13:            UDN-xchg(x, y+yhops); UDN-xchg(x, y-yhops)
14:        end if
15:        Barrier()
16:    end for
17:    for yhops ← 1,Ymax do // Left round (LR)
18:        for xhops ← 1,Xmax do // select row, column for this round
19:            currrow = Select-row(LR, yhops); currcol = Select-col(LR, xhops)
20:            if myrow,mycol == currrow,currcol then
21:                UDN-xchg(x-xhops, y+yhops); UDN-xchg(x+xhops, y-yhops)
22:            end if
23:            Barrier()
24:        end for
25:    end for
26:    for yhops ← 1,Ymax do // Right round (RR)
27:        for xhops ← 1,Xmax do // select row, column for this round
28:            currrow = Select-row(RR, yhops); currcol = Select-col(RR, xhops)
29:            if myrow,mycol == currrow,currcol then
30:                UDN-xchg(x+xhops, y+yhops); UDN-xchg(x-xhops, y-yhops)
31:            end if
32:            Barrier()
33:        end for
34:    end for
35: end function
```

Once the root has received notifications from all children, it broadcasts a notification back down the tree by replying to its children and returns from the barrier call, as do the children. UDN is used to send/receive synchronization packets and their replies. Figure 3 shows an example of a barrier tree for a 4x4 grid.

Flow control is not needed in barriers as a prerequisite of entering a barrier is that all outstanding sends/receives of local cores have completed. The synchronization packet is small enough so that the core can drop an entire synchronization packet into its output queue. It can then start a blocking send operation that halts the core's pipeline until synchronization packets become available. This technique significantly reduces synchronization costs when all cores are ready, yet conserves power when they are not.

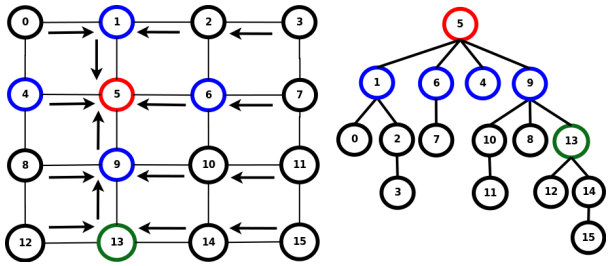

Fig. 3: Barrier Tree: Setup

2.3 Broadcast

Our Broadcast uses the SN of the TilePro64. The SN is more intricate to program and suffers from route setup overhead. However, message forwarding incurs zero overhead (due to a static route configuration). Since broadcast has a single sender and multiple receivers, the number of route configurations is low. This was the motivation behind using SN for the broadcast implementation.

We designed a tree-based algorithm rooted at the task performing the broadcast. Each task determines the root's row and column and configures the SN route. The route setup in the root is such that the message from the core is sent on its available links. All the tasks in the same column as the root have their route configured such that they receive from the root along the Y direction and send the message along other available links. Tasks in other columns receive along one X direction and send the message along the other X link.

For example, let the task with rank 5 initiate a broadcast. Then, its routes are set up to send the message from the core to all the links. The routes of tasks on cores in column one will be set up such that they send out the received message along the X and Y directions. The routes in all the other tasks will be set up in such a way that they will receive and forward along the X direction. This results in a broadcast tree as shown in Figure 4. Different nodes have different route setup depending on their relative position to the root node. The root node is highlighted in red. The blue nodes receive along the Y direction and send along X direction (East and West). The green nodes receive along the X direction (from West) and send along the X direction (toward East). All the other nodes have receive along the X direction.

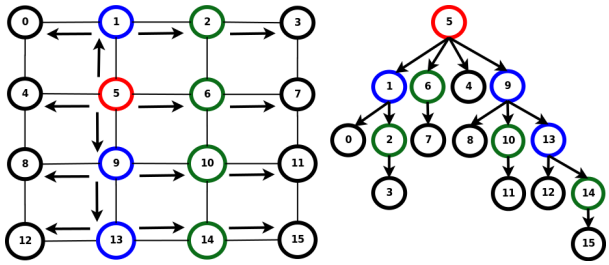

Fig. 4: Broadcast Tree: Static Routes Configuration

The static route of each task is configured inside the Broadcast call such that the message from the root flows to each leaf task. Our current implementation requires only a single route configuration per task and is contention-free.

2.4 Reduce and AllReduce

Our Reduce is similar to the barrier. The reduction operation is performed along the tree. Each child task sends its partial result upward toward the root. The root reduces the partial results to obtain the final result. The construction of the reduction tree is different from that of the Barrier. The reduction tree maps to a NoC grid such that the root task becomes the root of the tree. The tasks along its row become first-level children. The tasks in each column become second-level children to the first-level ones. Via recursive refinement, the algorithm extends to larger meshes, where more levels would be employed.

For example, let rank 5 be the root for the reduction operation. The tasks along its row become the first-level children (in this case, tasks with rank 4,6 and 7). These first-level children become children of the root. Each column will therefore have a root or a first-level child. All the other tasks become children of the root or first-level children along their column. In the example, rank 5 becomes the root with ranks 1,4,6,7,9 and 13 as its first-level children. Rank 4 will have ranks 0,8 and 12 as 2nd-level children etc. This reduction-tree setup is shown in Figure 5. To avoid contention we impose rounds of communication per level and distance: distance 2 cores to level 1 send their values first (e.g., core 15), followed by distance 1 (e.g, core 11); followed by distance 2 cores to root (e.g., core 7) and distance 1 (e.g., core 6). Each of these four rounds has no link contention by tree construction.

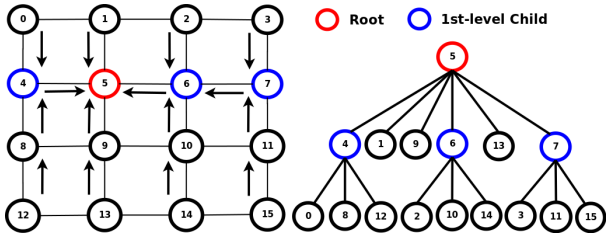

Fig. 5: Reduction Tree: Setup

A reduction tree constructed in this fashion has two major advantages: (1) The implementation is simple and scalable and (2) the entire reduction takes place in two steps irrespective of the size of the NoC grid. The first step occurs in parallel for the root and its 1st-level children, where they receive and reduce values from their respective 2nd-level children. In the second step, the root will receive partial results from the 1st-level children and perform the reduction operation.

AllReduce is an extension of Reduce. It is implemented by performing a Reduce relative to rank 0, followed by a broadcast from rank 0 to all other tasks in the group.

3 Framework

Experiments were conducted on a 700MHz 64-core Tilera TilePro processor (TilePro 64) and the Intel SCC. Our implementation is written in C and the programs were

compiled with Tilera's MDE 3.03 tool chain/Intel's ICC at the O3 optimization level with respective C/C++/Fortran compilers. Some cores are used for specific purposes and are not available to run user programs restricting the maximum grid size to 7x7 on Tilera. In our experiments, grid sizes ranging from 2x2 to 7x7 were utilized allowing for a total of six configurations (omitting rectangular, non-square configurations, which are also valid).

OperaMPI implements the MPI 1.2 standard [10] for C. It is layered over Tilera's iLib, an inter-tile communication library that utilizes the UDN NoC network. The iLib library is vendor-supplied software and allows developers to easily take advantage of many features provided by the Tilera architecture. Message passing is one such feature. Point-to-point messages are directly supported by iLib closely resembling the equivalent MPI semantics. Internally, iLib utilizes interrupt-based virtual channels and complex packet encodings to synchronize senders and receivers to set up such point-to-point communication. However, iLib only supports a limited number of collective operations, namely broadcast and barrier. Hence, OperaMPI creates virtual overlays (e.g., binomial trees for reductions, recursive doubling for allreduce) to implement more complex MPI collectives, which result in NoC contention in contrast to our approach. OperaMPI has been evaluated against the IBM, Intel, MPICH and SPEC MPI test suites. Some of these results and implementation details (such as tree reduce/alltoall) are discussed in [8], which provides a fair foundation for comparison to OperaMPI. Furthermore, OperaMPI is the only working MPI implementation on Tilera and other ports (MPICH or Open MPI) are not only beyond the scope of this work but would be subject to the same NoC design/implementation choices we made.

Results of NoCMsg, our scalable and predictable NoC messaging framework, and the third-party OperaMPI and RCKMPI [11, 12] were compared for Tilera and SCC, respectively. NoCMsg traffic is routed over the UDN/SN with 1-2 cycles and 1 cycle per hop, respectively, using polling instead of relying on iLib/OperaMPI interrupts with messages packaged in 128 byte flits.

4 Experimental Results

We experimented with of NoCMsg group communication repeating each run 21 times and reporting averages on the Tilera TilePro64 and also the Intel SCC [9] to demonstrate portability. We report the Tilera results first.

4.1 Single packet messages

The benchmark timing results for single packet messages are depicted in Figures 6-10 for 5 collectives. Time on the y-axis is plotted logarithmically in microseconds for averaged benchmark runs over different number of tasks (equal to cores) in the range from 4 to 49. Tables 1 and 2 depict execution time variances for each micro-benchmark for varying number of tasks for NoCMsg and OperaMPI, respectively.

We observe that as the number of tasks increases, the execution time of group communication increases. In case of Opera, the increase in runtime is significant for larger number of tasks. In comparison, our NoCMsg implementation is highly efficient, and increases in runtime are gradual. For Alltoall, our pattern-based approach effectively eliminates network contention resulting in a reduction of execution time by about 62% (from 2027 down to 761) for a 7x7 grid size with a variance of 0.4 to 5.6 depending on

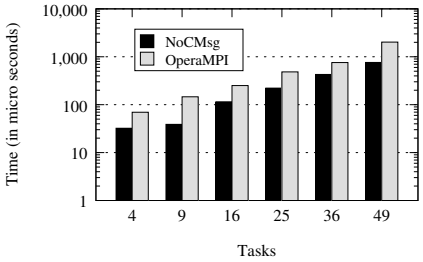

Fig. 6: Timing Results for Alltoall

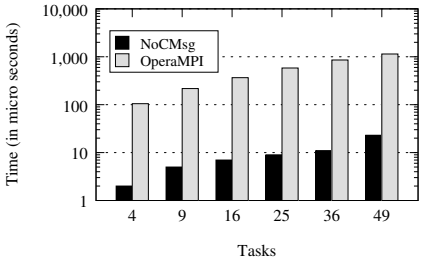

Fig. 7: Timing Results for Reduce

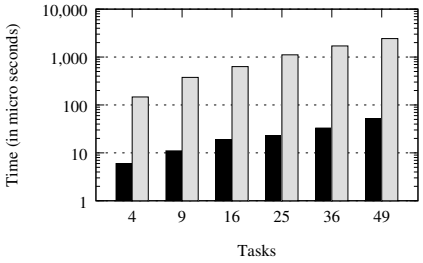

Fig. 8: Timing Results for AllReduce

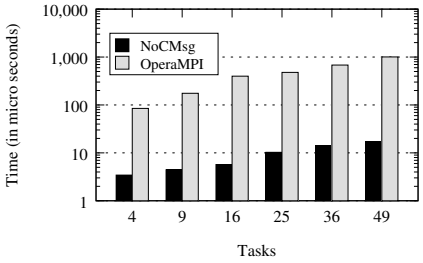

Fig. 9: Timing Results for Barrier

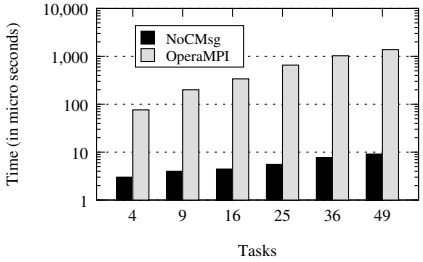

Fig. 10: Timing Results for Broadcast

the numbers of cores used. This variance is several orders of magnitude lower than that of the OperaMPI particularly for larger number of cores.

Barrier and Broadcast are our most efficient collectives with up to 98% reduction in execution time, i.e., Opera takes up to two orders of magnitude longer due to contention. Broadcast uses the SN with a single route setup (to configure the communication tree, setup time included) and minimal routing overhead. The SN is typically faster than the UDN, which makes Broadcast our most efficient and predictable collective in comparison. (Over UDN, broadcast would be close to the time of a barrier.) Execution time increases only by a factor of 3.5 as the grid size is gradually changed from 2x2 to 7x7 with a variance of less than 0.6 for all cases.

Table 1: NoCMsg Runtime Var. $[\mu s^2]$

Num tasks	4	9	16	25	36	49
Alltoall	0.7	0.4	0.7	5.6	1.3	1.6
Barrier	0.5	0.8	0.4	1.6	1.1	5.6
Broadcast	0	0	0.2	0.24	0.53	0.12
Reduce	0.2	0.12	1.1	1.06	3.96	7.27
AllReduce	1.34	2.29	2.49	26.77	31.55	50.82

Table 2: OperaMPI Runtime Variance $[\mu s^2]$

Num tasks	4	9	16	25	36	49
Alltoall	3	984	18	2277	133330	622903
Barrier	750	303	29385	1838	2911	32117
Broadcast	7	57	259	4541	3004	3869
Reduce	545	686	21	2007	9980	3431
AllReduce	11	51	49	3839	5536	7517

Our implementations of Reduce and Allreduce have 97% and 98% lower execution time, respectively, than OperaMPI for all tested grid sizes, i.e., OperaMPI is up to nearly two orders of magnitude slower. However, these collectives have a larger variance than others due to the two-step reduction employed by Reduce operations.

4.2 Varying message sizes

Figures 11-15 depict the averaged performance for varying message sizes and number of tasks (cores) for both our NoCMsg implementation and OperaMPI, the reference implementation. Notice that execution times are plotted on a logarithmic scale on the y-axis. The solid lines represent execution times for NoCMsg while the dotted lines represent execution times for OperaMPI. The legend further indicates the number of tasks, i.e., key N4 represents NoCMsg with 4 tasks while O4 depicts OperaMPI with 4 tasks with the same color coding for identical grid sizes (in the same order as the line graphs). The range of grid sizes ranges from 4 to 49 total number of tasks (cores).

Figure 11 shows the execution time of the Alltoall collective for message sizes up to 4KB, which is an inset to Figure 12, the latter of which extends to 1MB sizes. The execution and communication times increase with an increase in message size for both NoCMSg and OperaMPI. Our NoCMsg implementation of Alltoall performs very well for small messages with savings between 43%-62% up to a threshold (256 bytes to 4KB depending on message size and number of tasks, see Figure 11). Yet, as message sizes increase, performance degrades, and for message sizes greater than this threshold, OperaMPI outperforms NoCMsg (see Figure 12). This is because our Alltoall implementation is split into rounds of exchanges followed by barrier synchronization to ensure absence of contention. For large messages, this results in noticeable overhead. OperaMPI's Alltoall implementation is split into N-1 stages, where N is the total number of tasks. At each stage, one task takes a turn to send to a partner. While their setup is subject to contention to create a virtual channel, transmission proceeds without contention once a channel has been created, which provides higher bandwidth for large messages. Yet, prior work has shown that typical applications tend to utilize collectives with very small message payloads [13], which indicates that our NoCMsg covers the critical path for most applications and nicely complements OperaMPI's advantage for large messages.

The timing results for Reduce and Allreduce are shown in Figures 13 and 14, respectively. The execution time of our implementation is 48%-98% lower than that of OperaMPI for all message sizes up to 1MB (and beyond), i.e., OperaMPI is up to two orders of magnitude more costly. However, the gap gradually decreases to about one order of magnitude difference. Asymptotically, the performance results of the two implementations approach each other for very large (but, in practice, unrealistic) message sizes. The implementation of Reduce in OperaMPI uses a communication tree but does not map it to the NoC in a contention-free manner. The resulting contention causes larger communication/execution times. The same observation also holds for AllReduce, a Reduce followed by a Broadcast. Since Reduce dominates the communication and execution time in AllReduce, its behavior is the same as Reduce.

We also compared our version to one that uses recursive doubling. Figure 16 indicates that the former outperforms the latter for larger core sizes while the latter to results in less setup overhead and thus better performance for smaller number of tasks.

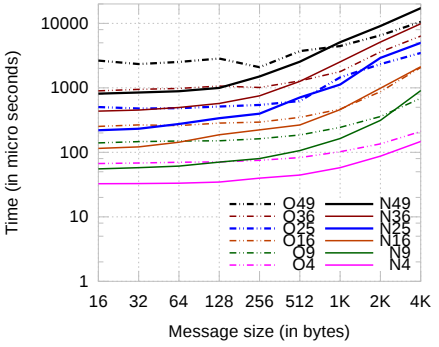

Fig. 11: Alltoall: Inset Msgs $\leq$ 4 KB

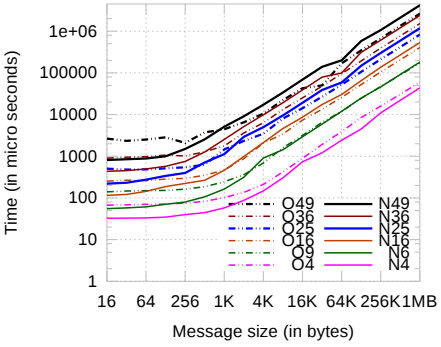

Fig. 12: Alltoall: Varying Msg Sizes

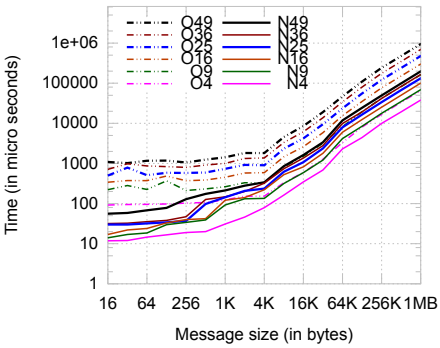

Fig. 13: Reduce: Varying Msg Sizes

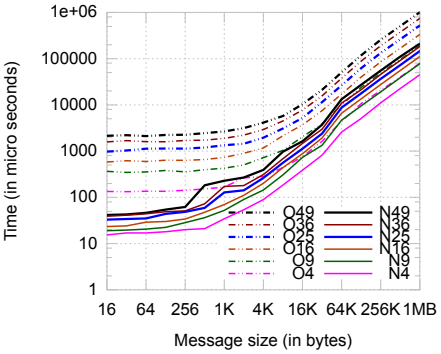

Fig. 14: AllReduce: Varying Msg Sizes

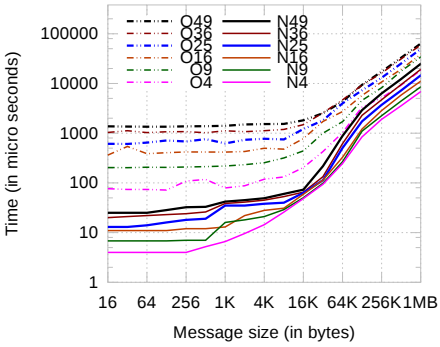

Fig. 15: Broadcast: Varying Msg Sizes

# tasks	reduce+bcast	recursive doubling
4	13.57	4
9	16.29	6
16	25.43	10
25	26.71	26
36	30.29	51
49	33.00	56

Fig. 16: Allreduce Variants: Avg. Time [usec]

Collectives tend to involve all tasks in a code, so scalability is key. At 49 cores, the former is almost twice as fast as the latter and has lower asymptotic behavior indicating a higher potential for scalability. We identified the absence of contention in our design as the main contributor of the reduced time over doubling.

Figure 15 represents the execution time of Broadcast for different message sizes. OperaMPI implements Broadcast using a tree-like communication pattern, where the root task initiates the broadcast by sending the message to another task. The two tasks

send the message to another two tasks. This transitive distribution of messages continues and eventually terminates after $log(N)$ steps, where N is number of tasks. This communication tree approach is efficient but does not map to the NoC in a contention-free manner. Similar to Reduce, there is always contention resulting in larger communication and execution time. Our Broadcast implementation uses SN unlike OperaMPI, which uses UDN. Routing overhead in SN is lower than that in UDN. This also contributes to better performance and lower execution time. From the NoCMsg curve, we can see that the execution time remains constant for message sizes up to 256 bytes since only 1-2 flits are required (latency bound). Beyond 256 bytes, the execution time of NoCMsg Broadcast increases at a higher rate than that of OperaMPI as it transitions from being latency to being bandwidth bound. This continues up to a message size of 128KB, after which the rate of increase in execution time with increase in message size is nearly same for both NoCMsg and OperaMPI (bandwidth bound). Again, the execution times of the two implementations approach each other asymptotically for very large (but, in practice, unrealistic) message sizes.

Overall, these results show that our NoCMsg implementation is ideal for all / small message sizes depending on the collective primitive. As prior work has indicated, typical MPI applications utilize collectives with very small message payloads [13], and embedded/real-time applications follow a similar trend for numerical, actuator-based control systems. This underlines the contribution of our work for high-end and embedded/real-time applications alike as NoCMsg provides better performance and timing predictability than prior related work for the common case, and, moreover, for realistic 2D meshes without wrap-around network links at grid boundaries.

4.3 NAS Parallel Benchmark

We used NPB version 3.3 to evaluate our implementation. NPB by default uses strong scaling, where input sizes stay fixed for different numbers of cores. We used strong scaling due to input constraints for MG (input class A, to mostly remain on chip in L2 cache) and our own weak scaling inputs for all other benchmarks [14]. Weak scaling ensures that the computational work per core remains the same as the number of cores cooperating in a parallel application is increased. We choose inputs to remain resident in private L2 so that results measure the NoC properties rather than being dominated by off-chip memory traffic.

Figure 17 depicts the results for MG with strong scaling. MG is memory intensive and uses long- and short-distance inter-processor communication. The number of processes grows as a power of 2 resulting in 5 different grid sizes. We observe that NoCMsg is faster than OperaMPI for all grid sizes. Strong input scaling causes the total time to reduce as the number of tasks increases. For small task sizes, NoCMsg is much faster than OperaMPI. As the number of tasks increases, the time difference decreases. This is because MG is memory intensive with limited inter-process communication, and for large grid sizes the performance improvement due to efficient communication declines.

We used weak scaling for IS, FT, CG and LU. IS sorts integer numbers and perform frequent all-to-all collective communication for rebalancing. FT is a discrete 3D Fast Fourier Transform solver for partial differential equations using all-to-all communication. CG estimates Eigenvalues using the conjugate gradient method with irregular

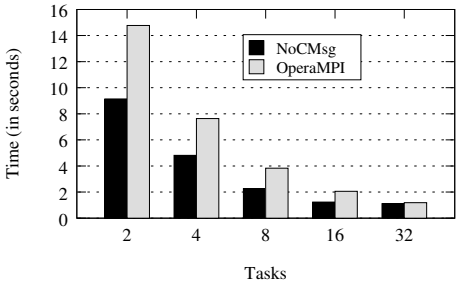

Fig. 17: NPB MG: Strong Scaling

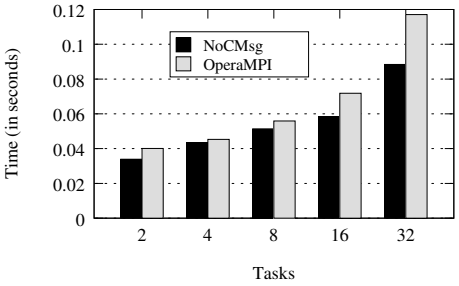

Fig. 18: NPB IS: Weak Scaling

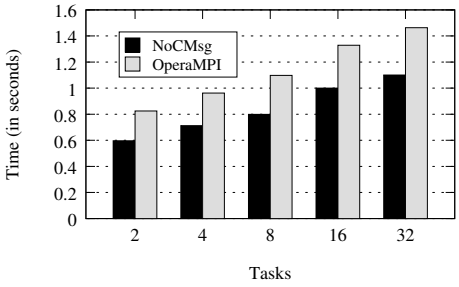

Fig. 19: NPB CG: Weak Scaling

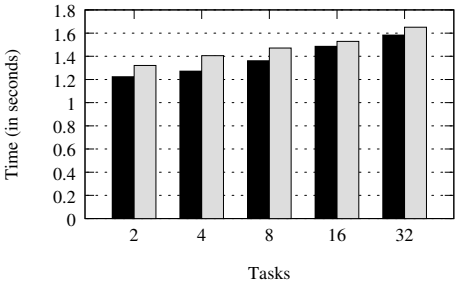

Fig. 20: NPB FT: Weak Scaling

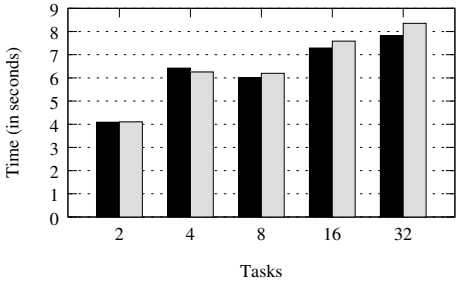

Fig. 21: NPB LU: Weak Scaling

memory accesses and communication. LU is a computational fluid dynamics code with stencil-style point-to-point communication.

For IS and CG (Figures 18+19), the execution time of NoCMsg is lower than that of OperaMPI. Performance benefits include both collectives and point-to-point improvements with improvements due to collectives (this paper) and point-to-point messages (for the latter, see [5, 6]). The difference in execution time increases with as the number of tasks grows since inter-process communication dominates. For FT (Figure 20), small differences in execution time between NoCMsg and OperaMPI are due to larger messages sizes.

For LU (Figure 21), OperaMPI is faster than NoCMsg for small numbers of tasks when computation dominates total execution time. As the number of tasks increases, the inter-task communication starts to dominate the total execution time. The execution time of NoCMsg grows slower than that of OperaMPI, indicating that for larger numbers of tasks NoCMsg provides better performance.

4.4 Portability

The Intel SCC is a single chip with 48 cores connected by a 6x4 mesh with x/y dimension-ordered routing, where each of the 24 routers is attached to one tile with two local cores [15, 9]. There is no cache coherence across the mesh; instead, message passing is supported via bare-bones message with RCCE and MPI-style communication with RCKMPI [11, 12]. The mesh is subject to link contention (as was Tilera) but also core contention due to two cores sharing a single router. We ported NoCMsg to the Intel SCC using the same design philosophy in that link (and, for the SCC, also core) contention should be avoided in the implementation of any collective communication primitive, which is not the case for RCCE and RCKMPI collectives. We were able to ensure absence of contention using pattern-based collectives in communication rounds over one-sided put/get primitives of RCCE, which is the lowest level of communication on the SCC.

Figures 22-25 depict the effective throughput for different message sizes for RCCE, RCKMPI and NoCMsg (from bottom to top), averaged over 10,000 runs each. We observe that NoCMsg results in 2-5 times higher maximum throughput depending on the type of collective as NoCMsg approaches the theoretical bandwidth of the SCC. (Figure 22 is lacking RCCE results since the API does not support alltoall natively.) This throughput is constrained by the shared MPB SRAM buffer of 384KB, which is split into 8KB regions per tile, and the mesh frequency of 1066MHz. Peaks are reached earlier for [All]Reduce (4KB) than for Broadcast/Alltoall (128KB and 196KB) due to the cost of reductions vs. significant link contention for Alltoall, whereas Broadcast levels out at 15 MB/s but remains high. This is larger than the individual MPB size, which shows that the MPB streams multi-packet messages efficient as long as the data is L2 cache resident, i.e., not exceed 256KB.

Figure 26 shows that the execution time of broadcasting a 236KB message remains constant even as the number of cores increases due to pattern-based communication whereas that of RCCE and RCKMPI is one to four orders of magnitude higher for 48 cores. (Notice that 16 or more cores show nearly the same performance. The mesh only has 24 routers, and from 24 to 48 cores, a single copy between core pairs over their pair-wise local bus interconnect suffices.) Figure 27 depicts the time of a 48-core barrier for each core involved in this collective. We observe that barriers are also 2-3 orders of magnitude faster under NoCMsg compared to RCCE/RCKMPI. We also see that barrier completion varies between even and odd cores, which is visible by the jagged curve for NoCMsg. This is due to two cores sharing a router, where the odd core is served first and then signal its sibling (even) core. The jagged lines for RCCE/RCKMPI are not visible since the y-axis is logarithmic.

Overall, NoCMsg is shown to be portable in its code base and results in significantly better throughput and performance for collectives than contention-oblivious communication methods.

5 Related Work

Communication patterns and communication trees as a means to implement collective operations have been well studied [16]. Barnett's broadcast [17] is loosely based on spanning binomial trees. Yang's tree-based multicast [18] constructs a quad-branch

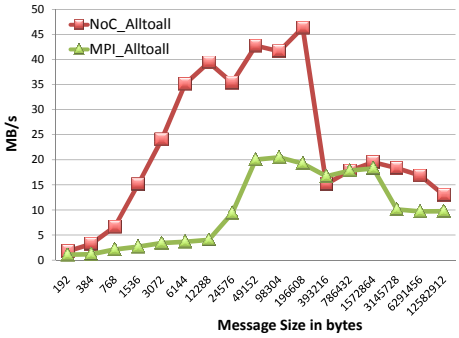

Fig. 22: Throughput for Alltoall

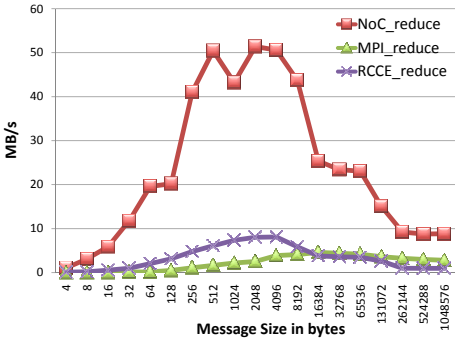

Fig. 23: Throughput for Reduce

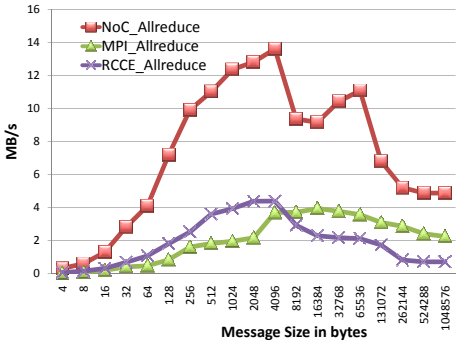

Fig. 24: Throughput for AllReduce

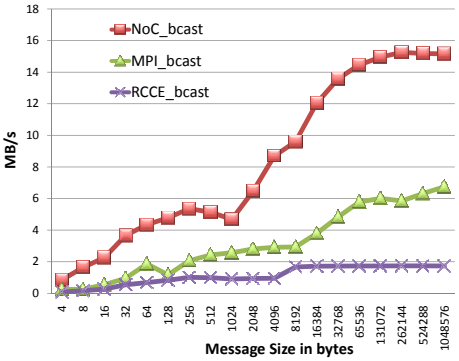

Fig. 25: Throughput for Broadcast

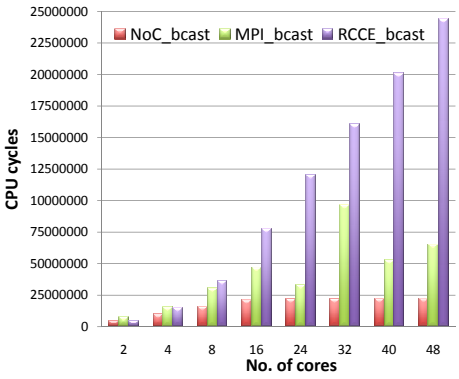

Fig. 26: Broadcast Timing over Cores

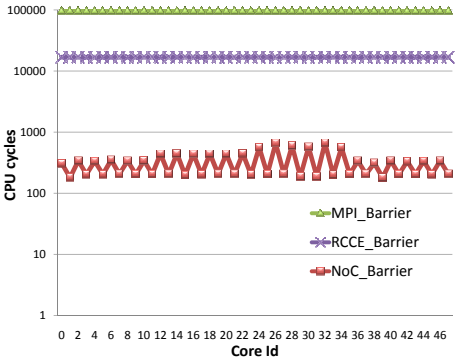

Fig. 27: Timing Results per Core for Barrier

multicast (QBM) tree, a logic tree rooted at the source node of a multicast and has four subtrees. Our implementation of Broadcast uses a communication tree in a contention-free manner. But unlike QBM, our approach does not require special registers/buffers in the routers, support of double-X/Y routing and changes to message headers. QBM also cannot handle alltoallv. Topology-aware collectives [19] reduce but cannot eliminate congestion for HPC interconnect whereas our work eliminates any contention.

Several approaches apply graph theoretic concepts to build efficient trees, e.g., a multi-level broadcast tree using extended dominator nodes (EDN) [20] but requires an expensive all-port instead of the current single-port NoC communication architectures.

Barriers were implemented in hardware for IBM's BlueGene [21]. Router-based barriers [22, 23] require long headers to carry the information of multiple destinations and incur additional processing at each node. Tree-based barriers [24] partition the 2D mesh into four overlapping quadrants using the chosen root node as the origin whereas Barrier Tree for Meshes (BTM) utilizes a 4-ary synchronization tree constructed in a recursive manner [25]. Our implementation of barrier also uses a tree rooted at a chosen root node, but, unlike the others, neither requires a 2D division into submeshes nor relies on special registers for tree construction.

Our implementation of alltoall exploits pattern-based communication to concurrently exchange messages between partners. On the surface, this approach shares design strategies with Thakur's direct algorithm [26], which assigns mesh nodes ordinal numbers 0..N-1 in a row-major fashion. During step k, for $k = 1..N-1$, the node with ordinal number i sends a message to the node whose ordinal number is an exclusive or (XOR) of i and k. This results in a communication pattern similar to ours under dimension order routing. Unlike our approach, their direct algorithm suffers from link contention, which is also the case for BG/L's random lists [27].

Others split tasks into disjoint communication groups [28–31]. Our implementation also uses a bottom-up approach, but neither requires a grid division into smaller submeshes nor results in contention.

More recent approaches focus on building static schedules for alltoall communication [32]. Some approaches perform path selection, core mapping and time-slot allocation intelligently to resolve conflicts on shared networks [33]. Others exploit time-division multiplexing on NoCs to solve the slot and path selection problem and this avoid contention [34]. Unlike these approaches, our implementation neither requires dynamic route calculations nor offline pre-calculations nor storage of large routing tables. Our tree-based implementations rely on the relative position of nodes to the root and take advantage of 2D mesh topology to map a tree in a contention-free manner, which is novel. This keeps our implementation simple, generic and scalable with minimum overhead.

6 Conclusion

We designed a set of efficient and predictable group communication primitives using message passing utilizing NoC architectures. The primitives employ highly efficient algorithms to provide contention-free communication and utilize advanced NoC hardware features. These primitives improve performance and reduce imbalance for HPC applications while providing higher timing predictability for embedded/real-time systems.

Our implementation of the most commonly used collectives reduces the communication time over a reference MPI implementation on TilePro64 by up to 95% for single packet messages and up to 98% for larger messages. NoCMsg has superior performance over OperaMPI irrespective of message size for all but one collective: For Alltoall, NoCMsg performs better for message sizes up to 256 Bytes while OperaMPI performs better for larger messages. The evaluation of NPB codes shows that NoCMsg outperforms OperaMPI for actual workloads. Our NoCMsg for the Intel SCC shows orders of magnitude increases in performance and 2-5 times higher throughput. NoCMsg

thus nicely complements prior work that is efficient at larger (yet less common) message sizes for this case. Additionally, the variance of execution times for our implementation is several orders of magnitude lower than that of the reference MPI implementation, making our implementation ideal for balanced high-end as well as embedded/real-time applications. And instead of assuming ideal NoC symmetry with wrap-around links on the 2D boundaries, our work addresses realistic 2D meshes without wrap-around, such as present in contemporary NoC hardware designs.

References

1. Intel: Tera-scale research prototype: Connecting 80 simple cores on a single test chip ftp://download.intel.com/research/platform/terascale/terascaleresearchprototypebackgrounder.pdf.
2. Tilera: Tilera processor family. `www.tilera.com/products/processors.php`
3. Wentzlaff, D., Griffin, P., Hoffmann, H., Bao, L., Edwards, B., Ramey, C., Mattina, M., Miao, C.C., III, J.F.B., Agarwal, A.: On-chip interconnection architecture of the tile processor. IEEE Micro **27** (2007) 15–31
4. Adapteva: Adapteva processor family. `www.adapteva.com/products/silicon-devices/e16g301/`
5. Zimmer, C., Mueller, F.: Nocmsg: Scalable noc-based message passing. In: International Symposium on Cluster Computing and the Grid (CCGRID). (2014) 186–195
6. Zimmer, C., Mueller, F.: Nocmsg: A scalable message passing abstraction for network-on-chips. ACM Transactions on Architecture and Code Optimization **12**(1) (February 2015)
7. Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J.J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L., Woodall, T.S.: Open MPI: Goals, concept, and design of a next generation MPI implementation. In: European PVM/MPI Users' Group Meeting. (September 2004) 97–104
8. Kang, M., Park, E., Cho, M., Suh, J., Kang, D.I., Crago, S.P.: Mpi performance analysis and optimization on tile64/maestro. In: Workshop on Multi-core Processors for Space — Opportunities and Challenges. (July 2009)
9. Mattson, T., van der Wijngaart, R., Riepen, M., Lehnig, T., Brett, P., Haas, W., Kennedy, P., Howard, J., Vangal, S., Borkar, N., Ruhl, G., Dighe, S.: The 48-core scc processor: the programmer's view. In: Supercomputing. (Nov 2010)
10. Gropp, W., Lusk, E., Doss, N., Skjellum, A.: A high-performance, portable implementation of the MPI message passing interface standard. Parallel Computing **22**(6) (September 1996) 789–828
11. Wijngaart, R.V.D., Mattson, T.: RCCE: A small library for many-core communication (2010)
12. Compres Urena, I., Riepen, M., Konow, M.: Rckmpi lightweight mpi implementation for intels single-chip cloud computer (scc). Recent Advances in the Message Passing Interface (2011) 208–217
13. Vetter, J., Mueller, F.: Communication characteristics of large-scale scientific applications for contemporary cluster architectures. In: International Parallel and Distributed Processing Symposium. (April 2002)
14. Gustafson, J.L.: Reevaluating Amdahl's law. Communications of the ACM **31**(5) (May 1988) 532–533
15. Howard, J.e.a.: A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In: IEEE International Solid-State Circuits Conference. (Feb 2010) 108–109
16. McKinley, P.K., jia Tsai, Y., Robinson, D.F.: A survey of collective communication in wormhole-routed massively parallel computers. IEEE COMPUTER **28** (1994) 39–50

17. Barnett, M., Payne, D.G., van de Geijn, R.A.: Optimal broadcasting in mesh-connected architectures. Technical report, Austin, TX, USA (1991)
18. Yang, J.S., King, C.T.: Efficient tree-based multicast in wormhole-routed 2d meshes. In: Proceedings of the 1997 International Symposium on Parallel Architectures, Algorithms and Networks. ISPAN '97, Washington, DC, USA, IEEE Computer Society (1997) 494–
19. Sack, P., Gropp, W.: Faster topology-aware collective algorithms through non-minimal communication. In: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. (2012) 45–54
20. Tsai, Y.j., McKinley, P.K.: Broadcast in all-port wormhole-routed 3d mesh networks using extended dominating sets. In: Proceedings of the 1994 International Conference on Parallel and Distributed Systems, Washington, DC, USA, IEEE Computer Society (1994) 120–127
21. Ramakrishnan, V., Scherson, I.D.: Efficient techniques for nested and disjoint barrier synchronization. J. Parallel Distrib. Comput. **58**(2) (August 1999) 333–356
22. Lin, X., McKinley, P.K., Ni, L.M.: Deadlock-free multicast wormhole routing in 2-d mesh multicomputers. IEEE Trans. Parallel Distrib. Syst. **5**(8) (August 1994) 793–804
23. Panda, D.K.: Fast barrier synchronization in wormhole k-ary n-cube networks with multidestination worms. In: Proceedings of the 1st IEEE Symposium on High-Performance Computer Architecture. HPCA '95, Washington, DC, USA, IEEE Computer Society (1995) 200–
24. Yang, J.S., King, C.T.: Designing tree-based barrier synchronization on 2d mesh networks. IEEE Trans. Parallel Distrib. Syst. **9**(6) (June 1998) 526–534
25. Moh, S., Yu, C., Lee, B., Youn, H.Y., Han, D., Lee, D.: Four-ary tree-based barrier synchronization for 2d meshes without nonmember involvement. IEEE Trans. Comput. **50**(8) (August 2001) 811–823
26. Thakur, R., Choudhary, A.: All-to-all communication on meshes with wormhole routing. In: In Proceedings of the 8 th International Parallel Processing Symposium. (1994) 561–565
27. Almási, G., Heidelberger, P., Archer, C.J., Martorell, X., Erway, C.C., Moreira, J.E., Steinmacher-Burow, B., Zheng, Y.: Optimization of mpi collective communication on bluegene/l systems. In: International Conference on Supercomputing. (2005) 253–262
28. Bokhari, S., Berryman, H.: Complete exchange on a circuit switched mesh. In: Scalable High Performance Computing Conference, 1992. SHPCC-92, Proceedings. (1992) 300–306
29. Sundar, N.S., Jayasimha, D.N., Panda, D., Sadayappan, P.: Complete exchange in 2d meshes. In: Scalable High-Performance Computing Conference, 1994., Proceedings of the. (1994) 406–413
30. Suh, Y.J., Shin, K.G.: All-to-all personalized communication in multidimensional torus and mesh networks. IEEE Trans. Parallel Distrib. Syst. **12**(1) (January 2001) 38–59
31. Suh, Y.J., Yalamanchili, S.: All-to-all communication with minimum start-up costs in 2d/3d tori and meshes. IEEE Trans. Parallel Distrib. Syst. **9**(5) (May 1998) 442–458
32. Brandner, F., Schoeberl, M.: Static routing in symmetric real-time network-on-chips. In: International Conference on Real-Time and Network Systems. (2012) 61–70
33. Hansson, A., Goossens, K., Rădulescu, A.: A unified approach to constrained mapping and routing on network-on-chip architectures. In: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis. CODES+ISSS '05, New York, NY, USA, ACM (2005) 75–80
34. Stefan, R., Goossens, K.: An improved algorithm for slot selection in the thereal network-on-chip. In: International Workshop on Interconnection Network Architecture: On-Chip, Multi-Chip. (2011) 7–10