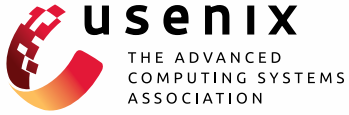

Shoal: Smart Allocation and Replication of Memory For Parallel Programs

Stefan Kaestle, Reto Achermann, and Timothy Roscoe, ***ETH Zürich;***
Tim Harris, ***Oracle Labs, Cambridge***

https://www.usenix.org/conference/atc15/technical-session/presentation/kaestle

This paper is included in the Proceedings of the
2015 USENIX Annual Technical Conference (USENIC ATC '15).

July 8–10, 2015 • Santa Clara, CA, USA

ISBN 978-1-931971-225

Open access to the Proceedings of the
2015 USENIX Annual Technical Conference
(USENIX ATC '15) is sponsored by USENIX.

Shoal: smart allocation and replication of memory for parallel programs

Stefan Kaestle, Reto Achermann, Timothy Roscoe, Tim Harris*
Systems Group, Dept. of Computer Science, ETH Zurich **Oracle Labs, Cambridge, UK*

Abstract

Modern NUMA multi-core machines exhibit complex latency and throughput characteristics, making it hard to allocate memory optimally for a given program's access patterns. However, sub-optimal allocation can significantly impact performance of parallel programs.

We present an array abstraction that allows data placement to be automatically inferred from program analysis, and implement the abstraction in Shoal, a runtime library for parallel programs on NUMA machines. In Shoal, arrays can be automatically replicated, distributed, or partitioned across NUMA domains based on annotating memory allocation statements to indicate access patterns. We further show how such annotations can be automatically provided by compilers for high-level domain-specific languages (for example, the Green-Marl graph language). Finally, we show how Shoal can exploit additional hardware such as programmable DMA copy engines to further improve parallel program performance.

We demonstrate significant performance benefits from automatically selecting a good array implementation based on memory access patterns and machine characteristics. We present two case-studies: (*i*) Green-Marl, a graph analytics workload using automatically annotated code based on information extracted from the high-level program and (*ii*) a manually-annotated version of the PARSEC Streamcluster benchmark.

1 Introduction

Memory allocation in NUMA multi-core machines is increasingly complex. Good placement of and access to program data is crucial for application performance, and, if not carefully done, can significantly impact scalability [3, 13]. Although there is research (e.g. [7, 3]) in adapting to the concrete characteristics of such machines, many programmers struggle to develop software applying these techniques. We show an example in Section 5.1.

The problem is that it is unclear which NUMA optimization to apply in which situation and, with rapidly evolving and diversifying hardware, programmers must repeatedly make manual changes to their software to keep up with new hardware performance properties.

One solution to achieve better data placement and faster data access is to rely on automatic online monitoring of program performance to decide how to migrate data [13]. However, monitoring may be expensive due to missing hardware support (if pages must be unmapped to trigger a fault when data is accessed) or insufficiently precise (if based on sampling using performance counters). Both approaches are limited to a relatively small number of optimizations (e.g. it is hard to incrementally activate large pages or switch to using DMA hardware for data copies based on monitoring or event counters)

We present Shoal, a system that abstracts memory access and provides a rich programming interface that accepts hints on memory access patterns at the runtime. These hints can either be manually written or automatically derived from high-level descriptions of parallel programs such as domain specific languages. Shoal includes a machine-aware runtime that selects optimal implementations for this memory abstraction dynamically during buffer allocation based on the hints and a concrete combination of machine and workload. If available, Shoal is able to exploit not only NUMA properties but also hardware features such as large pages and DMA copy engines. Our contributions are:

- a memory abstraction based on arrays that decouples data access from the rest of the program,
- an interface for programs to specify memory access patterns when allocating memory,
- a runtime that selects from several highly tuned array implementations based on access patterns and machine characteristics and can exploit machine specific hardware, features
- modifications to Green-Marl [20], a graph analytics language, to show how Shoal can extract access patterns automatically from high-level descriptions.

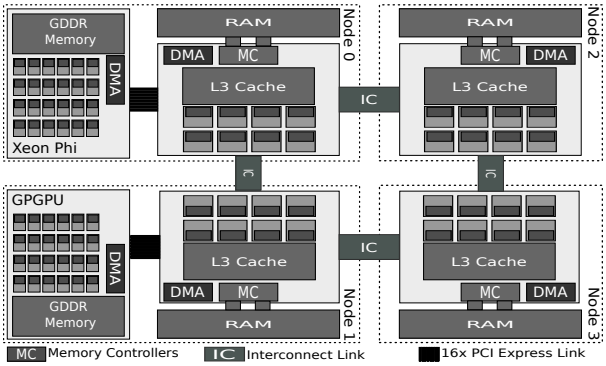

Figure 1: Architecture of a modern multi-core machine.

2 Motivation

Modern multi-core machines have complex memory hierarchies consisting of several memory controllers placed across the machine – for example, Figure 1 shows such a machine with four host main memory controllers (one per processor socket). Memory latency and bandwidth depend on which core accesses which memory location [8, 10]. The interconnect may suffer congestion when access to memory controllers is unbalanced [13].

Future machines will be more complex: they may not provide global cache coherence [21, 25], or even shared global physical addresses [1]. Even today, accelerators like Intel's Xeon Phi, GPGPUs, and FPGAs have higher memory access costs for parts of the physical memory space [1]. Such hardware demands even more care in application data placement.

This poses challenges to programmers when allocating and accessing memory. First, detailed knowledge of hardware characteristics and a good understanding of their implications for algorithm performance is needed for efficient scalable programs. Care must be taken when choosing a memory controller to allocate memory from, and how to subsequently access that memory.

Second, hardware changes quickly meaning that design choices must be constantly re-evaluated to ensure good performance on current hardware. This imposes high engineering and maintenance costs. This is worthwhile in high-performance computing or niche markets, but general purpose machines have too broad a hardware range for this to be practical for many domains. The result is poor performance on most platforms.

These problems can be seen in much code today. Programmers take little care of where memory is allocated and how it is accessed. In cases like the popular Streamcluster benchmark (evaluated in Section 5.1) and applications from the NAS benchmark suite [13], memory is allocated using a low-level malloc call which provides no guarantees about where memory is allocated or other details such as the page size to use.

For example, Linux currently employs a first-touch memory allocation strategy. Memory is not allocated directly when calling `malloc`, but mapped only when the corresponding memory is first accessed by a thread. This resulting page fault will cause Linux to back the faulting page from the NUMA node of the faulting core.

A surprising consequence of this choice is that on Linux the implementation of the initialization phase of a program is often critical to its memory performance, even through programmers rarely consider initialization as a candidate for heavy optimization, since it almost never dominates the total *execution time* of the program. To see why, consider that `memset` is the most widely used approach for initializing the elements of an array. Most programmers will spend little time evaluating alternatives, since the time spent in the initialization phase is usually negligible. An example is as follows:

```
// ---- Initialization (sequential) -----------
void *ptr = malloc(ARRSIZE);
memset(ptr, 0, ARRSIZE);
// ---- Work (parallel, highly optimized) -----
execute_work_in_parallel();
```

The scalability of a program written this way can be limited. `memset` executes on a single core and so all memory is allocated on the NUMA node of that core. For memory-bound parallel programs, one memory controller will be saturated quickly while others remain idle since all threads (up to 64 on the machines we evaluate) request memory from the same controller. Furthermore, the interconnect close to this memory controller will be more susceptible to congestion.

There are two problems here: (*i*) memory is not allocated (or mapped) *when* the interface suggests (memory is not allocated inside `malloc` itself but later in the execution) and (*ii*) the choice of where to allocate memory is made in a subsystem (the OS kernel) that has no knowledge of the intended access patterns of this memory.

This can be addressed by tuning algorithms to specific operating systems. For example, we could initialize memory using a parallel for loop:

```
// ---- Initialization (parallel) -------------
void *ptr = malloc(ARRSIZE);
#pragma omp parallel for
for (int i=0; i<ARRSIZE; i++)
  init(i);
// ---- Work (parallel, highly optimized) -----
execute_work_in_parallel();
```

This will be faster and retain scalability in current versions of Linux. The first-touch strategy will equally spread out memory across all memory controllers, which balances the load on them and reduces contention on individual interconnect links.

One drawback of this strategy is the loss of portability and scalability when the OS kernel's internal memory

allocation policies change. Furthermore, it also requires correct setup of OpenMP's CPU affinity to ensure that all cores participate in this parallel initialization phase in order to spread memory equally on all memory controllers. Finally, we might do better: allocate memory close to the cores that access it the most.

Beyond simple placement of data, ideas and techniques from traditional distributed systems like replication and partitioning can help to improve memory management [31]. Replication localizes data access by storing several copies of the same data, distributing load and reducing communication costs. Replication carries the cost of maintaining consistency when updating data, as well as increasing the program's memory footprint. Partitioning chunks data and places these blocks onto different nodes. This balances load, and, if work is scheduled close to data it is accessing, also localizes array accesses. The key challenge in applying these techniques is that the choice of a good distribution strategy depends critically on concrete combinations of machine and workload.

Our work starts with the observation that memory access patterns of applications are often encoded in high-level languages or known by programmers. We show how this information can be used to tune memory placement and access without programmers having to understand the characteristics of the machine at hand.

Automatic annotations from high-level DSLs. A trend we exploit is the emergence of high-level domain specific languages (DSLs) [20, 37, 41]. These languages are known for the ease of programming since their semantics closely match a specific application domain. DSLs typically compile to a low-level language (such as C), possibly with several backends depending on the target machine to execute the program. DSLs can provide us with memory access patterns directly from the input program with relatively simple modifications to high-level compilers. Listing 1 shows an example program for the Green-Marl graph analytics DSL.

Memory access patterns from two of Green-Marl's high-level constructs in the PageRank example can be determined as follows: (*i*) `Foreach (T: G.Nodes)` means the nodes-array will be accessed sequentially, read-only, and with an index, and (*ii*) `Sum(w: t.InNbrs)` implies read-only, indexed accesses on in-neighbors array.

```
Procedure pagerank(/* arguments */) {
    // .. initialization here
    Do {
        diff = 0.0; cnt++;
        Foreach (t: G.Nodes) {
            Double val = (1-d) / N + d*
                Sum(w: t.InNbrs) {
                   w.pg_rank / w.OutDegree()} ;
            diff += | val - t.pg_rank |;
            t.pg_rank <= val @ t;
        }
    } While ((diff > e) && (cnt < max));
}
```

Listing 1: Excerpt from Green-Marl's PageRank

We argue that memory access patterns encoded in DSLs present a significant performance opportunity, and should be passed to the runtime to enable automatic tuning of memory allocation and access. Since low-level code is generated by the DSL compiler, it is also relatively easy to change the programming abstractions used by the generated code for accessing memory. Only the compiler (rather than the input program) must be changed in such a case.

Manual annotations. Even without a DSL, programmers often know data access patterns when writing a program. They understand the semantics of their programs and, hence, how memory is accessed, but have no way of passing this knowledge to the runtime to guide data placement and access. Existing interfaces intended to enable this coarse-grained and inflexible. One example is libnuma's [34] NUMA-aware memory allocation, which allows a client to specify which node memory should be allocated from, but does not allow combining this with other allocation options (such as large pages), and requires a programmer to manually integrate this with parallel task scheduling.

In Shoal, we automatically tune data placement and access based on memory access patterns and hints provided by a high-level compiler or by programmers. We introduce (*i*) a new interface for memory allocation, including machine-aware `malloc` call that accepts hints to guide placement and (*ii*) an abstraction for data access based on arrays. For these arrays, we provide several implementations including data distribution, replication and partitioning. All implementations can be interchanged transparently without the need to change programs. Our abstraction also admits implementations that are tuned to hardware features (such as DMA engines) or accelerators (Xeon Phi). The Shoal library automatically selects array implementations based on array access patterns and machine specifications. We currently support adaptions based on the NUMA hierarchy, DMA engines, and large MMU pages.

The result is that Shoal allows programmers to write programs that achieve good performance without having (*i*) to understand machine characteristics and (*ii*) needing to constantly rewrite applications in order to keep up with hardware changes. We demonstrate Shoal using the Green-Marl graph DSL, and the Streamcluster low-level C program from the PARSEC benchmark suite.

3 Shoal's array abstraction

Shoal's memory abstraction is based on arrays. We found this sufficient for the workloads we have been looking at in the context of this research, but we expect to add more data types in the future. We provide several array implementations, but all of them implement the same interface. This allows Shoal to select an implementation transparently to the programmer. The optimal choice depends on machine characteristics and memory access patterns.

```
// allocate an array
template<class T>
shl_array<T>* shl__malloc_array(size_t size,
                        bool ro, bool indexed,
                        bool used);
// get element at position i
T get(size_t i);
// set element at position i to v
void set(size_t i, T v);
// number of elements
size_t get_size(void);
// copy from another Shoal array
int copy_from_array(shl_array<T> *src);
// initialize every element with value
int init_from_value(T value);
// copy from a non-Shoal array
void copy_from(T* src);
// calculate the CRC checksum
unsigned long get_crc(void);
// initialize the current thread
void shl__thread_init(void);
// synchronize replicas
void shl__repl_sync(void* src, void **dest,
                size_t num_dest, size_t size);
```

Listing 2: Interface of Shoal

3.1 Interface and programming model

Listing 2 illustrates Shoal's programming interface, which decouples computation and memory access allowing transparent selection of different array implementations. Besides the usual `set()` and `get()` operators, we provide a collection of high-level functions to initialize memory and copy between Shoal arrays.

Thread initialization. In OpenMP, Shoal uses builtin functions to determine the thread ID and the corresponding replica to be used. Per-thread array pointers can otherwise be setup by manually calling `shl__thread_init()` on each thread.

Array allocation. `shl__malloc_array` allocates Shoal arrays and selects the best implementation for the machine it is running on based on memory access pattern hints given as arguments. Shoal always maps all pages of an array to guarantee memory allocation and avoid non-determinism.

Data operations. Reads and writes to arrays are performed with `get()` and `set()`, but we also provide optimized high-level array operations for initializing and copying arrays. These provide relaxed consistency guarantees: the order in which elements are initialized or copied is not specified, allowing these operations to be parallelized and offloaded to DMA engines in an asynchronous fashion. Writes to replicas can be realized by writing to the master copy and propagating the changes to all replicas using `shl__repl_sync()`. This allows to re-initialize replicated arrays, for example to reuse otherwise read-only buffers in streaming applications.

3.2 Array types

We currently provide four array implementations.

Single-node allocation. Allocates the entire array on the local node. While limited in scalability, performance is independent of the OS since memory is guaranteed to be mapped in the allocation phase. Single-node arrays are rarely used for parallel programs.

Distribution. Distributed arrays allocate data equally across NUMA nodes. The precise distribution is not specified and depends on the implementation. This reduces pressure on memory controllers, but can lead to high latency or congestion if many accesses are remote. The performance of distributed arrays can be non-deterministic, as data is scattered semi-randomly and might vary between program executions.

Replication. Several copies of the array are allocated. We currently always place one replica on each memory controller. All data is then accessed locally. In addition to distributing load across the system, this reduces pressure on the interconnect at the cost of increased memory footprint.

Partitioning. Partitioning is a form of distribution where data is spread out in the machine such that work units can be executed local to where their data is allocated. If done carefully, array accesses are local as with replication, but without the increased memory footprint. This implies a scheduling challenge, since the working set for each thread must be known and the jobs scheduled accordingly.

3.3 Selection of arrays

In selecting an array implementation, we try to: (*i*) maximize local access to minimize interconnect traffic, (*ii*) load-balance memory on all available controllers to avoid points of contention, and (*iii*) transparently use hardware features when available (e.g. DMA and large pages).

We show our policy for selecting array implementations in Figure 2: 1. we use partitioning if the array is only accessed via an index, 2. we enable replication if the array is read-only and fits into every NUMA node of the host machine, and 3. we otherwise use a uniform distribution among all available memory controllers.

We only replicate read-only arrays, as we found that the cost for maintaining consistency dominates the performance benefits in current NUMA machines (Section 5.4) – however, we plan to revisit this for more complex NUMA hierarchies. In case of limited RAM, where the increase in working set size with replication is not tolerable, we can selectively activate replication based on the cost function of memory accesses extracted from the high-level program.

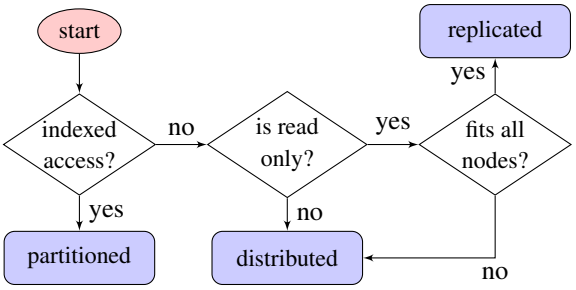

Figure 2: Array Selection

Large pages. If available, we use *large pages*. This is not always optimal, and the impact of using large pages is hard to anticipate, but on average enabling large pages improves performance (in the future, we plan to enable large pages per-array). Most current multi-core machines have independent TLBs for different page sizes, suggesting it might be useful to retain some arrays on normal pages. Also, the TLBs coverage for randomly accessed large arrays is still not sufficient to prevent TLB misses. One approach would use large pages for mostly-sequential array access, and 4k pages for randomly accessed data. We also plan to use *huge* pages (typically 1GB), which may keep most of the working set covered by the TLB even for big workloads.

Overall, we have found this policy to be simple, but effective.

4 Implementation

The Shoal runtime library is structured in two parts: (*i*) a high-level array representation as defined in Section 3 based on C++ templates and (*ii*) a low-level, OS-specific backend. We now describe the work-flow invoked when Shoal is used together with a high-level DSL, and then describe our low-level backends.

Figure 3 shows how Shoal is used with high-level, parallel languages.

High-level program. The input program is written in a high-level parallel language, which in this paper is Green-Marl, a DSL for graph analysis. Many other high-level languages such as SQL [18] and OptiML [37] provide similar resource usage information to the kind we extract from Green-Marl; we show an example of a Green-Marl expression of PageRank in Listing 1.

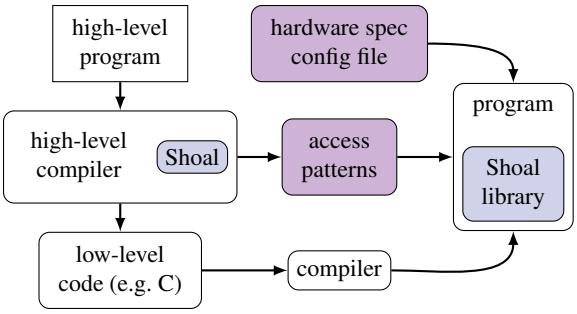

Figure 3: Shoal system overview

High-level compiler. High-level DSLs often encode access patterns in an intuitive way: for instance, the Green-Marl DSL features language constructs to access all nodes in a graph (see `Foreach (t: G.Nodes)` in Listing 1). From the language specification, we know that this represents a read-only and sequential data access to the nodes array. The high-level compiler translates the input into low-level code while such access information is lost. Our modifications to the Green-Marl compiler extract this knowledge about array access patterns from the input source code and makes it accessible to the generated code which uses Shoal's array abstraction.

Low-level code with array abstractions. The generated code – here, C++ – uses Shoal's abstraction to allocate and access memory. At compile time the concrete choice of array implementation is *not* made; this happens later at runtime based on hardware specifications.

Access patterns. In high-level languages, memory access are usually implicit and translated into simple load and store instructions. In Shoal, however, we use the compiler to generate important information about load/-store patterns which is then used by the runtime library.

Firstly, we capture the *read/write-ratio*. The number of reads and writes by a program is workload specific, but Shoal can still in many cases extract a formula estimating the number of reads and writes for a given input. For example, the number of reads in Green-Marl's PageRank rank array is: $kE * kN$, where E is the number of edges and N is the number of nodes. Currently, Shoal derives these formulas but only uses them to determine if the array is read-only; we expect more sophisticated uses of this information in the future. Secondly, we infer if all accesses to an array are based solely on the loop variable of a parallel loop. `tmp` indicates the array is used for temporary values, `ro` that it is read-only, etc.

An example of the automatically extracted information for Green-Marl's PageRank can be seen on Table 1.

Hardware specification. The Shoal runtime takes the hardware configuration of the system into account when selecting array implementations. Currently, we consider the following hardware features: (*i*) *NUMA topology*: Shoal has a NUMA-aware array allocation function that

array	N	E	ro	std	tmp	indexed
`begin`	y		y	y		
`r_begin`	y		y	y		
`r_node_idx`		y	y	y		
`pg_rank`	y					
`pg_rank_nxt`	y				y	y

Table 1: Shoal's extracted array properties for PageRank

attempts to distribute load on all memory controllers and localize access to reduce pressure on interconnects. (*ii*) *RAM size*: available memory can limit which arrays can be replicated. (*iii*) *Page size*: Modern systems offer various page sizes and often provide a dedicated TLB for each size. The use of large and huge pages is useful in reducing TLB misses [17] in large working sets. (*iv*) *DMA engines*: Some CPUs have integrated DMA engines [22]. We make use of these for copy and initialization.

Shoal program. The Shoal library takes care of selecting array implementations based on the extracted access patterns, and hardware specification of the machine. The executable generated by the compiler is a program binary which links against the Shoal library.

OS-specific backends. To improve portability, we separate high-level array implementations from low-level, OS-dependent functions which mediate access to the memory allocation facilities or DMA devices. Currently, we run on the Linux and Barrelfish [6] OSes to demonstrate portability.

Topology information. Shoal needs to obtain information about the system architecture including number of NUMA nodes, their sizes and corresponding CPU affinities. On Linux, this information can be obtained using *libnuma* [34]. In Barrelfish, hardware information is stored in the system knowledge base [33].

Scheduling. For replication and partitioning, Shoal must map threads to cores. On Linux we pin threads by setting the affinities, whereas on Barrelfish we directly create threads on specific cores. Given a concrete data distribution, scheduling can be optimized to execute work units close to where data is accessed. To date, Shoal is not fully integrated with the OpenMP runtime and we use a static OpenMP schedule for partitioning to ensure that work units are executed close to the partitions they are working on. This works well for balanced workloads, but can lead to significant slowdown compared to dynamic schedules if the cost of executing work units is non-uniform. In the future, we plan to design and integrate our own OpenMP runtime to provide us fine-grained control of scheduling without losing performance for unbalanced workloads. An alternative approach would schedule work units on partitions using OpenMP 4.0's `team`-statement.

Memory allocation. We want to provide strong guarantees on where memory is allocated, but allocation policies are not consistent across OSes – indeed, they even change between different version of the same OS. Linux, for instance, implements a first touch allocation policy, which causes confusion about where and when memory will actually be allocated. Libraries such as libnuma provide an interface which gives more control, but this lacks support for large and huge pages. Barrelfish [7] gives the user the ability to manage its own address space via self-paging [19]: an application requests memory explicitly from a specific NUMA node and maps it as it wishes.

These systems provide different trade-offs between complexity, portability, and maintainability of application code and efficient use of the memory system: an explicit, flexible interface imposes an additional burden on the client. We believe that programmers should not have to deal with this complexity and want to avoid manual tuning to adapt programs to new machines.

5 Evaluation

Our goal in this section: we show that programs scale and perform significantly better with Shoal than with a regular memory runtime. We also show a comparison of our array implementations and analyze Shoal's initialization cost, and finally we investigate the benefits of using a DMA engine for array copy.

Table 2 shows the machines used for our evaluation. Our results on both, 8x8 AMD Opteron and 4x8x2 Intel Xeon, are similar. For brevity, we focus on results from our 8x8 AMD Opteron unless stated otherwise. We use two workloads: Green-Marl and PARSEC Streamcluster.

Green-Marl. The Green-Marl compiler comes with a variety of programs of which we have selected three graph algorithms to demonstrate the performance characteristics of Shoal: (*i*) *PageRank* [30] iteratively calculates the importance of each node in the graph as a sum of the rank of all incoming neighbors divided by the number of outgoing edges they have, (*ii*) *hop-distance* calculates the distance of every node from the root using Bellman-Ford, and (*iii*) *triangle-counting* that counts the number of triangles in the input graph. This is implemented as a triple loop: for all nodes in the graph, it looks at all combinations of nodes reachable from it and checks if there is an edge connecting them.

For our evaluation we used two graphs: (*i*) the Twitter graph [23] having 41M nodes and 1468M edges. The total working set size is 2.459 GB with Green-Marl configured to 64 bit node and edge types (excluding unused arrays). We were not able to run triangle-counting on the Twitter graph on our system, hence we were falling back on the LiveJournal graph [5] for that workload. LiveJournal has 4M nodes and 69M edges with a total working set of 392 MB in Green-Marl.

machine	8x8 AMD Opteron	4x8x2 Intel Xeon	2x10 Intel Xeon
CPU	AMD Opteron 6378	Intel Xeon E5-4640	Intel Xeon E5-2670 v2
micro architecture	Piledriver	Sandy Bridge	Ivy Bridge
#nodes/#sockets	8/4 @ 2.4 GHz	4/4 @ 2.4 GHz	2/2 @ 2.5 GHz
L1 data size	16K /thread	32K /core	32K /core
L2/L3 size	2048K / 6144K	256K / 20480K	256K / 25600K
memory	512 GB (64 GB per node)	512 GB (128 GB per node)	256 GB (128 GB per node)

Table 2: Machines used for evaluation (L2 shared by core, L3 shared by socket)

PARSEC – Streamcluster. Streamcluster [9] solves the online clustering problem. Input data is given as an array of multi-dimensional points. We manually modified it to use Shoal for memory allocation and accesses.

5.1 Scalability

In highly parallel workloads, scalability is one of the key concerns. In this section we show the benefits of using Shoal over unmodified versions of the workloads and that allocating memory based on access patterns, if available, is favorable over online methods.

Green-Marl. We evaluated scalability of three Green-Marl workloads on an 8x8 AMD Opteron comparing Shoal (—■—) against the original Green-Marl implementation (—●—) and Carrefour [13] (—⋆—). Figure 4 shows that Shoal clearly outruns the original implementation by almost 2x and also performs better than the online method as a result of an optimized memory placement. Furthermore, our results show that an online method can harm the performance in case pages are getting migrated back and forth (hop-distance). Overall, except for triangle-counting, all implementations scale well. Note that we do not include the graph loading time and Shoal initialization.

We also executed the same measurements on Barrelfish (—♦—) to show Shoal's portability. Our intention is not to show that either operating system is faster than the other, but rather their comparability. On Barrelfish, only static OpenMP schedules are supported due to implementation limitations. This negatively impacts the performance for triangle-counting. However, Shoal still performs better than the original implementation, which uses dynamic OpenMP schedules.

PARSEC – Streamcluster. In contrast to Green-Marl, Streamcluster is implemented in C and hence there is no automatic method of extracting access patterns. We modified Streamcluster to use Shoal's array abstraction to demonstrate that using Shoal directly by programmers can improve scalability with little efforts for manual annotation. To make Streamcluster work with Shoal, we had to (*i*) abstract access to arrays using Shoal's get and set methods, (*ii*) initialize each thread using `shl__thread_init()` and change the array allocation to use `shl__malloc_array()` instead of `malloc()`.

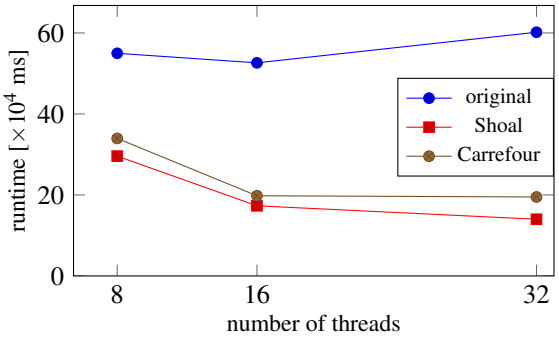

Figure 5: Scalability of PARSEC streamcluster on 8x8 AMD Opteron

Since Streamcluster is a streaming application, arrays for input coordinates are reused for each chunk of new streaming data, but are otherwise read-only. We use (*iii*) `shl__repl_sync()` to synchronize the master copy of the array to it's replicas once after a new chunk has been read. We compare the original Streamcluster implementation with Carrefour and Shoal (Figure 5). Our results confirm the bad scalability of Streamcluster due to the use of `memset()` after allocating arrays [13, 17], which causes all memory to be allocated on a single NUMA node. This leads to congestion of the interconnect and memory controllers of that node. Shoal achieves an 4x improvement over the original implementation. Shoal's annotated access allocation function outperforms Carrefour's online method. We want to emphasize here, that we replaced only one of the used arrays with a Shoal array (large pages and replication) and did not apply further optimizations.

5.2 Comparison of array implementations

We conducted a detailed analysis of Shoal's different array implementations using all physical cores of our machines. In this section we show, that Shoal achieves better performance than the original Green-Marl implementation regardless of which array configuration we use. Figure 6 shows our results normalized to the original Green-Marl implementation and Figure 8 shows the breakdown into initialization and computation times. The measurements were executed on the 8x8 AMD Opteron using all 32 physical cores. Following, we give

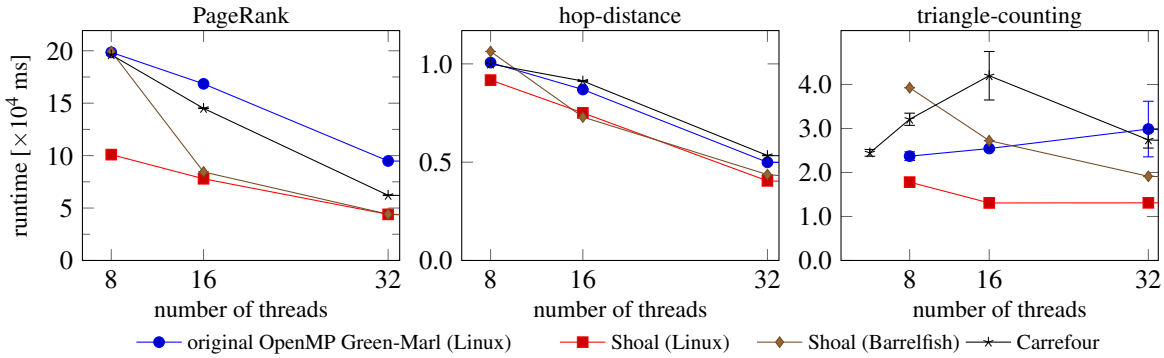

Figure 4: Scalability on 8x8 AMD Opteron. Workload: Twitter (LiveJournal for triangle-counting). For Shoal, we chose the best configuration each. We omit results for 64 threads: using SMT threads has similar performance to using only the 32 physical cores.

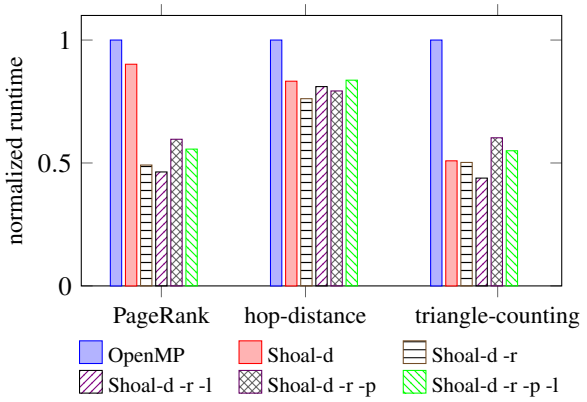

Figure 6: Comparison of various combinations of array implementations on 8x8 AMD Opteron: distribution (-d), replication (-r), partitioning (-p), large page (-l) (runtime normalized to stock-Green-Marl)

explanations for each configuration and relate them to our performance counter observations (Figure 7).

Distribution ( ). The original Green-Marl implementation already initializes memory for storing the graphs with a OpenMP loop to distribute memory in the machine. However, this is not done for dynamically allocated arrays (e.g. the `rank_next` in PageRank). With Shoal, *all* arrays are ensured to be distributed among the nodes, resulting in a more even distribution of memory and a better performance across all workloads. Initialization of distributed arrays relies on an OpenMP loop to allocate memory evenly across all NUMA nodes. Initializing memory is hence executed in parallel, which results in small initialization cost compared to other array types. We show this in Figure 8.

Our claims are supported by measurements of each memory controller's read- and write throughput. (Figure 7): compared to the original implementation (*i*) where all reads and writes are executed on socket 0, enabling distribution (*ii*) results in an evenly distributed load on all memory controllers. However, in both cases, the memory controllers are not saturated. Memory throughput suffers from the lower bandwidth of the interconnect links, i.e. 9.6GB/s for QPI. With randomly distribution of memory, only 1/4 of all memory accesses are expected to be local.

Distribution + replication ( ). In contrast to distribution, replication is applied only to read-only data. In our workloads, the graph itself is not altered by the program and hence replicated among the nodes. This results in a increased fraction of locally served memory accesses and lower interconnect traffic: memory accesses are evenly distributed among all memory controllers as shown in (*iv*) of Figure 7. Note, enabling replication without distribution allocates non-read-only arrays into single-node arrays resulting in an unbalanced memory access for that part of the working set, see (*iii*) of Figure 7. Initialization cost for replicated arrays are higher than distributed arrays because more memory needs to be allocated. We force correct allocation by touching each replica on its designated node. Finally, copying the master array to the other replicas causes some additional overhead when initializing such an array (Figure 8).

Partitioning (and). Replication of data increases the memory footprint of the application. Partitioning tries to preserve the locality of replication without increasing the memory footprint. Our current implementation requires a static OpenMP schedule for partitioning to ensure scheduling of work units to the right partitions. However, static schedules potentially lead to imbalance of work among the execution units as workloads may be skewed (e.g. in triangle-counting). Eventhough the same amount of memory has to be allocated as with distributed arrays, its initialization is more complex: using Linux' first touch policy, Shoal ensures memory is touched on the correct node by migrating a thread to where memory should be allocated and touching each page from there. This results in similar initialization time as with replication, but slightly less time to copy the data.

Large Pages (and). Modern CPUs support various page sizes and have a distinct TLB for each page size. A miss in the TLB enforces the CPU to do a full page table walk which drastically increases the access time. Shoal supports large pages for its arrays. Enabling large pages for PageRank and triangle-counting results in a slightly better performance, while hop-distance runtime increases slightly. Gaud *et al.* [17] concluded similar findings in their experiments with large pages. Enabling large pages reduces the total number of pages used and therefore the number of required first touches in the allocation process. This results in a decrease of the allocation time (Figure 8).

We conclude that despite the additional overhead of allocation and initialization, the total runtime with Shoal is still reduced. However, we want to emphasize here, that we do not consider initialization time as a main target of optimization as typically time spent for computation dominates the program execution. Nevertheless, allocation could be improved by (*i*) maintaining a cache of pre-allocated pages on each node, (*ii*) applying a smarter page mapping strategy or (*iii*) by initializing Shoal while input data (e.g. the graph) is loaded.

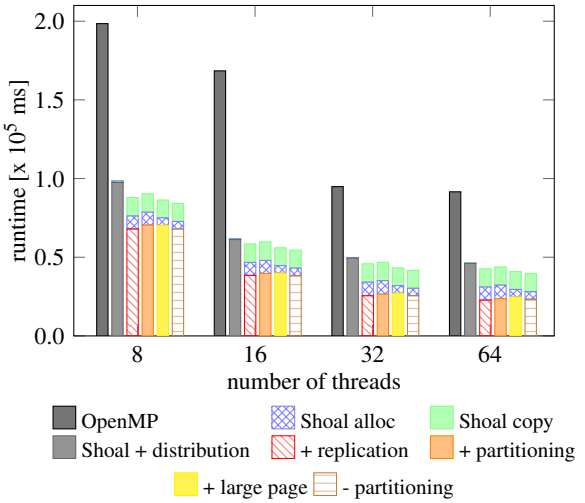

Figure 8: Shoal initialization and runtime on 8x8 AMD Opteron for various array configurations using PageRank with Twitter workload

5.3 Use of DMA engines

Modern CPUs have integrated DMA engines, which provide a rich set of memory operations. For instance, recent Intel server CPUs provide integrated CrystalBeach 3 DMA engines [22]. We evaluate the use of DMA engines for initialization and copy operations on a 2x10 Intel Xeon (our 8x8 AMD Opteron and 4x8x2 Intel Xeon do not have DMA engines). We run these experiments on Barrelfish, as user-level support for DMA engines is already integrated and requires no additional setup.

We now compare the raw copy performance of DMA controllers to CPU `memcpy()` and further evaluate how DMA engines can be used in Shoal. Asynchronous memory operations offered by DMA controllers can free up the CPU of the burden of copying data around and provide cycles to do actual work. Shoal offers an interface to start an asynchronous memory copy and to check for completion of the operation.

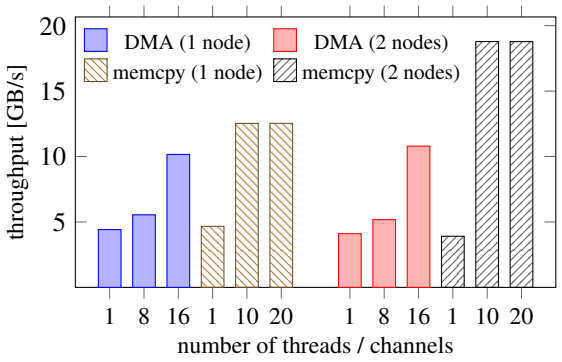

Figure 9: Comparison of parallel OpenMP copy and DMA copy on 2x10 Intel Xeon for large buffers ($\gg$ cache size)

Comparing raw memory throughput. Our results of a raw throughput evaluation (Figure 9) show, that the use of DMA engines does not necessarily improve the sequential performance, especially for blocking copy operations as used by PageRank. However, to outperform the DMA controller, all threads of the CPU have to be used for memory copying and hence no other computational task can be executed in the meantime.

We now show the DMA engines are useful if only a few threads are available for synchronously copying arrays or asynchronous copies. For example, we are planing to evaluate the use of DMA engines to propagate writes to replicas in the background.

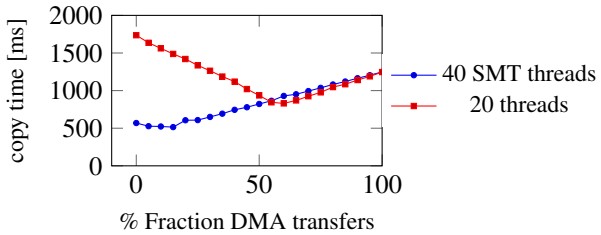

Figure 10: Initialization cost for copying data into Shoal arrays using the Twitter working set with replication on Barrelfish

DMA engines for initialization. We benchmark the initialization phase of PageRank where data is copied from the graph's memory into the Shoal arrays. We copy a certain ratio of the array using DMA engines asynchronously while using parallel OpenMP loops to copy the remaining elements. Figure 10 shows how varying

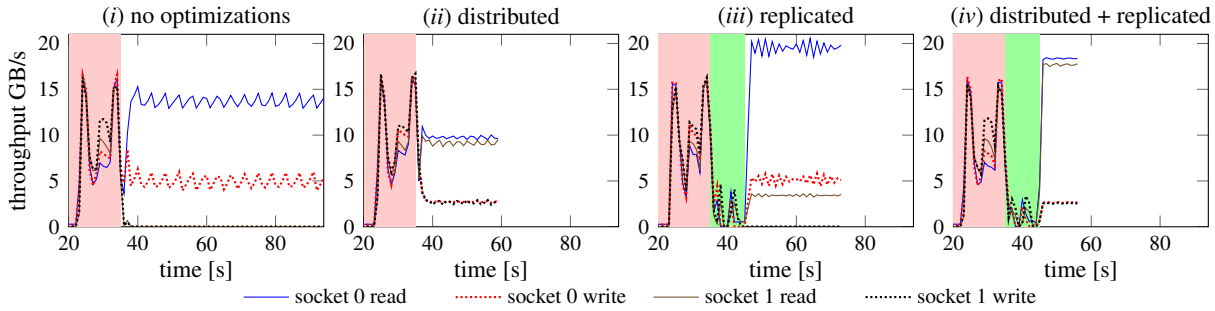

Figure 7: Memory throughput for sockets 0 and 1 on 4x8x2 Intel Xeon. In the first 35 seconds, the graph is loaded from disk. For replication, we show the replica initialization cost in green. Note: Sockets 2 and 3 are equal to socket 1 and left out for readability.

the ratio of how much of the array is copied using DMA engines vs. parallel OpenMP loops affects performance. For the parallel copy, we show the result for using all 20 physical threads and 40 SMT-threads respectively. First, we see a big difference if we enable SMT (—●—): as expected, memory access latency is hidden and the use of a DMA engine improves performance only slightly (about 10%). This is presumably because these 40 hyper-threads are sufficient to saturate all memory controllers on that machine. With SMT disabled (—■—), the memory latency cannot be hidden. Our results show clearly, that using DMA engines and CPU copy simultaneously reduces the time for copying arrays by 2x.

DMA engines for array copy. In our PageRank workload, the ranks are copied between two arrays in every iteration. With our implementation, we can use DMA engines to improve the copy time in that case too. However, our measurements show, that depending on the array configuration only 1-5% of the entire runtime is spent copying and hence optimize that part does not have a notable effect on PageRank's total runtime, hop-distance behaves similarly. However, workloads allowing asynchronously copy of data would be a more obvious candidate for optimizations based on DMA engines.

To sum up, the effect of using DMA controllers for memory operations highly depends on whether the program has to share the resources with other workloads or not. If all resources are available, DMA engines provide about 10% improvement. On the otherhand if resources are shared with other users, DMA engines provide up to 2x improvement in our case.

5.4 Writeable replication

Finally, we look at the issue of write-shared arrays.

Efficiently maintaining consistency of replicated data is difficult; updates must be propagated to all replicas. This can be achieved by issuing writes to all replicas or by applying techniques such as double-buffering and asynchronous copies. Both relax consistency guarantees, but are strong enough for use with OpenMP loops, where concurrent writes and reads in the same loop iteration would cause non-determinism.

In this section, we show that replicating non-read-only data does not deliver much benefit on current NUMA machines for already otherwise optimized workloads: the additional cost of house-keeping (e.g. maintaining write-sets) and propagating updates to all replicas outweighs the potential performance gain of replication.

We compare writeable replicas with single-node allocation and distribution (Table 3). Our results show that the cost of maintaining consistency grows with the number of replicas. Furthermore, replication not necessarily achieves better performance compared to distributed arrays as the load on the interconnect in the latter case is already relatively low.

We believe that writeable replication will be useful (and needed) in heterogeneous systems, where memory non-uniformity is more drastic (e.g. more NUMA nodes, slower links). In that case, replication of data in local memory is crucial for performance even in the presence of updates. Writeable replication could also have an application for more complex workloads (e.g. a smaller fraction of read-only data), where the simple mechanisms we presented in this paper cannot be applied.

dist configuration	cost	stderr	notes
single-node	214.0	11.0	
distributed	203.0	0.9	
wr-rep, 2 reps	248.8	7.0	nodes: 0,n-1
wr-rep, 4 reps	333.6	5.9	nodes: 0,n-1
wr-rep w/o copy op	202.9	7.2	

Table 3: Writeable replicas on 8x8 AMD Opteron. Workload: hop-distance with -d -r -h configuration

6 Related work

Our work was originally inspired by recent research in domain specific languages. Such languages are based on the observation that it is hard to write efficient code for a wide-range of different systems, as algorithms need to be tuned to a concrete machine in order to achieve good performance. DLSs express algorithms in a rich

and intuitive way. The Green-Marl [20] graph analytics DSL, OptiML [37], a machine learning DSL , as well as SPL [41], a signal processing language, all provide a rich set of powerful high-level operators. They use a compiler that generates code that is highly tuned to the target machine and makes heavy use of data parallelism. While all of these languages encode memory access patterns in their high-level languages, none uses them to adapt memory allocation at runtime.

Modern machines are becoming inherently complex: Baumann *et al.* argued that computers are already a distributed system in their own right [8]. They proposed a multikernel approach [7] which avoids sharing of state among OS nodes by replication and applying techniques from distributed systems. Similarly, Wentzlaff *et al.* [40] apply partitioning to OS services. Techniques from distributed systems are beneficial not only on an OS level, but also for applications. Multimed [31] replicates database instances within a single multi-core machine. Salomie *et al.* showed that congestion of memory controllers and interconnects impact the overall performance. Carrefour [13] attempts to reduce the contention on interconnect and memory controllers by online monitoring of memory accesses and auto-tuning NUMA-aware memory allocation. This approach can be applied to any application without modifications to the program code, but is less fine-grained. While Shoal derives a program's semantics from annotations or DSL compiler analysis, the former these approaches need to guess programmers' intentions in retrospect.

Systems such as SGI's Origin 2000 [35] use page-level migration and replication of data. Hardware monitors detect the access patterns to pages (e.g., which processors tend to access the page, and whether these are reads or writes). Based on the gathered data, pages are replicated or migrated towards a frequently accessing CPU.

With highly parallel workloads, efficient synchronization is crucial for application performance [14]. Lock cohorting [15] implements NUMA-aware locks by taking cache hierarchy and NUMA-topology into account. Shoal's treatment of memory is analogous to these systems' treatment of locks.

Access to large and huge MMU pages is provided by services and libraries like `libhugetlbfs` [4]. The latter, however, requires static setup of a large page pool, among other issues.

Cache coherence protocols like MOESI [2] allow cache-lines to be in a shared state which is a form of hardware-level replication. This is only effective with workloads having good locality and small working set, which is not the case for our graph workloads. Research systems, such as Stanford FLASH, have provided software control over this form of replication [36].

The Solaris operating system provides a `madvise` [28] operation to let an application give hints on future accesses to a memory region which results in a distributed or local allocation to the calling thread. Shoal extends this approach with a wider range of possible usage patterns, and infers appropriate settings to use.

Li [24] attempts to find the best algorithm for a specific task depending on machine characteristics and workload based on empirical search. In contrast, we decide a priori based on additional information extracted from high-level languages or given by manual annotations. Franchetti *et al.* [16] automatically tune FFT programs to multi-core machines. They argue that programming such machines is increasingly complicated, which increases the burden for programmers and makes a case for automatic tuning. Atune-IL [32] auto-tunes applications, including the number of threads etc. It explores all possible parameters, but tries to reduce the search space.

However, tuning data placement and parallelism individually is not optimal, because data and threads may not end up on the same node. Hence, affinity of threads and data need to be enforced in order to improve the performance of OpenMP programs [38].

Finally, PGAS languages such as UPC [39], co-array Fortran [27], X10 [12], Chapel [11], and Fortress [29] provide an abstraction of shared arrays which can be implemented across a distributed system. Code iterating over an array can execute on the node holding the portion of the array being accessed.

In high-performance computing, array abstractions [26] have been used to simplify programming while still providing good performance and scalability. They support high-level operations on arrays e.g. matrix multiplications or atomic operators.

7 Conclusion

In this paper, we presented Shoal, a library that provides an array abstraction and rich memory allocation functions that allow automatic tuning of data placement and access depending on workload and machine characteristics. Tuning is based on memory access patterns. These are either (*i*) given by manual annotation, or, ideally, (*ii*) by modifying compilers of high-level languages to extract that information automatically. We have shown that we can use this additional information to automatically choose array implementations that increase performance on today's NUMA systems. We report an up 2x improvement for Green-Marl, a high-level graph analytics workload, without changing the Green-Marl input program. We found our memory abstraction as well as the simple policy for selecting the array implementation sufficient for current workloads and machines, but believe that future machines can benefit from a more fine-grained selection of array implementations.

References

[1] ACHERMANN, R. *Message Passing and Bulk Transport on Heterogenous Multiprocessors*. ETH Zurich, 2014. Master's Thesis, http://dx.doi.org/10.3929/ethz-a-010262232.

[2] ADVANCED MICRO DEVICES. AMD64 Architecture Programmer's Manual Volume 2: System Programming. Online, 2013. http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/24593_APM_v21.pdf. Publication Number 24593. Revision 3.23.

[3] APPAVOO, J., SILVA, D. D., KRIEGER, O., AUSLANDER, M., OSTROWSKI, M., ROSENBURG, B., WATERLAND, A., WISNIEWSKI, R. W., XENIDIS, J., STUMM, M., AND SOARES, L. Experience Distributing Objects in an SMMP OS. *ACM Transactions on Computer Systems 25*, 3 (Aug. 2007).

[4] ARAVAMUDAN, N., LITKE, A., AND MUNSON, E. libhugetlbfs. Online, 2015. http://libhugetlbfs.sourceforge.net/.

[5] BACKSTROM, L., HUTTENLOCHER, D., KLEINBERG, J., AND LAN, X. Group Formation in Large Social Networks: Membership, Growth, and Evolution. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, NY, USA, 2006), KDD '06, ACM, pp. 44–54.

[6] BARRELFISH PROJECT. The Barrelfish Operating System. Online, 2015. www.barrelfish.org.

[7] BAUMANN, A., BARHAM, P., DAGAND, P.-E., HARRIS, T., ISAACS, R., PETER, S., ROSCOE, T., SCHÜPBACH, A., AND SINGHANIA, A. The Multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (New York, NY, USA, 2009), SOSP '09, ACM, pp. 29–44.

[8] BAUMANN, A., PETER, S., SCHÜPBACH, A., SINGHANIA, A., ROSCOE, T., BARHAM, P., AND ISAACS, R. Your computer is already a distributed system. Why isn't your OS? In *Proceedings of the 12th Conference on Hot Topics in Operating Systems* (Berkeley, CA, USA, 2009), HotOS'09, USENIX Association, pp. 12–12.

[9] BIENIA, C., KUMAR, S., SINGH, J. P., AND LI, K. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques* (New York, NY, USA, 2008), PACT '08, ACM, pp. 72–81.

[10] BOYD-WICKIZER, S., CHEN, H., CHEN, R., MAO, Y., KAASHOEK, F., MORRIS, R., PESTEREV, A., STEIN, L., WU, M., DAI, Y., ZHANG, Y., AND ZHANG, Z. Corey: An Operating System for Many Cores. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2008), OSDI'08, USENIX Association, pp. 43–57.

[11] CHAMBERLAIN, B., CALLAHAN, D., AND ZIMA, H. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications 21*, 3 (Aug. 2007), 291–312.

[12] CHARLES, P., GROTHOFF, C., SARASWAT, V., DONAWA, C., KIELSTRA, A., EBCIOGLU, K., VON PRAUN, C., AND SARKAR, V. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (New York, NY, USA, 2005), OOPSLA '05, ACM, pp. 519–538.

[13] DASHTI, M., FEDOROVA, A., FUNSTON, J., GAUD, F., LACHAIZE, R., LEPERS, B., QUEMA, V., AND ROTH, M. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2013), ASPLOS '13, ACM, pp. 381–394.

[14] DAVID, T., GUERRAOUI, R., AND TRIGONAKIS, V. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 33–48.

[15] DICE, D., MARATHE, V. J., AND SHAVIT, N. Lock Cohorting: A General Technique for Designing NUMA Locks. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2012), PPoPP '12, ACM, pp. 247–256.

[16] FRANCHETTI, F., VORONENKO, Y., AND PÜSCHEL, M. FFT Program Generation for Shared Memory: SMP and Multicore. In *Proceedings of the 2006 ACM/IEEE Conference on*

Supercomputing (New York, NY, USA, 2006), SC '06, ACM.

[17] GAUD, F., LEPERS, B., DECOUCHANT, J., FUNSTON, J., FEDOROVA, A., AND QUEMA, V. Large Pages May Be Harmful on NUMA Systems. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)* (2014), USENIX Association.

[18] GICEVA, J., SALOMIE, T.-I., SCHÜPBACH, A., ALONSO, G., AND ROSCOE, T. COD: Database / Operating System Co-Design. In *CIDR* (2013), www.cidrdb.org.

[19] HAND, S. M. Self-paging in the nemesis operating system. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation* (1999), pp. 73 – 86.

[20] HONG, S., CHAFI, H., SEDLAR, E., AND OLUKOTUN, K. Green-Marl: A DSL for Easy and Efficient Graph Analysis. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2012), ASPLOS XVII, ACM, pp. 349–362.

[21] HOWARD, J., DIGHE, S., VANGAL, S. R., RUHL, G., BORKAR, N., JAIN, S., ERRAGUNTLA, V., KONOW, M., RIEPEN, M., GRIES, M., ET AL. A 48-Core IA-32 Processor in 45 nm CMOS Using On-Die Message-Passing and DVFS for Performance and Power Scaling. *IEEE Journal of Solid-State Circuits 46*, 1 (2011), 173–183.

[22] INTEL CORPORATION. Intel Xeon Processor E5-1600/E5-2600/E5-4600 v2 Product Families, Datasheet - Volume One of Two. Online, 2014. `http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-e5-v2-datasheet-vol-1.pdf`, Document Number: 329187-003.

[23] KWAK, H., LEE, C., PARK, H., AND MOON, S. What is Twitter, a Social Network or a News Media? In *WWW '10: Proceedings of the 19th international conference on World wide web* (New York, NY, USA, 2010), ACM, pp. 591–600.

[24] LI, X., GARZARÁN, M. J., AND PADUA, D. A Dynamically Tuned Sorting Library. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization* (Washington, DC, USA, 2004), CGO '04, IEEE Computer Society, p. 111.

[25] MATTSON, T. G., VAN DER WIJNGAART, R., AND FRUMKIN, M. Programming the Intel 80-core network-on-a-chip Terascale Processor. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing* (Piscataway, NJ, USA, 2008), SC '08, IEEE Press, pp. 38:1–38:11.

[26] NIEPLOCHA, J., HARRISON, R. J., AND LITTLEFIELD, R. J. Global Arrays: A Nonuniform Memory Access Programming Model for High-Performance Computers. *The Journal of Supercomputing 10*, 2 (June 1996), 169–189.

[27] NUMRICH, R. W., AND REID, J. Co-Array Fortran for Parallel Programming. *SIGPLAN Fortran Forum 17*, 2 (Aug. 1998), 1–31.

[28] ORACLE CORPORATION. `madvise()` in Solaris 10. Online, 2015. `http://docs.oracle.com/cd/E19253-01/817-0547/whatsnew-updates-72/index.html`.

[29] ORACLE LABS. Fortress. Online, 2015. `https://projectfortress.java.net`.

[30] PAGE, L., BRIN, S., MOTWANI, R., AND WINOGRAD, T. The PageRank Citation Ranking: Bringing Order to the Web. Technical Report 1999-66, Stanford InfoLab, November 1999.

[31] SALOMIE, T.-I., SUBASU, I. E., GICEVA, J., AND ALONSO, G. Database Engines on Multicores, Why Parallelize When You Can Distribute? In *Proceedings of the 6th Conference on Computer Systems* (New York, NY, USA, 2011), EuroSys '11, ACM, pp. 17–30.

[32] SCHAEFER, C. A., PANKRATIUS, V., AND TICHY, W. F. Atune-IL: An Instrumentation Language for Auto-Tuning Parallel applications. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing* (Berlin, Heidelberg, 2009), Euro-Par '09, Springer-Verlag, pp. 9–20.

[33] SCHÜPBACH, A., PETER, S., BAUMANN, A., ROSCOE, T., BARHAM, P., HARRIS, T., AND ISAACS, R. Embracing diversity in the Barrelfish manycore operating system. In *Proceedings of the Workshop on Managed Many-Core Systems* (2008).

[34] SILICON GRAPHICS INTERNATIONAL CORPORATION. libnuma. Online, 2015. `http://oss.sgi.com/projects/libnuma/`.

[35] SILICON GRAPHICS INTERNATIONAL CORPORATION. Origin and Onyx2 Theory of Operations Manual. Online, 2015. `http://techpubs.sgi.com/library/tpl/`

cgi-bin/getdoc.cgi?coll=0650&db=bks&
srch=&fname=/SGI_Developer/OrOn2_
Theops/sgi_html/ch02.html.

[36] SOUNDARARAJAN, V., HEINRICH, M., VERGHESE, B., GHARACHORLOO, K., GUPTA, A., AND HENNESSY, J. Flexible Use of Memory for Replication/Migration in Cache-Coherent DSM Multiprocessors. In *Proceedings of the 25th Annual International Symposium on Computer Architecture* (Washington, DC, USA, 1998), ISCA '98, IEEE Computer Society, pp. 342–355.

[37] SUJEETH, A., LEE, H., BROWN, K., ROMPF, T., CHAFI, H., WU, M., ATREYA, A., ODERSKY, M., AND OLUKOTUN, K. OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)* (2011), pp. 609–616.

[38] TERBOVEN, C., AN MEY, D., SCHMIDL, D., JIN, H., AND REICHSTEIN, T. Data and Thread Affinity in OpenMP Programs. In *Proceedings of the 2008 Workshop on Memory Access on Future Processors: A Solved Problem?* (New York, NY, USA, 2008), MAW '08, ACM, pp. 377–384.

[39] UPC CONSORTIUM. *UPC Language and Library Specifications*, November 2013. Version 1.3. Online. http://upc.lbl.gov/publications/upc-spec-1.3.pdf.

[40] WENTZLAFF, D., AND AGARWAL, A. Factored Operating Systems (fos): The Case for a Scalable Operating System for Multicores. *SIGOPS Operating Systems Review 43*, 2 (Apr. 2009), 76–85.

[41] XIONG, J., JOHNSON, J., JOHNSON, R., AND PADUA, D. SPL: A Language and Compiler for DSP Algorithms. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation* (New York, NY, USA, 2001), PLDI '01, ACM, pp. 298–308.