NoCMsg: Scalable NoC-Based Message Passing

Christopher Zimmer, Frank Mueller
North Carolina State University, Raleigh, NC, USA
mueller@cs.ncsu.edu

***Abstract*—Current processor design with ever more cores may ensure that theoretical compute performance still follows past increases (resting from Moore's law), but they also increasingly present a challenge to hardware and software alike. As the core count increases, the network-on-chip (NoC) topology has changed from buses over rings and fully connected meshes to 2D meshes. The question is which programming paradigm provides the scalability needed to ensure performance is close to theoretical peak, where 2D meshes provide the most scalable design to date.**

This work contributes NoCMsg, a low-level message passing abstraction over NoCs. NoCMsg is specifically designed for large core counts in 2D meshes. Its design ensures deadlock free messaging for wormhole Manhattan-path routing over the NoC. 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. Results further demonstrate the potential of NoC messaging to outperform shared memory abstractions by up to 93% as core counts and inter-process communication increase, i.e., we observe that shared memory scales up to about 16 cores while message passing performs well beyond that threshold on this platform. To the best of our knowledge, this is the first head-on comparison of shared memory and advanced message passing specifically designed for NoCs on an actual hardware platform with larger core counts on a single socket.

I. Introduction

The future of computing is rapidly changing as multicore processors are becoming ubiquitous. While multicores 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. This presents 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 been shown to fall short in scaling, 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 mesh interconnects (e.g., Hypertransport, Quick Path Interconnect) with currently up to 16 cores per socket and, for high core counts, tile-based architectures with 2D meshed network-on-chip (NoC) interconnects [4], [5], [23], [18], [3].

This work was supported in part by NSF grants CNS-0905181, CNS-0958311, CNS-1239246 and a subcontract from SecurBoration.

But mesh-based systems with MESI-style coherence protocols enhanced by coherence filters [17] may limit scalability in the number of cores. E.g., the multikernel (aka. Barrelfish) follows a distributed kernel paradigm that employs messages in an off-chip mesh interconnect of Hypertransport links [8]. It shows that messaging can outperform shared memory for configurations of just eight processors.

Contributions: This work assesses whether large core counts with 2D mesh NoCs scale better in performance under a shared memory protocol or under NoC-based message passing. It further identifies flow control as a major hurdle in gleaning additional performance from message passing. It presents techniques within the implementation of an MPI-like runtime system [11] for NoCs, removes flow control where it is not needed and assesses the performance impact.

The paper details the design and implementation of NoCMsg, a low-level message passing abstraction over NoCs. The design reduces the number of software layers compared to prior work. 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 [6] 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. More significantly, NoCMsg is able to relax communication constraints by exploiting pattern-based communication common in MPI runtime systems to identify areas in which flow control is unnecessary.

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. NoCMsg is also significantly more scalable than prior messaging techniques, as shown for a subset of the NAS Parallel Benchmarks [7].

Another contribution is the comparison between message passing and shared memory on the same NoC architecture, where the former is supported in firmware while the latter is implemented by NoCMsg in software. Experiments demonstrate the potential of NoC messaging to outperform shared memory abstractions, such as OpenMP, up to 93% beyond 16 cores. To the best of our knowledge, this is the first head-on comparison of shared memory and advanced message passing specifically designed for NoCs on an actual single socket hardware platform with larger core counts.

II. Background

NoCs utilize traditional network communication for inter-processor communication. Data is transferred between cores as messages. These 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 [10], [5]. Contemporary NoCs feature increasing throughput in communication. This becomes feasible due to simplistic routing protocols to single cycle per-flit transfer latencies (in the absence of contention, as we will see).

Unfortunately, without more advanced hardware protocols or structured software libraries, bare-metal message-passing can lead to deadlocks. 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.

As an example of a deadlock, 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. However, when the buffer becomes full the switching network must wait until flits are drained before transferring any remaining flits. Exchanging contiguous buffers of size greater than 127 words will result in deadlock for the two cores and any messages that need to traverse the route between these two cores.

Solutions to avoid blocking commonly involve interrupt-based channel creation. Tilera's iLib communication library uses channel creation through protocol messages. Once a channel is created, the buffer 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 so that the sending process can drain pending receives before continuing to send packets.

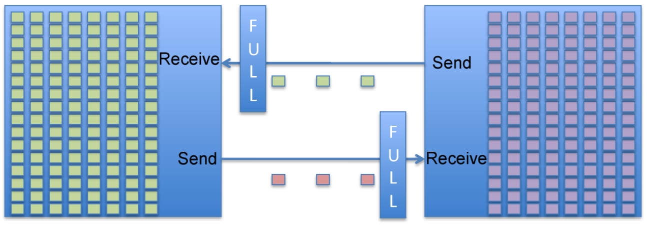


Figure 1. Message Passing Deadlock

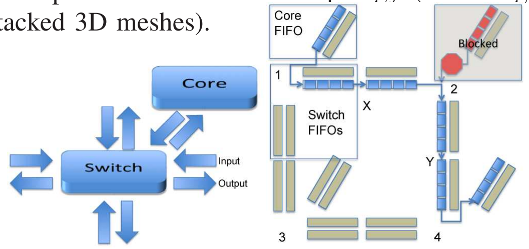
III. Design

The objective of our work is to design a close to bare metal NoC-level messaging protocol without deadlocks and with reduced flow control to lower overheads.

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. We describe constraints of such an architecture next and discuss its relation to our NoC message layer design.

A. *NoC Architecture*

NoC architectures use the network-on-chip to replace the conventional system bus or other topologies of connecting cores. This means that all memory, messaging, and 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 [5] five networks. The Intel SCC [3] only has a single network and does not natively support coherence over this network, 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. Yet, our contributions to flow control operate irrespective of the switch topology (extending to future stacked 3D meshes).



(a) Core - Switch Topology (b) Path-Based Back Pressure

Figure 2. NoC Routing over Switches and Links

B. *Cores*

A 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.

C. *Switches*

Switches are generally composed of multiple sets of input and output queues attached to a crossbar switch. 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. 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. This is shown in Figure 2(b) where core 1 is sending 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 until no more data can be transferred.

D. *Credit Monitoring: Back Pressure Check*

We assume that output queues 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 back pressure can only be resolved if node 4 actively drains the network to free up space within its hardware buffers. The second type of back pressure in Figure 2(b) occurs when a switch is unable to route a packet due to an open wormhole path. The message sent between cores 1 and 4 is blocking a message sent from cores 2 to 4. Here, blocking can only be resolved by ensuring the message between 1 and 4 completes.

Our design addresses this in a generalized fashion via a polling work loop that cycles between computation, sending, and receiving of data. The underlying credit checking scheme is specific to Tilera, but any other resource management could be used in its place, e.g., co-processor failure registers for non-blocking transfers.

IV. Implementation

Our high-level design has been implemented in NoCMsg on the Tilera platform, yet our general design from Section III extends to any 2D mesh NoC architectures. 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 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 architectural level instead of utilizing operating system / MPI runtime capabilities. (The sync parameter is explain next.)

```
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
```

A. *Point-to-Point Messages*

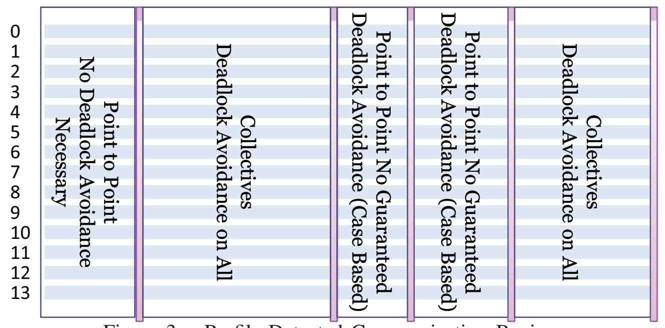
The basis of NoCMsg is factored around asynchronous work loops during which sends and receives are issued based on the availability of resources. These asynchronous messages provide building blocks for synchronous communication, collective operations, and barriers. As previously described, point-to-point messages are subject to deadlock in the absence of flow control due to the nature of the NoC switching architecture. As such, we employ back pressure monitoring to ensure absence of deadlock for any message transactions. This is ensured by implementing a work loo broken into two alternating operations for asynchronou communication.

(1) Trysend implements conditional sending of a messag During the send of a packet of flits, the output queue' available credits are inspected. We then place as many flit in the output queue as credits are available, i.e., credits ar queried for each and every transfer. If no credits are lef control is returned to the work loop.

(2) Tryreceive implements conditional data reception. Th MPI ready-send specification for point-to-point sends an receives requires synchronization between any send/receiv pairs [11]. For synchronous communication, this means send will not be completed until the sender has seen a acknowledgment from the receiver. In asynchronous com munication, send and receive will initiate communicatio yet may return from the API call before the operatio.. completes. Should a matching sender-side MPI_Wait() call follow, then a similar acknowledgment has to first be seen by the sender. An MPI_Wait() after an asynchronous receive, on the other hand, 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. When MPI_Wait calls are present, this can be exploited for flow control elimination.

NoCMsg introduces so-called synchronous non-flow controlled messages that diverge from MPI in terms of their 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. 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.

The implementation of non-flow controlled transfers requires a small setup overhead to synchronize the sender and receiver if the buffer is larger than a packet. 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 to 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 utilized but are beyond the scope of this paper.)



0
1
2
3
4
5
6
7
8
9
10
11
12
13
Point to Point No Deadlock Avoidance Necessary
Collectives Deadlock Avoidance on All
Point to Point No Guaranteed Deadlock Avoidance (Case Based)
Point to Point No Guaranteed Deadlock Avoidance (Case Based)
Collectives Deadlock Avoidance on All

Figure 3. Profile Detected Communication Regions

The challenge is to identify if flow control can be safely removed in certain message transfers. To this end, we profiled 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 region, where collective operations and barriers mark region boundaries. This data is subsequently analyzed to detect communication patterns that inhibit flow-control elimination (e.g., due to cycles). The approach is conservative in that regions that *may* require flow control are excluded when in question. For example, 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, will be excluded. On the positive side, special patterns, such as pairwise exchanges (send/receive pairs) are detected and subsequently optimized via ordering by rank to eliminate flow control. 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. Regions of code are marked to indicate the type of communication they contain.

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

(a) Original Code (b) Flow Control Removed

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

A concrete example is given in Figure 4, which 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 as this could result in low-level deadlocks as described before.

B. Collectives

Collective operations offer significant opportunities to eliminate flow control since one can make safe assertions about the content of the network messages in flight at a given point in time for NoC communication. This assumes 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.

The first criterion for flow control elimination is that there is a single known sender or a single known receiver. Broadcast and reduction meet this criterion. The second criterion is the presence of synchronization prior to the collective and that no asynchronous communication is in flight. This guarantees absence of in-flight point-to-point messages before non-monitored message transfers are triggered.

Alltoall and alltoallv are most demanding (in terms of network contention) and allow 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. A single receiver is acquiring data from all cores at any given time in our design ensuring deadlock freedom due to the acyclic pattern.

C. Barriers

Our current implementation of collectives requires prior synchronization of execution for deadlock free communication. 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 VI).

D. Network Partitioning

Flow-control elimination should be considered in the context of network partitioning. The techniques discussed in this work assume run-to-completion tasks and absence of cross communication from outside task sets. This does not mean that outside task sets are unable to use these links, only that they cannot address messages to nodes within an external task set. However, utilization of these switches by outside task sets can create additional communication contention affecting performance. With this in mind, tasks are mapped within grids to reduce the possibility of perturbation of results by parallel task deployments in our experiments.

V. Framework

Experiments were conducted on a Tilera TilePro processor, namely a 700MHz 64-core version (TilePro 64) with floating point emulation in software [5]. 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 local L1 and L2 caches, where portions of the L2 caches of all cores can optionally be combined into a distributed (virtual) L3 cache. The L3 cache is directory 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, as will be shown.

Experiments comparing NoCMsg and OperaMPI were conducted under disabled cache coherence (hash-based distributed virtual L3 turned off). All messages are routed over the UDN. OperaMPI [14] implements the MPI 1.2 standard [12] 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 make 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 all-to-all communication, all-gather/scatters, reductions etc.

Experiments were conducted for the NAS Parallel Benchmark (NPB) codes [7] Version 3.3 for OpenMP, OperaMPI and NoCMsg. Inputs were modified to allow weak scaling [13] 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 the assessment of asymptotic behavior of multicores with near-perfect locality (e.g., for perfect multi-level tiling) instead of being skewed by off-chip memory bandwidth of a given architecture.

We also ported a term frequency/inverse document frequency (TF*IDF) benchmark for document clustering based on prior work [24], which follows a map-reduce paradigm [9]. Its inputs also follow the weak scaling paradigm for L2 resident data sets.

VI. Experimental Results

We evaluate NoCMsg by comparing shared memory and message passing using micro benchmarks and application benchmarks on the Tilera. We refer to *shared* **data** *memory* whenever we use the term shared memory here. Instructions have little to no impact on data for tested benchmarks since instruction cache are warmed up first.

A. Microbenchmarks

We compared message passing over the UDN with shared memory transfers over the coherence interconnect in a bandwidth micro-benchmark. Figure 5 indicates 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. 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.) The differences between shared memory and message passing become even more significant as the distance (hop count) between cores in the NoC increases and as NoC contention increases. *These results indicate that message passing has the potential to outperform shared memory transfers with superior scaling characteristics of the former over the latter.*

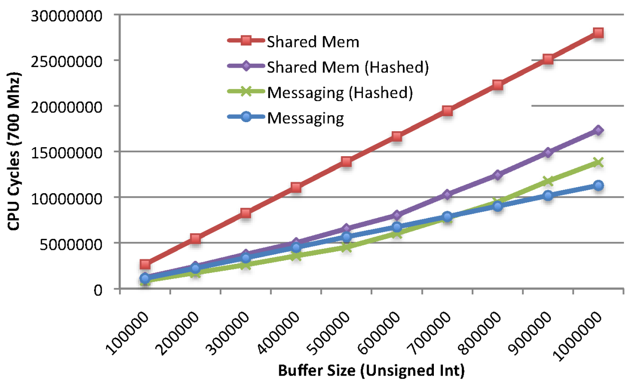


Figure 5. Shared Memory/Messages NoC Bandwidth

L3 hashing results in a uniformly distributed address space over the virtual L3 cache 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. *Furthermore, performance benefits come at the cost of jitter since accesses to distributed L3 have variable hop counts (NUCA) over the NoC.* Figure 6 shows the effects of data transfer in terms of jitter. In both message and shared memory transfers where home cache hashing is turned on. There is noticeable jitter even in the absence of contention due to additional tasks. In a shared-memory system design, both the operating system and the application increasingly suffer from such latencies as the core count increases. 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 to global synchronization (e.g., barriers). We term this effect *perturbation*, discussed it in the next experiments.

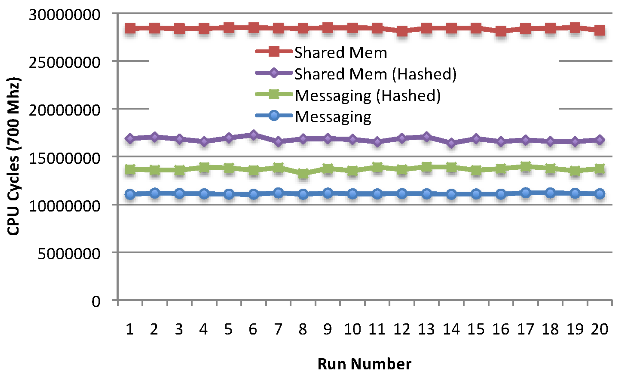
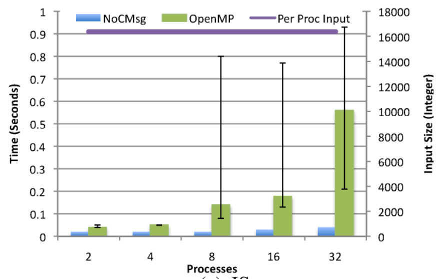


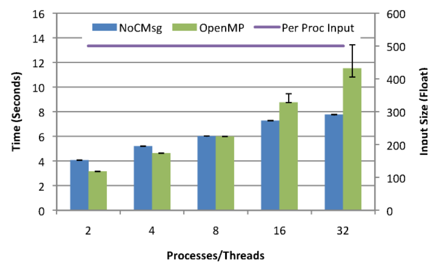
Figure 6. Shared Memory/Messages NoC Jitter [1MB Xfer]

B. NAS Parallel Benchmarks

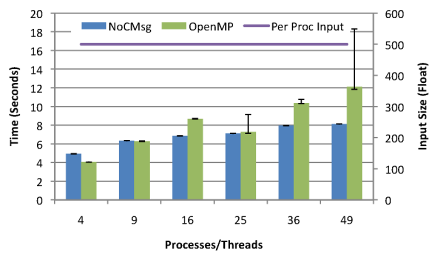
We chose NPB since OpenMP and MPI versions exist for each code, much in contrast to other parallel benchmarks that only provide shared memory codes. In contrast to NPB's default strong scaling inputs, we used our own weak scaling



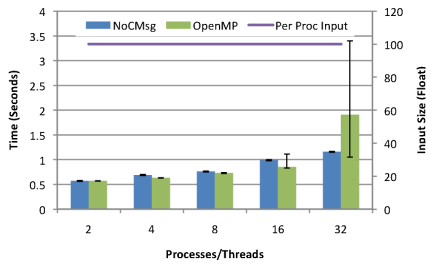
(a) IS



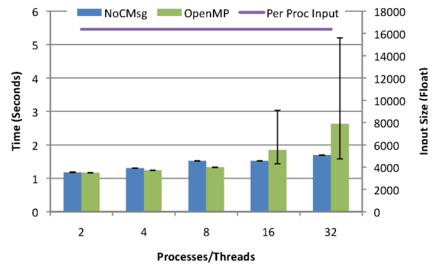
(b) LU



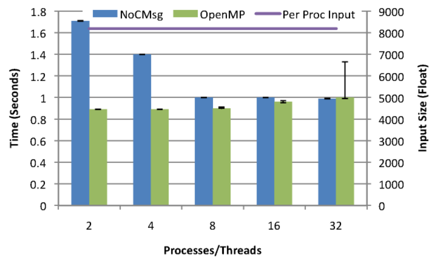
(c) SP



(d) CG



(e) FT



(f) MG

Figure 7. NPB Weak Scaling Results

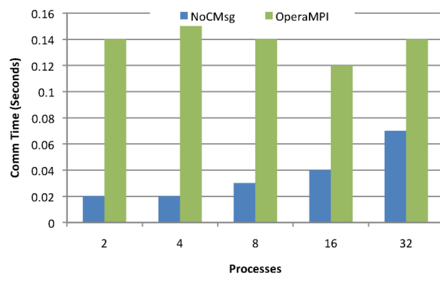
inputs [13] where the data set per core is of fixed size. This weak scaling input size is shown on the secondary y-axis in each of the following figures. Weak scaling ensures that the computational work per core remains the same as the number of cores cooperating in a parallel application is increased. 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 as discussed next.

Results for the integer bucket sort benchmark IS, the only integer benchmark for NPB, are depicted in Figure 7(a). 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. 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 with over a twelve-fold speedup at 32 cores.

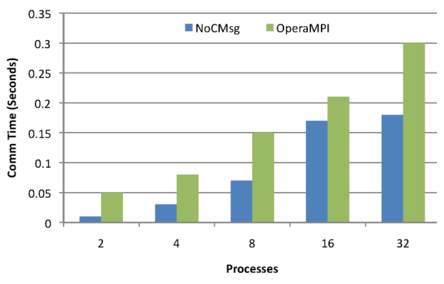
Figures 7(b) and 7(c) show results for the two NPB codes LU and SP. Both solve non-linear partial differential equations using standard solver techniques. In both benchmarks, the weak scaling input is 4KB per core (see horizontal line corresponding to the secondary y-axis). Shared memory (right bars) 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. It ultimately results in unbalanced computation and idle cores around global synchronization via collectives, e.g., barriers.

CG estimates eigenvalues using the conjugate gradient method. FT is a Fast Fourier Transform solver for partial differential equations. The results for CG and FT in Figures 7(d) and 7(e) 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. Perturbation from L3 contention in both benchmarks becomes dominant at 16 and 32 cores.

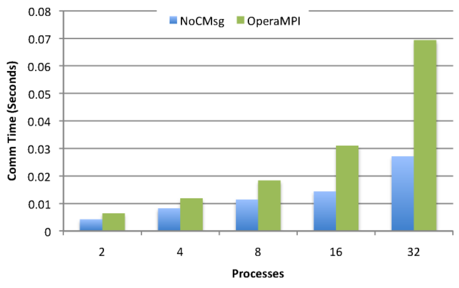
Figure 7(f) depicts the results for MG, a multigrid approximation benchmark for discrete Poisson equations. MG is the only benchmark without enough inter-processor communication to generate an effect on performance. This benchmark was extremely limited in sizes due to a communication pattern that grew with the number of processes. It is also an



(a) FT



(b) CG



(c) IS

Figure 8. NoCMsg vs. OperaMPI

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.

Benefits of message passing for larger core counts are dominated by savings in communication time for all NAS benchmarks, illustrated as stacked bars for computation and communication as lower/upper part, respectively, of the bars in Figure 9.

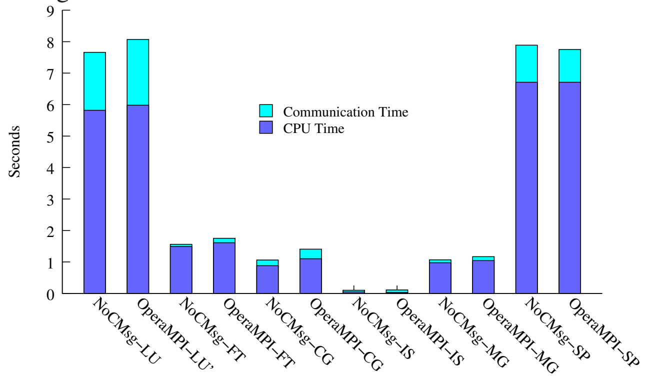


Figure 9. NPB Code: NoCMsg vs. OperaMPI over 32 Processors

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

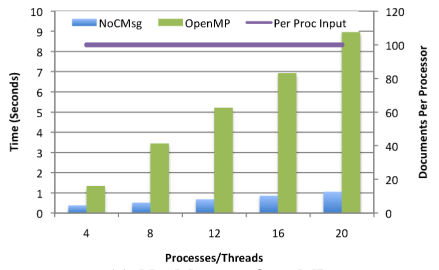
C. Flow Control Elimination

Our next set of experiments focuses on the elimination of flow control for easily identifiable coding patterns mostly inside of collectives utilized by the NPB codes. 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 8(a), 8(b), and 8(c) show the benchmark results just for the communication time of FT, CG and IS after varying amounts of flow control were removed in a safe/conservative manner (cf. design section).

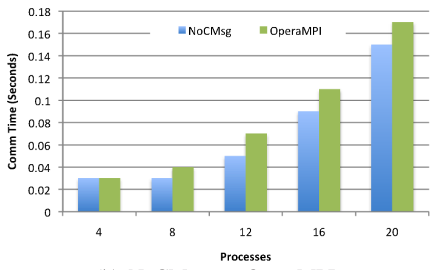
The primary communication in FT is an alltoall collective. Such collectives allow elimination of flow control since all processes participate. After eliminating flow control, significant improvements to the communication performance of NoCMsg were observed (see Figure 8(a)). The primary reason for the 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). OperaMPI incurs much higher overheads (factor $7X - 8X$) due to interrupts and protocol messages.

Figure 8(b) shows communication time results for CG. 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. From 16 to 32 processors, communication times stabilize again.

Results for IS are depicted in Figure 8(c). IS 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. Flow control elimination results in a 62% improvement in communication performance at 32 processes.



(a) NocMsg vs. OpenMP



(b) NoCMsg vs. OperaMPI

Figure 10. Integer Application Benchmark TF*IDF

For the remaining NPB codes, the communication patterns and use of collectives provided limited opportunities to eliminate flow control. 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 and omit figures due to that fact.

We also evaluated 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.

In our first TF*IDF experiment, we compared the wall-clock time for NoCMsg to OpenMP (see Figure 10(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 a distributed execution paradigm of separate address spaces.

The OpenMP problem could be addressed algorithmically by pre-allocating heap data at initialization time (similar to NPB codes). 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 application or system libraries.

Our second TF*IDF experiment compares the communication costs of NoCMsg and OperaMPI (see Figure 10(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.

VII. Related Work

Singh et al. [20] and Suh et al. [21] report the performance of FFTW and FFT/CRBlaster, respectively, on the Tilera Maestro platform. Serres et al. [19] 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. [16] 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 [15] but not for on-chip NoCs of multicores, which is our focus. NoCMsg follows addresses scalability problems via message passing, not just for shared-memory multiprocessors as the Multikernel [8] but for multicores in our case. It takes ideas like NoC-level message passing from Factored Operating Systems [22] to another level in supporting low-level NoCMsg as a basis for scalable NoC communication without deadlocks.

Flow control elimination is utilized by iWarp[2], 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.

VIII. Conclusion

This work presents NoCMsg, a specialized MPI library designed to take advantage of network-on-chip architectures to improve scalability and performance, over a base MPI implementation up to 86% and, more significantly, shared memory abstractions such as OpenMP up to 93%. 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. 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. Overall, this study shows the potential for message passing for current experimental and forthcoming mainstream large-scale multicores. 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. In practice, hybrid OpenMP programs with 16 threads combined with message passing between OpenMP regions may be a viable solution and are subject to our ongoing work beyond the scope of this paper.

References

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

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

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

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

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

[6] Tilera user architecture reference. www.tilera.com.

[7] 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.

[8] 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.

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

[10] 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.

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

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.