Highly Efficient and Predictable Group Communication over Multi-core NoCs [*]

Karthik Yagna, Frank Mueller
North Carolina State University
kyagna@ncsu.edu, mueller@cs.ncsu.edu

ABSTRACT

Massive multi-core embedded processors with network-on-chip (NoC) are becoming common in real-time systems. These architectures benefit real-time scheduling of tasks and provide higher processing capability due to abundance of cores. The core-to-core communication can be leveraged by adopting message passing to further increase system scalability. Despite these advantages, multicores pose predictability challenges.

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 real-time systems. Experimental results on the TilePro64 hardware platform show that our collectives can significantly reduce communication times by up to 95% for single packet messages. In addition, the primitives have significantly lower variance compared to prior work, thereby providing better real-time predictability.

1. INTRODUCTION

The future of computing is rapidly changing as multicore processors are becoming ubiquitous. Massive multi-core platforms with NoC architectures are being employed for real-time systems. These architectures provide a significant advancement due to an abundance of cores. This allows a large number of cooperating tasks to be scheduled together. These tasks can employ group communication via message passing over the NoC to achieve scalability and reduced latency.

However, poor group communication implementations can result in increased and highly variant latency due to NoC contention and results in loss of predictability. Consider the case where tasks on different cores are performing an all-to-all communication using message passing. One way to implement all-to-all is to have one task send its message to all other tasks, followed by the next one and so on. This implementation is not efficient and can be improved by allowing multiple partners to communicate in each round. Yet, such an optimization may lead to contention. 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 $4 \rightarrow 5$ results in a delay for one of these messages due to arbitration within the NoC hardware routers. Such situations can be avoided using intelligent scheduling of each round of message exchanges.

Additionally, implementations that do not leverage underlying NoC capabilities result in under utilization of the NoC hardware. Typically, NoC architectures provide multiple message queues and networks. On the TilePro64 [3], there are 4 distinct message queues and 2 distinct networks types available for users. One of them is called User Dynamic Network (UDN), and the other one is called Static Network (SN). UDN uses dynamic routing to forward messages from source core to destination. SN, on the other hand, uses statically configured routes to forward packets received on each link. SN is faster than UDN in terms of packet forwarding speed, but is difficult to program and has route setup overhead. Hence, UDN is used for all core-to-core communication purposes, leaving SN unused. Implementations which can leverage such unused hardware features can intelligently extract additional hardware performance.

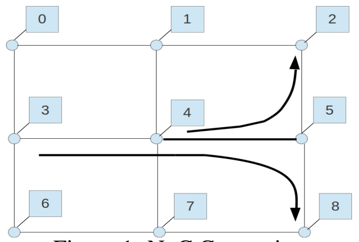

Figure 1: NoC Contention

In our implementation, we employ algorithms to reduce communication overheads and try to use available hardware features to provide better performance. We have currently implemented four commonly used group communications namely Barrier, Broadcast, Reduce and Alltoall [5]. Alltoallv and Allreduce can be built on top of these collectives. Efficient implementation of Alltoall is particularly challenging. Our implementation uses a bottom-up approach in which the communication proceeds from smaller segments to larger segments, but it does not require dividing the grid into smaller submeshes [6]. Other approaches require either dynamic route calculations of offline pre-calculations to store large routing tables [4]. In contrast, our implementation exploits simple pattern-based communication, common in MPI [5] 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, but is generic enough to be adopted to any 2D mesh based NoC architecture.

Experimental results on the TilePro hardware platform show that our implementation has lower latencies and lower timing variability than prior work. We used micro-benchmarks and compared the performance of our implementation against OperaMPI, an MPI implementation for the Tilera platform. Performance improvements of up to 95% are observed in communication for single packet messages with significantly high timing predictability.

2. DESIGN

Our work assumes a generic, generalized 2D mesh NoC switching architecture similar to existing fabricated designs with high core counts [2, 3, 7, 1]. Each core is composed of a compute core, network switch, and local caches. The network switch uses XY

[*]This work was supported in part by NSF grants 0905181 and 1239246.

dimension-ordered routing to forward messages.

2.1 NoC Architecture

NoC architectures use the network-on-chip to replace the conventional system bus or other topologies of connecting cores. This means that all memory, messaging, and IO communication occur over the NoC, often through physically separate networks to reduce contention. In this work, we focus on building group communication over the messaging network.

2.2 NoC Message Layer

Our implementation provides an MPI type message passing interface on top of NoC. This facilitates basic point-to-point communication used to support 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.

2.3 Group Communication Primitives

The key ideas behind our design of group communication primitives are (1) Reduce contention in NoC; (2) Exploit pattern-based communication to exchange messages concurrently; (3) Reduce the number of messages by aggregation; (4) Leverage hardware features to improve performance.

We have used different approaches for each group communication primitive to demonstrate the ways a NoC-based system can support timing reliability and reduced latency.

Alltoall

The Alltoall collective results in all the tasks in the group to exchange messages with each other. In our design, we exploit pattern-based communication to concurrently exchange messages between partners. The entire exchange is split into multiple rounds. In each round, a subset of tasks exchange messages using Manhattan-path (dimension-ordered) routing. The tasks in each round are scheduled in such a way that they do not result in link contention. In each round, the number of hops the message is forwarded to is incremented until all the tasks are covered.

Barrier

A barrier synchronizes a group of tasks. Each task, when reaching the barrier call, blocks until all tasks in the group reach the same barrier call. In order to provide scalable barriers, we designed tree-based barriers that distribute the work evenly among nodes. This also helps in minimizing the cycle differences upon barrier completion. Our design utilizes rooted k-ary trees to this end, where k is configurable. Typically k=3 provides optimal performance.

Broadcast

A broadcast sends a message from the process with rank "root" to all other processes in the group.Tree branches are mapped onto the NoC in a contention-free manner. In our design, we use the SN to implement broadcasts. We designed a tree-based broadcast rooted at the task performing the broadcast. The static route of each task is configured inside the broadcast primitive such that the message from the root flows to each leaf task. To minimize the overhead of route configuration, our design requires only a single route configuration per task, again using contention-free paths.

Reduce

This primitive applies a reduction operation on all tasks in the group and relays the result to one task. We designed our reduce collective similar to the barrier. The reduction operation is performed along the tree. Each task receives values from its children and performs a partial reduction. Tasks then send their partial result toward the root. The root will reduce partial results to obtain the final result.

Alltoallv and AllReduce

Alltoallv is designed as an extension to Alltoall. The AllReduce consists of a reduce followed by a broadcast. Our design follows the same idea.

3. IMPLEMENTATION

This section provides details on the implementation, called NoCMsg, of each group communication primitive. Our implementation of these collectives have an MPI-like API for easy usability.

We implemented the group communication on the Tilera TilePro64. Our implementation is generic and can be extended to any 2D mesh NoC architectures.

3.1 Alltoall and Alltoallv

Alltoall/Alltoallv are the most demanding collectives in terms of network contention, yet they allow flow-control elimination. Based on the particular internal send/receive orders in these collectives, it is possible to guarantee flow-control free communication for transfers between each pair of cores. Further optimization is provided by employing pattern-based communication, which allows several sets of tasks to exchange messages concurrently without contention. The entire exchange is split into multiple rounds.

The rounds are comprised of (1) direct (2) left and (3) right rounds. In direct rounds, each task sends messages only along a straight path to its partner task. 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.

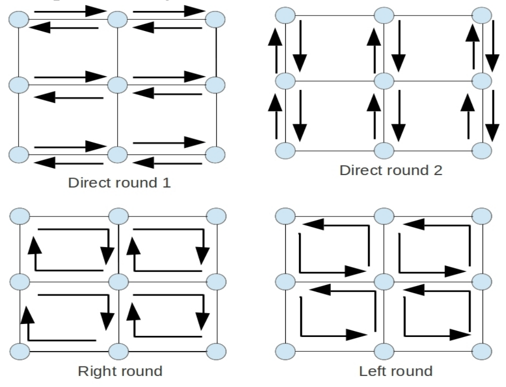

Figure 2: Alltoall Rounds

In each round, the number of hops the message is forwarded is incremented until all the tasks are covered. To begin, each task starts the direct round with one hop. In other words, they exchange messages with their immediate neighbors. Once the round is over, they increment their hop count and exchange messages with a neighbor 2 hops away. This is repeated until the entire width of the grid is covered. After an exchange along the X direction is done, the tasks start direct round 2 and send messages along the Y direction in a similar fashion. This set of rounds is followed by left and right rounds, thereby covering the entire grid. The logic of the algorithm is depicted in the Figure 3.

3.2 Broadcast

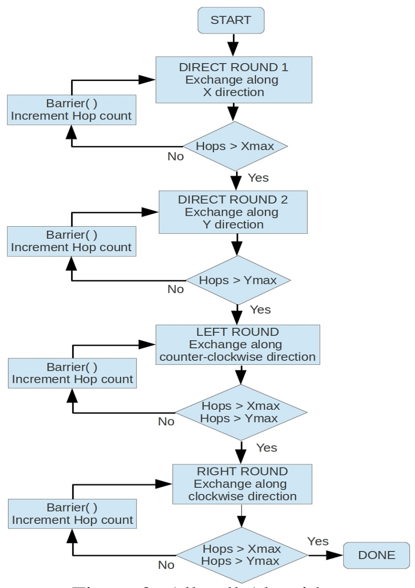
START
DIRECT ROUND 1 Exchange along X direction
Barrier( ) Increment Hop count
Hops > Xmax
No
Yes
DIRECT ROUND 2 Exchange along Y direction
Barrier( ) Increment Hop count
Hops > Ymax
No
Yes
LEFT ROUND Exchange along counter-clockwise direction
Barrier( ) Increment Hop count
Hops > Xmax Hops > Ymax
No
Yes
RIGHT ROUND Exchange along clockwise direction
Barrier( ) Increment Hop count
Hops > Xmax Hops > Ymax
No
Yes
DONE

Figure 3: Alltoall Algorithm

Our Broadcast implementation uses the SN of the TilePro64. The SN is difficult to program and suffers from route setup overhead. However, message forwarding incurs zero overhead (due to a static route setup). 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. 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.

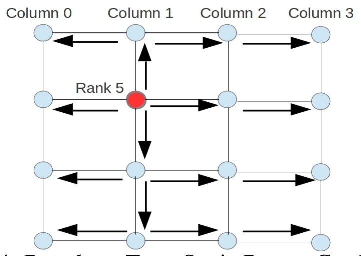

Figure 4: Broadcast Tree: Static Routes Configuration

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.

3.3 Barriers

We utilize 3-ary tree-based barriers that distributes the work evenly among nodes to minimize the cycle differences upon barrier completion. The root of this tree is placed in the center of the NoCMsg grid to minimize latency (hops). The process of synchronization involves the children notifying their parents when they have entered the barrier, up to the root. Once the root has received notifications from all children, it broadcasts a notification back down the tree by replying to its children and exits, as do the children. Flow control is not needed in the barrier as the prerequisite of entering into the barrier is that all outstanding sends/receives on the local core have completed. The synchronization packet is small enough to fit into the output queue, *i.e.*, the core can drop an entire synchronization packet into its output queue. It can subsequently begin 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.

3.4 Reduce and AllReduce

We designed our Reduce collective 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. Our current implementation uses a 3-ary tree rooted at the root of the reduce call.

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

4. EXPERIMENTAL RESULTS

We evaluated our group communication using micro benchmarks on the Tilera TilePro64. We compare the performance of our implementation against OperaMPI, an MPI library specific to the Tilera platform. Each experiment were conducted multiple times to get accurate timing results and variance.

4.1 Microbenchmarks

The micro-benchmarks have a single call to the group communication. In each experiment, we determined the time elapsed in completing the group communication. The basic template of micro-benchmark is as shown :

```
NoCMsg_Init(int argc, char **argv)
.........
NoCMsg_Barrier(NoCMsg_Comm comm)
NoCMsg_Timer_start(int timer_num)
NoCMsg_Bcast(void* buffer, int count,
             NoCMsg_Type datatype, int root,
             NoCMsg_Comm comm)
NoCMsg_Timer_stop(int timer_num)
..........
```

The summarized timing results are shown in Table 1 and 2. The plots in Figure 5,7,8 and 6 are timing results for alltoall, barrier, broadcast and reduce micro-benchmark respectively.

Table 1: OperaMPI Execution Times [μsec]

Num tasks	4	9	16	25	36	49
Alltoall	69.57	146.29	250	483.71	759.1	2027.4
Barrier	84.57	174.86	398.57	478.29	679.1	1003
Broadcast	76.29	200.14	337.85	657.43	1026.5	1380.4
Reduce	112.86	232.86	477.71	657.4	955.8	1269.5

Table 2: NoCMsg Execution Times [μsec]

Num tasks	4	9	16	25	36	49
Alltoall	32.28	38.86	114.71	221.29	428.43	761.71
Barrier	3.43	4.43	5.71	10.29	14.17	17.29
Broadcast	3	4	4.43	5.57	7.57	9.14
Reduce	12.57	39.43	27.43	46.83	68.17	87.71

The experimental results follow a similar trend. With increase in number of tasks, the execution time of group communication increases. In case of Opera, the increase in runtime is significant for larger number of task. In comparison, our implementation is highly efficient and increase in runtime is gradual. Our implementation

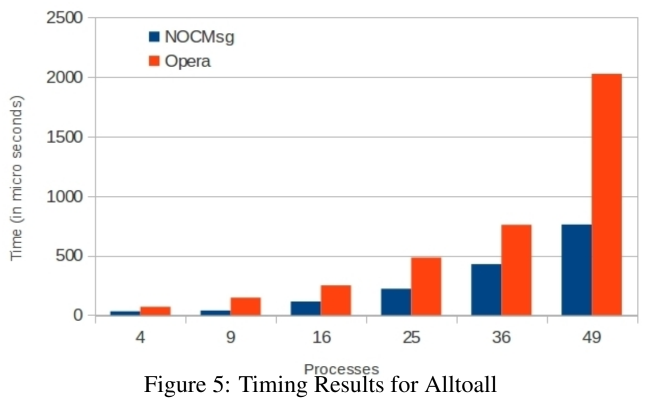

Figure 5: Timing Results for Alltoall

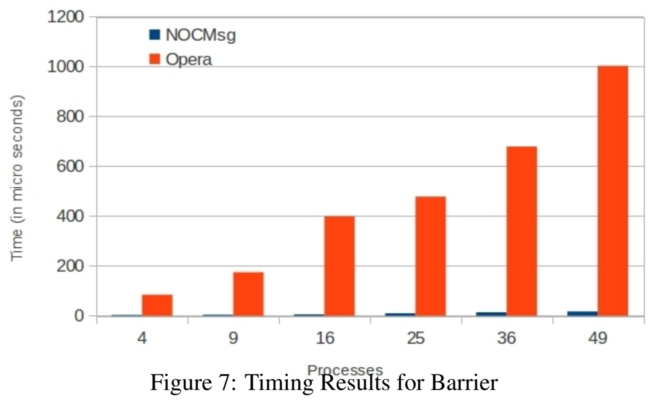

Figure 7: Timing Results for Barrier

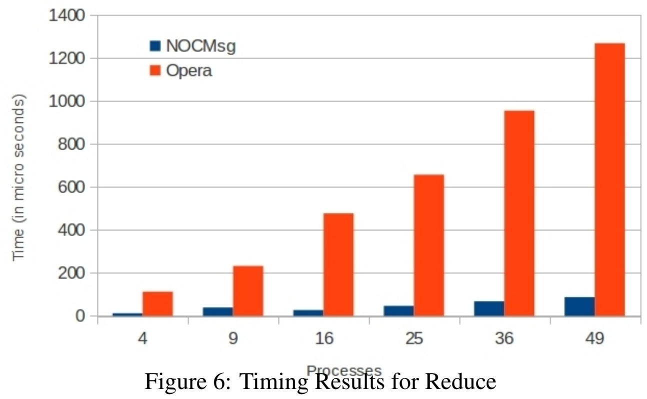

Figure 6: Timing Results for Reduce

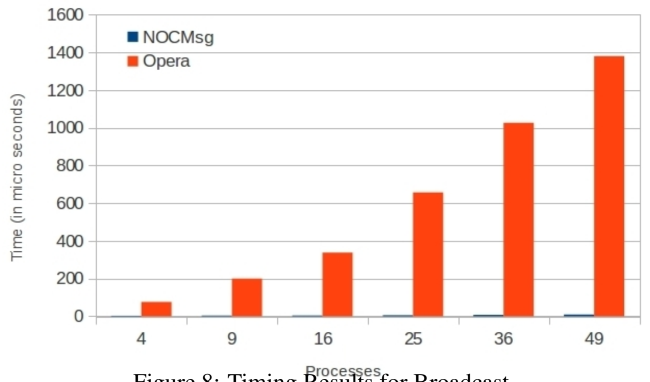

Figure 8: Timing Results for Broadcast

significantly reduced communication time by up to 95% for single packet messages.

The execution time variance for different micro-benchmark in case of OperaMPI and NocMsg are shown in Table 4 and 3. The variance of timing results for our implementation is several order lower than that of Opera. The difference is significantly large for Alltoall, Broadcast and Barrier case. The lower variance of our implementation results in better real-time predictability making our implementation ideal for real-time applications.

Table 3: NoCMsg Execution Time Variance

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	11.95	311.39	1.10	183.13	418.13	21.34

Table 4: OperaMPI Execution Time Variance

Num tasks	4	9	16	25	36	49
Alltoall	2.81	983.9	18.2	2276.8	133329.8	622903
Barrier	750.2	302.9	29384.5	1838.2	2910.7	32117
Broadcast	7.3	56.9	259.2	4540.8	3003.7	3869
Reduce	25.2	154.1	19422	4540	12560.9	3725

5. CONCLUSION

We have designed a set of efficient and predictable group communication primitives using message passing utilizing NoC architectures to improve performance and timing predictability specifically design for high-confidence real-time systems. Our implementation of the most commonly used collectives reduced the communication time over a reference MPI implementation by up to 95% for single packet messages. 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 real-time applications.

6. REFERENCES

[1] Single-chip cloud computer. blogs.intel.com/research/2009/12/sccloudcomp.php.

[2] Tera-scale research prototype: Connecting 80 simple sores on a single test chip. ftp://download.intel.com/research/platform/terascale/tera-scaleresearchprototypebackgrounder.pdf.

[3] Tilera processor family. www.tilera.com/products/-processors.php.

[4] Florian Brandner and Martin Schoeberl. Static routing in symmetric real-time network-on-chips. In *Proceedings of the 20th International Conference on Real-Time and Network Systems*, RTNS '12, pages 61–70, New York, NY, USA, 2012. ACM.

[5] 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 *European PVM/MPI Users' Group Meeting*, pages 97–104, September 2004.

[6] Young-Joo Suh and Sudhakar Yalamanchili. All-to-all communication with minimum start-up costs in 2d/3d tori and meshes. *IEEE Trans. Parallel Distrib. Syst.*, 9(5):442–458, May 1998.

[7] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown III, and Anant Agarwal. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27:15–31, 2007.