Machine-Aware Atomic Broadcast Trees for Multicores

Stefan Kaestle, Reto Achermann, Roni Haecki, Moritz Hoffmann, Sabela Ramos, and Timothy Roscoe, *ETH Zurich*

https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kaestle

This paper is included in the Proceedings of the
12th USENIX Symposium on Operating Systems Design
and Implementation (OSDI '16).

November 2–4, 2016 • Savannah, GA, USA

ISBN 978-1-931971-33-1

Machine-aware Atomic Broadcast Trees for Multicores

Stefan Kaestle, Reto Achermann, Roni Haecki, Moritz Hoffmann, Sabela Ramos, Timothy Roscoe
Systems Group, Department of Computer Science, ETH Zurich

Abstract

The performance of parallel programs on multicore machines often critically depends on group communication operations like barriers and reductions being highly tuned to hardware, a task requiring considerable developer skill.

Smelt is a library that automatically builds efficient inter-core broadcast trees tuned to individual machines, using a machine model derived from hardware registers plus micro-benchmarks capturing the low-level machine characteristics missing from vendor specifications.

Experiments on a wide variety of multicore machines show that near-optimal tree topologies and communication patterns are highly machine-dependent, but can nevertheless be derived by Smelt and often further improve performance over well-known static topologies.

Furthermore, we show that the broadcast trees built by Smelt can be the basis for complex group operations like global barriers or state machine replication, and that the hardware-tuning provided by the underlying tree is sufficient to deliver as good or better performance than state-of-the-art approaches: the higher-level operations require no further hardware optimization.

1 Introduction

This paper addresses the problem of efficiently communicating between cores on modern multicore machines, by showing how near-optimal tree topologies for point-to-point messaging can be derived from online measurements and other hardware information.

The problem is important because parallel programming with message-passing is increasingly used inside single cache-coherent shared-memory machines, modern safe concurrent programming languages, runtime systems, and for portability between single-machine and distributed deployments.

This in turn leads to the need for distributed coordination operations, e.g. global synchronization barriers or agreement protocols for ensuring consistency of distributed state, to be implemented over message-passing channels. Efficient use of these channels becomes critical to program performance.

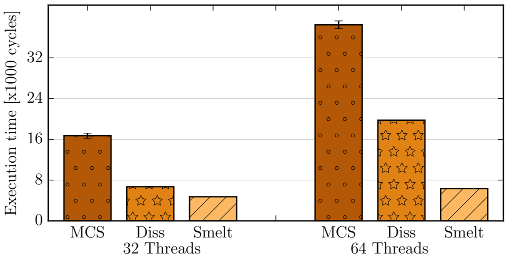


Figure 1: Comparison of thread synchronization using different barriers on Intel Sandy Bridge 4x8x2 with and without Hyperthreads. Standard error is < 3%.

The problem is hard because modern machines have complex memory hierarchies and interconnect topologies. Significant latency improvements for group operations can result from careful layout and scheduling of messages. Unlike classical distributed systems, the extremely low message propagation times within a machine mean that small changes to message patterns and ordering have large effects on coordination latency. Worse, as we show in our evaluation (§5.1), different machines show radically different optimal layouts, and no single tree topology is good for all of them.

In response, we automatically *derive* efficient communication patterns tuned for each particular machine based on online measurements of the hardware and data from hardware discovery. We realize this technique in Smelt, a software library which builds efficient multicast trees without manual tuning. Smelt serves as a fundamental building block for higher-level operations such as atomic broadcast, barriers and consensus protocols.

Smelt provides significant performance gains. Despite other modern barrier operations being highly tuned monolithic implementations, Smelt barriers are constructed over the multicast tree without the need for further hardware-specific tuning. Even so, they provide 3× better performance than a state-of-the-art shared-memory dissemination barrier, and is up to 6× faster than an MCS-based barrier (Figure 1) in our experiments (§5).

In the next section we further motivate this problem, and discuss the unexpected challenges that arise in solving it efficiently on real hardware platforms. We then describe Smelt's design (§3) and give details on its implementation (§4). We evaluate Smelt with a set of micro-benchmarks, existing runtime systems and applications in §5.

2 Motivation and background

We motivate our work in this section by first surveying the trend of building parallel programs using message-passing rather than shared-memory synchronization, even in a single cache-coherent machine. We then discuss messaging in a multicore machine, and how efficient implementations of group communication operations depend critically on the characteristics of complex and diverse memory systems. We then survey common tree topologies used in large computers for communication as a prelude to our discussion of Smelt in the next section.

2.1 The move to message passing

Modern high-end servers are large NUMA multiprocessors with complex memory systems, employing cache-coherency protocols to provide a consistent view of memory to all cores. Accessing shared data structures (e.g. shared-memory barriers or locks) thus entails a sequence of interconnect messages to ensure all caches see the updates [9]. This makes write-sharing expensive: cache lines must be moved over the interconnect incurring latencies of 100s of cycles. Atomic instructions like compare-and-swap introduce further overhead since they require global hardware-based mutual exclusion on some resource (such as a memory controller).

This has led to software carefully laying out data in memory and minimizing sharing using techniques like replication of state, in areas as diverse as high-performance computing [32], databases [35], and operating systems [3]. Systems like multikernels eschew shared-memory almost entirely, updating state through communication based on message-passing.

There are other reasons to use local message-passing. Several modern systems languages either have it as a first-class language feature (like Go) or impose strong restrictions on sharing memory access (like Rust).

Furthermore, while contemporary machines are mostly coherent, future hardware might not even provide globally shared *non*-coherent memory [13]. Here, efficient message-passing is not merely a performance optimization – it is required functionality. The same is true today for programs that span clusters of machines. A single paradigm facilitates a range of deployments.

Ironically, most NUMA message-passing mechanisms today use cache-coherence. With few exceptions [4], multicore machines provide no message-passing hardware. Explicit point-to-point message channels are implemented above shared-memory such that a cache line can be transferred between caches with a minimum number of interconnect transactions. Examples are URPC [5], UMP [2] and FastForward [16]; in these cases, only two threads (sender and receiver) access shared cache lines.

2.2 Communication in multicores

While cache-coherency protocols aim to increase multicore programmability by hiding complex interactions when multiple threads access shared-memory, this complexity of the memory hierarchy and coherency protocol makes it hard to reason about the performance of communication patterns, or how to design near-optimal ones.

The protocols and caches also vary widely between machines. Many enhancements to the basic MESI protocol exist to improve performance with high core counts [18, 28], and interconnects like QPI or HyperTransport have different optimizations (e.g., directory caching) to reduce remote cache access latency.

Worse, thread interaction causes performance variabilities that prevent accurate estimation of communication latency. For example, when one thread polls and another writes the same line, the order in which they access the line impacts observed latency.

Prior work characterized [29] and modelled [32] coherence-based communication, optimizing for group operations. However, these models require fine-grained benchmarking of the specific architectures, providing more accurate models but less portable algorithms.

Smelt is much more general: we abstract coherence details and base our machine model on benchmark measurements, simplifying tree construction while still adapting to underlying hardware. We show that sufficient hardware details can be obtained from a few microbenchmarks which are easily executed on new machines without needing to understand intricate low-level hardware details.

2.3 Group communication primitives

In practice, message-passing is a building block for higher-level distributed operations like atomic broadcast, reductions, barriers or agreement. These require messages to be sent to many cores, and so they must be sent on multiple point-to-point connections.

The problem described above is therefore critical, particularly in complex memory hierarchies. A large coherent machine like an HP Integrity Superdome 2 Server has hundreds of hardware contexts on up to 32 sockets, with three levels of caching, many local memory controllers, and a complex interconnection topology.

Consider a simple broadcast operation to all cores. Baumann et *et al.* [2] show how a careful tree-based approach to broadcast outperforms and out-scales both sequential sends *and* using memory shared between all recipients. Both the topology and the *order* to send messages to a node's children are critical for performance.

The intuition is as follows: unlike classical distributed systems, message propagation time in a single machine is negligible compared to the (software) send- and receive time as perceived by the sender and receiver. The store operation underlying each send operation typically takes a few hundred cycles on most machines. Loads on the receiver wait until the cache line transfer is done, which is between 300 and 1000 cycles depending on the machine. Since sequential software execution time therefore dominates, it is beneficial to involve other cores quickly in the broadcast and exploit the inherent parallelism available. Figure 2 shows an example of such a hierarchical broadcast in a 8-socket machine.

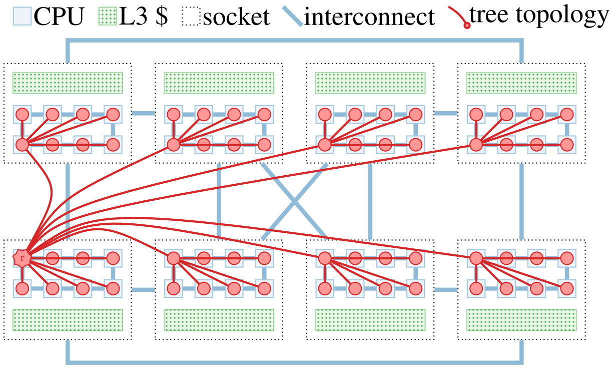


Figure 2: Multicore with message-passing tree topology

The cost of sending and receiving messages between cores is a subtle machine characteristic typically not known to programmers, and depends both on the separation of cores in the machine hierarchy and on more complex subtleties of the cache-coherency protocol. Very little of this information is provided by hardware itself (e.g. in the form of registers with hardware features) and vendor specifications are often incomplete or vague. Worse hardware *diversity* is increasing as much as complexity.

Despite this, prior work has built machine-optimized broadcast trees (e.g. Fibonacci trees [8]), and MPI libraries provide shared-memory optimizations for group or collective operations which try to account for NUMA hierarchies [25] using shared-memory communication channels [17]. Our results in §5 show that these trees are sometimes good, but there is no clear winner across all our machines we used for evaluation.

2.4 Common tree topologies

We now introduce the tree topologies we evaluate in this paper. The first three are hardware-oblivious, constructed without accounting for the underlying topology. We provide a visualization for each of these on all evaluation machines: http://machinedb.systems.ethz.ch/topologies ↗[1].

Binary trees have each node connected to at most two children. Here, each node n connects to nodes $2n + 1$ and $2n + 2$. Such trees often introduce redundant cross-NUMA links, causing unnecessary interconnect traffic. Performance is suboptimal since the low fanout (two) often means that nodes become idle even when they could further participate in the protocol. ↗

Fibonacci trees [22] are widely used unbalanced trees: left-hand subtrees are larger than right-hand ones. In contrast to binary trees, this imbalance allows more work to be executed in sub-trees that receive messages earlier, which prevents nodes from being idle when they could otherwise further participate in the broadcast. However, like binary trees they also have a fixed fanout and can exhibit redundant cross-NUMA transfers. ↗

Sequential trees are also widely used: a root sends a message to each other node sequentially (star-topology). Send operations are not parallelized since one node does all the work. This scales poorly for broadcasts on most large multicore machines. ↗

The other three trees we compare with consider machine characteristics in their construction:

Minimum spanning trees (MSTs) use Prim's algorithm [31] by adding edges in ascending order of cost until the graph is connected. This minimizes expensive cross-NUMA transfers, but does not optimize fanout and hence send parallelism. Among others, the resulting topology can be a star or a linear path, which both are purely sequential in sending. ↗

Cluster trees are built hierarchically, as in HMPI [25]. A binary tree is built between NUMA nodes, and messages are sent sequentially within a node. ↗

Bad trees are a worst-case tree example, built by running an MST algorithm on the inverse edge costs, maximizing redundant cross-NUMA links. We use this to show that the topology matters, and choosing a sub-optimal tree can be as bad as sequentially sending messages on some machines. ↗

3 Design

We now elaborate on the design considerations for Smelt and describe our multicore machine model (§3.1) including its properties and assumptions. Next, we show how we populate it (§3.2) and how our adaptive tree is built (§3.3). In §3.4, we will further show that exhaustive search is not a viable solution for finding the optimal broadcast tree.

Smelt is a library which simplifies efficient programming of multicore machines by providing optimized atomic broadcast trees adapted to the hardware at hand. The tree topology and sending order is generated automatically from a machine model constructed from information given by the hardware itself and a set of fine-grained micro-benchmarks.

Surprisingly, the trees derived by Smelt also function

[1] We refer to http://machinedb.systems.ethz.ch with the symbol ↗

well as building blocks for higher-level protocols – for example, given a Smelt tree, an efficient barrier can be implemented in two lines of code and performs as well as or better than state-of-the-art shared-memory barriers written and hand-tuned as monolithic components.

As a further evaluation of the applicability of Smelt, we also re-implemented the 1Paxos [10] consensus algorithm using Smelt trees (§5.6), which can be used to provide consistent updates on replicated data. Further, we build a key-value store on top of that (§5.7), which provides good performance for consistently replicated data.

Current server machines, of course, do not exhibit partial failure, and so our agreement protocol should be considered a proof-of-concept rather than a practical tool. However, we note that even in the absence of failures, replication within a multicore can be useful for performance [21, 35]. Coordination and synchronization would then still be needed to provide a consistent view of the system state. However, it seems likely that future machines will expose partial failures to programmers [13], requiring replication of services and data also for fault-tolerance.

Smelt combines various tools to a smart runtime system: (*i*) a machine model with micro-benchmarks to capture hardware characteristics, (*ii*) a tree generator for building optimized broadcast trees and (*iii*) a runtime library providing necessary abstractions and higher-level building blocks to the programmer. We visualize this in Figure 3.

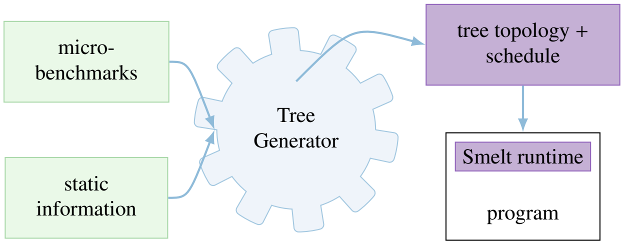


Figure 3: Overview of Smelt's design

3.1 Modelling broadcasts on multicore

Figure 4 visualizes a timeline for message-passing on a multicore system, specifically the time that it takes for a thread v_i to send a message to two threads v_j and v_k.

Unlike classical networks, t_{send} and $t_{receive}$ times dominate the total transmission time. We show this in our pairwise send and receive time (§3.2). This is significant as the sending and receiving threads are blocked for t_{send} and $t_{receive}$ respectively. It implies that the cost of sending n messages grows linearly with the number of messages to be sent, whereas in classical distributed systems, $t_{propage}$ dominates independently of how many messages are sent in a single round trip.

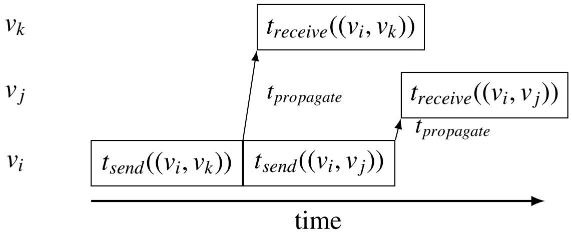


Figure 4: Visualization of a send operation: thread v_i sends a message to v_k followed by another message to v_j. Send operations are sequential, while the receive operations can be processed in parallel on threads v_j and v_k.

Multicore machine model: Communication in a multicore machine can be represented as a fully connected graph $G = (V, E)$. Vertices v_i correspond to threads, and edges $e = (v_i, v_j)$ model communication between threads v_i and v_j, with edge weights as a tuple $w(e) = (t_{send}, t_{receive})$. We show an example of such a graph in Figure 5. We now define t_{send}, $t_{receive}$ and $t_{propage}$:

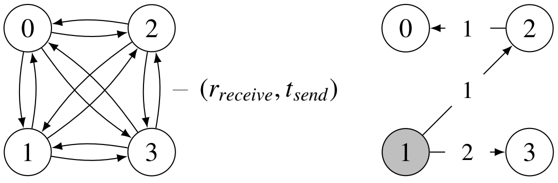
$t_{send}(e)$ denotes the time to send a message over an edge $e = (v_i, v_j)$ from sender v_i to receiver v_j. The sender v_i is blocked during this time. Sending a message involves invalidating the cache line to be written and therefore often depends on which cores the cache line is shared with and also depend on the sequence of previously sent messages at the sender. Moreover, it may vary with the state of the cache line to be written.

$t_{receive}(e)$ denotes the time to receive an already-queued message. The receiver is blocked while receiving the message. In many cache-coherency protocols, receiving (reading) changes the state of the cache line in the receiver's cache from invalid to shared, and from modified to shared in the sender.

$t_{propagate}(e)$ is the time it takes to propagate a message on the interconnect. Propagation time is neither visible on the sender nor receiver. We assume $t_{propagate} = 0$, as propagation time can be seen as part of the $t_{receive}$.

Our model is similar to the telephone model [38], with a few differences: In the telephone model, each participant has to dial other participants sequentially before transmitting data. Similarly, the Smelt model has a sequential component when sending: the thread is blocked for the duration of t_{send}, and consecutive sends cannot be executed until the previous ones are completed. However, note that several threads can send messages concurrently and independently of each other.

The weight of edges in our model is non-uniform: the cost of receiving and sending messages from and to cores that are further away (NUMA distance) is higher. This



input graph G

example output tree τ

Figure 5: Example of fully connected input graph for four CPUs and one possible resulting broadcast tree topology τ with root 1 and send order as edge weights.

is even true for sending messages, since cache lines may have to be invalidated on the receiver before they can be modified on the sender.

Output The desired output consists of three parts: (*i*) A root v_{root}, (*ii*) a tree $T = (V, E')$ with $E' \subseteq E$, where T is a spanning tree of G, (*iii*) edge priorities $w'(v)$ representing the order in which a vertex v sends messages to its children. T is to be chosen such that the latency $lat(T)$ is minimal. The latency is given recursively as

$$lat(T) = \max(\forall v \in V : lat(T, v))$$

with

$$lat(T, v_{root}) = 0$$
$$lat(T, v) = lat(T, v_p) + \sum_{i=1}^{k} t_{send}(v_p, v_i)$$

with v_p being parent of v and v the k-th child of v_p. Note that this latency includes the send-cost for $k-1$ children that v_p sends a message to first.

3.2 Populating the machine model

We derive the input values for our machine model from various sources: libraries such as `libnuma` [36], tools like `likwid` [34], or special OS provided file systems like `/proc` and `/sys` on Linux. The OS and libraries obtain their information by parsing ACPI [20] tables like the static resource affinity table (SRAT) and system locality information table (SLIT), which, for example, provide the NUMA configuration. However, this information is coarse-grained and insufficient for our purposes.

To address this, Smelt enriches relevant static machine information with a carefully chosen set of micro benchmarks that capture relevant hardware details that cannot be inferred from this static information.

Pairwise send and receive time (t_{send} and $t_{receive}$) Motivated by §3.1, we measure the pairwise send (t_{send}) and receive ($t_{receive}$) latency between all hardware threads in the system. Figure 6 shows a visualized output of this benchmark on a 32-core AMD machine (A IL 4x4x2 in Table 1). Both the receive and send time clearly show the NUMA hierarchy of this eight node system. Note that the receive costs are asymmetric: $t_{receive}$ does not only depend on the NUMA distance but also on the direction i.e. $t_{receive}(v_i, v_k) \neq t_{receive}(v_k, v_i)$. This observation is inline with [24].

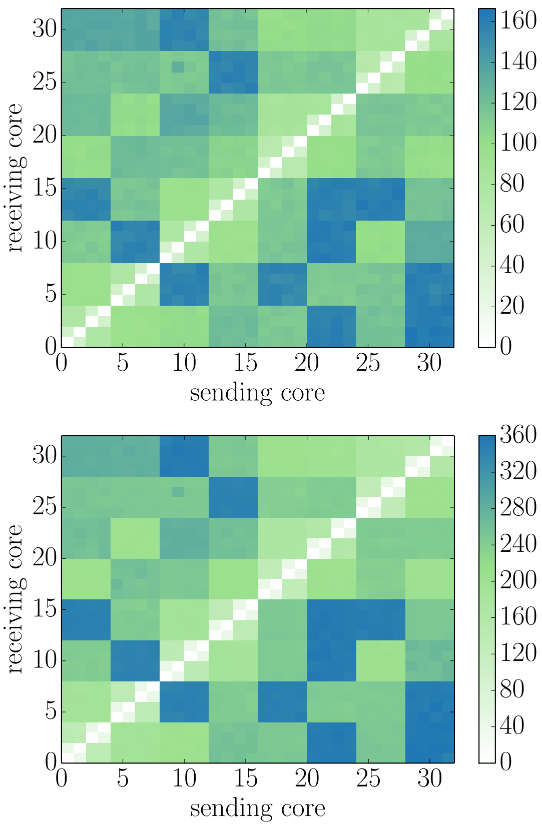


Figure 6: Pairwise send (upper) and receive time (lower) on A IL 4x4x2.

The send cost is smaller compared to receive and (on this machine) shows the same NUMA hierarchy. Note that we measure the cost of sending batch of 8 messages and take the average to compensate for the possible existence of a hardware write buffer: software is only blocked until the store is buffered (and not until the cache line is fetched). This hides the full cost of the cache-coherency protocol. Ideally, Smelt would measure the effect of the write buffer as well. This makes benchmarking significantly more complex as the cost of sending a message between two cores would also depend on the cost of previous send operations and the ocupancy of the write buffer.

Fortunately, write buffers are relatively small, so that their effects do not change the runtime behavior significantly. We keep such a benchmark open for future work and use the more simple batch sending approach instead.

Data from our pairwise send- and receive experiments allow us to predict the time a core is busy sending or receiving a message and when it will become idle again. The busy/idle pattern of the cores is essential to decide which topology to use and the send-order of messages. Our pairwise benchmark works independently of the cache-coherency protocol as it determines the cost of sending and receiving messages using the same software-message-passing library and hence including all overheads induced by interconnect transfers triggered by the cache-coherency protocol implemented on each machine.

Pairwise benchmarks for all our machines are available on our website (http://machinedb.systems.ethz.ch) and the pairwise benchmark's code can be found on http://github/libsmelt.

3.3 Tree generation: adaptive tree

We now describe how Smelt generates trees automatically based on the machine model. Since the tree topology for multicast operations depends on which cores are allocated for an application, a new tree has to be generated whenever an application is started. Smelt generates the tree topologies offline in a separate tree generation step. Currently, this is done in a separate process when the application is initialized, but we anticipate that generating a tree topology will be a frequent operation on future machines as a consequence of reconfiguration in the case of failure or if group communication membership changes. We show the tree topologies generated for each of our machines on http://machinedb.systems.ethz.ch ↗.

3.3.1 Base algorithm

When building our adaptive tree, we rely on the performance characteristics described in §3.2, and make use of the fact that the tree generator can defer a global view of the system state from message send and receive times. For example, it knows which messages are in transit and which nodes are idle. The tree generator operates as an event-based simulation using our pairwise measurements. Whenever a core is idle, it uses the model to choose the next core to send the message to.

The desired output of the tree generator is (*i*) the root of the tree, (*ii*) a spanning tree connecting all nodes, and (*iii*) a schedule that describes the send-order in each node. If not specified by the application, we select the root to be the thread having the lowest average send cost to every other node in the system. The task then is to design a good heuristic to find near-optimal solutions for finding the tree topology. For example, messages can first be sent on expensive links to minimize the cost of the link that dominates the broadcast execution time. Alternatively, we can send on cheap links first to increase the level of parallelism early up. We found that for current multicore machines, it is more important to send on expensive links first: local communication is comparably fast, so messages can be distributed locally and efficiently once received on a NUMA node.

Another trade-off is between minimizing expensive cross-NUMA links and avoiding nodes being idle. For the machines we evaluated, we found that there is little or no benefit from sending redundant cross-NUMA messages even if cores are otherwise idle. For the few exceptions, our optimizations described in §3.3.2 detect opportunities for additional cross-NUMA links and adds them iteratively later where appropriate. Our heuristics are as follows:

- Remote cores first: We prioritize long paths as they dominate the latency for broadcasts. We select the cheapest core from the most expensive remote NUMA node. Local communication is executed afterwards, since this is relatively cheap.
- Avoid expensive communication links: We send the message to a remote NUMA node only if no other core on that node has received the message. We can do this because our tree generator has global knowledge on the messages in flight. This minimizes cross-NUMA node communication.
- No redundancy: We never send messages to the same core twice. The tree generator knows which messages are in flight and will not schedule another send operation to the same core.
- Parallelism: We try to involve other nodes in the broadcast as much as possible. The challenge here is to find the optimal fan-out of the tree in each node. The result often resembles an imbalanced tree so that cores that received a copy of the message early have a larger sub-tree than later ones.

We describe the tree generation in detail in Algorithm 1. At any point during the generation run, a core in the tree generator can be in either of two states: (*i*) *active* meaning that it has received a message and is able to forward message to other nodes, or (*ii*) *inactive* otherwise. Inactive nodes are waiting to receive a message from their parent. The set of active cores is denoted as A_{cores}. NUMA nodes are active if at least one of its cores is active (A_{nodes}).

We observe that the tree obtained from this algorithm is a multilevel tree for most machines (↗). Message delivery is first executed across NUMA nodes and then further distributed within each node. The tree generator creates a multilevel hierarchy in either of these steps only if the send operation is relatively expensive compared to receive operations. Otherwise, it will sequentially send messages. For example, with a NUMA node, due to relatively low send costs, sequentially sending messages is often faster than a multilevel sub-tree, especially for systems with only a few cores per node.

In our evaluation (§5.1), we show that a tree generated with our tree generator performs comparably with or better than the best static tree topology on a wide range of machines. While the algorithm itself might have to be adapted in the future to cope with changes in hardware development, the approach of using micro bench-

Algorithm 1 Smelt's adaptive tree

```
$C_{all}$                                                  ▷ Set of cores
$A_{nodes} \leftarrow$ NODE_OF($c_{root}$)                 ▷ Active nodes
$A_{cores} \leftarrow c_{root}$                            ▷ Active cores
function PICK_MOST_EXPENSIVE($C$, $i$)
    return $\arg\max_{x \in C} (t_{send}(i,x) + t_{receive}(i,x))$
end function
function PICK_CHEAPEST($C$, $i$)
    return $\arg\min_{x \in C} (t_{send}(i,x))$
end function
function NODE_IDLE($i$)  ▷ Executed for active idle node $i$
    $C_{inactive} \leftarrow C_{all} \cap (A_{cores} \cup$ CORES_OF($A_{nodes}$) )
    $c_{next} \leftarrow$ PICK_MOST_EXPENSIVE($C_{inactive}$, $i$)
    if SAME_NODE($i$, $c_{next}$) then                     ▷ Local
        SEND($c_{next}$)
        $A_{cores} \leftarrow A_{cores} \cup c_{next}$     ▷ Mark core active
    else
        $C_{eligible} \leftarrow$ NODE_OF($c_{next}$)      ▷ All cores on node
        $c_{next} \leftarrow$ PICK_CHEAPEST($C_{eligible}$, $i$)
        SEND($c_{next}$)                                   ▷ Send remotely
        $A_{nodes} \leftarrow A_{nodes} \cup$ NODE_OF($c_{next}$)
        $A_{cores} \leftarrow A_{cores} \cup r$
    end if
end function
function ADAPTIVE_TREE                             ▷ Run for core $c_{self}$
    while $C_{all} \cap A_{cores} \neq \emptyset$ do
        if $c_{self} \in A_{cores}$ then
            NODE_IDLE($c_{self}$)
        else
            WAIT_MESSAGE
        end if
    end while
end function
```

marks to capture fine-grained hardware details for building machine-aware broadcast trees should still be applicable. Programmers then automatically benefit from an improved version of the algorithm constructing the tree, even in the presence of completely new and fundamentally different hardware without having to change application program code. Consequently, we believe our generator to be useful for future increasingly heterogeneous multicores.

Note that our algorithm is designed for broadcasts trees, but we show in §5 that it also works well for reductions. However, our design and implementation are flexible enough to use different trees for reductions and broadcasts if necessary for future hardware.

3.3.2 Incremental optimization

Smelt's base algorithm produces an hierarchical tree, where only one expensive cross-NUMA link is taken per node. This gives a good initial tree, but leaves room for further improvements, which we describe here:

Reorder sends: most expensive subtree Smelt's basic algorithm as described before sends on expensive links first. This is a good initial strategy, but can be further improved after constructing the entire tree-topology. In order to minimize the latency of the broadcast, the time until a message reaches the last core has to be reduced. Sending on links that have the most expensive sub-tree intuitively achieves that.

Shuffling: adding further cross-NUMA links As soon as a NUMA node is active, i.e. has received a message or has a message being sent to it already, it will not be consider for further cross-NUMA transfers. On larger machines, this can lead to an imbalance, where some threads already terminate the broadcast and become idle when they could still further participate in forwarding the message to minimize global latency of the broadcast tree.

Figure 7 shows this as a simple example for only two cores 0 and 14. Core 14 finishes early and does not consider sending any more messages, since each other NUMA node is already active and all its local nodes finished as well. Core 0 terminates considerably later. The time between core 0 and core 14 finishing is t_{slack} as indicated in the figure.

If an additional cross-NUMA link between core 14's and core 10's NUMA node would terminate faster despite adding another expensive cross-NUMA link, it is beneficial from a purely-latency perspective to allow core 14 to execute this additional NUMA link replacing the link that initially connected node 0 before this optimization.

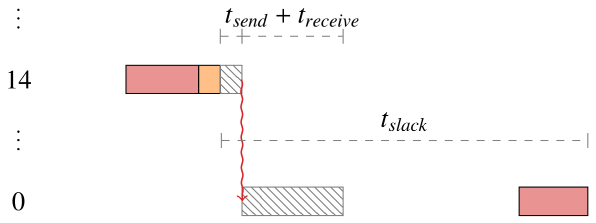


Figure 7: Optimization: add further cross-NUMA links

Smelt executes the following algorithm to decide based on the model if an additional cross-NUMA link $v_s \rightarrow v_e$ would reduce the latency of the broadcast. In each iterative step, we select nodes v_s and v_e as the node that first becomes idle and the node that terminates last respectively. If $t_{send} + t_{receive} < t_{slack}$, Smelt adds an additional cross-NUMA link. Then we iteratively optimize until adding edge $v_s \rightarrow v_e$ does not further reduce the latency of the tree. If this is the case, the resulting tree from replacing previous edge $v_x \rightarrow v_e$ with $v_s \rightarrow v_e$ is always better according to the model. If slower, the algorithm would not have chosen to optimize it and terminated.

The result can be further improved by sorting the edges. Hence, after each "shuffle"-operation, we reorder the scheduling of sends on each outgoing connection of a core by the cost of the receiving core's sub-tree as described in the previous section.

3.4 Finding the optimal solution

Despite being a type of minimum spanning tree, traditional graph algorithms cannot be used to solve the MST problem in our context as they do not consider the edge priorities. In fact, finding a broadcast tree in the telephone model for an arbitrary graph is known to be NP hard [37].

A brute-force approach to the problem is not feasible as the search space grows rapidly. To obtain the best tree given a set of nodes n, we need to construct first all the possible trees with n nodes. This is the Catalan Number [11] of order $n-1$ (C_{n-1}). Moreover, there are $n!$ possible schedules per tree. Hence, the number of possible configurations is shown in Equation 1.

$$Ntrees = n!C_{n-1} = \frac{(2n-2)!}{(n-1)!} \tag{1}$$

Assuming 1ms for evaluating the model of a single tree, this would take over 6 months for 10 cores. We ran a brute-force search for up to 8 cores (5 hours) for a subset of our machines, and compared the adaptive tree against the optimal solution under the model. The estimated runtime by our tree generator predicts that Smelt has a relative error of 9% versus the optimal solution. In real hardware the error is larger at around 13%.

Note that we designed our algorithm for large multicore machines, and evaluating its optimality for configurations of only 8 cores does not show the full potential of our methodology. Unfortunately, due to the high cost of calculating the optimal tree with a brute-force approach, we were not able to extend this validation to bigger machines.

4 Implementation

The Smelt runtime (SmeltRT) is a C++ library that allows the programmer to easily implement machine optimized higher-level protocols by abstracting the required channel setup and message-passing functionality. SmeltRT is structured in two layers: (*i*) a transport layer providing send/receive functionality and (*ii*) a collective layer supporting group communication. For each layer, we explain its core concepts, interfaces and abstractions.

4.1 Transport layer

SmeltRT uses message-passing as a communication mechanism between threads, which SmeltRT pins to cores. The transport layer provides point-to-point message-passing functionality between threads including `send()`, `receive()` and OS independent control procedures. Those message-passing channels are abstracted using a bi-directional *queuepair*, allowing different transport backends to be used transparently.

Properties All queuepair backends must implement the following properties: (*i*) reliability: a message sent over a queuepair is never lost or corrupted and will be received eventually. (*ii*) ordering: two messages sent over the same queuepair will be received in the same order. (*iii*) a queuepair can hold a pre-defined number of messages. Since today's multicores are reliable, only flow control has to be implemented to guarantee above properties. It is needed to notify the sender which slots can be reused for further messages.

Interface The basic send/receive interface can be seen in the following listing. Messages are abstracted using Smelt-messages which encapsulate payload and length to be sent over the queuepair. The send and receive operations may block if the queuepair is full or empty respectively. The state of a queuepair can be queried to avoid unecessary blocking.

```
errval_t
smlt_queuepair_send(struct smlt_qp *qp,
                    struct smlt_msg *msg);
errval_t
smlt_queuepair_recv(struct smlt_qp *qp,
                    struct smlt_msg *msg);
```

Message-passing backends The transport layer is modular and supports multiple backends. Each queuepair backend must adhere to the properties of a Smelt queuepair as stated above. Smelt's message-passing backend is an adapted version of the UMP channel used in Barrelfish [2], originally inspired by URPC [5]. A Smelt UMP queuepair consists of two circular, unidirectional buffers of cache line-sized message slots residing in shared-memory. Each message contains a header word which includes the sequence number for flow-control and an epoch bit to identify new messages. Each cache line has one producer and one consumer to minimize the impact of the cache-coherency protocol. The cache lines holding messages are modified only by the sender. The receiver periodically updates a separate cache line with the sequence number of the last received message. This line is checked by the sender to determine whether a slot can be reused.

Further, we implemented other shared-memory backends (such as FastForward [16]) and plan to support inter-machine message-passing backends over IP or RDMA protocols in the future.

4.2 Collective layer

The collective layer builds upon the transport layer and provides machine-aware, optimized group communication primitives such as broadcasts and reductions. A reduction is a gather operation which blocks until the node has received the value from all its children and the aggregate is forwarded to the parent. Such collective operations involve one or more threads in the system.

4.2.1 Concepts

Smelt's collective operations rely on Smelt's core concepts of *topologies* and *contexts*.

Topologies A topology describes the communication structure for the collective operation. It defines which participants are part of this communication group (i.e. multicast). Each topology has a distinct node, the root, where all broadcasts are initiated and all reductions end. The topology can be generated deterministically at runtime, loaded from a configuration file, or returned by a service. The latter two can describe a hardware aware topology either precalculated or supplied on demand respectively.

Contexts Smelt takes the topology description as a blueprint for creating the required transport links which are encapsulated in a context. A topology can be used to create multiple contexts. Collective operations require a valid context to identify the parent and child nodes for sending and receiving messages.

4.2.2 Collective operations

Next we show the interface for collective operations. The context identifies the location in the tree of the calling thread and defines the operations being executed. The reduction takes an `operation` argument – a function pointer – to implement the reduction operation. At the root, the `result` parameter contains the gathered value.

```
errval_t smlt_broadcast(struct smlt_context *ctx,
                        struct smlt_msg *msg);
errval_t smlt_reduce(struct smlt_context *ctx,
                     struct smlt_msg *input,
                     struct smlt_msg *result,
                     smlt_reduce_fn_t op);
```

Broadcasts Smelt's broadcast primitives guarantee reliability and ensure that all nodes in a given context receive messages in the same order (atomic broadcast). We assume that multicores today are fail-stop as a whole and hence either run reliable or the entire machine fails. Smelt's broadcasts start at a defined, per context root and therefore all messages are sent through the root, that acts as a sequentializer. It is possible to have multiple contexts with different roots. In that case, however, each core has to poll several memory locations for messages associated with multiple endpoints from different trees. This increases the receive overhead on each core, but also the latency, as multiple channels have to be polled. The sequentializer, together with the FIFO property of the edge links and reliable transmission, implements the atomic broadcast property.

Reductions Reductions do in-network processing on each node from payload received from all children, and pass the new value to the parent node. We use the same tree as in the broadcasts and the final value can be obtained at the root.

Barriers With the basic collective operations `reduce` and `broadcast`, we implement a `barrier` as shown in the code below. Note that we use the optimized zero-payload variants of `reduce` and `broadcast`. Despite its simplicity, our `barrier` outperforms or is comparable to state-of-the-art implementations, as we show in §5.4 and §5.5. This demonstrates that a highly-tuned generic machine-aware broadcast as implemented by Smelt can be used for higher-level protocols that benefit automatically from Smelt's optimizations.

```
void smlt_barrier(struct smlt_context *ctx) {
    smlt_reduce(ctx);
    smlt_broadcast(ctx);
}
```

5 Evaluation

We ran our experiments on eleven machines of two vendors with different microarchitectures and topologies (c.f. Table 1). Throughout this section, we indicate the number of threads as triple #sockets×#cores×#threads.

Due to space constraints, in most sections we focus on the largest machines for each vendor. In addition to the results shown here, we provide a website showing detailed results for all experiments on each machine: http://machinedb.systems.ethz.ch.

5.1 Message passing tree topologies

We evaluate the performance of Smelt's adaptive tree against the tree topologies shown in §2.4. We run atomic broadcasts, reductions, barriers and two-phase commit [23] as workloads on all of our machines.

We measure how long it takes until every thread has completed the execution of the collective operation. We avoid relying on synchronized clocks by introducing an additional message to signal completion: for atomic broadcast and two-phase commit a distinct leaf sends a message to the root. We measure the time until the root (i.e. the initiator of the operation) receives this message. We repeat this for all leaves and select the maximum time

machine	A MC 4x12x1	I IB 2x10x2	I NL 4x8x2	A BC 8x4x1	I KNC 1x61x4	I SB 2x8x2
CPU	AMD Opteron 6174	Intel Xeon E5-2670 v2	Intel Xeon L7555	AMD Opteron 8350	Xeon Phi	Intel Xeon E5-2660 0
micro arch	Magny Cours	IvyBridge	Nehalem	Barcelona	k1om	SandyBridge
#cores	4x12x1 @ 2.20GHz	2x10x2 @ 2.50GHz	4x8x2 @ 1.87GHz	8x4x1 @ 2.00GHz	1x60x4 @ 1.00GHz	2x8x2 @ 2.20GHz
caches	64K/ 512K/ 4M	32K/ 256K/ 25M	32K/ 256K/ 24M	64K/ 512K/ 2M	32K/ 512K/	32K/ 256K/ 20M
memory	126G	252G	110G	15G	15G	64G
machine	A SH 4x4x1	A IL 4x4x2	I SB 4x8x2	I BF 2x4x2	A IS 4x6x1	
CPU	AMD Opteron 8380	AMD Opteron 6378	Intel Xeon E5-4640 0	Intel Xeon L5520	AMD Opteron 8431	
micro arch	Shanghai	Interlagos	SandyBridge	Bloomfield	Istanbul	
#cores	4x4x1 @ 2.50GHz	4x4x2 @ 2.40GHz	4x8x2 @ 2.40GHz	2x4x2 @ 2.27GHz	4x6x1 @ 2.50GHz	
caches	64K/ 512K/ 6M	16K/2048K/ 6M	32K/ 256K/ 20M	32K/ 256K/ 8M	64K/ 512K/ 4M	
memory	16G	512G	505G	24G	15G	

Table 1: Machines used for evaluation. Caches given as L1 data / L2 / L3. (L2 per core, L3 socket). Number of cores represented as sockets / cores per socket / threads per core.

among them. In reductions, this is reversed as the root sends the message to the leaf. For barriers, we measure the cost on each core and take the maximum. Barriers are implemented as shown in §4.2.2. We repeat the experiment 10'000 times and collect 1'000 data points.

We first show a detailed breakdown for selected machines to motivate that no single tree topology performs well across all machines followed by giving an overview of the performance across all evaluated machines. There, we show that Smelt not only matches the best tree topology on each machine, but further improves performance on top of that.

Breakdown for selected machines Figure 8 shows the detailed comparison of an 4-socket AMD machine (A IL 4x4x2), a 4-socket Intel machine (I SB 4x8x2) and an Intel's Xeon Phi coprocessor (I KNC 1x60x4). The latter uses a ring topology to connect cores instead of a hierarchical interconnect. In our evaluation, we use only one thread per physical core.

Our results support the claim that there is no clear best static topology for all machines and that the best choice depends on the architecture and workload. As expected, sequential sending results in a significant slowdown compared to all other topologies. The other hardware oblivious trees, binary and Fibonacci, perform comparably but suffer from using too many inter-socket messages on hierarchical machines. The cluster topology performs well in many cases, but since it relies on the machine's hierarchy, it is slow on machines not having a hierarchical memory topology (I KNC 1x60x4). On A IL 4x4x2, the cluster is comparable but slower than Smelt. This is because of the rather static ordering for sending the messages as well as a node's outdegree in the tree that is not optimized. Despite using machine characteristics, the MST topology does not consider the protocol's communication patterns nor tries to maximize parallelism.

In contrast, Smelt's adaptive tree (AT) achieves good performance across all configurations due to the fact that it uses hardware information enriched with real measurements to capture fine-grained performance characteristics of the machine and adapt the message scheduling accordingly. Our results show that *generating* a tree based on our machine model as described in § 3.3 achieves good results without the programmer's awareness of hardware characteristics and manual tuning.

We demonstrate that the tree topology matters and that there is no static topology that performs best on all machines even when considering the NUMA hierarchy. We show that Smelt is able to adapt to a wide set of micro-architectures and machine configurations without manual tuning.

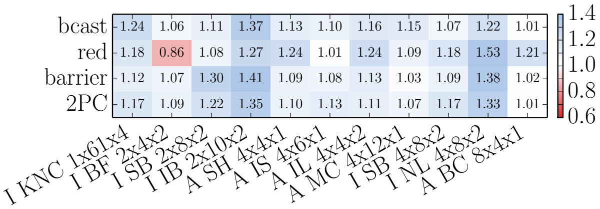


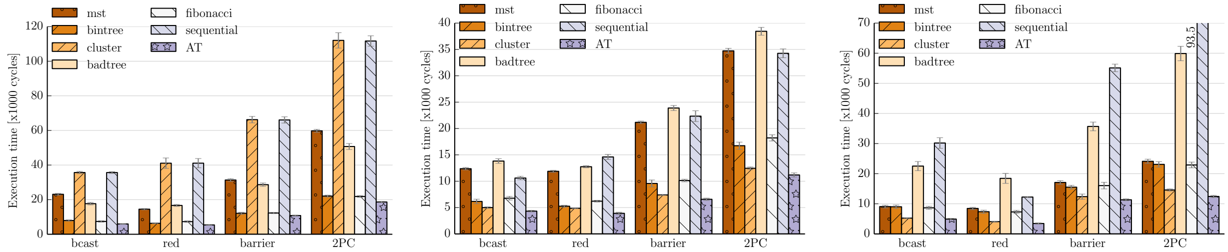
Figure 9: Comparison of Smelt to the best static tree topology on each machine. Ordered by the the number of sockets as indicated by the label.

Comparison with the best other topology. Figure 9 is a heat-map showing the speedup of Smelt compared to the best static tree on all machines. For example, if the "cluster" topology is the best tree topology besides Smelt on a machine, we use that as a baseline. All tree topologies use Smelt's transport layer (§4.1).

The Fibonacci tree achieves the best performance on three of these machines, the binary tree on one of them and the cluster tree on the remaining seven.

Smelt not only matches the best tree topology for all but one configuration, but also manages to achieve an average speedup of 1.16 over all machines, peaking at a speedup of up to 1.24x compared to the best static tree on AMD (A SH 4x4x1) and up to 1.53x on Intel (I NL 4x8x2).

To conclude, this experiment shows that even when the best static topology for a concrete machine is known, Smelt still manages to further improve the performance since, in addition to considering the hierarchy of the machine to avoid expensive cross-NUMA links, it optimally configures the outdegree in each node of the tree. It does



(a) I KNC 1x60x4 (b) A IL 4x4x2 (c) I SB 4x8x2

Figure 8: Details of micro benchmark results for all the evaluated tree topologies.

so based on the machine characteristics gathered from the pairwise send and receive measurements (§3.2).

5.2 Multicast topologies

Certain workloads require collective communications within a subset of the threads (i.e. multicast). We evaluate this scenario by running the benchmarks from § 5.1 with an increasing thread count starting from 2 up to the maximum number of threads of the machine. For this, we assign threads to NUMA nodes in round-robing, e.g. when showing four cores on a machine with four NUMA nodes, we would place one thread on each NUMA node.

Figure 10 shows the multicast scaling behavior of Smelt compared to the static trees on A IL 4x4x2 and A IS 4x6x1. For a low number of cores — for example one core per NUMA node to implement consistent updates to data replicated on a per-NUMA basis as in §5.7 — it is often best to send messages sequentially: we observe this behavior in both Figure 10b and 10a. At that point, all communication links are remote and the hierarchical cluster approach does not work well. When more cores are involved and the effects of local vs. remote communication become more obvious, the cluster topology performs best again. In summary, the choice of topology does not only matter on the machine, but also the multicast group intended to use.

Further, the performance benefit when using Smelt is often higher in intermediate configurations: for example in Figure 10b, the maximum speedup over the best static topology is 1.25 with 12 cores as to compared to 1.02 for a reduction involving all 24 cores. The maximum negative speedup of Smelt is 0.97 on A IS 4x6x1 for 22 cores.

5.3 Comparison with MPI and OpenMP

We compare Smelt with two established communication standards: MPI and OpenMP. MPI (Message Passing Interface) [30] is a widely used standard for message-based communication in the HPC community. MPI supports a wide range of collective operations, including broadcasts, reductions and barriers. Furthermore, the MPI libraries provide specific channels and optimizations for shared-memory systems.

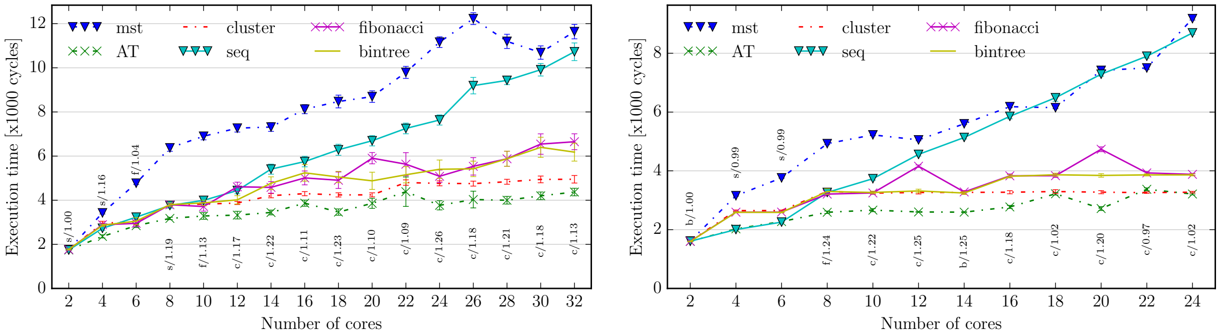
OpenMP 4.0 [39] is a standard for shared-memory parallel processing and is supported by major compilers, operating systems and programming languages. The OpenMP runtime library manages the execution of parallel constructs. Threads are implicitly synchronized after a parallel block or explicitly by the barrier directive.

We compare the collectives of MPI (Open MPI v1.10.2) and OpenMP (GOMP from GCC 4.9.2) with Smelt. For MPI we compare broadcasts, reductions, and barriers. OpenMP only provides reductions and barriers.

For each of the runtimes, we execute the experiment 3000 times and take the last 1000 measurements. At the beginning of each round, we synchronize all the threads with two dissemination barriers so that all threads enter the collective operation at the same time. This is different from the benchmark in § 5.1 since here the cost of the extra message depends on the used library. The broadcast is executed with a one byte payload and reduction has a single integer payload.

Figure 11 shows the results of the largest machines. Smelt outperforms MPI for all the tested collective operations. In broadcasts and reductions Smelt outperforms MPI with speedups between 1.6x and 2.6x. For barriers the lead is smaller (between 1.19x and 1.41x). OpenMP performs worse than MPI, showing that message-passing approaches are better-suited for large multicore machines than shared-memory programming models. Moreover, OpenMP is clearly outperformed by Smelt in reductions on the three machines. OpenMP barriers perform well on A IL 4x4x2 but still Smelt is between 1.5x and 3.8x faster.

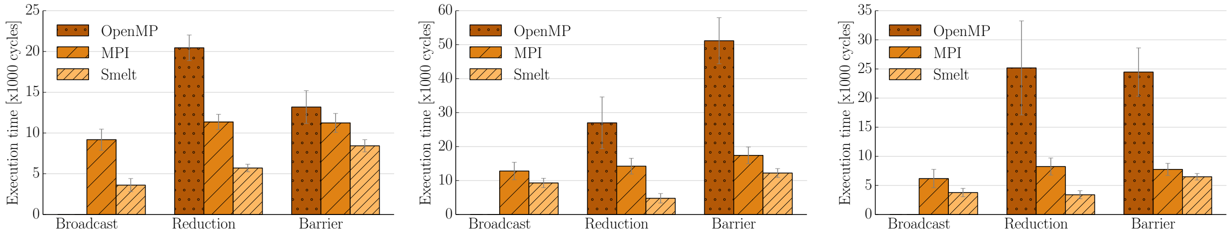
Since OpenMP uses explicit and implicit barriers after each parallel construct, we extend the evaluation to demonstrate how Smelt can be used to improve the GOMP library [15]. GOMP's standard barriers are based on atomic instructions and the futex syscall on Linux [12, 14, 26]. We replaced GOMP's barrier with Smelt and compared it against the vanilla version. As workload we took *syncbench* and *arraybench* from the EPCC OpenMP micro-benchmarks suite [27] using standard settings and 5000 outer repetitions. We ran the benchmark using all



(a) Broadcast A IL 4x4x2

(b) Reduction A IS 4x6x1

Figure 10: Multicast, cores allocated round-robin to NUMA nodes. Labels: speedup compared to the best static topology and first letter of that topology's name.



(a) A IL 4x4x2

(b) I NL 4x8x2

(c) I SB 4x8x2

Figure 11: Comparison with MPI and OpenMP.

available threads.

The results of the benchmark are shown in Table 2. Overall, Smelt performs significantly better or comparable to the original GOMP barriers. In the *BARRIER* micro-benchmark, we achieve up to 2.5x and 1.5x speedup respectively. These results show that replacing the standard barriers in GOMP with Smelt reduces the overhead for synchronization significantly.

machine	C	parlibMCS		dissemination		Smelt	
I SB 4x8x2	32	16,718	(495)	6,699	(63)	4,725	(6)
	64	38,494	(755)	19,762	(22)	6,348	(10)
I NL 4x8x2	32	13,836	(348)	5,777	(239)	4,035	(22)
	64	15,604	(1,366)	6,333	(185)	5,755	(26)
A IL 4x4x2	16	4,288	(7)	4,596	(92)	4,792	(9)
	32	5,989	(23)	5,220	(12)	7,016	(35)

Table 3: Barrier micro-benchmark for 32 and 64 threads, median of 100 calls [cycles], standard error in brackets.

5.4 Barriers micro-benchmarks

Barriers are important building blocks for thread synchronization in parallel programs. We compare our barrier implementation (§ 4.2.2) with the state-of-the-art MCS dissemination barrier [1] (parlibMCS) and a 1-way dissemination barrier that uses atomic flags [33] (dissemination). We show that our simple barrier implementation, based on broadcast and reduction, can compete with highly-tuned state-of-the-art shared-memory implementations.

In this evaluation, we synchronize threads using the different barriers in a tight for-loop of 10,000 iterations. This is yet another barrier benchmark and cannot be directly compared with the previous sections.

The results in Table 3 show significant differences between machines and whether or not hyperthreads are used. Whereas on A IL 4x4x2 Smelt performs worse and there is no clear winner, Smelt is up to 3x faster on Intel machines relative to the dissemination barrier and up to 6x faster compared to parlibMCS.

With this evaluation we have shown how a competitive barrier can be implemented easily using Smelt's hardware aware collective operations.

5.5 Streamcluster

PARSEC Streamcluster [6] solves the online clustering problem. We chose this benchmark because it is synchronization-intensive. We evaluate the performance of Smelt's barriers compared to PARSEC's default barriers, pthread barriers, and parLib dissemination barrier [1].

For all configurations, we used the native data set and run the benchmark with and without Shoal, a framework for optimizing memory placement and access based on access patterns [21]. Otherwise, Streamcluster's performance is limited by memory bandwidth and the effects of optimizing synchronization are less visible.

Our results in Table 4 confirm that optimizing both

		PARALLEL		FOR		BARRIER		SINGLE		COPYPRIV		COPY		PRIV	
A IL 4x4x2	GOMP	21.81	(112.16)	6.93	(0.15)	6.92	(0.15)	11.31	(1.41)	287.84	(122.97)	101.40	(1.22)	52.20	(301.56)
	Smelt	13.93	(0.15)	4.15	(0.07)	4.13	(0.07)	7.83	(0.08)	120.10	(24.52)	104.66	(0.94)	13.84	(0.28)
I SB 4x8x2	GOMP	55.41	(333.78)	9.50	(3.22)	9.51	(3.23)	14.85	(2.26)	319.44	(7.29)	135.16	(1.13)	46.69	(140.40)
	Smelt	38.31	(0.17)	5.64	(0.02)	5.63	(0.02)	10.27	(0.98)	141.95	(1.81)	128.48	(1.04)	37.20	(0.44)

Table 2: EPCC OpenMP benchmark. Average in microseconds, standard error in brackets.

	pthread	parsec	Smelt	parlib MCS
		A IL 4x4x2		
no Shoal	215.619	207.929	181.405	202.801
Shoal	51.816	52.075	41.116	41.061
		I SB 4x8x2		
no Shoal	236.476	235.394	124.492	125.184
Shoal	66.421	68.079	28.283	28.779

Table 4: Execution time of Streamcluster [seconds]

memory accesses and synchronization primitives matter for achieving good performance of parallel programs. We also show that our simple barrier implementation (§4.2.2) based on a generic broadcast tree performs better than shared-memory barriers and is competitive with a state-of-the-art dissemination barrier.

5.6 Agreement

We implemented the 1Paxos [10] agreement protocol using Smelt. 1Paxos is a Paxos-variant optimized for multicore environments. Normal operation is shown in Figure 12a: a client sends a request to the leader which forwards the request to the acceptor. Then there is a single broadcast from the acceptor to the replicas that we optimize using Smelt. Upon receiving the broadcast, the leader responds to the client.

We re-implemented 1Paxos with and without Smelt, because the original 1Paxos paper uses its own threading and message passing library. Furthermore, they create one thread for each incoming connection which has a large negative impact on performance [19]. The processing time then dominates over communication cost making it unsuitable for the evaluation of our work.

We vary the number of replicas from 8 to 28 and use 4 cores as load generators, which was sufficient to issue enough requests to keep the system busy. The measurements are averaged over three runs of 20 seconds each. Figures 12b and 12c present the performance of the agreement protocol and an atomic broadcast with the same threads.

The results show that the agreement protocol on multicore machines can benefit from an optimized broadcast primitive: using Smelt improves the throughput and response time up to 3x compared to sequential sending on 28 replicas. As we increase the number of replicas, the sequential broadcast quickly becomes the bottleneck. Our results show that 1Paxos is highly tuned towards multicores as its scaling behavior and performance are similar to a plain broadcast.

By using Smelt, we can improve the performance of agreement protocols on multicore machines, improving also the scalability to larger number of replicas.

5.7 Key-value store

We implemented a replicated key-value store (KVS) using 1Paxos from § 5.6 to ensure consistency of updates while reads are served directly by the replica. In our implementation, the nearest replica responds to the client request. Our implementation supports a get/set interface. We focus on small keys (8 byte) and values (16 byte) to avoid fragmentation. If fragmentation was implemented, larger messages would simply be split up in multiple smaller messages. This would cause a behavior similar to adding more clients to the system.

We placed a KVS instance on each NUMA node of the machine, 8 on A IL 4x4x2. An increasing number of clients connect to their local KVS instance and issue requests. We executed the benchmark for 20 seconds and 3 runs with a get/set ration of 80/20.

The *set* performance results are shown in Figure 13. We omit the *get* results as they are served locally. Our results demonstrate that Smelt is able to improve performance even for a small number of replicas. Scalability and stability under high load are even better, resulting in up to 3x improvement for throughput and response time.

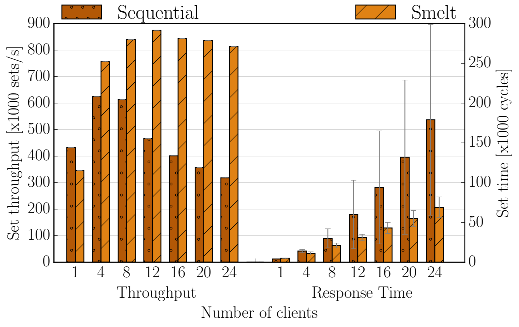
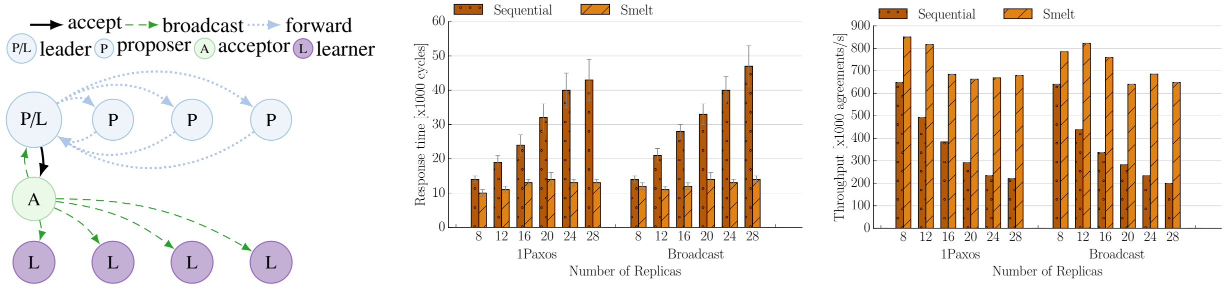


Figure 13: *Set* performance on A IL 4x4x2

6 Conclusion

Smelt is a new approach for tuning broadcast algorithms to multicore machines. It automatically builds efficient broadcast topologies and message schedules tuned



(a) 1Paxos failure-free case.

(b) Response time on A IL 4x4x2.

(c) Throughput on A IL 4x4x2.

Figure 12: Performance results of 1Paxos agreement.

to specific hardware environments based on a machine model which encodes both static hardware information and costs for sending and receiving messages, generated from micro-benchmarks that capture low-level machine characteristics. Smelt provides an easy-to-use API which can be used to build high-level applications on top of it. We have shown that the trees generated by Smelt match or outperform the best static topology on each of a variety of machines.

Moreover, we also show how other collective operations can be constructed using these trees as a building block and achieve good performance without requiring any extra tuning effort. Barriers implemented trivially on top of Smelt outperform state-of-the-art techniques, including shared-memory algorithms which do not use message passing. We also achieve good scaling with the number of parallel requests in an in-memory replicated key-value store built on top of Smelt's adaptive trees.

Consequently, we claim that automatically generated broadcast topologies can deliver high performance in parallel applications without requiring programmers to have detailed understanding of a machine's topology or memory hierarchy. Smelt is open source and available for download at https://github.com/libsmelt.

6.1 Future work

We believe that using Smelt will have an even larger benefit on future machines that experience partial transient hardware failures [7, 13], are more heterogeneous [7] and increasingly rack-scale.

In the case of failures, the tree has to be reconfigured dynamically. To make this fast, the changes in the tree topology should ideally be kept local. We believe that our optimizations to the basic tree topology as described in §3.3.2 are a good starting point to explore strategies to efficiently update trees topologies locally. Furthermore, the asymmetry of group communication as a result of spacial scheduling and failures will make static tree topologies less ideal (e.g. the cluster works best for a "regular" symmetric machine and less for irregular topologies).

Smelt will likely be able to handle more heterogeneous hardware, as microbenchmarks reflect these performance characteristics and the adaptive tree automatically handles such cases. We show this by running Smelt on the Xeon Phi, which exposes a completely different architecture, without having to modify Smelt. Furthermore, simulation based on pairwise send and receive cost should work equally well on multicore machines providing message-passing hardware, as long as t_{send} and $t_{receive}$ adequately represent the system's cost for communication. However, it is likely that more microbenchmarks have to be added in the future to precisely capture hardware characteristics as more intricate accelerator hardware is added (e.g. deeper write buffers). However, the general approach of modeling the machine combined with simulation will likely still work in such a setting.

Further, we plan to extend our machine model to include more hardware details that may have an additional impact in larger machines, like contention on inter-socket links and write-buffer effects. Also, to date Smelt does not address partial failures, and assumes the entire machine to be fail-stop. Extending the system to support multiple failure domains and fully networked communication at rack scale is a natural line of extension of our work.

Acknowledgments

We thank our mentor Peter Chen and the anonymous reviewers for their detailed and useful reviews and the Computer Architecture Group from the University of A Coruña for the access to their cluster Pluton (Project TIN2013-42148-P).

References

[1] Amplab, UC Berkeley. PARLIB, MCS Locks. Online. http://klueska.github.io/parlib/mcs.html. Accessed 05/10/2016.

[2] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and

A. Singhania. The Multikernel: A new OS Architecture for Scalable Multicore Systems. In *Proceedings of the 22nd ACM Symposium on Operating System Principles*, SOSP '09, pages 29–44, Big Sky, Montana, USA, 2009.

[3] A. Baumann, S. Peter, A. Schüpbach, A. Singhania, T. Roscoe, P. Barham, and R. Isaacs. Your computer is already a distributed system. Why isn't your OS? In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems*, May 2009.

[4] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. TILE64 - Processor: A 64-Core SoC with Mesh Interconnect. In *Digest of Technical Papers of the IEEE International Solid-State Circuits Conference*, ISSCC 2008, pages 88–598, Feb 2008.

[5] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. User-level Interprocess Communication for Shared Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(2):175–198, May 1991.

[6] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 72–81, Toronto, Ontario, Canada, 2008. ACM.

[7] S. Borkar and A. A. Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67–77, May 2011.

[8] J. Bruck, R. Cypher, and C.-T. Ho. Multiple Message Broadcasting with Generalized Fibonacci Trees. In *Proceedings of the 4th IEEE Symposium on Parallel and Distributed Processing*, pages 424–431, Arlington, Texas, USA, Dec 1992.

[9] T. David, R. Guerraoui, and V. Trigonakis. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP '13, pages 33–48, Farminton, Pennsylvania,USA, 2013. ACM.

[10] T. David, R. Guerraoui, and M. Yabandeh. Consensus Inside. In *Proceedings of the 15th International Middleware Conference*, Middleware '14, pages 145–156, Bordeaux, France, 2014. ACM.

[11] N. Dershowitz and S. Zaks. Enumerations of Ordered Trees. *Discrete Mathematics*, 31(1):9–28, 1980.

[12] U. Drepper. Futexes Are Tricky. Technical report, Red Hat, Inc., Dec 2011.

[13] P. Faraboschi, K. Keeton, T. Marsland, and D. Milojicic. Beyond Processor-centric Operating Systems. In *Proceedings of the 15th USENIX Conference on Hot Topics in perating Systems*, HOTOS' 15, pages 17–17, Kartause Ittingen, Switzerland, 2015. USENIX Association.

[14] H. Franke and R. Russell. Fuss, Futexes and Furwocks: Fast Userlevel Lockingin Linux. In *Proceedings of the 2002 Ottawa Linux Symposium*, OLS '02, pages 18:1–18:11, Denver, Colorado, USA, 2002.

[15] Free Software Foundation, Inc. Welcome to the home of GOMP. Online. https://gcc.gnu.org/projects/gomp/. Accessed 05/10/2016.

[16] J. Giacomoni, T. Moseley, and M. Vachharajani. FastForward for Efficient Pipeline Parallelism: A Cache-optimized Concurrent Lock-free Queue. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 43–52, Salt Lake City, Utah, USA, 2008. ACM.

[17] R. L. Graham and G. Shipman. MPI Support for Multi-core Architectures: Optimized Shared Memory Collectives. In *Proceedings of the 15th European PVM/MPI Users' Group Meeting*, EuroPVM/MPI ' 08, pages 130–140, Dublin, Ireland, 2008. Springer Science & Business Media.

[18] D. Hackenberg, D. Molka, and W. E. Nagel. Comparing Cache Architectures and Coherency Protocols on x86-64 Multicore SMP Systems. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 413–422, New York, New York, USA, 2009. ACM.

[19] R. Haecki. *Consensus on a Mulicore Machine*. ETH Zurich, 2015. Master's Thesis, http://dx.doi.org/10.3929/ethz-a-010608378.

[20] Hewlett-Packard, Intel, Microsoft, Phoenix, Toshiba. *Advanced Configuration and Power Interface Specification, Rev. 4.0a*, Apr. 2010. http://www.acpi.info/.

[21] S. Kaestle, R. Achermann, T. Roscoe, and T. Harris. Shoal: Smart Allocation and Replication of Memory for Parallel Programs. In *Proceedings of the 2015 USENIX Annual Technical Conference*, USENIX ATC '15, pages 263–276, Santa Clara, CA, 2015.

[22] D. E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, 2nd edition, 1998.

[23] B. W. Lampson and H. E. Sturgis. Crash Recovery in a Distributed Data Storage System, 1979.

[24] B. Lepers, V. Quema, and A. Fedorova. Thread and memory placement on numa systems: Asymmetry matters. In *Proceedings of the 2015 USENIX Annual Technical Conference*, USENIX ATC '15, pages 277–289, Santa Clara, CA, July 2015.

[25] S. Li, T. Hoefler, and M. Snir. NUMA-aware Shared-memory Collective Communication for MPI. In *Proceedings of the 22nd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '13, pages 85–96, New York, New York, USA, 2013. ACM.

[26] Linux Programmer's Manual. futex - fast user-space locking. Online. http://man7.org/linux/man-pages/man2/futex.2.html. Accessed 05/10/2016.

[27] Mark Bull and Fiona Reid. EPCC OpenMP micro-benchmark suite. Online. https://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/epcc-openmp-micro-benchmark-suite. Accessed 05/10/2016.

[28] D. Molka, D. Hackenberg, and R. Schöne. Main Memory and Cache Performance of Intel Sandy Bridge and AMD Bulldozer. In *Proceedings of the Workshop on Memory Systems Performance and Correctness*, MSPC '14, pages 4:1–4:10, Edinburgh, United Kingdom, 2014. ACM.

[29] D. Molka, D. Hackenberg, R. Schöne, and W. E. Nagel. Cache Coherence Protocol and Memory Performance of the Intel Haswell-EP Architecture. In *Proceedings of the 44th International Conference on Parallel Processing*, ICPP ' 15, pages 739–748, Beijing, China, 2015.

[30] MPI Forum. Message Passing Interface Forum. Online. Accessed 05/10/2016.

[31] R. C. Prim. Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36(6):1389–1401, Nov 1957.

[32] S. Ramos and T. Hoefler. Cache Line Aware Optimizations for ccNUMA Systems. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, pages 85–88, Portland, Oregon, USA, 2015. ACM.

[33] S. Ramos and T. Hoefler. Cache Line Aware Algorithm Design for Cache-Coherent Architectures. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 27(10):2824–2837, Oct 2016.

[34] RRZE - Regionales RechenZentrum Erlangen. likwid. Online, 2015. https://github.com/RRZE-HPC/likwid.

[35] T.-I. Salomie, I. E. Subasu, J. Giceva, and G. Alonso. Database Engines on Multicores, Why Parallelize when You Can Distribute? In *Proceedings of the 6th Conference on Computer Systems*, EuroSys '11, pages 17–30, Salzburg, Austria, 2011. ACM.

[36] Silicon Graphics International Corporation. libnuma. Online, 2015. http://oss.sgi.com/projects/libnuma/.

[37] P. J. Slater, E. J. Cockayne, and S. T. Hedetniemi. Information dissemination in trees. *SIAM Journal on Computing*, 10(4):692–701, 1981.

[38] Y.-H. Su, C.-C. Lin, and D. Lee. Broadcasting in Heterogeneous Tree Networks. In *Proceedings of the 16th Annual International Conference on Computing and Combinatorics*, pages 368–377. Springer-Verlag, 2010.

[39] The OpenMP Architecture Review Board. The OpenMP API specification for parallel programming. Online. http://openmp.org/. Accessed 05/10/2016.