Portable Performance from Workstation to Supercomputer: Distributing Data Structures with Qthreads

Kyle B. Wheeler[1,2], Douglas Thain[2], and Richard C. Murphy[1]

[1]Sandia National Laboratories*
Albuquerque, New Mexico, USA
{kbwheel,rcmurph}@sandia.gov

[2]University of Notre Dame
Computer Science and Engineering
Notre Dame, Indiana, USA
{kwheeler,dthain}@cse.nd.edu

Abstract

Locality and data layout are critical to the performance of threaded parallel programs, but standard threading interfaces incorrectly presume that data is equally accessible from all threads. The qthread library's locality framework has been expanded to provide convenient CPU affinity. This locality support enables development of three locality aware distributed data structures: a memory pool, an array, and a queue. This paper presents benchmark results illustrating the performance characteristics of these data structures on an Altix ccNUMA SMP system, a highly multithreaded Niagara-based server, and a conventional SMP workstation. The combination of locality and threading interface with adaptive distributed data structures provides scalable performance on multiple parallel architectures.

1. Introduction

Data placement is one of the most critical challenges in multicore application performance. Increasingly, multiprocessor machines have non-uniform memory access (NUMA) latencies. Unfortunately, parallel machines have a wide variety of structural differences that impose a large performance penalty on non-optimal data layout [5, 22], so data locality must be exposed for performance.

Standard threading interfaces — such as pthreads [11], OpenMP [20], and Intel Threading Building Blocks (TBB) [12] — are designed for commodity multiprocessing and multicore systems. They provide powerful and convenient tools for developing software tailored to such systems. While these interfaces make it easy to create parallel tasks, they assume that all data is equally accessible from all threads. Because this assumption is generally false, these interfaces do not maintain data and thread locality or provide tools to discover and exploit system topology. Some operating systems provide mechanisms to discover machine topology and to specify thread and memory block locality. Such mechanisms are system-specific and generally non-portable.

Our solution to this problem is the qthread library [23], which integrates locality with the threading interface. The qthread library is a cross-platform threading library that provides lightweight threads, an integrated locality framework, and lightweight synchronization in a way that maps well to developing multithreaded architectures. It supports many different operating systems and hardware architectures and provides a consistent interface to their unique methods of discovering and specifying locality. We expanded its locality framework and added distributed data structures that adapt to NUMA environments. The new distributed data structures hide the complexity of data locality. This paper presents the design of a distributed memory pool, a distributed array, and two forms of distributed queue.

Benchmark results, gathered from several parallel architectures, demonstrate the scalability of these data structures. By comparing operations-per-second and effective bandwidth, the benchmarks show the importance of choosing appropriate system-specific design parameters, such as memory distribution pattern and segment size.

The remainder of this paper is organized as follows. A survey of related work is presented in Section 2. Section 3 summarizes the qthread library and the locality features it provides. Section 4 describes the three architectures used to benchmark the distributed data structures. The design of the data structures, along with benchmark results demonstrating their efficiency, is presented in Section 5. Section 6 suggests future research.

[*] Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Thread Operations

qthread_init(*num_sheps***)**
initialize the library

qthread_finalize()
clean up, shut down the library

qthread_fork(*func*, *arg*, *ret***)**
spawn a thread

qthread_fork_to(*f*, *a*, *r*, *shep***)**
spawn a thread to the specified shepherd

qthread_migrate_to(*shep***)**
move the calling thread to the specified shepherd

qthread_distance(*src*, *dest***)**
returns the distance between two shepherd IDs

Memory Pools

qpool_create(*item_size***)**
create a pool of objects of the specified size

qpool_create_aligned(*i_s*, *align***)**
similar to qpool_create(), but returns aligned objects

qpool_alloc(*pool***)**
get an object from the pool

qpool_free(*pool*, *addr***)**
return an object to the pool

qpool_destroy(*pool***)**
deallocates all pool memory

Array Operations

qarray_create(*count*, *unit_size***)**
allocate an array, distribute its memory

qarray_create_tight(*c*, *u_s***)**
same as qarray_create(), but guarantees item size will not change

qarray_elem(*array*, *index***)**
returns a pointer to the specified element in the array

qarray_iter_loop(*array*, *func*, *arg***)**
iterate over the array elements in parallel

qarray_destroy(*array***)**
deallocate the array

Queue Operations

{qd|qlf}queue_create()
allocate a queue

{qd|qlf}queue_enqueue(*q*, *elem***)**
append an element to the the queue

{qd|qlf}queue_dequeue(*queue***)**
get the head off the queue

{qd|qlf}queue_empty(*queue***)**
check if the queue contains any elements

{qd|qlf}queue_destroy(*queue***)**
deallocate a queue

A qlfqueue is a lock-free queue, and a qdqueue is a distributed queue.

Figure 1. Qthread API (abridged)

2. Related work

This research draws from three categories of prior work: lightweight threading models, data structures, and topology interfaces. Cilk [4] and OpenMP [20] are examples of convenient and lightweight threading models. However, they ignore locality, forcing the programmer to rely on the scheduler to keep each thread near the data it is manipulating. Unfortunately, the work-stealing scheduling algorithms these threading models use assume that all tasks can be performed equally well on any processor, which is an invalid assumption on NUMA machines with a high latency variability.

Concurrent vectors, hashes, and queues, such as provided by Intel's Threading Building Blocks [12] are good examples of data structures designed around parallel management operations rather than data operations, ignoring locality. Co-array Fortran [18] provides a parallel container — the co-array — that integrates locality with the execution model, but uses loader-defined static distribution. A distributed queue is a special case of a concurrent pool [15], which is designed for parallel efficiency. Our distributed queue uses locality information to combine Johnson's stochastic queue [13], the push-based queue by Arpaci-Dusseau et. al. [3], and a high-speed lock-free queue based on the work of Michael and Scott [16].

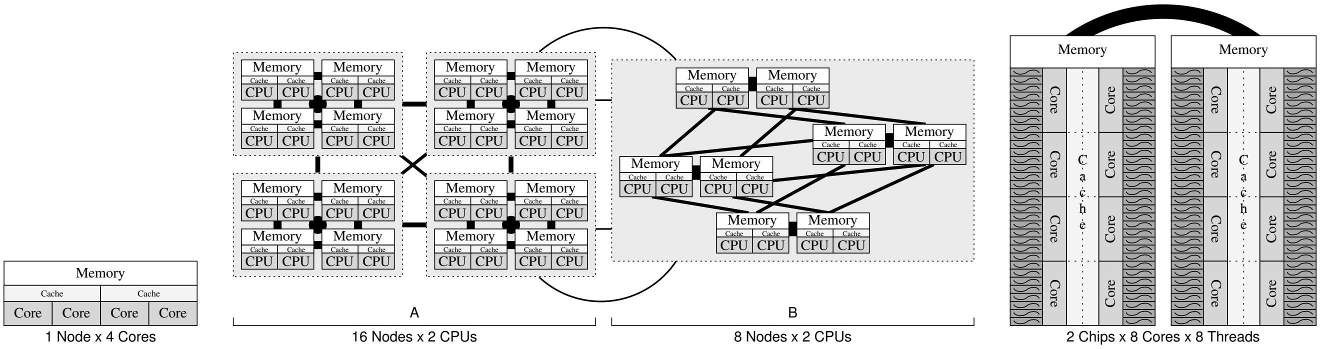
Counterintuitively, threading interfaces tend not to address memory topology. This is probably a result of the historical low variance in memory access latency [17]. Modern and future large-scale shared-memory machines have larger latency variances, and the impact is magnified by increasing CPU speeds. Topology interfaces tend to be operating-system specific. Linux systems largely rely on the libnuma [14] library to exploit system topology. This library presents computers as a set of numbered nodes, CPUs, and "physical" CPUs that overlap. The distance to a node's own memory is normalized to ten, and other distances are expressed on that scale. Without libnuma, Linux processes can define their CPU affinity using the sched_setaffinity () interface. Unfortunately, this interface changes across kernel versions. Software that must work reliably across multiple Linux systems must either detect the available interface variety or use the Portable Linux Processor Affinity (PLPA) library [19], which provides a stable interface. Solaris systems provide the liblgrp library [21], which presents a hierarchical description of topology. Each locality group (lgrp) contains a set of CPUs and/or additional lgrps and is associated with a block of memory. Recent versions of the library can report the latency between lgrps in machine-specific units. All of these interfaces enable thread CPU affinity and libnuma and liblgrp enable memory CPU affinity. Because these interfaces manipulate the operating system's scheduler and memory subsystem, they can only affect things that the operating system schedules, which are inherently heavyweight.

Programming languages such as Chapel [8], X10 [6], Fortress [2], and UPC [9], provide explicit locality as a function of the programming environment. These languages provide expressive semantics and can be effective in exploiting available hardware resources. Unfortunately, they are incompatible with existing software.

3. The qthread library

The qthread library [23] is a cross-platform lightweight threading library with an integrated locality framework and lightweight synchronization that bridges the gap between commodity systems and developing multithreaded architectures. A "qthread" is a nearly-anonymous thread with a small stack that lacks the expensive guarantees and features of heavyweight threads, such as per-thread process identifiers (PIDs), signal vectors, and the ability to be canceled. This lightweight nature supports large numbers of threads with fast context-switching, enabling high-performance fine-grained thread-level parallelism. The API of the qthread library is summarized in Figure 1. The locality framework has been expanded to support thread and data affinity in parallel systems.

Virtually all computer architectures provide atomic op-



(a) Xeon **(b) Altix** **(c) Niagara**

Figure 2. System Topologies

erations, yet they are not directly accessible in most programming interfaces; OpenMP is a notable exception. The qthread library provides an atomic increment via qthread_incr(), and atomic compare-and-swap with qthread_cas(). Data-based blocking synchronization is a powerful way to express task dependencies. The qthread library provides both full/empty bit (FEB) and mutex-like blocking operations. FEB operations are atomic read/writes that depend on per-word full/empty status. Some systems, such as the Cray MTA/XMT, provide hardware support for FEB operations, but usually they must be emulated.

The qthread library uses cooperative-multitasking: blocking operations trigger context switches. These context switches occur in user space via function calls and are thus cheaper than signal-based context switches. This approach allows threads to hide communication latencies and avoid context-switch overhead until unavailable data is needed.

Each qthread has a location, or "shepherd." Shepherds define immovable regions within the system. Qthreads can be spawned directly to a specific shepherd with qthread_fork_to(), and migrate between shepherds with qthread_migrate_to(). Explicit migration guarantees that threads cannot execute in unexpected places. Such control previously required the use of platform-specific libraries and heavyweight threads. The distance between shepherds may be determined using the qthread_distance() function.

4. Parallel architectures

Several different systems with dramatically different topologies are used to demonstrate using topology to inform runtime decisions: a dual-processor dual-core Intel Xeon 5150 workstation, a 48-processor SGI Altix 3700 ccNUMA SMP, and a dual-processor 16-core Sun Niagara 2 server. These machines were selected to demonstrate three different types of parallel systems: a common development workstation, a large ccNUMA system, and a massively multithreaded chip architecture. The topology of each of these machines is illustrated in Figure 2.

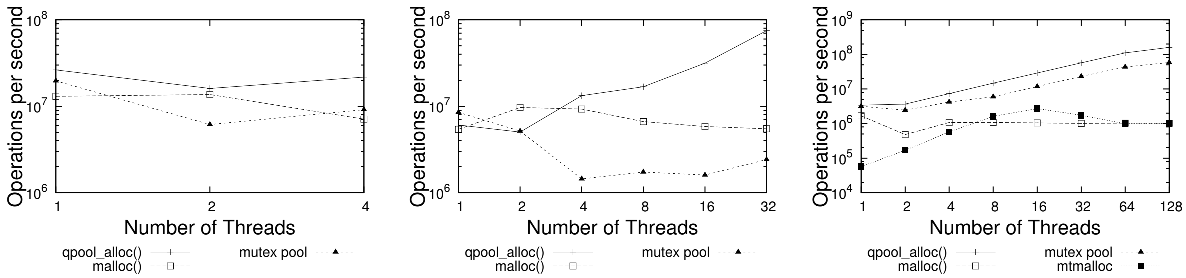
The Xeon system, Figure 2(a), is a dual-core dual-processor. Each processor has its own cache but shares the 1066Mhz front-side bus. This system runs Linux, providing the libnuma interface. Due to the shared bus, libnuma presents this system as a single node with four equidistant processors. This lack of detail is unfortunate, since cache-independence is an important aspect of the memory hierarchy. Without cache, the memory latency is uniform.

The Altix SMP, Figure 2(b), uses Intel Itanium 2 processors. The nodes are connected by dual 3.2 GB/s unidirectional links. Each node has two CPUs and a large block of RAM. The machine is divided into two components: one (A) with 16-nodes the other (B) with 8. Component A is arranged in four clusters of four nodes. Component B's nodes form a dual-plane fat-tree. The two components are asymmetrically connected by additional unidirectional links. The system runs Linux with libnuma.

The Niagara 2 server, Figure 2(c), has two eight-core processors. Each core supports eight concurrent hardware threads, for a total of 128 concurrent hardware threads. Each core has its own bank in the L2 cache which is accessible from any core in that processor. Each cache bank pair shares a dual channel FBDIMM memory controller. This system runs Solaris 10 and supports the liblgrp interface.

5. Distributed data structures

Locality awareness can improve the performance of distributed data structures. Three data structure designs demonstrate this opportunity: a pool, an array, and a queue. These data structure types are cornerstones of application design, and are used in a wide range of high-performance applications. The key feature of these new designs is that they adapt to the topology of the system in use at runtime.



(a) Xeon Allocation **(b) Altix Allocation** **(c) Niagara 2 Allocation**

Figure 3. Multithreaded 44-byte memory allocations/sec, 100-million allocations.

5.1. Distributed memory pool

Memory allocation is a frequently overlooked detail that can significantly impact performance. Standard memory allocation libraries are typically designed for general-purpose allocation in single-threaded applications, balancing allocation speed with limiting fragmentation, without concern for locality. Memory locality is typically specified per-page and needs an abstraction for general-purpose use.

A set of memory pools with mutexes to provide thread-safe access is functional, but suffers from high mutex overhead. The qthread memory pool, qpool, is implemented as a set of location-specific lock-free stacks which provide fast access to local memory. Topology information is used to pull memory from nearby memory pools when necessary.

A qpool is created via qpool_create(). Once created, elements can be fetched from the pool with qpool_alloc() and returned to the pool using qpool_free(). A pool is destroyed by qpool_destroy(). Each allocation is pulled from the local stack. Local memory is allocated in large blocks and portioned out in chunks of the size specified at pool creation. Memory that is freed is pushed onto a local lock-free re-use stack. Upon allocation, this stack is checked first. If the local stack is empty, memory is pulled from the current local large allocated block. When this block is exhausted, the re-use stacks of neighboring shepherd pools are checked, in random order. If none of them have memory, a new large block of memory is allocated from the local node's memory, and added to a list of allocated blocks. If allocation fails, the allocated blocks of all pools are checked, in order of their distance from the requesting thread. If no memory can be found, the allocation request fails.

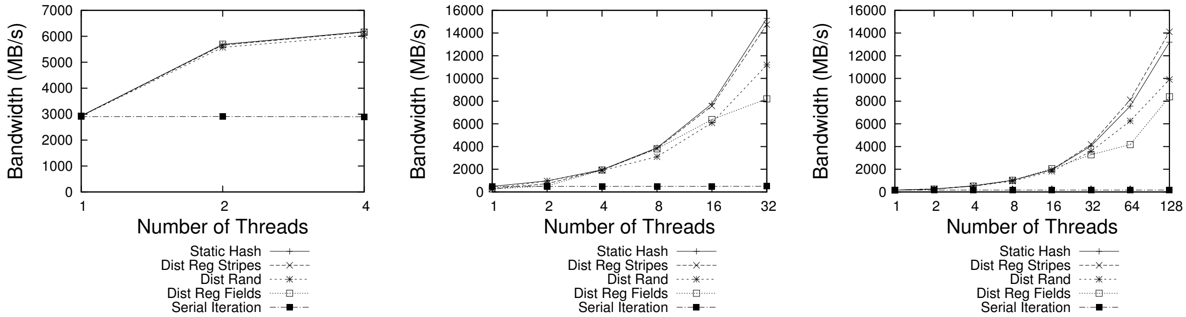
The graphs in Figure 3 compare the allocations per second of a multithreaded application that allocates, writes to, and deallocates 100 million separate 44-byte memory blocks (the size of a pthread_mutex_t on some systems). Three allocation methods are compared on the Linux systems: the qpool, a similar mutex-protected concurrent memory pool, and standard malloc. The qpool outpaces glibc malloc [10] at scale. The mutex-based memory pools suffer from Linux's slow mutex operations. The malloc implementation, while not returning location-specific memory, uses adaptive arena allocation to scale. The Solaris malloc library, designed for serial applications, is uniformly slow. Solaris's multithreaded mtmalloc library provides better performance for relatively low numbers of threads, but does not appear to be designed for more than sixteen threads and can only allocate blocks in power-of-two sizes.

5.2. Distributed array

The distributed array, or qarray, uses a basic "blocked" design, with node-specific array segments. We examine three aspects of this design: segment distribution pattern, segment size, and element size. The qarray distributes its memory when created, via qarray_create(). Once created, elements within the array can be accessed with qarray_elem(), or using pointer math within a segment. Convenient parallel iteration can be done with qarray_iter_loop(). This mechanism uses a user-specified function to process index ranges, ensuring that the function executes near the range it processes. Arrays are deallocated with qarray_destroy().

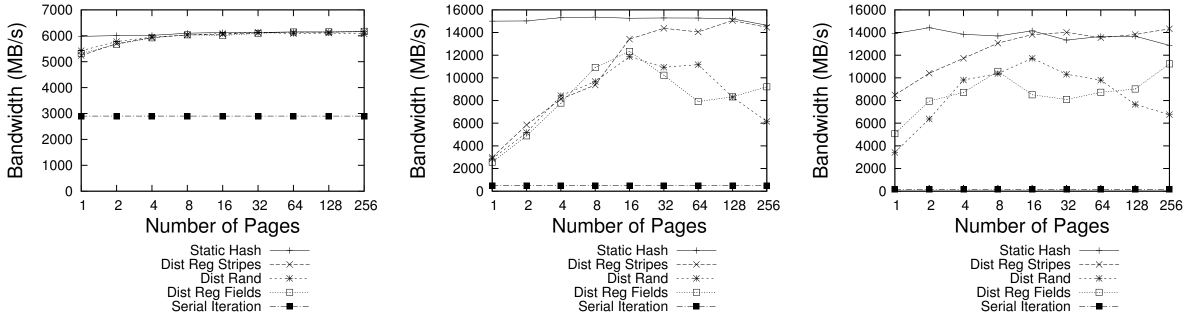
The distribution pattern and method of locating segments impacts performance. One option is to place each segment according to its order in the array, via a hash. This has the virtues of simplicity and uniformity, and avoids accessing memory to locate segments, but cannot relocate segments. Alternately, the location of each segment can be stored with the segment. Segments must be accessed to discover their location, but this enables a wider range of distribution patterns and segment relocation. Several options are compared in Figure 4. The "Static Hash" pattern uses the segment order to determine segment locations. The "Dist" patterns all store segment locations within the segments. "Dist Reg Stripes" distributes similarly to the Static Hash, "Dist Rand" distributes randomly, and "Dist Reg Fields" clusters sequential segments evenly. "Serial Iteration" is not a distribution pattern, but serves to compare the distribution patterns with typical non-qarray array iteration.

It is worth noting that parallel iteration scales well on



(a) Xeon (b) Altix (c) Niagara 2

Figure 4. Impact of distribution pattern on multithreaded memory bandwidth over large arrays.



(a) 4 threads, Xeon (b) 32 threads, Altix (c) 128 threads, Niagara 2

Figure 5. Impact of segment size and distribution pattern on multithreaded memory bandwidth over 100-million element arrays.

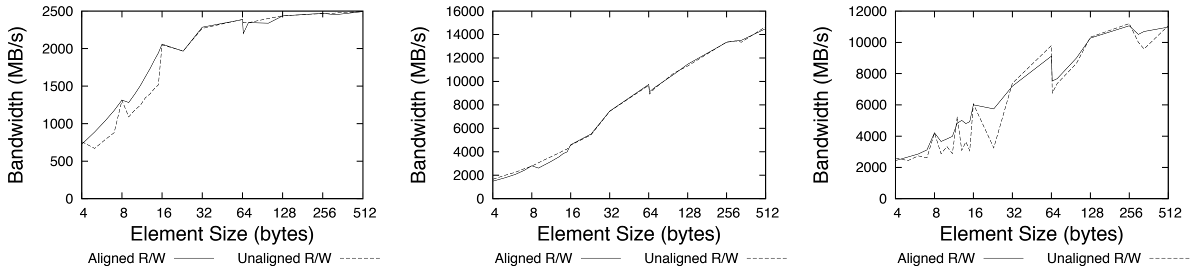
these systems. Iteration with 128 threads was 86.2x faster than serial iteration on the Niagara 2. Iteration with 32 nodes was 31.2x faster than serial iteration on the Altix. The Xeon workstation peaked at 2.1x faster than serial operation, likely because there are only two memory controllers, competing for a relatively low-bandwidth memory bus.

The static hash was the fastest on the Altix and Xeon because accessing remote memory to query its location is slow and causes incorrect prefetching. Thus, static hash is the default distribution method for all but the Niagara 2. The Niagara 2 benefits from fetching "incorrect" blocks because what is incorrect for one thread is likely correct for a nearby thread. Dist Reg Stripes leverages the Niagara 2's shared cache by clustering the working set of memory.

Distributed array performance also depends on the distribution granularity, or "segment size." Most locality-aware memory allocation methods operate on page-size memory blocks, thereby defining the minimum segment size. Thus, only large arrays can be efficiently distributed, and the segment size is a multiple of the page size. Figure 5 illustrates the impact of segment size. Larger segment sizes can either empower prefetching or create load imbalances. Distribution has a significant impact on optimal segment size. The static hash performance varied less than 12% on the Niagara 2 server, while the best performing segment size for Dist Reg Fields provides a 2.2x improvement over the worst performing size. The static hash distribution provided the best small segment size performance of the distribution methods tested; other methods need multiple pages to mask the cost of incorrect prefetching— 16 pages is a good default.

Alignment can be critical to taking full advantage of the cache as well as avoiding unnecessary bus traffic. Caches almost always load aligned data from memory. Accessing unaligned data can result in multiple load instructions, compositing instructions, and even crashes. For example, Figure 6(c) shows that the penalty for using unaligned data can be as high as a 50% reduction in bandwidth. The spikes and variances in bandwidth shown are repeatable, not the result of temporary issues. Avoiding such problems is a task that is often left to the compiler to handle, particularly for global and stack variables. Since a qarray performs layout at runtime, data alignment must be handled manually.

Figure 6 shows the impact of element size and alignment on performance. These graphs compare the memory band-



(a) 4 threads, Xeon **(b) 32 threads, Altix** **(c) 128 threads, Niagara 2**

Figure 6. Impact of alignment on multithreaded memory bandwidth over million-element arrays.

width achieved while accessing million-element arrays with a variety of element sizes. A packed array is compared to an 8-byte aligned array. Manually aligning data has a clear benefit for small element sizes, but has less benefit with large element sizes. This is likely because unaligned layout is more condensed, and that benefit outweighs the penalty for unaligned access beyond a certain size element. Thus, unless otherwise directed, the qarray automatically rounds element sizes under 64 bytes to the next-largest multiple of eight.

5.3. Distributed queue

Parallel queue designs depend on the use-case. If a queue is a buffer between two threads, assumptions can improve speed. For example, assuming low contention for the queue's head and tail, there need only be one of each. Queues are also used for distributing work among multiple threads, with multiple producers and multiple consumers. In that situation, the guarantees of a simpler queue, like global ordering, may be relaxed in order to increase speed. Such a queue may prefer dequeueing nearby elements over the oldest elements in the overall queue.

The qthread library provides both types of queue. The qlfqueue, based on Michael and Scott's lock-free queue [16], guarantees global ordering. The end-to-end ordered queue is the qdqueue. New qlfqueues are created with qlfqueue_create() and destroyed with qlfqueue_destroy(). Elements may be queued with qlfqueue_enqueue() and dequeued with qlfqueue_dequeue(). A quick element check may be performed with qlfqueue_empty(). Equivalent qdqueue functions begin with "qdqueue" instead of "qlfqueue."

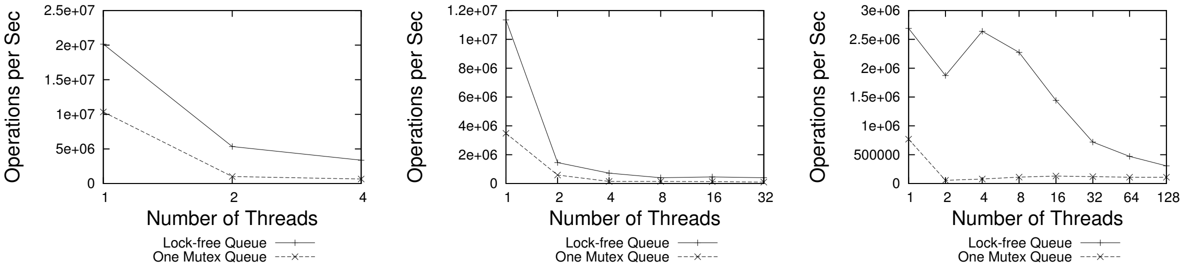
Figure 7 illustrates the effect of strict ordering on the scalability of a queue by comparing the lock-free qlfqueue to a single-mutex queue implementation from the cprops library [1]. Ensuring global ordering requires a serialized critical section. A lock-free queue is faster than a mutex-based queue because it uses hardware assistance to minimize ordering overhead. Nevertheless, this serialization acts as a bottleneck that precludes scaling.

The core of an end-to-end ordered queue is matching consumers to producers. A central matchmaker is a simple approach, but creates a bottleneck. Hierarchical matchmaking reduces consumer contention, but increases the work of producers without addressing their bottleneck issues. A "stochastic distributed queue" [13], where consumers repeatedly probe random producer queues until satisfied, is efficient when the queues are rarely empty. But if queues are frequently empty, random polling creates unnecessary work. Random polling can be avoided with advertisements.

Per location, the qdqueue uses a qlfqueue, a list of "ads" received, and a record of the last consumed element's source. Elements are enqueued locally. If the queue was not empty, ads are posted to nearby queues. An ad informs remote consumers that there is data available in this queue. The set of "neighbor" shepherds to receive advertisements from any given shepherd is determined at setup time, based on distance. Thus, the maximum work of a producer is fixed, independent of the size of the system. A producer can avoid resending ads by tagging them with a counter and tracking the highest consumed ad counter value. If the last-issued ads have not been consumed, ads need not be re-issued.

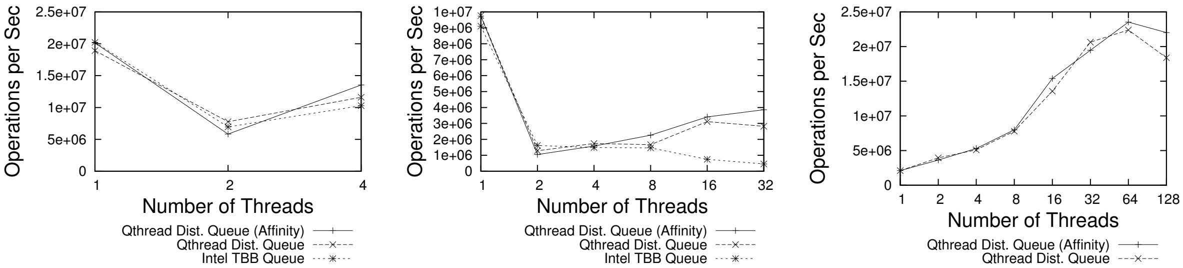
To dequeue, the local queue has priority. If the local queue was empty, any known ads are checked in order of distance. The source of the last-dequeued element is recorded locally. When responding to an ad, update the advertiser's record of ads consumed. If none of the ads result in an element, it is necessary to check *all* remote queues, in the order of distance from the consumer. If a remote queue is empty, that queue's last-consumed record is treated as an ad. If an ad is received while checking remote queues, it is checked immediately. Thus, consumers cooperate to find elements. If all remote queues are empty, the consumer must either return empty-handed or wait for an ad to be posted.

Figure 8 compares the scaling of the qdqueue with the TBB concurrent queue, which have similar ordering guarantees. The benchmark, in both cases, transfers word-size data using a variable number of enqueueing threads with an identical number of dequeueing threads. The qdqueue can use CPU pinning — labeled "(Affinity)." In single-



(a) Xeon **(b) Altix** **(c) Niagara 2**

Figure 7. The ops/sec of strictly ordered multithreaded queues.



(a) Xeon **(b) Altix** **(c) Niagara 2**

Figure 8. The ops/sec of end-to-end ordered multithreaded queues with small elements.

threaded mode the qdqueue is just as fast as the qlfqueue, but improves on the qlfqueue's multi-thread performance. At scale, the qdqueue performed 9.6x better than the qlfqueue on the Altix, 72x better on the Niagara 2, and 4x better on the Xeon. The Niagara 2's performance is a result of cheap communication via shared cache. Cache-coherency penalizes shared information on the Altix and Xeon. Figure 9, illustrates the impact of using larger (1024-byte) blocks of data. Large inter-node transfers impose a higher penalty than small ones, magnifying the benefit of location affinity.

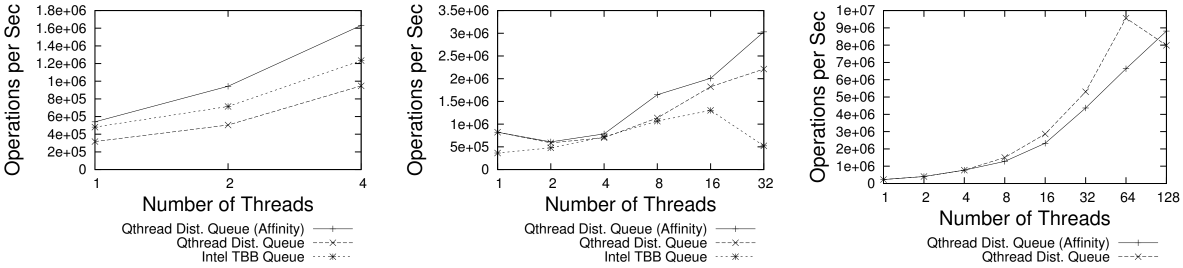
6. Future work

There are many additional array operations common in scientific computing that benefit from locality-awareness. Array stencils are common in signal processing, image processing, and solving partial differential equations. Foreknowledge of the stencils can be used to influence distributed array layout, such as by aligning stencil and segment boundaries and avoiding unnecessary thread spawns. Array combination, such as in string comparison, genetic research, matrix multiplication, and multi-dimensional arrays, are also common. These array combination operations have well-understood memory access patterns that provide opportunities for locality-aware optimization. A key question is where to best execute operations with disperse input.

MapReduce [7] is a powerful asynchronous and easily pipelined data-processing abstraction popularized by Google. It could be implemented with distributed queues instead of the usual "master" node approach, naturally preserving locality between stages. This design may be useful for exploring the best number of workers at each stage, the optimal information transfer method, and the effect of worker loss and worker migration.

7. Conclusion

Developing portable locality-aware data structures, even with the advantage a threading library with integrated locality, is a significant challenge. We have presented a portable threading interface that integrates locality, several data structure designs that use locality information, and examined the selection of optimal system-specific design parameters. We have demonstrated the effectiveness of locality-aware design in distributed data structures. Memory pools can be up to 155 times faster than traditional malloc() while providing location-specific memory. The qarray distributed array design supports strong-scaling on large NUMA systems, executing 31.2 times faster with 32 nodes than with one. The qdqueue distributed queue design provides up to a 47x improvement over a fast lock-free queue by providing only end-to-end ordering. Locality-awareness



(a) Xeon **(b) Altix** **(c) Niagara 2**

Figure 9. The ops/sec of end-to-end ordered multithreaded queues with large (1024 byte) elements.

provides up to an 8.3x improvement in performance over the state-of-the-art concurrent queues on a large NUMA system. Importantly, the use of locality information does not significantly impact serial performance or performance on small systems with uniform memory access latencies. Exposing hardware non-uniformity is becoming common, and the unification of thread handling and topology is an important direction for future shared-memory data structure research.

References

[1] I. Aelion. cprops - C prototyping tools. http://cprops.sourceforge.net, March 2009.

[2] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele Jr., and S. Tobin-Hochstadt. *The Fortress Language Specification v. 1.0β*. Sun Microsystems, 2007.

[3] R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. Patterson, and K. Yelick. Cluster I/O with River: Making the fast case common. In *Proc. of the 6th Workshop on I/O in Parallel and Distributed Systems*, pages 10–22, New York, NY, 1999. ACM.

[4] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *Proc. of the 5th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 207–216, New York, NY, 1995. ACM.

[5] J. Chapin, A. Herrod, M. Rosenblum, and A. Gupta. Memory system performance of UNIX on CC-NUMA multiprocessors. In *Proc. of the 1995 ACM SIGMETRICS Joint International Conf. on Measurement and Modeling of Computer Systems*, pages 1–13, New York, NY, 1995. ACM.

[6] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioğlu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proc. of the 20th Ann. ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 519–538, New York, NY, 2005. ACM.

[7] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proc. of the 6th Symp. on Operating Systems Design & Implementation*, page 10, Berkeley, CA, 2004. USENIX Assoc.

[8] R. Diaconescu and H. Zima. An approach to data distributions in chapel. *International Journal of High Performance Computing Applications*, 21(3):313–335, 2007.

[9] T. El-Ghazawi and L. Smith. Upc: unified parallel c. In *Proc. of the 2006 ACM/IEEE Conf. on Supercomputing*, page 27, New York, NY, 2006. ACM.

[10] W. Gloger. Dynamic memory allocator implementations in Linux system libraries. http://www.dent.med.uni-muenchen.de/ wmglo/, May 1997.

[11] Institute of Electrical and Electronics Engineers. *IEEE Std 1003.1-1990: Portable Operating Systems Interface (POSIX.1)*, 1990.

[12] Intel Corp. *Intel® Threading Building Blocks v. 1.6*, 2007.

[13] T. Johnson. Designing a distributed queue. In *Proc. of the 7th IEEE Symp. on Parallel and Distributed Processing*, pages 304–311, Washington, DC, 1995. IEEE Comp. Soc.

[14] A. Kleen. An NUMA API for Linux. http://halobates.de/numaapi3.pdf, August 2004.

[15] U. Manber. On maintaining dynamic information in a concurrent environment. In *Proc. of the 16th Ann. ACM Symp. on Theory of Computing*, pages 273–278, New York, NY, 1984. ACM.

[16] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. of the 15th Ann. ACM Symp. on Princ. of Dist. Comput.*, pages 267–275, New York, NY, 1996. ACM.

[17] D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta, and D. Ayguadé. Is data distribution necessary in OpenMP? In *Proc. of the 2000 ACM/IEEE Conf. on Supercomputing*, page 47, Washington, DC, 2000. IEEE Comp. Soc.

[18] R. W. Numrich and J. Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998.

[19] Open MPI Team. Portable Linux processor affinity. http://www.open-mpi.org/projects/plpa/, March 2009.

[20] OpenMP Architecture Review Board. *OpenMP Application Program Interface*, 2.5 edition, May 2005.

[21] Sun Microsystems, Inc., Santa Clara, CA. *Memory and Thread Placement Optimization Developer's Guide*, 2007.

[22] J. Tao, W. Karl, and M. Schulz. Memory access behavior analysis of NUMA-based shared memory programs, 2001.

[23] K. Wheeler, R. Murphy, and D. Thain. Qthreads: An API for programming with millions of lightweight threads. In *Proc. of the 22nd IEEE International Parallel & Distributed Processing Symp.* IEEE Comp. Soc., April 2008.