Performance Analysis of Broadcasting Algorithms on the Intel Single-Chip Cloud Computer

John Matienzo, Natalie Enright Jerger
Department of Electrical and Computer Engineering
University of Toronto
Toronto, Ontario, Canada
{matienz1, enright}@eecg.toronto.edu

***Abstract*—Efficient broadcasting is essential for good performance on distributed or multiprocessor systems. Broadcasts are commonly used to implement message passing synchronization primitives, such as barriers, and also appear frequently in the set up stage of scientific applications. The Intel Single-Chip Cloud Computer (SCC), an experimental processor, uses synchronous message passing to facilitate communication between its 48 cores. RCCE, the SCC's message passing library, implements broadcasting in a traditional way: sending $n-1$ unicast messages, where n is the number of cores participating in the broadcast. This implementation can hinder performance as the number of cores participating in the broadcast increases and if the data being sent to each core is large. Also in the RCCE implementation, the broadcasting core is blocked from doing any useful work until all cores receive the broadcast.**

This paper explores several broadcasting schemes that take advantage of the resources of the SCC and the RCCE library. For example, we explore a scheme that propagates a broadcast to multiple cores in parallel and a scheme that parallelizes off-chip memory accesses which would otherwise need to be done sequentially. Our best broadcast scheme achieves a 35× speedup over the RCCE implementation. We also demonstrate that our improved broadcasting substantially reduces the time spent on communication in some benchmarks. While the broadcast schemes presented in this paper are implemented specifically for the SCC, they provide insight into the more general problem of broadcast communication and could be adapted to other types of distributed and multiprocessor systems.

I. Introduction

Throughout the last decade, the computing industry has seen an increasing number of cores integrated on a chip thanks to Moore's Law. Recently, core counts have numbered in the dozens [1]–[3] and we are rapidly approaching systems with hundreds of cores on a single die. For example, the Single-Chip Cloud Computer (SCC) experimental processor [1] is a 48-core 'concept vehicle' created by Intel Labs as a platform for many-core software research. Systems such as this allow researchers to explore application development and better understand hardware and software bottlenecks that could impact the performance of future many-core systems. The SCC has 48 cores, arranged as 24 tiles connected via a 2D mesh on-chip network (OCN). Notably, the SCC does not have hardware support for cache coherence. Like many distributed systems, the Intel SCC uses message passing as its primary programming paradigm.

Many-core platforms such as the SCC promise tremendous compute power that can be leveraged by splitting computation across multiple processors. Ideally, this division would result in speedup equivalent to the number of nodes in the system. However, as the number of cores scale, the performance of the raw compute can be overshadowed by overheads such as inter-core communication. In addition to the already non-trivial task of writing correct parallel applications, programmers must now focus on optimizing and/or minimizing communication to ensure acceptable program run time.

The two prevalent programming paradigms for multiprocessor systems are shared memory and message passing. As scalable cache coherence remains an open problem [1], [4]–[6], it is worthwhile to consider the implementation of alternatives. The SCC uses message passing to facilitate communication between cores. The library provided with the SCC is called RCCE; RCCE implements a subset of MPI features [7]. We focus on broadcasting as it can represent a significant bottleneck in application performance; for example, once all cores have reached a barrier, we would want a very fast broadcast to enable all cores to move past the barrier and resume useful work. Although RCCE provides programmers with straightforward methods to communicate among cores, the current broadcasting scheme implemented by RCCE is slow. It uses $n-1$ unicasts (where n is the number of cores) to replicate a message to all cores [8]. This broadcasting scheme does not scale well as the time for a message to reach all cores increases linearly as the number of broadcast participants increase. Thus, we present and evaluate several new broadcasting protocols, each with two goals: (1) to provide better performance than the current RCCE broadcast, and (2) to scale well if the number of cores participating in the broadcast increases.

The main strategy for four of our implemented broadcasts is to utilize cores that have already received the broadcast. This allows the original broadcasting core to be responsible for only sending the message to a few processors (as opposed to all of them). The cores that have received the message are then responsible for forwarding the message to the other processors, which happens in parallel. Sections III-B to III-E describe these broadcasting algorithms in more detail. The remaining broadcasting protocols presented utilize concurrent accesses to a specific memory location (the memory location that contains the message) as the main implementation strategy. The best broadcast implemented achieves an overall speedup of 35× over the RCCE broadcast for large messages.

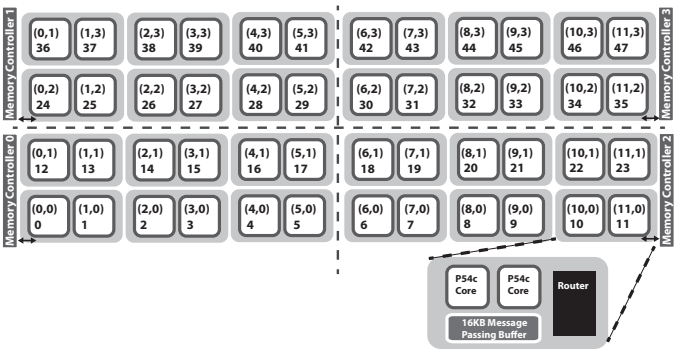


Fig. 1. Intel SCC Tile Layout.

II. Background

This section provides a high-level overview of the Intel SCC architecture, relevant details on Intel's Message Passing Interface (MPI) library, RCCE, and gives an overview of how RCCE handles messages.

A. Intel SCC

The Intel SCC's 48-core architecture is arranged in a 24-tile mesh, as depicted in Figure 1. Each tile (shaded in grey) contains two P54c cores, with 16KB of L1 instruction and data cache, 256KB of L2 cache per core, special on-chip memory known as the message passing buffer (MPB), and a router. The MPB on each tile is 16KB, for a total of 384KB of on-chip memory on the SCC [1]. There are four memory controllers that the SCC uses to access off-chip memory. Specifically, tiles are divided into four quadrants, and each quadrant has one designated tile that communicates with the memory controller.

B. RCCE

There are several message passing libraries implemented for the Intel SCC. One such library provided by Intel is RCCE [7]. RCCE is a synchronous message passing library that contains most MPI functionality. To facilitate fast communication of messages between cores, RCCE has cores communicate with each other by writing to and reading from the MPB. Messages are sent/received using a "pull" based method [9].

When a core wants to send a message to another core:

1) The message is copied from the sending core′s private off-chip memory to its portion of the MPB
2) The sending core notifies the receiving core of the message by setting a flag that is local to the receiving core (receiving core waits until the flag is set)
3) Receiving core copies message from the sending the core's MPB to its own private off-chip memory
4) Receiving core notifies the sending core once copying is complete by setting a flag that is local to the sending core

If the message is too big to fit in the sending core's MPB, the above process is repeated until the whole message is sent.

RCCE provides a simple MPI interface and a more advanced interface for programmers to use[1]. One of the main differences is that the advanced interface exposes the MPB to the user while the simple interface only exposes the traditional sending/receiving MPI functions. In the simple interface, the library takes care of the intricacies of the MPB. This paper uses the advanced interface, as some manipulation of the message passing buffer is needed for certain broadcasts.

III. Broadcasting Algorithms

This section describes: (1) the current broadcast implementation in RCCE, and (2) the new broadcasting algorithms that have been implemented using RCCE.

A. RCCE Broadcast

The broadcasting algorithm implemented in RCCE simply sends a unicast message to each core participating in the broadcast. This is highly inefficient, especially for synchronous message passing. Since the sending core must block, the last core will have to wait $(n-1) \times T_{Latency}$ to receive the broadcasted message (where n is the number of cores in the broadcast and $T_{Latency}$ is the average time to send a message through the on-chip network to a single core).

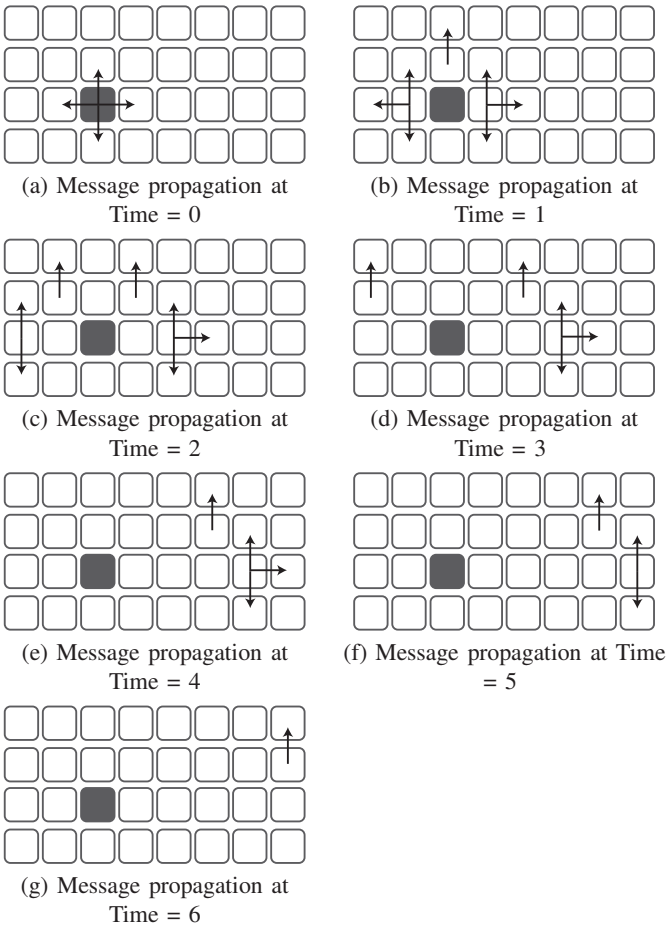
B. Parallel Broadcast

Our first implementation, the `parallel broadcast` algorithm takes advantage of cores that have already received the broadcasted message. This allows the broadcasted message to propagate to other cores in parallel. Specifically, the `parallel broadcast` scheme has the sending core broadcast the message to adjacent cores (see Figure 2 (a)). Once adjacent cores receive the message, each adjacent core then sends the message to cores that are adjacent to it (see Figure 2 (b)). Adjacent cores are defined as those cores located north, south, east and west of the sending core. Messages are sent through the network in an XY fashion; for example, a message received from an adjacent core to the south will forward the message north but not east and west. This ensures that each core only receives one copy of the broadcast. This forwarding process repeats until all cores receive the message.

C. Optimized Parallel Broadcast

The `optimized parallel broadcast` is similar to the parallel broadcast, except that it takes cores located at the edge of the mesh into special consideration. These edge cores have fewer adjacent cores to send their broadcast to which results in less parallelism. As a result, it takes fewer parallel hops to propagate a broadcast that originates in the center of the mesh compared to a broadcast that originate from a core located in the corner of the mesh. For example, it takes 8 parallel hops if the message is broadcast from a center core compared to 15 parallel hops if the message is sent from the left corner core. To address this discrepancy, if an edge core wishes to send a broadcast, the `optimized parallel broadcast` has that core first send the message to a center core (see Figure 3). Once the center core receives the message,

[1]These interfaces are referred to as non-gory and gory respectively in the SCC documentation.



(a) Message propagation at Time = 0
(b) Message propagation at Time = 1
(c) Message propagation at Time = 2
(d) Message propagation at Time = 3
(e) Message propagation at Time = 4
(f) Message propagation at Time = 5
(g) Message propagation at Time = 6

Fig. 2. Parallel Broadcast Propagation. The broadcast source node is shown in grey. The number of cores shown is simply for illustration purposes.

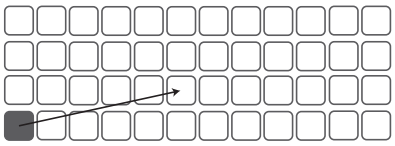
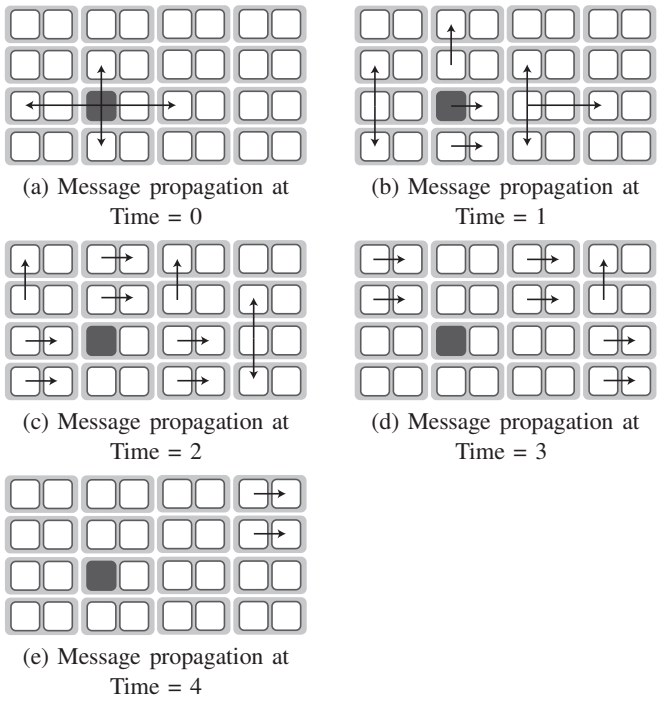


Fig. 3. Optimized Parallel Broadcast sending message to a more efficient center core.

it is then responsible for initiating the parallel broadcast. The location of this center core is determined based on the set of cores participating in the broadcast.

D. Tiled Parallel Broadcast

Each tile on the SCC has two cores and a MPB. The MPB in each tile is 16KB. RCCE divides the MPB equally among the two cores in each tile (8KB per core). Normally, when messages are sent, 4KB of the sending core's 8KB MPB are used for the message (the other 4KB are reserved for sending and receiving synchronization flags). Instead of only using 4KB for sending a broadcast message, the `tiled parallel broadcast` implementation allows the sending core to use 8KB. It uses its own 4KB and borrows 4KB from the adjacent core's MPB. The main advantage for utilizing more space in the MPB is that there is less blocking/stalling



(a) Message propagation at Time = 0
(b) Message propagation at Time = 1
(c) Message propagation at Time = 2
(d) Message propagation at Time = 3
(e) Message propagation at Time = 4

Fig. 4. Tiled Parallel Broadcast Propagation. Tiles are shown in grey. Once one core in a tile receives the broadcast, it first forwards the message to the adjacent tile and then sends the message to the adjacent core within its own tile.

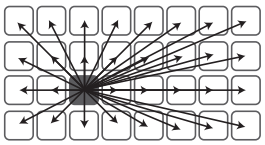


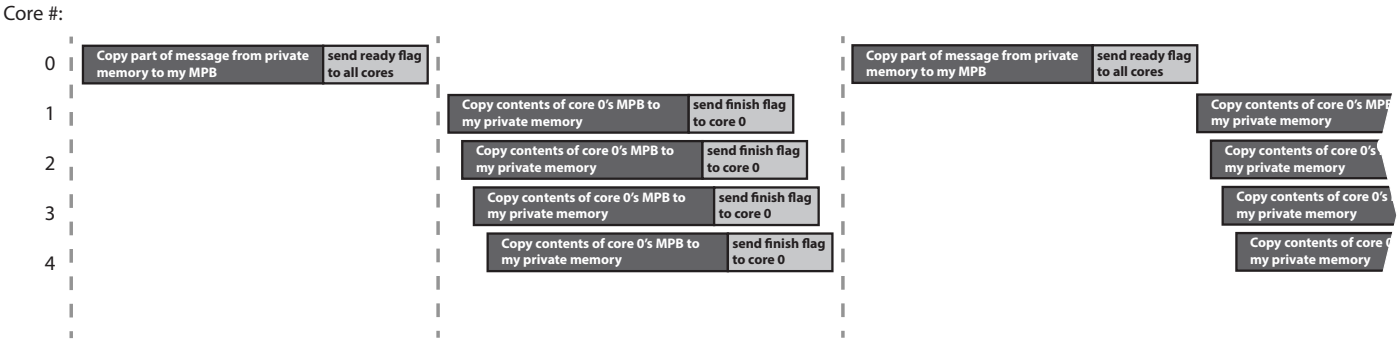
Fig. 5. MPB Broadcast Propagation.

when sending large messages, compared to the parallel and optimized parallel broadcasts.

The `tiled parallel broadcast` has a similar broadcast pattern to the parallel broadcast pattern, except that messages are sent to adjacent tiles, instead of adjacent cores. Specifically, the left core of each tile sends the message only to the left core of each adjacent tile. Once a tile has broadcast the message to adjacent tiles, it then sends the message to the other core in its tile (in this case the right core). The `tiled parallel broadcast` pattern is depicted in Figure 4. The `tiled parallel broadcast` increases the parallelism of the broadcast (compared to the parallel broadcast) and reduces overall mesh traffic by leveraging intra-tile communication.

E. Optimized Tiled Parallel Broadcast

The `optimized tiled parallel broadcast` is similar to the `tiled parallel broadcast`. However, like the `optimized parallel broadcast`, it gives special consideration to edge tiles by forcing them to first send their message to a center tile in order to increase the amount of parallelism.



(a) Sending core copies message from its private memory to its MPB.

(b) Receiving cores copy message from sending core's MPB to their private memories.

(c) Process repeats until whole message is received.

Fig. 6. MPB Broadcast Timing Diagram.

F. Message Passing Buffer (MPB) Broadcast

Recently, Chandramowlishwaran et al. [10] proposed a broadcasting optimization that we refer to as the `MPB broadcast`. The MPB broadcasting scheme has all receiving cores read the sending core's MPB at the same time, as depicted in Figure 5. A timing diagram is shown in Figure 6. Figure 6(a) shows the sending core copying the message from its private memory to the MPB. It then signals the receiving cores that there is a message to be received. Figure 6(b) depicts the receiving cores copying the message from the sending core's MPB to their own private memories. We have reimplemented this MPB broadcast as it provides an interesting point of comparison. Furthermore, we propose two broadcasting algorithms that leverage similar insights as the `MPB broadcast` and provide further optimization.

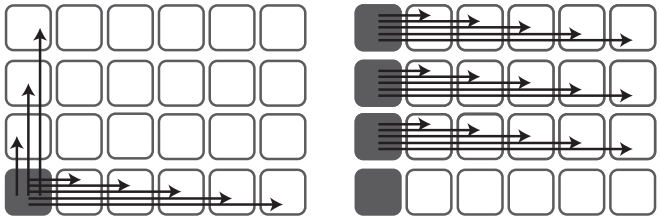
G. Off-Chip Broadcast

The `off-chip broadcast` is similar to the MPB broadcast; again each core reads the broadcast message at the same time. However, instead of the sending core copying the message to its MPB, it copies the entire message from its private memory to shared off-chip memory. Receiving cores then copy the entire message from off-chip shared memory to their own private memory. The main strategic advantage of this broadcast is that there is no need for extra handshaking/blocking for large messages since the messages do not need to be split into 4KB chunks to accommodate the small MPB size.

H. Modified MPB Broadcast

The `modified MPB broadcast` or `ModMBP` is similar to the `MPB broadcast`, but it has two distinguishing features:

1) Temporary broadcasting cores are created to off-load network traffic from the broadcasting core
2) Receiving cores' MPBs are utilized (normally only the sending core's MPB is utilized) to allow parallelization of off-chip memory writes (done by the receiving cores) with off-chip memory reads (done by the original sending core)



(a) Message propagation at Time = 0

(b) Message propagation at Time = 1

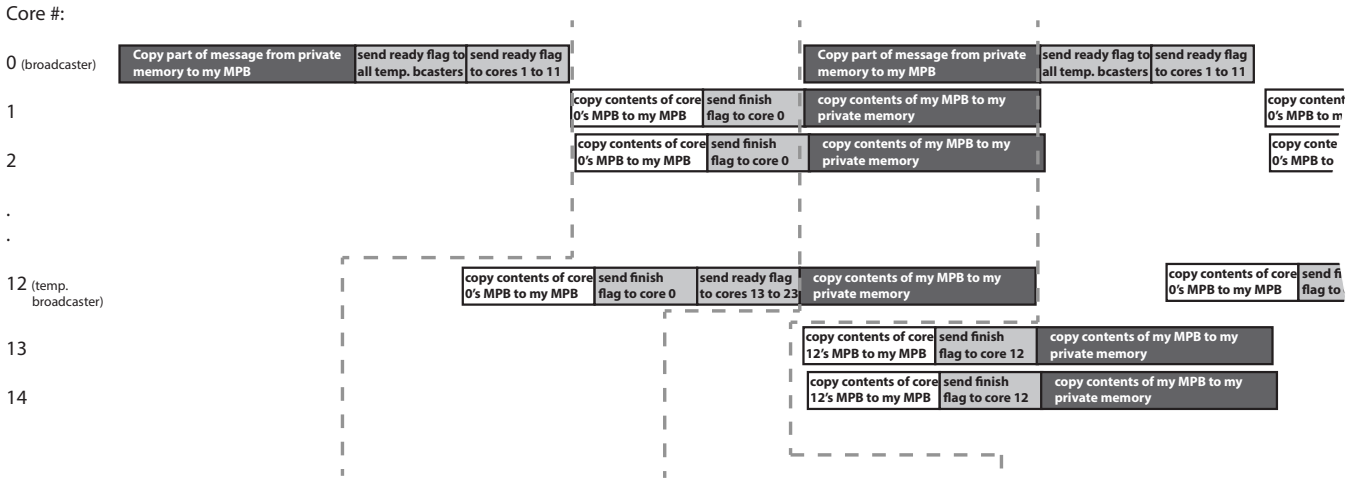
Fig. 7. ModMPB Broadcast Propagation.

The total number of broadcasting cores (including the original) is $\lceil n/12 \rceil$, where n is the number of cores in the broadcast. The number of broadcasters is optimized for the case where all 48 cores on the SCC are enabled, which translates to 4 broadcasting cores, one for each row of cores. These broadcasting cores are located near memory controllers 0 and 1, or the leftmost core of each row. Experiments showed that the placement of the broadcasting cores did not have any impact on the performance of the broadcast. Increasing the number of broadcasting cores past 4 starts to negatively impact performance. Figure 7(a) shows the original broadcasting core sending the message to designated temporary broadcasting cores and to a subset of cores. Figure 7(b) shows the temporary cores sending the messages to the subset of cores that they are responsible for.

Figure 8(c) illustrates how this broadcast is able to hide off-chip memory writes done by the receiving core with off-chip memory reads done by the sending core. Essentially, because the receiving cores copy the message to their MPB first before copying data to private memory, the sending core can start copying new parts of the message to its MPB once the receiving cores' have finished copying the message to their MPB. This is a key feature in this broadcast implementation.

IV. Evaluation

We have implemented all of the broadcasting schemes described in the previous section on the Intel SCC. We use the default configuration of cores running at 533MHz and



(a) Sending core copies message from its private memory to its MPB and notifies temp. cores (plus a subset of other receiving cores) that a message is ready

(b) Temporary broadcasting cores plus a subset of cores copy message from sending core's MPB to their MPBs

(c) Off-chip memory writes are parallelized with off-chip memory read

(d) Process repeats until whole message is received

Fig. 8. ModMPB Broadcast Timing Diagram

off-chip memory running at 800MHz. RCCE version 1.4.1.3 was used to implement, compile, and run our benchmarks. We use four micro-benchmarks to assess the performance (latency) of the broadcasts. We focus much of our analysis on micro-benchmarks as they enable us to tease out subtle differences between the broadcasting schemes. These micro-benchmarks are presented in Table I. However, understanding the impact of broadcast latency on real applications is also important. We present results for three benchmarks: matrix multiply, n-body and bucket sort. For these benchmarks, we compare RCCE against the best performing broadcast implementation for large messages as determined by the micro-benchmarks. We use execution time as the metric for comparison. In addition, we also compare average power for these two implementations.

TABLE I. DESCRIPTION OF MICRO-BENCHMARKS

Benchmark	Description
Message Size	Vary message size from 1B to 1MB
Message Source	Vary the location of the core sending the broadcast, with 1MB messages
Destinations	Vary the number of receiving cores, with 1 MB messages
Background traffic	Inject additional unicast traffic into the network

A. Impact of Message Size

Figure 9 shows the latency results as we increase the size of the broadcast message. The sending core for this benchmark is the left corner core (core 0 in Figure 1) and the number of participants in the broadcast is 48. The `ModMPB` implementation achieves the lowest latency with larger message sizes. For 1MB messages, `ModMPB` achieves a 35× speedup compared to RCCE. The `MPB broadcast` achieves a speedup of 32× for 1MB messages (compared to RCCE). The `off-chip broadcast` does well for message sizes smaller than 64 bytes, but performs poorly for larger messages. Based on our experiments, we speculate that this behaviour is caused by contention of the sending core's MPB. All the other broadcasts

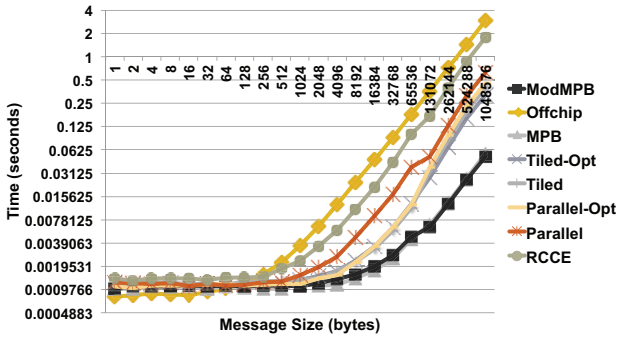


Fig. 9. Microbenchmark results: Broadcast latency when varying message size from 1B to 1MB.

except the off-chip one use the sender's MPB to get the message and also use the same MPB for synchronization flags. The off-chip broadcast uses the on-chip MPB for synchronization, but uses off-chip memory for the broadcasted message; thus, there are fewer accesses to the sender's MPB.

B. Impact of Message Source Location

Figure 10 shows the latency results for each broadcasting scheme when using a different core to initiate the broadcast. Physical placement in the network can have an impact on both latency and congestion [11]; a broadcasting core can produce a hot-spot in the network. Therefore, it is interesting to evaluate the impact of source placement. For this test, the message size is 1MB and the number of cores participating in the broadcast is 48.

The results for this micro-benchmark shows that most broadcasts achieve similar latency regardless of source location. The only exception is the `Tiled Parallel` and `Parallel` broadcasts. For these broadcasts, we see that cores

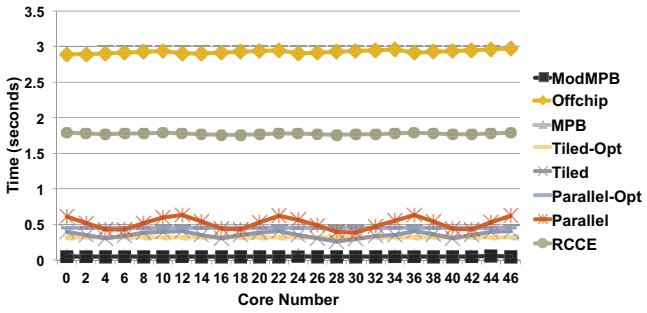


Fig. 10. Microbenchmark results: Broadcast latency when varying broadcasting source core.

with higher latency are those not located in the center of the mesh. This is the problem that the *optimized* versions fix. By redirecting edge broadcast to the center of the mesh, the *optimized* versions have fewer parallel hops leading to lower latency. Our results show that a well-designed broadcast can be placement agnostic which will lead to more predictable performance in these systems.

C. Impact of the Number of Participating Cores

Figure 11 shows the latency results when each broadcast has a varying number of participants. The message size is 1MB and core 0 is the source of the broadcast. This microbenchmark indicates that neither RCCE nor the `off-chip` broadcast scale well as the number of cores increases. In contrast, the `ModMPB` and `MPB` broadcasts are fairly stable. Both these broadcasts exhibit small fluctuations between values of 0.05 and 0.06 seconds. Their stable latencies indicate that they will scale well to even larger systems.

Figure 11 reveals a peculiar pattern for the `optimized tiled parallel` and `optimized parallel` broadcasts. These broadcasts see an increase in latency as the number of cores increases and then reveal periodic decreases in latency. This phenomenon is attributed to how these broadcasts choose the "center" core before parallelizing the broadcast. When the optimized broadcast selects a center core, it does so by determining the maximum perfect rectangle in the system. An example for an 18-core broadcast is shown in Figure 12. The maximum perfect rectangle is shaded in grey in Figure 12(a). Cores in this rectangle receive the broadcast

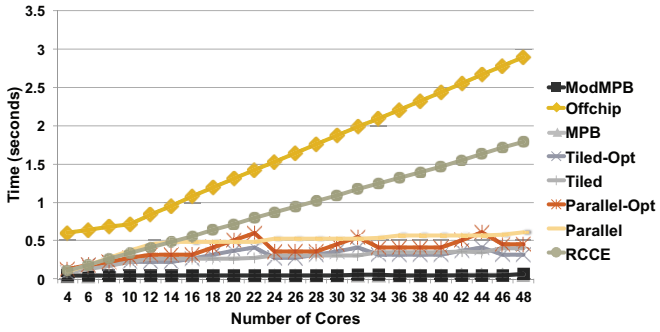
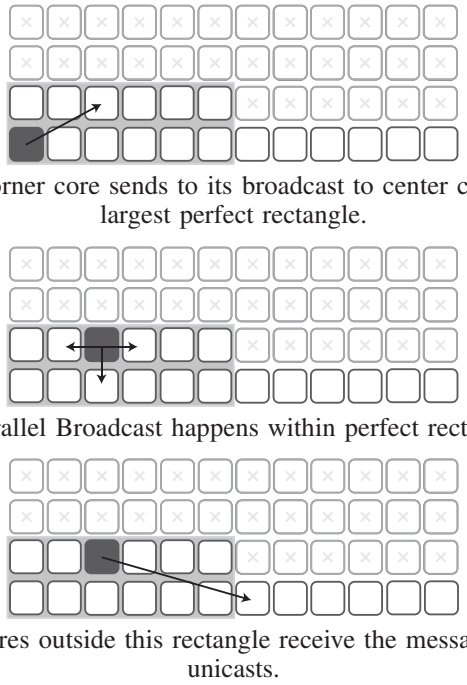


Fig. 11. Microbenchmark results: Broadcast latency when varying number of participating cores.



(a) Corner core sends to its broadcast to center core of largest perfect rectangle.

(b) Parallel Broadcast happens within perfect rectangle.

(c) Cores outside this rectangle receive the message via unicasts.

Fig. 12. Optimized Parallel Broadcast with 18 active cores. The maximum perfect rectangle is shown in light grey. Cores not participating in the broadcast are marked with a "x".

via the parallel method (Figure 12(b)). However, any cores outside the rectangle receive the message via unicasts from the broadcasting core (Figure 12(c)).

D. Impact of Background Traffic on Broadcasts

Figure 13 shows the effect of background traffic on each broadcasting scheme. For this test, each core writes 1MB of data to off-chip memory at an interval of x microseconds. Core 18 is chosen to be the broadcaster since the router associated with its tile's is subjected to the smallest amount of on-chip network traffic when the traffic pattern is dominated by off-chip requests. Core 18 (and other center cores) will experience less interference because the SCC is divided into four quadrants; the cores of each quadrant access the off-chip memory controller attached to that quadrant [12].

Intuitively, the broadcasts should perform better when there

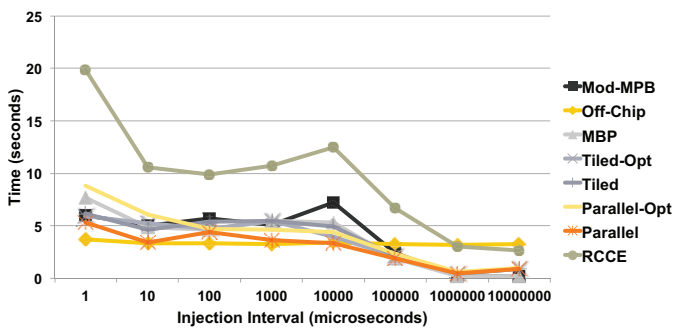


Fig. 13. Microbenchmark results: Impact on broadcast latency of background traffic in network.

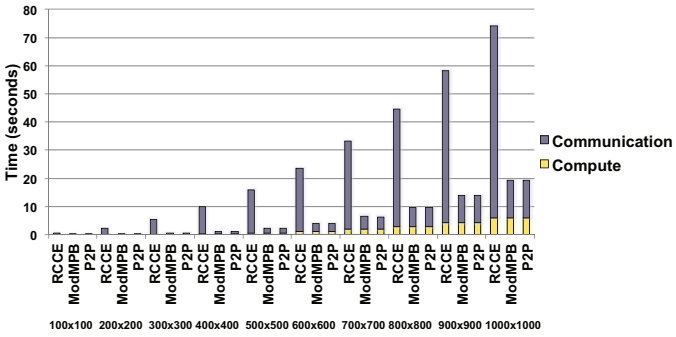


Fig. 14. Matrix multiply execution time broken down into time spent on compute and communication for various input matrix sizes.

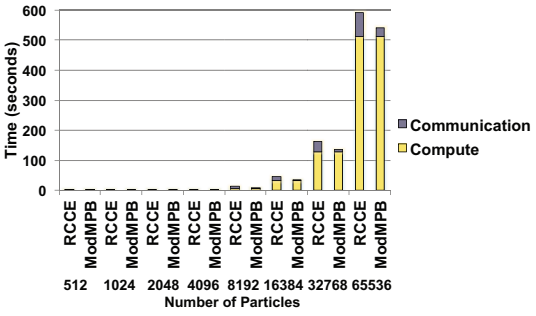


Fig. 15. N-Body execution time broken down into time spent on compute and communication for varying numbers of particles (each particle is 32 bytes).

is less background traffic (a longer interval period). This assumption is validated looking at the right-hand side of Figure 13. Interestingly, the `off-chip broadcast` is almost impervious to background traffic. In addition, when studying the effects of the broadcasts on the latency of the background traffic (not shown), results revealed that the `off-chip broadcast` affected background traffic the most. The stable performance of the off-chip broadcast comes at the cost of increased latency for background traffic, which is caused by the significant pressure being placed on the memory controllers.

Based on these evaluations, we determine that the `ModMPB` broadcast implementation has superior performance. In the following subsections, we will compare the performance of `ModMPB` to RCCE for three applications.

E. Matrix Multiply

The integer matrix multiply benchmark is implemented using the following algorithm:

1) Matrix A and Matrix B are broadcast to all cores by the master core
2) Each core calculates 1/48 rows of the resultant matrix (the master core is responsible for any left over rows)
3) Each core sends their results back to the master core

Figure 14 shows the execution time of calculating the product of two matrices on the SCC for various matrix sizes, using the ModMPB, and RCCE broadcasts. A point-to-point (P2P) implementation modifies step 1 of the above algorithm slightly. Matrix B is still broadcast to all cores using ModMPB, but for Matrix A, only elements needed by a remote core are sent to that core.

For this benchmark, the bottleneck is communication. Between the ModMPB, RCCE and P2P implementations, compute time remains the same for a given matrix size. However, due to communication overheads, matrix multiply using the RCCE broadcast takes up to 74 seconds to multiply two 1000x1000 matrices, while ModMPB takes only 20 seconds. For the P2P implementation, communication latency slightly outperforms ModMPB. For the 1000x1000 product matrix, P2P communication latency is better than ModMPB by 0.04 seconds. ModMPB's highly optimized design results in performance that is competitive with the P2P implementation.

F. N-Body Problem

The N-Body benchmark is implemented in a brute-force fashion using the following algorithm:

1) Master broadcasts all particle data to other cores
2) Each core performs calculations on a subset of particle data
3) Each core sends their results back to the master

Figure 15 shows the execution time of running the N-Body problem for 10 iterations. Unlike for matrix multiply, the N-Body problem is bottlenecked by compute, not communication (which makes it an excellent algorithm for the SCC). However, communication latency is still non-trivial. For 65K particles, there is a difference of $\sim$50 seconds between the RCCE and ModMPB implementations, which is attributed to communication latency.

G. Bucket Sort

When sending data to a subset of cores, using a broadcast can sometimes be simpler than using several point-to-point messages because the programmer does not need to determine which cores are the recipients. But can broadcasting provide competitive performance? We use the bucket sort benchmark to answer this question.

Unlike the brute-force version of the N-Body problem, bucket sort does not necessarily need the master core to disseminate all of the data to each core. However, calculating what data each peer needs is complicated and redundant. So, instead of the master core performing those calculations, the programmer could simply send all data to all peers, allowing the destination peer to determine whether the data it received is useful. For the bucket sort algorithms presented, there are 48 buckets and each core is responsible for a certain bucket. Each bucket is representative of a predetermined range, e.g. a bucket could hold numbers ranging from 256-511. Table II presents two possible algorithms for bucket sort.

In Figure 16, we compare the latency of point-to-point communication (Algorithm 1: Step 1) to initially broadcasting all data (Algorithm 2: Step 1). For Algorithm 2, we compare both RCCE and `ModMPB`. Using RCCE to broadcast the data severely limits scalability; for even a small number of elements, the programmer would be better off implementing Algorithm 1. However, with `ModMPB`, Algorithm 2 becomes a feasible option. Thus, with `ModMPB`, if using a broadcast

Algorithm 1 (No broadcast required): 1. Master core sends a subset of unsorted numbers to each core 2. Cores place the numbers given to them in their respective buckets 3. Each core sends the buckets to the respective core that owns that bucket 4. Once a core receives all of its buckets from the other cores, it combines the buckets and sorts all the numbers within the combined bucket 5. All cores send their sorted buckets to the master core
Algorithm 2: 1. Master core broadcasts all unsorted numbers to all cores 2. Cores take ownership of 1/48 of the unsorted array and places those numbers into buckets 3. Each core sends the buckets to the respective core that owns that bucket 4. Once a core receives all of its buckets from the other cores, it combines the buckets and sorts all the numbers within the combined bucket 5. All cores send their sorted buckets to the master core

TABLE II. PSEUDOCODE FOR TWO BUCKET SORT ALGORITHMS

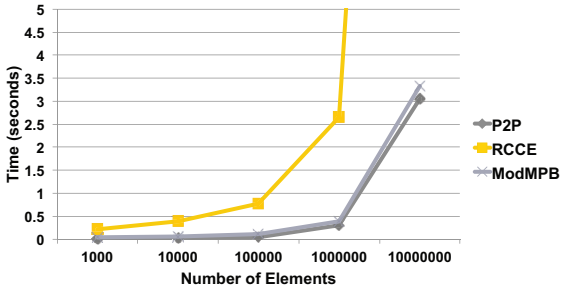


Fig. 16. Bucket sort communication latency (comparing bucket sort algorithms that use broadcasts to an algorithm that uses only point-to-point communication).

would substantially reduce the burden placed on the programmer to produce optimized code, this is an option. Efficient broadcasting can simplify programming with negligible performance loss in this case compared to smaller, less bandwidth-intensive point-to-point messages. Although this example is straightforward, one could imagine scenarios where efficiently partitioning the data is more difficult.

H. Average Power

Figure 17 shows the average power for the SCC including memory controllers, on-chip network and cores for both the RCCE and `ModMPB` broadcasts when sending a 1MB message from core 0 to an increasing number of recipient cores. Power readings were taken from the time the broadcast was initiated up to the point when last core received the message. The results show that `ModMPB` is more power efficient than the RCCE broadcast for large numbers of cores. In addition, as `ModMPB` executes the broadcast faster, it will consume less energy.

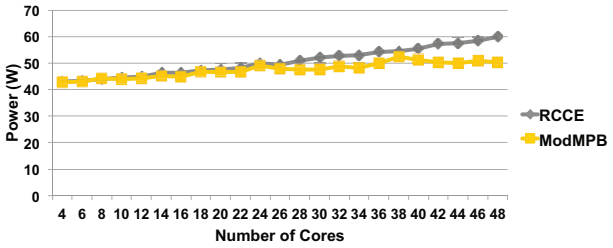


Fig. 17. Power consumption for RCCE and ModMPB broadcasts.

V. DISCUSSION

We have presented results for several broadcasting implementations on the Intel SCC. Using both microbenchmarks and real applications, we study the impact of various broadcasting algorithms on performance. In general, we found that the `ModMPB` broadcast results in superior performance for a range of experiments. However, to optimize the full range of message sizes, one could employ a hybrid approach that selects between the `off-chip` implementation for small messages, `MPB` for medium sized messages and `ModMPB` for larger messages sizes. We have not included results for this hybrid approach but found them to be consistent with the performance for each algorithm in their optimal operating range.

Although we have focused on the SCC platform and have leveraged specific hardware features of this platform, the insights from this work can be extended to other types of shared memory and distributed systems. For example, forwarding the message from an edge core to a center core before initiating a broadcast would likely improve performance on any network topology that lacks edge symmetry. By studying the impact of message sizes on broadcast performance, we see that the amount of on-die message passing buffer storage is important. This analysis has implications for future hardware design decisions.

Finally, we demonstrate that efficient broadcasting may have an impact on how algorithms are implemented on many-core platforms. We demonstrate that an algorithm using broadcasting requires less programmer effort and is able to achieve comparable performance to one with only point-to-point messages using our optimized broadcasting strategy, `ModMBP`. Although broadcasts may occur infrequently, they have significant performance impacts. Efficient broadcast support may result in an increased use of broadcasting to ease the burden on programmers.

VI. RELATED WORK

In this section, we discuss related work in communication and broadcasting on the SCC, optimizations to broadcasting in other message passing systems and on-chip network optimizations for broadcasts.

A. Broadcasting on the SCC

The SCC represents an interesting communication architecture in the space of many-core chips. As such, there has been interest in studying the behaviour of communication within the on-chip network and utilizing the message passing hardware.

Performance analysis of RCCE focusing on varying message sizes and message buffer availability has been explored [13]. Optimizations that exploit the special SCC hardware and focus on very small messages including collective operations have been proposed [14].

Broadcast and gather performance [15] focusing on a small number of message sizes and the number of cores involved in the broadcast have been analyzed. Their characterization of RCCE broadcast performance is consistent with ours. Furst and Coskun analyze power and performance of the RCCE message passing library [16]. In this work, they consider a broadcast scheme similar to the RCCE broadcast; they measured the IPC and execution time of sending a broadcast message and found the IPC peaked at 8 cores, which is substantially fewer than the cores provided by the SCC. Clearly, optimizing broadcasting is important to achieve desirable levels of performance and scalability.

OC-BCast [17] is another efficient broadcasting algorithm; this algorithm is similar to `ModMPB` as it also has the receiving cores copy the message from the sending core's MPB to their own MPBs. However, there are a couple distinct differences between the two broadcasts. In OC-Bcast, each core is responsible for k children while the `ModMPB` only uses 4 temporary cores to propagate the message. The other difference is the handling of large messages. OC-Bcast uses a double-buffer scheme that pipelines message propagation; this pipelining is not favorable for parallelizing off-chip memory writes and off-chip memory reads done by the broadcasting core. This overlapping is a key feature of ModMPB.

Petrovic et al. also implement an asynchronous broadcast using interrupts on the SCC [18]. They adapt their OC-BCast to an asynchronous implementation and then compare it to their synchronous implementation. Although their work only examines small message sizes (i.e. only messages that can fit into the MPB), their work reveals that their asynchronous broadcast performs better than its synchronous counterpart with messages 32 bytes or smaller.

In our results, we compare against the `MPB broadcast` [10]. The authors report a 22× speed-up compared to Intel's current broadcasting implementation. In our evaluation, a 32× speedup was achieved. This discrepancy is likely due to the fact that the broadcast message sizes in the original work are not as large as the broadcast message sizes that we evaluate.

B. MPI Broadcasts

There has been significant previous research on optimizing MPI libraries. Prior to the emergence of many-core architectures, MPI optimizations focused on distributed computing clusters [19]. One broadcast enhancement proposed by Barnett et al. is a scatter-gather type approach; information from the broadcasting core is scattered rather than broadcast, after which a gather is done by all cores [20]. While this optimization works on clusters, we suspect that the latency of confirming that each core received the initial scattered information followed by each core doing a gather would be higher compared with an on-chip network that had cores simply read from one core's on-chip memory location (even with network contention).

MPI optimizations have also targeted the Tilera Tile64 architecture [20]. Kang et al. implement a tree-like MPI broadcast on the Tile64. This broadcast bears some similarity to our `Parallel broadcast` and would likely have similar performance. Both of these broadcast could be implemented on either the SCC or the Tile64 since they do not leverage architecture-specific features.

C. On-Chip Network Support for Broadcasting

Software optimizations for efficient broadcasting can significantly improve performance. In addition to these techniques, there has been significant recent research into adding hardware support for broadcasting to the on-chip network [21]–[23]. By propagating fewer messages and making intelligent decisions about where to replicate messages in the network, these optimizations reduce latency by lowering contention in the network; they can also save power relative to the multiple unicast approach. Although not implemented in the SCC hardware, these types of techniques would likely further enhance the performance and may open up opportunities for hardware/software co-design. Hardware support for other collective communication mechanisms such as reduction operations have also received recent attention [22], [24]; these techniques reduce hotspots and power consumption in the network. Software optimizations on the SCC for reduction operations is left as future work.

VII. Conclusion

We present several novel broadcasting algorithms with two goals in mind: (1) providing better performance than the current RCCE broadcast, and (2) scaling well as the number of participating cores in the broadcast increases. Most of the broadcast strategies presented fulfill these two goals. In particular, the best performing broadcast, `ModMPB` shows significant speedups over the RCCE broadcast, especially with large messages. ModMPB also improves latency when varying the number of cores participating in the broadcast. However, ModMPB is not the best performing for all messages sizes; as a result, we see an opportunity to combine multiple broadcast implementations. Based on our results, one should use the `off-chip` implementation for small messages, `MPB` for medium-sized messages and `ModMPB` for larger messages sizes. Broadcasts such as `ModMBP` are tailored to exploit the specialized hardware on the SCC. Other broadcasts that we propose are not architecturally dependent on features of the SCC. We believe these algorithms could be extended and applied to other systems and networks in order to improve performance.

Acknowledgements

We would like to thank Intel for generously providing us with access to the SCC platform and associated support. This research has been supported in part by Natural Sciences and Engineering Research Council of Canada (NSERC) and the University of Toronto. We thank the anonymous reviewers for their valuable feedback on improving this work. We also thank Sam Vafaee and Steven Gurfinkel for their helpful suggestions.

REFERENCES

[1] J. Howard, S. Dighe, S. R. Vangal, G. Ruhl, N. Borkar, S. Jain, V. Erraguntla, M. Konow, M. Riepen, M. Gries, G. Droege, T. Lund-Larsen, S. Steibl, S. Borkar, V. K. De, and R. van der Wijngaart, "A 48-core IA-32 processor in 45nm CMOS using on-die message-passing and DVFS for performance and power scaling," *IEEE Journal of Solid-State Circuits*, vol. 46, no. 1, pp. 73–183, January 2011.

[2] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C. Miao, J. Brown, and A. Agarwal, "On-Chip Interconnection Architecture of the Tile Processor," *IEEE Micro*, vol. 27, no. 5, pp. 15–31, 2007.

[3] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar, "An 80-tile 1.28TFLOPS network-on-chip in 65nm CMOS," in *IEEE Int'l Solid-State Circuits Conference*, Feb 2007.

[4] M. Martin, M. Hill, and D. Sorin, "Why on-chip cache coherence is here to stay," *Communications of the ACM*, vol. 55, no. 7, pp. 78–89, 2012.

[5] J. H. Kelm, D. R. Johnson, W. Tuohy, S. S. Lumetta, and S. J. Patel, "Cohesion: An adaptive hybrid memory model for accelerators," *IEEE Micro*, vol. 31, no. 1, pp. 42–55, January/February 2011.

[6] B. Choi, R. Komuravelli, H. Sung, R. Bocchino, S. Adve, and V. Adve, "DeNovo: Rethinking hardware for disciplined parallelism," in *In Proceedings of the Second USENIX Workshop on Hot Topics in Parallelism (HotPar)*, 2010.

[7] T. Mattson and R. v. d. Wijngaart, "RCCE: a small library for many-core communication," Intel, Tech. Rep., January 2010. [Online]. Available: http://communities.intel.com/servlet/JiveServlet/previewBody/5628-102-3-22522/RCCE_Specification.pdf

[8] R. van der Wijngaart, "Broadcast functions," http://marcbug.scc-dc.com/svn/repository/trunk/rcce/src/RCCE_bcast.c, December 2010, [Online; accessed 11-October-2011].

[9] M. Konow, "Single-chip cloud computer - an experimental many-core processor from Intel labs," http://communities.intel.com/servlet/JiveServlet/previewBody/5902-102-1-9037/SCC_Sympossium_Mar162010_GML_final1123.pdf, March 2010, presented at the Intel Labs Single-chip Cloud Computer Symposium.

[10] A. Chandramowlishwaran, K. Madduri, and R. Vuduc, "Performance evaluation of the 48-core SCC processor," http://iccs.lbl.gov/assets/docs/ICCS_2011_Talks/34%20Aparna%20Chandramowlishwaran.pdf, January 2011, presented at the LBNL ICCS 2011 Workshop.

[11] D. Abts, N. Enright Jerger, J. Kim, D. Gibson, and M. Lipasti, "Achieving predictable performance through better memory controller placement in many-core cmps," in *Proceedings of the International Symposium on Computer Architecture*, 2009, pp. 451–461.

[12] Intel., "SCC external architecture specification," http://communities.intel.com/servlet/JiveServlet/downloadBody/5852-102-1-9012/SCC_EAS.pdf, Intel, Tech. Rep., November 2010.

[13] T. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe, "The 48-core SCC processor: The programmer's view," in *Proceedings of the 2010 ACM/IEEE Conference on High Performance Computing, Networking, Storage and Analysis (SC10)*, 2010, pp. 1–11.

[14] R. Rotta, T. Prescher, J. Traue, and J. Nolte, "In-memory communication mechanisms for many-cores–experiences with the Intel SCC," in *TACC-Intel Highly Parallel Computing Symposium (TI-HPCS)*, 2012.

[15] P. Gschwandtner, T. Fahringer, and R. Prodan, "Performance analysis and benchmarking of the Intel SCC," in *IEEE International Conference on Cluster Computing*, 2011, pp. 139–149.

[16] J.-N. Furst and A. K. Coskun, "Performance and power analysis of RCCE message passing on the Intel single-chip cloud computer," in *Proceedings of the 4th Many-core Applications Research Community (MARC) Symposium*. Marc Symposium, 2012, pp. 27–32.

[17] D. Petrovic, O. Shahmirzadi, T. Ropars, and A. Schiper, "High-performance RMA-based broadcast on the Intel SCC," in *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures*, 2012, pp. 121–130.

[18] D. Petrovic, O. Shahmirzadi, T. Ropars, and A. Schiper, "Asynchronous broadcast on the Intel SCC using interrupts," in *Proceedings of the 5th Many-core Applications Research Community (MARC) Symposium*, 2012, pp. 24–29. [Online]. Available: http://hal.archives-ouvertes.fr/docs/00/71/90/22/PDF/MARC6_Asynchronous-Broadcast-on-the-Intel-SCC-using-Interrupts.pdf

[19] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in MPICH," *International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005.

[20] M. Kang, E. Park, M. Cho, J. Suh, D.-I. Kang, and S. P. Crago, "MPI performance analysis and optimization on Tile64/Maestro," in *Workshop on Multi-core Processors for Space - Opportunities and Challenges*, July 2009.

[21] N. Enright Jerger, L.-S. Peh, and M. Lipasti, "Virtual Circuit Tree Multicasting: A case for on-chip hardware multicast support," in *Proceedings of the International Symposium on Computer Architecture*, Jun. 2008, pp. 229 –240.

[22] T. Krishna, L.-S. Peh, B. M. Beckmann, and S. K. Reinhardt, "Towards the ideal on-chip fabric for 1-to-many and many-to-1 communication," in *Proceedings of the International Symposium on Microarchitecture*, 2011, pp. 71–82.

[23] L. Wang, Y. Jin, H. Kim, and E. J. Kim, "Recursive Partitioning Multicast: A bandwidth-efficient routing for networks-on-chip," in *International Symposium on Networks-on-Chip*, May 2009, pp. 64 –73.

[24] S. Ma, N. Enright Jerger, and Z. Wang, "Supporting efficient collective communication in NoCs," in *International Symposium on High Performance Computer Architecture*, 2012, pp. 165–176.