Chaos: Scale-out Graph Processing from Secondary Storage

* Amitabha Roy[1] Laurent Bindschaedler[2] Jasmina Malicevic[2] Willy Zwaenepoel[2]

Intel[1] EPFL[2], Switzerland

firstname.lastname@{intel.com[1], epfl.ch[2]}

Abstract

Chaos scales graph processing from secondary storage to multiple machines in a cluster. Earlier systems that process graphs from secondary storage are restricted to a single machine, and therefore limited by the bandwidth and capacity of the storage system on a single machine. Chaos is limited only by the aggregate bandwidth and capacity of all storage devices in the entire cluster.

Chaos builds on the streaming partitions introduced by X-Stream in order to achieve sequential access to storage, but parallelizes the execution of streaming partitions. Chaos is novel in three ways. First, Chaos partitions for sequential storage access, rather than for locality and load balance, resulting in much lower pre-processing times. Second, Chaos distributes graph data uniformly randomly across the cluster and does not attempt to achieve locality, based on the observation that in a small cluster network bandwidth far outstrips storage bandwidth. Third, Chaos uses work stealing to allow multiple machines to work on a single partition, thereby achieving load balance at runtime.

In terms of performance scaling, on 32 machines Chaos takes on average only 1.61 times longer to process a graph 32 times larger than on a single machine. In terms of capacity scaling, Chaos is capable of handling a graph with 1 *trillion* edges representing 16 TB of input data, a new milestone for graph processing capacity on a small commodity cluster.

* This work was done when Amitabha Roy was at EPFL.

SOSP'15, October 4–7, 2015, Monterey, CA.

ACM 978-1-4503-3834-9/15/10...$15.

http://dx.doi.org/10.1145/2815400.2815408

1. Introduction

Processing large graphs is an application area that has attracted significant interest in the research community [10, 13, 14, 16–18, 21, 25, 26, 28, 30–32, 34]. Triggered by the availability of graph-structured data in domains ranging from social networks to national security, researchers are exploring ways to mine useful information from such graphs. A serious impediment to this effort is the fact that many graph algorithms exhibit irregular access patterns [20]. As a consequence, most graph processing systems require that the graphs fit entirely in memory, necessitating either a supercomputer or a very large cluster [13, 14, 25, 26].

Systems such as GraphChi [18], GridGraph [34] and X-Stream [31] have demonstrated that it is possible to process graphs with edges in the order of billions on a single machine, by relying on secondary storage. This approach considerably reduces the entry barrier to processing large graphs. Such problems no longer require the resources of very large clusters or supercomputers. Unfortunately, the amount of storage that can be attached to a single machine is limited, while graphs of interest continue to grow [30]. Furthermore, the performance of a graph processing system based on secondary storage attached to a single machine is limited by its bandwidth to secondary storage [22].

In this paper we investigate how to scale out graph processing systems based on secondary storage to multiple machines, with the dual goals of increasing the size of graphs they can handle, to the order of a trillion edges, and improving their performance, by accessing secondary storage on different machines in parallel.

The common approach for scaling graph processing to multiple machines is to first statically partition the graph, and then to place each partition on a separate machine, where the graph computation for that partition takes place. Partitioning aims to achieve load balance to maximize parallelism and locality to minimize network communication. Achieving high-quality partitions that achieve these two goals can be time-consuming, especially for out-of-core graphs. Optimal partitioning is NP-hard [12], and even approximate algorithms may take considerable running time. Also, static partitioning

cannot cope with later changes to the graph structure or variations in access patterns over the course of the computation.

Chaos takes a fundamentally different approach to scaling out graph processing on secondary storage. First, rather than performing an elaborate partitioning step to achieve load balance and locality, Chaos performs a very simple initial partitioning to achieve sequential storage access. It does this by using a variant of the streaming partitions introduced by X-Stream [31]. Second, rather than locating the data for each partition on a single machine, Chaos spreads all graph data (vertices, edges and intermediate data, known as updates) uniformly randomly over all secondary storage devices. Data is stored in large enough chunks to maintain sequential storage access. Third, since different streaming partitions can have very different numbers of edges and updates, and therefore require very different amounts of work, Chaos allows more than one machine to work on the same streaming partition, using a form of work stealing [8] for balancing the load between machines.

These three components together make for an efficient implementation, achieving sequentiality, I/O load balance and computational load balance, while avoiding lengthy pre-processing due to elaborate partitioning. It is important to point out what Chaos does not do: it does not attempt to achieve locality. A fundamental assumption underlying the design of Chaos is that machine-to-machine network bandwidth exceeds the bandwidth of a storage device and that network switch bandwidth exceeds the aggregate bandwidth of all storage devices in the cluster. Under this assumption the network is never the bottleneck. Locality is then no longer a primary concern, since data can be streamed from a remote device at the same rate as from a local device. This assumption holds true for clusters of modest size, in which machines, even with state-of-the art SSDs, are connected by a commodity high-speed network, which is the environment targeted by Chaos. Recent work on datacenter networks suggests that this assumption also holds on a larger scale [15, 27, 29].

We evaluate Chaos on a cluster of 32 machines with ample secondary storage and connected by a high-speed network. We are able to scale up the problem size by a factor of 32, going from 1 to 32 machines, with on average only a 1.61X increase in runtime. Similarly, for a given graph size, we achieve speedups of 10 to 22 on 32 machines. The aggregated storage also lets us handle a graph with a trillion edges. This result represents a new milestone for graph processing systems on small commodity clusters. In terms of capacity it rivals those from the high performance computing community [1] and very large organizations [2] that place the graph on supercomputers or in main memory on large clusters. Chaos therefore enables the processing of very large graphs on rather modest hardware.

We also examine the conditions under which good scaling occurs. We find that sufficient network bandwidth is critical, as it underlies the assumption that locality has little effect. Once sufficient network bandwidth is available, performance improves more or less linearly with available storage bandwidth. The number of cores per processor has little or no effect, as long as enough cores are available to sustain high network bandwidth.

The contributions of this paper are:

- We build the first efficient scale-out graph processing system from secondary storage.
- Rather than expensive partitioning for locality and load balance, we use a very cheap partitioning scheme to achieve sequential access to secondary storage, leading to a short pre-processing time.
- We do not aim for locality in storage access, and we achieve I/O load balance by randomizing[1] data location and access.
- We allow multiple machines to work on the same partition in order to achieve computational load balance through randomized work stealing.
- We demonstrate that the system achieves its goals in terms of capacity and performance on a cluster of 32 machines.

The outline of the rest of this paper is as follows. In Section 2 we describe the GAS programming model used by Chaos. Partitioning and pre-processing are covered in Section 3. In Section 4 we give an overview of the Chaos design, followed by a detailed discussion of the computation engine in Section 5, the storage engine in Section 6, and some implementation details in Section 7. We describe the environment used for the evaluation in Section 8, present scaling results for Chaos in Section 9, and explore the impact of some of the design decisions in Section 10. Finally, we present related work in Section 11, before concluding in Section 12.

2. Programming Model

Chaos adopts an edge-centric and somewhat simplified GAS (Gather-Apply-Scatter) model [10, 13, 31].

The state of the computation is stored in the value field of each vertex. The computation takes the form of a loop, each iteration consisting of a scatter, gather and apply phase. During the scatter phase, updates are sent over edges. During the gather phase, updates arriving at a vertex are collected in that vertex's accumulator. During the apply phase these accumulators are applied to produce a new vertex value. The

[1] The extensive use of randomization instead of any order is the reason for naming the system Chaos

```
while not done
  // Scatter
  for all e in Edges
    u = new update
    u.dst = e.dst
    u.value = Scatter(e.src.value)
  // Gather
  for all u in Updates
    u.dst.accum = Gather(u.dst.accum, u.value)
  // Apply
  for all v in Vertices
    Apply(v.value, v.accum)
```

Figure 1: GAS Sequential Computation Model

```
// Scatter
function Scatter(value val)
  return val.rank / val.degree

// Gather
function Gather(accum a, value val)
  return a + val

// Apply
function Apply(value val, accum a)
  val.rank = 0.15 + 0.85 * a
```

Figure 2: Pagerank using Chaos

```
// Apply
for all v in Vertices
  for all replicas v' of v
    Apply(v.value, v'.accum)
```

Figure 3: Apply in Distributed Computation Model

precise nature of the computation in each of these phases is specified by three user-defined functions, Gather, Apply and Scatter, which are called by the Chaos runtime as necessary.

Figure 1 provides pseudo-code for the overall computation. During the scatter phase, for each edge, the Scatter function is called, taking as argument the vertex value of the source vertex of the edge, and returning the value of an update sent to the destination vertex of the edge. During the gather phase, for each update, the Gather function is called, updating the accumulator value of the destination vertex of the update using the value supplied with the update. Finally, during the apply phase, for each vertex, the Apply function is called, applying the value of the accumulator to compute the new vertex value. Figure 2 shows, for example, how Pagerank is implemented in the GAS model.

When executing on multiple machines, vertices may be replicated to achieve parallelism. For each replicated vertex there is a master. Edges or updates are never replicated. During the scatter phase parallelism is achieved by processing edges (and producing updates) on different machines. The update phase is distributed by each replica of a vertex gathering a subset of the updates for that vertex in its local accumulator. The apply phase then consists of applying all these accumulators to the vertex value (see Figure 3).

The edge-centric nature of the programming model is evidenced by the iteration over edges and updates in the scatter and gather phases, unlike the vertex-centric model [21], in which the scatter and gather loops iterate over vertices. This model is inherited from X-Stream, and has been demonstrated to provide superior performance for graph processing from secondary storage [31]. The GAS model was introduced by PowerGraph, and naturally expresses distributed graph processing, in which vertices may be replicated [13]. Finally, Chaos follows the simplifications of the GAS model introduced by PowerLyra [10], scattering updates only over outgoing edges and gathering updates only for incoming edges.

As in other uses of the GAS model, Chaos expects the final result of multiple applications of any of the user-supplied functions Scatter, Gather and Apply to be independent of the order in which they are applied in the scatter, gather and apply loops respectively. Chaos takes advantage of this order-independence to achieve an efficient solution. In practice, all our algorithms satisfy this requirement, and so we do not find it to be a limitation.

3. Streaming Partitions

Chaos uses a variation of X-Stream's [31] streaming partitions to achieve efficient sequential secondary storage access. A streaming partition of a graph consists of a set of vertices that fits in memory, all of their outgoing edges and all of their incoming updates. Executing the scatter and gather phases one streaming partition at a time allows sequential access to the edges and updates, while keeping all (random) accesses to the vertices in memory.

In X-Stream the size of the vertex set of a streaming partition is – allowing for various auxiliary data structures – equal to the size of main memory. This choice optimizes sequential access to edges and updates, while keeping all accesses to the vertex set in memory. In a distributed setting other considerations play a role into the proper choice for the size of the vertex set. Main memory size remains an upper bound in order to guarantee in-memory access to the vertex set, and large sizes facilitate sequential access to edges and updates, but smaller sizes are desirable, as they lead to easier load balancing.

Therefore, we choose the number of partitions to be the smallest multiple of the number of machines such that the vertex set of each partition fits into memory. We simply partition the vertex set in ranges of consecutive vertex identifiers. Edges are partitioned such that an edge belongs to the partition of its source vertex.

This partitioning is the only pre-processing done in Chaos. It requires one pass over the edge set, and a negligible amount of computation per edge. Furthermore, it can easily be parallelized by splitting the input edge list evenly across ma-

chines. This low-cost pre-processing stands in stark contrast to the elaborate partitioning algorithms that are typically used in distributed graph processing systems. These complex partitioning strategies aim for static load balance and locality [13]. Chaos dispenses with locality entirely, and achieves load balance at runtime.

4. Design overview

Chaos consists of a computation sub-system and a storage sub-system.

The storage sub-system consists of a storage engine on each machine. It supplies vertices, edges and updates of different partitions to the computation sub-system. The vertices, edges and updates of a partition are uniformly randomly spread over the different storage engines.

The computation sub-system consists of a number of computation engines, one per machine. The computation engines collectively implement the GAS model. Unlike the conceptual model described in Section 2, the actual implementation of the model in Chaos has only two phases per iteration, a scatter and a gather phase. The apply phase is incorporated into the gather phase, for reasons of efficiency. There is a barrier after each scatter phase and after each gather phase. The Apply function is executed as needed during the gather phase, and does not imply any global synchronization.

The Chaos design allows multiple computation engines to work on a single partition at the same time, in order to achieve computational load balance. When this is the case, it is essential that each engine read a disjoint set of edges (during the scatter phase) or updates (during the gather phase). This responsibility rests with the storage sub-system. This division of labor allows multiple computation engines to work on the same partition without synchronization between them.

The protocol between the computation and storage engines is designed such that all storage devices are kept busy all the time, thereby achieving maximum utilization of the bottleneck resource, namely the bandwidth of the storage devices.

5. Computation sub-system

The number of streaming partitions is a multiple k of the number of computation engines. Therefore, each computation engine is initially assigned k partitions. This engine is the master for all vertices of those partitions, or, for short, the master of those partitions.

We start by describing the computation in the absence of work stealing. This aspect of Chaos is similar to X-Stream, but is repeated here for completeness. Later, we show how

```
// Scatter for partition P                                          1
function exec_scatter(P)                                            2
  for each unprocessed e in Edges(P)                                3
    u = new update                                                  4
    u.dst = e.dst                                                   5
    u.value = Scatter(e.src.value)                                  6
    add u to Updates(partition(u.dst))                              7
                                                                    8
// Gather for partition P                                           9
function exec_gather(P)                                             10
  for each unprocessed u in Updates(P)                              11
    u.dst.accum = Gather(u.dst.accum, u.value)                      12
                                                                    13
/////// Chaos compute engine                                       14
                                                                    15
// Pre-processing                                                   16
for each input edge e                                               17
  add e to Edges(partition(e.src))                                  18
                                                                    19
// Main loop                                                        20
while not done                                                      21
                                                                    22
  // Scatter phase                                                  23
  for each of my partitions P                                       24
    load Vertices(P)                                                25
    exec_scatter(P)                                                 26
                                                                    27
  // When done with my partitions, steal from others               28
  for every partition P_Stolen not belonging to me                  29
    if need_help(Master(P_Stolen))                                  30
      load Vertices(P_Stolen)                                       31
      exec_scatter(P_Stolen)                                        32
  global_barrier()                                                  33
                                                                    34
  // Gather Phase                                                   35
  for each of my partitions P                                       36
    load Vertices(P)                                                37
    exec_gather(P)                                                  38
                                                                    39
    // Apply Phase                                                  40
    for all stealers s                                              41
      accumulators = get_accums(s)                                  42
      for all v in Vertices(P)                                      43
        Apply(v.value, accumulators(v))                             44
    delete Updates(P)                                               45
                                                                    46
  // When done with my partitions, steal from others               47
  for every partition P_Stolen not belonging to me                  48
    if need_help(Master(P_Stolen))                                  49
      load Vertices(P_Stolen)                                       50
      exec_gather(P_Stolen)                                         51
      wait for get_accums(P_Stolen)                                 52
  global_barrier()                                                  53
```

Figure 4: Chaos Computation Engine

Chaos implements work stealing between computation engines. The complete pseudo-code description of the computation engine (including stealing) is shown in Figure 4.

5.1 Scatter phase

Each computation engine works on its assigned partitions, one at a time, moving from one of its assigned partition to the next (lines 23–33) without any global synchronization between machines. The vertex set of the partition is read into memory, and then the edge set is streamed into a large main memory buffer. As edges are processed, updates may be produced. These updates are binned according to the partition of their target vertex, and buffered in memory. When a buffer is full, it is written to storage. Multiple buffers

are used, both for reading edges and writing updates, in order to overlap computation and I/O.

5.2 The gather phase

Each computation engine works on its assigned partitions, one at a time, moving from one of its assigned partition to the next (lines 35–45) without any global synchronization between machines. The vertex set of the partition is read into memory, and then the update set is streamed into a large main memory buffer. As updates are processed, the accumulator of the destination vertex is updated. Multiple buffers are used for reading updates in order to overlap computation and I/O.

5.3 Work stealing

The number of edges or updates to be processed may differ greatly between partitions, and therefore between computation engines. Chaos uses work stealing to even out the load, as described next.

When computation engine i completes the work for its assigned partitions (lines 23–26 for scatter and lines 35–38 for gather), it goes through every partition p (for which it is not the master) and sends a proposal to help out with p to its master j (line 30 for scatter and line 49 for gather). Depending on how far along j is with that partition, it accepts or rejects the proposal, and sends a response to i accordingly. In the case of a negative answer, engine i continues to iterate through the other partitions, each time proposing to help. It does so until it receives a positive response or until it has determined that no help is needed for any of the partitions. In the latter case its work for the current scatter or gather phase is finished, and it waits at a barrier (line 33 for scatter and line 53 for gather).

When engine i receives a positive response to help out with partition p, it reads the vertex set of that partition from storage into its memory, and starts working on it. When two or more engines work on the same partition, it is essential that they work on a disjoint set of edges (during scatter) or updates (during gather). Chaos puts this responsibility with the storage system: it makes sure that in a particular iteration an edge or an update is processed only once, independent of how many computation engines work on the partition to which that edge or update belongs. This is easy to do in the storage system, and avoids the need for synchronization between the computation engines involved in stealing.

For stealing during scatter, a computation engine proceeds exactly as it does for its own partitions. Using the user-supplied `Scatter` function, the stealer produces updates into in-memory buffers and streams them to storage when the buffers become full.

Stealing during gather is more involved. As before, a computation engine reads updates from storage, and uses the user-supplied `Gather` function to update the accumulator of the destination vertex of the update. There are now, however, multiple instances of the accumulator for this vertex, and their values need to be combined before completing the gather phase. To this end the master of the partition keeps track of which other computation engines have stolen work from it for this partition. When the master completes its part of the gather for this partition, it sends a request to all those computation engines, and waits for an answer. When a stealer completes its part of the gather, it waits to receive a request for its accumulator from the master, and eventually sends it to the master (line 52). The master then uses the user-supplied `Apply` function to compute the new vertex values from these different accumulators, and writes the vertex set back to storage.

The order in which the master and the stealers complete their work is unpredictable. When a stealer completes its work before the master, it waits until the master requests its accumulator values before it does anything else (line 52). When the master completes its work before one or more of the stealers, it waits until those stealers return their accumulator (line 42).

On the plus side, this approach guarantees that all accumulators are in memory at the time the master performs the apply. On the minus side, there may be some amount of time during which a computation engine remains idle. An alternative would have been for an engine that has completed its work on a partition to write its accumulators to storage, from where the master could later retrieve them. This strategy would allow an engine to immediately start work on another partition. The idle time in our approach is, however, very short, because all computation engines that work on the same partition read from the same set of updates, and therefore all finish within a very short time of one another. We therefore prefer this simple and efficient in-memory approach over more complicated ones, such as writing the accumulators to storage, or interrupting the master to incorporate the accumulators from stealers.

5.4 To steal or not to steal

Stealing is helpful if the cost, the time for the stealer to read in the vertex set, is smaller than the benefit, the reduction in processing time for the edges or updates still to be processed at the time the stealer joins in the work. Since Chaos is I/O-bound, this decrease in processing time can be estimated by the decrease in I/O time caused by the stealer.

This estimate is made by considering the following quantities: B is the bandwidth to storage seen by each computation engine, D is the amount of edge or update data remaining to be read for processing the partition, H is the number of

computation engines currently working on the partition (including the master), and V is the size of the vertex state of the partition.

If the master declines the stealing proposal, the remaining time to process this partition is $\frac{D}{BH}$. If the master accepts the proposal, then $\frac{V}{B}$ time is required to read the vertex set of size V. Since we assume that bandwidth is limited by the storage engines and not by the network, an additional helper increases the bandwidth from BH to $B(H+1)$, and decreases the remaining processing time from $\frac{D}{BH}$ to $\frac{D}{B(H+1)}$. The master therefore accepts the proposal if and only if:

$$\frac{V}{B} + \frac{D}{B(H+1)} < \frac{D}{BH} \quad (1)$$

$$\Longrightarrow V + \frac{D}{(H+1)} < \frac{D}{H} \quad (2)$$

The master knows the size of the vertex set V, and keeps track of the number of stealers H. It estimates the value of D by multiplying the amount of edge or update data still to be processed on the local storage engine by the number of machines. Since the data is evenly spread across storage engines, this estimate is accurate and makes the decision process local to the master. This stealing criterion is incorporated in `need_help()` on lines 30 and 49 of the pseudo-code in Figure 4.

6. Storage sub-system

For an out-of-core graph processing system such as Chaos, computation is only one half of the system. The other half consists of the storage sub-system that supplies the I/O bandwidth necessary to move graph data between storage and main memory.

6.1 Stored data structures and their access patterns

For each partition, Chaos records three data structures on storage: the vertex set, the edge set and the update set. The accumulators are temporary structures, and are never written to storage.

The access patterns of the three data structures are quite different. Edge sets are created during pre-processing and are read during scatter.[2] Update sets are created and written to storage during scatter, and read during gather. After the end of a gather phase, they are deleted. Vertex sets are initialized during pre-processing, and always read in their entirety, both during scatter and gather. Read and write operations to edges and updates may be performed by the master or by any stealers. In contrast, read operations to the vertex state may be performed by the master or any stealers, but only the master updates the vertex values during apply and writes them back to storage.

[2] In an extended version of the model, edges may also be rewritten during the computation.

6.2 Chunks

One of the key design decisions in Chaos is to spread all data structures across the storage engines in a random uniform manner, without worrying about locality.

All data structures are maintained and accessed in units called chunks. The size of a chunk is chosen large enough so that access to storage appears sequential, but small enough so that they can serve as units of distribution to achieve random uniform distribution across storage engines. Chunks are also the "unit of stealing", the smallest amount of work that can be stolen. Therefore, to achieve good load balance, they need to be relatively small.

A storage engine always serves a request for a chunk in its entirety before serving the next request, in order to maintain sequential access to the chunk, despite concurrent requests to the storage engine.

6.3 Edge and update sets

Edges and updates are always stored and retrieved one chunk at a time.

Edges are stored during pre-processing, and updates during the scatter phase, but Chaos uses the same mechanism to choose a storage engine on which to store a chunk. It simply picks a random number uniformly distributed between 1 and the number of storage engines, and stores the chunk there.

To retrieve a chunk of edges or updates, a computation engine similarly picks a random number between 1 and the number of machines, and sends the request to that storage engine. In its request it specifies a partition identifier, but it does not specify a particular chunk.

When a storage engine receives a request for a chunk for a given partition, it checks if it still has unprocessed chunks for this partition, and, if so, it is free to return any unprocessed chunk. This approach is in keeping with the observation that the order of edges or updates does not matter, and therefore it does not matter in which order chunks of edges or updates are processed. For both edges and updates, it is essential, however, that they are read only once during an iteration. To this end, a storage engine keeps track of which chunks have already been consumed during the current iteration.

If all chunks for this partition on this storage engine have already been processed, the storage engine indicates so in the reply to the computation engine. A computation engine knows that the input for the current streaming partition is empty when all storage engines fail to supply a chunk.

6.4 Vertex sets

Vertex sets are always accessed in their entirety, but they are also stored as chunks. For vertices, the chunks are mapped to storage engines using the equivalent of hashing on the partition identifier and the chunk number. During pre-processing the chunks of the vertex set of a partition are stored at storage engines in this manner. During the execution of the main computation loop the computation engines wishing to read or write the vertex set use the same approach to find the storage engines storing these chunks, and request the next chunk from this storage engine.

6.5 Keeping all storage engines busy

As described, a computation engine only issues one request at a time to the storage system. Although randomization, on average, spreads such requests evenly across storage engines, the lack of coordination can cause significant inefficiencies due to some storage engines becoming instantaneously idle. With an equal number of computation and storage engines, and with storage and not computation being the bottleneck, there are always as many requests as there are storage engines. Without coordination between computation engines, several of them may address their request to the same storage engine, leaving other storage engines idle.

To keep all storage engines busy with high probability, each computation engine keeps, at all times, multiple requests to different storage engines outstanding. The number of such outstanding requests, called the batch factor k, is chosen to be the smallest number that with high probability keeps all storage engines busy all the time. The proper batch factor is derived as follows.

If a computation engine has k requests outstanding, then only some fraction are being processed by the storage sub-system. The other requests are in transit. To ensure there are k outstanding requests at the storage engines, the computation engines use a larger *request window* ϕk. This amplification factor ϕ can easily be computed by repeated application of Little's law [19]:

$$k = \lambda R_{storage}$$
$$\phi k = \lambda (R_{storage} + R_{network})$$

where λ is the throughput of the storage engine in terms of requests per unit time, $R_{storage}$ is the latency for the storage engine to service the request and $R_{network}$ is the latency of a round trip on the network. Solving we have the required amplification ϕ:

$$\phi = 1 + \frac{R_{network}}{R_{storage}} \tag{3}$$

With this choice of ϕ, we end up with k outstanding requests from each of m computation engines distributed at random across the m storage engines. We can then derive the utilization of a particular storage engine as follows. The probability that a storage engine is un-utilized is equal to the probability that no computation engine picks it for any of its k requests:

$$\left(\frac{C_k^{m-1}}{C_k^m}\right)^m = (1 - \frac{k}{m})^m$$

The utilization of the storage engine is therefore the probability that at least one computation engine picks it, a function of the number of machines m and the batch-factor k:

$$\rho(m,k) = 1 - (1 - \frac{k}{m})^m \tag{4}$$

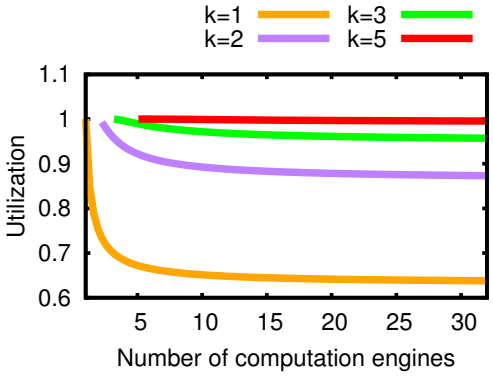

Figure 5: Theoretical utilization for different number of machines as a function of the batch factor k

Figure 5 shows the utilization as a function of the number of machines and for various values of k. For a fixed value of k, the utilization reduces with an increasing number of machines due to a greater probability of machines being left idle but *is asymptotic to a lower bound*. The lower bound is simply:

$$\lim_{m\to\infty} \rho(m,k) = 1 - \frac{1}{e^k} \tag{5}$$

It therefore suffices to pick a value for k large enough to approach 100% utilization regardless of the number of machines. For example, using $k = 5$ means that the utilization cannot drop below 99.3%.

6.6 Fault tolerance

The fact that all graph computation state is stored in the vertex values, combined with the synchronous nature of the computation, allows Chaos to tolerate transient machine failures in a simple and efficient manner. At every barrier at the end of a scatter or gather phase, the vertex values are checkpointed by means of a 2-phase protocol [11] that makes sure that the new values are completely stored before the old values are removed.

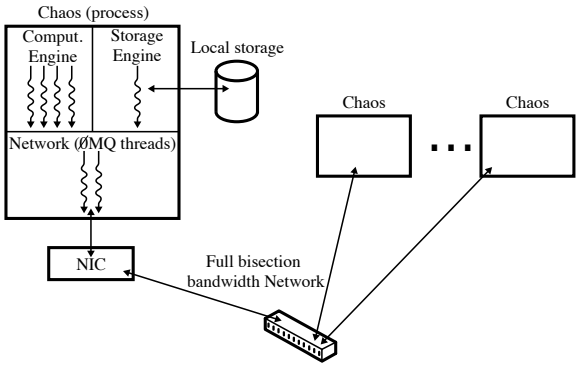

Figure 6: Architecture of Chaos.

In its current implementation Chaos does not support recovery from storage failures, although such support could easily be added by replicating the vertex sets.

7. Implementation

Chaos is written in C++ and amounts to approximately 15'000 lines of code. Figure 6 shows the high-level architecture and typical deployment of Chaos.

We run the computation engine and the storage engine on each machine in separate threads within the same process. We use ØMQ [3] on top of TCP sockets for message-oriented communication between computation and storage engines, assuming a full bisection bandwidth network between the machines. We tune the number of ØMQ threads for optimal performance.

The storage engines provide a simple interface to the local ext4 [23] file system. Unlike X-Stream, which uses direct I/O, Chaos uses pagecache-mediated access to the storage devices. On each machine, for each streaming partition, the vertex, edge and update set correspond to a separate file. A read or write causes the file pointer to be advanced. The file pointer is reset to the beginning of the file at the end of each iteration. This provides a very simple implementation of the requirement that edges or updates are read only once during an iteration. The chunk corresponds to a 4MB block in the file, leading to good sequentiality and performance.

8. Experimental environment and benchmarks

We evaluate Chaos on a rack with 32 16-core machines, each equipped with 32 GB of main memory, a 480GB SSD and 2 6TB magnetic disks (arranged in RAID 0). Unless otherwise noted, the experiments use the SSDs as storage devices. The machines are connected through 40 GigE links to a top-of-rack switch. The SSDs and disks provide bandwidth in the range of 400MB/s and 200MB/s, respectively, well within the capacity of the 40 GigE interface on the machine.

Algorithm	X-Stream	Chaos
Breadth-First Search (BFS)	497s	594s
Weakly Connected Comp. (WCC)	729s	995s
Min. Cost Spanning Trees (MCST)	1239s	2129s
Maximal Independent Sets (MIS)	983s	944s
Single Source Shortest Paths (SSSP)	2688s	3243s
Pagerank (PR)	884s	1358s
Strongly Connected Comp. (SCC)	1689s	1962s
Conductance (Cond)	123s	273s
Sparse Matrix Vector Mult. (SpMV)	206s	508s
Belief Propagation (BP)	601s	610s

Table 1: Algorithms, single-machine runtime for X-Stream and Chaos, SSD. The first five algorithms require an undirected graph while the remaining ones run on a directed graph.

We use the same set of algorithms as used by X-Stream [31] to demonstrate that all the single machine algorithms used in the evaluation of X-Stream can be scaled to our cluster. Table 1 presents the complete set of algorithms, as well as the X-Stream runtime and the *single machine* Chaos runtime for an RMAT-27 graph. As can be seen, the single-machine runtimes are similar but not exactly the same. In principle, Chaos running on a single machine is equivalent to X-Stream. The two systems have, however, different code bases and, in places, different implementation strategies. In particular, Chaos uses a client-server model for I/O, to facilitate distribution, and pagecache-mediated storage access, to simplify I/O for variable-sized objects.

We use a combination of synthetic RMAT graphs [9] and the real-world Data Commons dataset [4]. RMAT graphs can be scaled in size easily: a scale-n RMAT graph has 2^n vertices and 2^{n+4} edges. In other words, the size of the vertex and edge sets doubles with each increment in the scale factor. We use the newer 2014 version of the Data Commons graph that encompasses 1.7 billion webpages and 64 billion hyperlinks between them.

Input to the computation consists of an unsorted edge list, with each edge represented by its source and target vertex and an optional weight. If necessary, we convert directed to undirected graphs by adding a reverse edge. Graphs with fewer than 2^{32} vertices are represented in compact format, with 4 bytes for each vertex and for the weight, if any. Graphs with more vertices are represented in non-compact format, using 8 bytes instead. A scale-32 graph with weights on the edges thus results in 768 GB of input data. The input of the unweighted Data Commons graph is 1 TB.

All results report the wall-clock time to go from the unsorted edge list, randomly distributed over all storage devices, to the final vertex state, recorded on storage. All results therefore include pre-processing time.

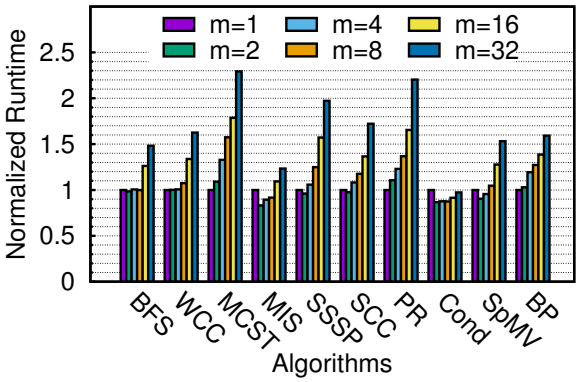

Figure 7: Runtime normalized to 1-machine runtime. Weak scaling, RMAT-27 to RMAT-32, SSD.

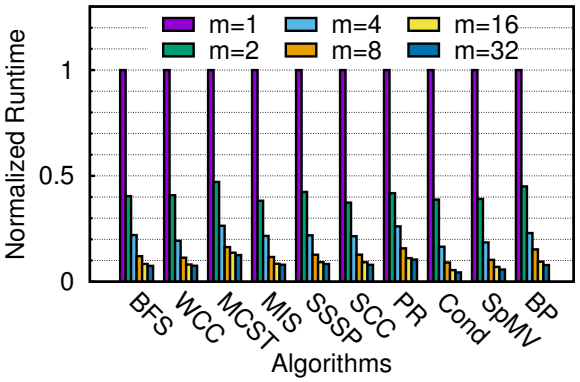

Figure 8: Runtime normalized to 1-machine runtime. Strong scaling, RMAT-27, SSD .

9. Scaling results

9.1 Weak scaling

In the weak scaling experiment we run RMAT-27 on one machine, and then double the size for each doubling of the number of machines, ending up with RMAT-32 on 32 machines.

Figure 7 shows the runtime results for these experiments, normalized to the runtime of a single machine. In this experiment, Chaos takes on average 1.61X the time taken by a single machine to solve a problem 32X the size on a single machine. The fastest algorithm (Cond) takes 0.97X, while the slowest (MCST) takes 2.29X.

The differences in scaling between algorithms result from a combination of characteristics of the algorithms, including the fact that the algorithm itself may not scale perfectly, the degree of load imbalance in the absence of stealing, and the size of the vertex sets. One interesting special case is Conductance, where the scaling factor is slightly smaller than 1. This somewhat surprising behavior is the result of the fact that with a larger number of machines the updates fit in the buffer cache and do not require storage accesses.

9.2 Strong scaling

In this experiment we run all algorithms on 1 to 32 machines on the RMAT-27 graph. Figure 8 shows the runtime, again normalized to the runtime on one machine. For this RMAT graph, 32 machines provide on average a speedup of about 13X over a single machine. The fastest algorithm (Cond) runs 23X faster and the slowest (MCST) 8X. The results are somewhat inferior to the weak scaling results, because of the small size of the graph.

To illustrate this, we perform a strong scaling experiment on the much larger Data Commons graph. This graph does not fit on a single SSD, so we use HDDs. Furthermore, given the long running times, we only present results for two representative algorithms, BFS and Pagerank. Figure 9 shows the runtimes on 1 to 32 machines, normalized to the single-machine runtime. Using 32 machines Chaos provides a speedup of 20 for BFS and 18.5 for Pagerank.

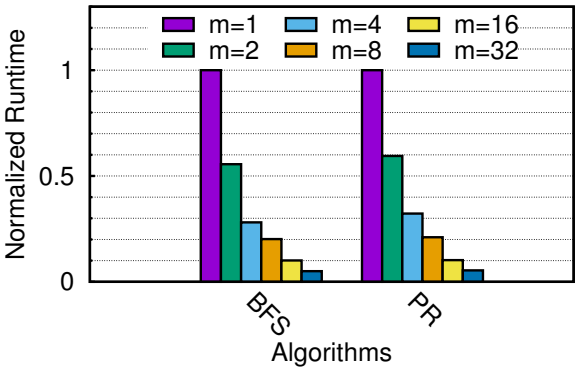

Figure 9: Runtime normalized to 1-machine runtime. Strong scaling, Data Commons, RAID0-HDD.

9.3 Capacity scaling

We use RMAT-36 with 250 billion vertices and 1 trillion edges to demonstrate that we can scale to very large graphs. This graph requires 16TB of input data, stored on HDDs. Chaos finds a breadth-first order of the vertices of the graph in a little over 9 hours. Similarly, Chaos runs 5 iterations of PR in 19 hours. These experiments require I/O in the range of 214 TB for BFS and 395 TB for PR, and the Chaos store is able to provide an aggregate of 7 GB/s from the 64 magnetic disks running in orchestration.

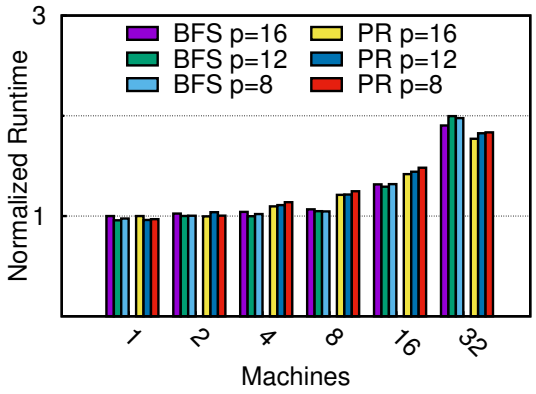

Figure 10: Runtime for Chaos with different number of CPU cores, normalized to 1-machine runtime with cores=16.

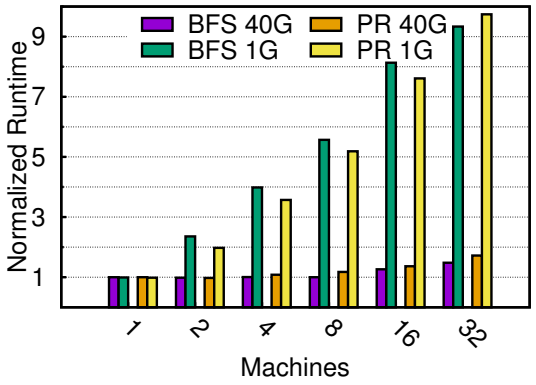

Figure 12: Runtime for Chaos with 1GigE and 40GigE, normalized to 1-machine runtime.

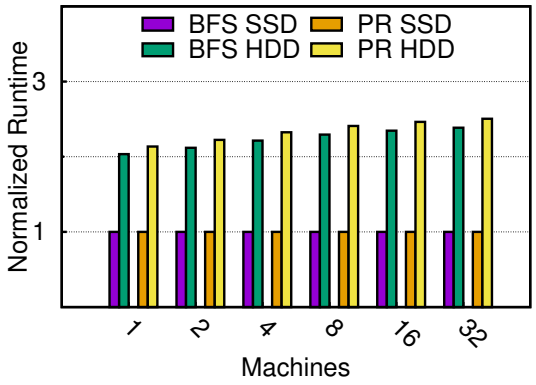

Figure 11: Runtime for Chaos with SSD and HDD, normalized to 1-machine runtime with SSD.

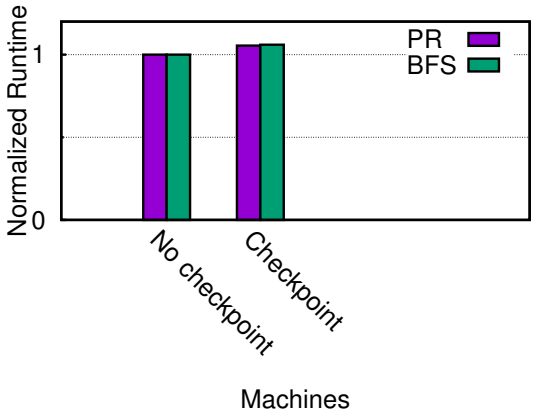

Figure 13: Chaos vs. Chaos with checkpointing enabled (32 machines, RMAT-35, HDD, normalized to Chaos runtime).

9.4 Scaling limitations

We evaluate the limitations to scaling Chaos with respect to the specific processor, storage bandwidth, and network links available.

Figure 10 presents the results of running BFS and PR as we vary the number of CPU cores available to Chaos. As can be seen, the system performs adequately even with half the CPU cores available. It is nevertheless worth pointing out that Chaos requires a minimum number of cores to maintain good network throughput.

Figure 11 compares the performance of BFS and PR when running from SSDs and HDDs. The HDD bandwidth is 2X less than the SSD bandwidth. Chaos scales as expected regardless of the bandwidth, but the application takes time inversely proportional to the available bandwidth.

Figure 12 looks into the performance impact of a slower network by using a 1GigE interface to connect all machines instead of the faster 40GigE. The throughput achieved by the 1GigE interface is approximately $1/4$th of the disk bandwidth. In other words, the network is the performance bottleneck. We conclude from these results that Chaos does not scale as well in such a situation, highlighting the need for network links which are faster (or at least as fast) as the storage bandwidth per machine.

9.5 Checkpointing

For large graph analytics problems, Chaos provides the ability to checkpoint state. Figure 13 shows the runtime overhead for checkpoints on a scale 36 graph for BFS and PR. As can be seen, the overhead is under 6% even though the executions write hundreds of terabytes of data to the Chaos store.

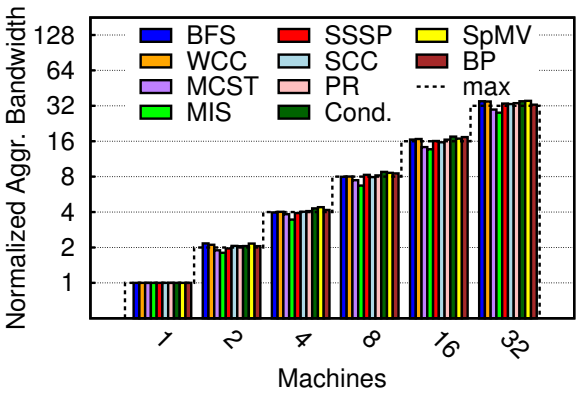

Figure 14: Aggregate bandwidth normalized to 1-machine bandwidth and maximum theoretical aggregate bandwidth.

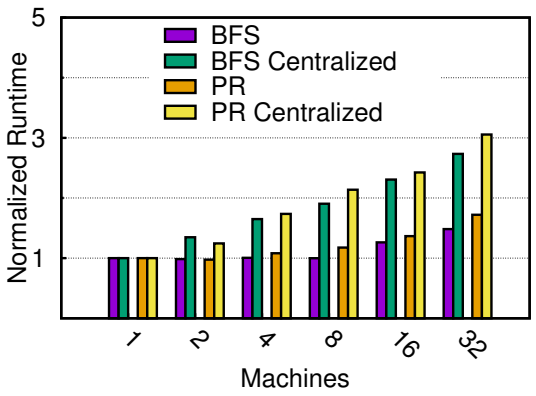

Figure 15: Chaos vs. centralized chunk directory (weak scaling, RMAT-27 to -32, SSD).

10. Evaluation of design decisions

Chaos is based on three synergistic principles: no attempt to achieve locality, dynamic instead of static load balancing, and simple partitioning. In this section we evaluate the effect of these design decisions. All discussion in this section is based on the weak scaling experiments. The effect of the design decisions for other experiments is similar and not repeated here. For some experiments we only show the results of BFS and Pagerank as representative algorithms.

10.1 No locality

Instead of seeking locality, Chaos spreads all graph data uniformly randomly across all storage devices and uses a randomized procedure to choose from which machine to read or to which machine to write data.

Figure 14 shows the aggregate bandwidth obtained as seen by all computation engines during the weak scaling experiment. The figure also shows the maximum bandwidth of the storage devices, measured by fio [5].

Two conclusions can be drawn from these results. First, the aggregate bandwidth achieved by Chaos scales linearly with the number of machines. Second, the bandwidth achieved by Chaos is within 3 percent of the available storage bandwidth, the bottleneck resource in the system.

We also evaluate a couple of more detailed design choices in terms of storage access, namely the randomized selection of storage device and the batching designed to keep all storage devices busy.

Figure 15 compares the runtime for Pagerank on 1 to 32 machines for Chaos to a design where a centralized entity selects the storage device for reading and writing a chunk. In short, all read and writes go through the centralized entity, which maintains a directory of where each chunk of each vertex, edge or update set is located. As can be seen, the running time with Chaos increases more slowly as a function of the number of machines than with the centralized entity, which increasingly becomes a bottleneck.

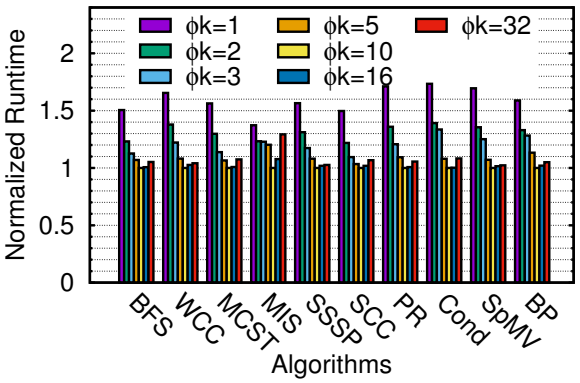

Figure 16: Runtime as a function of batch factor (32 machines, RMAT-32, SSD) normalized to Chaos ($\phi k = 10$).

Next we evaluate the efficacy of batching in our disk selection strategy. Figure 16 shows the effect of increasing the window size of outstanding requests on performance. We measured the latency to the SSD to be approximately equal to that on the 40 GigE network. This means $\phi = 2$ (Equation 3). The graph shows a clear sweet spot at $\phi k = 10$, which corresponds to $k = 5$. This means an utilization of 99.56% with 32 machines (Equation 4), indicating that the devices are near saturation. The experiment therefore agrees with theory. Further, Equation 5 tells us that even if we increase the number of machines in the deployment, this choice of settings means that we cannot drop below 99.3% given a fixed latency on the network. The increased runtime past this choice of settings can be attributed to increased queuing delays and incast congestion.

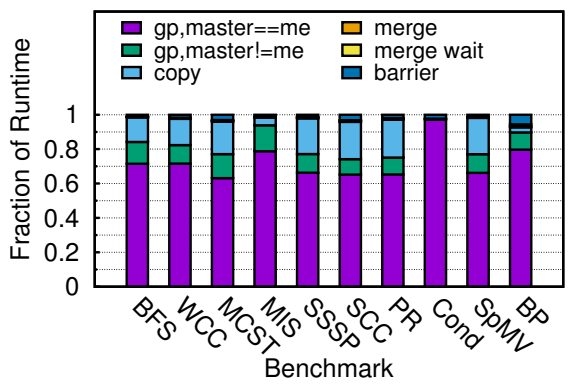

Figure 17: Breakdown of runtime (32 machines, RMAT-32, SSD).

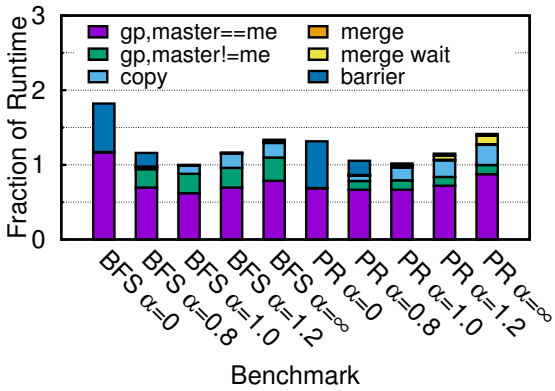

Figure 18: Breakdown of runtime with work-stealing bias. 32 machines, RMAT-32, normalized to $\alpha = 1$.

10.2 Dynamic load balancing

Chaos balances the load between different computation engines by randomized work stealing.

Figure 17 shows a breakdown of the runtime of the weak scaling experiments at 32 machines in three categories: graph processing time, idle time, and time spent copying and merging. The first category represents useful work, broken down further into processing time for the partitions for which the machine is the master and processing time for partitions initially assigned to other machines. The idle time reflects load imbalance, and the copying and merging time represents the overhead of achieving load balance.

The results are somewhat different for different algorithms. The processing time ranges from 74 to 87 percent with an average of 83 percent. The idle time is very low for all algorithms, below 4 percent. The cost of copying and merging varies considerably, from 0 to 22 percent with an average of 14 percent. Most of the idle time occurs at barriers between phases. Overall, we conclude from Figure 17 that load balancing is very good, but comes at a certain cost for some of the algorithms.

Next, we evaluate the quality of the decisions made by the stealing criterion we describe in Section 5.3. To do this, we introduce a factor α in Equation 2 as follows:

$$\frac{V}{B} + \frac{D}{B(H+1)} < \alpha \frac{D}{BH}$$

Varying the factor α allows us to explore a range of strategies.

- No stealing: $\alpha = 0$
- Less aggressive stealing: $\alpha = 0.8$
- Chaos default: $\alpha = 1$
- More aggressive stealing: $\alpha = 1.2$
- Always steal: $\alpha = \infty$

Figure 18 shows the running times for BFS and Pagerank. The results clearly show that Chaos (with α=1) obtains the best performance - providing support to the reasoning of Section 5.4.

As additional evidence for the need for dynamic load balancing, we compare the performance of Chaos to that of Giraph [6], an open-source implementation of Pregel, recently augmented with support for out-of-core graphs. Giraph uses a random partitioning of the vertices to distribute the graph across machines, without any attempt to perform any dynamic load balancing (similar to the experiment reported in Figure 18, with α equal to zero).

Out-of-core Giraph is an order of magnitude slower than Chaos in runtime, apparently largely due to engineering issues (in particular, JVM overheads in Giraph). To eliminate these differences and to focus on scalability, Figure 19 shows the runtime of both Chaos and Giraph on Pagerank on RMAT-27, normalized to the single-machine runtime for each system. The results clearly confirm that the static partitions in Giraph severely affect scalability.

10.3 Partitioning for sequentiality rather than for locality and load balance

An important question to ask is whether it would have been better to expend pre-processing time to generate high-quality partitions to avoid load imbalance in the first place instead of paying the cost of dynamic load balancing. To answer this question, we compare, for each algorithm and for 32 machines, the worst-case dynamic load balancing cost across all machines to the time required to initially partition the graph. We use Powergraph's [13] grid partitioning algorithm, which requires the graph to be in memory. We lack the necessary main memory in our cluster to fit the RMAT scale-32 graph,

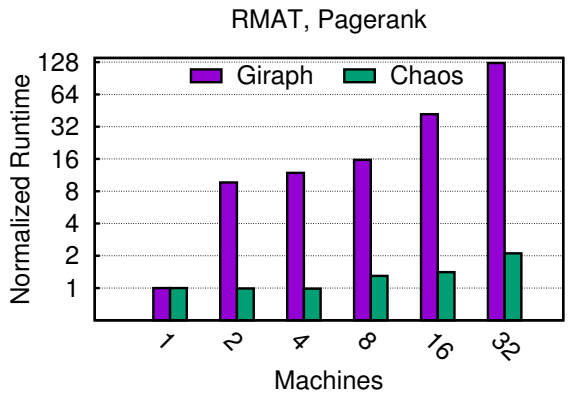

Figure 19: Runtime for Chaos and Giraph, normalized to the 1-machine runtime of each system.

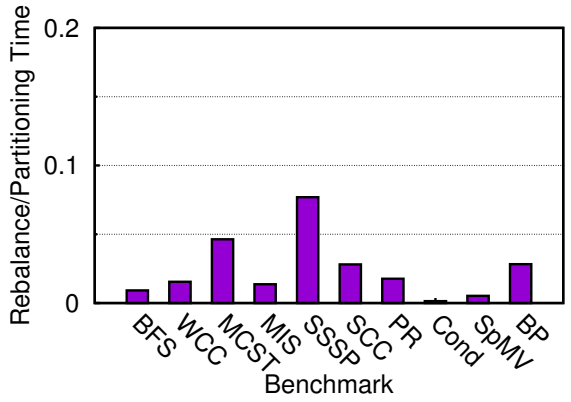

Figure 20: Runtime for Chaos dynamic load balancing vs. PowerGraph static partitioning (RMAT-27)

that Chaos uses on 32 machines. We therefore run the Powergraph grid partitioning algorithm on a smaller graph (RMAT scale-27), and assume that the partitioning time for Powergraph scales perfectly with graph size. As Figure 20 shows, Chaos dynamic load balancing out-of-core takes only a tenth of the time required by Powergraph to partition the graph in memory. From this comparison, carried out in circumstances highly favorable to partitioning, it is clear that dynamic load balancing in Chaos is more efficient than upfront partitioning in Powergraph. Chaos therefore achieves its goal of providing high performance graph processing, while avoiding the need for high-quality partitions.

11. Related Work

In recent years a large number of graph processing systems have been proposed [10, 13, 14, 16–18, 21, 25, 26, 28, 30–32, 34]. We mention here only those most closely related to our work. We also mention some general-purpose distributed systems techniques from which we draw.

11.1 Distributed in-memory systems

Pregel [21] and its open-source implementation Giraph [6] follow an edge-cut approach. They partition the vertices of a graph and place each partition on a different machine, potentially leading to severe load imbalance. Mizan [17] addresses this problem by migrating vertices between iterations in the hope of obtaining better load balance in the next iteration. Chaos addresses load imbalance within each iteration, by allowing more than one machine to work on a partition, if needed. Pregel optimizes network traffic by aggregating updates to the same vertex. While this optimization is also possible in Chaos, we find that the cost of merging the updates to the same vertex outweighs the benefits from reduced network traffic.

Powergraph proposes the GAS model, and PowerLyra [10] introduces a simpler variant, which we adopt in Chaos. Powergraph [13] introduces the vertex-cut approach, partitioning the set of edges across machines and replicating vertices on machine that have an attached edge. PowerLyra improves on Powergraph by treating high- and low-degree nodes differently, reducing communication and replication. Both systems require lengthy pre-processing times. Also, in both systems, each partition is assigned to exactly one machine. In contrast, Chaos performs only minimal pre-processing, and allows multiple machines to work on the same partition.

Finally, a recent graph processing system called GraM [33] has shown how a graph with a trillion edges can be handled in the main memory of the machines in a cluster. Chaos represents a different approach where the graph is too large to be held in memory. Thus, while Chaos is slower than GrAM it requires only a fraction of the amount of main memory to process a similarly sized graph.

11.2 Single-machine out-of-core systems

GraphChi [18] was one of the first systems to propose graph processing from secondary storage. It uses the concept of parallel sliding windows to achieve sequential secondary storage access. X-Stream [31] improves on GraphChi by using streaming partitions to provide better sequentiality. In recent work, GridGraph [34] further improves on both GraphChi and X-Stream by reducing the amount of I/O necessary. Chaos extends out-of-core graph processing to clusters.

11.3 Distributed systems techniques

The work on flat datacenter storage (FDS) [27] shows how one could take our assumption of local storage bandwidth being the same as remote storage bandwidth and scale it out to an entire datacenter. Chaos is the first graph processing system that exploits this property but at the smaller scale of a rack of machines. Also, unlike FDS (and similar systems

such as CORFU [7]), we leverage the order-independence of our workload to remove the central bottleneck of a meta-data server.

The batching in Chaos is inspired by power-of-two scheduling [24], although the goal is quite different. Power-of-two scheduling aims to find the least loaded servers in order to achieve load balance. Chaos aims to prevent storage engines from becoming idle.

12. Conclusion

Chaos is a system for processing graphs from the aggregate secondary storage of a cluster. It extends the reach of small clusters to graph problems with edges in the order of trillions. With very limited pre-processing, Chaos achieves sequential storage access, computational load balance and I/O load balance through the application of three synergistic techniques: streaming partitions adapted for parallel execution, flat storage without a centralized meta-data server, and work stealing, allowing several machines to work on a single partition.

We have demonstrated, through strong and weak scaling experiments, that Chaos scales on a cluster of 32 machines, and outperforms Giraph extended to out-of-core graphs by at least an order of magnitude. We have also quantified the dependence of Chaos' performance on various design decisions and environmental parameters.

Acknowledgments: We would like to thank our anonymous reviewers, shepherd Michael Swift, Rong Chen, Peter Peresini, Diego Didona and Kristina Spirovska for their feedback that improved this work. We would also like to thank Florin Dinu for his feedback, help in setting up the cluster and for motivating us to keep working on graph processing.

References

[1] http://www.graph500.org/results_jun_2014

[2] https://www.facebook.com/notes/facebook-engineering/scaling-apache-giraph-to-a-trillion-edges/10151617006153920

[3] http://zeromq.org/

[4] http://webdatacommons.org/hyperlinkgraph/

[5] http://freecode.com/projects/fio

[6] http://giraph.apache.org/

[7] Balakrishnan, M., Malkhi, D., Prabhakaran, V., Wobber, T., Wei, M., and Davis, J. D. CORFU: A shared log design for flash clusters. In *Proceedings of the conference on Networked Systems Design and Implementation* (2012), USENIX Association.

[8] Blumofe, R. D., and Leiserson, C. E. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM) 46*, 5 (1999), 720–748.

[9] Chakrabarti, D., Zhan, Y., and Faloutsos, C. R-MAT: A recursive model for graph mining. In *Proceedings of the SIAM International Conference on Data Mining* (2004), SIAM.

[10] Chen, R., Shi, J., Chen, Y., and Chen, H. PowerLyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the European Conference on Computer Systems* (2015), ACM, pp. 1:1–1:15.

[11] Elnozahy, E. N., Johnson, D. B., and Zwaenepoel, W. The performance of consistent checkpointing. In *Proceedings of the Symposium on Reliable Distributed Systems* (1992), IEEE, pp. 39–47.

[12] Garey, M. R., Johnson, D. S., and Stockmeyer, L. Some simplified NP-complete graph problems. *Theoretical computer science 1*, 3 (1976), 237–267.

[13] Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., and Guestrin, C. Powergraph: distributed graph-parallel computation on natural graphs. In *Proceedings of the Conference on Operating Systems Design and Implementation* (2012), USENIX Association, pp. 17–30.

[14] Gonzalez, J. E., Xin, R. S., Dave, A., Crankshaw, D., Franklin, M. J., and Stoica, I. GraphX: Graph processing in a distributed dataflow framework. In *Proceedings of the Conference on Operating Systems Design and Implementation* (2014), USENIX Association, pp. 599–613.

[15] Greenberg, A., Hamilton, J. R., Jain, N., Kandula, S., Kim, C., Lahiri, P., Maltz, D. A., Patel, P., and Sengupta, S. VL2: A scalable and flexible data center network. *SIGCOMM Comput. Commun. Rev. 39*, 4, 51–62.

[16] Han, W.-S., Lee, S., Park, K., Lee, J.-H., Kim, M.-S., Kim, J., and Yu, H. Turbograph: a fast parallel graph engine handling billion-scale graphs in a single PC. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining* (2013), ACM, pp. 77–85.

[17] Khayyat, Z., Awara, K., Alonazi, A., Jamjoom, H., Williams, D., and Kalnis, P. Mizan: A system for dynamic load balancing in large-scale graph processing. In *Proceedings of the European Conference on Computer Systems* (2013), ACM, pp. 169–182.

[18] Kyrola, A., and Blelloch, G. GraphChi: Large-scale graph computation on just a PC. In *Proceedings of the Conference on Operating Systems Design and Implementation* (2012), USENIX Association.

[19] Little, J. D. A proof for the queuing formula: $L = \lambda W$. *Operations Research 9*, 3 (May 1961), 383–387.

[20] Lumsdaine, A., Gregor, D., Hendrickson, B., and Berry, J. Challenges in parallel graph processing. *Parallel Processing Letters 17*, 1 (2007), 5–20.

[21] Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. Pregel: a system for large-scale graph processing. In *Proceedings of the International Conference on Management of Data* (2010), ACM, pp. 135–146.

[22] MALICEVIC, J., ROY, A., AND ZWAENEPOEL, W. Scale-up graph processing in the cloud: Challenges and solutions. In *Proceedings of the International Workshop on Cloud Data and Platforms* (2014), ACM, pp. 5:1–5:6.

[23] MATHUR, A., CAO, M., BHATTACHARYA, S., DILGER, A., TOMAS, A., AND VIVIER, L. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux Symposium* (2007), vol. 2, pp. 21–33.

[24] MITZENMACHER, M. The power of two choices in randomized load balancing. *Trans. Parallel Distrib. Syst. 12*, 10 (2001).

[25] NELSON, J., HOLT, B., MYERS, B., BRIGGS, P., CEZE, L., KAHAN, S., AND OSKIN, M. Latency-tolerant software distributed shared memory. In *Proceedings of the Usenix Annual Technical Conference* (2015), USENIX Association, pp. 291–305.

[26] NGUYEN, D., LENHARTH, A., AND PINGALI, K. A lightweight infrastructure for graph analytics. In *Proceedings of the Symposium on Operating Systems Principles* (2013), ACM, pp. 456–471.

[27] NIGHTINGALE, E. B., ELSON, J., FAN, J., HOFMANN, O., HOWELL, J., AND SUZUE, Y. Flat datacenter storage. In *Proceedings of the Conference on Operating Systems Design and Implementation* (2012), USENIX Association, pp. 1–15.

[28] NILAKANT, K., DALIBARD, V., ROY, A., AND YONEKI, E. PrefEdge: SSD prefetcher for large-scale graph traversal. In *Proceedings of the International Conference on Systems and Storage* (2014), ACM, pp. 4:1–4:12.

[29] NIRANJAN MYSORE, R., PAMBORIS, A., FARRINGTON, N., HUANG, N., MIRI, P., RADHAKRISHNAN, S., SUBRAMANYA, V., AND VAHDAT, A. PortLand: A scalable fault-tolerant layer 2 data center network fabric. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication* (2009), ACM, pp. 39–50.

[30] PEARCE, R., GOKHALE, M., AND AMATO, N. M. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In *Proceedings of the International conference for High Performance Computing, Networking, Storage and Analysis* (2010), IEEE Computer Society, pp. 1–11.

[31] ROY, A., MIHAILOVIC, I., AND ZWAENEPOEL, W. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the ACM symposium on Operating Systems Principles* (2013), ACM, pp. 472–488.

[32] WANG, K., XU, G., SU, Z., AND LIU, Y. D. Graphq: Graph query processing with abstraction refinement: Scalable and programmable analytics over very large graphs on a single PC. In *Proceedings of the Usenix Annual Technical Conference* (2015), USENIX Association, pp. 387–401.

[33] WU, M., YANG, F., XUE, J., XIAO, W., MIAO, Y., WEI, L., LIN, H., DAI, Y., AND ZHOU, L. GraM: Scaling graph computation to the trillions. In *Proceedings of the Symposium on Cloud Computing* (2015), ACM.

[34] ZHU, X., HAN, W., AND CHEN, W. GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *Proceedings of the Usenix Annual Technical Conference* (2015), USENIX Association, pp. 375–386.