Grappa: Scalable Computing on Commodity Hardware for Irregular Applications

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

University of Washington

Submitted for publication, please keep confidential.

Abstract

Grappa is a latency tolerant runtime system for commodity clusters of multicore computers. Grappa's goal is to provide scalable performance for irregular parallel applications. These applications are of growing importance as interactive analysis of large data sets emerges as a commercial and government priority, typically in the form of graph processing. Grappa serves both as a C++ user library and as a foundation upon which higher level languages can be developed or adapted to support development of irregular parallel applications. Grappa tolerates the delays to distant memory by multiplexing thousands of lightweight tasks to each processor core; balances load via fine-grained distributed work-stealing; takes full advantage of network characteristics by aggregating smaller requests into large ones; and provides efficient synchronization and remote operations across its single shared address space. We present a detailed description of the Grappa system, programming examples using the library interface, and compare performance for several irregular benchmarks to the Cray XMT, a custom system used to target the real time graph analytics market.

1. Introduction

Irregular applications generate tasks with work, interdependences, or memory accesses that are highly sensitive to input. Classic examples of irregular applications include branch and bound optimization, SPICE circuit simulation, contact algorithms in car crash analysis, and network flow. Important contemporary examples include processing large graphs in the business, national security, machine learning, data-driven science, and social network computing domains. For these emerging applications, fast response – given the sheer amount of data – requires multinode systems. The most broadly available multinode systems are those built from x86 commodity computing nodes interconnected via ethernet or infiniband. Our goal is to enable scalable performance of irregular applications on these mass market systems.

Our goal is challenging for two key reasons:

Irregular applications exhibit little spatial locality. It is not atypical for any given task's data references to be spread randomly across the entire memory of the system. This makes current memory hierarchy features ineffective. Caches are of little assistance with such low data re-use and spatial locality. Commodity prefetching hardware is effective only when addresses are known many cycles before the data is consumed or the accesses follow a predictable pattern, neither of which occurs in irregular applications. As a consequence, commodity microprocessors stall often when executing irregular applications.

Irregular applications frequently request small amounts of off-node data. On multinode systems, the challenges presented by low locality are analogous, and exacerbated by the increased latency of going off-node. Irregular applications also present a challenge to mass market network technology, which is designed to transfer large blocks of data, not the word-sized references emitted by irregular application tasks. Injection rate into the network is insufficient to utilize wire bandwidth when blocks are below about two-kilobytes, so any straightforward communication strategy severely under-utilizes the network.

While some irregular applications can be restructured to better exploit locality, aggregating requests to increase message size, and manage the additional challenges of load balance and synchronization across multinode systems, the work to do so is formidable and requires knowledge and skills pertaining to distributed systems far beyond those of most application programmers.

Luckily, many of the important irregular applications (e.g., graph processing, our focus in this paper) naturally offer large amounts of concurrency. This immediately suggests taking advantage of concurrency to tolerate the latency of data movement. The fully custom Tera MTA-2 [3] 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 hide 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, while an excellent match to extremely irregular applications, the MTA was not cost-effective on applications that could exploit locality and had very poor single-thread performance, making it a commercial failure. The Cray XMT approximates the Tera MTA-2, substantially reducing its cost but not overcoming its narrow range of applicability.

In the twenty years that have elapsed since the Tera MTA, commodity microprocessors have become much faster and multicore has driven down the absolute price of computation; commodity network price-performance has improved as well. This shift has afforded us the opportunity to attack the challenges posed by irregular applications by emulating in software and inexpensive mass market hardware, the approach taken by Tera. We exploit the increased aggregate instruction rate per socket relative to chip bandwidth, using what would otherwise be wasted instructions to manage the multiplexing of as many as several thousand tasks per core, thus

tolerating memory latency, reducing stalls, and making better use of available bandwidth. Ultimately, the opportunity is to cover the spectrum of irregular to regular computation: where tasks exhibit locality, multiplex fewer tasks and expend fewer instructions on context switching; where locality is lacking, multiplex more tasks at a higher rate to tolerate latency. Thus we make the best use of task parallelism – either to scale to more cores or to tolerate latency – and of caches – either to exploit application locality or to house more task contexts.

In this paper we introduce Grappa, a software runtime system that allows a commodity x86 distributed-memory HPC cluster to be programmed as if it were a single large shared-memory machine and provides scalable performance for irregular applications. Grappa is designed to smooth over some of the performance discontinuities in commodity hardware, giving good performance when there is little locality to be exploited while allowing the programmer to exploit it when it is available.

Grappa leverages as much freely available and commodity infrastructure as possible. We use unmodified Linux for the operating system and an off-the-shelf user-mode infiniband device driver stack [30]. MPI is used for process setup and tear down. GASnet [11] is used as the underlying mechanism for remote memory reads and writes using active message invocations. To this commodity hardware and software mix Grappa adds three main software components: (1) a *lightweight tasking* layer that supports a context switch in as few as 38ns and distributed global load balancing; (2) a *distributed shared memory* layer that supports normal access operations such as *read* and *write* as well as synchronizing operations such as *fetch-and-add* [22]; and (3) a *message aggregation* layer that combines short messages to mitigate the aforementioned problem that commodity networks are designed to achieve peak bandwidth only on large packet sizes, yet irregular applications tend to fetch only a handful of bytes at a time. As we will show later, Grappa can tolerate latencies way beyond that of the network. Therefore, Grappa can afford to *trade latency for throughput*: by *increasing* latency in key components of the system we are able to increase our effective random access memory bandwidth (by delaying and aggregating messages), our synchronization bandwidth (by delegating operations to remote nodes), and our ability to improve load imbalance (work stealing increases latency).

Our evaluation of Grappa shows that it runs several graph-crunching applications (classic examples of irregular behavior) very efficiently on a commodity cluster. Our yardstick for comparison is the XMT hardware itself. Using the same number of network interfaces, Grappa is in the same ballpark as the XMT: For unbalanced tree search, Grappa is over 3X faster and shows greatly improved scalability; conversely, for breadth first search and betweenness centrality Grappa is 2.5X slower. In Section 8 we explore the factors that underpin this performance. Most importantly, however, for significantly less real world cost, users can *add* significantly more processors to a commodity cluster than an XMT machine and use Grappa to achieve scalable performance.

2. Background

Grappa is built on many existing ideas in programming languages, systems and architecture. In this section we discuss related frameworks and key enabling technologies that Grappa builds upon.

Comparable frameworks Distributed graph processing frameworks like Pregel [28] and Distributed GraphLab [26] share similar goals as Grappa. Pregel adopts a bulk-synchronous parallel (BSP) execution model, which makes it inefficient on workloads that could prioritize vertices. GraphLab, on the other hand, schedules vertex computations individually, allowing prioritization, which gives faster convergence in a variety of iterative algorithms.

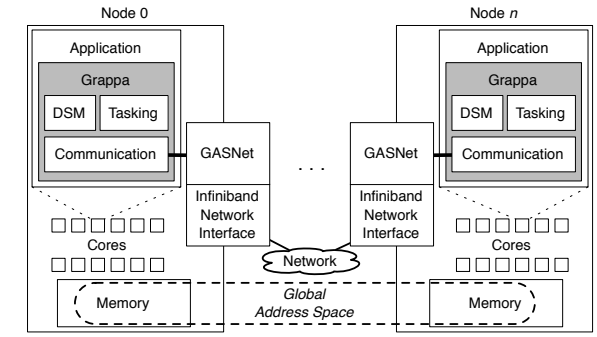

Figure 1. Grappa system overview

GraphLab, however, imposes a rigid computation model where programmers must express computation as transformations on a vertex and its edge list only, 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.

Global memory Grappa includes a custom implementation of a software distributed shared memory (DSM) system. Many traditional software DSM systems are page based [5, 9] and aim to hide the fact that they are built in software from applications by exploiting the processor's paging mechanisms, therefore relying heavily on locality. Instead, Grappa, like other partitioned global address space (PGAS) models, implements its DSM at the language, rather than system level. Languages such as Chapel [12], X10 [13], and UPC [18] make accesses to shared structures look like normal memory references. As we describe later, Grappa chooses a middle ground, where global addresses are explicit in the API and local accesses are emitted conventionally by the compiler. While Grappa's DSM system is conceptually similar to prior work, its implementation is tuned for irregular computations. Past DSM work, being page-based, could exploit the RDMA capabilities of network hardware to move large page-sized blocks of data from node to node. In our experience, when these networks move small blocks of data (a few bytes), only a fraction of the available bandwidth is achieved. In addition, the DSM system in Grappa ends up being tightly coupled to the task scheduler in order to overlap long latency memory operations with useful computation. For these two reasons it became necessary to build a new DSM system specifically for Grappa.

Multithreading Grappa uses multithreading to tolerate memory latency. This is a well known technique. Hardware implementations include the Tera MTA [3], Cray XMT [20], Simultaneous multithreading [34], MIT Alewife [1], Cyclops [2], and even GPUs [19]. As we describe in this paper, Grappa use a lightweight user-mode task scheduler to multiplex *thousands* of tasks on a single processing core. The large number of tasks is required because of the extremely high internode latency Grappa is mitigating. Grappa's task library employs several optimizations: an extremely fast task switch, a small task size, and judicious use of hardware prefetching to bring task state into the cache long before that task is actually scheduled.

3. Grappa Overview

Grappa (Figure 1) has three main software components:

Tasking system. Our tasking system supports lightweight multithreading to tolerate communication latency and global distributed workstealing (i.e., tasks can be stolen from any node in the system), which provides automated load balancing.

Distributed shared memory. Our DSM system provides support for fine-grain access to data anywhere in the system. It supports synchronization operations on global data, explicit local caching of any memory in the system, and support for operation on remote data (delegating operations to home node). By tight integration with the tasking system and the communication layer, our DSM system offers high aggregate random access bandwidth for accessing remote data.

Communication layer. As discussed earlier, modern commodity networks support high bandwidth only for large messages. Since irregular applications tend to need frequent communication of small requests, the main goal of our communication layer is to aggregate small messages into large ones to better exploit what the network can offer. It is largely invisible to the application programmer.

In the next three sections we describe both Grappa's main capabilities and how they are implemented.

4. Tasks

The basic unit of execution in Grappa is a *task*. Each task is represented by a function pointer and its arguments. A new task is enqueued at spawn time; when resources are free, it is allocated a stack, bound to a core, and executed.

During execution, a task yields control of its core whenever it performs 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 via the Grappa API calls shown in Figure 2. To minimize yield overhead, the Grappa scheduler operates entirely in user-space and does little more than store register state of one task and load that of another. Context switch times are as low as 38ns even when switching amongst thousands of tasks.

`yield()`
Yields core to scheduler, enqueuing caller to be scheduled again soon

`suspend()`
Yields core to scheduler, enqueuing caller only once another task calls wake

`wake( task * t )`
Enqueues some other task t to be scheduled again soon

Figure 2. Grappa API: scheduling

4.1 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 four methods for expressing parallelism, shown in Figure 3:

First, when the programmer identifies work that can be done in parallel, the work may be wrapped up in a function and queued with its arguments for later execution using a `spawn`.

Second, a programmer can use `spawn_on` to spawn a task on a specific core in the system or at the home core of a particular memory location.

Third, the programmer can invoke a parallel for loop, provided that the trip count is known at loop entry. The programmer specifies a function pointer along with start and end indices and an optional threshold to control parallel overhead. Grappa does *recursive decomposition* of iterations, similar to Cilk's cilk_for construct [10]. and TBB's `parallel_for` [33]. It generates a logarithmically-deep tree of tasks, stopping to execute the loop body when the number of iterations is below the required threshold.

Fourth, a programmer may want to execute an active message; that is, to run a small piece of code on a particular core in the system without waiting for execution resources to be available. For example, custom synchronization primitives execute this way, as a function executed on the core where the data lives. Grappa provides the `call_on` call for this purpose.

`spawn( void (*fp)(args) )`
Creates a new stealable task

`spawn_on( core, (*fp)(args) )`
Creates a new private task that will run on a specific core

`parallel_for( (*fp)(args), start, end )`
Executes iterations of a loop as stealable tasks

`call_on( core, (*fp)(args) )`
Runs a limited function on a specific core without consuming Grappa execution resources

Figure 3. Grappa API: expressing parallelism

4.2 Implementation tasks

Tasks and workers Grappa tasks are 32-byte entities: a 64-bit function pointer plus three 64-bit arguments. We use three arguments because tasks are often generated via a parallel loop decomposition, in which each task needs three kinds of data.

Function pointer addresses the routine to run. We configure the system nodes to disable address randomization, so that function pointers are valid across process images.

Private argument often a loop index.

Shared argument typically data shared amongst a group of tasks, or the number of loop iterations to be performed by this task

Synchronization argument often used to determine when all tasks that are part of a loop have finished; a global pointer to a synchronization object allocated at the core that spawned a group of tasks.

While these are the most common uses of the three task arguments, they are treated as arbitrary 64-bit values in the runtime, and can be used for any purpose.

Tasks are not allocated any execution resources until the scheduler decides to run them; when this occurs, tasks are matched with *worker* threads. Each worker is simply a collection of status bits and a stack, allocated at a particular core.

Context switching Grappa context switches between tasks 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 [4]. 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 the compiler sees all calls to the context switch routine, we can save even less state. Our context switch routine appears to the compiler as inline assembly; we declare all the registers we need to save as "clobbered" by the inline assembly routine, and the compiler will issue its own save and restore code as needed. This allows the compiler to avoid saving any registers that are not

used, or are used for temporary values that are not needed after the context switch.

Scheduling Each core in the Grappa system has its own independent scheduler. Each scheduler has three main tasks to perform: servicing communication requests as described in Section 6; rescheduling tasks that have been waiting on long-latency operations; and assigning ready tasks to worker resources that have become idle.

Each scheduler has three queues:

Ready worker queue This is a FIFO queue of tasks that are matched with workers and are ready to execute.

Private task queue This is a FIFO queue of tasks that must run on this core.

Public task queue This is a LIFO 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 or suspends, the scheduler makes a decision about what to do next. Servicing communication requests is given priority to ensure responsiveness, but to minimize overhead should context switches be frequent, servicing is performed only if sufficient time has elapsed. Second, the scheduler determines if any workers with running tasks are ready to execute; if so, one is scheduled. Finally, if there are no workers ready to run, but there are tasks waiting to be matched with workers, an idle worker is woken (or a new worker is spawned), matched with a task, and scheduled.

Work stealing When the scheduler finds no work to assign to its workers, it commences to steal work 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. We choose to initiate one outstanding steal whenever the ready queue is empty there are idle worker threads.

5. Memory

Applications written for Grappa utilize two forms of memory: local and global. Local memory is local to a single core in the system. Accesses occur through conventional pointers. The compiler emits an access and the memory is manipulated directly. 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 that is 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 4.

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. The heap metadata is stored on a single node. Currently all heap operations serialize through this node; while this has been sufficient for our benchmarks, in the future Grappa will provide parallel performance through combining [16, 25].

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 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 indicates 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} endpoints. Any node-local data can be made accessible by other nodes in the system by wrapping the address and node ID into a 2D global address. This address can then be accessed with a delegate and can also be cached by other nodes. At the destination the address is converted 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 5 shows how 2D and linear addresses can refer to other cores' memory.

```
global_address global_malloc( size )
global_free( global_address )
  Allocates and frees memory in the global heap
delegate_read( global_address, local_var )
delegate_write( global_address, local_var )
delegate_cas( global_address, local_var )
delegate_fetch_inc( global_address, local_var )
  Performs a memory operation at the home core of a global address
cache_acquire( global_address, local_buf, {RO,RW,WO})
cache_release( global_address, local_buf )
  Perform cache operations to acquire/release global data. Acquire, returns a local pointer after all data has been copied to the local node. Release, optionally writes data back to global memory and frees up management resources.
```

Figure 4. Grappa API: accessing memory

Global memory access There are two general approaches Grappa applications use to *a*ccess global memory. When the programmer expects a computation on shared data to have spatial locality to exploit, *cache* operations may be used. When there is no locality to exploit, *delegate* operations are used.

Explicit caching. Grappa provides an API to fetch a global pointer of any length and return a local pointer to a cached copy of the global memory. Grappa cache operations have the usual read-only and read-write variants, along with a write-only variant used to initialize data structures. Languages for distributed shared memory systems have done optimizations to achieve a similar goal. For example, the UPC compiler coalesces struct and array accesses into remote get/put [14], and Fortran D compiler's message vectorization hoists small messages out of a loop [24]. Caching in Grappa additionally provides a mechanism for exploiting temporal locality by operating on the data locally.

Under the hood, Grappa performs the mechanics of gathering chunks of data from multiple system nodes and presenting a conventional appearing linear block of memory as a local pointer into a cache. The strategy employed is to issue all the constituent requests of a cache access request (as Active Messages) and then yield until all responses have occurred. Currently, Grappa caches are *not* coherent, requiring the programmer to maintain consistent access to

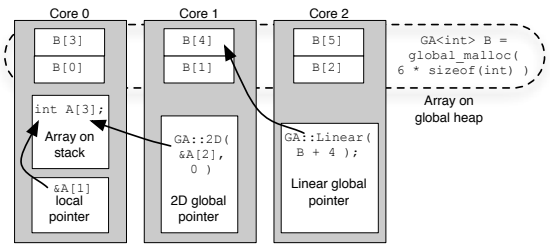

Figure 5. Grappa memory structure

data. Future work will develop a software directory based coherence scheme to simplify consistent access to global data.

Delegate operations. When the 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 provide this capability. Applications can dispatch computation to be performed on individual machine-word sized chunks of global memory to the memory system itself (e.g., *fetch-and-add*). Delegate operations, proposed in [29] and [27], are also the primary synchronization method in Grappa.

Delegate operations are always executed at the home core of their address, and while arbitrary memory operations can be delegated, we restrict the use of delegate operations in three ways to make them more useful for synchronization. First, we limit each task to one outstanding delegate operation to avoid the possibility of reordering in the network. Second, 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. Finally, no context switches are allowed while the data is being modified. 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 atomic operations. Figure 6 depicts an example of how delegate and cache operations interact.

6. Communication

In order to mitigate the low message injection rate limits of commodity networks, Grappa's communication stack has two layers: one for user-level messages and one for network-level messages.

At the upper layer, Grappa implements asynchronous active messages [35]. Each message consists of a function pointer, an optional argument payload, and an optional data payload. When a task sends a message, the message is copied to a send queue associated with the message's destination and the task continues execution.

Grappa's lower networking layer aggregates the upper layer's messages to improve performance. 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 and processing on the CPU within the driver stack). Our measurements confirm manufacturers published data [15], that with this packet size the bisection bandwidth is only a small fraction, less than 3% of the peak bisection bandwidth.

In our experiments the vast majority of requests were smaller than 44 bytes, far too small to make efficient use of the network. To make the best use of the network, we must convert our small messages into large ones. When a task sends a message, it is not immediately sent, but rather placed in a queue specific to the destination.

There are three situations in which a queue of aggregated messages is sent. First, each queue has a message size threshold of 4096 bytes, chosen to give reasonable network performance. If the size in bytes of a queue is above the threshold, the contents of the queue are sent immediately. Second, each queue has a wait time threshold ($\approx$1ms). If the oldest message in a queue has been waiting longer than this threshold, the contents of the queue are sent immediately, even if the queue size is lower than the message size threshold. Third, queues may be explicitly flushed in situations where the programmer wants to minimize the latency of a message at the cost of bandwidth utilization.

The network layer is serviced by polling. Periodically when a context switch occurs, the Grappa scheduler switches to the network polling thread. This thread has three responsibilities. First, it polls the lower-level network layer to ensure it makes progress. Second, it deaggregates received messages and executes active message handlers. Third, it checks to see if any aggregation queues have messages that have been waiting longer than the threshold; if so, it sends them.

Underneath the aggregation layer, Grappa uses the GASNet communication library [11] to actually move data. All interprocess communication, whether on or off a cluster node, is handled by the GASNet library. GASNet is able to take advantage of many communication mechanisms, including ethernet and infiniband between nodes, as well as shared memory within a node.

Some networks provide access to a remote machine's memory directly. This would seem to be a good fit for a programming model focused on global shared memory, but in fact we do not use it. In our experiments, we found that RDMA operations are subject to the same message rate limitations as all other messages on these cards, and thus using raw RDMA operations for our small messages would make inefficient use of bandwidth. Instead, we implement remote memory operations with active messages. A byproduct of this design decision is that Grappa is not limited to RDMA-capable networks.

7. Methodology

To explore the performance of the Grappa runtime we have implemented three algorithms: breadth first search, betweenness centrality, and unbalanced tree search. These algorithms were implemented for the Cray XMT (our baseline) and for Grappa. The metric we use is algorithmic time, which means startup and loading of the data structure (from disk) is not included in the measurement. Data is collected on real systems, which means minor variations in runtime exist from run to run. The average of multiple runs are used, and where appropriate, confidence in the result is reported.

One question that must be answered when making a comparison between the Cray XMT and a typical HPC cluster is what is a fair comparison – these systems are quite different. Three options immediately come to mind: equal number of processing cores, equal number of network interfaces, and equal dollars. We discounted the last option fairly quickly, because it isn't a lasting data-point – the cost of hardware shifts over time, skewing the interpretation of results. The first option, cores, has some merit, but Grappa is designed for applications that have no locality in their computation. This means almost all of their memory accesses are remote. The factor that limits their performance is not processing, but communicating. Hence, we have chosen the middle option, network interfaces as the way to normalize across the XMT and our cluster. Each processor in the XMT system has its own network interface to access shared memory. In the HPC cluster we use, each processor (which contains up to 32 cores, although we only use 6 in our ex-

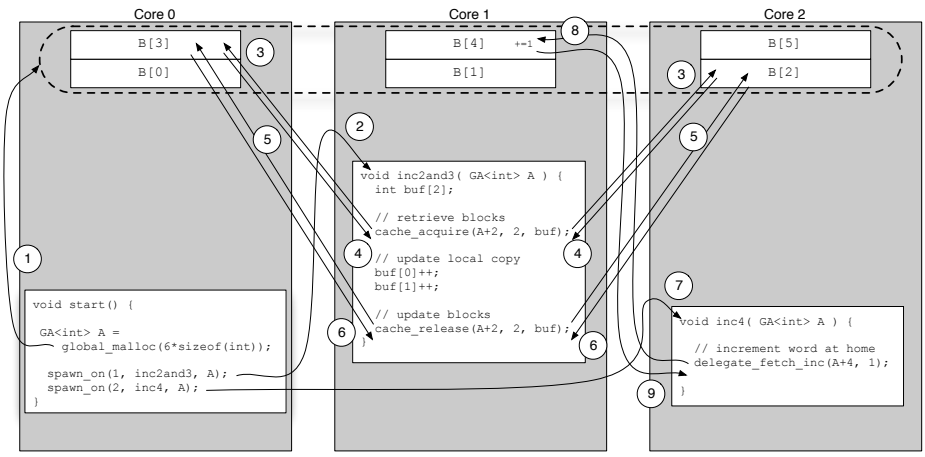

Figure 6. Delegation and cache example: In step 1, a core allocates an array in the global heap. It then spawns two tasks on remote cores to increment elements of the array. The first task increments two elements of the array using cache operations. In step 2, the task is invoked. A cache request is issued for two adjacent integers starting at the second element of the array. Since these element are stored in the memories of two different cores, this requires the sending of two messages in step 3. The task is suspended until both responses arrive in step 4. The data carried in these responses is stored in the local buffer. The elements are then increment in the buffer. Then in step 5, the modified data is sent back to the home node. Acknoledgements are returned in step 6 so the task knows when the writes are complete. The second task increments an element of the array with a delegate operation. In step 7, the task is invoked. A delegate request is sent to the home core of the array element with the increment value. The task suspends until the response is received. In step 8, the increment is executed on the remote core. A response is returned in step 9 with the previous value of the array element.

periments), has a single infiniband interface. Hence, for our results we scale up XMT processors one for one with full system nodes.

7.1 Systems

For measurements, we run Grappa on a 144-node cluster of AMD Interlagos processors. Nodes have 32-core (every pair share a floating-point unit) 2.1-GHz processors, 64GB of memory, and 40Gb Mellanox ConnectX-2 infiniband network card. The cluster uses a QLogic infiniband switch. We configure the nodes to have 32 1-GB hugepages to minimize TLB misses for the random access patterns we expect from irregular applications.

We compare to the MTA using 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.

7.2 Applications

We have used three benchmarks to explore the performance of Grappa:

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 [31]. 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. We call this benchmark UTS-Mem, and the timed portion is this traversal of the in-memory tree. This in-memory traversal has no knowledge of the tree structure beforehand. The Grappa version of the in-memory tree search uses the asynchronous parallel for loop over a visited vertex's children list.

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 [23]. 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. One of the reference implementations of Graph500 BFS is for the XMT; this code fails to scale past 16 XMT processors because it does not expose enough parallelism, so we modified the code to use a recursive loop decomposition similar to Grappa's. We compare this modified version against a straightforward Grappa implementation. We do not employ algorithmic improvements, though there are many [8, 36].

Betweenness Centrality An important measure of the importance of particular vertices in a network is betweenness centrality (BC) [21]. By this measure, the "importance" of each vertex is computed by finding the fraction of shortest paths that pass through it, which can optionally be approximated by only computing shortest paths using a subset of the vertices as starting points. BC can be useful for understanding which vertices may have the greatest impact, so in social networks this could be the primary person linking two communities. Because it requires multiple breadth-first traversals across the entire graph and in reverse, on power-law degree graphs, this algorithm exercises random accesses rate, and load balancing. It also requires fine-grained synchronization on updates to vertex centrality values because multiple paths will update the same vertex. BC is a kernel in the DARPA High Performance Computing Systems (HPCS) Scalable Synthetic Compact Applications

graph analysis (SSCA#2) benchmark [6]. Performance for BC is also measured in TEPS, where the number of traversed edges is the total number of edges in the graph multiplied by the number of random starting vertices used in the approximate computation. We use the XMT implementation of BC implemented as part of the GraphCT [17] library and a comparable Grappa implementation, both of which use the parallel BC algorithm developed by Bader et al. [7].

8. Evaluation

We evaluate Grappa in four ways. First, we evaluate its performance with respect to the XMT. Second, we explore the benefit that aggregation gives us. Third, we investigate sensitivity of two key parameters in the runtime. Fourth, we investigate scaling to more system nodes.

8.1 Performance

To evaluate Grappa's performance with respect to the XMT, we ran each of our three benchmarks on up to 16 nodes of each machine. Grappa used 6 cores per node, with the best parameters chosen for each point. In some cases, the XMT could not run the benchmark with 2 nodes, so the point is omitted.

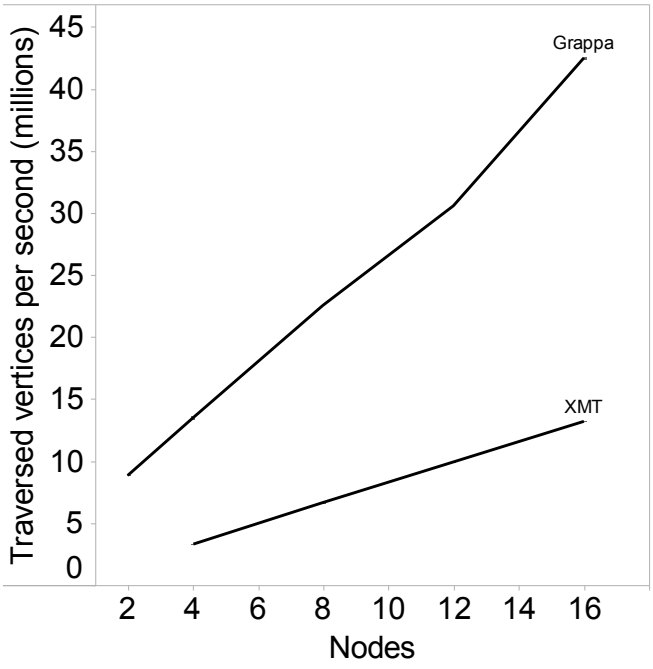

Figure 7. Performance of in-memory unbalanced tree search.

Unbalanced tree search We ran UTS-mem with a geometric 100M-vertex tree (T1L). Figure 7 shows the performance in terms of number of vertices visited per second versus number of compute nodes. Grappa is 3.2 times faster than the XMT at 16 nodes. As we will show later, the performance advantage Grappa has over XMT increases as more nodes are added. The main reason Grappa performs better is the software-based delegate synchronization obviates the need for the retry-based synchronization that XMT uses.

BFS We ran BFS on a synthetic Kronecker graph with 2^{25} vertices and 2^{29} edges (25 GB of data). Figure 8 shows our performance in terms of graph edges traversed per second. The XMT is 2.5 times faster than Grappa at 16 nodes. Performance does scale at a constant rate for Grappa, suggesting that adding more nodes will increase performance.

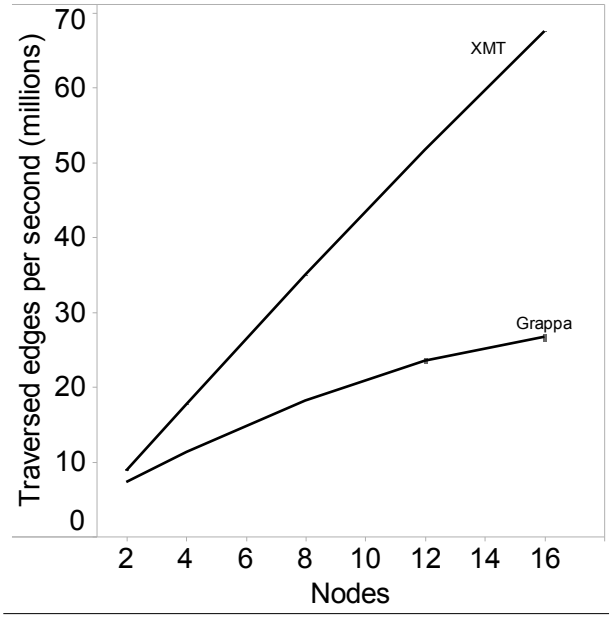

Figure 8. BFS performance

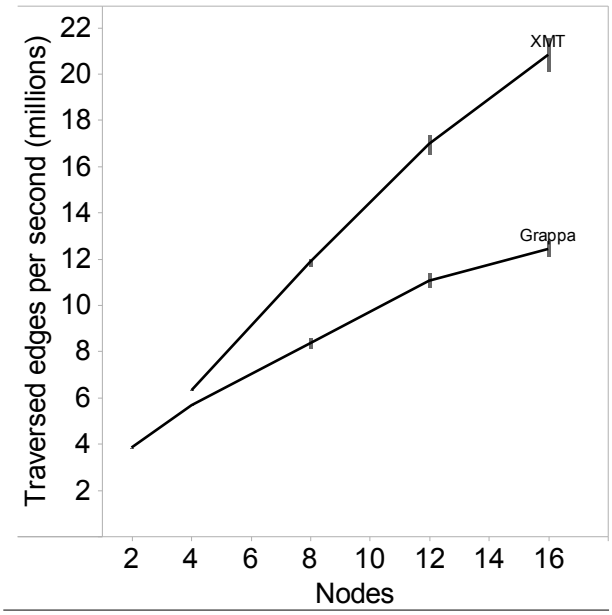

Figure 9. Centrality performance

Centrality We ran Betweenness Centrality on the same scale 25 Kronecker graph as we did for BFS. Figure 9 shows our performance in terms of graph edges traversed per second. At 16 XMT processors/cluster nodes, the XMT is 1.75 times faster than Grappa.

8.2 Network aggregation performance

To evaluate the benefits of network aggregation, we ran two experiments. First, we ran a simple unidirectional ping test to see the maximum benefit the aggregator can provide in terms of improved network efficiency. Second, we ran BFS with the aggregator disabled in order to measure its benefit on an application.

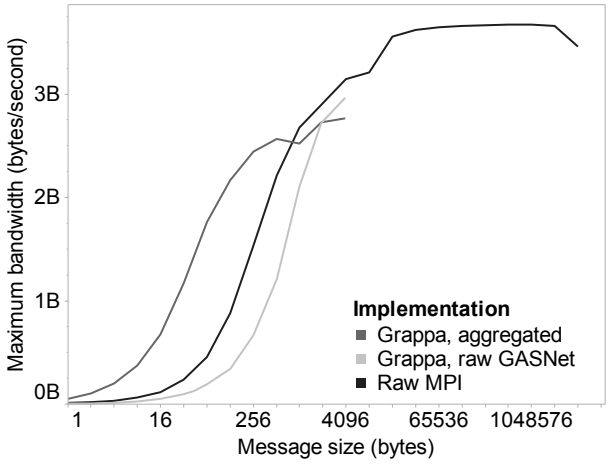

Figure 10. Bandwidth versus message size unidirectional ping test for Grappa with aggregation, Grappa with raw GASNet messages, and MPI. Aggregation provides an 11x bandwidth benefit at our common operating point.

To implement the ping test, we wrote a simple Grappa application where the cores of one node send messages as fast as possible to the cores of another node. We vary the size of the payload up the maximum payload size supported by the aggregator (nearly 4KB). Each core has a single task sending to a single destination, so this is a best case scenario for the aggregator. To see the benefit of the aggregator, we added a bypass that lets us send messages directly through GASNet. We also compare against the OSU `osu_mbw_mr` benchmark [32] compiled against OpenMPI 1.5.3; this benchmark has the same pattern of communication but doesn't have the overhead of Grappa's context switching.

The results are shown in Figure 10. There are two key observations.

First, small message performance against the existing libraries is, as expected, poor. The MPI application test shows us that peak per node bandwidth supported by our infiniband card is 3.4GB/s. This is achievable only with large messages; we must send 16KB packets to get within 5 percent of peak bandwidth. But in our benchmarks, we saw average message between 32 and 64 bytes. At 32 bytes, the MPI test is using less than 7 percent of its peak bandwidth. Grappa sending messages directly through GASNet uses less than 3 percent of the peak bandwidth.

Second, aggregation has the potential to improve this situation by an order of magnitude. With aggregation, Grappa is able to send 32-byte messages over 12 times faster than using GASNet directly. This is a more respectable 32 percent of peak bandwidth. Due to expedient design decisions, Grappa's aggregator limits its aggregation to 4KB; this limits its peak achievable bandwidth to 75 percent of the actual peak.

This comparison is the best possible case for the aggregator. In order to verify that the aggregator still has value on actual applications at scale, we ran a small (100M node tree) UTS-Mem with the aggregator disabled, on 16 nodes. Figure 11. At this configuration, the aggregator improves our application performance by 10x.

8.3 Sensitivity

Aggregator timeout One of the key parameters of the aggregator is the message timeout. All messages that are queued must eventually be sent in order to ensure progress. In the best case, we are able to aggregate enough messages to fill an aggregation buffer and cause it to be sent, but as we scale up, the average rate of messages heading to a common destination decreases, and this gets harder. To

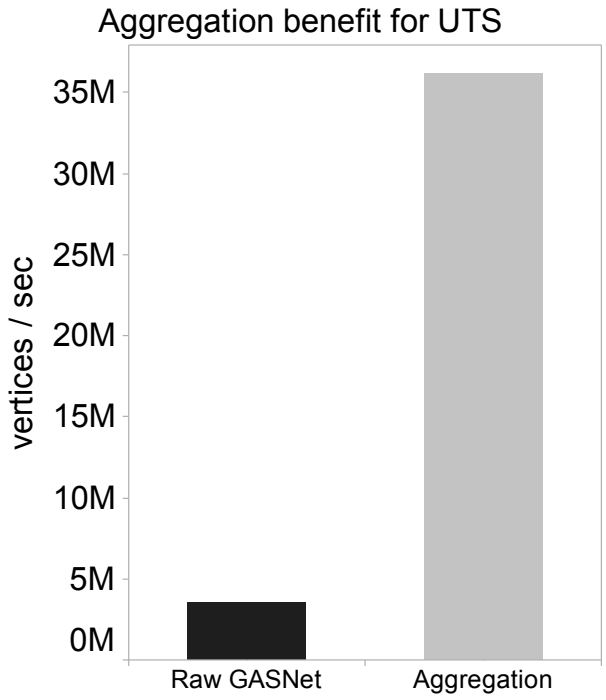

Figure 11. Performance of UTS on 16 nodes with and without Grappa's aggregation.

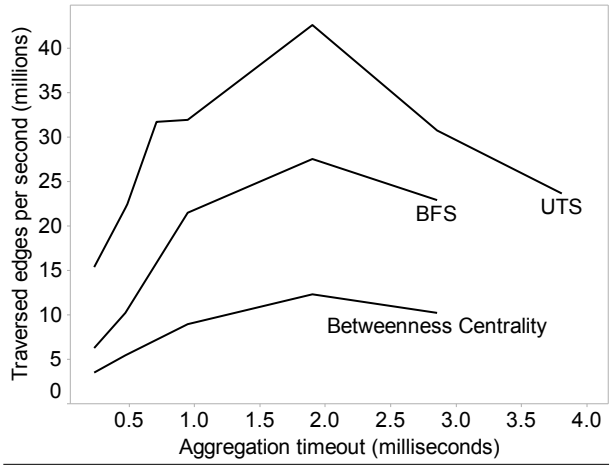

Figure 12. Sensitivity to aggregation delay

bound the problem, the aggregator includes a timeout. Any packet waiting this long is sent the next time the communications layer is serviced.

Figure 12 shows a sweep of this parameter for UTS, BFS, and Betweenness Centrality on 16 nodes, using the datasets described previously. The maximum number of workers is fixed at 2048. All the benchmarks show a performance peak with a 2 millisecond timeout; at this point we are delaying long enough to aggregate the largest packets we can; setting the parameter higher causes tasks to wait longer for responses, but few new requests are being generated.

Number of active tasks When a task issues a request that requires a response, it blocks to allow other tasks to utilize its core. These tasks may also block. To support the many milliseconds of latency

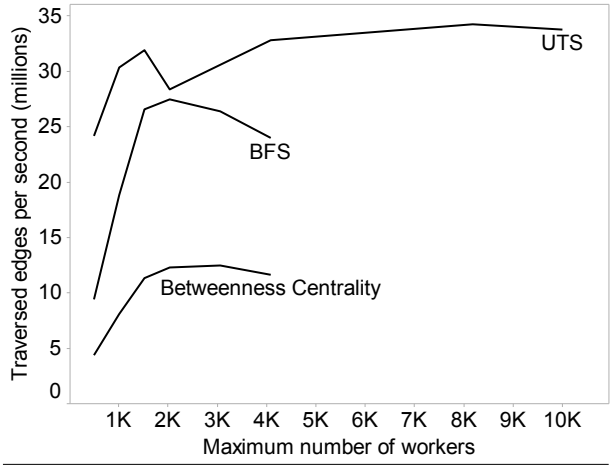

Figure 13. Sensitivity to maximum active tasks

aggregation adds, we need to support many thousands of blocked tasks. One of the key parameters of the runtime is the number of blocked tasks allowed; we need enough to cover the network and aggregation latency, but too many running tasks can add extra latency as they all must be multiplexed onto the same core.

Figure 13 shows a sweep of the maximum number of active tasks (workers) per core for each of our three benchmarks on 16 nodes. The aggregator timeout is set at 1 ms for UTS and 2 ms for BFS and Betweenness Centrality. The performance peak shifts in this case, with UTS peaking at 1536 workers, BFS peaking at 2048 workers, and Betweenness Centrality peaking at 3072 workers. This is the point where we have enough workers to cover the latency of aggreation. The different values reflect the different amounts of work done by a task in each benchmark; UTS does the least, while Betweenness Centrality does the most.

Parallel loop threshold Parallel overhead—in the form of context switches, task spawns, and synchronization—can reduce the performance benefit of parallelism. Grappa sees a benefit to limiting the amount of parallelism created by a recursive loop decomposition. The parallel loop threshold ("parallel granularity") parameter tells the runtime when to stop creating new tasks and just execute iterations sequentially. This allows us to amortize the overhead of task creation. In addition, assigning sequential iterations to a single task provides the potential to exploit locality when data for adjacent iterations is also adjacent in memory. The ability to exploit this locality that exists in the application is an important advantage. We found that in UTS and BFS, increasing the threshold from 1 up to 8 or 16, respectively, increases performance by more than 60%.

8.4 Scaling

To determine how Grappa's performance scales compared to the performance of the entire XMT, we ran a set of experiments up to all 128 XMT processors and 128 cluster nodes. For the XMT, the number of allowed processors was varied up to the entire machine, with some minor tuning of stream parameters needed to get optimal performance. For Grappa, parameters such as cores per node, aggregator timeouts, and parallel threshold were tuned to get the best performance for each node count. All of the benchmarks continue to improve out to 128 nodes for Grappa. UTS continues to fare better than the XMT with large node counts, with the XMT appearing to plateau at 60 processors due to contention from synchronization retries, while Grappa handles this by suspending tasks until messages return. For BFS and Centrality, the XMT scales approximately a constant factor better than Grappa. We attribute this to a limitation in the current aggregator design and network stack that Grappa uses. This limits the practical number of cores we can use to 6 per node (adding more cores per node *decreases* performance). Ironically, this limitation makes Grappa applications compute-bound instead of network-bound. Work is ongoing to rework the Infiniband driver stack and aggregation interface to remove this limitation and improve aggregation addressing using local routing.

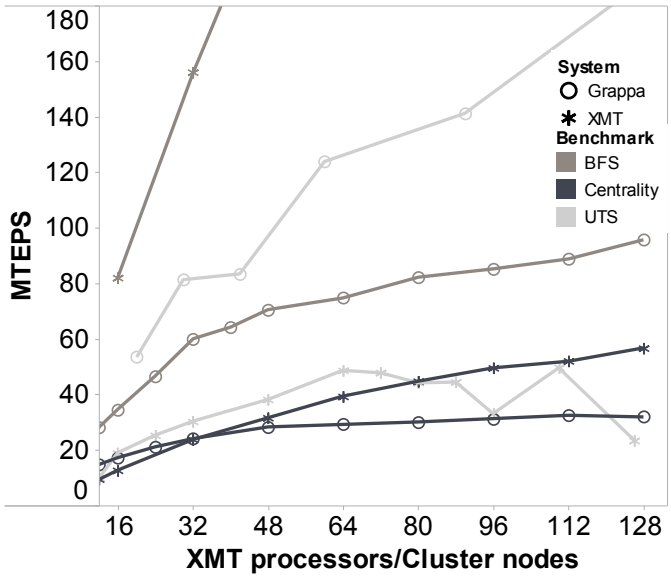

Figure 14. Scaling number of nodes: Grappa continues to perform significantly better than XMT for UTS but scales a constant factor slower than XMT for BFS (4x slower) and Centrality (2x slower).

9. Conclusion

Irregular computations are both important and challenging to execute quickly. Scaling these applications easily to commodity hardware has been a historical challenge. Grappa is a runtime framework that simplifies this task for software developers and compiler writers. This paper describes the Grappa framework and its three main components: a task library, a distributed shared memory system, and a network aggregator to make commodity networks efficient with small message sizes. Grappa 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.

We explored the performance of Grappa on three algorithms: unbalanced tree search, breadth first search, and betweenness centrality. Performance comparisons to the Cray XMT for a small number of nodes (16 nodes / 96 cores) show great promise: Grappa performs on-par with the far more expensive and custom XMT system, sometimes achieving better performance and sometimes worse. Our initial large scaling experiments to a higher number of nodes shows both promise and room for improvement. Grappa trounces XMT on UTS, but while Grappa performance scales on BFS and betweenness centrality, limitations in the current aggregator design give the performance edge to XMT. Work is ongoing on further refining this component.

References

[1] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The MIT

Alewife machine: Architecture and performance. In *22nd Annual International Symposium on Computer Architecture*, June 1995.

[2] G. Almási, C. Caşcaval, J. G. Castaños, M. Denneau, D. Lieber, J. E. Moreira, and H. S. Warren, Jr. Dissecting Cyclops: A detailed analysis of a multithreaded architecture. *SIGARCH Computer Architecture News*, 31:26–38, March 2003.

[3] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. 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.

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

[5] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *Computer*, 29(2):18 –28, feb 1996.

[6] D. Bader, J. Feo, J. Gilbert, J. Kepner, D. Koester, and E. Loh. HPCS scalable synthetic compact applications #2: Graph analysis, 2006.

[7] D. Bader and K. Madduri. Parallel algorithms for evaluating centrality indices in real-world networks. *The 35th International Conference on Parallel Processing (ICPP)*, pages 539–550, 2006.

[8] S. Beamer, K. Asanovi, and D. Patterson. Direction-optimizing breadth-first search. In *Conference on Supercomputing (SC-2012)*, November 2012.

[9] J. K. Bennett, J. B. Carter, and W. 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, 1990.

[10] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *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.

[11] D. Bonachea. GASNet Specification, v1.1, 2002.

[12] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the Chapel Language. *International Journal of High Performance Computing Application*, 21(3):291–312, Aug. 2007.

[13] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. 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.

[14] W.-Y. Chen, C. Iancu, and K. Yelick. Communication optimizations for fine-grained UPC applications. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, PACT '05, pages 267–278, Washington, DC, USA, 2005. IEEE Computer Society.

[15] H. A. Council. Interconnect analysis: 10GigE and InfiniBand in high performance computing. `http://www.hpcadvisorycouncil.com/pdf/IB_and_10GigE_in_HPC.pdf`, July 2012.

[16] D. Dice, V. J. Marathe, and N. Shavit. Flat-combining NUMA locks. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 65–74, New York, NY, USA, 2011. ACM.

[17] D. Ediger, K. Jiang, J. Riedy, D. Bader, C. Corley, R. Farber, and W. Reynolds. Massive social network analysis: Mining Twitter for social good. In *39th International Conference on Parallel Processing (ICPP)*, pages 583–593. IEEE, 2010.

[18] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick. *UPC: Distributed Shared Memory Programming*. John Wiley and Sons, Inc., Hoboken, NJ, USA, 2005.

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

[20] J. Feo, D. Harper, S. Kahan, and P. Konecny. Eldorado. In *Proceedings of the 2nd Conference on Computing Frontiers*, CF '05, pages 28–34, New York, NY, USA, 2005. ACM.

[21] L. C. Freeman. Centrality in social networks conceptual clarification. *Social Networks*, page 215, 1978.

[22] A. Gottlieb, R. Grishman, C. Kruskal, K. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer: Designing an MIMD shared memory parallel computer. *IEEE Transactions on Computers*, C-32(2):175 –189, feb. 1983.

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

[24] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, Aug. 1992.

[25] S. Kahan and P. Konecny. MAMA!: A memory allocator for multithreaded architectures. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 178–186, New York, NY, USA, 2006. ACM.

[26] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *PVLDB*, 2012.

[27] R. Lublinerman, J. Zhao, Z. Budimlic, S. Chaudhuri, and V. Sarkar. Delegated isolation. In *OOPSLA'11*, pages 885–902, 2011.

[28] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. 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.

[29] J. Nelson, B. Myers, A. Hunter, L. Ceze, D. Grossman, M. Oskin, C. Ebeling, S. Kahan, and P. Briggs. Crunching large graphs on commodity processors. *3rd USENIX Workshop on Hot Topics in Parallelism (HotPar)*, May 2011.

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

[31] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. 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.

[32] Osu mpi benchmarks. `http://mvapich.cse.ohio-state.edu/benchmarks/`, July 2012.

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

[34] D. M. Tullsen, S. J. Eggers, and H. 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.

[35] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. 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.

[36] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek. A scalable distributed parallel breadth-first search algorithm on BlueGene/L. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, pages 25–25. IEEE, 2005.