Modeling Communication in Cache-Coherent SMP Systems - A Case-Study with Xeon Phi*

Sabela Ramos
Computer Architecture Group
University of A Coruña
Spain
sramos@udc.es

Torsten Hoefler
Scalable Parallel Computing Lab
ETH Zurich
Switzerland
htor@inf.ethz.ch

ABSTRACT

Most multi-core and some many-core processors implement cache coherency protocols that heavily complicate the design of optimal parallel algorithms. Communication is performed implicitly by cache line transfers between cores, complicating the understanding of performance properties. We developed an intuitive performance model for cache-coherent architectures and demonstrate its use with the currently most scalable cache-coherent many-core architecture, Intel Xeon Phi. Using our model, we develop several optimal and optimized algorithms for complex parallel data exchanges. All algorithms that were developed with the model beat the performance of the highly-tuned vendor-specific Intel OpenMP and MPI libraries by up to a factor of 4.3. The model can be simplified to satisfy the tradeoff between complexity of algorithm design and accuracy. We expect that our model can serve as a vehicle for advanced algorithm design.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Modeling techniques

Keywords

Cache coherency; Communication modeling; Shared memory systems; Intel Xeon Phi

1. MOTIVATION

The recent stop of frequency and Dennard scaling while Moore's law still holds, caused processor manufacturers to move into the direction of multi- and many-core architectures. Eight- or sixteen-core CPUs are standard in today's commodity machines, and the number of cores is growing, for example in graphics processing units (GPUs) which are exceedingly used for general purpose computations. However, programming GPUs requires a deviation from traditional latency-optimized programming to stream-optimized computing. Thus, it is considered viable to push the standard architectures, e.g., `x86` into the many-core era. Those architectures typically offer automated *cache coherency* to the user, a mechanism where changes in one cache are automatically transferred to other caches. Indeed, coherency protocols are usually the only means of communication between cores. Cache-coherency protocols are often implemented using fully connected crossbars and broadcast protocols for smaller numbers of cores. However, crossbar switches and broadcast protocols do not scale to larger numbers of cores and are commonly replaced by directory-based approaches.

Some ISA's, e.g. `x86`, do not offer explicit communication functions between cores, thus, communication must be implemented through loads and stores, essentially relying on the underlying cache-coherency protocol. The growing number of cores makes it increasingly important to understand the performance characteristics of such protocols. Analytic performance models, a formalization of such performance characteristics, can be used to design intelligent and scalable multi-core algorithms.

Current broadcast- and directory-based cache coherency protocols are implemented using a rather complex state machine where every cache line (a fixed unit of memory cells) can be in a different state in cache. In this work, we create a complete performance model for communication in cache-coherent systems by assigning a cost to each transition for a cache line in different caches. We then show how to reduce the complexity of the full model to enable its use for algorithm design. We demonstrate how to use this model mechanically for optimizing algorithms as well as the development of optimal algorithms. Our target architecture is the Intel's Xeon Phi accelerator, the currently most scalable cache-coherent single-chip architecture. Our developed algorithms are up to *4.3 times faster* than Intel's hand-tuned implementation in high-performance libraries and compilers (MPI and OpenMP).

In summary, the main contributions of this work are:

- We propose a novel state-based modeling approach for memory communication in cache-coherent systems.
- We demonstrate the applicability of the model to Intel Xeon Phi and show how it can be simplified for algorithm design.
- We demonstrate how our model can be used to design and optimize algorithms far beyond previous hand-optimized versions.

*This work was performed during a visit of S. Ramos at ETH, financed by HiPEAC

HPDC'13 June 17–21, 2013, New York, NY, USA.

Furthermore, based on those insights, we argue that the addition of explicit communication interfaces for on-chip communication could enhance the performance of parallel algorithms significantly.

2. A PERFORMANCE MODEL FOR COMMUNICATION IN CACHE-COHERENT SYSTEMS

In most multi-core systems, the only way to communicate data from one thread, T_0, to another thread, T_1, is to issue load and store instructions from and to main memory. For example, if T_0 wants to send a word (initially in a register) to T_1, it would store it to a specific memory location which is then read by T_1 (assuming appropriate synchronization).

Early multi-core systems, using the MESI protocol [13], would perform this communication through main memory, i.e., the line would be evicted from T_0's cache to main memory and loaded into T_1's cache from memory. However, the costs to communicate the line to memory and back are significantly more expensive than on-chip transfers. Thus, more recent cache-coherency protocols [10] such as MOESI, MESIF, and extended MESI [1, §2.1.3] protocols allow direct cache-to-cache transfers.

We discuss the MESI protocol as an example and note that it can be extended with additional states to model advanced protocols. Table 1 summarizes the MESI states and their semantics.

Table 1: MESI protocol states

		Who may own it		
		This core	Other cores	Is it Modified?
M	Modified	Yes	No	Yes
E	Exclusive	Yes	No	No
S	Shared	Yes	Yes	No[1]
I	Invalid	No	Yes	-

Since memory is generally arranged in blocks (or lines), we will argue in terms of cache lines in the following. First we assume that the communicated data fits in one line and then we will extend the model to cover multi-line transfers.

The cache coherency protocol state will determine the location of each cache line[2] and the operations needed in order to fetch it, thus the latency of reading or writing will depend on the state of the requested line.

When communicating a cache line, it is always transferred between the caches of two cores running threads T_0 and T_1, respectively. We assume the most general model where the line may be in any state in each cache. Let us assume that T_0 wants to communicate a line that is in state invalid (I) before the communication. Thread T_0 will write the send data and the line would transition from $I \rightarrow M$ in its local cache. The following read by T_1 would transfer the line to $T_1's$ cache and transition it in both caches into the shared state. We assume that each state update and line transfer has a specific cost associated with it.

A program $\mathcal{P}$ now defines a set of transitions for each used cache line in each cache of the parallel computation. The possible transitions for each cache line (as mandated by $\mathcal{P}$) can be modeled with a finite automaton with an initial state I (invalid).

We can now model the state changes and costs for all pairs of line states in T_0 and T_1. Figure 1 shows our initial cost model for the transfer of one cache line from T_0 to T_1. Let both threads be running in two different cores using an extended MESI protocol in which a modified line can be shared (other protocols would require slightly different but conceptually identical transition cost graphs). Each vertex represents the state of the line in each of the two cores, while edges represent the transitions. The possible states of a line in one core are M (modified), E (exclusive), S (shared) and I (invalid). Each vertex would be composed by the combination of two of these states to represent the cache line in each of the two cores. Vertices that are not shown represent invalid combinations of protocol states (e.g., the same line cannot be exclusive to two caches!).

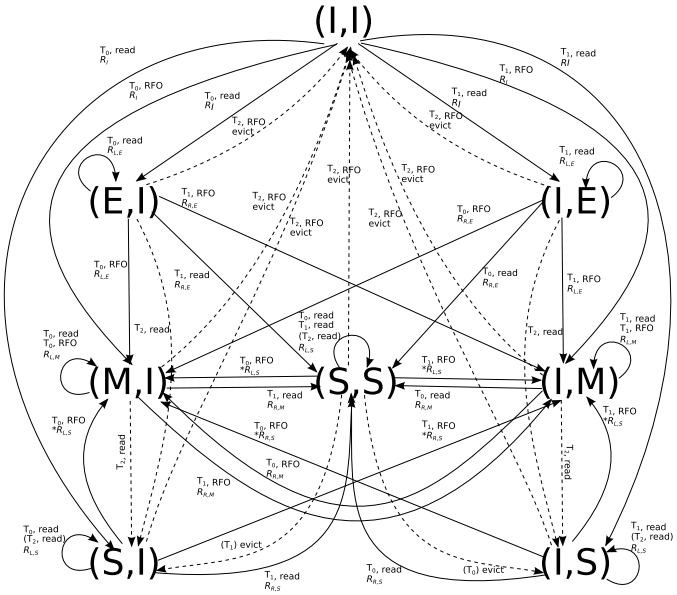


Figure 1: Transition diagram for the MESI protocol for communicating one cache line with two cores. The vertices form combinations of states (see Table 1), and the edges show actions that cause transitions and the associated costs.

Each transition is labeled with the required action (in the form $T_i, action$) and the cost associated. Dotted edges either represent external actions taken by a third thread (T_2) that runs in another core, or local capacity evictions. Thus, those edges do not have any associated cost. The actions can be *read*, *RFO* (request for ownership) and *evict*. When several actions appear at one edge, it indicates that any of them can be taken to transition to the target vertex.

The costs for all line transfers can be modeled in terms of line reads since a cache line is read into the local cache for both, load and store operations. Line writes only happen if lines are evicted to main memory and are thus outside of our cache-to-cache communication model. We denote the costs for reading lines as $R_{\mathcal{L},\mathcal{S}}$, where $\mathcal{L}$ is the location of the line (**L**ocal for the own cache of the thread performing the action, and **R**emote if the line is in other core cache), and $\mathcal{S}$ is the state of the line before the transition (M, E, S,

[1]This state can be extended to allow sharing of modified lines as we describe later.

[2]we use "cache line" and "cache line data" synonymously if the meaning can easily be inferred

or I). If $\mathcal{S} = I$, the location is not relevant since the line has to be fetched from main memory.

The symbol $*$ preceding some of the costs indicate that there can be an overhead as a consequence of the need to invalidate the line in other caches. This occurs, for example, with an RFO of a shared line: with a modified or exclusive line the target of invalidation is well-defined (the current owner), but with a shared line there can be multiple cores holding the line.

All costs ($R_{\mathcal{L},\mathcal{S}}$) in the model can be benchmarked with a methodology similar to the one proposed by Molka et al. [18]. Due to space limitations, we only discuss the most scalable and thus most interesting architecture, Intel Xeon Phi, in the following. The extended technical report version of this paper [22] contains model parameters for other architectures as well.

2.1 Intel Xeon Phi Architecture

The Intel Xeon Phi coprocessor is a many-core system based on the Intel MIC (Many Integrated Core) architecture. Its cores are arranged on a bidirectional ring bus that provides high scalability. Figure 2 represents the basic architecture of the Xeon Phi including the cores, the bus, the memory controller and the tag directories. The current commercial Xeon Phi (5110P) has 60 simplified Intel CPU cores running at 1056 MHz and supports 4 threads per core with hyperthreading (thus, 240 threads in the die). The cores have a vector unit with 64 byte registers featuring a new vector instruction set known as Intel Initial Many Core Instructions (IMCI). Each core has a 32 kb L1 data cache, 32 kb L1 instruction cache, and a private 512 kb L2 unified cache which is kept coherent by a distributed tag directory system (DTDs). There are 64 tag directories connected to the ring and the address-mapping to the tag directories is based on hash functions over the memory addresses, leading to an even distribution around the ring. The bidirectional ring to which cores and DTDs are connected has three independent rings in each direction [7]: the data block ring (64 bytes wide), the address ring (send/write commands and memory addresses) and the acknowledgment ring (flow control and coherence messages). The memory controllers, also connected to the ring, provide access to the GDDR5 memory (8 GB of global memory). The coprocessor runs a simplified Linux-based OS in one of the cores.

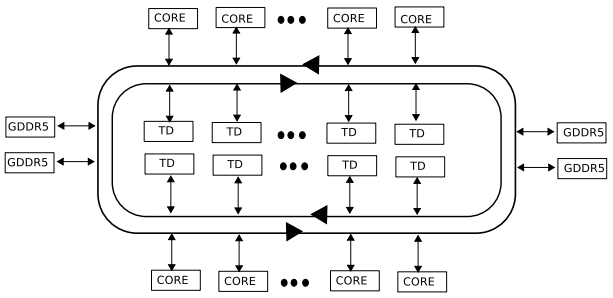


Figure 2: Architecture of the Intel Xeon Phi coprocessor

The main advantage of the Xeon Phi over other accelerators or coprocessors is that it provides the well-known `x86` ISA and memory model, hence the programming effort is just focused on how to better exploit performance, but it can be done with known techniques and languages such as OpenMP or MPI. Xeon Phi can be used as a mere coprocessor in which the host offloads code to be accelerated, or as an independent unit that runs a whole application or that communicates in a symmetric manner with the host [1, §6].

2.1.1 Directory-Based Cache Coherency

The cache coherency on Intel Xeon Phi chips is implemented using an extended MESI protocol [1, §2.1.3]. The main difference to standard MESI-based systems is that the shared (S) state was extended with a directory-based cache coherency protocol called GOLS (Globally Owned, Locally Shared) in order to avoid broadcast storms on the address buses. GOLS extends the shared state to allow modified as well as unmodified lines. Each cache needs to consult the GOLS protocol to determine if a line is in modified state or not.

The global coherency is maintained via *Distributed Tag Directories* (DTDs) that hold the GOLS coherency state of each line. Lines are assigned to each DTD using a hash function based on the address of the line. This results in an even load distribution (assuming an even distribution of memory addresses) but does not take advantage of locality in the network. This means that the DTD which is responsible for a line held by a specific core is not local to the core, in fact, on average, it will be at a distance of *15 cores* due to the ring topology. Table 3 describes the different states of the GOLS directory protocol.

Table 3: GOLS protocol states

		Number of owners	Is it Modified?
GOLS	Globally Owned Locally Shared	Several	Yes
GE /GM	Globally Exclusive /Modified	One	Yes or No (the core has M or E)
GS	Globally Shared	Several	No
GI	Globally Invalid	None	-

When a core encounters a cache miss, it requests the line from the according DTD that will answer depending on the GOLS state and will either request the memory or the core owning the line to answer with the line data. If a core owns the line, it will acknowledge the DTD and send the data to the requester cache, which will then acknowledge the DTD that it has received the line, for the DTD to update the line state. In addition, any eviction has to notify the DTD before evicting the line.

A direct consequence of having a distributed directory protocol based on line addresses is that there are high differences in access latencies that are not dependent on the distance among cores but on the DTD that is holding the line. Since we cannot control the address mapping onto DTDs, we will use randomized accesses and work with averages and standard deviations to avoid DTDs bias in the benchmarking results and, thus, in the modeling. In fact, we observed up to a 5x variation in latency when not using randomization.

The protocol states are rather similar to MESI, the only difference lies in the fact that some reads are more expensive due to interaction with the DTDs and the extended shared state. However, the extended MESI model developed in the previous section is sufficient to model all transitions.

	Same core		Adjacent cores		Middle distance		Largest distance	
	avg	stdev	avg	stdev	avg	stdev	avg	stdev
M	8.6	0.2	241.2	21.7	234.7	25.6	240.1	10.4
E	8.6	0.2	227.4	20.6	235.8	25.5	237.4	27.7
S	8.7	0.9	232.0	10.2	233.4	35.0	233.4	22.5
I	277.7	34.0	274.3	25.2	278.8	34.4	284.5	29.6

(a) Results of the BenchIT [18] latency benchmark in nanoseconds

Label	Cost
$R_{L,M}$	8.6
$R_{L,E}$	8.6
$R_{L,S}$	8.7
$R_{R,M}$	234.7
$R_{R,E}$	235.8
$R_{R,S}$	233.4
R_I	277.7

(b) Model parameters (nanoseconds)

Label	Cost
$R_{L,*} = R_L$	8.6
$R_{R,*} = R_R$	235.8
R_I	277.7

(c) Simplified parameters (nanoseconds)

Table 2: Parametrizing and simplifying the model: (a) shows latency results and standard deviations for different distances, (b) extracts the relevant model parameters (cf. Fig. 1), (c) shows the simplified model parameters (cf. Fig. 3).

2.2 Parametrizing the Model for Xeon Phi

We use the BenchIT [18] benchmark to determine the base parameters of our model. The benchmark measures the latency of T_0 reading random lines from a buffer owned by T_1 varying the state of the lines and the placement of T_1. We place T_1 in the same core (distance 0), in an adjacent core (distance 1), in a distant core (distance 15) and in a core located at the opposite side of the ring (distance 30). When T_1 is in the same core as T_0, the reads of M, E, and S lines are performed inside the local cache, but, when T_1 is located in another core, T_0 has to communicate with the DTD in charge of the cache line (the S state is achieved by sharing the line between T_1 and a third thread).

Our measurements in Table 2a show that communication with the DTD makes the distance between the two cores nearly irrelevant. In fact, the distance-invariant performance is a design goal for excellent application scalability on Xeon Phi. Our model parameters can be derived from those measurements (we use the "Middle distance") and are shown in Table 2b. If the line is in I state, that is, it has to be fetched from memory, it does not matter if T_1 is in the local core or in a remote one, and for Table 2b we chose the values for "Same core".

The full model in Table 2b indicates that some of the states in Figure 1 can be collapsed due to nearly identical transition costs. We can form three groups of operations, local reads, remote reads, and reads of invalid lines. Table 2c and Figure 3 summarize the simplified model for Intel Xeon Phi.

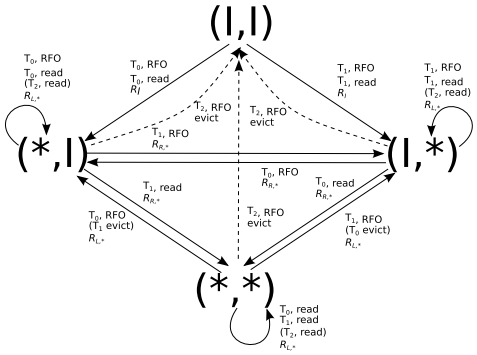


Figure 3: Graph of the simplified MESI transitions of a line within two cores

The simplifications we applied to make the model easier to use for algorithm design are system-specific. We tested the methodology on other systems, such as Sandy Bridge and Nehalem, and found that similar simplifications can be performed. The main difference is that, on those systems, the performance is generally depending on the distance between the caches. We found that this can be included with constant cost-offsets for each distance.

3. COMMUNICATION MODELS

In this section, we utilize the basic cache communication model to develop slightly more complex models for typical communications in parallel programming. The first and simplest model is for a single cache-line ping-pong benchmark where a send-buffer is communicated to a distinct receive buffer. The main difference is that this benchmark involves two memory locations and two lines, instead of one location as in the previous model. Later, we will extend the single-line ping-pong to multiple lines and then investigate the effect of contention while accessing single lines from multiple threads. This will result in a complete model for communications in cache-coherent systems that covers all scenarios that, for example, the LogP model family [9] expresses.

3.1 The Single-line Ping-Pong Model

The single-line ping-pong benchmark resembles the design of traditional ping-pong benchmarks: a send buffer is copied to a buffer at the receiver and, after reception, another buffer is copied back to the sender. This requires two sets of buffers on each process (thread) and a synchronization between sender and receiver. We are using a designated byte, which we call *canary value* in the receive buffer as a synchronization flag such that the receiver waits for the message by repeatedly reading this byte (polling) until the byte changes. Such "canary protocols" are used in practice for small-message synchronizations [12][3]. Figure 4 shows the benchmark schematically.

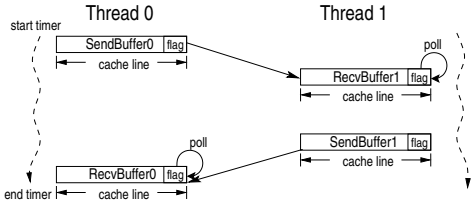


Figure 4: Ping-Pong test between two threads using four buffers.

The desired coherence state for each line is established before each ping-pong exchange. An S state indicates that

[3]We remark that such canary protocols are typically limited to a single aligned cache line due to relaxed memory consistency in modern multiprocessors.

the line is shared between the two threads performing the ping-pong. Considering the symmetry of the benchmark, send-buffers from both threads have the same cache coherence state ($\mathcal{S}_s$), and so do the recv-buffers ($\mathcal{S}_r$). The estimation of the latency of a ping-pong can be based upon the latencies of reading each line.

We assume that the protocol is sender-driven, i.e., the sender copies the data to the receiver. To perform the copy, the sender reads its send-buffer, which is in state $\mathcal{S}_s$ in the local cache, in time $R_{L,\mathcal{S}_s}$. This is followed by a read of the correspondent recv-buffer, which is in state $\mathcal{S}_r$ in the remote cache, in time $R_{R,\mathcal{S}_r}$. Then (ignoring the overhead that the write operation could provoke) the receiver reads its recv-buffer, that has been modified by the sender, and thus is in modified state in the sender's cache, in time $R_{R,M}$. This process is repeated with switched roles. Equation (1) shows $\mathcal{T}_1$, the single-line latency and the simplified model. The term O stands for an overhead that might be introduced by the coherency traffic due to having two active communicating threads (ideally, $O = 0$).

$$\mathcal{T}_1 = R_{L,\mathcal{S}_s} + R_{R,\mathcal{S}_r} + R_{R,M} + O = R_L + 2R_R + O \quad (1)$$

We measured 5000 independent iterations (using the high-precision `x86 RDTSC` counter) with pseudo-random addresses to avoid bias caused by the DTDs. The average and standard deviation form a Gaussian Distribution of the samples. We applied the t-test to each result to assess the statistical significance of our results and modeling. Where the result of the t-test was to reject the null hypothesis of equality of averages between both distributions, the overhead O was estimated as the difference of the distributions. Since the variances are unknown and not equal, we used Welch's t-test [25]. We found that we could reject the equality of averages with more than a 90% confidence in every scenario, confirming that there is an extra overhead imposed by the coherency traffic.

For the two threads T_0 and T_1 on different cores and $\mathcal{S}_s = \mathcal{S}_r = E$, we measured 497.1$\mu s$ (standard deviation $\sigma = 77.2\mu s$) while the simplified model predicts 479.1μs ($\sigma = 36.1\mu s$). For the states $\mathcal{S}_s = I$ and $\mathcal{S}_r = E$, we measured 842.8μs (standard deviation $\sigma = 102\mu s$) while the simplified model predicts 748.1μs ($\sigma = 49.6\mu s$)[4].

The state $\mathcal{S}_r = I$ cannot be measured with our method because it cannot be guaranteed that the receive buffer is in invalid state when the sender attempts to fetch it. This is due to the benchmark design: the receiver is polling the recv-buffer and it can fetch it from memory before the sender requests it.

The results show that the overhead $O \simeq 18$ for the in-cache configuration and $O \simeq 95$ if the send-buffer is in memory. However, the overhead is lower than the standard deviation of the measurements, thus, although our t-test shows that it is statistically significant, it could be within the noise of real measurements.

3.2 The Multi-line Ping-Pong Model

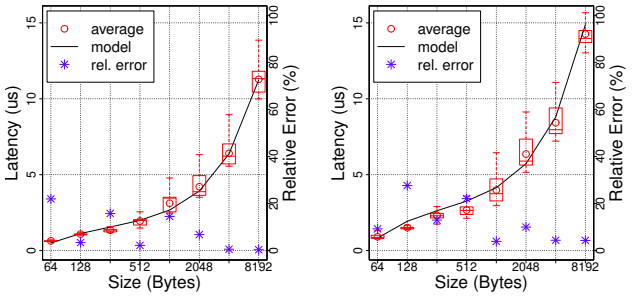
We now show how to model multi-line ping-pong transfers, i.e., buffers can contain more than one cache line and, assuming `x86` total store order [20], the receiver will only poll for the canary value on the last line of the recv-buffer while the sender copies the content of the send-buffer. The analysis of different cache states is limited to 8 kb buffers due to the use of four buffers per pair of threads and the L1 cache size (32 kb). From now on, and given that we want to analyze the effect of having threads in different cores, we will assume a one-to-one mapping of threads to cores.

Assuming pipelining, the sender fetches every line in $\frac{2N}{P} R_{\mathcal{L},\mathcal{S}}$ where N is the number of cache lines of each buffer and P is the number of outstanding memory requests per core. After the copy, the receiver reads the last line, that has been modified by the sender, in $R_{R,M}$.

However, this simple model misses several factors that affect performance as the eviction overhead, the hardware prefetcher, the signal buses or the DTD capabilities to serve the outstanding requests. To approach this overhead, we tested a multiplicative factor based on the results, but, although it was asymptotically accurate, the relative error reached the 40-50% for small messages (2-12 lines) when the send-buffer was in I, and around 30% when it was in E. To obtain a more accurate model, we use linear regression with a typical transfer function. Equation (2) shows the model function where o is the asymptotic fetch latency for each cache line (including hardware prefetch, etc.), and p, q model the startup overhead which consists of a fixed part q that is amortized partially by the number of fetched lines with the factor p.

$$\mathcal{T}_N = o \cdot N + q - \frac{p}{N} \quad (2)$$

If we apply this model to a single line broadcast, it will essentially lead to the one-line model discussed before in Section 3.1: $\mathcal{T}_1 = R_L + 2R_R + O = q + o - p$.



(a) Sender and receiver buffers in Exclusive state. The parameters of the model (in nanoseconds) are $76.0 \cdot N + 1521.0 - \frac{1096.0}{N}$

(b) Sender and receiver buffers in Invalid state. The parameters of the model (in nanoseconds) are $94.9 \cdot N + 2750.0 - \frac{2017.5}{N}$.

Figure 5: Latency and performance model for a multi-line ping-pong

The parametrization of the model has been performed with ping-pong tests using buffers from 64 bytes (one cache line) to 8 kb, varying the initial cache state of the buffers[5].

The results of our ping-pong measurements and the model fits are shown in Figure 5. The measurement for each size were repeated 5000 times and timed separately using `x86 RDTSC`. The left axis shows boxplots [17] of each value where the horizontal line is the median, the upper and lower parts of the box denote the first and third quartile and the whiskers show the minimum and maximum data values (out-

[4]An extensive analysis of every combination of states and locations of the threads has been carried out and are included in the extended version of this article [22].

[5]This test can use the $\mathcal{S}_r = I$ because the receiver is polling only the last line, and when the sender fetches the recv-buffer lines they are all invalidated except for the last one.

liers were removed). We use boxplots to visualize the statistical noise across measurements. The right axis and asterisks show the relative error of the model.

3.3 DTD Contention Model

On Xeon Phi, the DTDs may cause delays when they are contended [7]. Thus, we include an additional contention model to capture this effect. This may not be necessary in broadcast and snooping-based cache-coherency protocols.

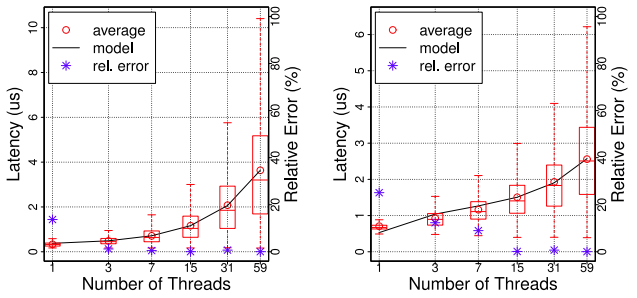
Contention is benchmarked using a global send-buffer owned by one thread that every other thread (receivers) copies into a private recv-buffer. When having only two threads, the performance is expected to be $R_{\mathcal{L}_s,\mathcal{S}_s} + R_{\mathcal{L}_r,\mathcal{S}_r}$. For the rest of the section, and given that the sender is an idle thread that only owns the global buffer, we will assume that the number of threads is the number of receivers.

The contention on MIC for cached lines can be estimated with a linear model $\mathcal{T}_C(n_{th}) = c \cdot n_{th} + b$, where n_{th} is the number of threads, and c represents the slope and the overhead imposed when adding a new thread. If $n_{th} = 1$, there is no contention and $\mathcal{T}_C(1) = R_L + R_R = c + b$ (the cost of copying a global send-line into a private recv-line). Equation (4) shows the DTD contention model when buffers are in E state in the owner's cache.

$$\mathcal{T}_C(n_{th}) = R_L + R_R + c \cdot (n_{th} - 1) = b + c \cdot n_{th} \qquad (3)$$

However, if the global line is in memory, the performance is limited by the access to memory and the model is similar to the one developed for the multi-line ping-pong in terms of the number of threads accessing the line instead of the message size.

$$\mathcal{T}_C(n_{th}) = c \cdot n_{th} + b - \frac{a}{n_{th}} \qquad (4)$$



(a) Sender and receiver buffers in Exclusive state. The parameters of the model (in nanoseconds) are $320.5 + 56.2 \cdot n_{th}$.

(b) Sender and receiver buffers in Invalid state. The parameters of the model (in nanoseconds) are $23.4 \cdot n_{th} + 1202.0 - \frac{695.8}{n_{th}}$.

Figure 6: Contention in the access to the same line

Figure 6 shows the results of the benchmark for different number of threads and state of the global and private buffers (E for both in Figure 6a and I in Figure 6b).

3.4 Ring Contention

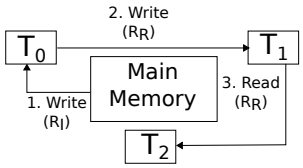
We have also analyzed how the number of running threads affects the performance of the cache line transfers. For this purpose, we have designed two ping-pong benchmarks. The first one arranges threads into groups of four where the communicating pairs are interleaved (e.g., if a group is formed by T_0, T_1, T_2 and T_3, and T_i is running in core i, the pairs are $T_0 - T_2$, and $T_1 - T_3$). The second benchmark forces pairs to communicate through the same part of the ring (e.g, with 6 threads, the pairs will be $T2 - T3$, $T1 - T4$, $T0 - T5$) assuming that communications will use the shortest path. Due to the ring structure of the Xeon Phi, from the 16th pair on, communications will go through the other half of the ring.

Regardless of the initial cache state of the buffers, both benchmarks showed that there is no congestion caused by having several pairs of threads communicating simultaneously if they are accessing different memory addresses. The differences that appeared were not related to the number of running threads and the most feasible reason is the assignment of the requested lines to DTDs.

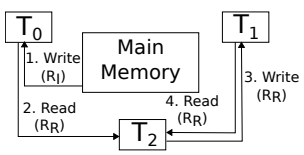
We have shown that the communication model can be parametrized accurately for simple communication tasks such as ping-pong. We now discuss how the model can be used to develop and parametrize suitable algorithms for communications on Xeon Phi.

4. DESIGNING COMMUNICATION ALGORITHMS IN THE MODEL

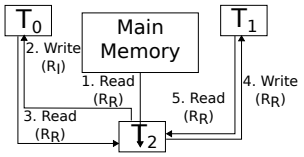
The communication algorithms tackled are different patterns of data exchange where interference among threads hugely increase variability. E.g., if two threads, T_0 and T_1 have to write to a line that T_2 is polling, waiting for them to write the value, the performance will depend on the order in which the three operations occur, as shown in Figure 7. If T_0 and T_1 write to the line, and then T_2 checks it (Fig. 7a), the cost is $R_I + 2R_R$. But, if T_2 checks it right after the write of T_0 (Fig. 7b), the line makes an extra travel to T_2 before going to T_1, and T_2 has to read it again to get the expected value ($R_I + 3R_R$). And there is still a worse scenario: when T_2 is the first that gets the line (Fig. 7c) and keeps polling, causing the line to travel from T_2's cache to each writer and the other way round, increasing the cost up to $R_I + 4R_R$.



(a) No interference



(b) One Interference



(c) Two Interference

Figure 7: Interference in the access to a cache line by two writers (T_0 and T_1) and a reader that is polling the line waiting for writes.

To capture all the variations, the algorithms are expressed as *Min-Max Models* including the best and the worst case.

In the design of algorithms, it is assumed that data buffers are initially in exclusive state in the owner's cache to simplify the discussion. Similar results were obtained with buffers in invalid state and the models can be adapted by applying the invalid-state equations where the exclusive-state models are

used. For the same reason, shared structures are assumed to be in memory (invalid) at the beginning of each algorithm.

4.1 Fast Message Broadcasting

The broadcast operation consists of sending one message from one thread, called *root*, to every other thread. We will talk in terms of trees since they are the more common communication patterns in broadcast algorithms.

In a shared memory scenario, a send-receive pair of operations can be performed in two different ways. In a *sender-driven* approach, the sender copies the data into the recv-buffer, similar to the pingpong benchmark from Section 3.1; the receiver may notify the sender with the canary protocol that the recv-buffer is ready. In a *receiver-driven* approach, the receiver would copy the message after the sender has notified that it is ready (notification forwards). In addition, the receiver has to acknowledge the reception of the message (notification backwards).

For the broadcast operation, where the sender communicates with several receivers, the receiver-driven approach allows simultaneous copies and thus leads to better load balancing for larger numbers of threads despite the additional acknowledgment.

4.1.1 Notification

The notification forwards and backwards uses shared structures in order for them to be accessible to every thread. There, the root can notify that the message is ready to be copied and the rest of threads can confirm that they have received the message, so that the root can free the shared structure. If the algorithm uses a tree, each parent has to communicate with its descendants and every descendant has to notify backwards to is parent, thus, several notification substructures will be needed.

Given that the parent has to provide, along with a notification flag, the data that is going to be copied or the address where it is stored, the notification forwards can be seen as a notification with payload where data and flag can be fetched in a single line. Hence, if data is small enough to fit in the same line, the descendants will poll the notification line and they will copy the data directly from there. If the space in the notification line is not enough, the parent will set the flag and an address (zero-copy protocol) from which descendants will copy the data.

The backwards notification from the descendants to the parent uses cache lines that are independent from the notification forwards structures to avoid interference in the copy of the data. We analyze two variants of this notification: The first one with one cache line in which every thread adds a value after finishing. The parent reads this value and checks if the operation is done. This requires every child thread, to write to the same line, and, since only one thread can write a line at a time, these writes are going to be serialized. In the second variant, each thread avoids serialization by writing its own notification line, but the parent has to read them all to check if the operation is done. We will focus on the use of one line because both the model and the empirical results confirmed that it provides better performance.

The model for the notification backwards assumes that each thread writes an immediate value and thus there is no cost associated with reading an additional cache line.

Since all threads, but the root, write to the same line, every thread (but the root) has to read and modify the notification line ($R_I + (n_{th} - 2)R_R$). Then, in the best case, the root only reads the line at the end (R_R).

$$\mathcal{T}_{nb,min}(n_{th}) = R_I + (n_{th} - 1) \cdot R_R \tag{5}$$

In the worst case, the root will check the notification line after each time a thread wrote to it. This scenario is modeled by Equation (6).

$$\mathcal{T}_{nb,max}(n_{th}) = R_I + 2(n_{th} - 1) \cdot R_R \tag{6}$$

4.1.2 Small Broadcast

Once the notification has been analyzed, the design of the broadcast algorithm focuses on the stages needed to copy the data from the root into every other descendant. Karp et al. developed an optimal algorithm [14] in the LogP model, but this is very different from our state-based min-max model with separate notification. However, we can use a similar technique to design our optimal tree taking into account that all the descendants of a given node can get the data at the same time.

First of all, we will describe the structure of a generic tree assuming that each level i can use a different number of descendants (k_i) and that the height of the tree is d. In this structure, the number of threads in each level (n_i) of the tree is given by equation 7.

$$n_0 = 1, n_i \leq \prod_{j=1}^{i} k_j \tag{7}$$

Hence, the total number of threads can be expressed as:

$$n_{th} \leq 1 + \sum_{i=1}^{d} \prod_{j=1}^{i} k_j \tag{8}$$

All of the k_i descendants of one thread from level i are accessing the same line, thus, by increasing the number of descendants, we also increase contention. Hence, every one of these descendants is able to get the data in $\mathcal{T}_C(k_i)$. It is also worth mentioning that different threads accessing different data should not cause any congestion, thus, it is possible to apply the contention model to each group of descendants ignoring other groups of threads. Thus, the latency of copying a message throughout this tree is:

$$\begin{aligned} \mathcal{T}_{tree} &= \sum_{i=1}^{d} \mathcal{T}_C(k_i) = \sum_{i=1}^{d} (c \cdot k_i + b) \\ &= \sum_{i=1}^{d} (R_R + R_L + c \cdot (k_i - 1)) \end{aligned} \tag{9}$$

The optimized tree has to find a tradeoff between the number of threads that get the value at the same time, thus causing congestion ($c \cdot k_i$), and the number of levels of the tree. It is expected that the values of k_i decrease while descending throughout the tree since the lower the value of k_i, the lower the latency of acquiring one message.

Figure 8 presents an example of a 10-threads broadcast tree using (arbitrarily chosen) d=2, $k_1 = 3$, $k_2 = 2$. The backwards arrows indicate from which node the receivers copy the message. Threads from level 1 get the message in $t_1 = c \cdot k_1 + b = 3c + b$ and, then, leaf nodes will copy it in $c \cdot k_2 + b = 2c + b$, thus, the total time will be $t_2 = c \cdot (k_1 + k_2) + 2b = 5c + 2b$.

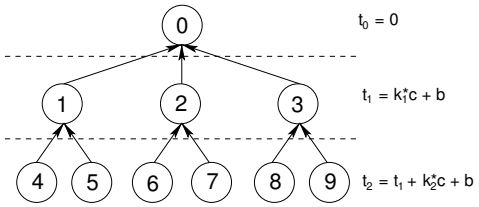


Figure 8: Tree for an 10-threads broadcast assuming d = 2, $k_1 = 3$, $k_2 = 2$.

Once the tree structure is defined, we have to take into account the notification. The total time for notification backwards is the time spent from the moment in which the last descendant receive the message until the root is aware that every thread has it. The correspondent equations from Section 4.1.1 must be applied to each level of the tree, providing the cost of notification backwards in the critical path.

Regarding notification forwards (i.e., the parent notifying to its descendants that the buffer is ready to be copied), first, there is a global flag where the root sets the shared structure as occupied by the current operation (R_I). Each descendant has to check the flag and copy the data (that are on the same line), which can be estimated by the contention model (Equation (9)), and then, each parent has to read its own structure (R_I), copy the data into this structure (R_L) and set it as ready (R_L). In the worst case, the descendants can read the flag before it is set and interfere while the parent is copying the data and setting the flag. Moreover, when interference involves several threads, they will cause contention. Although the first reading affected by contention is an invalid line, the contention model for a cached line is used for simplicity purposes. Equation (10) show the best (min) and worst (max) case model for this notification forwards.

$$\begin{aligned} \mathcal{T}_{fw,min} &= R_I + \sum_{i=1}^{d} (R_I + 2R_L) = (d+1)R_I + 2dR_L \\ \mathcal{T}_{fw,max} &= R_I + \sum_{i=1}^{d} 2\left(R_R + \sum_{i=1}^{d} (c \cdot k_i + b)\right) \end{aligned} \tag{10}$$

The optimal tree to perform a one-item broadcast is thus the solution to the minimization problem expressed in equation (11), combining notifications and reception of data.

$$\begin{aligned} \mathcal{T}_{sbcast} = \min_{d,k_i} \left(\mathcal{T}_{fw} + \sum_{i=1}^{d} (c \cdot k_i + b) + \sum_{i=1}^{d} \mathcal{T}_{nb}(k_i + 1) \right) \\ N \leq 1 + \sum_{i=1}^{d} \prod_{j=1}^{i} k_j, \quad \forall i < j, k_i \leq k_j \end{aligned} \tag{11}$$

This equation can be solved with numerical methods to obtain d and all k_i for the optimal broadcast tree.

4.1.3 Large Broadcast

When each thread has to copy N lines using the tree developed for the small broadcast and assuming that the N lines are sent in N_{pack} packets of size N_{cl}, the leafs will not start copying until the first package has arrived.

In order to avoid having idle threads in the first stages, and given that it is possible to divide the N-line message in $N_{pack} = n_{th} - 1$ slices, we can construct an algorithm in stages in which every thread, but the root, starts copying one different slice of the message. Having every line of the message in the root's cache could cause some contention (the root has to communicate with the DTDs to change the state of each line from E or M to S) and for the next stages, each thread copy one slice of the message from a different thread, having only one thread copying from the same location at the same time. The performance model of this pipelined algorithm is stated in Equation (12).

$$\begin{aligned} \mathcal{T}_{pipedbcast} &= \mathcal{T}_{init} + \mathcal{T}_{1^{st}} + \mathcal{T}_{rest} + \mathcal{T}_{fin} \\ \mathcal{T}_{init,min} &= R_I + 2R_L \\ \mathcal{T}_{1^{st},min} &= 2R_L + \mathcal{T}_C(n_{th} - 1) + \mathcal{T}_{N_{cl}} \\ \mathcal{T}_{rest,min} &= (n_{th} - 2)(R_R + R_L + \mathcal{T}_{N_{cl}}) \\ \mathcal{T}_{fin,min} &= 2R_R + R_L \end{aligned} \tag{12}$$

This algorithm will use the same notification structure than the small broadcast, but since only one thread accesses this information in each stage, it is possible to have the flag, address and notification in the same line, allowing the receiver to fetch it only once during the stage. Moreover, the owner of the line only checks the notification at the end of the whole algorithm, minimizing interference. The model has been divided in four parts: (1) initialization ($\mathcal{T}_{init}$), in which every thread checks its notification line and sets the local buffer address. (2) First stage ($\mathcal{T}_{1^{st}}$): the root sets its flag to ready (R_L), the rest of threads check it ($\mathcal{T}_C(n_{th} - 1)$ is an upper bound to the real value because the contention model implies the copy of a line and in this scenario threads only read the value) and copy of the first slice of the message ($\mathcal{T}_{N_{cl}}$ using the multi-line model) and sets its own flag to ready (R_L). (3) Rest of stages ($\mathcal{T}_{rest}$), where the rest of packets ($n_{th} - 2$) are copied, including the check for readiness (R_R) and the notification to the owner (R_L). And (4) Finalization ($\mathcal{T}_{fin}$), each thread checks for completion (R_R) and sets the own structure as free (R_L). The extra R_R represents the notification to the root. To avoid interference and serialization in this notification, each thread will notify the first copy in a different stage.

Having only one thread accessing one location at every stage minimizes interference, however, there are still some points in which it can appear. In $\mathcal{T}_{1^{st}}$, as happened in the notification forwards from Equation (10), the polling threads can interfere with the root. Moreover, in $\mathcal{T}_{rest}$, any thread (e.g., T_2) can finish its stage earlier tan others and try to read a flag before it is set, e.g., by T_1. When setting it, T_1 forces the line to be evicted from T_2's cache, that will have to fetch it again later. And finally, it is possible to assume that the last thread writes the notification after the first check for completion, adding some extra costs.

$$\begin{aligned} \mathcal{T}_{init,max} &= R_I + 2(R_R + \mathcal{T}_C(n_{th} - 1)) + R_R \\ \mathcal{T}_{1^{st},max} &= 2R_R + \mathcal{T}_C(n_{th} - 1) + \mathcal{T}_{N_{cl}} \\ \mathcal{T}_{rest,max} &= (n_{th} - 2)(2R_R + \mathcal{T}_{N_{cl}}) \\ \mathcal{T}_{fin,max} &= 2R_R + R_L \end{aligned} \tag{13}$$

Although this algorithm minimizes contention and interference, it can also preclude the benefits of prefetching (Equation (2)) that is only exploited for each packet.

Thus, we analyze a second algorithm, a flat tree, that optimizes for prefetching. In a flat tree, all receivers access the whole message after the root notified them. This algorithm ends when the receivers acknowledge the root that

they have copied the message. Since the number of threads colliding is large, the notification system uses two lines, in the same way as the small broadcast. The analysis to be done here is how contention affects the performance of requesting multiple contiguous lines, thus, we have to combine the contention and the multi-line models. For this purpose, we use the slope factor of the multi-line ping-pong model (o) as the time that it takes for one thread to get the message. This operation will be affected by the congestion caused by the rest of threads but the root ($n_{th}-2$). As intercept or constant factor, we arbitrarily chose the b from the contention model (assuming that the buffers are in exclusive state). In this scenario, the Flat Tree algorithm represents a good tradeoff between the benefits of prefetching and the drawbacks of contention. The rest of the model is equivalent to one stage of the small broadcast tree. Equation (14) reflects the best and worst models for this algorithm.

$$\begin{aligned} \mathcal{T}_{ftbcast} &= \mathcal{T}_{notif} + \mathcal{T}_{copy} \\ \mathcal{T}_{copy} &= b + c \cdot (n_{th}-2) + o \cdot N \\ \mathcal{T}_{notif,min} &= R_I + 3R_L + R_R + T_C(n_{th}-1) \\ &\quad + (n_{th}-1)R_R \\ \mathcal{T}_{notif,max} &= R_I + R_L + 3R_R + 2T_C(n_{th}-1) \\ &\quad + 2(n_{th}-1)R_R \end{aligned} \tag{14}$$

We expect the second algorithm to perform better for large message sizes.

4.2 Barrier Synchronization

A barrier synchronization involves every thread acknowledging that every other thread has reached the synchronization point. We have modeled it as a dissemination barrier since it has been proven to be the best algorithm for single-port LogP systems, but optimizing the parameters within our min-max models. The dissemination algorithm uses $r = log_m(n_{th})$ rounds in which thread T sends a notification to thread $(T + i(m+1)^r) \mod n_{th}, 0 < i \leq r$ and waits for the notifications from $(T - i(m+1))^r) \mod n_{th}, 0 < i \leq r$. In our shared memory scenario, assuming that every thread owns a notification line, each "send" operation consists of setting a flag and waiting until the receivers acknowledge that they have read this flag; and, "receive" is to notify to the senders the read of the corresponding flags.

In the best case, the owner was the last reader of its line (to check its value in the previous round), having it in cache when setting it to ready (R_L), and, the cost of checking it after every receiver has finished is R_R. Moreover, it has to read m threads' flags and, assuming no interference and that flags are already set, the thread will read and write to them just fetching each line once. Although every thread has to read m lines, they are not contiguous and exposed to be prefetched, thus we will not apply the multi-line model. The contention model does not apply either because, although m threads are accessing each line, they are performing writes that have to be serialized. The total cost is shown in Equation (15). The m value must be chosen to minimize this cost.

$$\begin{aligned} \mathcal{T}_{barr,min} &= r(R_L + R_R \cdot m + R_R) \\ r &= \lceil log_m(n_{th}) \rceil \end{aligned} \tag{15}$$

However, in every round, the own line can be in other core's cache, e.g. if other thread is already checking the flag,(R_R) and the notification value can be checked once and every time that it is modified by a notifier thread ($(2m+1)R_R$). Finally, if the first read of other thread's flags results in failure (the flag has not been set yet), at least another read of the line has to be performed. Taking into account that other $m-1$ threads can get the line and modify it in between, this interference could result in $(3m) \cdot R_R + m(m-1)R_R$. Since it is unlikely to happen, the model includes only one interference per line m.

$$\begin{aligned} \mathcal{T}_{barr,max} &= r(R_R + 4m \cdot R_R + (2m+1)R_R) \\ r &= \lceil log_m(n_{th}) \rceil \end{aligned} \tag{16}$$

The best m can again be found using numerical methods.

4.3 Small Reduction

A reduction is the application of an operation to data collected from all threads. In this section we will analyze the implementation of the reduction of one item.

The root is receiving from multiple threads, thus, the operation is very different from broadcast. A first approach could be having all those threads writing to a common location. Then, each thread will have to (1) check a flag to see if the buffer is ready (R_R), (2) read the buffer (R_L), (3) apply the reduction operation to the buffer using its private data (R_L), (4) write the result to the data buffer and (5) notify that it has finished (R_R). If several threads are accessing the same buffer, steps 2 to 5 have to be performed in an atomic manner, thus, serializing. To avoid serialization, the root has several buffers in which each descendant writes its data. Then the root reads them all and performs the operation.

This scheme can be structured in a tree similar to the broadcast one. In this tree, each thread from level i has k_i buffers where its k_i children copy their own data. Then, the parent performs the operation with the data from these buffers. In each stage of the tree, the parent has to set a flag (R_I) that their children (k_i) read (causing some contention) before writing to the corresponding buffer ($R_R + R_L$) and notifying that the data is ready (R_R). Once the parent gets the acknowledgment (R_R), it performs the operation (which is modeled using the multi-line model). The tree minimizing Equation 17 forms our solution.

$$\begin{aligned} \mathcal{T}_{red,min} &= \sum_{i=1}^{d} [R_I + \mathcal{T}_C(k_i) + (1+k_i)R_R + R_L + \mathcal{T}_{k_i}] \\ &\quad + R_R \end{aligned} \tag{17}$$

The interference in the notification forwards (some threads read the parent's flag before it is set) and in the notification backwards (the parent checks the notification before it is complete) are reflected in the worst case in Equation (18).

$$\begin{aligned} \mathcal{T}_{red,max} &= \sum_{i=1}^{d} [R_I + 2\mathcal{T}_C(k_i) + 2(1+k_i)R_R + R_L + \mathcal{T}_{k_i}] \\ &\quad + R_R \end{aligned} \tag{18}$$

Here again, we compute the optimal d and k_is using numeric techniques.

5. EVALUATION

The evaluation of the designed algorithms has been performed on an Intel Xeon Phi 5110P with 60 cores at 1052 MHz, the host machine is a Intel Xeon E5-2670 Sandy Bridge with 8 cores at 2.60 Ghz. The Intel MIC software stack is the MPSS Gold update 2.1.4346-16, with the Intel Composer XE 2013.0.079, the Intel Compiler v.13.0 and Intel MPI v.4.1.0.024. The benchmarks used are the EPCC OpenMP Benchmarks 3.0 and the Intel MPI Benchmarks (IMB) 3.2.

The benchmarks used to measure the performance of our algorithms were developed to ensure a given cache state in each of the 1000 iterations. Before each iteration, threads are synchronized with a custom `RDTSC`-based synchronization and the data lines are placed in the desired cache state.

To guarantee that all threads start at the same time we used the feature that the `RDTSC` is consistent among cores [1, §2.1.7] in a normal power state of the Xeon Phi. Thus, we generate time intervals for threads to achieve before starting, then assuring that they enter each operation at the same time. A second synchronization before the collective operation is performed. The time is measured for every operation call and the whole distribution of times is used for statistical analysis of the obtained results. The goal is to check whether the model predictions are accurate enough and compare the results with existing solutions.

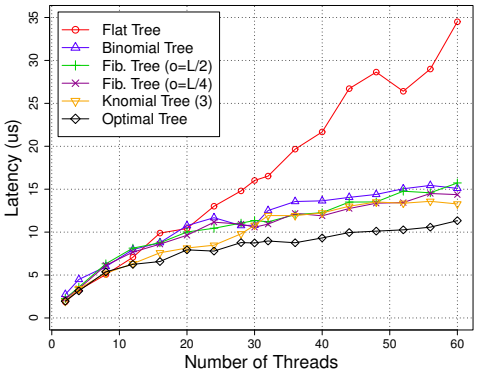


Figure 9: Small Broadcast performance comparing our optimal algorithm with widely used broadcast trees.

Before testing, all the parametrizable algorithms (small broadcast, reduction and synchronization) have been optimized to obtain the best parameters by minimizing the best case models from Section 4. The optimization for the worst case was also taken into account with similar results, hence, only the parameters obtained for the best case model are shown in the graphs. As an example, when having 30 threads, the parameters obtained for a small broadcast were $d = 2$, $k_1 = 5$, $k_2 = 5$; for reduction $d = 3$, $k_1 = 3$, $k_2 = 3$, $k_3 = 2$; and for barrier $m = 6$ ($r = 2$). For 60 threads, the parameters were $d = 3$, $k_1 = 4$, $k_2 = 4$ and $k_3 = 3$ for small broadcast and reduction, and $m = 4$ ($r = 3$) for barrier. Parameters differ for each number of threads and that is the reason of some variations in the models as seen around 28 processes in Figure 12.

All benchmarks launch one thread per core and, when using 60 threads, the variability increases because it is not possible to avoid the core that runs the OS.

Figure 9 shows a comparison between different algorithms for small broadcast: flat tree, binomial tree, k-nomial tree (k=3), Fibonacci tree and our optimal tree, all of them with buffers in E state. For Fibonacci trees [14], given that they are designed for LogP and that in this system it is not exactly applicable, we have chosen $o = L/2$ and $o = L/4$ to construct the tree. As expected, the optimal algorithm developed using the model obtains the lowest latency even though, some of the other algorithms, e.g., Fibonacci Trees, are optimal in other models.

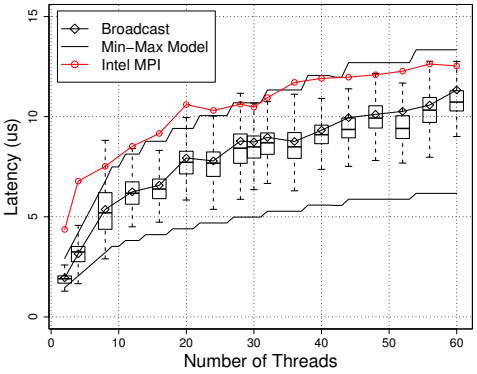


Figure 10: Small Broadcast performance compared to the model and the Intel MPI implementation.

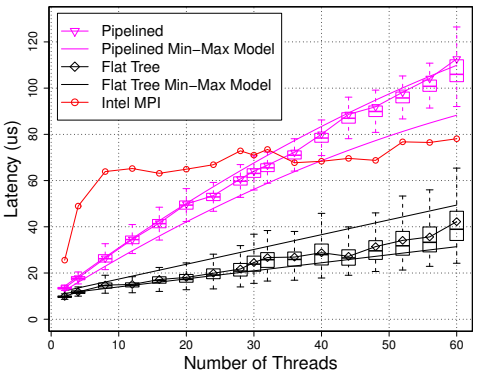


Figure 11: Large Broadcast (8 kb) algorithms compared to the model and the Intel MPI implementation.

Figures 10 to 13 represent the performance obtained with the algorithms modeled in Section 4. Results are presented with the corresponding boxplots and the min-max model. The large broadcast uses 8 kb messages because it is the higher buffer size that was modeled for the multi-line ping-pong, and the operation used in the reduction is a summation. As it can be seen, the min-max model is able to capture the inherent variability of the use of threads and allowed us to obtain the best parameters for the small broadcast, the small reduce and the barrier.

To compare the results with current shared memory communication solutions, the graphs also include the latency obtained with MPI and OpenMP (when applicable). It is worth mentioning that the benchmarks used for OpenMP and MPI measure the average result without synchronizing threads before each iteration and without forcing any cache state, avoiding the eviction of the shared data and taking advantage of temporal locality across iterations. Our benchmark forces the data to be in exclusive state in the buffer owner's cache, thus invalidating it in any other cache.

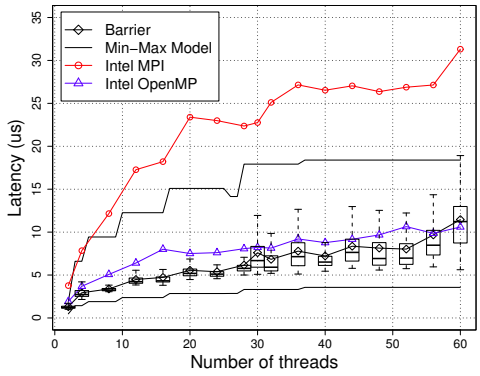


Figure 12: Barrier Synchronization results compared to the model and the Intel OpenMP and MPI implementations.

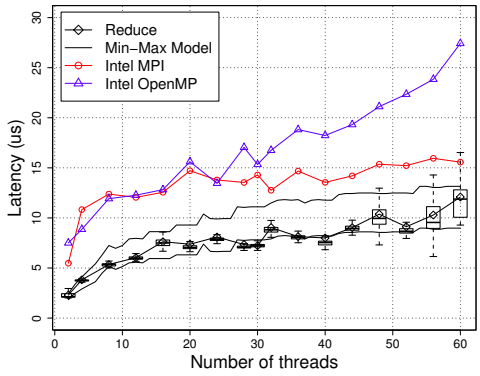


Figure 13: Small Reduction performance compared to the model and the Intel OpenMP and MPI implementations

However, even in that case, our algorithms outperform MPI and OpenMP except for two scenarios. In Figure 12, with 60 threads, the OpenMP barrier obtained a latency that is lower than our algorithm, however, it seems that it is highly optimized for a large number of threads while ours is optimized separately for each number of threads. Moreover, they take advantage of the non-cache-invalidation policy between iterations used in the benchmarks. In the results of the large broadcast (Fig. 11), the "pipelined" algorithm is outperformed by MPI when the number of threads is larger than 32, although it does not happen if the flat tree algorithm is used. As mentioned in Section 4.1.3, the flat tree obtains a good tradeoff between contention and prefetching while the pipelined algorithm is not able to take advantage of prefetching.

6. RELATED WORK

The optimization of parallel computation is based on the study of the architectural features that can influence performance. However, algorithm design requires models that simplify and abstract complex systems, e.g., LogP [9], LogGP [3], LoGPC [19], PlogP [16] or Hockney [11] model the communications in distributed memory systems. On the other hand, models like PRAM [15], that assume that processors can access global memory without cost, study the logical structure of parallel computation removing communication from the analysis. Another approach followed in [6] and [5] is the inclusion of memory concerns in the model to measure the effects that memory buffer copies have in point-to-point communications. Our work is focused on shared memory architectures were point-to-point communications are translated directly to memory transfers, thus, buffer transformations would be treated as common memory operations.

These models have been successfully used to design optimal algorithms for communication operations. In [14], Karp et al. show that Fibonacci trees are optimal for small broadcasts. The authors of [23] use a simple linear communication model to develop bandwidth-optimal broadcast and reduction algorithms.

With the increase in the number of cores per processor, the modeling of shared memory communications is also crucial to develop efficient algorithms to transfer information through shared memory among the cores of the system. Petrovic et al. [21] discuss communications in the precursor of the Intel Xeon Phi, the Intel SCC. However, this system did not provide cache coherency, which simplifies the interactions among threads greatly.

The cache coherency protocols have been also widely studied, specially in terms of internal memory hierarchy models analyzing the effects of evictions and memory locality. Agarwal et al. [2] presents a comprehensive model for associative caches and other works like [24] or [4] study the behavior of the memory hierarchy on multi-core systems, but focus on the behavior of caches and the optimization of parallel codes avoiding cache misses, and does not discuss the effects of communications among cores.

Early experiences on the Intel Xeon Phi coprocessor [8] showed that this architecture provides scalable performance, which combined with the possibility of obtaining highly parallel applications with standard programming paradigms, makes it really interesting to explore the communications among cores in a shared memory environment.

7. DISCUSSION AND CONCLUSIONS

We found that, especially for small data, the notification system and interference caused by threads in the polling stages, can impact performance more than the actual data transfer. In order to model these effects, we had to resort to min-max models that complicate the algorithm development considerably. Nevertheless, our model allows algorithm designers to abstract away from the architecture and the detailed cache coherency protocols and design algorithms on purely analytic ground. We showed that our models can be combined into a powerful framework for tuning and developing parallel algorithms.

In general, we found that optimizing for cache-coherency protocols is harder than optimizing for systems that offer direct remote memory access. The developed models and techniques are more complex than, for example algorithms in the LogP model. Based on results gathered in [21], we would assume that direct remote cache access (DRCA) would lead to parallel systems with higher performance and better predictability and transparency. Thus, we conjecture that DRCA would greatly simplify the design of parallel algorithms.

However, if such architectures are not an option, our models describe a viable method for designing parallel algorithms on cache-coherent architectures. Indeed, our simplified model can be used rather mechanically to optimize and parametrize well-known algorithms. In addition, we showed how to develop new and optimal algorithms requir-

ing slightly more effort. While all our models do not provide precise predictions rather than a range of possible performance, we demonstrated how they can be used to guide algorithm design and development.

The algorithms we developed with the help of our analytical models show performance improvements over Intel's hand-tuned MPI and OpenMP libraries in nearly all configurations with a maximum improvement of 4.3 times. Our method can also be used for other architectures and algorithms.

Acknowledgments

We thank the Swiss National Supercomputing Center (CSCS), especially Hussein Harake, Thomas Schoenemeyer, and Thomas Schulthess, for providing access to and support with Xeon Phi hardware. S. Ramos thanks the HiPEAC network, the University of A Coruña, the Ministry of Science and Innovation of Spain [Project TIN2010-16735], and the Xunta de Galicia CN2012/211, partially supported by FEDER funds for financial support.

8. REFERENCES

[1] Intel® Xeon Phi™ Coprocessor: Software Developers Guide. `https://www-ssl.intel.com/content/www/us/en/processors/xeon/xeon-phi-coprocessor-system-software-developers-guide.html`, 2012.

[2] A. Agarwal, J. Hennessy, and M. Horowitz. An Analytical Cache Model. *ACM Trans. Comput. Syst.*, 7(2):184–215, 1989.

[3] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating Long Messages into the LogP Model - One Step Closer towards a Realistic Model for Parallel Computation. In *Proc. 7th Annual ACM Symp. on Parallel Alg. and Arch. (SPAA'95)*, pages 95–105, S. Barbara, CA, USA, 1995.

[4] D. Andrade, B. B. Fraguela, and R. Doallo. Accurate Prediction of the Behavior of Multithreaded Applications in Shared Caches. *Parallel Computing*, 39(1):36 – 57, 2013.

[5] K. W. Cameron, R. Ge, and X. H. Sun. lognP and log3P: Accurate Analytical Models of Point-to-Point Communication in Distributed Systems. *IEEE Trans. Computers*, 53(3):314–327, 2007.

[6] K. W. Cameron and X. H. Sun. Quantifying Locality Effect in Data Access Delay: Memory logP. In *Proc. 17th IEEE Intl. Parallel & Distrib. Processing Symp. (IPDPS'03)*, page (8 pages), Nice, France, 2003.

[7] G. Chrysos. Intel® Xeon Phi™ Coprocessor (Codename Knights Corner). Keynote talk at the 24th Hot Chips: A Symp. on High Perf. Chips, 2012.

[8] T. Cramer, D. Schmidl, M. Klemm, and D. an Mey. OpenMP Programming on Intel Xeon Phi Coprocessors: An Early Performance Comparison. In *Proc. Many-core Applications Research Community (MARC) Symp. at RWTH Aachen University*, pages 38–44, 2012.

[9] D. Culler et al. LogP: towards a Realistic Model of Parallel Computation. *SIGPLAN Not.*, 28(7):1–12, 1993.

[10] D. Hackenberg, D. Molka, and W. E. Nagel. Comparing Cache Architectures and Coherency Protocols on x86-64 Multicore SMP Systems. In *Proc. 42nd Annual IEEE/ACM Intl. Symp. on Microarchitecture (MICRO'42)*, pages 413–422, New York, NY, USA, 2009.

[11] R. W. Hockney. The Communication Challenge for MPP: Intel Paragon and Meiko CS-2. *Parallel Computing*, 20(3):389 – 398, 1994.

[12] T. Hoefler and T. Schneider. Optimization Principles for Collective Neighborhood Communications. In *Proc. 25th ACM/IEEE Intl. Supercomputing Conf. for High Performance Computing, Networking, Storage and Analysis (SC'12)*, Salt Lake City, UT, USA, 2012.

[13] L. Ivanov and R. Nunna. Modeling and Verification of Cache Coherence Protocols. In *Proc. 2001 IEEE Intl. Symp. on Circuits and Systems (ISCAS'01)*, pages 129–132, 2001.

[14] R. M. Karp et al. Optimal Broadcast and Summation in the LogP Model. In *Proc. 5th Annual ACM Symp. on Parallel Alg. and Arch. (SPAA'93)*, pages 142–153, Velen, Germany, 1993.

[15] R. M. Karp and V. Ramachandran. A Survey of Parallel Algorithms for Shared-Memory Machines. Technical report, Berkeley, CA, USA, 1988.

[16] T. Kielmann, H. E. Bal, and K. Verstoep. Fast Measurement of LogP Parameters for Message Passing Platforms. In *Proc. 15th IPDPS 2000 Workshops on Parallel & Distrib. Processing*, pages 1176–1183, 2000.

[17] R. McGill, J. W. Tukey, and W. A. Larsen. Variations of Box Plots. *The American Statistician*, 32(1):12–16, 1978.

[18] D. Molka, D. Hackenberg, R. Schoene, and M. S. Mueller. Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System. In *Proc. 18th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'09)*, pages 261–270, Raleigh, NC, USA, 2009.

[19] C. A. Moritz and M. I. Frank. LoGPC: Modeling Network Contention in Message-Passing Programs. *IEEE Trans. on Parallel and Distrib. Systems*, 12(4):404–415, 2001.

[20] S. Owens, S. Sarkar, and P. Sewell. A Better x86 Memory Model: x86-TSO. In *Proc. 22nd Intl. Conf. on Theorem Proving in Higher Order Logics (TPHOLs'09)*, pages 391–407, Munich,Germany, 2009.

[21] D. Petrović, O. Shahmirzadi, T. Ropars, and A. Schiper. High-performance RMA-based Broadcast on the Intel SCC. In *Proc. 24th ACM Symp. on Parallelism in Alg. and Arch. (SPAA'12)*, pages 121–130, Pittsburgh, PA, USA, 2012.

[22] S. Ramos and T. Hoefler. Modeling Communications in Cache Coherent Systems . Technical report, University of A Coruña, ETH Zurich, 2013.

[23] P. Sanders, J. Speck, and J. L. Träff. Two-Tree Algorithms for Full Bandwidth Broadcast, Reduction and Scan. *Parallel Comput.*, 35(12):581–594, 2009.

[24] L. G. Valiant. A Bridging Model for Multi-core Computing. *Journal of Computer and System Sciences*, 77(1):154 – 166, 2011.

[25] B. L. Welch. The Generalization of 'Student's' Problem when Several Different Population Variances are Involved. *Biometrika*, (1-2):28–35, 1947.