NUMA-aware algorithms: the case of data shuffling

Yinan Li[†*] Ippokratis Pandis Rene Mueller Vijayshankar Raman Guy Lohman

[†]University of Wisconsin–Madison
Madison, WI, USA
yinan@cs.wisc.edu

IBM Almaden Research Center
San Jose, CA, USA
{ipandis,muellerr,ravijay,lohmang}@us.ibm.com

ABSTRACT

In recent years, a new breed of non-uniform memory access (NUMA) systems has emerged: multi-socket servers of multi-cores. This paper makes the case that data management systems need to employ designs that take into consideration the characteristics of modern NUMA hardware. To prove our point, we focus on a primitive that is used as the building block of numerous data management operations: data shuffling. We perform a comparison of different data shuffling algorithms and show that a naïve data shuffling algorithm can be up to 3× slower than the highest performing, NUMA-aware one. To achieve the highest performance, we employ a combination of thread binding, NUMA-aware thread allocation, and relaxed global coordination among threads. The importance of such NUMA-aware algorithm designs will only increase, as future server systems are expected to feature ever larger numbers of sockets and increasingly complicated memory subsystems.

1. INTRODUCTION

The last decade has seen the end of clock-frequency scaling, and the emergence of multi-core processors as the primary way to improve processor performance. In response, there have been concerted efforts to develop data management systems that exploit multi-core parallelism.

Another equally important trend is towards multi-socket systems with non-uniform memory access (NUMA) memory architectures. Each socket in a NUMA system has its own 'local' memory (DRAM) and is connected to the other sockets and, hence to their memory, via one or more links. Access latency and bandwidth therefore varies depending on whether a core in a socket is accessing 'local' or 'remote' memory. To get good performance on a NUMA system, one must understand the topology of the interconnect between sockets and memories and accordingly align the algorithm's data structures and communication patterns.

[*] Work done while the author was at IBM.

6th Biennial Conference on Innovative Data Systems Research (CIDR'13) January 6-9, 2013, Asilomar, California, USA.

There is no standard NUMA system: the server market has a smögåsbord of different configurations with varying numbers of sockets, cores per socket, memory DIMMs per socket, and interconnect topologies. For example, focusing only on Intel, server configurations range from one to eight sockets, each socket having four to ten cores and anywhere from a single 2 GB DIMM to sixty-four 32 GB DIMMs and vastly varying inter-processor interconnect ("QPI", in Intel's terminology) topologies. Figure 1 shows the configuration of 4 different Intel servers we have in our lab: a 2-socket, two 4-socket systems, and an 8-socket system. The configurations also change quite a bit from one generation to the next.

Primitives: DBMS servers need to perform well across this hardware spectrum, but it would be impossible to maintain a distinct implementation of the core operations of a DBMS for each machine configuration as they are complex and consist of millions of lines of code. So we argue that DBMS software should isolate performance-critical *primitives*: smaller, isolated modules that can be heavily tuned for a hardware configuration, and whose specification is simple enough that they can be provided as libraries by hardware developers (a good analogy here are the Basic Linear Algebra Subprograms, or BLAS, linear algebra primitives used heavily in scientific computing [17]).

In this paper, we study data shuffling as an exemplary data management primitive. We define *data shuffling* broadly as threads exchanging roughly equal-sized pieces of data among themselves. Specifically, we assume each of the N threads has initially partitioned its data, that reside on its local memory, into N pieces. The goal is to transmit the ith partition from each thread onto the ith thread, for all i.

Shuffling is needed in many data management operations. The most famous example, of course, is Map-Reduce, in which mapper tasks shuffle data to reducer tasks [8]. Shuffle also appears naturally in parallel joins, aggregation, and sorts [3, 9, 11].

There has been much recent research on multi-core-friendly join, sort, etc., but comparatively less attention on the NUMA aspects of these algorithms. We find that NUMA awareness is particularly important for shuffling. In this paper, we analyze the behavior of shuffling on different NUMA machines, show that a straightforward implementation significantly underutilizes some parts of the inter-processor interconnect, and present a hardware-tuned shuffle implementation that is up to 3 times more efficient. We also quantify the potential of thread and state migration instead of data shuffling as

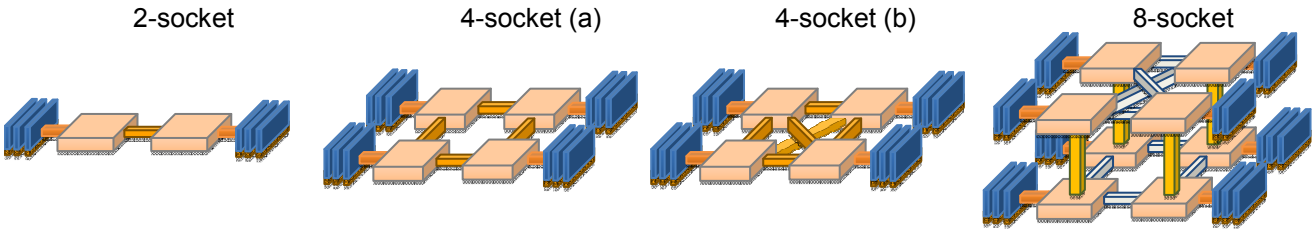


Figure 1: Interconnect topologies for four Intel servers. Blue bars represent memory DIMMs, orange blocks are processors, and pipes are data interconnects. See Section 2 for details.

another approach that achieves equivalent results (multiple threads access data that are provided by other threads).

The contributions of this paper are three-fold:

- We demonstrate the importance of revising database algorithms to take into consideration the non-uniform memory access (NUMA) characteristics of multi-socket systems having multiple cores.
- We focus on shuffling, a primitive that is used in a variety of important data management algorithms. We show that a naïve algorithm can be up to 3 times slower than a NUMA-aware one that exploits thread binding, NUMA-aware memory allocation, and thread coordination. This optimized data shuffling primitive can speed up a state-of-the-art join algorithm by 8%.
- We show the potential of thread migration instead of data shuffling for highly asymmetric hardware configurations. In particular, we show that if the working state of threads is not large, migrating a thread (moving the computation to the data) can be up to 2× faster than data shuffling (moving the data to the computation). We show that thread migration can speed up parallel aggregation algorithms by up to 25%.

The rest of the document is structured as follows. Section 2 discusses the new breed of NUMA systems. The problem of data shuffling in NUMA environments is described in Section 3. Section 4 presents a performance comparison of various data shuffling algorithms. We discuss related work in Section 5 and our conclusions in Section 6.

2. NUMA: HARDWARE & OS SUPPORT

Even the most cost-effective database servers today consist of more than one processor socket (e.g., two sockets for the Facebook Open Compute Platform [21]). Today's commercial mid-range and enterprise class servers may contain up to eight processor sockets. Moreover, each socket accommodates up to ten CPU cores, along with several memory DIMM modules that are attached to the socket through different memory channels. An interconnect network among the sockets allows each core to access non-local memory and maintains the coherency of all caches and memories. Figure 1 shows different configurations for Intel-based servers. Though NUMA architectures have been around for more than two decades, today's multi-socket processors and point-to-point interconnect networks (as opposed to shared buses) make designing high-bandwidth, low-latency solutions more challenging than ever before. Threads running on different cores but within the same socket have (near-) uniform memory access, whereas access to off-socket memory is

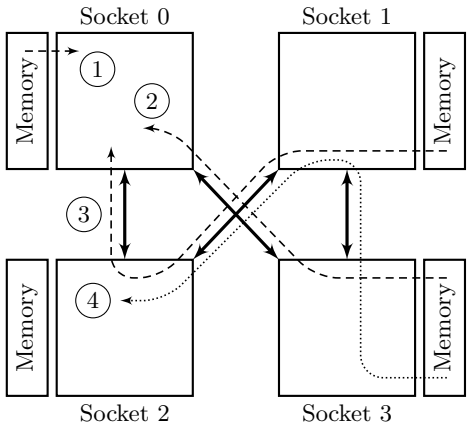


Figure 2: Data access pattern in a 4-socket configuration with 4 QPI links.

slower due to the latencies incurred by the interconnect network. Contention caused by other traffic on the interconnect can further increase latencies of remote accesses.

Example: 4-socket system with 4 QPI links. Consider the 4-Socket (b) configuration shown in Figure 1. This topology is used in a fully populated IBM X Series x3850 X5 system. It contains four 8-core Nehalem-EX X7560 processors that are fully connected through six bi-directional 3.2 GHz *Intel QuickPath Interconnect (QPI) links*.

For illustration purposes, we disconnect two QPI links by removing the two QPI wrapper cards, thus obtaining the 4-link interconnect depicted in Figure 2. Using socket-local and socket-remote memory read accesses, we create four different data flows shown in the figure. Table 1 lists the measured bandwidth and latency for each flow. Our measurements show that we need at least 12 reader threads per socket or QPI link to achieve maximum bandwidth. Flow ① corresponds to a read of the local memory of Socket 0. The maximum aggregate bandwidth on one socket for this flow was measured to be 24.7 GB/s using 12 threads and a sequential read pattern. The latency for data-dependent random access [1] is 340 CPU cycles (≈ 150 ns). [2] Adding a hop

[1] In a data-dependent access pattern, the memory location referenced next will be determined by the content of the memory location that is currently being accessed.

[2] The latency numbers are given for data-dependent random accesses, where the first word of a cache line is used to determine the next read location. The latency increases by 5 cycles if the last word of a cache line is used instead.

over one single QPI link (flow ②) decreases the bandwidth to 10.9 GB/s, which corresponds to the uni-directional bandwidth of a single QPI link, and increases the random-access latency by 80 CPU cycles (≈ 35 ns). A second hop (flow ③) adds another 100 CPU cycles (≈ 45 ns) to the random access latency. For data-independent accesses, these latencies can be hidden through pipelining. Hence, adding a second hop does not further affect the sequential bandwidth.

Cross traffic can lead to contention on the interconnect links, which lowers the sequential bandwidth for a thread and increases the access latency. For example, when running flow ③ concurrently with flow ④, the aggregate bandwidth of the 12 reader threads on flow ③ drops to 5.3 GB/s and the median of the random access latency increases by 10 CPU cycles (≈ 4.4 ns).

In a micro-benchmark in which 12 threads on each socket simultaneously access data on all three remote sockets (4 threads per remote socket, and hence per QPI link), we measure an aggregate bandwidth of 61 GB/s. The aggregate bandwidth would be 99 GB/s if all threads accessed socket-local memory. Hence, the measured interconnect bandwidth corresponds to 60 % of the socket-local bandwidth. The Nehalem-EX architecture measured in this example implements a *Source-based Snooping* protocol [13, page 20] to maintain coherency. These snooping messages (and replies) are exchanged on the interconnect even when all threads access socket-local memory. Therefore, even socket-local memory access can affect the traffic on the interconnect.

This simple performance evaluation shows that just assigning threads to cores and exploiting socket locality in memory accesses is insufficient. If the data accesses are carefully orchestrated, performance can be improved by reducing contention on the interconnect. These NUMA effects have to be considered, particularly for latency-critical applications such as transaction processing, as shown in [24]. Systems such as DORA [22] avoid uncoordinated data accesses common in OLTP systems by binding threads (cores) to data.

Operating systems are aware of NUMA architectures. Linux partitions memory into NUMA zones, one for each socket. For each NUMA zone, the kernel maintains separate management data structures. Unless explicitly bound to a specified socket (NUMA zone) through the `mbind()` system call, the Linux kernel allocates memory on the socket of the core that executes the current thread. More precisely, the allocation to the NUMA zone will only happen after the first write access to the memory, which then causes a page fault. The kernel tries to satisfy the memory allocation from the current zone. If there is not enough memory available in that zone, the kernel allocates memory from the next zone with memory available.

The operating system scheduler tries to minimize thread migration to other cores. Under certain conditions, such as under-utilization of other cores, the scheduler may decide to migrate a thread to a core in another zone. The kernel does not automatically migrate the associated memory pages along with the thread. Instead, *Page Migration* can be performed through the `mbind()` system call, either manually or automatically by a user-space daemon process.

3. THE CASE OF SHUFFLING IN NUMA

Shuffling data among data partitions is a ubiquitous building block of most distributed data management operations.

Table 1: Bandwidth and Latency of a 4-socket Nehalem-EX system with 4 QPI links

	Bandwidth seq. memory access (12 threads)	Latency data-dependent random accesses (1 thread)
local memory access ①	24.7 GB/s	340 cycles/access (≈ 150 ns/access)
remote memory 1 hop ②	10.9 GB/s	420 cycles/access (≈ 185 ns/access)
remote memory 2 hops ③	10.9 GB/s	520 cycles/access (≈ 230 ns/access)
remote memory 2 hops ③ with cross traffic ④	5.3 GB/s	530 cycles/access (≈ 235 ns/access)

For example, in relational systems, partitioning-based join and aggregation algorithms have a step in which each thread in the system partitions data based on a global partitioning function. Then either each producer thread needs to *push* portions of the data to its corresponding consumer thread, or each consumer needs to *pull* the data it needs on demand. Obviously, data shuffling is a fundamental operation in the MapReduce framework, as well.

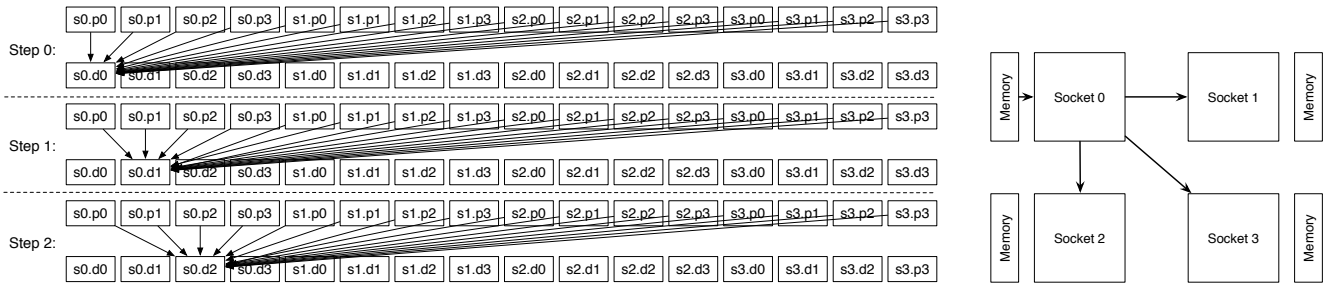
In this paper, we focus on efficient data shuffling algorithms for NUMA machines. In particular, the problem we consider is the efficient transmission of data between threads in a machine composed of multiple multi-core sockets that are statically connected by links having finite but possibly different bandwidths. Let S denote the number of sockets in the machine and s_i denote the i^{th} socket ($0 \leq i < S$). Each thread is bound to a core on a socket. Let $s_i.p_j$ denote the j^{th} thread running on socket s_i. Suppose that we have P threads running on each socket, and thus we have a total of $N = SP$ threads. Thread $s_i.p_j$ produces some data, denoted as $s_i.d_j$, that needs to be consumed by the corresponding thread. For every pair of thread $s_{i1}.p_{j1}$ and data $s_{i2}.d_{j2}$, in a shuffle operation involving N threads ($N = SP$), a data transmission between s_{i2} and s_{i1} occurs when thread $s_{i1}.p_{j1}$ accesses the data $s_{i2}.p_{j2}$. Each transfer has a unique source and destination buffers, which means that there is no need for the threads to synchronize because they the do not modify any shared data structure. The goal is to minimize the overall time of this procedure. Note that this data shuffling problem is very similar to the data consumption problem, in which all the cores of a machine need to consume a buffer of data produced from all N cores of the machine. As a matter of fact, we use this formulation of the problem for the rest of the discussion and experimentation.

3.1 Naive vs. coordinated shuffling

3.1.1 Naive shuffling

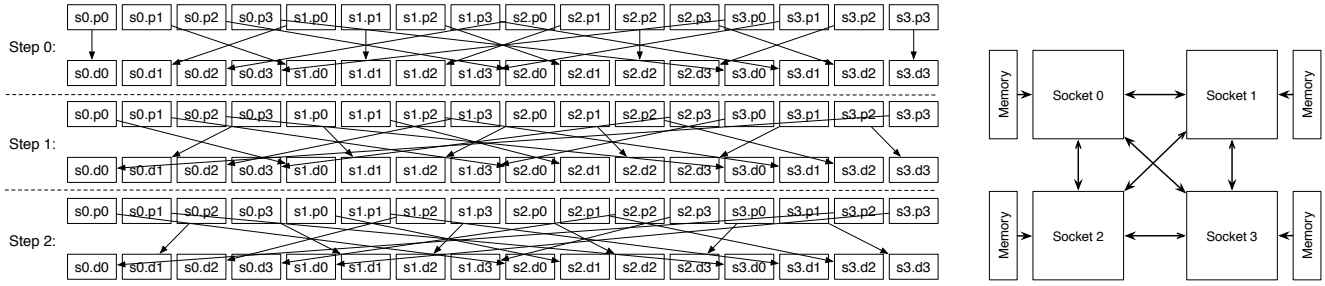
The *naive* shuffling method does not consider the NUMA characteristics of the machine. In the naive shuffling algorithm, each thread pulls data generated by all other threads starting from partition $s_0.d_0$ to $s_{S-1}.d_{P-1}$. This leads to an implementation using two nested loops, as shown below:

```
1: for i ← 0...S − 1 do
2:     for j ← 0...P − 1 do
3:         pull s_i.d_j
```



(a) Access pattern (b) QPI activity

Figure 3: The uncoordinated, naive shuffling method (first three steps).



(a) Access pattern (b) QPI activity

Figure 4: The NUMA-aware, coordinated, ring shuffling method (first three steps).

However, this naive, uncoordinated method may cause various bottlenecks. Either one of the QPI links or one of the local memory controllers may become over-subscribed, causing delays. It is difficult for a developer to predict such delays, because they are dependent on the hardware characteristics such as the interconnect topology.

Figure 3 illustrates the access pattern and QPI activity of the naive shuffling method for a 4-socket, fully-connected NUMA machine. In this example, suppose that we run 16 threads (4 threads / socket) for the naive shuffling method. As it can be seen in Figure 3(a), all threads pull data from the same partition in each step. The QPI activity of the naive shuffling method is shown in Figure 3(b). The data is pulled from the memory banks attached to Socket 0, and then is transmitted to the other three sockets through three QPI links. Consequently, the naive method underutilizes the available bandwidth: it only uses $1/4$ of the available memory channels and three of the twelve QPI links.

3.1.2 *Coordinated shuffling*

The coordinated shuffling method coordinates the usage of the bandwidth of the QPI links across all threads in the system. The basic idea behind the coordinated shuffling method is illustrated in Figure 5. There are two rings in the figure: the ring members of the inner ring represent partitions $s_0.d_0, \ldots, s_{S-1}.d_{P-1}$, while the ring members of the outer ring identify the threads $s_0.p_0, \ldots, s_{S-1}.p_{P-1}$. All data partitions are ordered in the inner ring first by thread number on a socket and then by socket number, i.e., $s_0.d_0, s_1.d_0, s_2.d_0, \ldots, s_{S-1}.d_0, s_0.d_1, s_1.d_1, \ldots$ In contrast, in the outer ring, the threads are ordered first by socket number and then by thread number, i.e., $s_0.p_0, s_0.p_1, s_0.p_2, \ldots, s_0.p_{P-1}, s_1.p_0, s_1.p_1, \ldots$

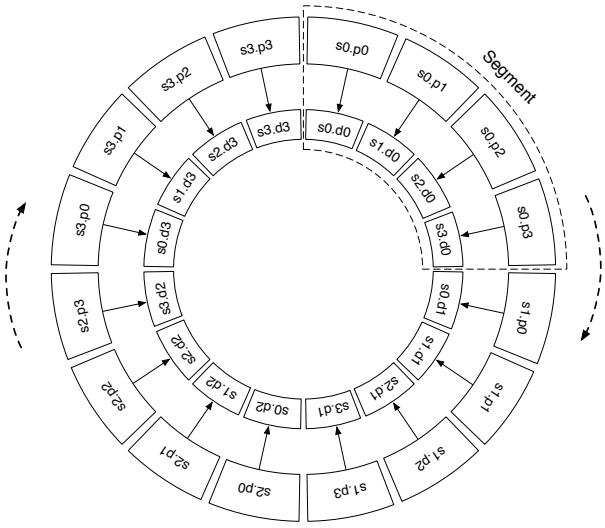


Figure 5: Ring shuffling, an example of a coordinated data shuffling policy.

During the coordinated shuffling, the inner ring remains fixed while the outer ring is rotating clockwise. At each step, all threads access the partition they point to. For instance, as shown in Figure 5, thread $s_0.p_0$ accesses the partition $s_0.d_0$, thread $s_0.p_1$ accesses the partition $s_1.d_0$, and so forth. Once the step is complete, all ring members of the outer ring are rotated clockwise to the positions of their immediately next ring members. The rotation continues until the outer ring completes a full cycle.

We briefly explain why this coordinated method achieves high bandwidth utilization on the NUMA machine. First, it is straightforward that a given thread will have accessed all partitions once the outer ring completes a full cycle. Second, consider the segment in the outer ring that spans over all threads in a given socket s_i, i.e., $s_i.p_0, \ldots, s_i.p_{P-1}$. It is clear that these partitions pointed to by the segment are always evenly located on all sockets, because partitions on various sockets are organized in an interleaved fashion. This means that there are the same number of data transfers over all links from other sockets to the socket s_i, i.e., all links have up to $\lceil P/S \rceil$ data transfers.

Given that during a shuffling operation, for a given QPI link between socket s_i and socket s_j, there is a total of $P \times P = P^2$ transmissions through the QPI link (each of the P threads on s_j transmits a data partition from each of the P partitions on s_i). Since the shuffling operation runs in $N = PS$ steps, each QPI link has at least $\lceil P^2/PS \rceil = \lceil P/S \rceil$ transmissions in each step. If the machine has a fully-connected topology, i.e., there is a QPI link for each transmission path, the coordinated shuffling method matches the lower bound on the number of transmissions.

Figure 4 illustrates the access pattern and resulting QPI activity of the coordinated shuffling method. Unlike the naive method, each partition is accessed by only one thread at each step. The coordinated method uses all QPI links as well as all memory channels in each step, as shown in Figure 4(b). Consequently, the coordinated method increases the utilization of the QPI bandwidth and provides higher performance than the naive one. Note that for the coordinated shuffling policy to perform as expected, the developer needs to control on which socket each thread is running and where the buffers are allocated. Furthermore, the developer must know the interconnect topology of the machine; to achieve the higher transmission bandwidth the interconnect network should be fully connected or, at least, symmetric.

3.2 Shuffling vs. thread migration

An alternative method to data shuffling is thread migration, in which the process and its working set moves across the sockets, rather than the data. In each step of shuffling operation, we can rebind each thread to a different core on another socket to migrate the thread to other sockets. The working sets of the threads also need to be moved to the target socket through QPI links. Essentially, thread migration can be viewed as a dual approach to data shuffling: in thread migration, data accesses go to local memory while thread state accesses potentially go through the interconnect network, whereas in data shuffling, data is explicitly moved through the interconnect network while the thread-state references remain local.

When the data size is larger than the working set of the threads, thread migration is potentially more efficient than shuffling. The trade-off depends on the ratio between data size and working set, and it is discussed in Section 4.3.

Note that the choice between shuffling and thread migration is orthogonal to the uncoordinated/coordinated methods described in Section 3.1 and, hence both, the naive (uncoordinated) and coordinated methods can also be applied in thread migration for moving the working sets. With the coordinated method, thread migration achieves higher bandwidth utilization for transmitting the working sets.

4. EVALUATION

We use two IBM X Series x3850 servers for our experiments. The first is a quad-socket with four X7560 Nehalem-EX 8-core sockets and 512 GB of main memory (128 GB per socket), in a fully-connected QPI topology. The second connects two 4-socket x3850 chassis to form one single 8-socket system with eight 8-core X7560 Nehalem-EX sockets and 1 TB main memory (128 GB per socket). The 4-socket server runs RedHat Enterprise Linux 6.1, whereas the 8-socket runs SuSE Enterprise Linux 11 SP1.

In this section, we first study in detail the performance of various shuffling methods in Section 4.1. Then we show the usefulness of data shuffling for implementing sort-merge joins in Section 4.2. Finally, in Section 4.3, we analyze the trade-offs of shuffling versus thread migration in a typical group-by database operation with aggregation.

4.1 Shuffling for NUMA

We start with a simple experiment, using shuffling methods to shuffle 1.6 billion 64-bit integers. We have N threads, each of which has $1/N$ th of the data partitioned into N pieces (and stored in its local memory). The i^{th} thread accesses the i^{th} piece of data on all threads' partitions and computes the mean of all integers on the piece of data.

4.1.1 Comparison of various policies

Figure 6 compares the maximum bandwidth achieved for the shuffling task using various shuffling methods, on both the four-socket and the eight-socket machines. We repeat each experiment 10 times and plot the mean. The vertical error bars show the min/max of the 10 runs for each experiment.

In this experiment, we evaluate the four methods described earlier; naive shuffling (Section 3.1.1), ring shuffling (Section 3.1.2), and thread migration (Section 3.2), as well as coordinated random shuffling. In the latter method, each thread randomly chooses the partition to access next. In addition, we also analyze two different implementation variants for the coordinated shuffling and thread migration methods; one in which threads are synchronized and operate in lock step from one step in the shuffling process to the next. We call those implementations *"tight"*. We remove this synchronization barrier between steps in the second implementation variant, such that threads independently move to the next step. We refer to this implementation as *"loose"*. The idea of synchronization is that it potentially avoids interference caused by diverging threads, i.e., threads that end up in different steps. The flip side of synchronization, however, is the extra cost of the synchronization itself.

We first discuss the left side of Figure 6, which shows the performance on the four-socket machine. The naive method performs poorly, achieving only a bandwidth of 22 GB/s. In this method, all threads pull data from the memory of one single socket (as shown in Figure 3(b)), and therefore are significantly limited by the local memory bandwidth of that single socket (24.7 GB/s).

The random shuffling method (labeled "Coord random (tight)" in Figure 6) achieves better performance than the naive policy, but is still inferior to the other methods. In addition, the random shuffling experiences a higher variance in bandwidth compared to the other methods (as shown by the error bars). The relatively low performance of random shuffling indicates that a simple random permutation of the

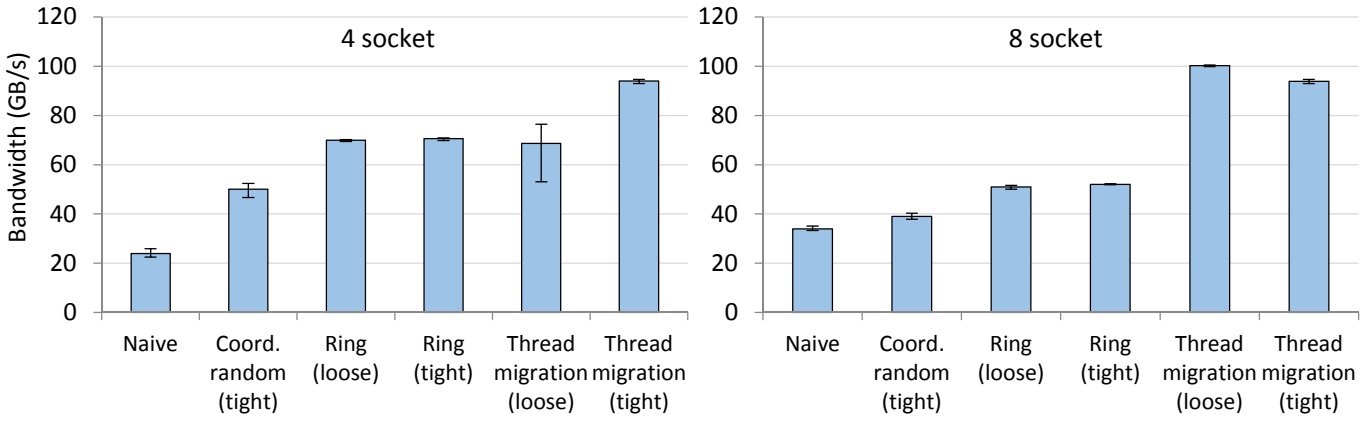


Figure 6: Data shuffling bandwidth achieved by various shuffling policies. Policies that consider the underlying QPI topology achieve up to 3× higher bandwidth than the naive approaches. Thread migration may achieve even higher bandwidth.

producer-consumer pairs without taking the hardware topology into consideration is not sufficient.

The ring shuffling method tries to maximize the QPI utilization and achieves 3× higher bandwidth than the naive method on the four-socket machine. The synchronized implementation ("Ring (tight)" in Figure 6) is slightly faster than the unsynchronized version ("Ring (loose)"). The 70 GB/s bandwidth achieved by the ring shuffling method is still lower than the aggregate bandwidth of all memory channels (4×24.7 GB/s $= 98.8$ GB/s) or all QPI links (12×10.9 GB/s $= 130.8$ GB/s). The gap is mainly due to the transmissions of snooping messages (and replies) that are exchanged on the interconnects for the system's coherency protocol.

Thread migration achieves even higher bandwidth than the ring shuffling. The synchronized implementation of the migration method achieves a bandwidth of 94 GB/s, which is close to the theoretical maximum bandwidth on that machine (4×24.7 GB/s $= 98.8$ GB/s). Here, all threads access data through the memory channel to their local memory, so only snooping messages are exchanged on the interconnect, and the number and size of these messages are small enough that they do not saturate the interconnect. The unsynchronized version of the migration method on a system with a fully populated interconnection network exhibits lower performance and has significantly higher variability. The reason is that, at any point in time, threads may end up on the same core, forming convoys. Note that each thread in this experiment has almost no state information. This represents the best use case for the thread migration method. We further evaluate the potential of thread migration in Section 4.3.

The right side of Figure 6 shows the bandwidths of these methods on our eight-socket machine. This system achieves similar bandwidths as the four-socket machine. Surprisingly, the bandwidths of the ring shuffling and migration methods do not increase on the eight-socket machine with double the local memory bandwidth. Instead, thread migration achieves similar performance on both machines, whereas the ring shuffling method performs even worse on eight sockets. The reasons for this are the non-uniform QPI links and the coherency traffic. It turns out that four of the twelve

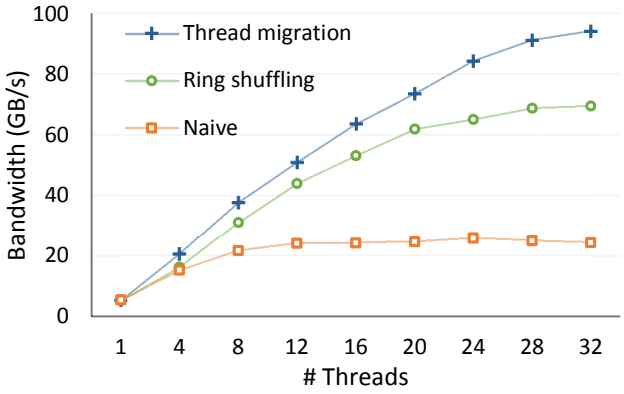


Figure 7: Bandwidth achieved when shuffling uniformly distributed data, as the number of threads increases, in a four socket system.

QPI links in the eight-socket system have a lower bandwidth (7.8 GB/s vs. 10.9 GB/s) and become the bottleneck on the overall performance. Furthermore, more snooping messages (and replies) are exchanged in the interconnect in the case of the eight-socket configuration. Those messages not only consume more bandwidth of the interconnect but also increase the latencies of the memory requests.

The results indicate that the source-based snooping protocol used for cache coherency in our Nehalem-EX systems does not seem to scale well for eight sockets. In fact, Intel notes [18, p. 72] that "[the source-snoop protocol] works well for small systems with up to four or so processors that are likely to be fully connected by Intel QPI links. However, a better choice exists for systems with a large number of processors ... The home snoop behavior is better suited for these systems where link bandwidth is at a premium and the goal is to carefully manage system traffic for the best overall performance." Unfortunately, we have not found this second protocol implemented in any system available to us.

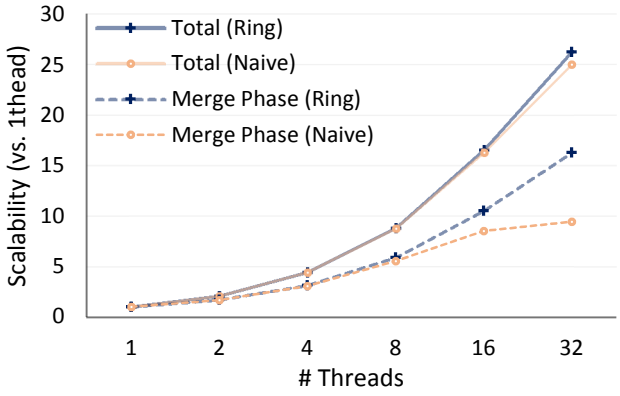


Figure 8: Scalability of the total time (solid lines) and of the merge phase (dashed lines) of a state-of-the-art sort-merge-based join implementation [1], when employing the naive and the NUMA-aware ring data shuffling methods. The merge phase, where the shuffling takes place, has superior scalability with ring shuffling.

4.1.2 *Scalability*

Figure 7 shows the bandwidth achieved (in GB/sec) as the number of threads increases when shuffling data among a four-socket system. As can be seen in Figure 7, all the three shuffling methods achieve nearly linear speed-ups before their bandwidths approach the bandwidth bottlenecks. The three methods have similar bandwidths when using less than 4 threads. However, the scaling stops at about 12 threads and 22 GB/s, because the memory bandwidth of a single socket becomes the bottleneck. The ring shuffling method achieves a linear speed-up for up to 20 threads, at which point the interconnect network becomes saturated by both the data and snooping messages. Among the three methods, thread migration scales the best as the number of threads increases. It shows linear speedup close to optimal, except for 32 threads, where the capacity limit of the memory channels is reached.

4.2 Shuffling as a primitive

An efficient, platform-optimized implementation of the shuffling primitive can be used as a primitive or as a building block in the construction of database systems. As an example, we integrate the synchronized ("tight") implementation of the ring shuffling method in the sort-merge-based join algorithm described by Albutiu et al. [1]. Their Range-partitioned Massively Parallel Sort-Merge (MPSM) join, called P-MPSM, is optimized for NUMA machines by trying to reduce the amount of data transferred among sockets.

The P-MPSM join has four phases: (1) The fact table is split into chunks and sorted locally. This implicitly partitions each chunk into a set of fact runs. (2) The dimension table is similarly split into dimension chunks, which are then range partitioned. (3) Each thread collects all associated partitions from all dimension chunks and sorts them locally into dimension runs. (4) Finally, each worker thread merges a dimension run with all corresponding fact runs. We integrated the ring shuffling algorithm into the fourth phase to read the dimension runs from remote sockets.

In this experiment, we analyze the resulting performance gain when using the ring shuffling primitive in the P-MPSM join. We join 1.6B tuples with 16M tuples, where each tuple contains a 64-bit key and a 64-bit value. Figure 8 shows the scalability of this approach as the speed-up over a single-threaded implementation. At first, the disappointing observation is that the use of the primitive can only slightly increase the overall scalability of the R-MPSM join from 25.0× to 26.2×. The reason is that the execution time is not dominated by the merge phase (4).

The time breakdown for the P-MPSM join in Figure 9 shows that sorting the fact table chunks in phase (1) overwhelmingly dominates the execution time. Since the shuffling primitive is only used in the merge phase (4), we plot the scalability of this phase separately in Figure 8. For the merge phase alone, our more efficient implementation of the ring shuffling is able to increase the scalability from 9.4× to 16.3×, and hence, almost doubles the scalability.

The left-hand side of Figure 9 depicts the time breakdown for the naive shuffling implementation, and the right-hand side for the synchronized ring shuffling. It can be seen that the already small fraction of the merge phase is further reduced by the ring shuffling. The difference in the merge phase between the two implementations becomes noticeable when using more than 8 threads, i.e., when the number of threads in a single socket is exceeded, so inter-socket communication is required.

Although sorting the fact table is currently the dominating phase in this case, the merge phase is likely to dominate as the number of threads increases. Furthermore, since the time spent for sorting is likely to be reduced by hardware accelerators or by exploiting more cache-efficient algorithms, and future systems are likely to increase their number of cores significantly, it is quite probable that the performance of the join will increasingly depend on the efficiency of the data shuffling.

4.3 Data shuffling vs. thread migration

Data shuffling is not a panacea and, as we saw earlier in Section 4.1.1, there is potential for moving the computation to the data rather than shuffling the data around. Migrating threads makes sense, especially when the accompanying working set of the migrating thread is relatively small. In this section, we try to evaluate the trade-off between data shuffling (moving the data) and thread migration (moving the computation) further. In order to do that, we devised a simple benchmark that models the following group-by operation traditionally found in database systems:

```
SELECT key, SUM(value) FROM table GROUP BY key
```

We measure the bandwidth at which a large number of key-value pairs can be grouped by their key and the values with the same key summed up. The implementation uses partitioning by the key. The size of both the key and the value is 8 bytes.

Figure 10 shows the aggregation bandwidth, as the number of distinct keys per partition increases. The number of partitions is equal to the number of processing cores each machine has. That is, there are 32 partitions for the four-socket system and 64 partitions for the eight-socket system. We compare the performance of four implementations. The first two are based on shuffling — one using the naive method, the other with ring shuffling. We also evaluate two

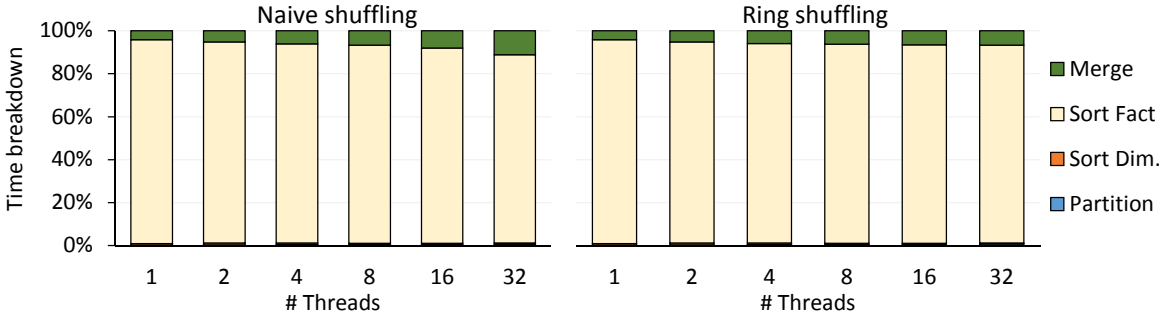


Figure 9: Breakdown of the time spent in each phase of the sort-merge-based join implementation when data shuffling uses the naive approach (left) and when it uses the coordinated ring shuffling (right). The merge phase, in which the data shuffling take place, takes half the fraction of time when done with the NUMA-aware method.

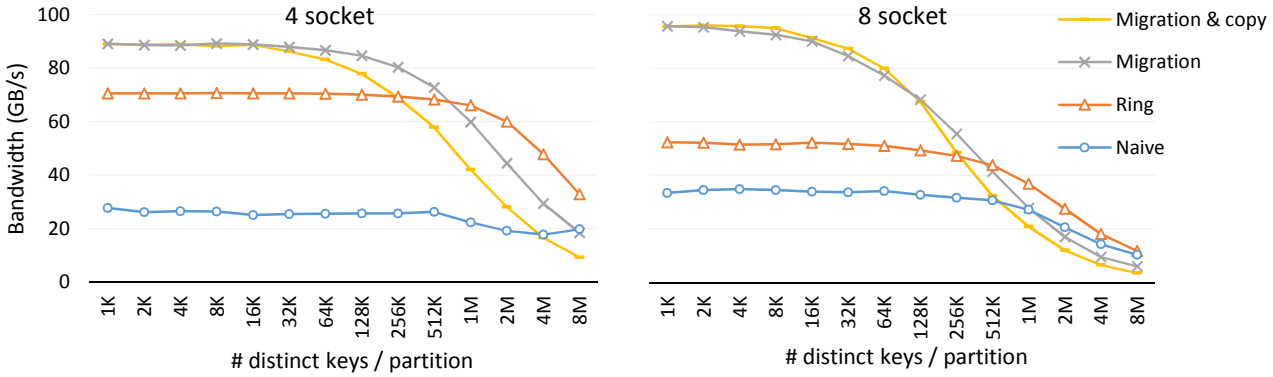


Figure 10: Effective aggregation bandwidth of two shuffling-based and two migration-based implementations of partitioned aggregation, as the number of distinct groups per partition increases.

thread migration-based implementations. In the first one, the threads do not copy their state (aggregated data) along when they migrate to the socket containing the partition they need to aggregate next. This implies that the first memory request to each thread's aggregation state will cause a cache miss and a resulting remote memory access over the interconnect. We label this implementation as *Migration* in Figure 10. The last implementation avoids these remote accesses during the processing of the aggregation by explicitly copying over each thread's aggregation state to its new socket before migrating the threads themselves, i.e., each migrating thread copies its aggregation state from its local memory to the memory location of the soon-to-be consumed partition. We call this implementation *Migration & copy* in Figure 10.

Figure 10 shows that the performance of migration is significantly higher than shuffling if the number of distinct groups per partition (i.e., the state of each migrating thread) is less than 256k keys. In particular, if the partitions contain less than 32k keys and so each thread's aggregation state fits into L2 cache, the difference between the performance of the shuffling and migrating methods corresponds to the bandwidth difference we observed in Figure 6. As the thread state grows, i.e., the number of distinct keys per partition increases, we observe a crossover point at which the optimized shuffling implementation is more efficient. The crossover point is at about 512k keys per partition. That means that when the total number of distinct groups in an aggregation query is larger than 16M or 32M keys for the four-socket machine — and 32M or 64M keys for the eight-socket machine — the shuffling-based implementation should be preferred over the migration-based one. A query optimizer capable of predicting the expected number of distinct keys may then choose the appropriate aggregation implementation.

In order to analyze the impact of the individual approaches on the overall bandwidth, a time breakdown is given in Figure 11 for the four-socket system. We classify the operations that are performed during the aggregation into three categories and report the total time spent for each category. The first contains accesses to the unaggregated data (labeled as *Read partitions* in Figure 11). The second category includes operations to perform the actual aggregation, i.e., accesses and updates to the aggregation state with the newly added data (labeled *Upd. hash table*). The third category is only applicable for *Migration & copy*. It represents the cost to transfer the intermediate aggregation result from the local memory into the memory that contains the next partition to be aggregated (labeled *Copy state* in Figure 11).

Several observations can be made from Figure 11. First, the naive method is bottlenecked by the access to the un-

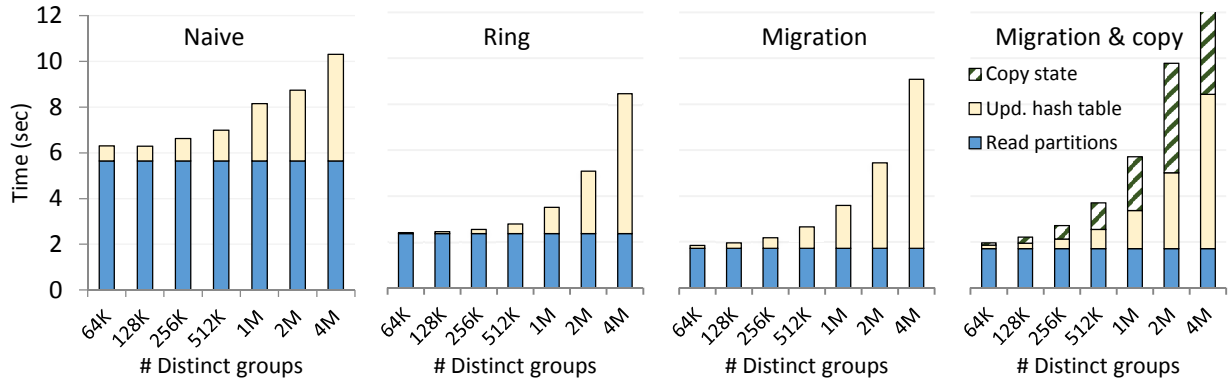


Figure 11: Time breakdown for the four aggregation implementations, each with a different shuffling approach, as the number of distinct groups per partition increases, in the four-socket machine. The time is broken into the time to scan the table, update the grouping hash table, and move the state, in the case of migration & copy implementation.

aggregated data (scans). It takes as much as 3 times more to access the data, which is consistent with the results presented in Section 4.1.1. Second, the thread migration policies access the unaggregated data 25% faster than the ring shuffling method, but as the size of the local aggregation result (and thus thread state) increases, the execution time is dominated by the time it takes to access the local aggregation result. The crossover point is between 1M and 2M distinct groups. Finally, the method that migrates the thread and copies over the local aggregation result is much slower on large result sizes, because of the overhead of copying over the local aggregation result.

5. RELATED WORK

Exchanging partitioned data among parallel processes, particularly to do joins, has a long history in the database literature (e.g., [5, 10, 11]). Most of this work simply tried to minimize the aggregate amount of data transmitted, rather than concern itself with differing limits on the capacity of the individual links between nodes, and did not try to orchestrate the timing or routing of these transmissions between nodes. More recent papers have focussed on multi-core efficiency (e.g., [7, 12, 14]), but assumed that any core could access the memory of another core in about the same time as their own. The architecture community has also studied many aspects of DBMS performance from a perspective of efficiency on parallel hardware (e.g., [16, 25]). Much of this work has focused on symmetric multiprocessors (SMPs) or clusters; multi-socket multi-core systems are still fairly new, and relatively less studied.

The memory performance of NUMA machines has been evaluated e.g., in [19, 20]. These studies as complementary to the NUMA memory bandwidths reported in Section 2.

Recently, Neumann et al. presented a NUMA-aware distributed sort-merge join implementation [1]. This algorithm did not focus on the data shuffling step of the algorithm, but our NUMA-aware data shuffling algorithm was able to speed up their algorithm by up to 8% compared to a naive shuffle, as shown in the previous section.

On the other hand, on-line transaction processing (OLTP) is a primarily latency-bound application, since synchronization is latency-bound. Thus, the NUMA effect is even more profound in OLTP. Porobic et al. [24] describe how performance of OLTP systems is sensitive to the latency variations arising in NUMA systems, proposing that in many cases the NUMA hardware should be treated as a cluster of independent nodes. [15] presents a NUMA-aware variation of an efficient log buffer insert protocol.

In this paper we present evidence for the potential of thread migration instead of data shuffling. Moving the computation to where the data resides seems to be especially beneficial when the working set of the thread is small. Analogous execution models have been proposed for both business intelligence (e.g., the DataPath system [2]) as well as transaction processing (e.g., the data-oriented transaction execution model of [22, 23]).

Data shuffling has been studied extensively in the MapReduce framework. Orchestra [6] was proposed as a global control architecture to manage transfer activities, such as broadcast and shuffle, in cloud environments. Similar to our work, the scheduling in Orchestra is topology-aware. For a shuffle operation, each node simultaneously transfers multiple transmissions to various destination nodes. Orchestra coordinates the speeds of each transmission to achieve the global optimal performance. In contrast, in our method, each thread only accesses one target data at one time. The NUMA-aware shuffling method coordinates the order of transmissions to various destinations.

Recently, the applicability of the MapReduce model was evaluated on NUMA machines [27]. Phoenix++ [26] is a revision of the Phoenix framework that applies the MapReduce model on NUMA machines. Our work provides an opportunity to optimize the shuffling operations in the MapReduce model on NUMA machines.

The NUMA architecture also impacts on the design of operating systems. A new OS architecture, called multi kernel, was proposed to meet the networked nature of the NUMA machines [4]. The multikernel treats a NUMA machine as a small distributed system of processes that communicate via message-passing. In contrast, our work proposes to provide an efficient shuffling primitive on NUMA macines that is used for building data management systems.

6. CONCLUSIONS

The days when software performance automatically improved with each new hardware version are long gone. Now, transistor counts continue to grow, but one has to carefully design software to eke out every bit of performance from the hardware.

However, over-engineering software for each supported hardware configuration further complicates already complex DBMS systems. So we argue instead that it is crucial to identify core, performance-critical DBMS primitives that can be tuned to specific hardware and provided as hardware-specific libraries (as done, for example in BLAS [3]), while keeping the remaining DBMS code fairly generic.

In this paper, we have used the common data management primitive of data shuffling to illustrate the significant benefit of optimizing algorithms for NUMA hardware. Going forward, we expect many other such primitives to be similarly investigated, and to be hardware-optimized or even co-designed with new hardware configurations and technologies, to realize major performance gains.

Acknowledgments

We cordially thank Volker Fischer from IBM's Systems Optimization Competence Center (SOCC) for setting up and providing help with the 8-socket machine.

References

[1] M.-C. Albutiu, A. Kemper, and T. Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *PVLDB*, 5(10), 2012.

[2] S. Arumugam, A. Dobra, C. M. Jermaine, N. Pansare, and L. Perez. The DataPath system: a data-centric analytic processing engine for large data warehouses. In *SIGMOD*, 2010.

[3] C. Baru and G. Fecteau. An overview of DB2 parallel edition. In *SIGMOD*, 1995.

[4] 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 *SOSP*, 2009.

[5] H. Boral, W. Alexander, L. Clay, G. P. Copeland, S. Danforth, M. J. Franklin, B. E. Hart, M. G. Smith, and P. Valduriez. Prototyping Bubba, a highly parallel database system. *IEEE TKDE*, 2, 1990.

[6] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing data transfers in computer clusters with orchestra. In *SIGCOMM*, 2011.

[7] J. Cieslewicz and K. A. Ross. Adaptive aggregation on chip multiprocessors. In *VLDB*, 2007.

[8] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI*, 2004.

[9] D. J. Dewitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. The Gamma database machine project. *IEEE TKDE*, 2(1), 1990.

[10] D. J. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *CACM*, 35, 1992.

[3] http://www.netlib.org/blas/

[11] G. Graefe. Encapsulation of parallelism in the Volcano query processing system. In *SIGMOD*, 1990.

[12] N. Hardavellas, I. Pandis, R. Johnson, N. Mancheril, A. Ailamaki, and B. Falsafi. Database servers on chip multiprocessors: Limitations and opportunities. In *CIDR*, 2007.

[13] Intel. *Intel Xeon Processor 7500 Series Architecture: Datasheet, Volume 2*, 2010. Available at http://www.intel.com/Assets/PDF/datasheet/323341.pdf.

[14] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: a scalable storage manager for the multicore era. In *EDBT*, 2009.

[15] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki. Scalability of write-ahead logging on multicore and multisocket hardware. *The VLDB Journal*, 20, 2011.

[16] K. Keeton, D. A. Patterson, Y. Q. He, R. C. Raphael, and W. E. Baker. Performance characterization of a quad Pentium Pro SMP using OLTP workloads. In *ISCA*, 1998.

[17] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3), 1979.

[18] R. A. Maddox, G. Singh, and R. J. Safranek. *Weaving High Performance Multiprocessor Fabric*. Intel Press, 2009.

[19] Z. Majo and T. R. Gross. Memory system performance in a NUMA multicore multiprocessor. In *SYSTOR*, 2011.

[20] C. McCurdy and J. S. Vetter. Memphis: Finding and fixing NUMA-related performance problems on multicore platforms. In *ISPASS*, 2010.

[21] A. Michael, H. Li, and P. Sart. Facebook open compute project. In *HotChips*, 2011.

[22] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *PVLDB*, 3(1), 2010.

[23] I. Pandis, P. Tözün, R. Johnson, and A. Ailamaki. PLP: page latch-free shared-everything OLTP. *PVLDB*, 4(10), 2011.

[24] D. Porobic, I. Pandis, M. Branco, P. Tözün, and A. Ailamaki. OLTP on hardware islands. *PVLDB*, 5(11), 2012.

[25] P. Ranganathan, K. Gharachorloo, S. V. Adve, and L. A. Barroso. Performance of database workloads on shared-memory systems with out-of-order processors. In *ASPLOS-VIII*, 1998.

[26] J. Talbot, R. M. Yoo, and C. Kozyrakis. Phoenix++: modular MapReduce for shared-memory systems. In *MapReduce*, 2011.

[27] R. M. Yoo, A. Romano, and C. Kozyrakis. Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system. In *IISWC*, 2009.