Grappa: A Latency-Tolerant Runtime for Large-Scale Irregular Applications

Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, Mark Oskin

University of Washington
Department of Computer Science and Engineering
{nelson, bholt, bdmyers, preston, luisceze, skahan, oskin}@cs.washington.edu

Abstract

Grappa is a runtime system for commodity clusters of multicore computers that presents a massively parallel, single address space abstraction to applications. Grappa's purpose is to enable scalable performance of irregular parallel applications, such as branch and bound optimization, SPICE circuit simulation, and graph processing. Poor data locality, imbalanced parallel work and complex communication patterns make scaling these applications difficult.

Grappa serves both as a C++ user library and as a foundation for higher level languages. Grappa tolerates delays to remote memory by multiplexing thousands of lightweight workers to each processor core, balances load via fine-grained distributed work-stealing, increases communication throughput by aggregating smaller data requests into large ones, and provides efficient synchronization and remote operations. We present a detailed description of the Grappa system and performance comparisons on several irregular benchmarks to hand-optimized MPI code and to the Cray XMT, a custom system used to target the real-time graph-analytics market. We find Grappa to be 9X faster than MPI on a random access microbenchmark, between 3.5X and 5.4X slower than MPI on applications, and between 2.6X faster and 4.4X slower than the XMT.

1. Introduction

Irregular applications exhibit workloads, dependences, and memory accesses that are highly sensitive to input. Classic examples of such applications include branch and bound optimization, SPICE circuit simulation, and car crash analysis. Important contemporary examples include processing large graphs in the business, national security, and social network computing domains. For these emerging applications, reasonable response time – given the sheer amount of data – requires large multinode systems. The most broadly available multinode systems are those built from x86 compute nodes interconnected via Ethernet or InfiniBand. However, scalable performance of irregular applications on these systems is elusive for two reasons:

Poor data locality and frequent communication Data reference patterns of irregular applications are unpredictable and tend to be spread across the entire system. This results in frequent requests for small pieces of remote data. Caches are of little assistance because of low temporal and spatial locality. Prefetching is of limited value because request locations are not known early enough. Data-parallel frameworks such as MapReduce [24] are ineffective because they rely on data partitioning and regular communication patterns. Consequently, commodity networks, which are designed for large packets, achieve just a fraction of their peak bandwidth on small messages, starving application performance.

High network communication latency The performance challenges of frequent communication are exacerbated by high network latency relative to processor performance. Latency of commodity networks runs anywhere from a few to hundreds of microseconds – tens of thousands of processor clock cycles. Since irregular application tasks encounter remote references dynamically during execution and must resolve them before making further progress, stalls are frequent and lead to severely underutilized compute resources.

While some irregular applications can be manually restructured to better exploit locality, aggregate requests to increase network message size, and manage the additional challenges of load balance and synchronization, the effort required to do so is formidable and involves knowledge and skills pertaining to distributed systems far beyond those of most application programmers. Luckily, many of the important irregular applications naturally offer large amounts of concurrency. This immediately suggests taking advantage of concurrency to tolerate the latency of data movement by overlapping computation with communication.

The fully custom Tera MTA-2 [5, 6] system is a classic example of supporting irregular applications by using concurrency to hide latencies. It had a large distributed shared memory with no caches. On every clock cycle, each processor would execute a ready instruction chosen from one of its 128 hardware thread contexts, a sufficient number to fully tolerate memory access latency. The network was designed with a single-word injection rate that matched the processor clock frequency and sufficient bandwidth to sustain a reference from every processor on every clock cycle. Unfortunately, the MTA-2's relatively low single-threaded performance meant that it was not general enough nor cost-effective. The Cray XMT approximates the Tera MTA-2, reducing its cost but not overcoming its narrow range of applicability.

We believe we can support irregular applications with good performance and cost-effectiveness with commodity hardware for two main reasons. First, commodity multicore processors have become extremely fast with high clock rates, large caches and robust DRAM bandwidth. Second, commodity networks offer high bandwidth as long as messages are large enough. We build on these two observations and develop Grappa, a software runtime system that allows a commodity cluster of x86-based nodes connected via an InfiniBand network to be programmed as if it were a single, large, shared-memory NUMA (non-uniform memory access) machine with scalable performance for irregular applications. Grappa exploits fast processors and the memory hierarchy to provide a lightweight user-level tasking layer that supports a context switch in as little as 38ns and can sustain a large number of active workers. It bridges the commodity network bandwidth gap with a communication layer that combines short messages originating from many concurrent workers into larger packets.

As a general design philosophy, Grappa trades latency for throughput. By *increasing* latency in key components of the system we are able to: increase effective random access memory bandwidth by delaying and aggregating messages; increase synchronization rate

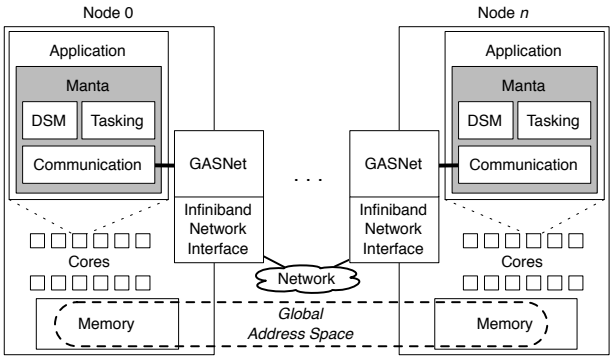


Figure 1: Grappa system overview

by delegating atomic operations to gatekeeper cores, even when referencing node-local global data; and improve load balance via work-stealing. Grappa then exploits parallelism to tolerate the increased latency.

Our evaluation of Grappa shows that its core components provide scalable performance for irregular applications. It can multiplex thousands of workers on a multicore CPU and is limited only by DRAM bandwidth, not latency, to fetch worker state. The communication layer helps Grappa achieve over 1 billion random updates per second across 64 system nodes. Grappa provides performance approaching and in some cases exceeding that of the custom-hardware XMT, as well as MPI and UPC implementations of the same benchmarks. When we compare Grappa to the XMT, we find Grappa to be between 2.6X faster and 4.3X slower depending on the benchmark. When compared with MPI, Grappa is 9X faster on a random access microbenchmark and 3.5X to 5.4X slower on applications. We find that the benchmarks that are faster contain specialized implementations of features Grappa provides generally.

2. Grappa Overview

Grappa's design is motivated by two key observations of today's systems. First, modern compute cores can execute hundreds of thousands of instructions in the few microseconds that it takes for one remote memory read to complete. Second, the injection rate of modern commodity networks is low and the bandwidth is optimized for large messages. Given the frequent communication caused by high degree of fine-grained random access in irregular applications, it is necessary to both overlap communication with computation as much as possible and to aggregate network messages efficiently. Providing these two properties depend on exploiting large amounts of concurrency. Hence, Grappa's programming model focuses on enabling programmers to easily express concurrency.

Grappa comprises three main software components, shown in Figure 1:

Tasking system The tasking system supports lightweight multithreading to tolerate communication latency and global distributed work-stealing (i.e., tasks can be stolen from any node in the system), which provides automated load balancing. The scheduler oversubscribes to have more worker threads than required for latency tolerance. By keeping at least four workers per core ready to run at all times, the scheduler can prefetch a worker's state into cache to reduce the chance of stalling on memory accesses during a context switch.

Communication layer The main goal of our communication layer is to aggregate small messages into large ones. This process is invisible to the application programmer. Its interface is based on active messages [55]. Since aggregation and deaggregation of messages needs to be very efficient, we perform the process in parallel and carefully use lock-free synchronization operations.

GASNet [15] is used as the underlying mechanism for bulk remote memory reads and writes using active message invocations, with an off-the-shelf user-mode InfiniBand device driver stack [46]. MPI is used for process setup and tear down.

Distributed shared memory The DSM system provides fine-grain access to data anywhere in the system, with delegated operation at the core of its design. Every piece of global memory is owned by a particular core in the system, and all others may only access that memory by delegating their requests to the owning core. It supports normal access operations such as *read* and *write* as well as synchronizing operations such as *fetch-and-add* [29]. Due to delegation, the memory model offered is similar to what underpins C/C++ [14, 35], so it is familiar to programmers. The DSM system design relies on the lightweight tasking system and communication layer in order to offer high aggregate random access bandwidth for accessing remote data.

3. Tasking System

Below we discuss the implementation of our task management support and then describe how applications expose parallelism to the Grappa runtime.

3.1 Task Support Implementation

The basic unit of execution in Grappa is a *task*. When tasks are ready to execute, they are mapped to a *worker,* which is akin to a user-level thread. Each hardware core has a single operating system thread pinned to it.

Tasks Tasks are specified by a closure (or "function object" in C++ parlance) that holds both code to execute and initial state. The functor can be specified with a function pointer and explicit arguments, a C++ struct that overloads the parentheses operator, or a C++11 lambda construct. These objects, typically very small (on the order of 64 bytes), hold read-only values such as an iteration index and pointers to common data or synchronization objects. Task functors can be serialized and transported around the system, and eventually executed by a worker, as described next.

Workers Workers execute application and system (e.g., communication) tasks. A worker is simply a collection of status bits and a stack, allocated at a particular core. When a task is ready to execute it is assigned to a worker, which executes the task functor on its own stack. Once a task is mapped to a worker it stays with that worker until it finishes.

Scheduling During execution, a worker yields control of its core whenever performing a long-latency operation, allowing the processor to remain busy while waiting for the operation to complete. In addition, a programmer can direct scheduling explicitly. To minimize context-switch overhead, the Grappa scheduler operates entirely in user-space and does little more than store state of one worker and load that of another. When a tasks encounters a long-latency operation, its worker is suspended and subsequently woken when the operation completes.

Each core in a Grappa system has its own independent scheduler. The scheduler has a collection of active workers ready to execute called the *ready worker queue*. Each scheduler also has three queues of tasks waiting to be assigned a worker:

deadline task queue a priority queue of tasks that are executed according to task-specific deadline constraints;

private task queue a queue of tasks that must run on this core and is therefore not subject to stealing;

public task queue a queue of tasks that are waiting to be matched with workers. It is a local partition of a shared task pool.

Whenever a task yields, the scheduler makes a decision about what to do next. First, any task in the deadline task queue whose deadline is imminent is chosen for execution. This queue manages high priority system tasks, such as periodically servicing communication requests. Second, the scheduler determines if any workers with running tasks are ready to execute; if so, one is scheduled. Finally, if no workers are ready to run, but tasks are waiting to be matched with workers, an idle worker is woken (or a new worker is spawned), matched with a task, and scheduled.

Context switching Grappa context switches between workers non-preemptively. As with other cooperative multithreading systems, we treat context switches as function calls, saving and restoring only the callee-saved state as specified in the x86-64 ABI [7]. This involves saving six general-purpose 64-bit registers and the stack pointer, as well as the 16-bit x87 floating point control word and the SSE context/status register. Thus, the minimum amount of state a cooperative context switch routine must save, according to the ABI, is 62 bytes.

Since Grappa keeps a very large number of active workers, their context data will not fit in cache. By oversubscribing on the number of workers beyond what is required for local DRAM latency tolerance, the scheduler can ensure there is always some number of context pointers in the ready queue. This allows the scheduler to prefetch contexts into cache using software prefetch instructions; the size of the L1 cache is sufficient to hold enough contexts to tolerate the latency to main memory. Empirically we find that prefetching the fourth worker in the scheduling order is sufficient. This prefetching constrains the types of task scheduling decisions that can be made but makes context switching effectively free of cache misses, even to hundreds of thousands of workers. We provide an analysis of our context switch performance in Section 7.1.

Work stealing When the scheduler finds no work to assign to its workers, it commences to steal tasks from other cores using an asynchronous `call_on` active message. It chooses a victim at random until it finds one with a non-zero amount of work in its public task queue. The scheduler steals half of the tasks it finds at the victim. Work stealing is particularly interesting in Grappa since performance depends on having many active worker threads on each core. Even if there are many active threads, if they are all suspended on long-latency operations, then the core is underutilized. The stealing policy must predict whether local tasks will likely generate enough new work soon; a similar problem is addressed in [54].

3.2 Expressing Parallelism

Grappa programmers focus on expressing as much parallelism as possible without concern for where it will execute. Grappa then chooses where and when to exploit this parallelism, scheduling as much work as is necessary on each core to keep it busy in the presence of system latencies and task dependences.

Grappa provides three methods for expressing parallelism. First, a single task can be created to execute in parallel with the current task by calling `spawn` with a functor. This adds it to the queue of tasks which will be executed the next time a worker is available. Second, the programmer can invoke a parallel for loop with `parallel_for`, provided that the trip count is known at loop entry. The programmer specifies a functor which takes the loop index as a parameter, and an optional threshold to control parallel overhead. Grappa does *recursive decomposition* of iterations, similar to Cilk's cilk_for construct [13], and TBB's `parallel_for` [49]. It generates a logarithmically-deep tree of tasks, stopping to execute the loop body when the number of iterations is below the required threshold. Third, parallelism can be expressed via asynchronous delegate operations, which are explained next, in Section 4.

Figure 2 shows sample code using Grappa for a parallel tree search. Note how the code looks very similar to a recursive search procedure for a shared-memory system, without regard for communication, and Grappa's parallel loop construct allows easy parallelization of the search.

```
class Vertex {
  Key key
  int64_t numChildren;
  GlobalAddress<Vertex> children;
};

void search(GlobalAddress<Vertex> vtx_addr,
      Key key, GlobalAddress<Vertex> result) {
  // blocking remote read to get vertex info
  Vertex vtx = delegate_read(vtx_addr);
  if (vtx.key == key) {
    // key found
    delegate_write(result, vtx);
  } else {
    // spawn stealable tasks for iterations
    parallel_for(0,vtx.numChildren,[=](int i) {
      // recursive search
      search(vtx.children+i, key, result);
    });
  }
}
```

Figure 2: Sample Grappa code illustrating a parallel tree search similar to the unbalanced tree search benchmark we describe later. Children are spread over the system, so each parallel recursive search performs a delegate read to get vertex data.

4. Distributed Shared Memory

Applications written for Grappa utilize two forms of memory: local and global. Local memory is local to a single core within a node in the system. Accesses occur through conventional pointers. Applications use local accesses for a number of things in Grappa: the stack associated with a task, accesses to localized global memory in caches (see below), and accesses to debugging infrastructure local to each system node. Local pointers cannot access memory on other cores, and are valid only on their home core.

Large data that is expected to be shared and accessed with low locality is stored in Grappa's global memory. All global data must be accessed through calls into Grappa's API, shown in Figure 3.

Global memory addressing Grappa provides two methods for storing data in the global memory. The first is a distributed heap striped across all the machines in the system in a block-cyclic fashion. The `global_malloc` and `global_free` calls are used to allocate and deallocate memory in the global heap. Addresses to memory in the global heap use *linear addresses*. Choosing the block size involves trading off sequential bandwidth against aggregate random access bandwidth. Smaller block sizes help spread data across all the memory controllers in the cluster, but larger block sizes allow the locality-optimized memory controllers to provide increased sequential bandwidth. The block size, which is configurable, is typically set to 64 bytes, or the size of a single hardware cache line, in order to exploit spatial locality when available.

Grappa also allows any local data on a core's stacks or heap to be exported to the global address space to be made accessible to other cores across the system. Addresses to global memory allocated

in this way use *2D global addresses*. This uses a traditional PGAS (partitioned global address space [25]) addressing model, where each address is a tuple of a rank in the job (or global process ID) and an address in that process. The lower 48 bits of the address hold a virtual address in the process. The top bit is set to indicate that the reference is a 2D address (as opposed to linear address). This leaves 15 bits for network endpoint ID, which limits our scalability to 2^{15} cores. Any node-local data can be made accessible to other cores in the system by wrapping the address and node ID into a 2D global address. This address can then be accessed with a delegate operation and even be buffered by other cores. The address is converted at the destination into a canonical x86 address by replacing the upper bits with the sign-extended upper bit of the virtual address. 2D addresses may refer to memory allocated from a single processes' heap or from a task's stack. Figure 4 shows how 2D and linear addresses can refer to other cores' memory.

Allocation in the global heap:

```
GlobalAddress<T> global_malloc<T>( size )
global_free( GlobalAddress<T> )
```

Delegate operations:

```
T    delegate_read( GlobalAddress<T>)
Promise<T> delegate_read_async( GlobalAddress<T> )
void delegate_write( GlobalAddress<T>, T value)
void delegate_write_async( GlobalAddress<T>, T value)
bool delegate_cas( GlobalAddress<T>, T cmp, T set)
T    delegate_fetch_inc( GlobalAddress<T>, T inc)
void delegate_inc_async( GlobalAddress<T>, T inc)
```

Figure 3: Grappa API for memory accesses.

Global memory access Access to Grappa's distributed shared memory is provided through *delegate* operations, which are short memory accesses performed at the memory location's home node. When the data access pattern has low-locality, it is more efficient to modify the data on its home core rather than bringing a copy to the requesting core and returning it after modification. Delegate operations [39, 43] provide this capability. Applications can dispatch computation to be performed on individual machine-word sized chunks of global memory to the memory system itself. Delegates can execute arbitrary code, provided they do not block, to ensure communicator workers make progress. Provided they touch only memory owned by a single core, we can use them to perform simple *read/write* operations to global memory, as well as more complex *read-modify-write* operations (e.g., *fetch-and-add*). We use these primitive operations to implement higher-level synchronization mechanisms such as mutexes, condition variables, and full-empty bits.

Delegate operations are *always* executed at the home core of their address. The remote operation may not perform any operations that could cause a context switch; this ensures any modifications are atomic. We limit delegate operations to operate on objects in the 2D address space or objects that fit in a single block of the linear address space so they can be satisfied with a single network request. Given these restrictions, we can ensure that delegate operations for the same address from multiple requesters are always serialized through a single core in the system, providing atomic semantics without using actual atomic operations (and thus avoiding their typical high cost).

Delegate operations can be either *blocking* or *asynchronous*. With blocking operations, the task issuing the delegate call blocks until the delegate operation completes, which is necessary, for example, to ensure that synchronization has finished before continuing. On the other hand, remote data accesses often can overlap, and delegates with no return value may not need to block the caller. To avoid unnecessary waiting, we support asynchronous delegate operations. For

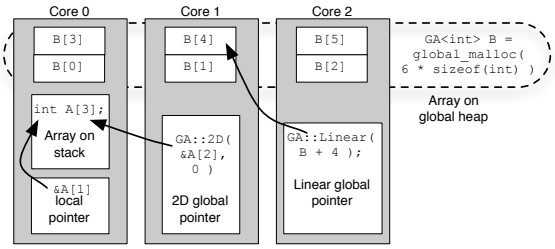


Figure 4: Global memory referencing in Grappa

reads, we support a "futures"-like mechanism which allows tasks to issue reads in parallel and block on the "promises" returned. Delegate write operations may also be performed asynchronously, but synchronization is still needed to ensure that asynchronous operations have completed. Grappa provides a `GlobalCompletionEvent` synchronization object, which asyncronous operations (including tasks) can be enrolled. Tasks can block on these objects to be woken when all enrolled operations are complete.

When programmers want to operate on data structures spread across multiple nodes, accesses must be expressed as multiple delegate operations along with with appropriate synchronization operations. Grappa's API also includes calls for gathering and scattering contiguous blocks in the global heap, but the user is responsible for ensuring correct synchronization.

Memory consistency model discussion As mentioned earlier, all synchronization operations are done via delegate operations. Since they all execute on their home core in some serial order, they are guaranteed to be globally linearizable [32], with their updates visible to all cores across the system in the same order. In addition, only one synchronous delegate will be in flight at a time from a particular task. Therefore, synchronization operations from a particular task are not subject to reordering. Consequently, all synchronization operations execute in program order and are made visible in the same order to all cores in the system. These properties are sufficient to guarantee a memory model that offers sequential consistency for data-race-free programs [2] (all accesses to shared data are separated by synchronization). This is the memory model that underpins C/C++ [14, 35].

Note, however, that if the application code uses explicit buffers or asynchronous delegates to access shared data, all updates must be published back to the home core before the synchronization operation that protects the data is performed. This is done using release operations on cached regions and using the `GlobalCompletionEvent` object to determine that asynchronous delegates have completed.

5. Communication Support

Grappa's communication support has two layers: user-level messaging interface based on active messages; and network-level transport that supports request aggregation for better communication bandwidth.

Active message interface At the upper (user-level) layer, Grappa implements asynchronous active messages [55]. Each message consists of a function pointer, an optional argument payload, and an optional data payload.

Message aggregation In our experiments the vast majority of upper layer message requests are smaller than 44 bytes. Our measurements confirm manufacturers' published data [15]; with 44-byte packets, the available bisection bandwidth is only a small fraction (3%) of the peak bisection bandwidth. As mentioned earlier, commodity networks including InfiniBand achieves their peak bisection bandwidth

only when the packet sizes are relatively large – on the order of multiple kilobytes. The reason for this discrepancy is the combination of overheads associated with handling each packet (in terms of bytes that form the actual packet, processing time at the card, multiple round-trips on the PCI Express bus and processing on the CPU within the driver stack). Consequently, to make the best use of the network, we must convert small messages into large ones.

Message processing mechanics Since communication is very frequent in Grappa, aggregating and sending messages efficiently is very important. To achieve that, Grappa makes careful use of caches, prefetching, and lock-free synchronization operations.

Each processing core of a system node maintains an array of outgoing message lists. The array size is the number of system cores in the Grappa system. The outgoing message lists and messages are located in a region of memory shared across all cores in a Grappa node (thus enabling cores to peek at each other's message lists). When a task sends a message, it allocates a buffer (typically on its stack), determines the destination system node, and links the buffer into the corresponding linked list.

Each processing core in a given system node is responsible for aggregating and sending the resulting messages from all cores on that node to a set of destination nodes. Each core periodically executes a task responsible for sending messages. This task examines the private (to each core) message lists for each destination node it is responsible for managing and, if the list is long enough or a message has waited past a time-out period, all messages to a given destination system node from that source system node are sent. Aggregating and sending a message involves manipulating a set of shared data-structures (the message lists). This is done using CAS (compare-and-swap) operations to avoid high synchronization costs. Note that we use a per-core array of message lists that is only periodically modified across processor cores after experimentally determining that this approach was faster (sometimes significantly) than a global per-system node array of message lists.

Each node has a region of memory with send buffers where the final aggregated messages are built. These buffers are visible to the network card, and messages are sent with user-mode operations only. When the worker responsible for outbound messages to a given system node has received a sufficient number of message send requests or a timeout is reached, the linked list of messages is walked and messages are copied to a send buffer. This process requires careful prefetching because most of the outbound messages are *not* in the processor cache at this time (recall that a core can be aggregating messages originating from other cores in the same node). Once the send buffer has been formed, it is handed off to GASNet for transfer to the remote system node. RDMA is used if the underlying network supports it.

There are two useful consequences of forming the send buffer at the time of message transmission instead of along the way as individual upper layer message send requests are received. First, as previously mentioned, most of the messages are not in the cache and prefetching is used to run ahead in the linked list of messages in order to avoid cache misses. But once the send buffer is formed, it is in the cache (for the most part). Hence, when it is handed off to GASNet for transfer across the physical wire, the network card can pull the message buffer from the processor cache instead of main memory, which we have found speeds performance. The second consequence of this decision is that we do not need to pre-allocate buffers for all destination nodes in the system, as the buffer can be allocated on the fly. Nevertheless we have found it efficient to build a flow-control-like protocol of outstanding message buffers between pairs of system nodes.

Once the remote system node has received the message buffer, a management task is spawned to manage the unpacking process. The management task spawns a task on each core at the receiving system to simultaneously unpack messages destined for that core. Upon completion, these unpacking tasks synchronize with the management task. Once all cores have processed the message buffer, the management task sends a reply to the sending system node indicating the successful delivery of the messages.

6. Methodology

We implemented the Grappa in C++ for the Linux operating system. The core runtime system system is about 15K lines of code. We ported a number of benchmarks to Grappa as well as collected and optimized a set of comparison benchmarks for XMT, MPI, and UPC [17]. We ran the Grappa, MPI, and UPC experiments on a cluster of AMD Interlagos processors. Nodes have 32 2.1-GHz cores in two sockets, 64GB of memory, and 40Gb Mellanox ConnectX-2 InfiniBand network cards. Nodes are connected via a QLogic InfiniBand switch. We also compare Grappa to a 128-node Cray XMT (3rd generation MTA). Each node consists of a 500-MHz MTA Threadstorm multithreaded processor that supports 128 streams. The machine uses Cray's proprietary SeaStar2 interconnection network.

We use a variety of benchmarks:

Unbalanced tree search in-memory (UTS-Mem) Unbalanced Tree Search (UTS) is a benchmark for evaluating the programmability and performance of systems for parallel applications that require dynamic load balancing [47]. It involves traversing an unbalanced implicit tree: at each vertex, its number of children is sampled from some probability distribution, and this number of new nodes are added to a work queue to be visited. While this benchmark captures irregular, dynamic *computation,* we actually want to evaluate performance of algorithms with irregular *memory* access patterns. Thus we augment UTS by using the existing traversal code to create a large tree in memory, and then we traverse the in-memory tree. In our modified UTS, the timed portion is this traversal of the in-memory tree. This in-memory traversal has no knowledge of the tree structure beforehand.

Breadth-first-search (BFS) This is the primary kernel for the Graph500 benchmark and is what currently determines the ranking of machines on the Graph500 list [30]. As a whole, the Graph500 benchmark suite is designed to bring the focus of system design on data-intensive workloads, particularly large-scale graph analysis problems, that are important among cybersecurity, informatics, and network-understanding workloads. The BFS benchmark builds a search tree containing parent nodes for each traversed vertex during the search. While this is a relatively simple problem to solve, it exercises the random-access and fine-grained synchronization capabilities of a system as well as being a primitive in many other graph algorithms. Performance is measured in *traversed edges per second* (TEPS), where the number of edges is the edges making up the generated BFS tree. With some modifications to the XMT reference version of Graph500 BFS, the XMT compiler can be made to recognize and apply a Manhattan loop collapse, exposing enough parallelism to allow it to scale out to 64 nodes for the problem scales we show. In order to make comparison easier, we do not employ algorithmic improvements for any of these versions, though there are many [10, 57]; this makes our results difficult to compare with published Graph500 results. Grappa can be expected to benefit the same as MPI due to decreased communication.

IntSort This sorting benchmark is taken from the NAS Parallel Benchmark Suite [8, 44] and is one on which the Cray XMT's early predecessor once held the world speed record [1]. The largest problem size, class D, ranks two billion uniformly distributed random integers using either a bucket or a counting sort algorithm, depending on the strengths of the system. Bucket sort executes a greater number of loops, but is able to leverage locality and avoid communication completely in the final phase, ranking within buckets. For these reasons, the MPI reference version and our Grappa implementation

use bucket sort. On the other hand, the Cray XMT cannot take advantage of locality, but has an efficient compiler-supported parallel prefix sum, so it performs best using the counting sort algorithm. The performance metrics for NAS Parallel Benchmarks, including IntSort, are "millions of operations per second" (MOPS). For IntSort, this "operation" is ranking a single key, so it is roughly comparable to "GUPS" or "TEPS."

PageRank This is a common centrality metric for graphs. PageRank is an iterative algorithm with a common pattern of gather, apply, and scatter on the rank of vertex. The algorithm is often implemented by sparse linear algebra libraries, with the main kernel being the sparse matrix dense vector multiply. For the multiply step, Grappa parallelizes over the rows and parallelizes each dot product. PageRank has the fortunate property that the accumulation function over the in-edges is associative and commutative, so they can be processed in any order or in parallel. Rather than the programmer writing the parallel dot product as local accumulations with a final all-reduce step, we simply send streaming increments to each element of the final vector. We compare PageRank to published results for the Trilinos linear algebra library implemented in MPI [48], and multithreaded PageRank for the XMT [9]. For Grappa, we run on a scale 29 graph using the Graph500 generator.

The metric we use is algorithmic time, which means startup and loading of the data structure (from disk) is not included in the measurement. Grappa collects statistics about application behavior (packets sent, context switches, etc) and these are discussed where appropriate.

7. Evaluation

The goal of our evaluation is to understand whether the core pieces of the Grappa runtime system, namely our tasking system and the global memory/communication layer, work as expected and whether together they are able to efficiently run irregular applications. We evaluate Grappa in three basic steps:

- We present results that show that Grappa can support large amounts of concurrency, sufficient for remote memory access and aggregation. The communication layer is able to sustain a very high rate of global memory operations. We also show the performance of a graph kernel that stresses communication and concurrency together.
- We characterize system behavior, including profiling where execution time goes, and how aggregation affects message size and rates.
- Finally, we show how some more realistic irregular workloads on Grappa compare to the Cray XMT and hand-tuned MPI code.

7.1 Basic Grappa Performance

User-level context switching Fast context switching is at the heart of Grappa's latency tolerance abilities. We assess context switch overheads using a simple microbenchmark that runs a configurable number of workers on a single core, where each worker increments values in a large array.

Figure 5 shows the average context switch time as the number of workers grow. At our standard operating point (≈1K workers), context switch time is on the order of 50ns. As we add workers, the time increases slowly, but levels off: we also ran with 500,000 workers (10 times what is shown in the figure) and found that context switch time was around 75ns. In comparison, for the same yield test using kernel-level Pthreads on a single core, the switch time is 450ns for a few threads and 800ns for 1000–32000 threads.

If we calculate aggregate context switch *rate* of all cores in a node, we find that with prefetching, Grappa context switching is limited *not* by memory latency, as normally assumed, but rather

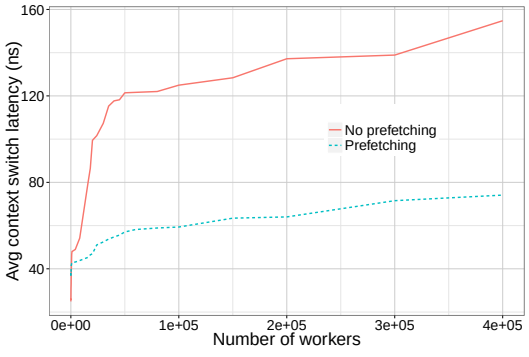


Figure 5: Average context switch time with and without prefetching.

memory bandwidth. Specifically, we empirically found that 4 cache lines (1 for worker struct and 3 for stack data) was sufficient to avoid cache misses in the microbenchmark. Every context switch then requires 8 cache-line transfers. The off-chip bandwidth of a single socket in our system is 270M cache lines per second [41, 43]. This implies that, in the limit, we can sustain at most 34M context switches per second per socket (a context-switch time of 29ns).

In summary, our tasking layer is able to efficiently sustain very high concurrency and, as we will show later, the amount of concurrency sustained is sufficient for the latencies Grappa needs to hide.

Global memory and communication We measure the performance of Grappa's global memory and communication layers using a faithful implementation of the giga updates per second (GUPs) benchmark, which measures cluster-wide random access bandwidth. Read-modify-write updates are dispatched at random to a global large array. This benchmark stresses the communication layer of Grappa separately from the scheduler, because only a single worker is used per system node. Figure 6 shows that Grappa is able to sustain well over a billion updates per second with 64 nodes. Note also that when aggregation is turned off, the update rate is nearly flat. Clearly aggregation is instrumental for good communication performance.

This compares very favorably to published results [34] for other high-end HPC systems. Though the actual computation done by GUPS is not useful, irregular, data-intensive applications typically must be able to sustain a high rate of random accesses in order to, for example, visit and mark vertices during a graph traversal. High random access rate in a distributed setting has been a long-standing challenge in HPC.

Putting it all together with Unbalanced Tree Search (UTS) Figure 7 shows the overall performance of Grappa running UTS. This experiment demonstrates that Grappa's context switching and communication layers can be used together, while balancing workload, to run an irregular application efficiently.

Visiting vertices in the distributed tree requires mostly remote accesses, and because each vertex in the tree must be visited before it can be expanded, blocking remote reads are required. In this case, we are forced to context switch to tolerate the remote access and continue aggregating. Figure 2 shows a closely analogous tree search in Grappa.

We look at two classes of trees, T1 and T3, from the original benchmark. T1 trees are very shallow and wide (i.e., significant parallel slack [53]), while T3 trees are very deep (i.e., little parallel slack). Given that access to each vertex is a random access, the critical

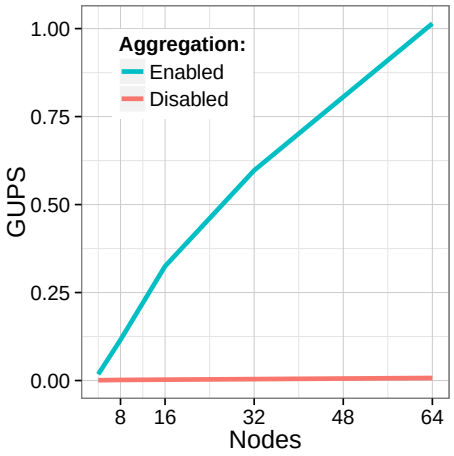


Figure 6: GUPS (giga updates per second) for Grappa as the number of nodes grows with and without message aggregation.

path to search T3 trees is very long, hence the low performance and scalability. On such trees, we do not expect there to be sufficient concurrency for any system, including Grappa, to achieve high throughput – at the 16-node data point, the average active tasks per core over the search is an order of magnitude larger for T1 than for T3. Given the lack of parallelism, scaling up only serves to reduce throughput by distributing the tree to more machines. On the other hand, Grappa performs and scales very well for T1 trees.

We compared this to a in-memory modification of the UPC implementation of UTS [47], where each core hosts a UPC "THREAD." Berkeley UPC (BUPC) is built upon GASNet, like Grappa, but it uses RDMA transfers directly and the compiler overlaps remote communication when possible. We expect BUPC to perform similarly on the T3 tree, where there is little parallelism to overlap communication and computation, and no better than T3 on T1, given the inability to overlap vertices on each core without multithreading (a multithreaded variant of UPC has been evaluated in [42]). Our experiments confirm that BUPC achieves lower throughput than Grappa for both T1 and T3 trees.

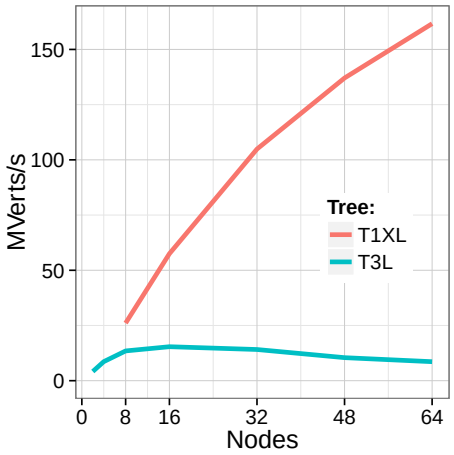


Figure 7: Vertices per second in UTS on Grappa as the number of nodes grows.

7.2 Comparing Grappa to Other Systems

In order to put Grappa's performance into a general context, we compare it with XMT running BFS, PageRank, IntSort, GUPS, and UTS. Since XMT is a different hardware platform, we also compare Grappa with optimized MPI versions of BFS and GUPS running on the same hardware. We run all experiments with 64 nodes and 16 cores per node.

	Grappa	XMT	MPI
GUPS	1	2.23	0.11
BFS	1	1.63	3.52
IntSort	1	3.59	5.36
UTS (T1)	1	0.38	–
Pagerank	1	4.35	4.87

Table 1: Comparing Grappa with XMT and optimized MPI. Numbers are presented as throughput on 64 nodes normalized to Grappa.

Table 1 shows the results. Grappa is able to provide performance approaching and sometimes exceeding that of the other systems. This data shows that, while Grappa is good at the operations for which it was designed, there is a cost to providing Grappa's generality. In general, the benchmarks whose MPI implementations are faster than Grappa include a implementation of a subset of Grappa's functionality specialized for their particular problem. We now discuss each benchmark in more detail.

UTS It is perhaps not surprising that the XMT is faster than Grappa on many of the benchmarks; after all, the XMT is custom hardware optimized for irregular applications. The XMT can context-switch every cycle, while Grappa takes 50ns. The XMT can complete 100 million random-access remote reads per second per node in hardware, while Grappa must spend many instructions managing and aggregating each of these reads in software.

What is more surprising is that Grappa is able to exceed the XMT's performance on UTS. This is a demonstration of the inflexibility of custom hardware. Recall that the UTS benchmark searches an unbalanced tree and spawns a data-dependent number of child tasks at each vertex. The XMT's spawn/join semantics, whose implementation is spread across the hardware, OS, and runtime, require a task visiting a vertex to block until all its child tasks have completed, even though the benchmark does not require those tasks to have completed until the end of the tree search. Furthermore, each new task is immediately made available to all other processors in the system, even though it is likely the processor that spawned a task will end up executing it.

Since Grappa's tasks are implemented entirely in software, it is easy to support arbitrary synchronization semantics. Grappa allows both XMT-like per-vertex task joins as well as whole-problem joins that are a better fit for UTS. Grappa's work-stealing distributed task queue encourages locality in task execution so that the system is optimized for the common case of a processor executing a task it spawned. Furthermore, Grappa's parallel loop granularity and buffering features provide a disciplined way for the programmer to easily take better advantage of spatial locality when it oviously exists, specifically when reading a vertex's edgelist.

BFS The Grappa and MPI BFS implementations are quite similar. They both use the same graph representation, and they both depend on aggregation for performance. The difference is that the BFS aggregation is integrated with the traversal code and specialized for the problem: since the only messages required are edge traversals, the aggregation buffers store only pairs of 8-byte vertex IDs. In contrast, Grappa pays execution and space overhead in order to support aggregation of arbitrary messages; the messages Grappa aggregates

include the two 8-byte vertex IDs along with 8 bytes of deserialization information as well as 8 bytes of synchronization information. Furthermore, the MPI implementation writes the edge information directly into the aggregation buffer while Grappa executes additional code to serialize and deserialize the messages.

GUPS The MPI GUPS implementation includes an implementation of aggregation, but it is not optimized to work when the aggregated data exceeds cache, and it has limited support for concurrent communication with multiple destinations. Grappa benefits from being optimized for both of these cases.

Pagerank The MPI Pagerank implementation is built on top of the highly-optimized sparse matrix support in the Trilinos [33] library. In contrast, the Grappa implementation is a straightforward nested loop.

IntSort Both MPI and Grappa IntSort implement a bucket sort. For Grappa, distributing keys into buckets accounts for the bulk of the execution time; Grappa does this using asynchronous delegate operations to write the keys directly to their final destination. The MPI implementation, on the other hand, first does a local sort of the keys on each core, and then uses an `MPI_Alltoallv()` collective operation to move all the keys to their destination nodes in bulk. This is essentially specialized aggregation combined with collective communication. When we measure the time spent in this region of both benchmarks, we find that the the sort and `MPI_Alltoallv()` are $5\times$ faster than the Grappa version using generic aggregation and point-to-point communication. An interesting future direction would be to extend Grappa to support aggregation via collective operations.

Summary Overall, Grappa provides a general programming model at a moderate performance cost. While Grappa's core functionality performs well, applications can specialize similar functionality for their problems and obtain better performance than Grappa on the same hardware. This is, however, not the end of the story for Grappa's performance; Grappa is a young library and, as discussed in the next section, is limited by its implementation rather than the hardware on which it runs. We believe there are opportunities for optimization that will improve its performance.

7.3 Characterization

	GUPS	BFS	IntSort	UTS	Pagerank
App. message rate/core (K/s)	984	983	732	411	474
Avg. App. message bytes	31.8	33.9	30.6	47.3	45.5
Network BW per node (MB/s)	478	511	343	298	332
Avg. Network message bytes	23.2K	4.3K	12.8K	4.2K	3.0K
Avg. active tasks/core	0.9	58.2	0.9	326.2	429.3
Max. active tasks/core	1	128	1	507	1024
Avg. ready queue length/core	2.3	6.1	1.3	186.1	300.4
Avg. Ctx switch rate/core (K/s)	34.2	539	127	543	336
Steal attempts/core/s	0	0	0	54.4	0
Steal successes/core/s	0	0	0	17.0	0

Table 2: Internal runtime metrics, for 64-node, 16-core-per-node benchmark runs

Runtime metrics Table 2 show a number of internal runtime metrics collected while executing the benchmarks from the previous section. These are per-core averages computed over all 1024 cores of the 64-node, 16-core-per-node jobs.

The first group of four metrics relate to the communication layer. Application messages refer to those issued by the user code; network message refer to the aggregated packets sent over the wire. The data show that most application messages are only a handful of bytes, but our aggregator is able to turn them into packets of many kilobytes.

The next group of five metrics relate to the scheduler. We show the average and maximum number of concurrently-executing tasks, along with the average length of the ready queue and the average context switch rate. The last two lines shows the rate of work-stealing from other cores. Only UTS depends on work-stealing for performance; the other applications exploit locality by binding tasks to specific cores Nevertheless, even in UTS, steals are an infrequent occurrence and account for a small fraction of the execution time.

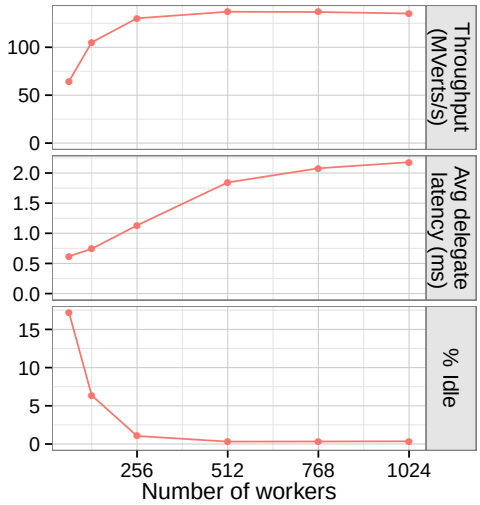


Figure 8: Throughput, request latency, and idleness as number of concurrent workers is varied

How much concurrency does Grappa require? How much latency does it add? Grappa depends on concurrency to cover the latency of aggregation and remote communication. How much is required for good performance?

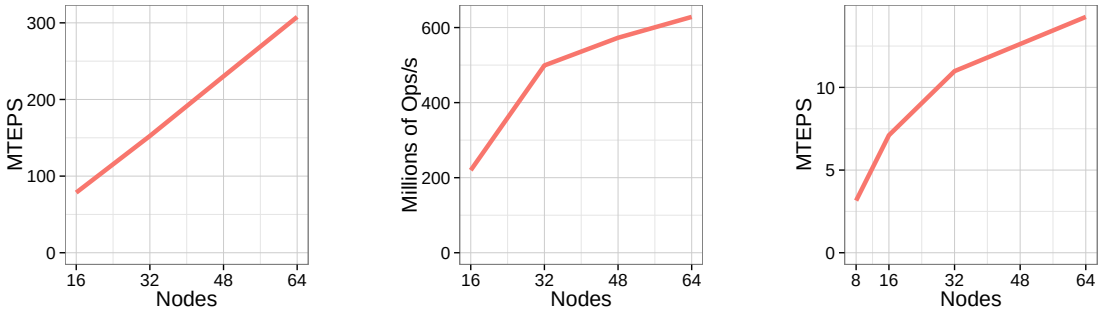
Figure 8 shows a 48-node, 16-core-per-node run of UTS, varying the number of concurrently executing tasks on each core. The top pane shows the overall throughput of the tree search. The middle pane shows average blocking delegate operation latency in microseconds. The bottom pane shows idle time; that is, the fraction of the time the scheduler could not find a ready task to run.

We can observe three things from this figure. First, above 512 concurrently executing tasks per core, idle time is practically zero: these tasks generate requests fast enough to cover the latency of aggregation and communication. This matches the results seen in the throughput plot; throughput peaks at 512 workers and gradually decreases after that due to the overhead of unnecessary context switches. Finally, we see that with 512 workers, the average per-request latency is 1.8ms.

Does Grappa scale? Figures 9, 6, and 7 show scaling out to 64 nodes. Grappa scales best on BFS and worst on IntSort, but is in general able to make use of more nodes. Unfortunately memory limitations in our current network library keep us from exploring scaling beyond 64 nodes; this will be addressed in future work.

In the limit, aggregation does not scale: the time it takes to aggregate enough random requests to build a buffer of reasonable size scales with the size of the cluster. However, we believe our current approach will work for clusters with hundreds of nodes. In the future, we will explore hierarchical, collective techniques to aggregate requests from multiple nodes as well; we believe this can apply to clusters with thousands of nodes.

What limits Grappa's performance? The most common operation in Grappa is sending a message. Three key operations occur in a



(a) BFS (b) IntSort (c) Pagerank

Figure 9: Scaling for BFS, IntSort, and Pagerank. Refer to Figure 6 for GUPS scaling and Figure 7 for UTS scaling

message's lifetime: creating and enqueuing a message, serializing a message in the aggregator, and deserializing a message at the destination. We benchmarked each of these steps individually to shed light on the mechanism.

Minimum-sized messages can be created and enqueued at a rate of 16M/s, serialized into an aggregation buffer at 32M/s, and deserialized from an aggregation buffer at 210M/s. Together, these rates limit us to 10M/s per core in the best case. In practice, Grappa does not achieve this maximum; the overhead of cross-core communication in the current aggregator design limits our message rate to to 1M/s per core with messages of common sizes. Future work will reduce this cost.

8. Related Work

Multithreading Grappa uses multithreading to tolerate memory latency. This is a well known technique. Hardware implementations include the Denelcor HEP [51], Tera MTA [6], Cray XMT [27], Simultaneous multithreading [52], MIT Alewife [3], Cyclops [4], and even GPUs [26].

Grappa's closest ancestor is the Threaded Abstract Machine [22]. This was a software runtime system designed as a prototyping platform for dataflow execution models on distributed memory supercomputers, and contained support for inter-node communication and management of the memory hierarchy as well as context switching and scheduling computation. The Active Messages [55] work that grew out of this project inspired our communication layer. One of the conclusions of this work [23] was that context switch costs can be low only when contexts are in cache, and that latency tolerance was not sufficient to guarantee performance on commodity processors. Grappa demonstrates that times have changed: modern commodity processors have sufficient bandwidth and prefetch capacity to stream contexts from DRAM and sustain a very large number of active contexts.

Grappa implements its own software-based multithreading with a lightweight user-mode task scheduler to multiplex *thousands* of tasks on a single processing core. The large number of tasks is required to tolerate the very high inter-node communication latency of commodity networks. Grappa's task library employs several optimizations: an extremely fast task switch, a small task size, and judicious use of software prefetch instructions to bring task state into the cache sufficiently long before that task is actually scheduled. The main difference between Grappa's support for lightweight threads and prior work such as QThreads [56] and Capriccio [11] is context prefetching, which is needed for good performance when multiplexing such a large number of tasks.

Software distributed shared memory. The goal of providing a shared memory abstraction for a distributed memory system goes back nearly 30 years. Much of the innovation in SDSM has occurred around reducing the synchronization cost of doing updates. The first DSM systems, including IVY [37], used frequent invalidations to provide sequential consistency, which imposed significant communication cost for write-heavy workloads. Later systems supported more relaxed consistency models, including release consistency, which reduced the cost by allowing updates to be buffered until synchronization occurred. Furthermore, multiple writer protocols that sent only modified data were developed to help combat false sharing. The Munin [12, 18] and TreadMarks [36] systems exploited both of these ideas, but some coherence overhead was still required. In contrast, Grappa's delegate-based approach to updates avoids synchronization overhead entirely, providing sequential consistency for data-race-free programs without the cost of a full coherence protocol.

Another way in which SDSM systems differ is in the granularity of access control. Many SDSM systems, including IVY and TreadMarks, tracked ownership at a granularity of a kilobyte or more. There are two main justifications for this design choice: first, networks are more efficient with large packets, and second, multi-kilobyte granularity allowed systems to reuse the processor's paging mechanisms to accelerate access control checks and to provide shared memory transparently. Unfortunately, this meant that these systems depended on a fair ammount of locality to amortize the cost of moving these large blocks, and the systems were very susceptible to false sharing. Other systems, including Munin and Blizzard [50], allowed tracking ownership with variable granularity to address these problems. Grappa's delegate-based approach is similar; since updates are always performed at data's home node, only modified data needs to be moved.

SDSM systems often required extensive modifications to the system software stack, including the OS (IVY, Blizzard) and compilation infrastructure (Munin). This made these systems difficult to port to new platforms. Grappa follows the lead of TreadMarks and provides SDSM entirely at user-level through a library and runtime.

In summary, while Grappa's DSM system is conceptually similar to prior work, we accept the random access aspect of irregular applications and optimize for throughput rather than low latency. Our DSM system must support enough concurrency to tolerate the latency of the network and additional latency overhead impose by the runtime system.

Partitioned Global Address Space languages. The high-performance computing community has largely discarded the coherent distributed shared memory approach in favor of the Partitioned Global Address Space (PGAS) model. Example include Split-C [21], Chapel [19],

X10 [20], Co-array Fortran [45] and UPC [25]. Grappa shares many parts of its design philosophy with these languages.

There are two key ideas Grappa draws from PGAS languages. First, PGAS languages implement DSM at the language, rather than the system level. This allows for a number of optimizations, including efficient split-phase access, variable data granularity, and support for user-customizable synchronization operations. Grappa follows this approach for the same reason.

Second, in PGAS languages, each piece of data has a single canonical location on a particular node. Grappa takes the same approach. However, PGAS languages often expect programmers to modify algorithms to take advantage of locality by processing node-local data as much as possible; access to data stored on other nodes is possible but is seen as something to avoid if possible. Grappa takes the opposite view, as it optimizes for random access to data anywhere in the cluster, and exploiting any available locality in the application is a secondary concern.

Most PGAS languages adopt a SPMD programming model: the programmer must reason about what is happening on every node in the system. Chapel is the exception: it provides a global view of control, while allowing programmers to direct individual cores when necessary. Grappa follows the same approach as Chapel, providing a single-machine abstraction to the programmer along with support for controlling the locality of computation when requested.

Distributed graph processing systems. While Grappa is a general runtime system for any large-scale concurrent application, it performs especially well on graph analysis. Other distributed graph processing frameworks include Pregel [40] and Distributed GraphLab [38]. Pregel adopts a bulk-synchronous parallel (BSP) execution model, which makes it inefficient on workloads that could prioritize vertices. GraphLab overcomes this limitation with an execution model that schedules vertex computations individually, allowing prioritization, which gives faster convergence in a variety of iterative algorithms. GraphLab, however, imposes a rigid model that requires programmers to express computation as transformations on a vertex and its edge list, with information only from adjacent vertexes. Pregel is only slightly less restrictive, as the input data can be any vertex in the graph. Grappa also supports dynamic parallelism with asynchronous execution, but parallelism is expressed as tasks or loop iterations, which is a far more general programming model for irregular computation tasks. PowerGraph [28] improves the performance of GraphLab for real-world graphs with power-law degree distributions by using a vertex cut for graph partitioning. Algorithmic transformations like this would also improve the performance of graph applications on Grappa.

While the bulk-synchronous MapReduce [24] model and related systems such as Hadoop [31] are not a good fit for irregular or graph problems, the ideas have been extended to support a subset of related problems that require iteration, including some graph-based machine learning applications. HaLoop [16] and Spark [58] are two examples of this sort of system. This class of systems is generally characterized by having a restricted programming model and restricted interface to access data, focused on streaming data from disk. In particular, direct communication between nodes in the cluster is not supported; communications must happen through the parallel data structures provided. Grappa is general enough to run these jobs, but the MapReduce-derived systems have much more complete IO support and the ability to tolerate from node failures.

9. Conclusion

Irregular computations are both important and challenging to execute quickly. Scaling these applications easily on commodity hardware has been a historical challenge. Grappa simplifies this task for software developers and compiler writers. Grappa's key aspect is extreme latency tolerance, which not only hides network latency but also enables the system to spend time on sophisticated work stealing and network optimizations, trading latency for even more throughput.

Our evaluation of Grappa shows that the core components, scheduling and communication, achieve their design goals. Thousands of workers can be efficiently context switched on a multicore processor, limited by DRAM bandwidth. Aggregating messages enables Grappa to achieve over 1.0 GUPS on 64 nodes. We also explored four other algorithms: unbalanced tree search, breadth first search, PageRank, and integer sort, comparing performance of these algorithms on Grappa, the Cray XMT, and optimized MPI implementations. Grappa is 9X faster than MPI on GUPS, 3.5X to 5.4X slower than MPI on applications, and between 2.6X faster and 4.3X slower than the XMT. When Grappa is slower than MPI, we find that the MPI implementation includes a specialized implementation of a feature Grappa provides generally to all applications. This generality comes at a performance cost but eases the application developer's task. Moreover, there are significant tuning opportunities the Grappa runtime system. When cost-performance is considered, Grappa on mass-market x86-based clusters is a highly attractive option.

Acknowledgements

This work was funded by Pacific Northwest National Laboratory, NSF Grant CCF-1335466, and a gift from NetApp.

References

[1] Tera MTA Computer Sets New World Speed Record. Mainframe Computing, 1999.

[2] Sarita V. Adve and Mark D. Hill. Weak ordering – A new definition. In *ISCA-17*, 1990.

[3] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Ken Mackenzie, and Donald Yeung. The MIT Alewife machine: Architecture and performance. In *22nd Annual International Symposium on Computer Architecture*, June 1995.

[4] George Almási, Călin Caşcaval, José G. Castaños, Monty Denneau, Derek Lieber, José E. Moreira, and Henry S. Warren, Jr. Dissecting Cyclops: A detailed analysis of a multithreaded architecture. *SIGARCH Computer Architecture News*, 31:26–38, March 2003.

[5] Gail Alverson, Robert Alverson, David Callahan, Brian Koblenz, Allan Porterfield, and Burton Smith. Exploiting heterogeneous parallelism on a multithreaded multiprocessor. In *Proceedings of the 6th international conference on Supercomputing*, ICS '92, pages 188–197, New York, NY, USA, 1992. ACM.

[6] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *Proceedings of the 4th International Conference on Supercomputing*, ICS '90, pages 1–6, New York, NY, USA, 1990. ACM.

[7] AMD64 ABI. `http://www.x86-64.org/documentation/abi-0.99.pdf`, July 2012.

[8] David H. Bailey, Eric Barszcz, John T. Barton, David S. Browning, Russell L. Carter, Rod A. Fatoohi, Paul O. Frederickson, Thomas A. Lasinski, Horst D. Simon, V. Venkatakrishnan, and Sisira. K. Weeratunga. The NAS parallel benchmarks. Technical report, The International Journal of Supercomputer Applications, 1991.

[9] Brian W. Barrett, Jonathan W. Berry, Richard C. Murphy, and Kyle B. Wheeler. Implementing a portable multi-threaded graph library: The MTGL on Qthreads. In *IEEE International Symposium on Parallel Distributed Processing*, pages 1–8, 2009.

[10] Scott Beamer, Krste Asanovi, and David Patterson. Direction-optimizing breadth-first search. In *Conference on Supercomputing (SC-2012)*, November 2012.

[11] Rob Von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Capriccio: Scalable threads for internet services. In

In Proceedings of the 19th ACM Symposium on Operating Systems Principles, pages 268–281. ACM Press, 2003.

[12] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the Second ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPOPP '90, pages 168–176, New York, NY, USA, 1990. ACM.

[13] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM.

[14] Hans-J. Boehm. A Less Formal Explanation of the Proposed C++ Concurrency Memory Model. C++ standards committee paper WG21/N2480 = J16/07-350, `http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2007/n2480.html`, December 2007.

[15] Dan Bonachea. GASNet Specification, v1.1, 2002.

[16] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. Haloop: Efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3(1-2):285–296, September 2010.

[17] William W. Carlson, Jesse M. Draper, David E. Culler, Kathy Yellik, Eugene Brooks, and Karen Warren. Introduction to UPC and language specification, 1999.

[18] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, pages 152–164, New York, NY, USA, 1991. ACM.

[19] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel programmability and the Chapel Language. *International Journal of High Performance Computing Application*, 21(3):291–312, August 2007.

[20] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 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*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM.

[21] David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel programming in Split-C. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, Supercomputing '93, pages 262–273, New York, NY, USA, 1993. ACM.

[22] David E. Culler, Seth Copen Goldstein, Klaus Erik Schauser, and Thorsten von Eicken. TAM – A compiler controlled threaded abstract machine. *Journal of Parallel and Distributed Computing*, 18:347–370, July 1993.

[23] David E. Culler, Klaus E. Schauser, and Thorsten von Eicken. Two fundamental limits on dataflow multiprocessing. In *Proceedings of the IFIP WG10.3. Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, PACT '93, pages 153–164, Amsterdam, The Netherlands, The Netherlands, 1993. North-Holland Publishing Co.

[24] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, January 2008.

[25] Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. *UPC: Distributed Shared Memory Programming*. John Wiley and Sons, Inc., Hoboken, NJ, USA, 2005.

[26] Kayvon Fatahalian and Mike Houston. A closer look at GPUs. *Communications of the ACM*, 51:50–57, October 2008.

[27] John Feo, David Harper, Simon Kahan, and Petr Konecny. Eldorado. In *Proceedings of the 2nd Conference on Computing Frontiers*, CF '05, pages 28–34, New York, NY, USA, 2005. ACM.

[28] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association.

[29] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir. The NYU Ultracomputer: Designing an MIMD shared memory parallel computer. *IEEE Transactions on Computers*, C-32(2):175 –189, February 1983.

[30] Graph 500. `http://www.graph500.org/`, July 2012.

[31] Hadoop. Hadoop website `http://hadoop.apache.org`.

[32] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[33] Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley. An overview of the Trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, 2005.

[34] HPCC. HPCC random-access benchmark `http://icl.cs.utk.edu/hpcc/hpcc_results.cgi`.

[35] ISO/IEC JTC1/SC22/WG21. ISO/IEC 14882, Programming Language, C++ (Committee Draft). `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2800.pdf`, 2008.

[36] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. TreadMarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the USENIX Winter 1994 Technical Conference*, WTEC'94, pages 115–131, Berkeley, CA, USA, 1994. USENIX Association.

[37] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, November 1989.

[38] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *PVLDB*, 2012.

[39] Roberto Lublinerman, Jisheng Zhao, Zoran Budimlic, Swarat Chaudhuri, and Vivek Sarkar. Delegated isolation. In *OOPSLA'11*, pages 885–902, 2011.

[40] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[41] Anirban Mandal, Rob Fowler, and Allan Porterfield. Modeling memory concurrency for multi-socket multi-core systems. In *Proceedings of the 2010 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS2010)*, pages 66–75, White Plains, NY, March 2010. IEEE.

[42] Seung-Jai Min, Costin Iancu, and Katherine Yelick. Hierarchical workstealing on manycore clusters. In *Fifth Conference on Partitioned Global Address Space Programming Models*, October 2011.

[43] Jacob Nelson, Brandon Myers, A. H. Hunter, Preston Briggs, Luis Ceze, Carl Ebeling, Dan Grossman, Simon Kahan, and Mark Oskin. Crunching large graphs with commodity processors. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Parallelism*, HotPar'11, pages 10–10, Berkeley, CA, USA, 2011. USENIX Association.

[44] NAS parallel benchmark suite 3.3. `http://www.nas.nasa.gov/publications/npb.html`, 2012.

[45] Robert W. Numrich and John Reid. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, August 1998.

[46] OpenFabrics Alliance. `https://www.openfabrics.org/index.php`, July 2012.

[47] Stephen Olivier, Jun Huan, Jinze Liu, Jan Prins, James Dinan, P. Sadayappan, and Chau-Wen Tseng. UTS: An unbalanced tree search benchmark. In *Proceedings of the 19th International Conference on Languages and Compilers for Parallel Computing*, LCPC'06, pages 235–250, Berlin, Heidelberg, 2007. Springer-Verlag.

[48] Steven J. Plimpton and Karen D. Devine. MapReduce in MPI for large-scale graph algorithms. *Parallel Comput.*, 37(9):610–632, September 2011.

[49] James Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly Media, 2007.

[50] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain access control for distributed shared memory. In *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, ASPLOS VI, pages 297–306, New York, NY, USA, 1994. ACM.

[51] Burton J. Smith. Architecture and applications of the HEP multiprocessor computer system. In *Proceedings of SPIE 0298, Real-Time Signal Processing IV, 241*, volume 298, pages 241–248, July 1982.

[52] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, ISCA '95, pages 392–403, New York, NY, USA, 1995. ACM.

[53] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.

[54] Rob V. van Nieuwpoort, Thilo Kielmann, and Henri E. Bal. Efficient load balancing for wide-area divide-and-conquer applications. In *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, PPoPP '01, pages 34–43, New York, NY, USA, 2001. ACM.

[55] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, pages 256–266, New York, NY, USA, 1992. ACM.

[56] Kyle B. Wheeler, Richard C. Murphy, and Douglas Thain. Qthreads: An API for programming with millions of lightweight threads. In *IPDPS*, pages 1–8. IEEE, 2008.

[57] Andy Yoo, Edmond Chow, Keith Henderson, William McLendon, Bruce Hendrickson, and Ümit Çatalyürek. A scalable distributed parallel breadth-first search algorithm on BlueGene/L. In *Proceedings of the ACM/IEEE 2005 Conference on Supercomputing*. IEEE, 2005.

[58] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.