XPS: Towards an Architecture for Large-Scale, Irregular and Low-Locality Applications

Luis Ceze **Simon Kahan** **Mark Oskin (PI)**
University of Washington

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, machine learning, ...) 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 [**?**] 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.

Our preliminary work on 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 **??** 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 Related work

3 Description of the Current Grappa System

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.

Grappa is 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 [**?**]. MPI is used for process setup and tear down. GASnet [**?**] 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* [**?**]; 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).

Grappa (Figure **??**) 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.

3.1 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 1. 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 1: Grappa API: scheduling

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 four methods for expressing parallelism, shown in Figure 2:

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 [**?**] and TBB's `parallel_for` [**?**]. 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 2: Grappa API: expressing parallelism

3.3 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 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. 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 [**?**, **?**].

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 4 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 3: 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 [**?**], and Fortran D compiler's message vectorization hoists small messages out of a loop [**?**]. Caching in Grappa additionally provides a mechanism for exploiting temporal locality by operating on the data locally.

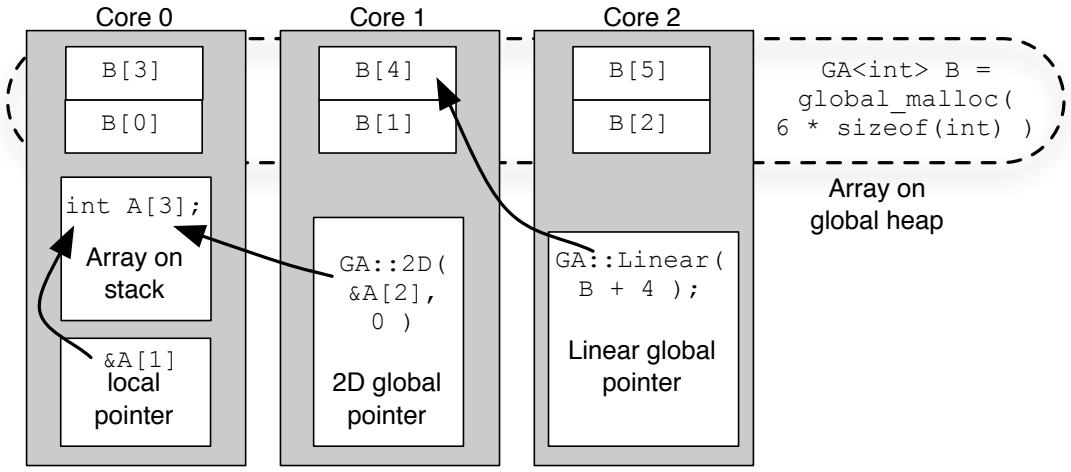

Figure 4: Grappa memory structure

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 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 [**?**] and [**?**], 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 5 depicts an example of how delegate and cache operations interact.

3.4 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 [**?**]. 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

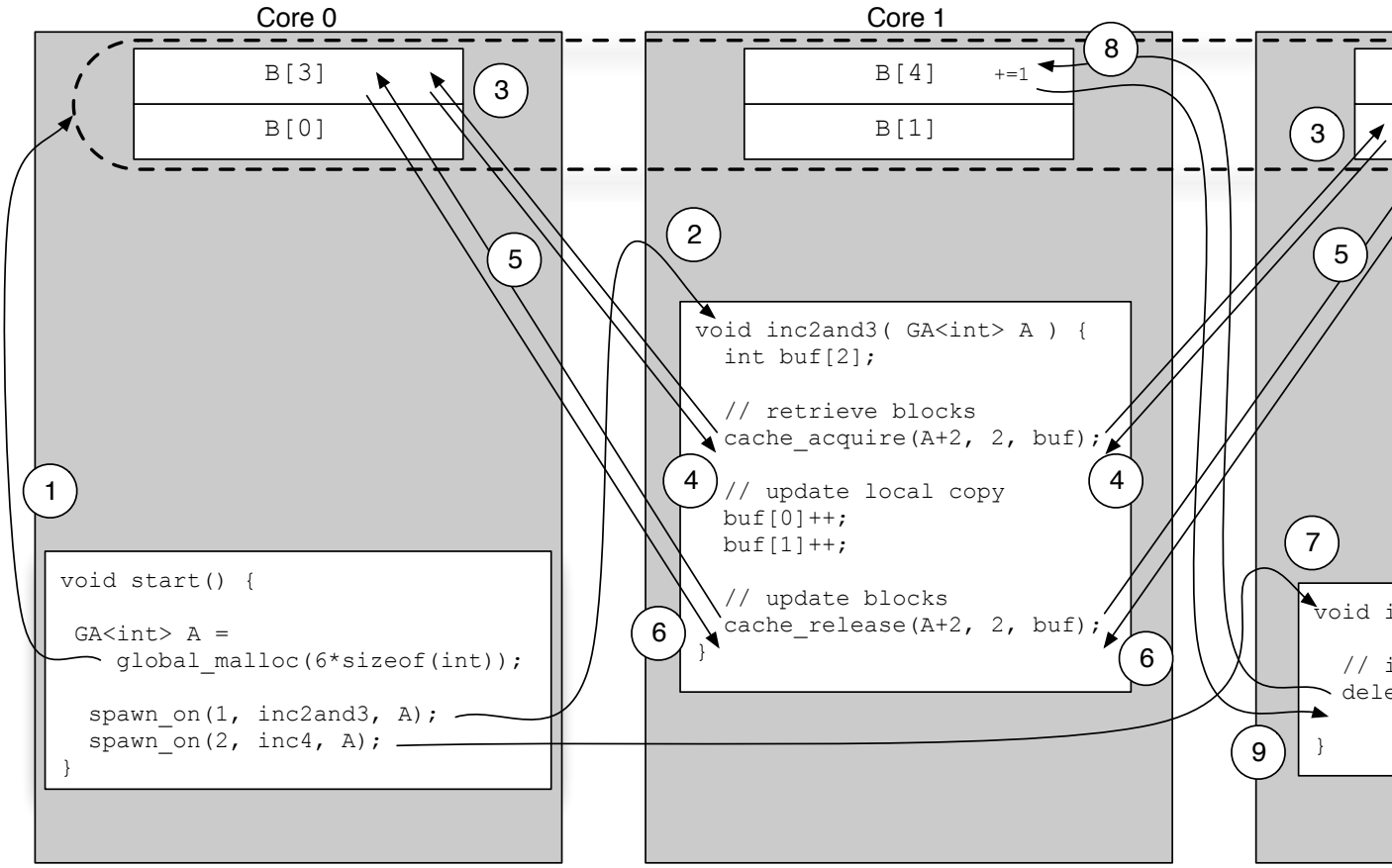

Figure 5: **Delegation and cache example:** In step 1, a core allocates an array in the global heap. It then spawns two tasks o elements of the array using cache operations. In step 2, the task is invoked. A cache request is issued for two adjacent integers memories of two different cores, this requires the sending of two messages in step 3. The task is suspended until both respons The elements are then increment in the buffer. Then in step 5, the modified data is sent back to the home node. Acknoledgem second task increments an element of the array with a delegate operation. In step 7, the task is invoked. A delegate request suspends until the response is received. In step 8, the increment is executed on the remote core. A response is returned in ste

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 [**?**], 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 [**?**] 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.

3.5 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.

Unbalanced tree search We ran UTS-mem with a geometric 100M-vertex tree (T1L). Figure 6 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 7 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.

Centrality We ran Betweenness Centrality on the same scale 25 Kronecker graph as we did for BFS. Figure 8 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.

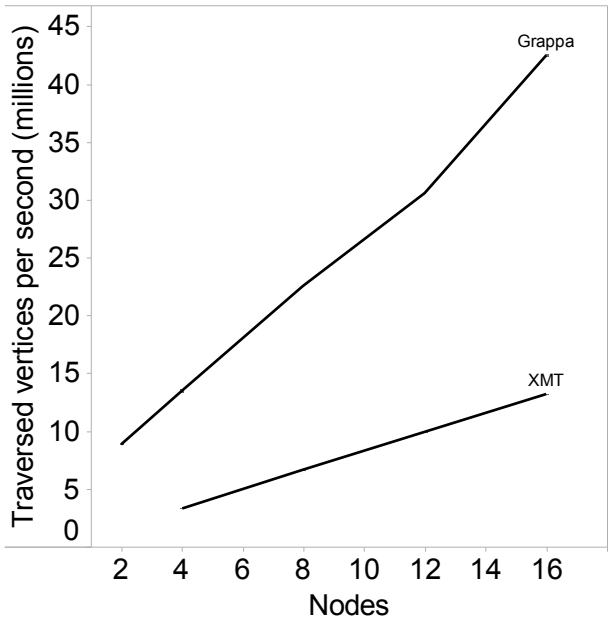

Figure 6: Performance of in-memory unbalanced tree search.

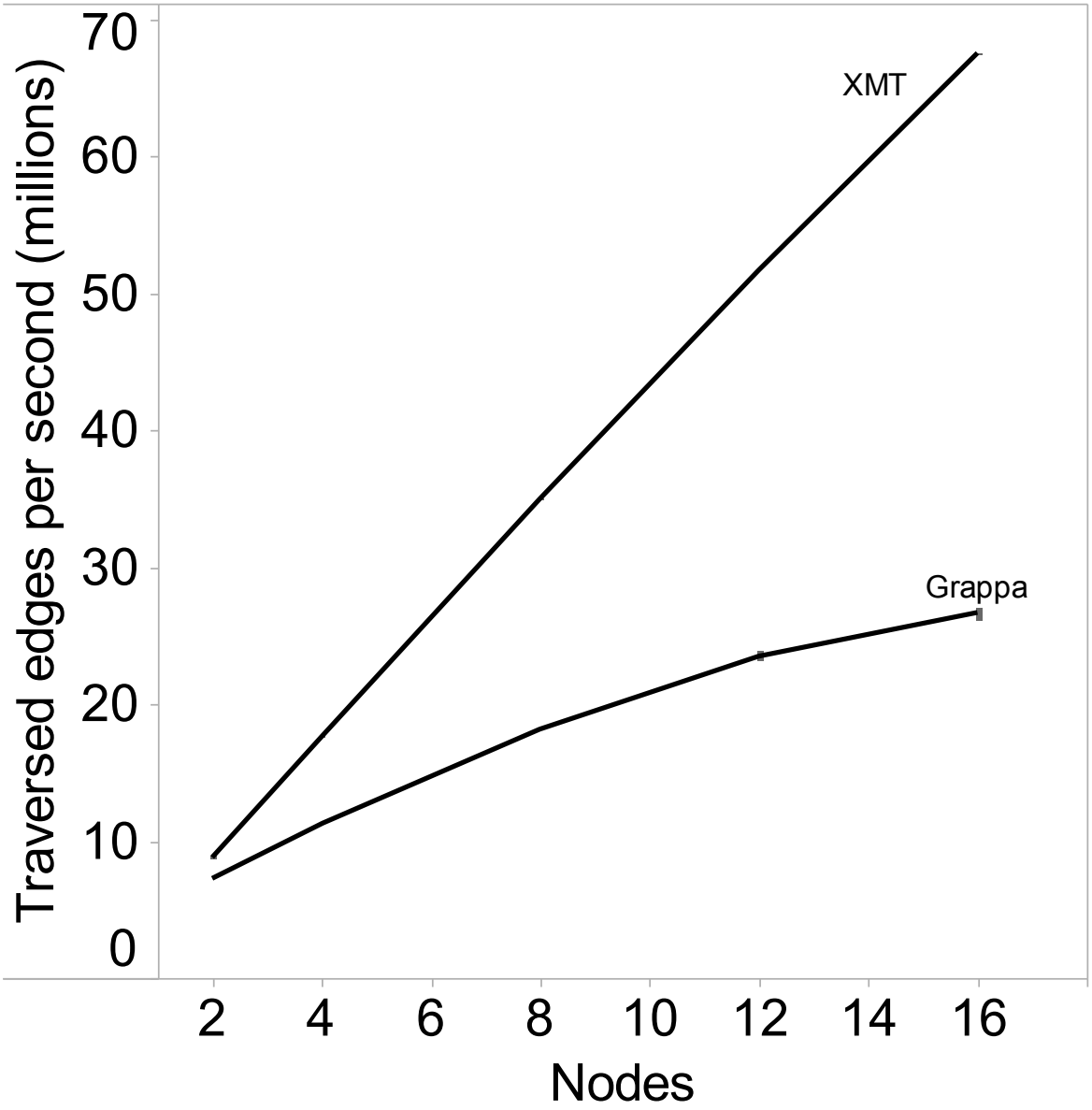

Figure 7: BFS performance

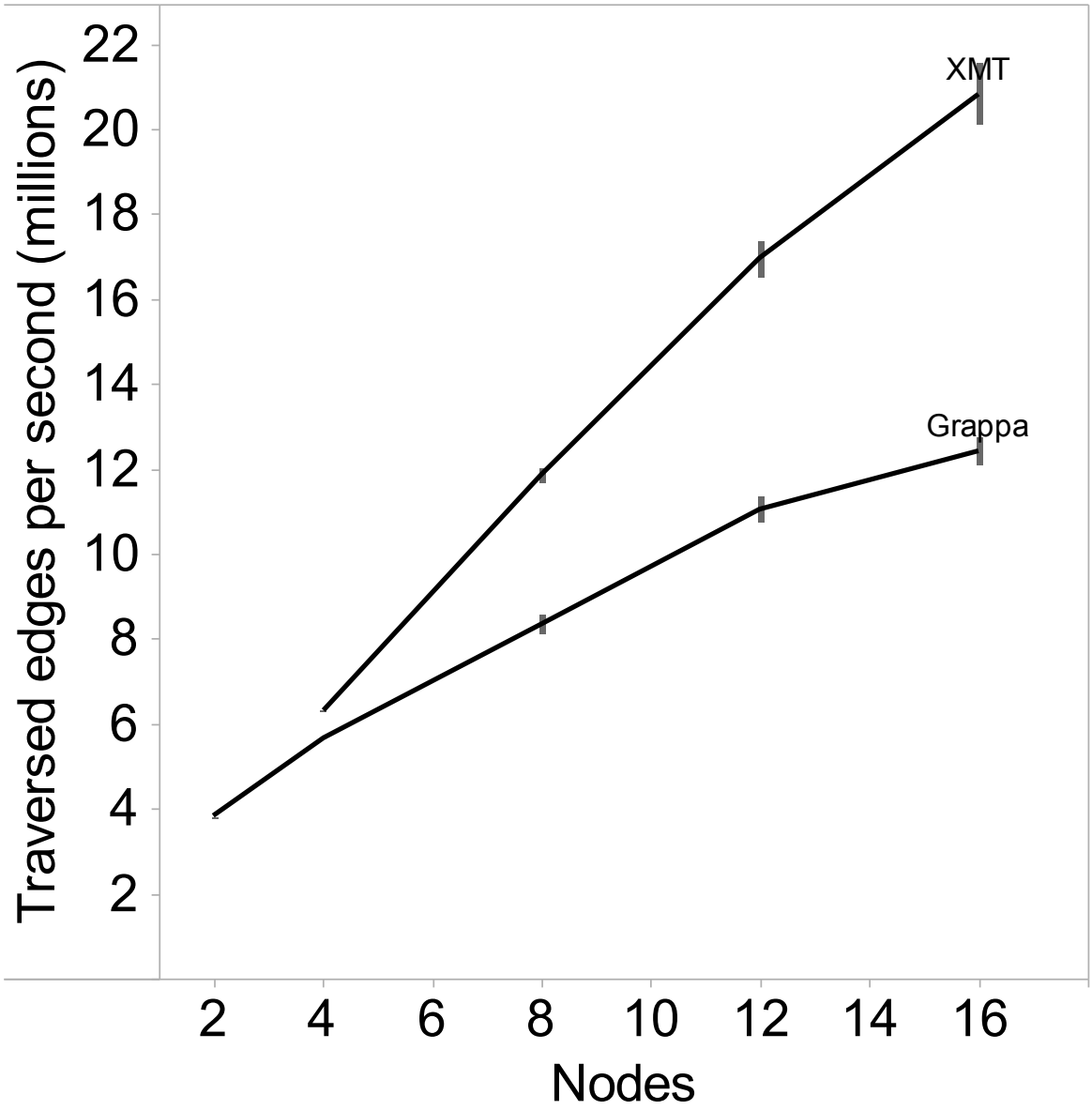

Figure 8: Centrality performance