

# Scaling Runtimes for Irregular Algorithms to Large-Scale NUMA Systems

Andrew Lenharth and Keshav Pingali, University of Texas at Austin

*The Galois system can automatically parallelize irregular algorithms written in a serial programming model and execute them efficiently on nonuniform memory access (NUMA) machines. Experimental results for five complex irregular algorithms show that the system scales up to 420× on large NUMA systems at 512 threads.*

**M**any applications in areas like social network analysis and machine learning use irregular algorithms in which dependencies between computations are functions of runtime values. Therefore, these applications cannot be parallelized using the restructuring compiler technology that has found some success in computational science.<sup>1</sup> One solution is to use domain-specific languages (DSLs) or domain-specific parallelization techniques. This approach has been popularized in graph-analytics packages such as Pregel,<sup>2</sup> GraphLab,<sup>3</sup> and Ligra.<sup>4</sup> However, these systems support a very restricted class of operations on graphs—for example, none permits modification of the graph structure by adding and removing nodes and edges, so even graph construction must be done outside these systems. Most complex irregular applications cannot be programmed in such systems.

The Galois system,<sup>5</sup> in contrast, does not require DSLs or domain-specific parallelization techniques; instead, it provides application programmers with a carefully

restricted sequential programming model that can be implemented in any object-oriented language (we use C++, though prior implementations used Java). All concurrency control is managed within a library of concurrent data structures, such as work lists and graphs, and within the runtime system.

Galois was originally designed for uniform memory access (UMA) shared-memory systems, limiting its scalability on modern hardware. Here, we describe an implementation of the Galois programming model and runtime system that scales to non-UMA (NUMA) machines with 512 cores. This is achieved by changes to the concurrent data-structure library and runtime system, the most important of which are the following:

- ▶ locality information is embedded pervasively throughout the Galois API, runtime system, and concurrent data-structure library;
- ▶ algorithms from distributed systems are used in the Galois runtime system where appropriate;

- › memory management for common-case memory-allocation patterns is handled by a fast, scalable, NUMA-aware custom memory-management system;
- › concurrent data structures are tailored to use scalable memory-allocation methods;
- › exception handling in C++ is re-implemented in a scalable way; and
- › interactions between fast runtime components that result in nonscalable behavior are controlled.

To demonstrate Galois scaling capabilities, we implemented five complex irregular applications: Delaunay mesh refinement (DMR), Delaunay triangulation (DT), betweenness centrality (BC), Barnes–Hut (BH), and triangle finding (TRI). On a 512-core SGI UV NUMA machine, all applications scaled to more than 200× and two scaled to more than 400×. This is the first time that some of these applications have scaled to such levels.

### GALOIS PROGRAMMING MODEL AND LOCALITY EXTENSIONS

Galois implements a data-centric parallelism pattern called *amorphous data-parallelism* (ADP).<sup>5</sup> Here we briefly describe the programming model and features for supporting NUMA architectures; a more complete discussion can be found elsewhere.<sup>5,6</sup>

#### Programming model

A graph algorithm performs computation on the graph's *active* nodes or edges. This computation is the *operator* for the algorithm. The application of an operator to an active node is an *activity*. The set of data elements read and written by an activity is

its *neighborhood*. In a correct parallel implementation, the operator must appear to execute atomically. Here we consider only unordered algorithms—those in which activities can be executed in any order. One way to parallelize unordered algorithms is to execute parallel activities with disjoint neighborhoods.

Galois is implemented in C++ and follows standard C++ idioms. Computations to be executed in parallel are written as ADP iterators, which are similar to the standard library's *for-each* loop. The ADP iterators take an *iteration range* and a C++-style *functor*. The iteration range is the initial set of active elements, and the functor is the operator. Galois applies the functor to each element in the iteration range but can add new elements to the range, unlike conventional *for-each* loops. The Galois runtime passes a context object to the functor in addition to the active element. The context exposes a small API that allows the activity to create new activities (dynamically generate work) and exposes optimized memory allocators. Galois provides a DSL for scheduling and synthesizes efficient and scalable schedulers for each loop.

Figure 1 shows an example Galois operator for triangle counting. This implementation uses sorted edges to speed up the search and is very close to a simple serial implementation. Note the Galois-provided graph, loop construct, and accumulator.

The Galois runtime system exploits parallelism by applying the functor to elements in the iteration range in parallel. The key goal is to ensure atomic execution of each of these applications because the set of data elements read and written by each functor application is not statically known.

To this end, Galois provides a set of concurrent data structures including graphs, sets, trees, and bags that work with the runtime system to track the elements accessed by each activity; if two activities try to access the same element, the runtime system rolls back one of the conflicting activities. Galois makes this speculative parallelization lightweight by requiring all functors to be *cautious*<sup>5</sup>—that is, they must read all graph elements in their neighborhood before modifying any element.

The combination of smart data structures and runtime parallelization removes all concurrency concerns from user code; iterations in the code are executed with transactional semantics, with the runtime system discovering and dealing with data dependencies. Therefore, application code can be written as though only one activity is active at any point.

#### Locality extensions

For NUMA architectures, we extend the Galois programming model with a single directive, *local*, to expose a notion of data placement to the programmer and to the runtime system. This is sufficient to implicitly express partitioning and data placement. We do this, rather than treating partitioning and data placement as first-class entities, to maintain an implicitly threaded programming model.

All Galois data structures, such as graphs and sets, have an implicit notion of locality. All elements are internally owned by a thread and allocated on memory local to that thread. The data structures allow the runtime system to iterate over elements owned by a particular thread. Conceptually, the runtime system can convert a loop iterating over a data structure to a

```

GAccumulator <size_t> numTriangles;

struct tricount {
    void operator()(const GNode& n, UserContext <GNode>&) const {
        // Partition neighbors, assumes no self loops
        // [first, ea] [n] [bb, last)
        auto first = graph.edge_begin(n);
        auto last = graph.edge_end(n);
        auto ea = lowerBound(first, last, n);
        auto bb = upperBound(first, last, n);

        for (; bb != last; ++bb) {
            GNode B = graph.getEdgeDst(bb);
            for (auto aa = first; aa != ea; ++aa) {
                GNode A = graph.getEdgeDst(aa);
                if (binarySearch(graph.edge_begin(A), graph.edge_end(A), B))
                    numTriangles += 1;
            }
        }
    }
};

for_each_local(graph, tri_count());
std::cout << "NumTriangles:" << numTriangles.reduce() << "\n";

```

**FIGURE 1.** Example Galois operator for triangle counting.

per-thread loop iterating over thread-local data. User-specified scheduling policies might overrule this. Locality extensions are sufficient to load partitioned graphs into memory, matching partitions to NUMA nodes.

## DESIGN PRINCIPLES FOR SCALABLE IRREGULAR RUNTIME

Scaling runtime to numerous threads requires adhering to designs that respect the hardware’s inherent costs and organization. We group design principles into two classes: those arising from the programming model and from the hardware.

### Optimizing speculative execution

*Implement conflict checking to ensure isolation between iterations without imposing an extra cache penalty for communication when there is no contention.* Communication is necessary to discover conflict, but absent conflict, no extra communication should occur.

*Reduce memory latency by processing data locally wherever possible.* Because a core in a NUMA system has different access latencies to different memory banks, cores should process data on the closest (lowest-latency) memory bank when possible. This locality concern is in addition to the normal multicore considerations of cache locality.

### Optimizing for hardware characteristics

*Tailor algorithms to prefer local communication while minimizing remote communication.* Communication costs are hierarchical due to cache nesting and the topology of the interconnect network between NUMA nodes. Cores sharing an L3 cache, for example, communicate faster than cores separated by the NUMA interconnect.

*Avoid unnecessary cache invalidation.* Implementations in which multiple cores write to the same cache line do not scale. Although data sharing through writes to memory is easy in a shared-memory programming model, the cost is high if the writes are to the same cache line. Thus, writes to shared cache lines should be given the same consideration as sending an explicit message would in a distributed-memory design.

*Ensure that logically independent operations on a data structure are independent at the cache level.* Simple implementations of many data structures on shared memory create mutable state shared among many threads. A scalable data structure cannot do this—for example, adding a node to a graph should be an entirely thread-local operation (including allocation).

### Implications

*Threads sharing a cache can coordinate through shared data structures.* As long as contention is controlled, they can scale to a few close threads. This allows the implementation simplicity of shared data structures for local clusters of threads.

*Threads that do not share a cache should use distributed algorithms for coordination.* Due to the high cost of cache-coherence traffic across NUMA machines, the system-wide algorithms used in the runtime system should be distributed. Communication is implicit through writes rather than message sends, but this only affects the implementation of the runtime system’s algorithms; the design must be communication-aware.

*The scheduler must know the memory layout of data structures and topology of the memory system to optimally schedule tasks.* The cost of cache misses can vary widely, depending on the data being accessed. Scheduling tasks on cores near their data minimizes cache-miss latency.

*Reader-writer locks should not be used.* Common implementations of such locks require updating shared state even for readers, making the locks non-scalable even for all-reader workloads.

*Lock-free data structures cannot solve scaling problems in NUMA systems.* Lock-free data structures have writes to common memory locations, which does not scale.

### GALOIS SOFTWARE STACK

Irregular algorithms use language and system features in ways that regular algorithms do not. Initially, a few key components in the Galois software stack greatly limited scalability, requiring careful re-implementations of those components to conform with the design principles discussed above.

#### Memory allocation

Memory allocation is common in complex irregular algorithms. Even at low thread counts, the standard allocation subsystems *malloc* and *mmap* are major scalability bottlenecks. For simple algorithms that do not *morph* the underlying graph's structure and hence do not allocate or delete storage for nodes and edges, allocation is needed only in the scheduler for dynamically generated work and for temporary allocations in the operator. More complex algorithms can be very allocation intensive because they add and remove nodes from a graph, expand a tree, or track temporary variable-sized state within a task.

Four of the five algorithms we evaluated require substantial memory allocation. DMR, for example, increases the graph by approximately 550 bytes per task—at 512 threads, this can exceed 50 gibibytes (GiB) per second. This is simply the persistent data and does not include temporary data structures used by the runtime system.

An additional complication on NUMA systems is that the physical location of memory used to satisfy an allocation is important. Memory used by a thread should be allocated on the memory bank local to that thread to minimize cache-miss latency.

**Allocation scalability.** While efforts to create scalable memory allocators<sup>7,8</sup>

have been considerable, they have not addressed NUMA concerns directly or produced an allocator capable of scaling to large, allocation-intensive multithreaded workloads.

Currently, the rate of kernel memory allocation, even with huge pages, is insufficient to meet the allocation demands of algorithms like those we evaluated at high thread counts. Further, contention in the kernel causes unpredictable and potentially unbounded sleep. Thus, we pre-allocate huge pages from the kernel into a page pool in our memory system. This has the added benefit of making it trivial to track the thread and NUMA affinity of each page of memory used, allowing layered allocators to behave correctly.

**Exploiting structure in allocator usage.** The Galois programming model offers considerable structure in allocator usage. We exploit this structure by using fast, scalable specialized allocators where appropriate. Allocator use falls into three categories: per-task variable-sized allocation, data-structure allocation, and general allocation. We optimize the first two cases, avoiding the third.

*Per-task variable-sized allocation.* Some irregular algorithms use temporary data structures to track state. DMF, for example, collects a set of triangles to be replaced. The Galois runtime system provides a per-task region allocator that can be used for these types of allocations. It supports standard C++ allocator semantics, making it compatible with all standard containers, and allows efficient, node-local, variable-sized allocations in the user code. The allocator is reused within a thread, reducing cold misses in the cache.

*Data-structure allocation.* Data structures are the most common

source of runtime allocation. We codesign data structures with a fixed-size allocator family to efficiently support these allocations. Internally, the system creates fixed-sized allocators that are efficient and NUMA-friendly. Data structures are written to exclusively use these allocators.

*Node-local memory.* The memory system targets two problems. First, memory allocation must scale to many threads as well as high allocation rates. The system meets this goal with a series of specialized allocators, careful data-structure design, and a layered allocation strategy. Second, memory allocation must be NUMA-aware—that is, it must return memory local to the NUMA node on which it is being allocated. The memory system achieves this through the use of per-thread allocators backed by a NUMA-aware page pool. Per-thread allocators are more fine-grained than NUMA nodes and naturally return node-local memory. The page pool tracks memory pages by NUMA node and never returns remote memory in response to a request. The page pool ensures that memory allocated from the OS resides on the node that performs the allocation.

#### Isolation: locking, speculation, and retry

The Galois programming model requires isolation between tasks. We use speculative execution to implement isolation, and we exploit common structure in irregular algorithms to avoid the costs of generic solutions, such as software transactional memory. As a task executes, the data-structure implementation marks elements used by the task. If a mark cannot be acquired, the task aborts and is retried later. Marks are made in an

## RELATED WORK

Parallel programming models, languages, and frameworks are manifold.

Implicitly parallel programming<sup>1</sup> attempts to recover parallelism from a sequential programming model with compiler analysis. It uses stylized tasks to simplify this discovery. We implement stylized loops as entry points to the parallel runtime system and perform the analysis at runtime to discover parallelism.

Cilk<sup>2</sup> and OpenMP<sup>3</sup> provide constrained, explicit forms of parallelism requiring programmer annotation. They leave data synchronization and safety to the programmer. We put the responsibility for safety on the runtime system. Although it is possible to write NUMA-scalable applications in these frameworks (except, perhaps, for scheduling), one would have to implement most of the data structures and mechanisms demonstrated here.

Chapel<sup>4</sup> and UPC<sup>5</sup> provide a partitioned global address space in which memory is explicitly partitioned between local and remote, though addressing is still global. This differs from our approach in that we make partitioning of the memory space implicit in the data structures, rather than having data structures explicitly mapped to partitions.

Global Arrays<sup>6</sup> and PETSc<sup>7</sup> provide both NUMA and distributed process-centric programming models with shared data structures and explicit data movement. We believe that for NUMA architectures, a successful programming model should require little change in how the average programmer thinks about programs.

Several OpenMP implementations<sup>2,8</sup> add features such as data-layout, memory-migration, and computation-placement directives; hierarchical

task stealing; topology- and NUMA-aware load balancing; and various scheduling hints.

Charm++ has a number of NUMA optimizations, most notably NUMA-aware load balancing.<sup>9</sup>

### References

1. W. Hwu et al., "Implicitly Parallel Programming Models for Thousand-Core Microprocessors," *Proc. 44th Ann. Design Automation Conf. (DAC 07)*, 2007, pp. 754–759.
2. M. Frigo, C.E. Leiserson, and K.H. Randall, "The Implementation of the Cilk-5 Multi-threaded Language," *Proc. 1998 ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI 98)*, 1998, pp. 212–223.
3. R. Chandra et al., *Parallel Programming in OpenMP*, Morgan Kaufmann, 2001.
4. B.L. Chamberlain, D. Callahan, and H.P. Zima, "Parallel Programmability and the Chapel Language," *Int'l J. High Performance Computing Applications*, vol. 21, no. 3, 2007, pp. 291–312.
5. UPC Consortium, *UPC Language Specifications, V1.2*, tech. report LBNL-59208, Lawrence Berkeley Nat'l Lab, 2005.
6. J. Nieplocha, R.J. Harrison, and R.J. Littlefield, "Global Arrays: A Non-Uniform-Memory Access Programming Model for High-Performance Computers," *J. Supercomputing*, vol. 10, no. 2, 1996, pp. 169–189.
7. S. Balay et al., "Efficient Management of Parallelism in Object Oriented Numerical Software Libraries," *Modern Software Tools in Scientific Computing*, E. Arge, A.M. Brueset, and H.P. Langtangen, eds., Birkhäuser Press, 1997, pp. 163–202.
8. L.C. Freeman, "A Set of Measures of Centrality Based on Betweenness," *Sociometry*, vol. 40, no. 1, 1977, pp. 35–41.
9. L.L. Pilla et al., *Improving Parallel System Performance with a NUMA-Aware Load Balancer*, tech. report, Univ. of Illinois at Urbana-Champaign, 2001.

object header, so marking incurs very little overhead for subsequent access of the object. Data structures are updated in place after all marks for a task are acquired. The communication involved in marking an object is necessary to discover independent tasks, but is minimized by storing the mark in the object header. Thus, marking

will bring into the cache a part of the object that the task will later access.

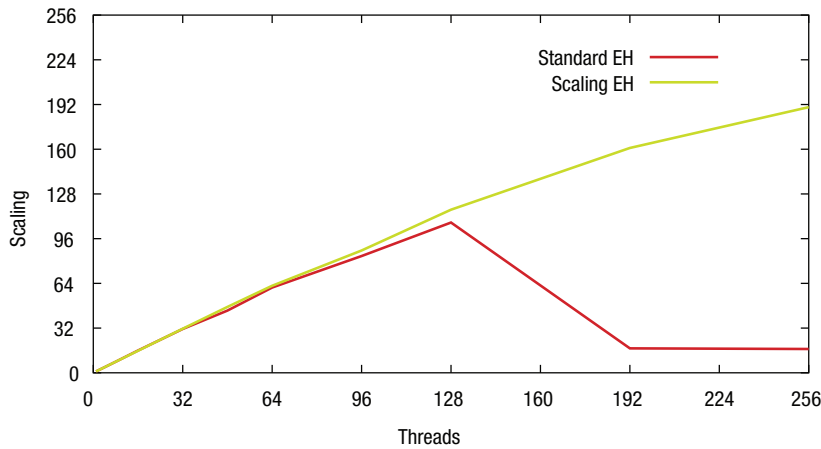
The handling of aborts and retries has several contradictory goals. Although rare in most algorithms, conflicts can occur at large thread counts. Simple cache invalidation at scale can be a performance bottleneck. It is therefore critical to resolve

most conflicts and avoid live-lock by minimizing communication and synchronization between threads.

### Low-overhead live-lock freedom.

At runtime, a task marks objects it requires for exclusive access. The task is speculatively executed, on the assumption that it will be able

## IRREGULAR APPLICATIONS



**FIGURE 2.** Delaunay mesh refinement scaling with and without scalable exception handling (EH). Contention in the allocator and reader-writer lock (only taken in read mode), although only present in less than 1 percent of tasks, severely limit scaling.

to mark all required objects. Speculation fails when one task attempts to mark an object owned by another task. Speculation failure is very rare—usually a fraction of a percent of all tasks—but does increase with the number of threads. The highest failure rate in our evaluation is 4.5 percent for DMR, which is also the best scaling benchmark.

Dealing with speculation failure—freeing locks and deciding when to retry the task—does not require any communication. Simply retrying execution would require no communication, but often leads to live-lock. Therefore, we apply a layered strategy that deals with aborted work in the common case using only local communication (for the machines in this study, communication between threads which share an L3 cache). Tasks are retried after some random delay. Subsequent aborts of retried work incur more expensive communication to prevent live-lock. All packages are numbered consecutively and, after the initial retry, an aborted task is moved to a per-package (for example, L3 cache) retry queue with a number that is half the current package's number. This forms a binary tree over packages that a retried task moves up in after each subsequent abort. If all work continuously aborts, aborted work will serialize into a single thread on the first package, thereby avoiding live-lock. In

practice, work items rarely abort twice, let alone more often than that.

**Scalable exception handling.** The Galois runtime system uses exceptions to take advantage of compiler-supported state rollback in conjunction with per-task region allocators to reclaim memory. Conceptually, this rollback mechanism is entirely thread-local and thus scalable, but real implementations fail to achieve this. Although conflicts, and thus exceptions, are a small fraction of total tasks, at hundreds of threads we can expect at least one conflict at any point in time.

The compiler, standard exception-handling library, and C library cooperate to implement exception handling. This combination has two non-scalable components. First, the standard library uses dynamic allocation to store the thrown exception object. Second, exception handling and dynamic library opening and closing have a potential race condition protected by a reader-writer lock. The cache invalidations from acquiring this lock in read mode when throwing exceptions inhibit scaling. We modify the relevant libraries to disable dynamic library loading and unloading during parallel loops. This allows lock-free access, which is now read only, to the various exception tables. An alternate implementation would be to change the type of reader-writer lock to one

that does not require communication for reader-only workloads.

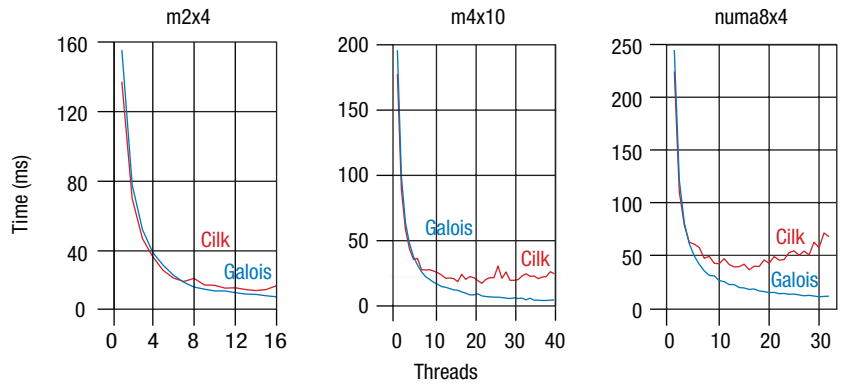
Figure 2 shows DMR scaling with and without scalable exception handling.

### Scheduling

We use the scheduling language developed by Donald Nguyen and one of this article's authors, Keshav Pingali,<sup>9</sup> to describe good scheduling orders for loops. The choice of scheduling policy is algorithm-specific, and an algorithmically good scheduling policy can have orders-of-magnitude more benefit than a purely locality-based policy. Consequently, we do not simply implement a NUMA-aware scheduler and apply an owner-computes policy to all loops. Instead, we use a twofold approach: we present the initial work in a loop to the scheduler in a locality-preserving way and supply NUMA-aware implementations of common scheduling policies.

The notions of locality embedded in the data structures and runtime system are used during scheduling. A for-loop presents the portions of the iteration range owned by each thread to the scheduler on that thread. This is possible because the iteration range presented to the loop is over a data structure that exposes data placement to the runtime system. Thus, the scheduler sees the data on the optimal node. How work is scheduled after this point entirely depends on the scheduling policy and specific scheduler synthesized for the loop.

Overhead is a critical aspect of scheduling for many irregular algorithms. Tasks can be as small as 100 cycles, so scheduling decisions must be very efficient. Scheduling is complicated by the fact that many algorithms dynamically generate work, and simple local LIFO (last-in, first-out) or



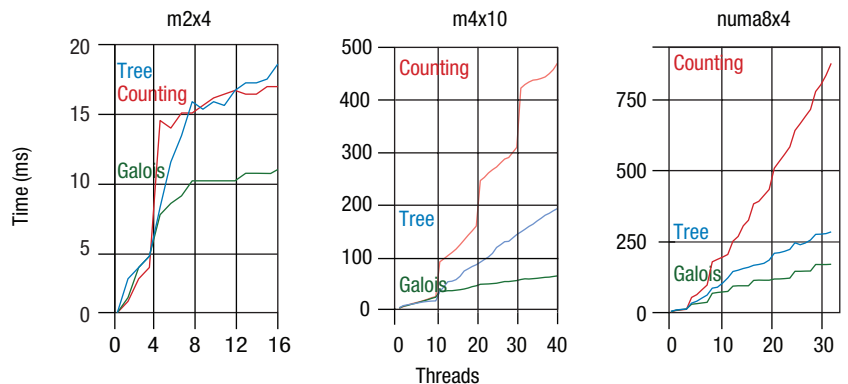
**FIGURE 3.** Runtime of fibonacci(30), a task-scheduling microbenchmark, implemented using two tasks per recursive step. m2x4 is a two-socket, four-cores-per-socket Intel Xeon system; m4x10 is a four-socket, 10-cores-per-socket Intel Xeon system; and numa8x4 is an eight-socket, four-cores-per-socket SGI UV Intel Xeon-based system.

FIFO (first-in, first-out) policies, such as those implemented in Cilk, are not ideal for all algorithms. For a sense of scale, the TRI benchmark schedules 9.6 billion tasks per second at 512 threads.

Several changes to the core Galois work-list components improved NUMA scaling. Primarily, we converted the main Galois schedulers from hierarchically partitioned queue implementations to per-thread queues with NUMA-aware work stealing. When a thread's work list is empty, it tries to steal half of the work from another thread within the same package. This preserves locality, as work does not leave the L3 cache, while performing limited load balancing. When all threads in a package are out of work, a single master thread from each package attempts to steal the entire work list from another master thread in a different package. This limits the number of robbers and victims communicating across the NUMA interconnect. The thread whose work was stolen will first try to steal work locally, effectively drawing from a reserve of work not available to be stolen.

Figure 3 shows the effect of NUMA-aware work stealing on a scheduler microbenchmark. The baseline is the Cilk scheduler using random work stealing. Using the same queuing policy, but NUMA-aware work stealing, significantly improves performance and scaling.

**Interactions between stealing and termination detection.** We used the Dijkstra-Scholten two-pass, ring-based termination detection algorithm (DS) to exit a parallel loop. The stealing protocol and termination detection interact in a way that can drastically affect scaling. Once a thread is out of work, it enters a loop in which it first checks for work, attempting to steal,



**FIGURE 4.** Runtime for  $16 \times 1,024$  barrier invocations. *Counting* is a simple counting barrier, *Tree* is a classic MCS barrier, and *Galois* is our hybrid NUMA-aware barrier. The machines are the same as those described in Figure 3.

and then checks for and passes the termination token. With the initial stealing protocol, stealing by each thread had a cost proportional to the number of threads  $O(n)$ . The token passing also required a number of steps proportional to the number of threads  $O(n)$ . On average, we expect the token to arrive in the middle of the stealing loop, resulting in local delay proportional to the number of threads before a thread checked for and passed on the termination token. This leads to  $O(n^2)$  runtime for exiting a loop.

We modified the stealing protocol in all work lists to only attempt a single steal operation before failing. Future stealing attempts will try the next victim. This limits the local delay in propagating the termination token to a single stealing attempt from

a single victim. At 256 threads, this change reduced shutdown time by 30 ms, which at this scale was more than 50 percent of some loop runtimes.

**NUMA-aware hybrid barrier.** We observed that, because of fast coherence, a simple counting barrier is more efficient within a shared cache than a classic tree-based MCS barrier. Inter-NUMA-node, tree-based barriers are significantly more efficient than counting barriers. We leverage this fact to use local counting barriers between fast communicating threads and select one from each group to participate in a tree-based barrier algorithm. Figure 4 shows that the NUMA-aware hybrid barrier outperforms both classic implementations.

TABLE 1. Algorithm inputs and configurations.

Algorithm	Input and configuration
Delaunay mesh refinement	10 million bodies generated using a Plummer model; tolerance = 0.05; time step = 0.50; events/s = 0.05
Delaunay triangulation	20 million triangles in a square, 50 percent bad
Barnes-Hut	10 million points randomly distributed in a square
Betweenness centrality	Random graph with average degree 4 and $2^{18}$ nodes
Triangle finding	Random planar graph with average degree 4 and $2^{18}$ nodes

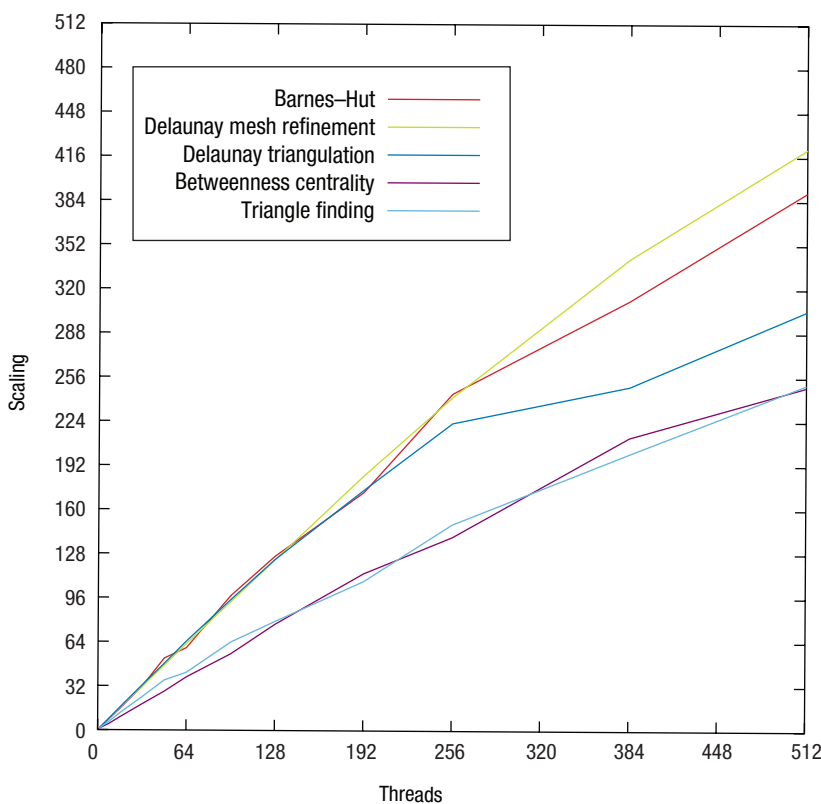


FIGURE 5. Self-relative scaling of five irregular algorithms on the Pittsburgh Supercomputing Center’s Blacklight at up to 512 threads.

EXPERIMENTAL RESULTS

To assess Galois scaling capabilities, we ran several irregular algorithms from graphics, computational science, and graph analytics:

- › DMR takes a Delaunay triangular mesh and produces a resulting mesh that satisfies several quality guaranties. We modified the Lonestar<sup>10</sup> implementation

to sort the initial triangles using a space-filling curve. This functions as a partitioner that assigns consecutive blocks of triangles to each thread.

- › DT takes a set of points and produces a triangular mesh satisfying the property that the *circumcircle* (bounding circle) for each triangle must not contain points in the mesh that are not

part of the triangle. We extended the Lonestar implementation to sort points according to a space-filling curve. DT builds a spatial acceleration structure to map points to triangles in the mesh.

- › BH is a classic  $n$ -body simulation algorithm that builds a spatial decomposition tree containing center-of-mass information and uses that information to approximate weak forces. We modified the Lonestar implementation to sort the bodies using a space-filling curve and build the octree in parallel, resulting in tree nodes local to the thread that will access the nodes most often.
- › BC<sup>11</sup> computes a metric for nodes in a network that captures each node’s relative importance. We used an outer-loop parallelization in which each thread computes the single-source shortest path and the update for one source node in the graph. Each iteration requires accessing the entire input graph, so we randomly distributed the graph across all NUMA nodes.
- › TRI<sup>12</sup> counts the number of triangles (3-cliques) in a graph. This algorithm is unchanged from the Lonestar implementation. We used a repartitioned graph.

Self-relative scaling

We tested our algorithms at the Pittsburgh Supercomputing Center on Blacklight, an SGI UV NUMA system containing 4,096 cores and 32 terabytes of RAM (our machine allocation was limited to 512 cores). Each NUMA node contains 16 cores running at 2.27 GHz on two processors and 128 GBytes of memory. We compiled using g++ 4.7 at -O3. Table 1 summarizes the inputs



and configurations. We obtained similar results on smaller-scale NUMA systems. Even standard large server-class machines with two to four sockets showed notable improvement with these changes to the Galois runtime system.

As Figure 5 shows, DMR and BH achieve self-relative, strong scaling of 422× and 390×, respectively, at 512 threads. This equates to 82 and 75 percent parallel efficiency for programs written in a sequential programming style. DT scales up to 304× at 512 threads, due in part to memory contention inserting into the lookup-acceleration tree. BC requires reading the entire graph by each iteration. Adding NUMA nodes while simultaneously increasing parallelism increases the average latency of memory accesses for all threads. This causes BC to scale at only about 50 percent efficiency. Although the graph size is small enough to fit in the L3 cache, the temporary data necessary for an outer-loop parallel BC calculation is proportional to the graph's size, so in actual parallel execution the graph could not remain in cache. TRI also scales at only about 50 percent efficiency.

### Runtime comparisons

As Table 2 shows, serial runtime performance of our implementations of these algorithms compares favorably to third-party implementations.<sup>1,3,13,14</sup> Our goal is not necessarily to have the best-performing serial implementation, especially because some use hand-crafted, problem-specific data structures, but to show that we are within an acceptable margin of custom implementations while using the generic data structures provided by our runtime system. For our serial runs, we use

**TABLE 2.** Serial runtime implementations of the algorithms.

Algorithm	Implementation	No. of threads	Runtime (s)
Delaunay mesh refinement	Triangle	1	96
	Galois	1	155.7
	Galois	512	0.37
Delaunay triangulation	Triangle	1	1,185
	Galois	1	56.6
	Galois	512	0.18
Barnes-Hut	Splash-2	1	>6,000*
	Galois	1	1,386
	Galois	512	3.55
Betweenness centrality	HPCS SSCA	1	6,720
	Galois	1	5,394
	Galois	512	21.6
Triangle finding	GraphLab	2	531
	Galois	1	7.03
	Galois	512	0.028

\*Timed out after 100 minutes

**TABLE 3.** Runtimes of three algorithms at 512 threads for spatial and random partitioning.

Algorithm	Partitioning	Runtime (s)
Delaunay mesh refinement	Spatial	0.37
	Random	0.77
Delaunay triangulation	Spatial	0.18
	Random	0.22
Barnes-Hut	Spatial	3.55
	Random	3.85

the full parallel system including work lists and contention management.

Table 3 compares the runtimes of DMR, DT, and BH at 512 threads for random partitioning and partitioning based on a space-filling curve. All data structures built at runtime, other than the initial graph, maintain their default NUMA-aware allocation patterns. We see that good data placement


is responsible for a 10 to 50 percent improvement in the runtime. For DMR, this is the difference between scaling to 422× as opposed to only 203×.

Irregular algorithms stress both the hardware and support runtimes of modern computing systems. Nevertheless, a carefully designed runtime

## ABOUT THE AUTHORS

**ANDREW LENHARTH** is a research associate in the Institute for Computational Science and Engineering (ICSE) at the University of Texas at Austin. His research focuses on novel uses of compilers in operating systems, computer architecture, and parallel programming. Lenharth received a PhD in computer engineering from the University of Illinois at Urbana-Champaign. He is a member of ACM. Contact him at lenharth@ices.utexas.edu.

**KESHAV PINGALI** is a professor in the Department of Computer Science at the University of Texas at Austin and a member of ICES, where he holds the W.A. “Tex” Moncrief Chair. His research interests include programming languages and tools for multicore processors. Pingali received an ScD in computer science from MIT. He is a Fellow of ACM, IEEE, and the American Association for the Advancement of Science (AAAS). Contact him at pingali@cs.utexas.edu.

system can enable an implicitly parallel programming model to scale to large numbers of processors in a NUMA system. As modern systems become increasingly nonuniform, runtime and algorithm design must adapt by recognizing that although shared memory provides a simple abstraction which hides communication, communication happens nonetheless. The design of large-scale NUMA systems shares many performance considerations with that of distributed systems. However, even on large NUMA machines, closely coupled execution units can benefit from fast shared-memory design. 

### REFERENCES

1. J.R. Shewchuk, “Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator,” *Applied Computational Geometry: Towards Geometric Engineering*, M.C. Lin and D. Manocha, eds., Springer, 1996, pp. 203–222.
2. G. Malewicz et al., “Pregel: A System for Large-Scale Graph Processing,” *Proc. 2010 ACM SIGMOD Int’l Conf. Management of Data (SIGMOD 10)*, 2010, pp. 135–146.
3. Y. Low et al., “GraphLab: A New Framework for Parallel Machine Learning,” *Proc. 26th Conf. Uncertainty in Artificial Intelligence (UAI 10)*, 2010; [www.select.cs.cmu.edu/publications/paperdir/uai2010-low-gonzalez-kyrola-bickson-guestrin-hellerstein.pdf](http://www.select.cs.cmu.edu/publications/paperdir/uai2010-low-gonzalez-kyrola-bickson-guestrin-hellerstein.pdf).
4. J. Shun and G.E. Blelloch, “Ligra: A Lightweight Graph Processing Framework for Shared Memory,” *Proc. 18th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP 13)*, 2013, pp. 135–146.
5. K. Pingali et al., “The Tao of Parallelism in Algorithms,” *Proc. 32nd ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI 11)*, 2011, pp. 12–25.
6. D. Nguyen, A. Lenharth, and K. Pingali, “A Lightweight Infrastructure for Graph Analytics,” *Proc. 24th ACM Symp. Operating Systems Principles (SOSP 13)*, 2013, pp. 456–471.
7. UPC Consortium, *UPC Language Specifications, V1.2*, tech. report LBNL-59208, Lawrence Berkeley Nat’l Lab, 2005.
8. R. Chandra et al., *Parallel Programming in OpenMP*, Morgan Kaufmann, 2001.
9. D. Nguyen and K. Pingali, “Synthesizing Concurrent Schedulers for Irregular Algorithms,” *Proc. 16th Int’l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS 11)*, 2011, pp. 333–344.
10. M. Kulkarni et al., “Lonestar: A Suite of Parallel Irregular Programs,” *Proc. IEEE Int’l Symp. Performance Analysis of Systems and Software (ISPASS 09)*, 2009, pp. 65–76.
11. L.C. Freeman, “A Set of Measures of Centrality Based on Betweenness,” *Sociometry*, vol. 40, no. 1, 1977, pp. 35–41.
12. T. Schank, “Algorithmic Aspects of Triangle-Based Network Analysis,” PhD dissertation, Univ. of Karlsruhe, 2007.
13. D.A. Bader and K. Madduri, “Design and Implementation of the HPCS Graph Analysis Benchmark on Symmetric Multiprocessors,” *Proc. 12th Int’l Conf. High Performance Computing (HiPC 05)*, 2005, pp. 465–476.
14. S.C. Woo et al., “The SPLASH-2 Programs: Characterization and Methodological Considerations,” *Proc. 22nd Ann. Int’l Symp. Computer Architecture (ISCA 95)*, 1995, pp. 24–36.



Selected CS articles and columns are also available for free at <http://ComputingNow.computer.org>.