Cache Line Aware Algorithm Design for Cache-Coherent Architectures

Sabela Ramos, and Torsten Hoefler, *Member IEEE*

Abstract—The increase in the number of cores per processor and the complexity of memory hierarchies make cache coherence key for programmability of current shared memory systems. However, ignoring its detailed architectural characteristics can harm performance significantly. In order to assist performance-centric programming, we propose a methodology to allow semi-automatic performance tuning with the systematic translation from an algorithm to an analytic performance model for cache line transfers. For this, we design a simple interface for cache line aware optimization, a translation methodology, and a full performance model that exposes the block-based design of caches to middleware designers. We investigate two different architectures to show the applicability of our techniques and methods: the many-core accelerator Intel Xeon Phi and a multi-core processor with a NUMA configuration (Intel Sandy Bridge). We use mathematical optimization techniques to tune synchronization algorithms to the microarchitectures, identifying three techniques to design and optimize data transfers in our model: single-use, single-step broadcast, and private cache lines.

Index Terms—Cache coherence, shared memory, communication algorithms, performance modeling, Xeon Phi, Sandy Bridge.

1 Motivation

Coherent shared memory simplifies the initial design of parallel programs in current multi- and many-core architectures, but the complexity of the coherence protocols often leads to poorly-scalable solutions. Even tuned variants can usually be improved significantly. This is mainly because the complex interactions are hidden from programmers, who need to design highly-scalable parallel algorithms to utilize the exponentially growing number of cores. In order to enable the consideration of cache coherence hardware during algorithm-design, we propose a Cache Line aware (CLa) design methodology. In CLa, middleware programmers assume minimal structure, the existence of cache lines, while designing and analyzing algorithms and implementations. We provide a simple interface that enables reasoning about algorithm structure and that eases the translation to performance models. *While it may seem complex to expose the existence of (typically hidden) cache lines, we propose to virtualize their allocation in the interface and only expose the minimal assumption of the fixed-size design.*

In this work, we focus on two basic types of directives needed to design parallel shared memory algorithms: thread synchronization and data transport. For *synchronization*, we identify four modes depending on the number of threads involved. Each mode has non-trivial performance trade-offs for different implementations. For example, many threads writing to a single cache line may lead to high coherence traffic, while a thread that reads multiple lines written by others may observe high local polling overheads. We use a parametrized analytical performance model of the cache coherence protocol to determine the best synchronization mechanism. We also model *data movement* using cache line transfers, potentially involving multiple cache lines.

To guide performance-centric development in CLa, we identify three basic principles: (1) *Single-use synchronization lines*: Many synchronization patterns benefit from utilizing each cache line only once when synchronizing different groups of threads. These groups can refer to different sockets or different stages of an algorithm. This technique reduces the variability caused by having multiple threads reading and writing the same line. (2) *Single-step broadcast*: Most cache coherence protocols provide fast mechanisms to push a cache line to a large set of cores. (3) *Line privatization*: Some algorithms benefit from assigning a private cache line to each core to perform data movement.

In summary, the specific contributions of our work are:

1) We propose Cache Line aware (CLa) optimization, a method for performance-centric programming of cache coherent systems. We show how CLa can be used to optimize shared-memory data movement and synchronization algorithms in tandem.
2) We identify three basic principles: single-use synchronization, single-step broadcast, and line privatization to design high-performance algorithms.
3) We show how to systematically model the performance of algorithms analytically and find close-to-optimal design trade-offs using established mathematical optimization tools.
4) We design a methodology to translate shared memory communication algorithms directly into an analytical performance model.
5) We conduct a practical study with a 5110P Intel Xeon Phi and a dual-socket Intel Xeon E5-2660 architecture yielding speedups between 1.3x and 44.6x over optimized libraries.

The rest of the paper is organized as follows: We describe the cache coherence model and CLa methodology in Sec-

• *Scalable Parallel Computing Lab, Computer Science Department, ETH Zürich, Switzerland. E-mail: sramos@udc.es*
• *S. Ramos was with the Computer Architecture Group, University of A Coruña, Spain when developing part of this work.*

tion 2, and we exemplify its application to the development of shared memory algorithms in Sections 3, 4, and 5. We summarize the related work in Section 6, and present our conclusions in Section 7.

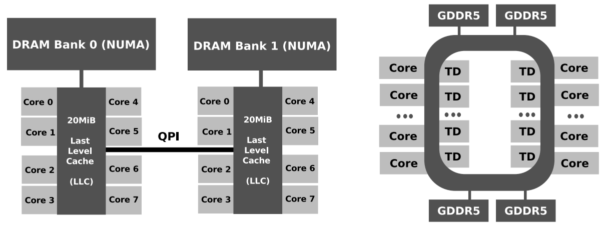
2 CLa Performance Model

In order to analyze algorithms in terms of (cache) line transfers, we propose a basic performance model based on a set of building blocks. We identify two main primitives which we parametrize through benchmarking considering thread location and coherence state: single-line and multi-line transfers. Moreover, the interaction between threads may introduce additional overheads. Some interactions, such as contention (several threads accessing the same cache lines) and congestion (several threads accessing different lines) can be quantified. Other interactions depend on the real-time order in which operations are performed and are not predictable (see Section 2.3). Nondeterministic interaction prevents us from obtaining a precise performance prediction and forces us to work with lower and upper time bounds. Nevertheless, our model is accurate enough to perform algorithm design and even performance prediction.

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.

2.1 Hardware Description

Our conclusions and methods are not limited to a specific architecture and we now briefly describe two different systems on which we exemplify our techniques: a Sandy Bridge-based ccNUMA system and the many-core accelerator Xeon Phi KNC.



(a) Dual-socket Sandy Bridge Xeon E5-2660 multi-core processor (b) Intel Xeon Phi KNC coprocessor

Fig. 1. Sandy Bridge and Xeon Phi KNC architectures

2.1.1 Sandy Bridge

Our first testbed is a dual-socket eight-core processor Intel Sandy Bridge [1] Xeon E5-2660, at 2.20GHz (see Figure 1a) with Hyper Threading activated and Quick Path Interconnect (QPI, 8 GT/s). Each socket has three levels of cache. L1 (32 KB data and 32 KB instructions) and L2 (256 KB unified) caches are private to each core. An L3 cache (or LLC) of 20 MB is shared by all cores within a socket and divided in eight slices. The internal components of the chip, including the LLC slices, are connected via a ring bus and all cores can use any LLC's cache slices, thus having access to data stored in all of them.

The chip uses the MESIF cache coherence protocol [2], based on the classic MESI (Modified-Exclusive-Shared-Invalid) protocol [3]. It adds a *Forward* state to optimize the management of shared states. When a line is shared among several cores, one of them (the *forwarder*) has it in *F* state, and it is in charge of answering when another core requests this line, preventing multiple answers. The last core requesting this line becomes the new forwarder, and the previous forwarder becomes a sharer (*S*).

Although it globally behaves like a snooping protocol, cache coherence within each socket is kept by the LLC, an inclusive cache that holds a set of bits indicating the state of the line and which cores have a copy. Among sockets, the Quick Path Interconnect (QPI) is used to implement the cache coherence protocol. It is in this scenario when the *F* state is useful to avoid multiple answers from different sockets to a snooping request.

2.1.2 Xeon Phi (KNC)

The Intel Xeon Phi coprocessor is a many-core system based on the Intel MIC (Many Integrated Core) architecture (Figure 1b). Its cores are arranged on a bidirectional ring bus that provides high scalability. We use a KNC-5110P Xeon Phi with 60 simplified Intel CPU cores running at 1056 MHz. It supports 4 threads per core with multithreading (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) with up to 64 tag directories connected to the ring. 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.

The cache coherence on Intel Xeon Phi chips is implemented using an extended MESI protocol [4, §2.1.3] that allows to share modified as well as unmodified lines using a directory-based cache coherence protocol called GOLS (Globally Owned, Locally Shared). The global coherence is maintained via *Distributed Tag Directories* (DTDs) that hold the GOLS coherence state of each line. Lines are assigned to each DTD using a hash function based on the address of the line [5]. 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 often not local to the core, in fact, on average, it will be at a distance of *15 cores* due to the ring topology. A direct consequence 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 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.

2.2 Model Parameters

The necessary building blocks to construct and parametrize our performance model are obtained by benchmarking cache line transfers and thread interactions. There are differences in the building blocks for the different architectures. Nevertheless, the methodology is applicable to characterize these blocks for a large number of cache coherent systems.

2.2.1 *Single-line Transfers*

The basic block in our model is the transfer of a cache line between two cores. Line transfers are caused by two operations: *read*, and *RFO* (Read For Ownership). Both involve fetching lines, but the latter indicates the intention to write. We estimate the cost of both as a *read* (R), although there could be some differences, e.g., an *RFO* of a shared line means that all copies must be invalidated. But we first analyze transfers between two threads, where this difference is not significant. We use $R_{i,j}$ to represent the cost of reading a line from location i with cache state j. A location can be L (local), R (remote, core in the same socket), and Q (remote, core accessed through QPI). The cache state is any MESIF state and $*$ indicates any location or state.

We implemented a simple ping-pong data exchange to analyze the impact of thread location and coherence state. In this operation, there are only transfers of one line involved and we can model the RTT[1] using $R_{i,j}$ parameters, as shown in Equation (1). Thread t_0 copies its send buffer (initially, the data is either local, $R_{L,*}$, or invalid, $R_{*,I}$) into a receive buffer owned by t_1 ($R_{R,*}$ or $R_{Q,*}$ depending on t_1's location), while t_1 is polling the last byte to check if the transfer has finished [6] (charging $R_{R,M}$ or $R_{Q,M}$). Then, both threads switch roles and t_1 copies while t_0 polls. Both transactions use different send and receive buffers.

$$\frac{RTT}{2} = \begin{Bmatrix} R_{L,*} \\ R_{*,I} \end{Bmatrix} + \begin{cases} R_{R,*} + R_{R,M} & \text{if } t_1 \text{ is in the same socket} \\ R_{Q,*} + R_{Q,M} & \text{if } t_1 \text{ is in another socket} \end{cases} \quad (1)$$

On Sandy Bridge [7] we observe significant differences when varying the location of threads and lines, but only minor variations for different cached states. On Xeon Phi [8], on average, communication with the DTD makes the distance between the two cores nearly irrelevant. Hence, we cluster the costs for line transfers in the five classes shown in Table 1. We parametrize the cost of a line transfer for each class with the memory benchmark from BenchIT [2]. It estimates the cost of transferring one line between two threads depending on its coherence state and thread location.

TABLE 1
One Line Transfers

	State	Location	Cost (ns)	
			Sandy Bridge	Xeon Phi
R_L	Cached	Same core	2.3	8.6
R_R	Cached	Other core, same socket	35.0	235.8
R_Q	Cached	Other core, other socket	94.0	-
R_I	Invalid	Same NUMA region	70.0	277.7
R_{QI}	Invalid	Other NUMA region	107.0	-

Sandy Bridge supports loading half lines [9, §2.2.5.1] which is cheaper than always loading full lines. However, other architectures always transfer entire lines, thus, we will work with full lines for generality and clarity.

2.2.2 *Contention and Congestion*

We evaluate contention (threads accessing the same CL) and congestion (threads accessing different lines) with two benchmarks in which threads read an external send buffer and copy it into a local receive buffer. The measurements for each size were repeated 5,000 times and timed separately using `x86 RDTSC`.

1. Round Trip Time

Intra-socket results on Sandy Bridge show very low contention. When there are readers in both sockets, the requested line is transferred once through QPI and the LLCs are in charge of distributing it to all cores. Figure 2a shows results for congestion on Sandy Bridge when copying cache lines from one socket to another. We did not observe significant congestion for intra-socket transfers (less than 10% difference) and thus omit the results from the figure for clarity. The x-axis represents the number of threads copying from one socket to another. The figure also shows the model described in the following section, where we further analyze QPI congestion together with multi-line cache transfers.

On Xeon Phi, the DTDs may cause delays when they are contended [5]. However, there is no congestion when several pairs of threads communicate simultaneously if they access different memory addresses. The observed differences were not related to the number of running threads and the most feasible reason is the assignment of the requested lines to DTDs. The contention on Xeon Phi for cached lines can be estimated with the linear model from Equation (2),where n_{th} is the number of threads, and c represents the slope and the overhead imposed when adding a new thread.

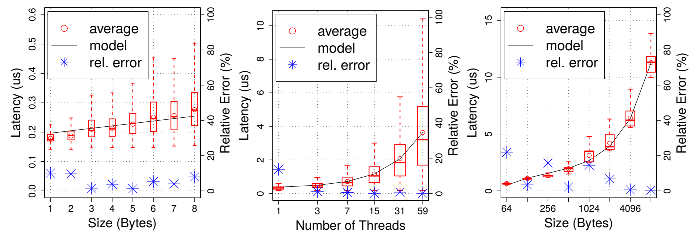
$$\mathcal{T}_C(n_{th}) = R_L + R_R + c \cdot (n_{th} - 1) = b + c \cdot n_{th} \quad (2)$$

However, if the global line is in memory, the performance is limited by the access to memory, as shown in Equation (3). The parameters of the model are in Table 2. Figure 2b shows the results of the benchmark for varying numbers of threads and buffers in E state.

$$\mathcal{T}_C(n_{th}) = c \cdot n_{th} + b - \frac{a}{n_{th}} \quad (3)$$

TABLE 2
Parametrization of the contention model for Xeon Phi KNC.

sendbuf	recvbuf	**b** $[ns]$	**c** $[\frac{ns}{thread}]$	**a** $[ns \cdot thread]$
E	**E**	320.5	56.2	0
E	**I**	604.4	57.6	0
I	**E**	863.6	23.9	667.4
I	**I**	1202.0	23.4	695.8



(a) QPI congestion on Sandy Bridge when copying one line. (b) Contention on Xeon Phi KNC when copying one line. (c) Multi-line pingpong on Xeon Phi KNC.

Fig. 2. Latency and performance models for contention, congestion, and multi-line transfers. Sender and receiver buffers are in Exclusive state.

2.2.3 *Multi-line Transfers*

We evaluate multi-line transfers with two benchmark strategies: ping-pong and one-directional transfers (similar to those used for contention and congestion).

2.2.3.1 Sandy Bridge: On Sandy Bridge, ping-pong times exhibit significant variability when using invalid lines. This variability stems from the use of DRAM and different NUMA regions and we developed approximate models to mitigate it. These models are not aimed to provide an exact prediction, rather, they allow us to simplify algorithm optimization and comparison. Without loss of generality, we work with cached multi-line transfers from now on. For cached lines, we empirically parametrize the model in Equation (4). N is the number of lines and n is the number of simultaneously accessing threads. The architecture parameters are summarized in Table 3: o is the latency increase per line, and the term cnN corresponds to congestion in the QPI link, hence, it is zero in intra-socket scenarios.

$$T_m(p,N) = q + oN + cnN \tag{4}$$

TABLE 3
Parameters for multi-line transfer of cached lines on Sandy Bridge.

	q $[ns]$	**o** $[\frac{ns}{line}]$	**c** $[\frac{ns}{line \cdot thread}]$
Intra socket	63.4	11.1	0
Inter socket	180.65	7.5	3.0

2.2.3.2 Xeon Phi: The model for Xeon Phi in Equation (5) is slightly different given the architectural differences between both systems: o is the asymptotic fetch latency for each cache line (including hardware prefetch, etc.), and p, q model the startup overhead using a fixed part q that is amortized partially by the number of fetched lines with the factor p. The parametrization is shown in Table 4.

$$\mathcal{T}_N = q + o \cdot N - \frac{p}{N} \tag{5}$$

TABLE 4
Parameters for multi-line transfers on the Xeon Phi KNC.

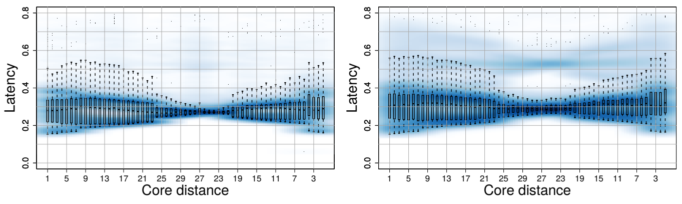
sendbuf	recvbuf	**q** $[ns]$	**o** $[\frac{ns}{line}]$	**p** $[ns \cdot line]$
E	E	1521.0	76.0	1096.0
E	I	1778.4	73.2	1276.9
I	E	2698.5	94.4	1868.5
I	I	2750.0	94.9	2017.5

The results of our measurements and the model fits are shown in Figure 2c. 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). In our previous work [8], we analyze how contention affects multi-line transfers. Our experience is that the o parameter from the multi-line model can be combined with the c, b, and a terms from the contention model.

2.2.4 *DTD impact on Xeon Phi*

Some authors [10], [11] demonstrate significant variations in cache line transfer latencies related to the use of arbitrary DTDs. In order to measure this effect, we use a benchmark with two running threads: a writer that owns a send buffer and a receiver with a recv buffer. Each buffer consists of a different cache line per iteration, forced to the desired cache state. Running threads are synchronized and we measure the cost of writing the send buffer into the recv buffer (performed by the writer thread). We explore different configurations, but the only variable that seems to have significant impact in the result is the randomization of the address line, as shown in Figure 3.

The y-axis represents the latency and the x-axis is the distance between cores (note that the ring is bidirectional, i.e., distance increases and then decreases). The writer always runs in core 1. The blue area is the frequency of the results (darker areas represent more frequent latencies). The main observation is that the average does not vary with core distance. However, the closer the cores, the higher the standard deviation. The cause is that, for closer cores, the DTD might be very close to both, but also very distant. When the cores are in opposite sides of the ring, any DTD is going to be either in the middle or close to one of them. Moreover, the plot is not symmetric, suggesting that some messages were not getting through the shortest path. Finally, for randomized accesses, the frequency plot has a light duplicate at a higher latency. A possible explanation is the existence of collisions and the need to recover data from memory. Since the average is the same in all cases, we will use the values provided by the model.



(a) Consecutive addresses. (b) Randomized addresses

Fig. 3. DTD effect using consecutive and randomized addresses.

2.3 Invalidation and Cache Line Stealing

The described building blocks can be used to model algorithms in terms of line transfers but we need to consider two additional sources of overhead due to interactions between threads or cores: The first one is on *RFO* of a shared line which involves invalidation at its n owners. On Xeon Phi, the cost is the same since it is arbitrated by the DTD. However, on Sandy Bridge, it costs nR_R instead of R_R. The second interaction, *cache line stealing*, is inherent to concurrency and it appears in multiple situations:

n-writers: n threads write the same line, one thread (t_0) polls until it gets the desired value (e.g., all writes have been performed). In the best case t_0 fetches the line *after* all writes have been performed. The cost is $R_I + (n-1)R_R + R_R$ assuming that the line is originally not cached. However, t_0 may read the line first, so the first writer invalidates t_0's copy. In fact, t_0 may *steal* the line between every two writes, increasing the cost dramatically to $R_I + n(2R_R)$.

n-readers: n threads poll a line and t_0 writes a notification value. The best scenario is when t_0 writes first and the n readers get the line at the same time (there is no contention). If we again assume that the line is not cached initially, the operation costs $R_I + R_R$ or $R_I + T_C(n)$. However, readers can *steal* the line before t_0. Writing the line then is an *RFO* of a shared line, requiring the invalidation in all n cores. To estimate this worst case within a socket on Sandy Bridge, we use $R_I + nR_R$. On Xeon Phi we have to consider that several readers stealing the same line may cause contention.

Both scenarios get more complex with more than one reader or writer, respectively. To capture all these variations, we use *min-max* models [8] that provide performance bounds by estimating the best and worst stealing scenario. The parametrized building blocks of our performance model, together with the analysis of thread interaction, enables to derive the minimum (T_{min}) and maximum (T_{max}) bounds. For a given algorithm, we construct an optimization problem to minimize T_{min} because T_{max} is usually too pessimistic, although it enables the analysis of the impact of thread interactions such as stealing.

In order to expose cache coherence interactions and apply our performance model, we propose a simple methodology that starts by expressing algorithms in a cache line centric manner using primitives that represent cache line transfers. Then, we use graphs to translate the algorithm into performance bounds, analyze thread interactions and construct the optimization problem.

2.4 A Candidate CLa Interface

We propose a simple set of cache line transfer primitives that can be implemented in various ways in most languages. While we do not prescribe a certain interface, we define an illustrative C API without loss of generality. In the remainder of this paper, we assume that the used language provides the facilities to allocate fixed-size blocks of aligned memory. This view does not only avoid false sharing [12] but it also allows us to tune the algorithm to the microarchitecture of cache-coherent systems.

We implement these primitives with direct load/store ISA instructions. When they are used for synchronization, we implement them with atomic instructions for writing, but not for reading and polling, because atomics often force the eviction of lines from other caches. The cost of each operation can be expressed in terms of location and state of the given lines. We define the following operations:

1) `cl_copy(cl_t* src,cl_t* dest,int N)` copies N lines from `src` to `dest`. It involves a load and a store.
2) `cl_wait(cl_t* line,clv_t val,op_t comp=eq)` polls until `comp(*(line),val)` is `true`. We omit the parameter `comp` if it has the default value `eq` (equal).
3) `cl_write(cl_t* line,clv_t value)` copies `value` into `line`.
4) `cl_add(cl_t* line,clv_t value)` adds `value` to `line`. If the `value` is read from a line, it is equivalent to a load plus a store.

Once we have the CLa pseudo-code, we construct a graph in which nodes are the CLa operations performed by each thread, linked by four types of edges[2]:

E1 The sequence of operations performed by one thread, represented by dotted directed edges.
E2 Logical dependencies between threads (i.e., reading or polling a line that has been written by others), represented by directed edges.
E3 Sequential restriction between threads that operate on the same data sequentially (the order is not defined), while another thread is waiting for the result of these operations. It is represented by non-directed edges.
E4 Line stealing caused in non-related operations (e.g., one thread polls the same line in two different stages of an algorithm), represented by dotted edges. It is only relevant for T_{max} and not considered for T_{min}.

2. We show examples of CLa graphs with different edges in Sections 3-5

Next, we assign costs to the nodes with these rules:

1) Flags are initially in memory, i.e., the first access to a flag costs R_I.
2) Access to data already accessed by the same thread (no incoming edges from other threads) costs R_L.
3) The access to the same line by the same thread in consecutive operations is counted once. E.g., a thread that adds values to the same line consecutively.
4) If the operation has an incoming edge from another thread, the cost is R_R or R_Q depending on the location of threads (same or different socket).
5) On Sandy Bridge, read operations with incoming edges from the same node can execute simultaneously without contention. E.g., threads copying a line that has been written by another thread. On Xeon Phi, we need to apply the contention model.

In order to derive the T_{min} (cost of the critical path), we define a path as a sequence of nodes linked by E1, E2, and E3 edges, starting in a node with no incoming E1 and E2 edges, and finishing in a node with no outgoing edges. Special consideration has to be given to E3 edges. They link all sequential writes that have outgoing E2s towards the same `wait`. For example, multiple threads incrementing one flag that other thread is polling. When searching for the critical path, we analyze reorderings of these writes, ensuring that the path visits each one before going towards the `wait`. In the example, all increments are performed in any order before the polling finishes. When some E3s represent inter-socket communications, we select the reordering with less QPI transfers.

To identify QPI congestion, we need to identify transfers performed simultaneously. Thus, we look for directed arrows between sockets that have: (1) different start and end points (accesses to different addresses performed by different threads), and (2) previous paths of similar cost (the transfers are performed at the same time).

Finally, T_{max} is calculated by analyzing line stealing. The main cause of line stealing is the `wait` operation, that may steal lines from every incoming edge from other threads. We can refine T_{max} by considering which operations might not overlap and not cause line stealing.

This set of rules is enough to derive graphs and performance models for the thread interaction algorithms explained in this paper (besides, we have successfully applied our methodology to several lock algorithms). It is easily extensible to cover other interactions.

We assume that the algorithm uses a given thread topology or communication structure. In some scenarios, like data movement primitives, we propose algorithms that receive a tree structure as parameter. In those cases, we construct the graph (and the model) for multiple communication structures to obtain the best one.

2.5 Benchmarking Methodology

The following sections present how we used the CLa methodology to derive and optimize communication algorithms. We benchmark all algorithms against OpenMP as well as ***MPI implementations that use shared memory directly*** to ensure a fair comparison. When available, we include a comparison with the HMPI NUMA-aware collectives [13]. We use a thread-based implementation to avoid inter-process communications and compare algorithms directly. We analyze performance in terms of speedup $S = T_{ref}/T$, where T is the latency of our algorithm and T_{ref} the latency of the library that we are comparing with. We use `RDTSC` intervals [14] to make the threads start iterations at the same time and we force synchronization data-structures and user-data in the desired cache states. Without loss of generality, we assume that user-data is cached in M state at the beginning of our algorithms. For synchronization structures, we assume that they are evicted (I state) during each computation phase. To support the NUMA memory, we use Linux' first-touch policy and assume that data is located in the NUMA region of the first reading thread (we charge R_I in the model). We ensure that threads are statically assigned to cores during the program execution. In all benchmarks, we measure the finishing times of all threads but only report the latency of the slowest thread per iteration (modeling bulk synchronous computations).

The systems used are a dual-socket eight-core processor Intel Sandy Bridge Xeon E5-2660, described in Section 2.1, with an Intel Xeon Phi 5110P with 60 cores at 1052 MHz. The OS is CentOS 6.4, the Intel MIC software stack is the MPSS 3.4.1. Compilers are Intel icc/ifort v.13.1.1 (for our algorithms, Intel MPI and Intel OpenMP) and GNU gcc/gfortran v.4.4.7 (for Open MPI and OpenMP), and MPI libraries Intel MPI v.4.1.4 and Open MPI 1.7.2. On Sandy Bridge, we use packing to schedule threads in the experiments: up to eight threads in one socket and the rest of them in the second one. This simplifies the comparison of intra- and inter-socket performance. In the result graphs we use a shaded area for the min-max model, and the results of our algorithms are shown with boxplots to represent the statistical variation of the measurements.

3 Synchronization Primitives

We use our methodology to model and design synchronization mechanisms. Depending on the number of threads involved, we identify four communication modes : one-to-one, many-to one, one-to-many, and many-to-many.

The simplest scenario is one-to-one: one thread, t_0, writes a line that the other thread, t_1 reads. It can be treated as a particular case of one-to-many or many-to-one synchronizations, thus, we will not analyze it separately.

3.1 Many-to-one

In many-to-one synchronizations, n threads notify one thread t_0. We analyze two basic implementations: (1) all n threads write the *same line* (Figure 4a) resembling the n-writers case with stealing (cf. Section 2.3), and (2) all n threads write *different lines* (one per notifier) where writes happen at the same time but t_0 needs to check n flags. The best case without stealing is when all lines are set before t_0 checks them. The worst stealing case depends on how t_0 checks lines, for which we differentiate two scenarios with the same cost: In the first one, t_0 iterates continuously over all lines until all are set. In the second scenario, t_0 checks each line only once, waiting until the current line is set before checking the next one (shown in Figure 4b).

Algorithm 1: Many-to-one

```
if $me==t_0$ then
[S1]    cl_wait(&flag,n);
else
[S2]    cl_add(&flag,1);
end
```

(a) Many-to-one (same line)

Algorithm 2: Many-to-one

```
if $me==t_0$ then
    for $i$=0...n, $i \neq t_0$ do
[S1]        cl_wait(flag[i],1);
    end
else
[S2]    cl_write(flag[me],1);
end
```

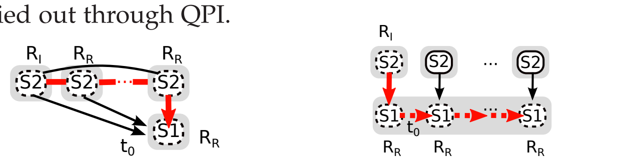
(b) Many-to-one (different lines)

Fig. 4. Pseudo-code for many-to-one synchronization patterns.

We use the CLa graphs from Figure 5 to construct the performance models. Each operation is a white node, labeled with the line number from the pseudo-code (Figure 4). Operations performed by the same thread are grouped in light-grey rectangles. The graphs have three types of edges:

- (E1) The dotted arrows indicate that t_0 reads n flags sequentially (cf. Figure 5b).
- (E2) The `wait` operations have a logical dependence from at least one `write` (or `add`) operation performed by other thread(s).
- (E3) In Figure 5a, n threads write (`cl_add` in the pseudo-code) the same line sequentially (in any order).

In order to derive the T_{min}, we assign costs to the nodes considering that fetching a flag costs R_I and the following uses of this flag in other threads cost R_R (R_Q if threads are in different sockets). Next, we search the critical path (thicker red edges and nodes with dotted lines) taking into account the peculiarities of E3 edges. E.g., in Figure 5a, we start in any of the S2 nodes, but we have to visit once all other S2 nodes before reaching the S1 (`wait`). Figure 5 shows the cost of the nodes in the critical path of an intra-socket (or Xeon Phi) scenario. For T_{max}, we include the cost of line stealing caused by `wait` operations. Table 5 presents the derived cost models for both architectures. For inter-socket synchronizations, we consider which transfers are carried out through QPI.



(a) Many-to-one (same line) (b) Many-to-one (different lines)

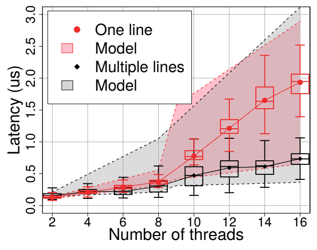
Fig. 5. CLa graphs for many-to-one patterns.

Figure 6a shows the results on Sandy Bridge. When using multiple lines, the real cost is lower than the prediction from Table 5 because threads arrive to the synchronization at the same time and t_0 is able to read the flags as a series of consecutive lines using prefetching. Hence, we used Equation (4), $T_m(1,n)$, to represent how t_0 reads lines. When there are processes in both sockets, we use the parameter for inter-socket transfers using two different o's: one to multiply the number of intra-socket threads and another one for the inter-socket threads. However, we recommend the analysis based on equations from Table 5 when threads are not expected to be already synchronized.

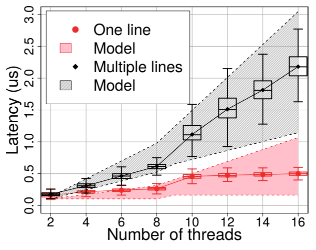
It is also interesting to point out that the single-line approach is faster for intra-socket transfers, and the different-

TABLE 5
Models for many-to-one synchronization patterns.

		Intra-socket/ Xeon Phi		Inter-socket	
Same Line	$T_{min}(n)$	$R_I + nR_R$	(6)	$R_I + sR_R + qR_Q$	(7)
	$T_{max}(n)$	$R_I + 2nR_R$	(8)	$R_I + 2(s+q)R_Q$	(9)
Different Lines	$T_{min}(n)$	$R_I + nR_R$	(10)	$R_I + sR_R + qR_Q$	(11)
	$T_{max}(n)$	$nR_I + 2nR_R$	(12)	$(s+q)R_I + 2(sR_R + qR_Q)$	(13)



(a) Many-to-one



(b) One-to-many

Fig. 6. Synchronizations on Sandy Bridge.

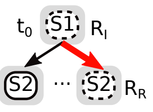
```
Algorithm 3: One-to-many
if $me==t_0$ then
[S1]    cl_write(&flag,1);
else
[S2]    cl_wait(&flag,1);
end
```

(a) One-to-many (same line)

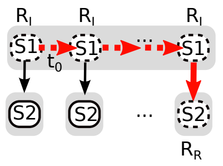
```
Algorithm 4: One-to-many
if $me==t_0$ then
    for $i$=0...n, $i \neq t_0$ do
[S1]        cl_write(flag[i],1);
    end
else
[S2]    cl_wait(flag[me],1);
end
```

(b) One-to-many (different lines)

Fig. 8. Pseudo-code for one-to-many synchronization patterns.



(a) One-to-many (same line)



(b) One-to-many (different lines)

Fig. 9. CLa graphs for basic synchronization patterns.

lines approach is the best for inter-socket transfers. This made us consider the use of one line per socket to isolate polling from separate sockets, identifying the technique *Single-use synchronization lines*.

Given the high number of cores of the Xeon Phi and the serialization involved in both approaches (writes to the same flag, or reads of multiple lines), we explore a notification tree in which each group of children and parent behaves like the single-line synchronization. In order to find the best tree, the analysis of all possible trees becomes prohibitive with more than 20 threads. Hence, we apply the following heuristic: for a given number of threads, we analyze (1) the number of sons of the root, and (2) the organization of the remaining threads in groups. For each group, we use the tree obtained previously for this number of threads. Each first level of a subtree is equivalent to the CLa graph of Figure 5a. Different subtrees use different flags (single-use synchronization lines), hence there is not line stealing between subtrees. Figure 7b shows the results and the model of this algorithm. Latency and variability are lower than in the basic versions (Figure 7a).

3.2 One-to-many

In one-to-many synchronizations, t_0 notifies n threads. We analyze two options: (1) n threads read the *same line* that thread t_0 writes (Figure 8a) resembling the n-readers case with stealing, and (2) t_0 writes n *different lines* and each thread reads its own (Figure 8b). We construct the CLa graphs in Figure 9 to derive the models. The figure shows the cost of the model for Sandy Bridge (intra socket). For Xeon Phi, we need to take into account the contention of all threads reading a single line. Table 6 shows the models derived for Sandy Bridge and Xeon Phi. Regarding the inter-socket scenario, we consider $n = s + q$ threads, where s is the number of readers within t_0's socket and q is the number of readers from the other socket.

The results for Sandy Bridge in Figure 6b show that the use of one line is faster due to the multiple writes required by the different lines scenario. The same happens for Xeon Phi in Figure 7c. To mitigate Xeon Phi's contention, we explore the use of a notification tree. In this tree, each parent writes i flags and each flag is read by a different group of threads (single-use synchronization lines). Once a thread reads the flag it becomes a notifier and generates its own subtree. The first level of each subtree corresponds to a CLa graph similar to the one in Figure 9a.

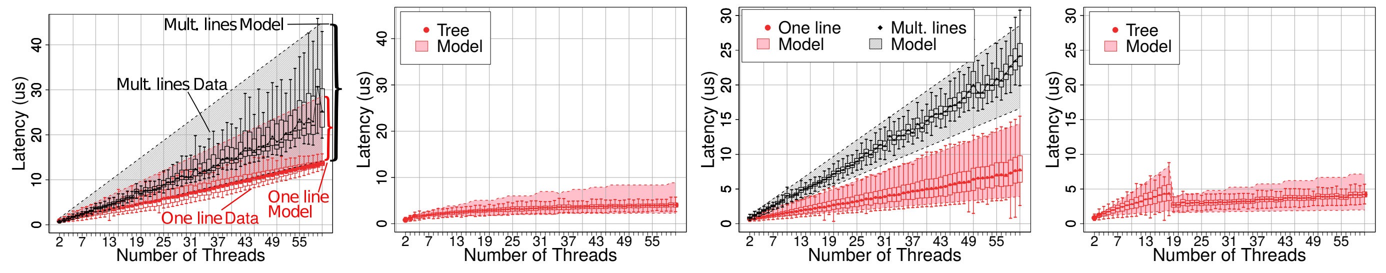
In order to generate the best tree, we need to analyze not only all possible trees, but also all forms of grouping the descendants of each parent. Hence, we work with a simplification that considers that the root writes a flag that is read by a set of descendants. Then, the remaining threads are split evenly among the available parents. The cost is:

$$T_{min} = R_R + T_C(k) + \max_i(T_{min,stree_i})$$
$$T_{max} = R_R + 2kR_R + \max_i(T_{max,stree_i}) \qquad (26)$$
$$k = \text{number of threads that read the first flag}$$

Figure 7d shows the results and the model of this algorithm. Although latency is almost 3 times smaller than in previous examples, the peak for less than 20 threads appears because we optimize for T_{min} and our estimation is conservative in the generation of new stages, which may increase contention.

3.3 Many-to-many

A many-to-many synchronization is a barrier in which every thread blocks until every other thread has reached the barrier function call. We discuss how to optimize a dissemination barrier, which is optimal for single-port LogP systems [15]. We phrase the algorithm with CLa primitives and apply our model to obtain the best configuration for a multi-core system.



(a) Many-to-one. (b) Many-to-one, tree-based. (c) One-to-many. (d) One-to-many, tree-based.

Fig. 7. Synchronization on Xeon Phi KNC.

TABLE 6
Models for one-to-many synchronization patterns.

		Intra-socket		Inter-socket		Xeon Phi	
Same line	$T_{min}(n)$	$R_I + R_R$	(14)	$R_I + R_Q$	(15)	$R_I + \mathcal{T}_C(n_{th})$	(16)
	$T_{max}(n)$	$R_I + nR_R$	(17)	$R_I + nR_R + R_Q$	(18)	$R_I + n_{th}R_R$	(19)
Different lines	$T_{min}(n)$	$nR_I + R_R$	(20)	$(s+q)R_I + R_Q$	(21)	$n_{th}R_I + R_R$	(22)
	$T_{max}(n)$	$R_I + 2sR_R$	(23)	$(s+q)R_I + 2(sR_R + qR_Q)$	(24)	$R_I + 2n_{th}R_R$	(25)

3.3.1 *Intra-socket and Xeon Phi Dissemination*

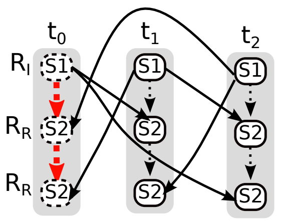
The m-way dissemination algorithm for n threads uses $r = \lceil \log_{m+1}(n) \rceil$ rounds. In every round, each thread sends a notification to m threads (i.e., writes its own flag) and receives m other notifications (i.e., reads m other flags), with cost $R_I + mR_R$. Assuming that *flag[n]* is an array of lines holding one flag per thread, and that each thread has m peers per round, each flag is written r times and read mr times. And each `wait` may interfere with other rounds in which the same flag is written (E4 edges). To minimize line stealing, each thread could use a different flag per peer and round (*flag[n][r][m]*), writing m new flags per round. However, *single-use synchronization lines* form the best tradeoff between stealing and memory overhead at each stage of the algorithm. Each thread writes a different flag per round read by m peers (*flag[n][r]*), as shown in Figure 10a. Here, stealing is limited to the m readers of each round, which is always bounded by $\lceil \sqrt[r]{n} \rceil$ (since $r = \lceil \log_{m+1}(n) \rceil$). 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 full CLa graph is homogeneous (as it happens in Figure 10b with three threads and $m = 2$): all threads perform the same operations per round but with different peers, hence, we just multiply the number of rounds by the cost of each round. The optimization problem to obtain the best m is shown in Equation (27).

Algorithm 5: Dissemination

```
for $r$=0..$\lceil \log_{m+1} n \rceil$-1 do
[S1]  cl_write(flag[me][r],r);
      for i=1..m do
          p=(me-i(m+1)^r);
          p=p mod n;
[S2]      cl_wait(flag[p][r],r,≥);
      end
end
```

(a) Pseudo-code



(b) CLa graph with $n = 3$ and $m = 2$

Fig. 10. Dissemination barrier.

$$\begin{aligned} \underset{r,m}{\text{minimize}} \quad & T_{diss,min}(r,m) = r(R_I + mR_R) \\ \text{subject to} \quad & r = \lceil \log_{m+1}(n) \rceil \quad (m+1)^r \geq n \end{aligned} \tag{27}$$

In the worst case, m readers cause line stealing with every flag. However, this is a too pessimistic scenario and we refine the T_{max} by considering one interference per operation, as shown in Equation (28).

$$T_{diss,max}(r,m) = r(R_I + 2R_R)(m+1) \tag{28}$$

Regarding Xeon Phi, we can use the same model if we do not consider contention: although m threads are accessing each line, threads read different combinations of m lines and in different order. However, since delays in communications may cause several threads to read the same flag at the same time, we also consider the equations with contention in T_{min}, as shown in Equation (29). For T_{max}, we consider interferences that cause threads not to read flags simultaneously, avoiding contention.

$$\begin{aligned} \underset{r,m}{\text{minimize}} \quad & T_{diss,min}(r,m) = r(R_I + m\mathcal{T}_C(m)) \\ \text{subject to} \quad & r = \lceil \log_{m+1}(n) \rceil \quad (m+1)^r \geq n \end{aligned} \tag{29}$$

For Xeon Phi we design and evaluate a generalization of dissemination barrier that uses multiple m values: thread t writes a flag (read by $m_{t,j}$ threads) and reads $m_{t,i}$ flags (with cost $R_I + m_{t,i}R_R$). This reduces communications because in an m-way dissemination the information collected at the end of the algorithm (and, thus, the communications performed) is $(m+1)^r$, while ideally we only need n. In this multi-m dissemination we can obtain a more accurate estimation using $\prod_{i=1}^{r}(m_i+1)$. The implementation and the model in CLa are very close to an m-way dissemination.

3.3.2 *Inter-socket Dissemination*

We compare three variants for inter-socket dissemination, with different trade-offs between line stealing and congestion. First, we use an optimized dissemination barrier, estimating the number of QPI transfers per round with the average of QPI communications per thread. For the worst case, we assume that all lines are stolen through QPI.

A second approach is one barrier per socket with a final exchange of flags using pair-wise communications: each thread selects a peer from the other socket to exchange final flags. Equation (30) shows an estimation for best and worst case, using the cost of the slowest socket plus the final exchange. Parameters r_i and m_i represent the values of r and m per socket, n is the total number of threads and n_i the number of threads in socket i. It has a coarse approach for T_{max} in which all threads cause congestion when reading different flags, and $|n_0 - n_1|$ represents the maximum number of threads reading the same flag when the number of threads is not the same in both sockets, causing line stealing.

$$\begin{aligned} T^{QPI}_{diss1,min} &= max(T_{diss,min}(r_i, m_i)) + R_I + R_Q \\ T^{QPI}_{diss1,max} &= max(T_{diss,max}(r_i, m_i)) + 2(R_I + \\ &\quad 2T_m(n,1)) + |n_0 - n_1|R_Q \\ r_i &= \lceil \log_{m_i+1}(n_i) \rceil, \quad i = 0,1 \end{aligned} \tag{30}$$

Finally, in order to minimize congestion, we can select a root per socket that sets a flag read by all threads from the other socket. The cost is similar to the pair-wise approach but, although we remove congestion, we increase the overheads due to line stealing, as shown in Equation (31).

$$\begin{aligned} T^{QPI}_{diss2,min} &= max(T_{diss,min}(r_i, m_i)) + R_I + R_Q \\ T^{QPI}_{diss2,max} &= max(T_{diss,max}(r_i, m_i)) + 2R_I + nR_Q \\ r_i &= \lceil \log_{m_i+1}(n_i) \rceil, \quad i = 0,1 \end{aligned} \tag{31}$$

3.3.3 *Performance Results*

Figure 11a shows the results of the dissemination barrier on Sandy Bridge using the best m, obtained by estimating the average of QPI transfers. Our algorithm provides a maximum speedup of 26.5x over Open MPI and 12x over Intel MPI. The difference is high because MPI needs to synchronize processes (the implementation could cooperate with the operating system to utilize shared memory for this task). Although Intel MPI shows low latency for 16 threads, it has issues with non-balanced cases like 10 or 12 threads. When compared to a thread library like OpenMP, our algorithm obtains a speedup of 1.7x over GNU OpenMP and up to 3.8x over Intel OpenMP. Since in the HMPI collectives there is no recommendation for barrier, we compare our algorithm with a dissemination barrier with no parameter optimization ($m = 1$) and no single-use flags. In this case the maximum speedup is 1.93x.

Figure 11b compares the three dual-socket approaches in order to analyze the trade-offs between line-stealing and congestion. "Pair-wise" uses a final flag per thread read by a peer from the other socket, suffering from congestion in the QPI link. "Socket-root" uses one root per socket whose final flag is read by all threads from the other socket, suffering from line stealing. We increased the scale of this figure to enable the comparison of the different barriers, thus removing MPI results.

When the difference in the number of threads per socket is high (e.g., 10 threads: 8 threads in one socket and 2 in other) stealing and congestion are limited (in the pair-wise, there is less congestion since there are less different flags, while in the socket-root there is less stealing). Thus, these approaches perform better than the pure dissemination. However, when increasing the number of threads in both sockets, the pair-wise approach suffers from a higher overhead due to the congestion caused by the reads of the different flags, while the increase in line stealing in the socket-root has less impact. Regarding the use of dissemination without parameter optimization ($m = 1$) and no single-use flags, it almost doubles the latency of our optimized barriers.

Figure 11c shows the results for the dissemination on Xeon Phi using the model with contention, providing speedups of up to 3x over OpenMP and 11x over MPI. Our experiments showed that, although the parameters m and r change when considering the model with contention, the average of latency is not significantly affected. However, the model with contention provides parameters that produce performance results with less variability. This is because the use of contention tends to provide $m = 2$ or $m = 1$, increasing the number of rounds but reducing line stealing. Regarding the multi-m dissemination (Figure 11d), it provides speedups of up to 1.28x over dissemination, but contention limits its benefits.

4 Broadcast

A broadcast transfers data from one thread called *root* to n others. We designed an algorithm identifying our next technique *single-step line broadcast*: all children of a given node can copy the data at the same time. However, the more children the root has, the higher the contention, the more lines may be stolen and the more costly the synchronization is: The parent notifies that data is ready (one-to-many, T_{o2m}) and children notify to the parent that they have copied (many-to-one, T_{m2o}) so the parent can free the structure. Hence, we use the CLa performance model to obtain the best broadcast tree.

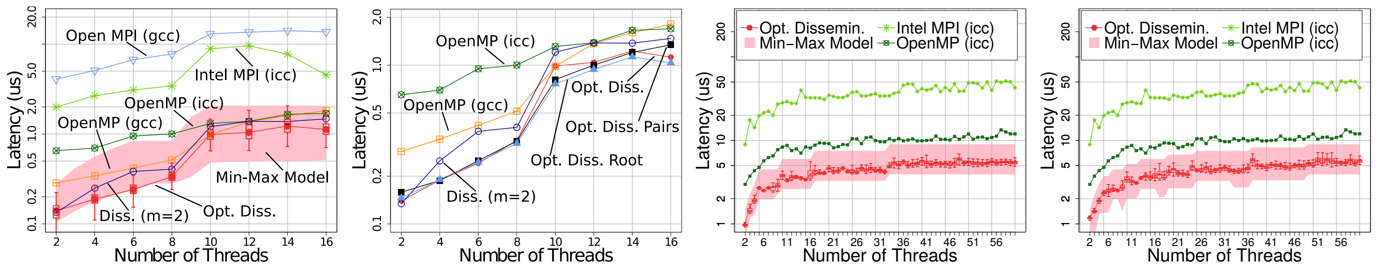
We distinguish two scenarios depending on message size but with the same algorithmic structure: A generic tree in which each node i can use a different number of children (k_i) and all k_i children of thread i copy the same data. We use a tool that generates all structurally equivalent trees [16], calculating the broadcast latency to select the best structure. This tree could change slightly in a scenario with contention like Xeon Phi, in which we may have rounds of children accessing the same data at different stages.

4.1 Single-line Broadcast

When the message is smaller than a line, we use notification with payload for the one-to-many synchronization. Thus, we use the same line approach (Equations (14) and (17)) adding the cost of the parent and the children copying the data to/from this same line. Note that, on Sandy Bridge, there is no contention and, in an intra-socket scenario, threads accessing different data do not cause congestion.

For the many-to-one synchronization, we use the same line approach (Equations (6) and (8)). Although the different variants have the same T_{min}, this version presents lower T_{max}. Figure 12 shows the algorithm with the synchronization modes selected. The first `if` block corresponds to children that wait for their parent's flag and copy the data. In the second one, a parent sets the data and the flag, and waits for its children to copy. In the final block, children notify to their parent that they have copied the data.

For a given tree structure, we construct the CLa graph and search the critical path. Figure 13 shows an example of a four-node binary tree with the costs for the intra-socket Sandy Bridge scenario (the critical path has thicker edges and nodes with dotted circles). The E1s link operations



(a) M-way dissemination on Sandy Bridge. (b) Different inter-socket barriers on Sandy Bridge. (c) M-way dissemination on Xeon Phi KNC. (d) Multi-m dissemination on Xeon Phi KNC.

Fig. 11. Performance results of the barrier. Latency is shown in logarithmic scale.

```
Algorithm 6: One line broadcast
Function OneLineBroadcast(int me, cl_t * mydata, tree_t tree)
    if tree.parent != -1 then
[S1]    cl_wait(tree.pflag[tree.parent],1);          //one-to-many
[S2]    cl_copy(tree.data[tree.parent],mydata,1);
    if tree.nsons > 0 then
[S3]    cl_copy(mydata,tree.data[me],1);
[S4]    cl_write(tree.pflag[me],1);                  //one-to-many
[S5]    cl_wait(tree.sflag[me],tree.nsons);          //many-to-one
    if tree.parent != -1 then
[S6]    cl_add(tree.sflag[tree.parent],1);           //many-to-one
    end
end
```

Fig. 12. One line broadcast in CLa pseudo-code.

within each thread and we use E2s in the synchronizations and data copies. Finally, there is an E3 because t_1 and t_2 write the same flag, where t_0 polls.

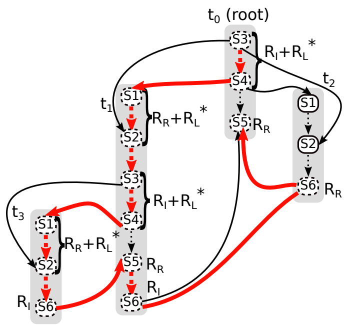


Fig. 13. CLa graph for a one line broadcast using a four-node binary tree. The critical path is calculated for a single-socket Sandy Bridge. Costs with '*' represent situations in which the same thread operates consecutively with the same line and the cost of accessing is counted only once.

Since we use trees, we observe regularities in the critical path: It includes the transfer of data from the root to its children plus the synchronizations ($T_{lev}(k_0) = T_{o2m}(k_0) + T_{data} + T_{m2o}(k_0)$), plus the cost of the most expensive subtree ($T_{bc}(subtree_i)$, in Figure 13 the left subtree). Hence, we can generalize it as shown in Equation (32)[3], where k_i is the number of children of thread i.

$$\underset{k_i}{\text{minimize}}\ T_{bc}(tree) = T_{lev}(k_0) + \max_{i=1,\ldots,k_0}(T_{bc}(subtree_i))$$
$$\text{subject to } T_{bc}(leaf) = 0 \quad \sum_{i=0}^{n} k_i = n,\ k_i \geq 0 \tag{32}$$

4.1.1 Considerations for Sandy Bridge

The minimization has to balance the number of threads that get the value at the same time, and the notification cost. In a multi-socket broadcast some edges become QPI links. Having the tree nodes in an ordered list, we generate permutations to locate the QPI links in different edges[4]. To calculate the cost of each permuted tree and select the best one, we apply Equation (32) considering:

1) Transfers across sockets cost R_Q.
2) To isolate QPI transfers and minimize line stealing, we use *single-use synchronization lines*: each synchronization uses one line per socket.
3) We do not consider QPI congestion caused by different subtrees because this complicates the model and our experiments showed that the benefits are minimal.

The best dual-socket trees for Sandy Bridge are almost always flat. This is because, many-to-one synchronization from threads located in different sockets can overlap if we use separate lines per socket.

Figure 14a shows the performance results of our algorithm compared to two MPI libraries. Broadcasts with an imbalanced number of threads per socket (e.g., ten threads) use different trees depending on the socket where the root is located. Our algorithm clearly outperforms both MPI libraries obtaining a speedup of 14x when compared to Open MPI and 8x relative to Intel MPI. Authors of the NUMA-aware HMPI library recommend the use of a flat tree with synchronization based on barriers. When compared with this approach, we obtain a speedup of up to 1.8x.

4.1.2 Considerations for Xeon Phi

On Xeon Phi, we can apply the same algorithm and equations but considering that the copy of the data involves contention. In order to prune the search of the best tree, we use the same strategy than in the many-to-one synchronization (cf. Section 3.1). For a given number of threads, we analyze the combinations of: (1) number of sons of the root and (2) organization of the remaining threads in subtrees. For each subtree, we select the optimal configuration obtained for its number of threads.

Since Xeon Phi exhibits contention, we consider trees in which each thread can be parent of several groups of threads. The operation of one parent is as follows: (1) It copies the message into a shared location and sets the notification flag. (2) If it has more children, it re-copies the message into another shared location and sets a new flag. And (3) it uses one-line many-to-one synchronization to receive the notification from all its children. In order to generate the

3. If we use a global flag where the root sets the shared structure as occupied by the current operation, we have to add R_I to the equation.

4. We do not need all permutations: there is no difference among threads from the same socket.

trees and select the best one, we use the same simplification than in the one-to-many synchronization (cf. Section 3.2): first the root writes the data and a flag that is read by a set of descendants and then, the remaining threads are split evenly among the available parents. In this algorithm, the root is able to write the new data and flag immediately after writing the previous flag. Hence, its children will receive the message before the rest of subtrees. In order to alleviate this effect, intermediate nodes (with parent and children) can copy directly the parent data into a shared structure from where their children will get it. And then, they will make the copy into its own buffer. Performance results in Figure 15a show up to 4.7x improvement over MPI. The model underestimates the real cost due to the simplification and because the implementation requires some indirections. However, it predicts accurately the performance trends, and the average results are within the min-max model. Our results show that this algorithm shows less variability in performance than the use of simple trees.

4.2 Multi-line Broadcast

With multi-line messages the cost is generally driven by the copy, not by the synchronization. Hence, it is most efficient if children copy data directly from the original location. In this case, T_{data} corresponds to a multi-line transfer. Given the cost of the copy, T_{max} has to consider that children may not copy the message at the same time, delaying the whole algorithm. If data is too big to fit in cache, we split it in chunks that can be cached. At most, one thread will need to maintain its copy and its parent data.

4.2.1 *Considerations for Sandy Bridge*

The multi-line transfer model for T_{data} is Equation (4). In a multi-socket scenario, we ignore QPI congestion as we have not observed a significant performance impact in this algorithm. Figure 14b shows results for an 8 KB broadcast (128 lines) that, when compared to Open MPI and Intel MPI, obtains a speedup of up to 7.3x. Again, HMPI uses a flat tree with barriers. Here, the speedup is lower (up to 1.3x) because the cost is driven by data copies, and our optimization also proposes flat trees in most cases.

4.2.2 *Considerations for Xeon Phi*

Since the children copy the data from the original location (zero-copy strategy), having several groups of descendants from a same parent only limits line stealing and contention when checking the flag. Thus, only we analyze the approach in which each parent has one group of descendants.

The copy of the data (T_{data}) has to combine the multi-line model and the contention model. 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 contention caused by the rest of children. As intercept or constant factor, we arbitrarily chose the b from the contention model (assuming that the buffers are in exclusive state) [8].

The result of this approach is shown in Figure 15b, with a speedup of up to 44.6x over MPI. The cause of the peak for 45 threads is that we optimize for the best case (the min-model does not present any peak).

5 Reduction

Reduction is widely used in parallel programming, e.g., to combine results after splitting computation between threads. Its communication pattern is the opposite to broadcast: the *root* performs an operation with data received from n threads, thus, data from $n + 1$ threads (including the data from the *root*) have to be combined. We explore two different algorithms depending on the message size.

5.1 Single-line Reduction

To avoid the serialization caused if several threads write to a common location, we use a technique that we call *cache line privatization*: each thread copies data to a private buffer, and thus these writes can overlap [8]. Then, the root performs the operation reading these buffers. Although privatization forces the root to read multiple buffers, it compensates the serialization (note that the root is not polling this buffer, so stealing is not an issue). Regarding synchronization, the root has a flag to notify that buffers are ready (one-to-many) and children use another CL to notify that copies have been made (many-to-one), as in Figure 16: When a thread has children (first `if` block), it notifies that it is ready, waits for its children notification, and operates with the content of their buffers. When a thread has a parent (second block), it waits for the parent to be ready, copies its own data into a buffer and notifies its parent.

Algorithm 7: One-line reduction

```
Function OneLineReduction(int me, cl_t * mydata, tree_t tree)
    if tree.nsons > 0 then
[S1]    cl_write(tree.pflag[me],1);           //one-to-many
[S2]    cl_wait(tree.sflag[me],tree.nsons);   //many-to-one
        for i=0,...,tree.nsons-1 do
[S3]        cl_add(mydata, tree.sdata[me][i]);
        end
    if tree.parent != -1 then
[S4]    cl_wait(tree.pflag[tree.parent],1);   //one-to-many
[S5]    cl_copy(mydata, tree.mybuffer, 1);
[S6]    cl_add(tree.sflag[parent],1);         //many-to-one
    end
end
```

Fig. 16. Reduction of a one line message in CLa.

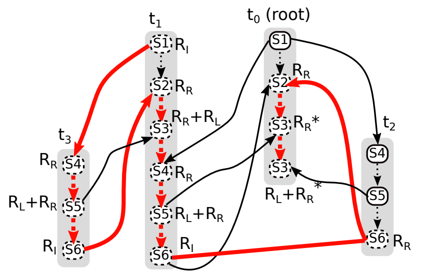
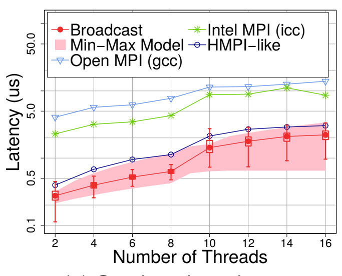
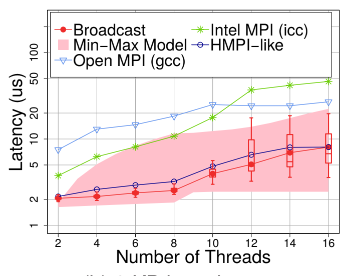


Fig. 17. CLa graph for a one line reduce using a four-node binary tree. The critical path is calculated for a single-socket Sandy Bridge. Costs signaled with '*' represent situations in which the same thread operates consecutively with the same line and the cost of accessing is counted only once.

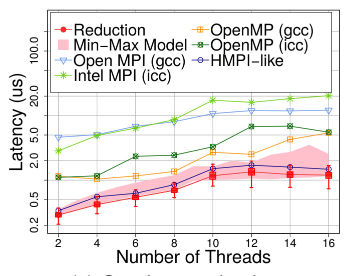
Figure 17 represents the CLa graph of a four-node binary tree, and its critical path. The E1 edges link operations within each thread, E2s appear in the synchronizations and data copies, and the E3 represents that t_1 and t_2 write the same flag, where t_0 polls. As in broadcast, the regularities in the critical path allow us to generalize it for a generic tree



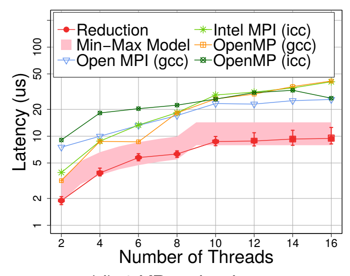
(a) One byte broadcast.



(b) 8 KB broadcast.

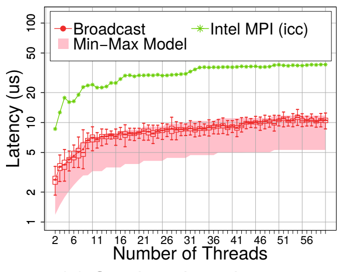


(c) One byte reduction.

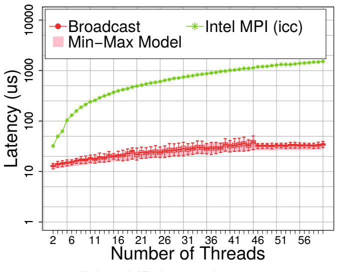


(d) 4 KB reduction.

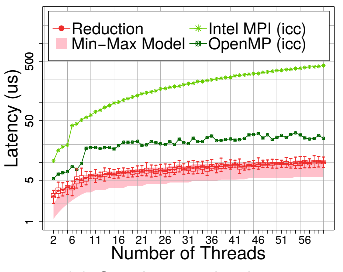
Fig. 14. Performance results on Sandy Bridge. Latency is shown in logarithmic scale.



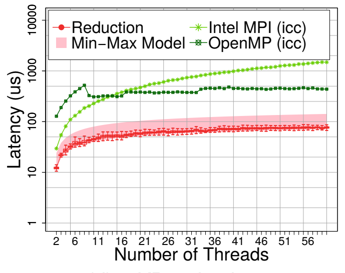
(a) One byte broadcast.



(b) 8 KB broadcast.



(c) One byte reduction.



(d) 4 KB reduction.

Fig. 15. Performance results on Xeon Phi KNC. Latency is shown in logarithmic scale in 15b, 15c, and 15d.

where thread i has k_i children and k_i buffers for data copies. Note that children with cheaper subtrees (like the subtree of t_2) notify first, thus, their notifications to the parent may overlap with the reduction of more expensive subtrees (like the subgraph formed by t_1 and t_3). However, if we do not consider this overlap we still get an upper bound for T_{min}.

5.1.1 *Considerations for Sandy Bridge*

Equations (33) show the cost of a level of the tree (cost of line stealing is reflected in $T_{lev,max}$) and Equation (34) represents our solution (k_i is the number of children of thread i, and k_0 the number of children of the root).

$$\begin{aligned} T_{lev,min}(k) &= R_I + 2((k+1)R_R + R_L) \\ T_{lev,max}(k) &= R_I + 2((2k+1)R_R + R_L) \end{aligned} \tag{33}$$

$$\begin{aligned} &\underset{k_i}{\text{minimize}}\ T_{red}(tree) = T_{lev}(k_0) + \max_{i=1,\dots,k_0}(T_{red}(subtree_i)) \\ &\text{subject to } T_{red}(0) = 0 \quad \textstyle\sum_{i=0}^{n} k_i = n,\ k_i \geq 0 \end{aligned} \tag{34}$$

With two sockets, we use single-use synchronization cache lines to isolate polling from different sockets. As in broadcast, we generate arbitrary trees and permutations of the QPI link locations and we consider QPI congestion for children of the same parent.

Figure 14c shows the latency of a reduction using a sum of floats for our algorithm, MPI and HMPI, and a sum of integers for OpenMP. Our implementation improves MPI libraries by up to 16x and OpenMP ones by up to 5.6x. We observed that Intel libraries present latency peaks for imbalanced scenarios (10 threads for Intel MPI and 12 and 14 for Intel OpenMP). HMPI uses a binary tree, delaying QPI communications to the last step. When comparing our approach with this algorithm (both using our synchronization system), we obtain a speedup of up to 1.3x.

5.1.2 *Considerations for Xeon Phi*

We use the algorithm described for Sandy Bridge and the approach from Section 3.1 to avoid the generation of all possible trees. We modify the model from Equations (33) and (34) to introduce contention in the T_{o2m} synchronization, and we evaluate the impact of considering overlapping among the children of the same parent (which reduces contention and line stealing). This results in trees with larger fan-outs and lower T_{min}, although real results show that the trees obtained with both approaches perform similar: with improvements of up to 41x over MPI and 3.3x over OpenMP.

5.2 Multi-line Reduction

Splitting computation tasks between threads may require a reduction of more than one element. Using a multi-line version of Equation (34) forces each parent to read long buffers in a serialized manner. To maximize overlap, we use binomial trees, applying the model to construct an optimized implementation.

At a given stage of the tree, remaining threads are arranged in pairs: thread t_0 reads and reduces data from thread t_1. Each pair needs two one-to-one synchronizations: From t_1 to t_0 to notify that data is ready, and from t_0 to t_1 to notify that it has been copied and t_1 can leave the operation. After each stage, the number of active threads is halved. Assuming that the overhead of the operation is similar to a copy, the cost of one stage is $2(R_I + R_R) + T_m(1, N)$. In this algorithm, and due to the message size (as in the multi-line broadcast from Section 4.2) line stealing does not capture all the variability and we have to consider the delay that may occur if some transfers do not overlap.

5.2.1 *Considerations for Sandy Bridge*

The T_m estimation uses one buffer to load data from, and another one to store the data. In the multi-line reduction there are two buffers to load data from. Hence, we use $2T_m$ in the approximation. However, even when not considering this difference, we obtain a good prediction of the behavior of the algorithm. Given the data size, simultaneous transfers can congest the QPI link. Thus, we use one binomial tree per socket and a final QPI transfer. Let n_0+1 and n_1 be the number of threads per socket (the root is in socket 0), T_{min} for the binomial reduction is shown in Equation (35).

$$\begin{aligned} T_{stage} &= 2(R_I + R_R) + T_m(1, N) \\ T_{inter} &= 2(R_I + R_Q) + T_m(1, N) \\ T_{bin,min}(n+1) &= lceil \log_2(n+1) \rceil T_{stage} \\ T^{QPI}_{bin,min}(n_0+1, n_1) &= T_{bin,min}(\max(n_0+1, n_1)) + T_{inter} \end{aligned} \tag{35}$$

Figure 14d shows the results of the binomial algorithm compared to MPI and OpenMP. We do not compare with HMPI because they recommend the use of a similar tree. We used 4 KB messages to have all buffers cached. They are smaller than for broadcast because there are more inner copies. Our reduction obtains a speedup of up to 4.33x over MPI libraries and up to 4.8x over OpenMP. OpenMP supports the reduction of a list of variables, but the multi-line reduction is only supported in Fortran versions.

5.2.2 *Considerations for Xeon Phi*

The model is similar than for Sandy Bridge, but a Xeon Phi core makes a 512-bit load and a 512-bit store in one cycle, and its vector instruction set is optimized for three-operand instructions [17], hence, we do not observe the same latency differences regarding T_m. If we want to refine the estimation, we could use $1.5T_m$, but $2T_m$ overestimates the results. Figure 15d shows improvements of up to 13.3x over OpenMP and 20x over MPI.

6 Related Work

Analytical performance models have been largely used to optimize parallel computation. Models like LogP [18], LogGP [19], PlogP [20] or Hockney [21], enable the analysis of algorithm performance in distributed environments. Some of them [22], [23] were extended with memory concerns to measure effects of buffer copies in communication. The PRAM [24] model studies the logical structure of parallel computation, without taking into account communication among processors nor access to global memory. Recent works include multi-core features, like mPLogP [25] for Cell B.E. processors. Others [26], [27] extract features from code, hardware and input data to develop high level models to configure application scheduling on multi-cores.

The use of models for algorithm optimization has been tackled in multiple works. Karp et al. [28] use LogP to show that Fibonacci trees are optimal for broadcast, while Sanders et al. [29] use a simple linear communication model to develop bandwidth-optimal broadcast and reduction algorithms. Li et al. [13] tackle NUMA-aware MPI collectives using a simplified model. Mellor-Crummey et al. [30] analyze different synchronization algorithms in a multiprocessor environment but with almost no cache coherence concerns. Hijma et al. [31] provide optimization guidelines in several architectures but avoid cache coherence issues.

Regarding cache coherence, most works focus on memory hierarchy and cache conflicts [32], [33], [34]. David et al. [35] analyze lock synchronization for multi-cores, considering cache states and memory operations. Putigny et al. [36] use benchmarking to model algorithms but limit the analysis to one-socket Sandy Bridge architectures and obviate thread interaction. Our performance model extends our previous works [7], [8], a cache coherence based model for homogeneous many-core processors and hierarchical NUMA machines. We focus on analytical optimization, generalizing its applicability and the algorithm design.

7 Discussion and Conclusions

While cache coherence simplifies the management of synchronization and communication between cores, it exhibits complex performance properties and thus complicates high-performance code design. We address this issue with cache line aware (CLa) optimizations, a semi-automatic design and optimization method that eases the translation of an algorithm to a performance model in a systematic manner. We demonstrate algorithm development techniques for CLa that improve performance between 1.3x and 44.6x in comparison to highly-optimized vendor-provided communication libraries.

One of the main difficulties for scalability is dealing with thread interaction, which is inherent to concurrency and hidden by the cache coherence protocols. CLa design enables to quantify and localize these interactions that may harm performance severely. Using CLa graphs, we can locate contention (threads accessing the same address at the same time), congestion (threads accessing different addresses simultaneously), and line stealing. And the min-max models present the expected variability and predictability of the algorithm. Kernels or primitives with shared variables and thread interaction are the algorithmic parts that will benefit the most from the use of our methodology. Note that CLa is useful to identify interactions that affect the same line but it relies on the designer to decide which variables share or do not share lines.s

The insight gained with the CLa methodology enables us to identify good design practices (single-use synchronization lines, single-step broadcast, line privatization) and quantify the benefits of these techniques in the different architectures. These design practices are oriented to bound the variability caused by thread interaction, thus reducing the distance between the min and max models. Moreover, by parametrizing the building blocks of the performance model, hardware designers could quantify the impact that architectural design decisions might have in shared memory algorithms.

We demonstrate the use of CLa to optimize algorithms for two architectures: the x86 accelerator Xeon Phi and the dual-socket NUMA processor Sandy Bridge. The homogeneity of the Xeon Phi ring highly simplifies algorithm optimization in terms of modeling. However, the impossibility of taking advantage of locality makes communication among cores much slower than in Sandy Bridge. Moreover, the DTD contention limits the applicability of the single-step broadcast technique. Regarding the Sandy Bridge architecture, the main drawback is congestion through the QPI link and the cost of inter-socket line stealing.

This work is a major step in order to construct a full automatic optimization system for multi- and many-core architectures, to enable the location and quantification of performance and scalability bottlenecks. We aim to introduce refinements in the min-max models to be able to predict the expected behavior, as well as to include memory and TLB concerns to be able to optimize a wider range of algorithms and kernels in multiple architectures.

Acknowledgments

We thank the support from Alexander Supalov and Robert Blankenship from Intel. This work was supported by the Ministry of Economy and Competitiveness of Spain and FEDER funds of the EU (Project TIN2013-42148-P).

References

[1] S. Saini *et al.*, "Performance Evaluation of the Intel Sandy Bridge Based NASA Pleiades Using Scientific and Engineering Applications," in *Proc. 4th Intl. WS. on Perf. Modeling, Bench. and Sim. of HPC Systems (PMBS'13)*, Denver, CO, USA, 2013.

[2] D. Molka *et al.*, "Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System," in *Proc. 18th Intl. Conf. on Parall. Arch. and Compilation Techniques (PACT'09)*, Raleigh, NC, USA, 2009, pp. 261–270.

[3] D. Hackenberg *et al.*, "Comparing Cache Architectures and Coherency Protocols on x86-64 Multicore SMP Systems," in *Proc. 42nd Annual IEEE/ACM Intl. Symp. on Microarch. (MICRO'42)*, New York, NY, USA, 2009, pp. 413–422.

[4] Intel, "Intel® Xeon Phi™ Coprocessor: Software Developers Guide," 2014.

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

[6] T. Hoefler and T. Schneider, "Optimization Principles for Collective Neighborhood Communications," in *Proc. 25th ACM/IEEE Intl. Supercomp. Conf. for High Perf. Comp., Networking, Storage and Analysis (SC'12)*, Salt Lake City, UT, USA, 2012.

[7] S. Ramos and T. Hoefler, "Cache Line Aware Programming for ccNUMA Systems," in *Proc. 24th Intl. Symp. on High-perf. Parall. and Distrib. Comp. (HPDC'15)*, Portland, OR, USA, 2015, pp. 85–88.

[8] ——, "Modeling Communication in Cache-coherent SMP Systems: a Case-study with Xeon Phi," in *Proc. 22nd Intl. Symp. on High-perf. Parall. and Distrib. Comp. (HPDC'13)*, New York, NY, USA, 2013, pp. 97–108.

[9] Intel, "Intel® 64 and IA-32 Architectures Optimization Ref. Manual," 2014.

[10] V. Volkov, "Intro to MIC performance," BeBOP meeting, http://www.cs.berkeley.edu/~volkov/volkov12-MIC.pdf, 2012.

[11] R. Dolbeau, "Address Selection for Efficient Barriers on the Intel Xeon Phi," CAPS Enterprise white paper, http://www.dolbeau.name/dolbeau/publications/barrierphi.pdf, 2013.

[12] J. Torrellas *et al.*, "False Sharing and Spatial Locality in Multiprocessor Caches," *IEEE Trans. on Computers*, vol. 43, no. 6, pp. 651–663, 1994.

[13] S. Li *et al.*, "NUMA-aware Shared-memory Collective Communication for MPI," in *In Proc. 22nd Intl. Symp. on High-perf. Parall. and Distrib. Comp. (HPDC'13)*, New York, NY, USA, 2013, pp. 85–96.

[14] S. Ramos and T. Hoefler, "Benchmark Suite for Modeling Intel Xeon Phi," http://gac.des.udc.es/~sramos/xeon_phi_bench/xeon_phi_bench.html, 2012.

[15] T. Hoefler *et al.*, "Fast Barrier Synchronization for InfiniBand," in *Proc. 20th IEEE Intl. Parall. & Distrib. Processing Symp., CAC'06 WS.*, Rhodes, Greece, 2006.

[16] G. Li and F. Ruskey, "Advantages of Forward Thinking in Generating Rooted and Free Trees," in *Proc. 10th ACM-SIAM Symp. on Discrete Alg. (SODA'99)*, Baltimore, MD,USA, 1999, pp. 939–940.

[17] R. Rahman, "Intel® Xeon Phi™ Coprocessor Vector Microarchitecture," 2012.

[18] D. Culler *et al.*, "LogP: towards a Realistic Model of Parallel Computation," *SIGPLAN Not.*, vol. 28, no. 7, pp. 1–12, 1993.

[19] A. Alexandrov *et al.*, "LogGP: Incorporating Long Messages into the LogP Model - One Step Closer towards a Realistic Model for Parallel Computation," in *Proc. 7th ACM Symp. on Parall. Alg. and Arch. (SPAA'95)*, S. Barbara, CA, USA, 1995, pp. 95–105.

[20] T. Kielmann *et al.*, "Fast Measurement of LogP Parameters for Message Passing Platforms," in *Proc. 15th IPDPS WS. on Parall. & Distrib. Processing*, Cancun, Mexico, 2000, pp. 1176–1183.

[21] R. W. Hockney, "The Communication Challenge for MPP: Intel Paragon and Meiko CS-2," *Parall. Comp.*, vol. 20, no. 3, pp. 389 – 398, 1994.

[22] K. W. Cameron *et al.*, "lognP and log3P: Accurate Analytical Models of Point-to-Point Communication in Distrib. Systems," *IEEE Transactions on Computers*, vol. 53, no. 3, pp. 314–327, 2007.

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

[24] R. M. Karp and V. Ramachandran, "A Survey of Parallel Algorithms for Shared-Memory Machines," Univ. of California, Berkeley, CA, USA, Tech. Rep., 1988.

[25] L. Li *et al.*, "mPlogP: A Parallel Computation Model for Heterogeneous Multi-core Computer," in *Proc. 10th IEEE/ACM Intl. Conf. on Cluster, Cloud and Grid Comp. (CCGRID'10)*, Melbourne, Australia, 2010, pp. 679–684.

[26] F. Blagojevic *et al.*, "Scheduling Dynamic Parallelism on Accelerators," in *Proc. 6th ACM Conf. Comp. Front. (CF'09)*, Ischia, Italy, 2009, pp. 161–170.

[27] J. E. Savage and M. Zubair, "A Unified Model for Multicore Architectures," in *Proc. 1st Intl. Forum on Next-generation Multicore/Manycore Tech. (IFMT'08)*, Cairo, Egypt, 2008, pp. 9:1–9:12.

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

[29] P. Sanders *et al.*, "Two-Tree Algorithms for Full Bandwidth Broadcast, Reduction and Scan," *Parall. Comp.*, vol. 35, no. 12, pp. 581–594, 2009.

[30] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors," *ACM Trans. on Computer Systems*, vol. 9, no. 1, pp. 21–65, 1991.

[31] P. Hijma *et al.*, "Stepwise-refinement for Performance: a Methodology for Many-core Programming," *Concurrency and Computation*, p. In press, 2015.

[32] A. Agarwal *et al.*, "An Analytical Cache Model," *ACM Transactions on Computer Systems*, vol. 7, no. 2, pp. 184–215, 1989.

[33] L. G. Valiant, "A Bridging Model for Multi-core Computing," *Jnl. of Comp. and Syst. Sciences*, vol. 77, no. 1, pp. 154 – 166, 2011.

[34] D. Andrade *et al.*, "Accurate Prediction of the Behavior of Multithreaded Applications in Shared Caches," *Parall. Comp.*, vol. 39, no. 1, pp. 36 – 57, 2013.

[35] T. David *et al.*, "Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask," in *Proc. 24th ACM Symp. on Operating Systems Principles (SOSP'13)*, Farminton, PA, USA, 2013, pp. 33–48.

[36] B. Putigny *et al.*, "A Benchmark-based Performance Model for Memory-bound HPC Applications," in *Proc. 12th Intl. Conf. High Perf. Comp. & Sim. (HPCS'14)*, Bologna, Italy, 2014, pp. 943–950.

Sabela Ramos received the B.S. (2009), M.S. (2010) and Ph.D. (2013) degrees in Computer Science from the University of A Coruña, Spain. From September 2015, she is a postdoctoral researcher at ETH Zürich, Switzerland. Her research interests are in the area of High Performance Computing, focused on message-passing communications and performance modelling on multi and manycore architectures.

Torsten Hoefler is an Assistant Professor of Computer Science at ETH Zürich, Switzerland. He is also a member of the Message Passing Interface (MPI) Forum where he chairs the "Collective Operations and Topologies" working group. His research interests revolve around the central topic of "Performance-centric Software Development" and include scalable networks, parallel programming techniques, and performance modeling.