A

NoCMsg: A Scalable Message Passing Abstraction for Network-on-Chips

Christopher Zimmer, Frank Mueller, North Carolina State University

The number of cores of contemporary processors is constantly increasing and thus continues to deliver ever higher peak performance (following Moore's transistor law). Yet, high core counts present a challenge to hardware and software alike. Following this trend, the network-on-chip (NoC) topology has changed from buses over rings and fully connected meshes to 2D meshes.

This work contributes NoCMsg, a low-level message-passing abstraction over NoCs, which is specifically designed for large core counts in 2D meshes. NocMsg ensures deadlock free messaging for wormhole Manhattan-path routing over the NoC via a polling-based message abstraction and non-flow controlled communication for selective communication patterns. Experimental results on the TilePro hardware platform show that NoCMsg can significantly reduce communication times by up to 86% for single packet messages and up to 40% for larger messages compared to other NoC-based message approaches. On the TilePro platform, NoCMsg outperforms shared memory abstractions by up to 93% as core counts and inter-process communication increase. Results for fully pipelined double precision numerical codes show speedups of up to 64% for message passing over shared memory at 32 cores. Overall, we observe that shared memory scales up to about 16 cores on this platform while message passing performs well beyond that threshold. These results generalize to similar NoC-based platforms.

Categories and Subject Descriptors: C.4 [**Performance of Systems**]: Measurement Techniques; C.4 [**Performance of Systems**]: Modeling Techniques

General Terms: Experimentation, Tracing, Compression

Additional Key Words and Phrases: Multicore Architectures, Shared Memory, Message Passing

ACM Reference Format:
Christopher Zimmer, Frank Mueller, 2014. NoCMsg: A Scalable Message Passing Abstraction for Network-on-Chips. *ACM Trans. Architec. Code Optim.* V, N, Article A (January YYYY), 23 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Multicore processors are becoming ubiquitous and offer tremendous opportunities to meet processing demand. This comes at the expense of limited scalability due to on-chip (interconnect) and off-chip (memory) resource contention.

Such high core counts present a challenge to server, cloud and high-performance computing with projections requiring programmers to harness node-level parallelism of hundreds of cores. Contemporary shared memory techniques have struggled to scale, particularly as the single system image (SSI) remains the traditional system abstraction. SSI was a good match for bus-based multiprocessors in the past. However, bus-based designs do not scale well (even beyond four processors) and have been replaced by ring and mesh interconnects (e.g., Hypertransport, Quick Path Interconnect)

This work was supported in part by NSF grants CNS-0905181, CNS-0958311, CNS-1239246 and a subcontract from SecurBoration.
Author's addresses: C. Zimmer and F. Mueller, Dept. of Computer Science, North Carolina State University, Raleigh, NC, USA, email: mueller@cs.ncsu.edu

DOI: http://dx.doi.org/10.1145/0000000.0000000

with currently up to 16 cores per socket and, for high core counts, tile-based architectures with 2D meshed network-on-chip (NoC) interconnects, including Intel's Phi/MIC code-named "Knights Landing" (KNL) and Kalray's 256 core [5; 6; 33; 27; 4; 12].

Embedded architectures increasingly feature multicores with 4-64 processors (e.g., ARM's MPCore, Qualcomm's Snapdragon X6, Samsung's Exynos family, Cavium's Octeon, Freescale's QorlQ, and TI's TMS320C80 MVP). Even GPUs (mostly from NVIDIA and AMD) and heterogeneous APUs (AMD Fusion APUs, NVIDIA's Tegra3/4/K1 and ARM's MALI designs) with hundreds of compute elements have been considered for embedded/real-time systems [19; 20].

Shared memory systems generally provide cache coherence via MESI-style protocols enhanced by coherence filters [24]. On mesh-based systems, such protocols may limit scalability as the number of cores increases. Consider the multikernel (aka. Barrelfish), which follows a distributed kernel paradigm that employs messages in an off-chip mesh interconnect of Hypertransport links [9]. For configurations of just eight processors, messaging was shown to outperform shared memory for a number of parallel benchmark codes.

Contributions: This work demonstrates that NoC efficiency benefits significantly from reduced congestion and backpressure. It contributes runtime optimizations for message passing of the NoC via flow-control elimination techniques based on these principles and develops NoCMsg, an MPI-like communication library. NoCMsg builds on the abstraction of a distributed memory architecture between cores, i.e., it does not utilize *shared* **data** *memory* at all. It is specifically designed for large core counts in 2D meshes.

Its design ensures deadlock free messaging for wormhole Manhattan-path (dimension-ordered) routing over the NoC. This is in contrast to low-level NoC messaging, where limited message buffer space may result in deadlock [7] when a pair of cores sends messages to each other, i.e., they may send flits of messages until all buffers overflow without ever draining them by issuing receives. This results in senders involuntarily stalling their processor pipeline until the transfer can complete. Instead of employing virtual channels that monopolize NoC links between end points, NoCMsg adaptively alternates between sending and receiving by sensing buffer thresholds.

NoCMsg further relaxes communication constraints by exploiting pattern-based communication common in MPI runtime systems to identify areas in which flow control is unnecessary and provides an MPI-like runtime system [15] interface with unprecedented performance. Experimental results on the TilePro hardware platform show that NoCMsg has lower latencies and provides higher throughput for small messages than past NoC-based messaging abstractions. Performance improvements of up to 86% are observed in communication for single packet messages and of up to 40% for larger messages. For a subset of the NAS Parallel Benchmarks [8], NoCMsg is also shown significantly more scalable than prior messaging techniques.

A heads-on comparison between message passing and shared memory is provided on the Tilera platform, where the former is supported in firmware while the latter is implemented by NoCMsg. Experiments demonstrate that NoC messaging outperforms shared memory abstractions of OpenMP NAS PB program by up to 93% beyond 16 cores for integer-based workloads and up to 64% for double-precision numerical codes on the Tilera platform.

2. BACKGROUND

In this section, we motivate the necessity of a low-cost deadlock free message-passing library for NoC architectures. We specifically discuss trade-offs that are made in favor of high-throughput communication and their effect on deadlock potentials for the NoC. We also discuss the current state of the art strategies for facilitating flow control.

Inter-processor communication in NoCs is realized via traditional network communication, i.e., data is transferred between cores as messages. These messages are broken into fixed sized packets composed of flow control digits (flits). Messages are packetized and transferred via XY dimension-ordered wormhole routing in 2D meshes. This design is common to several NoC architectures [14; 6; 4], including Intel's Phi/MIC code-named "Knights Landing" (KNL). Contemporary NoCs feature increasing throughput in communication. This becomes feasible due to simplistic routing protocols

with single cycle per-flit transfer latencies. However, such latencies can only be guaranteed in the absence of contention. This work contributes methods to address this challenge.

Another problem is that bare-metal message passing may lead to deadlocks — unless more advanced hardware protocols or software libraries internally imposing structured communication protocols, both implying additional overheads. As an example, consider wormhole routing on the TilePro 64. Wormhole routing describes a packet transfer strategy, where pathways through the switching network are opened by the head of the packet and remain open until the final flit of the packet is seen. The ramification of this is that packets of other messages crossing a currently open path remain blocked until this wormhole is closed. This alone does not result in deadlock as long as packets transfer successfully. The problem arises when SRAM buffers reach capacity on a receiving switch and its attached core is unable to drain the buffer. When this situation occurs, a crossing packet will be stalled mid-flight, blocking the packet's sender and any other cores sending data that share any portions of that packet's path. Consider two tasks shown in Figure 1 transferring fixed size buffers to each other concurrently. In the Tilera architecture, the receiving tasks can buffer up to 127 words in a flit. However, when the buffer becomes full the switching network must wait until flits are drained before transferring any remaining flits. Exchanging contiguous buffers exceeding 127 words will result in a deadlock (infinite blocking), not just for the two cores but also affecting any messages going across this link between the two cores on some route.

Such deadlocks can be avoided. For instance, indefinite blocking is often avoided by interrupt-based channel creation. Tilera's iLib communication library uses channel creation through protocol messages. It sends a single flit message to a destination and awaits an acknowledgment. If the request is not acknowledged, it is retried after a timeout and then reissued (from the source core) until it finally succeeds. Once a channel is created, a message of one or more flits can be transferred. Unfortunately, protocol messages are also subject to deadlock. Hence, the library must provide a timeout interrupt to break out of the communication. The sending process can thus drain pending receives (without acknowledgment since messages will be re-sent). It may continue to send packets once all blocked receives are drained.

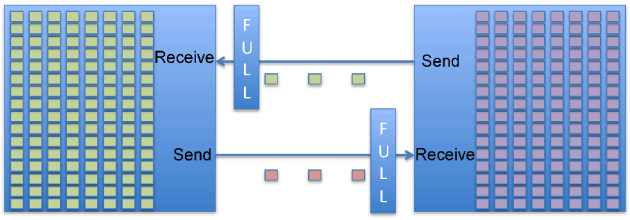


Fig. 1. Message Passing Deadlock

3. DESIGN

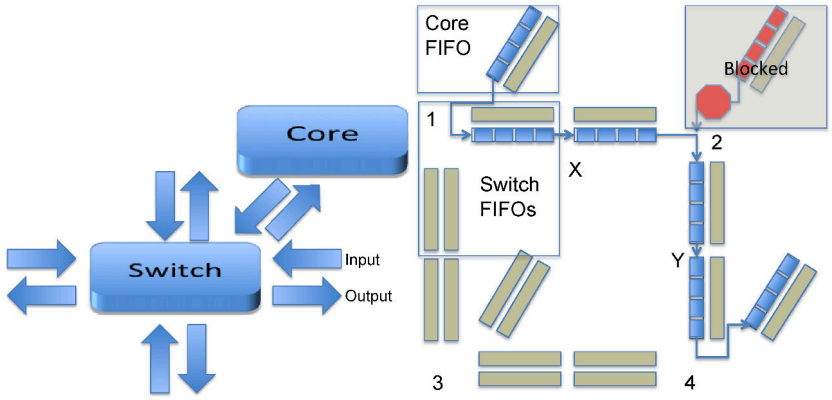
Our design, the NoCMsg layer, is driven by the objective to create a close to bare metal NoC-level messaging protocol. Since this requires low-level NoC capabilities to be used, one becomes exposed to wormhole routing problems, such as potential deadlocks. Our NoCMsg design ensures absence of deadlocks with reduced flow control to lower overheads within its protocol layer. We assume a generic, generalized 2D mesh NoC switching architecture similar to existing fabricated designs with high core counts, which is a viable solution for future microprocessor design. Notice that even 3D stacking of memories still assumes a single silicon layer of processing cores at the top of the stack due to thermal constraints, likely with a 2D NoC mesh to ensure scalability. Each core is composed of a compute core, network switch, and local caches. Communication over the NoC is assumed to be reliable, lossless, and without duplication. We assume wormhole Manhattan-path (XY dimension-ordered) routing over the NoC, but the method is applicable to any static routing scheme. (Dynamic

routing is not likely to become feasible for NoCs due to single cycle routing requirements.) We discuss its relation to our NoCMsg layer design and describe constraints of such an architecture next.

3.1. On-Chip Interconnect

Today's network-on-chip architectures replace the conventional system bus or other topologies of connecting cores, such as rings, with a more scalable 2D mesh interconnect for manycores. This means that all memory, messaging, and I/O communication occur over the NoC, often through physically separate networks to reduce contention. E.g., processors from Adapteva [1] feature three networks and Tilera's TilePro [6] five networks. The Intel SCC [4] and Kalrays [12] design only have a single network and do not natively support coherence over their 2D mesh NoC, just messaging.

For the purpose of this work, we focus on the messaging network. In NoCs, messages are used for inter-processor communication. This deviates from system-bus networks that only support shared memory as a means of communication. Similar to traditional networks, messages are split into packets containing information for routing within the switching network. A packet contains a payload of data for the recipient. Our work focuses on 2D mesh core layouts with wormhole routing. Yet, our contributions to flow control operate irrespective of the switch topology, i.e., our approach can easily be extended to 3D meshes for future stacked architectures.



(a) Core - Switch Topology (b) Path-based Backpressure

Fig. 2. NoC Routing over Switches and Links

3.2. Manycores

Today's manycores feature a large number of independent compute cores in a 2D mesh NoC. Each compute core interacts with its switch using input and output queues that are accessed via specialized registers as depicted in Figure 2(a). When the output queue from the core to the switch becomes full, subsequent writes to this queue will stall the pipeline until there is space for the write. The inverse also holds: when the input queue is empty and the queue is read, the pipeline stalls until data is available. Hence, NoC communication supports a blocking communication API at one level and a non-blocking/interrupt triggered API for polling/event-driven communication at another level.

3.3. NoC Switches

Each core is generally associated with a switch, which is composed of multiple sets of input and output queues attached to a crossbar interconnect. Each output queue is mapped to input queues of neighboring switches to support the flow of flits. In wormhole networks, header packets create mappings of output queues to input queues as they traverse the network. The mappings are revoked

as a switch services the tail flit of the corresponding packet. To enable the detection of open input ports, output ports maintain a set of N transfer credits. When a flit of data is placed into an output queue, a credit is consumed. When that credit is transferred to the subsequent input port, either between the core and switch or between two separate switches, the credit is refunded. Credits are checked when an internal mapping is established between an input queue and an output queue, i.e., when a connection is established through wormhole routing. If the output queue is unable to receive any data due to a lack of credits, no additional output data may be transferred until credits are refunded.

Figure 2(b) depicts an example where core 1 sends a message to core 4. Using XY dimension-ordered routing, this message passes from core 1's output queue to switch 1's East output queue. Each post decrements a credit when the flit of data enters the queue. Switch 1's east output queue will then transfer to switch 2's input queue, and switch 2 will set the cross bar to transfer the packet to switch 2's South output queue, if enough credits exist in the South output queue. Subsequently, switch 4's Northern input queue will receive the flits from switch 2's Southern output queue and refund credits. Switch 4 will then create a mapping of the Northern input queue onto the core's input queue. Incoming flits into core 4's input queue will be automatically buffered in a larger SRAM FIFO buffer. Once this buffer fills up, no more data can be transferred and corresponding attempts result in blocking at the API.

3.4. Credit Monitoring: Backpressure Check

Output queues are assumed to maintain a series of credits. Only if credits are greater than zero can more data be added to a queue; otherwise, the pipeline will stall. *It is these credits that make up the basis of our flow control technique.* We assert that by checking credits on the sender side's output queue, we can avoid deadlock and reduce the cost of sending messages using virtual channel flow control techniques. In the following, we characterize back-flow resulting from two types of blocking in the network. The first is receiver-side buffer blocking. In this situation, the receiver-side SRAM buffer has reached capacity and is unable to accept any more data. This implies that the receiver's local input queues are unable to move any data into SRAM, effectively halting refunds of queue credits to the output queues on the previous core in the path. Figure 2(b) gives an example where node 4 is unable to receive any more data. This backpressure can only be resolved if node 4 actively drains the network to free up space within its hardware buffers.

The second type of backpressure occurs when a switch is unable to route a packet due to an open wormhole path. In Figure 2(b), the message sent between cores 1 and 4 is blocking a message sent from cores 2 to 4. Here, one needs to ensure that the message between 1 and 4 completes to resolve blocking.

The design of our NoCMsg layer avoids deadlocks in a generalized fashion. We utilize a polling work loop that cycles between computation, sending, and receiving of data. In this work loop, a buffered message is only sent if sufficient credits are available in the output queue; otherwise, the credit check is repeated in the next iteration of the work loop. When the head of a message opens a wormhole path, NoC links on the path remain reserved until the full payload (encoded in the head) has been received, which severely limits the number of concurrent paths. If two (or more) paths share links or endpoints, then the later will block at an input queue of a switch. As this later sender submits more flits, it would eventually experience head-of-line blocking, which would block the sender in the work loop and prevent it from receiving more flits, which could result in deadlocks or circular send/receive dependencies. To prevent such blocking, we no longer send flits when the output queue at the sender is full. But we still receive flits, which ensures that no deadlock can occur since receives reduce backpressure.

In general, a resource monitoring approach equivalent to that of our output queues can be applied to a variety of NoC architectures if they provide feed-back on failure conditions. Work loops provide a solid strategy for balancing communication and computation within the cores without costly interrupt service routines and buffers protected by locks.

4. IMPLEMENTATION

We implemented our high-level design in NoCMsg on the Tilera platform. Nonetheless, our general design from Section 3 extends to any 2D mesh NoC architectures and has been ported [25] to the Intel SCC [4], but this port is beyond the scope of this paper.

NoCMsg provides an MPI-like API with modified semantics to specifically unleash the potential of NoC efficiency, e.g., by integrating credit-checking flow control and optional elimination of flow control. This results in significant performance improvements when an application or internal message-passing runtime routines allow the omission of flow checking. The difference between flow- and non-flow control communication is seen in the following API prototypes, which underline the close resemblance between NoCMsg and MPI. A regular "Send" operation even mimics the flow control constraints (in terms of blocking requirements) of its equivalent MPI call. In contrast, "Xsend" eliminates flow control altogether, i.e., it differs fundamentally in the underlying semantics and operates at the low-level NoC messaging layer instead of utilizing operating system / MPI runtime capabilities. (The sync parameter is explained next.) Since the APIs are a close match, MPI programs can easily be ported to NoCMsg.

```
NoCMsg_Send(void *buf, uint32_t size,
 NoCMsg_Datatype dt, uint32_t dest,
 NoCMsg_Comm comm) // flow control
NoCMsg_Xsend(void *buf, uint32_t size,
 NoCMsg_Datatype dt, uint32_t dest,
 NoCMsg_Comm comm, bool sync) // no flow control
```

4.1. Point-to-Point Messages

At the core of NoCMsg lies a work loop in which sends and receives are issued based on the availability of resources, and work is performed when no communication is outstanding, where sends are conditionally issued if sufficient credits exists. As previously described, low-level point-to-point messages are subject to deadlock in the absence of flow control due to the nature of the NoC switching architecture. The conditional sends implement a means of back pressure monitoring to ensure absence of deadlock for any message transaction.

The underlying credit monitoring scheme is specific to Tilera. We read an existing memory-mapped hardware register indicating the credits of the output queue at the core-local NoC switch. In general, any other resource management of other NoC architectures could be used in its place, e.g., co-processor failure registers for non-blocking transfers.

Our user-level asynchronous message API is built on top of this low-level work loop. It provides building blocks for user-level synchronous communication, collective operations, and barriers. Two alternating operations for asynchronous communication comprise the core of the work loop in our implementation.

(1) Trysend: This API call implements conditional sending of a message. During the send of a packet of flits, the output queue's available credits are inspected. We then place as many flits in the output queue as credits are available, i.e., credits are queried for each and every transfer. Control is returned to the work loop if no credits are left.

(2) Tryreceive: This API implements conditional data reception. The MPI ready-send specification for point-to-point sends and receives requires synchronization between any send/receive pairs [15]. For synchronous communication, this means a send will not be completed until the sender has seen an acknowledgment from the receiver. In asynchronous communication, send and receive will initiate communication, yet may return from the API call before the operation completes. Should a matching sender-side MPI_Wait() call follow, then a similar acknowledgment has to first be seen by the sender. A completed MPI_Wait() after an asynchronous receive simply indicates that the receive completed. These requirements for acknowledgments and completion of calls ensure ordering within the packet stream with respect to a given sender/receiver pair. This can be exploited for flow control elimination when MPI_Wait calls are present.

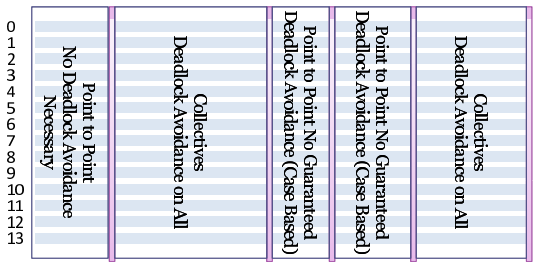


Fig. 3. Profile Detected Communication Regions

There is one subtle difference between MPI and NoCMsg: *NoCMsg introduces so-called synchronous non-flow controlled messages that diverge from traditional MPI semantics.* Its objective is to exploit common communication patterns found in the implementation of collectives within the message-passing runtime but also in application codes. NoCMsg guarantees contention free communication and tries to maximize the number of concurrent communication paths utilized for any specific pattern to reduce protocol overhead and end-to-end execution time of a communication primitive. Synchronous non-flow controlled communication is supported for send and receive operations for (a) regions between collective communication and (b) within the implementation of barriers if no flow control is required. We identify these patterns based on (a) the communication object of collectives and (b) the analysis of communication patterns in benchmarks, both of which can be automated (beyond the scope of this paper) but were manually conducted in this work.

Non-flow controlled transfers require that (a) messages sent simultaneous travel on disjoint paths and do not share common sinks and (b) a small setup overhead to synchronize the sender and receiver if the buffer is larger than a packet, i.e., a single packet is exchanged to ensure that the receiver is ready to receive. If these requirements are met, deadlock freedom is guaranteed even in the absence of flow-control checking. We ensure these requirements by explicitly creating disjoint paths in patterns specific to a communication directive. This is shown in the code presented above: The non-flow controlled calls feature a synchronization boolean and execute a send that bypasses any credit checking. This has the side effect of avoiding data congestion, which increases performance. After this synchronization, full messages can be transferred without the use of any interrupts or credit checking.

A drawback of exposing flow-control free operations is that semantic correctness, when utilized, is not dynamically checked. A developer could choose this capability for optimization and subsequently introduce errors to the program logic that may result in communication deadlock. To avoid such semantic violations, we promote an inspector-executor step detailed next. Static or dynamic checkers could also be developed to this end. Again, fully automated tools to validate correctness when substituting API calls with equivalent non-flow-controlled ones can be developed. One would need to analyze all communication calls in regions delimited between collectives, determine endpoints and then apply rules R1-R4 that constrain flow elimination (see rules below). The challenge is to determine endpoints in a non-input specific manner or even for non-deterministic (partially randomized) endpoints, but channel determinism might be a sufficient property [26]. This is beyond the scope of this work.

We took a different approach due to the prototypical nature of our investigation. We identified if flow control can be safely removed in certain message transfers by profiling applications to identify regions of code with suitable communication patterns. A NoCMsg profiling run produces information about sender and receiver, code region mapping, and communication type, *i.e.*, synchronous or asynchronous. This data is the basis for the construction of unique communication flow graphs for each code region, where collective operations and barriers mark region boundaries. This graph construction is currently manual but could easily be automated as discussed above.

This data is subsequently analyzed to detect communication patterns that inhibit flow-control elimination following the rules stated below. The approach is conservative in that regions that *may* require flow control are excluded when in question. We establish the following rules: (R1) Asynchronous communication that crosses a collective, i.e., an asynchronous send before the barrier on one side with a matching receive after the corresponding barrier on the other side, prevent flow-control elimination. (R2) Point-to-point communication where multiple sources share a sink prevent flow-control elimination. (R3) Point-to-point communication where source/sink pairs share a (directional) link prevent flow-control elimination. (R4) A circular source-sink chain prevents flow-control elimination.

These constraints can be relaxed for special cases. (C1) We support pairwise exchanges (send/receive pairs that form a cycle of length two), which are detected and subsequently optimized via ordering by rank (first the lower sends and higher receives, then vice versa) to eliminate flow control despite R4 (but are still subject to the other rules). (C2) Flow control is eliminated for collectives with a common sink despite R2 and R3. This is supported by serialization of communication on common paths provided by the hardware's wormhole routing algorithm, i.e., end-of-line blocking may occur but is upper bounded in time by the number of sources times the maximum single pair transfer time.

Figure 3 shows an example of a set of detected patterns. In the figure, bars show barriers or other collectives that separate different regions of code. To indicate the type of communication they contain, regions of corresponding code are marked.

```
for i = 1 .. n
  MPI_Irecv(_,stencil[i],_, req);
  MPI_Send(_,stencil[i],_);
  MPI_Wait(req,_);
```

(a) Original Code

```
for i = 1 .. n
  NoCMsg_Xchng(_,stencil[i],_);
  NoCMsg_Barrier(_);
```

(b) Flow Control Removed

Fig. 4. Flow Elimination for NAS Benchmark CG (Stylized)

Figure 4 provides a concrete example. It depicts a stylized code excerpt from the NAS benchmark CG before and after flow control elimination. Before, a non-blocking receive followed by a blocking send and a wait (for receive completion) are issued per rank/core. The dynamic profile indicated that CG uses a 2D neighbor communication pattern ("stencil") of pairwise independent exchanges. After flow elimination, exchanges are followed by a barrier, where the exchange initiates a receive followed by a send if the local rank is lower than the destination, otherwise vice versa. Notice that the barrier separates rounds of pairwise neighbor communication and thus contributes to a contention free NoC. Such flow elimination would not have been legal if nodes were subject to multiple receives per round. If they were, as described before, deadlocks could occur at the low-level layer.

4.2. Collectives

For collective, one can make safe assertions about the content of the network messages in flight at a given point in time for NoC communication. This offers significant opportunities to eliminate flow control in collectives. To this end, we assume that the NoCMsg program is the only program executing on the NoC (or, at the very least, is contained in a hard-walled NoC grid) effectively isolating the grid network ports.

Collectives communicate data among all processes of a group. As an example, consider two common collectives, broadcast and reduction. Their semantics require no flow control to exchange messages (so long as underlying point-to-point paths are disjoint or their messages are separated by internal barriers). The second rule (R2) for flow control elimination requires absence of a common receiver. Broadcast meets this criterion while reduction only meets it under special case (C2), which allows a common receiver for collectives. If the process group synchronizes prior to the collective and no asynchronous communication is in flight across it (see R1), then no in-flight point-to-point

messages before non-monitored message transfers may exist, i.e., no such message endpoints (even from synchronous point-to-point communication) can cross the collective.

The most demanding collectives are Alltoall and alltoallv in terms of network contention. They also provide opportunities for the elimination of flow control. Based on the particular internal send and receive orders in these collectives, it is possible to guarantee flow-control free communication for pairwise core transfers. To ensure deadlock freedom by absence of cycles (due to the acyclic pattern), a single receiver is acquiring data from all cores at any given time in our design following the design case (C2) above. Synchronization separates rounds with different receivers from one another.

4.3. Barriers

For deadlock free communication, our current implementation of collectives requires prior synchronization of execution to ensure that no point-to-point messages are in flight. We have created a new barrier interface specifically for this purpose that also improves performance over a shared memory barrier design. In order to provide scalable barriers, we implemented tree-based barriers that distribute the work evenly among nodes and thus improve balance by reducing the cycle differences upon barrier completion.

Our Tilera implementation utilizes rooted n-ary trees to this end. The root of this tree is placed in the center of the NoCMsg grid to minimize latency (hops). The process of synchronization is simple: Children notify 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 sending to its children and exits, as do the children. To guarantee isolation for processes that have not yet entered the barrier, we use a separate SRAM buffer. This also eliminates the need to use the standard packet header, which would unnecessarily increase the size of a synchronization packet.

Flow control is not needed in barriers as the prerequisite of entering into a barrier is that all outstanding sends and receives on the local core are complete. 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 (see Section 6).

4.4. Network Partitioning

In the context of network partitioning, flow-control elimination should also be considered. The techniques discussed in this work assume run-to-completion tasks and absence of cross communication from outside task sets.

As an example, consider the TilePro design but assume that the 64 cores are partitioned into four quadrants (NE, NW, SE, SW) or 16 cores each. Jobs of up to 16 tasks can then run in parallel on different quadrants without any messages ever interfering. In contrast, any mixed partitioning (e.g., a set of even and a set of odd cores) could result in common link usage between an even and an odd task. This would violate the flow-elimination requirements since a non-flow controlled region of a job may coexist with a flow-controlled region of another job, which can result in deadlock due to violation of R3 for the former job with respect to the latter. To ensure correctness, tasks should be mapped within isolated grids. This also reduces performance perturbation between coexisting jobs. Our experiments systematically follow a grid-isolated layout for tasks.

5. FRAMEWORK

We conducted experiments on a Tilera TilePro processor, namely a 700MHz 64-core version (TilePro 64) with floating point emulation in software [6]. Programs were compiled with Tilera's MDE 3.03 tool chain at the O3 optimization level with Tilera's C/C++/Fortran compilers that also support OpenMP. OpenMP experiments run with enabled coherence (L3 on). The L3 is called a virtual cache since the processor has core-private L1 and L2 caches. Portions of the L2 caches of all cores can optionally be combined into a distributed (virtual) L3 cache. The L3 cache is direc-

tory based (uses address hashing) and supported by the memory dynamic network (MDN). Notice that the MDN has twice the bandwidth of the user dynamic network (UDN). Even though this puts NoCMsg at a bandwidth disadvantage relative to shared memory, NoCMsg over UDN comes out ahead beyond 16 cores for our workload (see next section).

In contrast, experiments comparing NoCMsg and OperaMPI were conducted under disabled cache coherence, i.e., hash-based distributed virtual L3 was turned off. All messages are routed over the UDN.

OperaMPI [18] implements the MPI 1.2 standard [16] for C. It is layered over Tilera's iLib, an inter-tile communication library that utilizes the UDN NoC network. We ported iLib and OperaMPI from MDE 2.0 to 3.03 for a fair comparison. We also made OperaMPI compatible with Fortran by adding wrappers.

The iLib library is vendor-supplied and allows developers to easily take advantage of many of the features provided by the Tilera architecture, including message passing. Point-to-point messages are directly supported by iLib and closely resemble the equivalent MPI semantics. Internally, iLib utilizes interrupt-based virtual channels and complex packet encodings to synchronize senders and receivers for establishing point-to-point connections. However, iLib only supports a limited number of collective operations, namely broadcast and barrier. Hence, OperaMPI creates virtual overlaps (e.g., trees for reductions) to implement more complex MPI collectives such as reduction, all-to-all, all-gather/scatters etc.

We chose the NAS Parallel Benchmark (NPB) codes [8] Version 3.3 for OpenMP, OperaMPI and NoCMsg to conduct experiments. For the initial set of experiments, inputs were modified to allow weak scaling [17] within L2 sizes: As the number of cores is increased, overall problem input sizes are proportionally increased as well so that the core-specific data remains constant and fits into the L2 cache of a local core. Constraining the problem to L2 exposes the overheads of NoC-level communication for these benchmarks without being skewed by off-chip memory references, which otherwise dominate. Hence, the L2 fit of data allows to assess the asymptotic behavior of multicores with near-perfect locality (e.g., for perfect multi-level tiling). This ensures that results are not dominated by off-chip memory bandwidth latencies but instead focus on the on-chip computation and communication. We also assessed a term frequency/inverse document frequency (TF*IDF) benchmark for document clustering in experiments, in a port of the benchmark derived from [35]. The inputs are again weakly scaling to ensure L2 residency of data sets. The code follows a map-reduce paradigm [13].

In a second set of experiments, results for strong scaling are also provided for input class A of the NAS PB codes. This ensures that inputs fit into L2 cache for the smallest number of cores. As the number of cores P is scaled up, inputs only require a fraction of the L2 inversely proportional to P. The third set of experiments assesses the suitability of our approach again under weak scaling, yet this time for native floating-point units (FPUs). Conventional Tilera architectures lack FPUs and utilize software emulation of floating point operations instead. In contrast, the Maestro board [10] provides a Tilera architecture that features FPU support. However, the Maestro compiler and runtime suite did not support Fortran, so that we were forced to conduct experiments with micro-benchmarks on the hardware to then extrapolate the savings due to native FPUs for the NAS PB and TF*IDF codes.

6. EXPERIMENTAL RESULTS

The performance of NoCMsg is assessed by first comparing shared memory and message passing using micro benchmarks on the Tilera platform. More specifically, we refer to *shared* **data** *memory* whenever we use the term shared memory here. In the evaluated benchmarks, instructions can still be shared with little to no impact on other executing applications since we warm up the instruction cache so that nearly all of the instruction references hit in L1 cache since it is sufficiently large.

Next, we compare NoCMsg with OpenMP using the NPB suite. We then compare NoCMsg to OperaMPI, an MPI library specific to the Tilera platform, for the NPB codes. We also evaluate TF*IDF, a document clustering algorithm, by comparing NoCMsg to both OpenMP and OperaMPI.

Finally, we assess the performance potential of NoCMsg for a Tilera architecture with native floating point support.

6.1. Microbenchmarks

An initial assessment of the potential of message passing over the UDN is provided by comparing it with shared memory transfers over the coherence interconnect in a bandwidth micro-benchmark. Two threads exchange varying amounts of data via shared memory writes/reads or send/receive messages. Figure 5 depicts the number of cycles (y-axis) for varying sizes of the transferred buffer. The graphs indicate that shared memory incurs roughly twice the cost of message passing (both without hashing). UDN messages follow a one-sided push model (sender initiated) while shared memory accesses are pull based (receiver initiated) and require at least two messages for a single transfer (due to the MESI protocol).

Hash-based distributed caches reduce the shared memory overhead but the overhead still remains higher than sending messages without hashing, especially for larger transfers. (Notice: Hashing interferes with larger messages while reducing overhead for shorter ones as long as the transferred data fits into local caches.) These results indicate that message passing has the potential to outperform shared memory transfers. In particular, message passing has superior scaling characteristics that become dominant for larger number of cores.

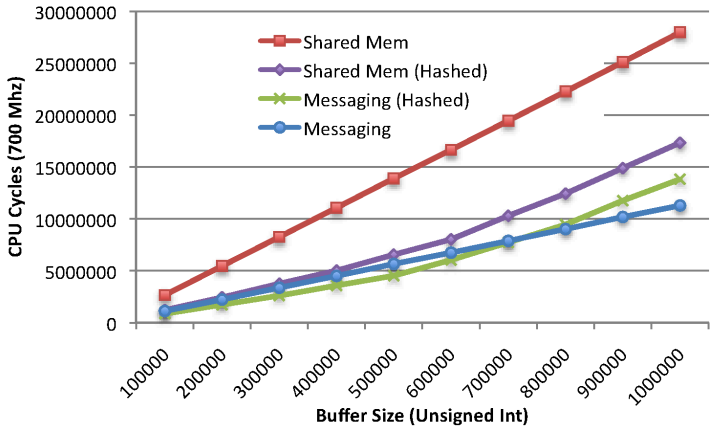


Fig. 5. Shared Memory/Messages NoC Bandwidth

The distributed virtual L3 provides a uniformly distributed address space where core affinity is determined by a hash function. Hashing can thus significantly increase the performance of shared memory by reducing the average distance to cached data and by increasing cache capacity of L3 to the aggregate of all L2 caches. However, this performance increase does not come for free. Even accesses to small data structures that might otherwise fit into L2 are redirected to remote L3. These performance benefits may come at the cost of jitter since accesses to distributed L3 have variable hop counts (NUCA) over the NoC, as discussed next.

Figure 6 depicts the execution time (y-axis) over 20 experiments of 1 MB data exchanges. It shows that shared memory accesses without hashing experience less jitter than hashed ones, i.e., runs deviate by no more than 1% from the average without hashing while they fluctuate by up to 3% with hashing. For message passing, the jitter is less than 1% without caching and up to 2% with hashing. This seems moderate, but keep in mind that only two cores are involved in the data exchange.

In another micro-benchmark, each core accessed varying amounts of data, where a certain fraction of this data was shared by pairs of neighboring cores. With L3 hashing, these references may be

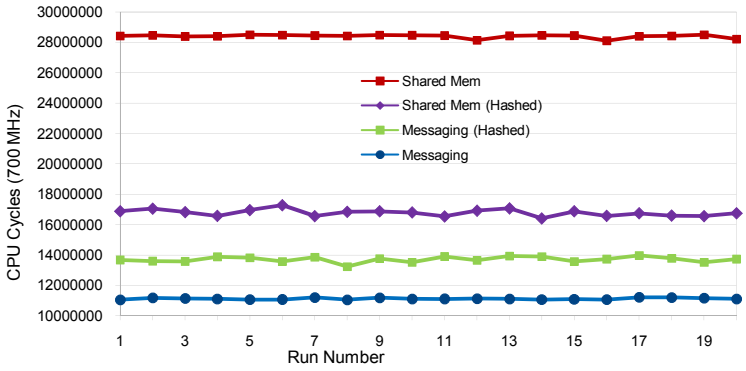


Fig. 6. Shared Memory/Messages NoC Jitter [1MB Xfer]

cashed at remote L2 caches, which incurs contention on the shared memory part of the NoC as the number of cores increases from 8 to 54 cores (4 to 27 pairs of threads sharing memory). Figure 7 depicts the performance in cycles (y-axis) for varying number of cores (x-axis) and different fractions of shared data (0%, 50%, 100%) of all memory accesses. We observe that without sharing, the performance scales linearly since references are resolved out of L1 cache. With 50% sharing, the cost increases slightly, and with 100% sharing the cost for memory accesses increases super-linearly (about quadratically due to contention in the 2D mesh) as the number of cores is increased.

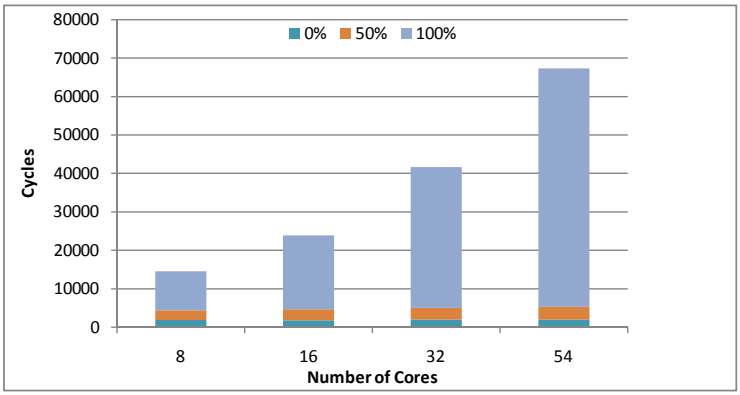


Fig. 7. Shared Memory Latency for Contention Levels

We also conducted an experiment to assess the variance when 400 bytes are exchanged under different rates of contention for both shared memory and message passing. Pairs of neighboring cores perform the exchange, where L3 is off in both setting. Figure 8 depicts the performance results in cycles (y-axis) for up to 32 cores (up to 16 pairs exchanging data) in a tight loop. Error bars depict the minimum and maximum performance over the execution times of each core. We observe that the performance of shared memory increases linearly with the amount of contention (cores) while it remains the same for messaging. Moreover, the execution time varies significantly under shared memory, and this variance increases at about twice the rate of the average time. In contrast, messaging shows little variance with no change as we scale up the number of cores.

These results show that any memory shared accesses may be subject to increasingly higher latencies and jitter as the core count increases on the Tilera platform. While the total amount of jitter may be small for single-threaded code, jitter has the potential to aggregate as the number of cores increases. This may result in unbalanced execution where more and more cores remain idle prior

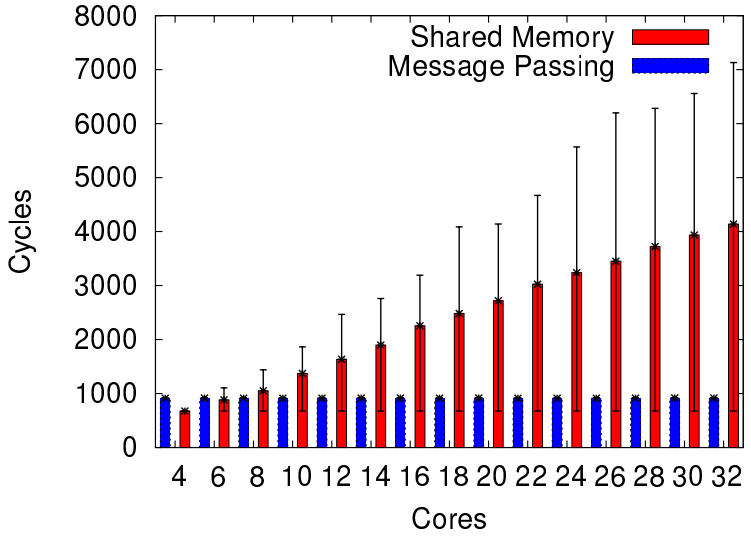


Fig. 8. Data Exchange under Contention for Shmem and Messaging

to global synchronization (e.g., barriers). We term this effect *perturbation*, and discuss it in the following set of measurements.

6.2. Weak Scaling Results

The micro-benchmarks illustrate the tradeoffs between shared memory and message passing on Tilera. To assess these affects using real-world benchmarks, we evaluated several of the NPB codes on the TilePro 64 over (a) shared memory (OpenMP) and (b) NoCMsg. We chose NPB since OpenMP and MPI versions exist for each code, much in contrast to other parallel benchmarks that only provide shared memory codes. We also study the TF*IDF benchmark under the same settings.

In contrast to NPB's default strong scaling inputs, we used our own weak scaling inputs [17] where the data set per core is of fixed size. This weak scaling input size is shown on the secondary y-axis in each of the following figures. Weak scaling ensures that the computational work per core remains the same as the number of cores cooperating in a parallel application is increased. Note that all of these benchmarks, except IS and TF-IDF, operate on floating point or complex data types. The TilePro 64 does not contain any floating point pipelines, *i.e.*, floating point calculations are realized via software emulation. This leads to more time spent in computation vs. inter-processor communication, which gives shared memory an advantage (due to a reduced fraction of communication) over message passing. This advantage is discussed in the next set of experiments.

Figure 9(a) depicts average performance in seconds (y-axis) and minimum/maximum performance (error bars) over repeated experiments with the integer bucket sort benchmark IS, the only integer benchmark in the NPB suite. The weak scaling input is 64KB per core (horizontal line above bars corresponding to the secondary y-axis). The primary y-axis indicates wall-clock time of the benchmark run for different numbers of threads/cores on the x-axis.

NoCMsg (left bars) is roughly at par with shared memory (right bars) up to 4 processors but then significantly outperforms shared memory. This is due to dominating frequent collectives (alltoall[v]) relative to the computational part. We not only observe significantly higher performance but also lower perturbation of NoCMsg starting at just 8 processors. The execution time under OpenMP increases quadratically while that under NoCMsg remains close to linear as the processor count increases. At 32 cores, we observe a speedup or more than 12x.

Results for the code LU and SP and depicted in Figures 9(b) and 9(c). Both solve non-linear partial differential equations using standard solver techniques. In both benchmarks, the weak scaling

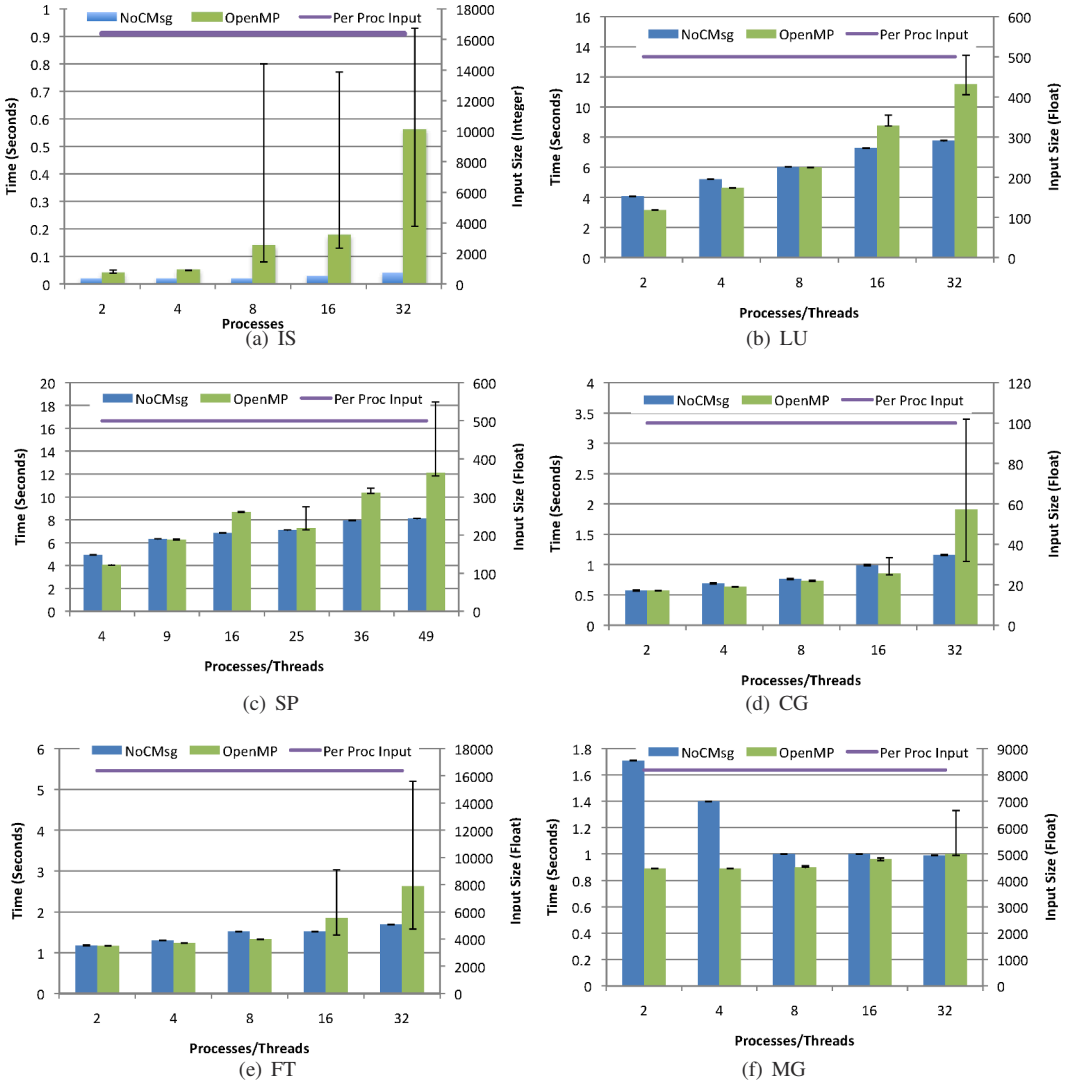


Fig. 9. NPB Weak Scaling Results

input is 4KB per core (see horizontal line corresponding to the secondary y-axis). Shared memory (right bars) provides faster performance than message passing (left bars) for low core counts. This is due to the fact that the shared memory network has twice the bandwidth of the UDN (for messages). At 16 cores, inter-processor communication and L3 contention start to hurt performance due to perturbation, indicated by the range of execution times depicted through the error bars. For LU at 32 cores, perturbation becomes more frequent. For SP at 49 cores, the worst measured perturbation is almost 50% greater than the average performance. The perturbation shown across all of these results is caused by increased wait times for shared memory accesses as inter-processor communication increases with the core count. Around global synchronization via collectives, e.g., barriers, this ultimately results in unbalanced computation and idle cores.

Experimental results for CG and FT are depicted in Figures 9(d) and 9(e). CG estimates eigenvalues using the conjugate gradient method. FT is a Fast Fourier Transform solver for partial dif-

ferential equations. The results for both codes are similar to those of LU and SP. However, CG and FT exhibit less computation and more inter-processor communication. Both benchmarks show that inter-processor communication eventually dominates results under core scaling resulting in considerably fluctuating time perturbation, even though a significant amount of computational power is expended on software emulation of floating point operations. OpenMP thus shows significantly worse performance and larger perturbation (error bars) for higher core counts. At 16 and 32 cores, Perturbation from L3 contention in both benchmarks becomes dominant.

Results for MG, a multigrid approximation benchmark for discrete Poisson equations, are depicted in Figure 9(f). MG is the only benchmark without enough inter-processor communication to generate an effect on performance. This benchmark was extremely limited in sizes due to a communication pattern that grew with the number of processes. It is also an extremely memory intensive benchmark resulting in large performance benefits of OpenMP over NoCMsg at two cores. But these benefits rapidly diminish at larger core counts. Once again, this benchmark shows a trend toward high perturbation under OpenMP with increasing core count. This indicates that subsequent increases in process/thread count beyond 32 might lead to decreased performance for MG, just as in the other NPB codes. Unfortunately, due to hardware limitations and power of two constraint in core counts of the MG code, we were unable to test at 64 processes/threads. Figure 10 shows a break down of the computation and communication as lower/upper part, respectively, of stacked bars. Benefits of message passing for larger core counts are dominated by savings in communication time for all NAS benchmarks.

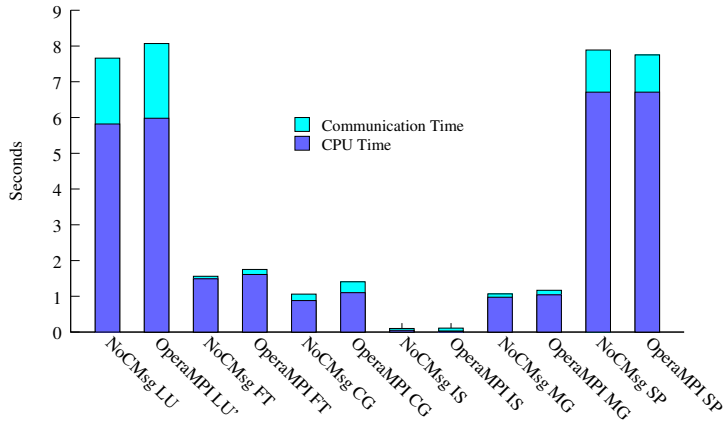


Fig. 10. NPB Code: NoCMsg vs. OperaMPI over 32 Processors

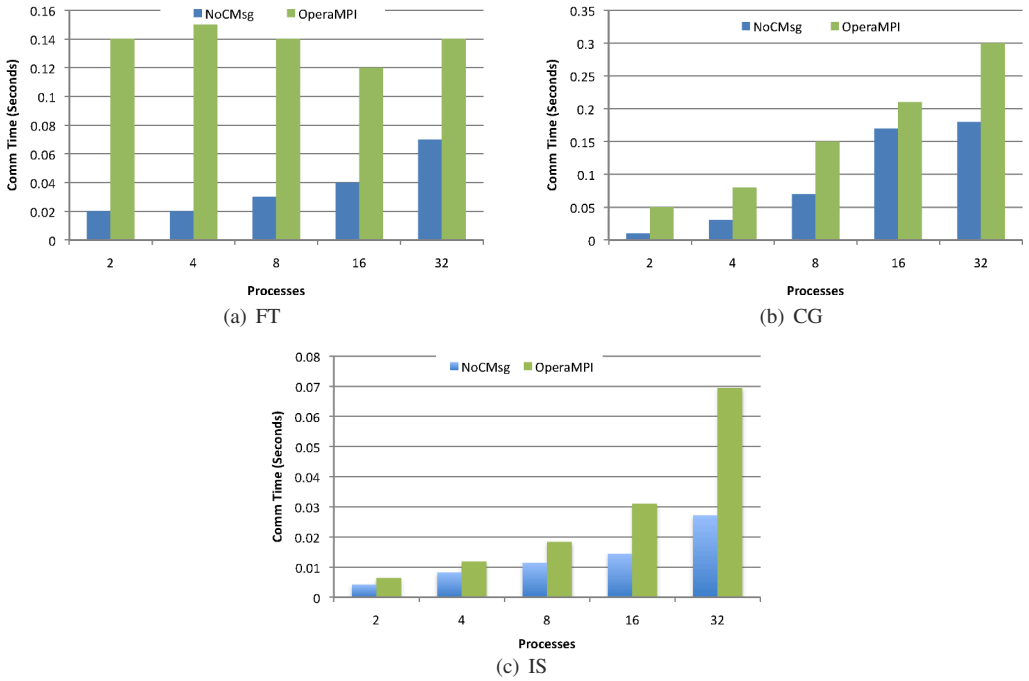
The potential of this architecture for other codes (NPB and beyond) is underlined by the performance differences between the NPB floating-point codes and the integer code IS. If a pipelined floating point unit were added, the performance of these benchmarks would increase significantly creating an even wider gap between OpenMP and NoCMsg as communication would become more dominant relative to computation. This hypothesis is confirmed in experiments in Section 6.5.

6.3. Flow Control Elimination

We next determine easily identifiable coding patterns, mostly inside of collectives utilized by the NPB codes, that can be subjected to the elimination of flow control. Initial findings indicate that while our flow-control method is portable, synchronization requirements within the MPI specification coupled with flow control resulted in NoCMsg and the interrupt-based OperaMPI to perform

at par for virtually all of the benchmarks. However, as detailed in the design section, the implementation of collectives in the runtime and application-side point-to-point communication provide opportunities to relax synchronization constraints by employing flow-control free communication.

Figures 11(a), 11(b), and 11(c) show the benchmark results just for the communication time in seconds (y-axis) for varying numbers of cores (x-axis) of FT, CG and IS after varying amounts of flow control were removed in a safe/conservative manner (cf. design section). The primary communication in FT is an alltoall collective. Such collectives allow elimination of flow control since all processes participate. After eliminating flow control, significant improvements to the communication performance of NoCMsg (left bar) were observed compared to OperaMPI (right bar) as seen in Figure 11(a). The primary reason for the scalability of NoCMsg is that the minimum cost transfer is very small for flow-control free communication (on the order of just a few cycles). Due to interrupts and protocol messages, OperaMPI incurs much higher overheads (factor $7X - 8X$).



(a) FT (b) CG

(c) IS

Fig. 11. NoCMsg vs. OperaMPI

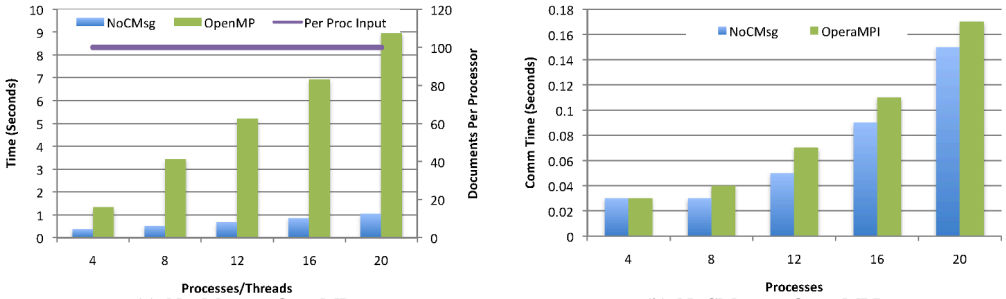
Communication time results for CG are shown in Figure 11(b). CG has several regions where synchronized MPI communication can be replaced with flow-control free communication. Since CG exclusively transfers data as a series of exchanges, it can guarantee that flow control free communication can be utilized, i.e., message ordering is guaranteed due to the application and NoC characteristics. By replacing these regions with flow-control free exchanges, improvements up to 40% are observed for NoCMsg at 32 processes. Notice that there is a synchronization requirement in CG when transitioning from 8 to 16 processes due to a changing communication pattern resulting in a significant increase in communication cost due to additional synchronization messages. Communication times stabilize again from 16 to 32 processors.

Figure 11(c) depicts results for IS. This benchmark features several patterns where flow control can be reduced without major modification to the application. The most significant one is in the implementation of the alltoallv collective. This function represents the majority of communication

in IS. At 32 processes, flow control elimination results in a 62% improvement in communication performance.

The communication patterns and use of collectives provided limited opportunities to eliminate flow control for the remaining NPB codes. Their performance behavior is dominated by MPI synchronization and flow control. Hence, we observe equivalent communication times for OperaMPI and NoCMsg for SP, LU, and MG. Due to that fact, we omit figures.

Another integer code was also subjection to evaluation, namely TF*IDF, a document classification technique to identify important terms over large sets of documents. TF*IDF is broken into two separate algorithms. TF (term-frequency) classifies unique terms and their occurrence frequencies on a per-file basis. IDF (inverse document frequency) combines TF data and accounts for term frequencies over the full set of documents. This problem is traditionally used in data mining. Two challenges in this problem are the large amount of required dynamic memory allocation and the reduction of IDF data in a parallel implementation.



(a) NocMsg vs. OpenMP (b) NoCMsg vs. OperaMPI

Fig. 12. Integer Application Benchmark TF*IDF

In our first TF*IDF experiment, we compared the wall-clock time for NoCMsg to OpenMP (see Figure 12(a)). We observe a disparity between performance that is almost a factor of 9X at 20 processes. This is primarily due to the required synchronization for heap allocation (C++ new) of STL calls for OpenMP. Heap allocation is protected by a lock to ensure thread safety. This lock contention results in inferior scalability for OpenMP due to increasing number of threads contending for the lock by spinning on shared memory inflicting high coherence protocol traffic. NoCMsg does not experience this problem since it features separate address spaces under a distributed execution paradigm.

By pre-allocating heap data at initialization time (similar to NPB codes), the OpenMP problem could be addressed algorithmically. But the TF*IDF algorithm does not adhere itself to pre-allocated data as data structures are dynamically determined and allocated, which is common for many C++/STL codes. One could implement private heaps for the TF calculation, yet would have to switch to global ones for IDF, where the problem remains. We did not go this route as we wanted to assess the benefits of TF*IDF without excessive changes to the system libraries or application.

In another TF*IDF experiment, the communication costs of NoCMsg and OperaMPI are compared (see Figure 12(b)). Since TF*IDF largely works on map-type data of terms and frequencies, data must be serialized for messaging. This communication is structured as a tree-based reduction where flow control is not necessary. This is largely responsible for the 12% improvement of NoCMsg at 20 processes.

6.4. Strong Scaling Results

We further conducted experiments under strong scaling for the NAS PB codes with input class A. The choice of input class ensures that inputs fit into L2 cache for the smallest number of cores. As

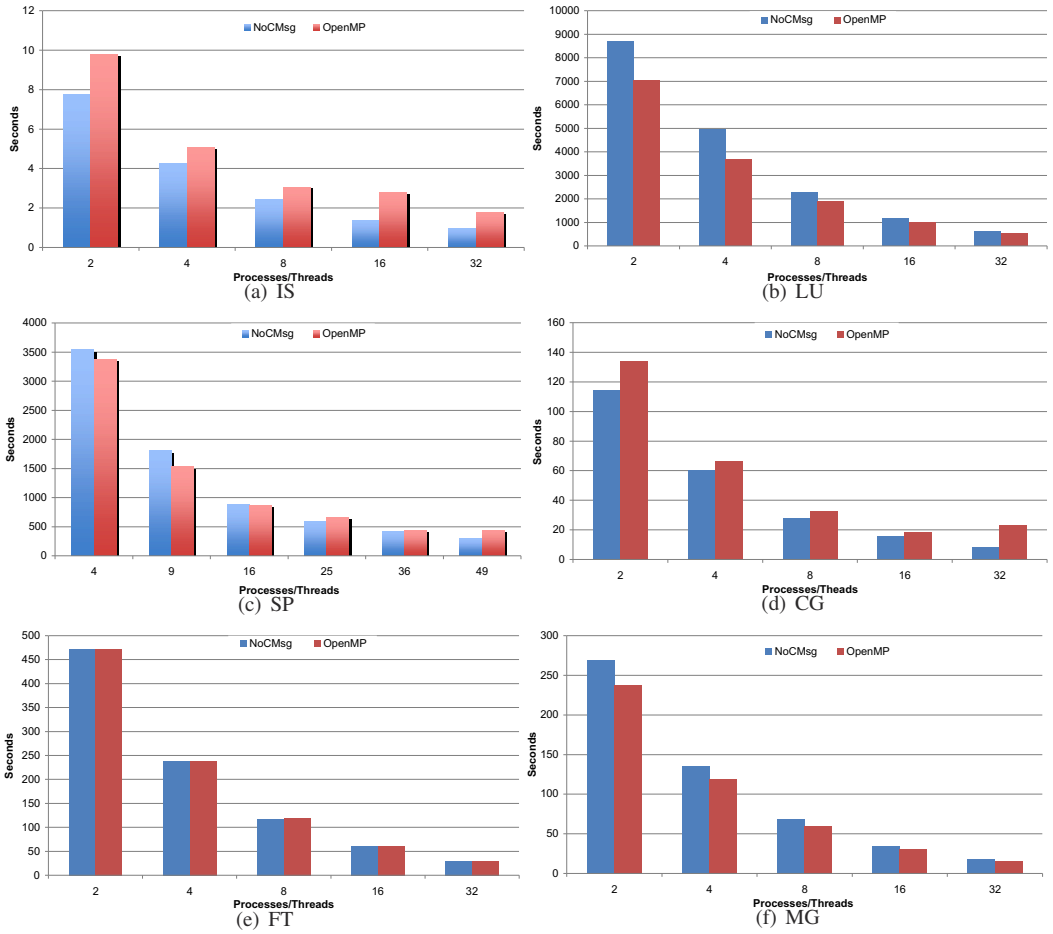


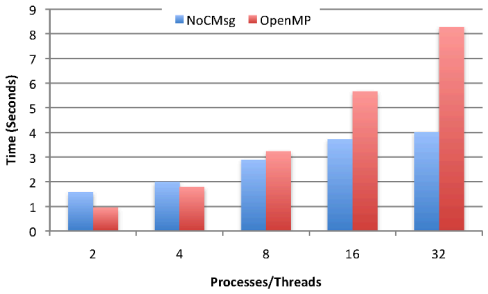
Fig. 13. NPB Strong Scaling Results for Input Class A

the number of cores P is scaled up, inputs only require a fraction of the L2 inversely proportional to P. Results for IS are depicted in Figure 13(a) reporting execution times in seconds (x-axis) for varying number of cores (y-axis). We observe superior performance for NoCMsg (left bar) compared to OpenMP (right bar), a trend that increases with the number of cores from 20% to 45%. Similarly, CG (Figure 13(d)) results in performance benefits of NoCMsg ranging from 15% to 65%. FT (Figure 13(e)) shows insignificant differences between NoCMsg and OpenMP. For SP (Figure 13(c)), the performance differences of NoCMsg range from a 5% penalty to a 33% benefit. Conversely, LU (Figure 13(d)) results in performance penalties of NoCMsg ranging from 23% to 11%. And for MG (Figure 13(f)), the performance penalty of NoCMsg is nearly constant around 13%.

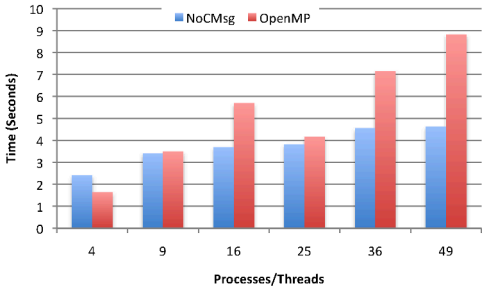
Overall, strong scaling results are inconclusive in terms of generalization as the behavior depends on the amount and size of message exchanges as well as the messaging type (point-to-point vs. collective). More significantly, as the number of cores increases, the remaining execution time under strong scaling becomes ever smaller, and L2 caches are no longer fully utilized. This reflects a poor utilization of any architecture in terms parallelization as performance scaling provides diminishing returns due to Amdahl's law, e.g., after 16 cores for SP. For this reason, the remainder of the experiments will focus on weak scaling again.

Operation	Native	Soft
Add Opera	20	105
Mult Opera	21	112
Add TilePro	14	74
Mult TilePro	17	91

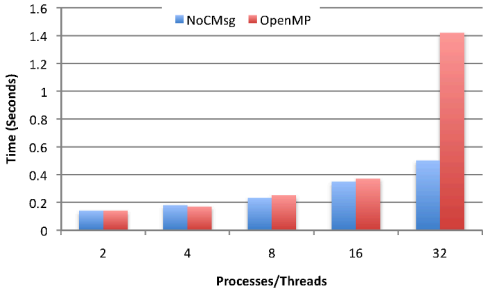
(a) Floating Point Performance Metrics



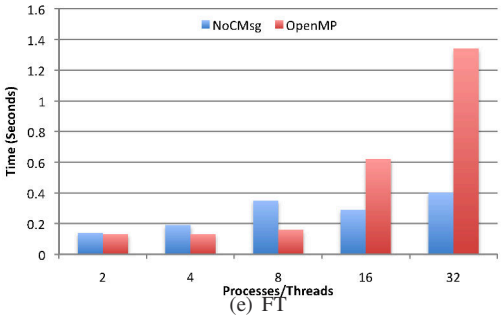
(b) LU



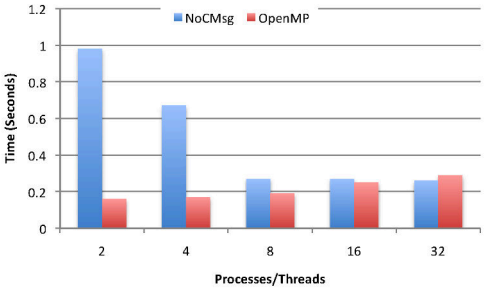
(c) SP



(d) CG



(e) FT



(f) MG

Fig. 14. NPB Weak Scaling FPU Reduction Results

6.5. FPU Results

We also conducted experiments to evaluate the benefits of NoCMsg in hardware environments supporting double-precision floating point units (FPUs). The Maestro [10] project features a radiation hardened 49-core Tilera board with integrated FPU. Unfortunately, the Maestro board does not support software for a Fortran compiler, and the Tilera compiler is limited to software floating point emulation.

Due to these limitations, we performed experiments to approximate the effect of native double-precision FPUs on the TilePro 64 from our original results. We evaluated this by using dual-loop timing for native and soft floating point operations on the Maestro board to ascertain the performance disparity between them. We then used cross-product ratios to extrapolate the potential performance of FPU operations for a 700MHz TilePro 64. The results shown in Table 14(a) depict the cycle latencies of soft and native floating point operations on both the TilePro and the Maestro boards. The difference between soft-float cycles of Maestro and TilePro is due to different compiler versions, where TilePro generates slightly improved code. We were unable to eliminate these

differences due to the age of the compiler's only available version for the Maestro platform and a variance in executable format prohibiting us from evaluating code on the Maestro board using the newer TilePro compiler.

We next extracted hardware performance counters from NPB floating-point codes using likwid [2] on the x86 architecture to determine the number of floating point adds/multiplies executed per benchmark. This allowed us to assess the total cost difference for the computational NPB kernels and to determine the approximate performance difference.

Recall that under FPU emulation both SP and LU results were dominated by computational overhead. With FPU hardware, computational overhead was significantly reduced as shown by Figures 14(b) and 14(c) depicting the extrapolated results. This indicates the potential of NoCMsg for even more significant performance benefits over OpenMP.

CG (Figure 14(d)) and FT (Figure 14(e)) show similar scalability trends with performance benefits increasing to similarly high levels as the integer-based IS benchmark shown in Figure 9(a). In these results, CG's performance is increasing by 64% over OpenMP on hardware FPUs at 32 cores.

MG (Figure 14(f)) again shows interesting scaling performance. Due to communication pattern changes that occur as the problem size changes, OpenMP shows better performance than NoCMsg at small core counts. But the trend under weak scaling shows that NoCMsg outperforms OpenMP at 32 cores with FPU hardware, albeit by a small margin.

The most important take-away from these results is that NoCMsg sees larger performance gains over OpenMP as computational overhead decreases. This is consistent with our original experiments from the previous section where the integer workload of IS showed the most significant performance gains. This was largely due to IS being heavily influenced by IPC over computational overhead and the fact that IS was an integer-only benchmark without software emulation. It is also important to remember that the results presented in this section are an approximation and do not account for several important changes that would occur if the computational overhead was reduced. This includes increased contention on IPC pathways and controllers due fewer stall cycles.

7. RELATED WORK

Singh et al. [29] and Suh et al. [30] report the performance of FFTW and FFT/CRBlaster, respectively, on the Tilera Maestro platform. Serres et al. [28] report on the performance of UPC implemented over GasNet plus Pthreads/OperaMPI on a TilePro 64. UPC versions of NPB 2.2 under class A show better performance for Pthreads than MPI for benchmarks with significant communication components under strong scaling experiments (input class A). Martin et al. [22] report on techniques for integrating coherence state and semantics into shared caches to increase scalability. However, the authors acknowledge that these techniques will not improve scalability for all algorithms and that techniques such as message passing are here to stay. Additionally, this paper focuses solely on coherence with little mention of additional performance degradation due to NUMA/NUCA architectures integrated within many-cores. We compare shared memory against message passing and, in contrast to this past work, assess the effect of enabling coherence for the former while disabling it for the latter. Furthermore, we conduct weak scaling experiments, which reveal the potential and limitations of multicore architectures in terms of parallelization speedup in scenarios where on-chip caches are fully utilized. Finally, we determine the benefits of message passing at the lowest possible level in software instead of multi-layer protocols.

Prior work compared MPI and OpenMP for shared-memory multiprocessors [21], which does not cover on-chip NoCs of multicores, which is our focus. Clauss et al. [11] performed a comparison of shared memory and message passing for the Intel SCC [23]. Since the SCC does not provide memory coherent at the DRAM level, they have to rewrite load and store instructions as put() and get() directives that operate on the message-passing buffers, which are small SRAM buffers (8KB) that can be configured to be coherent, and then flush L1 cache lines explicitly. This approach results in slightly better performance of shared memory than message passing (via RCCE [31]) for the Jacobi solver at higher core counts. Our study is based on native shared memory at the DRAM level that does not require load/stores to be rewritten, nor is it restricted to 8KB shared data. We recently

ported NoCMsg to the Intel SCC [25] and found that our flow control elimination in conjunction with contention-free communication patterns provide similar performance improvements on the Intel SCC compared to the Tilera [36; 34].

NoCMsg follows addresses scalability problems via message passing, not just for shared-memory multiprocessors as the Multikernel [9] but for multicores in our case. It takes ideas like NoC-level message passing from Factored Operating Systems [32] to another level in supporting low-level NoCMsg as a basis for scalable NoC communication without deadlocks.

Flow control elimination is utilized by iWarp[3], a protocol that works at an OS level to reduce the overhead of TCP. The major difference is that NoCMsg operates directly at the hardware level without OS intervention, that NoCMsg is a library, not a protocol, and that NoCMsg benefits directly from application-level flow elimination.

8. CONCLUSION

NoCMsg is a specialized MPI library with a novel angle in that it takes advantage of network-on-chip architectures to improve scalability and performance. In experiments, performance benefits of up to 86% were observed over a base MPI implementation on the Tilera platform. More significantly, NoCMsg reaches performance benefits of up to 93% over shared memory abstractions, such as OpenMP, on this platform. NoCMsg improves scalability by providing a polling-based message-passing implementation. Our results indicate that as processor counts and problem sizes increase, even on-chip solutions that employ shared memory are not as scalable as their message passing counterpart on the Tilera platform.

We further develop methods for synchronization and flow control that guarantee deadlock free communication, both of which are essential to communication performance. We demonstrate that communication analysis and pattern-based code replacement around collectives and other code regions of benchmarks allow the elimination flow control in a safe but conservative manner.

These contributions provide significant benefits in performance in terms of wall-clock time, particularly with respect to communication overheads. These benefits materialize in particular under weak scaling when caches are fully utilized while strong scaling results were mixed. For numerical codes, weak scaling is particularly attractive when fully pipelined native floating point execution is supported.

Overall, this study shows the potential for message passing for the Tilera architecture. These results generalize to similar NoC-based platforms, most notably the Intel SCC [25]. It indicates that shared memory scales up to about 16 cores while message passing performs well beyond that threshold. While the concrete threshold of cores is platform dependent, the NoC contention problem is universal for meshes. We further hypothesize that hybrid OpenMP programs with 16 threads combined with message passing between OpenMP regions may be a viable solution, but ongoing experiments are beyond the scope of this paper.

REFERENCES

Adapteva processor family. www.adapteva.com/products/silicon-devices/e16g301/.

Lightweight performance tools. http://code.google.com/p/likwid/.

A remote direct memory access protocol specification. tools.ietf.org/html/rfc5040.

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

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

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

Tilera user architecture reference. www.tilera.com.

D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The Int'l Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new os architecture for scalable multicore systems. In *Symposium on Operating Systems Principles*, pages 29–44, 2009.

M. Cabanas-Holmen, E. H. Cannon, C. Neathery, R. Brees, B. Buchanan, A. Amort, and A. Kleinosowski. Maestro processor single event error analysis.

C. Clauss, S. Lankes, P. Reble, and T. Bemmerl. Evaluation and improvements of programming models for the intel scc many-core processor. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pages 525–532, July 2011.

B. D. de Dinechina, P. G. de Massasa, G. Lagera, C. Legera, B. Orgogozoa, J. Reyberta, and T. Strudela. A distributed run-time environment for the kalray mppar-256 integrated manycore processor. In *International Conference on Computational Science*, page 16541663, 2013.

J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *Symposium on Operating Systems Design and Implementation*, pages 137–150, 2004.

D.-R. Fan, N. Yuan, J.-C. Zhang, Y.-B. Zhou, W. Lin, F.-L. Song, X.-C. Ye, H. Huang, L. Yu, G.-P. Long, H. Zhang, and L. Liu. Godson-t: An efficient many-core architecture for parallel program executions. *Journal of Computer Science and Technology*, 24:1061–1073, 2009. 10.1007/s11390-009-9295-3.

M. P. I. Forum. MPI: A Message-Passing Interface Standard Version 3.0, 09 2012.

W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, Sept. 1996.

J. L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, May 1988.

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.

S. Kato, K. Lakshmanan, Y. Ishikawa, and R. R. Rajkumar. Resource sharing in gpu-accelerated windowing systems. In *IEEE Real-Time Embedded Technology and Applications Symposium*, pages 191–200, 2011.

S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, , and R. Rajkumar. Rgem: A responsive gpgpu execution model for runtime engines. In *IEEE Real-Time Systems Symposium*, pages 57–66, 2011.

G. Krawezik and F. Cappello. Performance comparison of mpi and openmp on shared memory multiprocessors: Research articles. *Concurr. Comput. : Pract. Exper.*, 18(1):29–61, Jan. 2006.

M. M. K. Martin, M. D. Hill, and D. J. Sorin. Why on-chip cache coherence is here to stay. *Commun. ACM*, 55(7):78–89, July 2012.

T. G. 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 *Supercomputing*, pages 1–11, 2010.

A. Moshovos, G. Memik, A. Choudhary, and B. Falsafi. Jetty: Filtering snoops for reduced energy consumption in smp servers. In *High Performance Computer Architecture*, pages 85–96, 2001.

O. Patil. Effcient and lightweigth inter-process collective operations for massive multi-core architectures. Master's thesis, North Carolina State University, June 2014.

T. Ropars, T. V. Martsinkevich, A. Guermouche, A. Schiper, and F. Cappello. Spbc: Leveraging the characteristics of mpi hpc applications for scalable checkpointing. In *Supercomputing*, 2013.

K. Sankaralingam, R. Nagarajan, P. Gratz, R. Desikan, D. Gulati, H. Hanson, C. Kim, H. Liu, N. Ranganathan, S. Sethumadhavan, S. Sharif, P. Shivakumar, W. Yoder, R. McDonald, S. Keckler, and D. Burger. The distributed microarchitecture of the trips prototype processor. In *Int'l Symposium on Microarchitecture*, Nov. 2006.

O. Serres, A. Anbar, S. Merchant, and T. El-Ghazawi. Experiences with upc on tile-64 processor. In *2011 IEEE Aerospace Conference*, pages 1–9, 2011.

K. Singh, J. P. Walters, J. Hestness, J. Suh, C. M. Rogers, and S. P. Crago. Fftw and complex ambiguity function performance on the maestro processor. In *IEEE Aerospace Conference*, pages 1–8, 2011.

J. Suh, K. Mighell, D.-I. Kang, and S. Crago. Implementation of fft and crblaster on the maestro processor. In *IEEE Aerospace Conference*, pages 1–6, march 2012.

R. F. van der Wijngaart, T. G. Mattson, and W. Haas. Light-weight communications on intel's single-chip cloud computer processor. *SIGOPS Oper. Syst. Rev.*, 45(1):73–83, Feb. 2011.

D. Wentzlaff and A. Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *SIGOPS Oper. Syst. Rev.*, 43:76–85, April 2009.

D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. B. III, and A. Agarwal. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27:15–31, 2007.

K. Yagna. Efficient collective communication for multi-core noc interconnects. Master's thesis, North Carolina State University, May 2013.

Y. Zhang, F. Mueller, X. Cui, and T. Potok. Large-scale multi-dimensional document clustering on gpu clusters. In *Int'l Parallel and Distributed Processing Symposium*, Apr. 2010.

C. Zimmer and F. Mueller. Nocmsg: Scalable noc-based message passing. In *International Symposium on Cluster Computing and the Grid (CCGRID)*, page (accepted), 2014.