CPSC 538B: Akka GPS Project

ALEX TROSTANOVSKY, JOSEPH POREMBA, and SEPAND DYANATKAR

In this report, we discuss our project developing Akka-GPS, a distributed graph processing system. Akka-GPS is designed to be a simple and modular system that can serve as a platform for experimentation and be approachable for newcomers to distributed graph processing. It is implemented using the Akka library for actor-based distributed computation. We document the high level design and implementation details of Akka-GPS. We provide an evaluation demonstrating its execution in a fully distributed environment, and discuss how its design facilitates its purpose as an accessible tool.

1 INTRODUCTION

Graph-structured data is everywhere: from social networks [13] to protein structures [20], supply chains, and COVID-19 spread modelling [3]. In recent years, the data scale of graphs has grown from the scale of billions of edges to the trillions [19]: the social network Facebook graph contains over one trillion edges [5]. Since such graphs cannot fit in the main memory of any single commodity server, researchers and developers invested significant resources to advance the state of distributed graph processing systems (GPS).

Making use of the continued improvements in processors, memory (including disaggregation), and distributed computing, researchers have combined theory and software engineering to devise systems such as Pregel. Pregel uses the Think-Like-a-Vertex (TLAV) model, designed around local computation on vertices in a "superstep" and message passing across vertices between these steps [16]. Pregel proposed an abstract, scalable, and fault-tolerant implementation, but did not make its source code publicly available. Projects like Apache Giraph [2] provide this implementation as open source software (OSS). The TLAV model inspired numerous other successful models and systems, both distributed and single machine, such as PowerGraph, PowerLyra, and Apache Spark's GraphX [4, 9, 10]. In this project, we introduce *Akka-GPS*, a simple distributed GPS based on the TLAV paradigm, implemented using the actor model for distributed computation.

1.1 Motivation and Goals

Building a scalable implementation of a distributed GPS is often done by specialized software teams. The models and implementations discussed above are not approachable for the average developer or new researchers in this field. We found ourselves in this situation, as we wanted to experiment with existing systems, but found them difficult to dive into. A similar problem plagued Boolean satisfiability (SAT) solving systems: the complexity and lack of approachability in implementations of SAT-solvers became an impediment for newcomers attempting to learn about, modify, apply these methods. To address this, researchers developed an implementation called MiniSAT [6], whose primary goal was simplicity to facilitate its modification and use as a starting point for new SAT-solvers. Akka-GPS is our attempt to bridge a similar gap between complex, industrial GPSs and an average researcher or developer.

We also recognized a natural correspondence between the TLAV graph processing model and the general actor model for distributed computation, such as is provided by the open-source Akka [17] library for Scala. We decided Akka would be a solid foundation on which to build a simple but robust GPS, which we believe to be a novel choice.

Akka-GPS aims to achieve the following goals:

- Implement a graph processing system that can be deployed and run in a distributed environment with multiple machines, using an actor system.
- Provide users with a simple API with which to "plug-in" their own graph algorithm implementations.
- Act as an accessible and robust reference tool for experimentation, allowing modification to its partition system and execution engine, in order to serve as a template for more advanced and custom distributed GPS implementations (for example, as MiniSat provied for SAT-solvers).

Authors' address: Alex Trostanovsky, atrostan@cs.ubc.ca; Joseph Poremba, jporemba@cs.ubc.ca; Sepand Dyanatkar, sepandd@student.ubc.ca.

We do not aim to mimic or rebuild a complex GPS like Pregel or PowerGraph, nor do we expect to outperform them. Performance is not our primary concern, but ideally Akka-GPS should scale in terms of execution time and node memory usage as the number of machines increases. As part of a course in distributed systems, developing our solution additionally taught us about building distributed, graph processing, and actor-based systems. This summary of our work serves as insight on the decisions made in building the system, their consequences, as well as the existing literature and resources which guided these decisions.

2 BACKGROUND

2.1 Think-Like-a-Vertex Graph Processing

Think-Like-a-Vertex (TLAV) is a paradigm for expressing distributed graph algorithms. In this paradigm, vertices[1] maintain some local state. A *vertex program* is executed at the vertices, during which the vertex sends messages to and receives messages from neighbouring vertices and update its local state.

A system may be *synchronous* or *asynchronous* depending on how it chooses to schedule messages and the order of vertex program execution. In this project, we focus on synchronous graph processing systems. Computation proceeds in rounds called *supersteps*. In a superstep, the vertex program is run at each[2] vertex, conceptually in parallel. The vertex program can access messages sent to the vertex in the previous superstep, update local state, and send messages to neighbours of the vertex (which will be read in the next superstep).

As an example, consider graph colouring. The objective is to assign colours to vertices so that neighbours have different colours. One colouring algorithm in a synchronous GPS is local maxima colouring, pictured in Figure 1. Vertices begin uncoloured. At superstep n, a fresh colour c_n is selected. Each vertex receives the IDs of its uncoloured neighbours. If a vertex is uncoloured and has a larger ID than every uncoloured neighbour, it colours itself with the colour c_n. Any vertices still uncoloured send their IDs to their neighbours for superstep $n + 1$.

Messages may be sent along out-edges (as in PageRank), in-edges, or both (as in local maxima colouring), but for ease of exposition we say messages are sent to out-neighbours and received from in-neighbours.

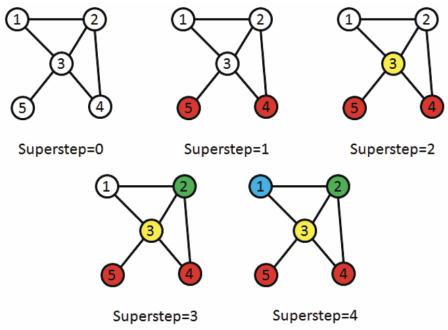

Fig. 1. An example of local maxima colouring. From [7].

2.2 Actor Model for Distributed Computation

The actor model is a model for parallel and distributed computation, implemented by frameworks such as Akka [17]. Actors are units which perform computation and communicate by passing messages [14]. They can do 3 things: create other actors, send messages to actors, define behaviour on receipt of the next message. There is no shared memory between actors; communication is handled exclusively through message passing. In an actor, messages are processed one at a time through what is called a mailbox queue. Important features of actors are strict decoupling of execution and signaling, independent state, and queues to ensure that they are non-blocking [11, 14]. As a result, actors make reasoning about parallel, distributed and/or asynchronous computation easy.

In the TLAV model, each vertex contains state and sends messages along its edges to other vertices. As such, there is a natural correspondence between actors and vertices in a GPS.

3 DESIGN

In this section, we discuss the high-level design of Akka-GPS.

[1] In this report, we never refer to vertices of a graph as *nodes*, which we reserve for computational nodes in the distributed system.
[2] technically only at *active* vertices, but this will be discussed more in Section 3.3

3.1 Partitioning

Partitioning refers to how the graph is split across computational nodes. The two main choices are *vertex partitioning*, also known as *edge-cut*, and *edge partitioning*, also known as *vertex-cut* (See Fig. 2). In vertex partitioning, the vertex set is completely partitioned amongst the computational nodes, and each vertex tracks all of its incident edges. In edge partitioning, the edge set is completely partitioned amongst the computational nodes. A copy of each vertex exists on every node for which it has an incident edge, but that copy is only aware of the edges on that node.

Many real-world graphs of social networks and the web are power-law graphs [4]. In these graphs, many of the vertices have a relatively small degree, but a small number of vertices (called hub or hot vertices) account for a large proportion of the total number of edges. Using vertex partitioning to partition power-law graphs may cause partition imbalance due to the exclusive assignment of hub vertices to partitions [4]. Edge partitioning is preferable in these scenarios because the large number of incident edges to the hub vertices can be evenly distributed amongst partitions.

Akka-GPS uses edge partitioning. This decision trades simplicity for efficiency and scalability. It increases the complexity of the system, since there must be coordination between the copies of a vertex to aggregate their updates. While we are generally aiming for simplicity in the final system, we believe this trade off is necessary since edge partitioning is the more practical solution for scaling a system to handle large real-world input graphs.

Akka-GPS provides the following partitioning algorithms:

- *1D Partitioning*: Each edge is assigned to a one-dimensional partitioning space by hashing either the source or destination vertex of that edge.
- *2D Partitioning*: Each edge is assigned to two-dimensional partitioning space by hashing the incident vertices separately.
- *Balanced Hybrid Cut Partitioning* [4]: differentiates the partitioning strategy for low-degree and high-degree vertices. A high-degree vertex is a vertex whose degree is larger than the average vertex degree in the graph.
  - For low-degree vertices, evenly assign vertices along with their out-edges to partitions by hashing the source of the out-edge.
  - For high-degree vertices, distribute all out-edges by hashing their destination.

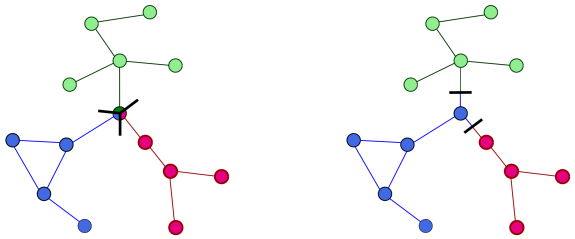

Fig. 2. (a) Edge Partition (Vertex-Cut Partition) vs (b) Vertex Partition (Edge-Cut Partition). From [12].

3.2 Gather, Apply, Scatter (GAS) Model

As a consequence of edge partitioning, there may be multiple copies of a given vertex on different partitions, each tracking only a subset of incident edges. These copies need to coordinate to produce correct state updates. To solve this problem, Akka-GPS adopts the Gather, Apply, Scatter (GAS) architecture of PowerGraph [9].

In the GAS architecture, if a vertex has copies on multiple partitions, one copy is declared the *main* copy, and the others *mirror* copies. The vertex program executes in three phases. In the *Gather* phase, each vertex copy reads messages sent from its in-neighbours during the previous superstep and turns it into an *accumulator* value. The copy combines these accumulator values using a commutative and associative *sum* function, and if it is a mirror, sends its final sum to the main copy. For example, in local maxima colouring, the accumulator values are integers representing neighbour IDs, and the sum function takes the maximum of two IDs (since the vertex only needs to determine if its ID is larger than all neighbour IDs it receives). The main copy sums its local accumulator sum with the sums it receives from its mirrors, computing a final main accumulator value. In the *Apply* phase, the main copies use the final main accumulator value to update the vertex state. In the *Scatter* phase, the resulting vertex state is propagated back to the mirrors, and each copy sends messages to their out-neighbours for the next superstep. Vertices may choose to not send messages during the Scatter phase.

Conceptually, a vertex program author does not need to know about mains and mirrors, they just need to know that vertex programs execute in these phases, and there is a sum function which combines messages destined to the same vertex. Breaking supersteps into phases means that vertex programs must follow a more rigid structure: rather than a monolithic function, a vertex program must be explicitly decomposed into the three phases. This becomes apparent in our vertex program API (see Section 3.4). All TLAV algorithms we have encountered naturally decomposition into these phases, so only the sum function is an extra burden on algorithm writers. Additionally, even non-GAS systems such as Pregel [16] optionally allow a sum function (which Pregel calls "message combiners") to reduce message traffic. In Pregel, combiners are optional, but in GAS systems, sum functions are mandatory.

3.3 Termination Semantics

To determine when to terminate computation, most GPS systems give vertices binary *activation states*. *Active* vertices participate in computation, and *inactive* vertices do not. The system terminates when all vertices are inactive. Different systems vary on the mechanics of activation state transitions. In PowerGraph, vertices always de-activate after executing the Apply phase, and may choose to explicitly re-activate themselves or any subset of neighbours during the Scatter phase. In Pregel, vertices may explicitly de-activate themselves after the current superstep, and are implicitly re-activated upon receiving a message. Though our GAS architecture is inspired by PowerGraph, we saw no need for the extra granularity in its activation semantics, so we opted for the simpler Pregel semantics.

3.4 Specifying Algorithms with Vertex Program API

Akka-GPS provides an API for a user to specify TLAV algorithms using the GAS model. Specifically, a user extends the `VertexProgram` trait (the Scala equivalent of a Java Interface) with an object for their algorithm, and implements the functions required by GAS and our termination semantics:

- `gather`: map messages into accumulator values,
- `sum`: commutative and associative sum function for combining accumulator values,
- `apply`: update vertex state, given the final accumulator value,
- `scatter`: what messages to send (if any), given the new vertex state
- `deactiveSelf`: whether a vertex should deactivate itself after this superstep, given the new vertex state
- `defaultVertexValue`: initial vertex state
- `defaultActivationStatus`: initial activation state
- `mode`: whether messages are sent along out-edges, in-edges, or both

We implemented four algorithms using our API to ensure it is sufficiently expressive: local maxima colouring, single-source shortest paths, weakly connected components, and PageRank. Our implementation of local maxima colouring is shown in Figure 3.

4 IMPLEMENTATION

4.1 Akka with Cluster Sharding

Akka is a modern, open source, and flexible library for building actor based systems [17]. It is community developed and supported by the company Lightbend, which has various affiliations with the Scala and Java programming languages [17]. We chose to use Akka in particular due to its message-passing based, scalable and robust system which exposes an expressive and customizable programming interface. These are similar to our system's objectives, making it a natural fit.

Cluster sharding is a module of the Akka library for managing distributed applications [17]. It has a very generalized and flexible configuration, which controls a hierarchical structure of elements primarily consisting of nodes, shard regions, shards, and entities as seen in figure 4. Entities are wrapped, shard-managed actors. This frees actors from any single actor system and thus allowing shard-coordinated state control for managing redistribution, shutdown and restart of actors. We coordinate entities

```scala
object LocalMaximaColouring extends VertexProgram[Int, Int, Int, Colour] {

  override def gather(edgeVal: Int, message: Int): Int = message

  override def sum(a: Int, b: Int): Int = Math.max(a, b)

  override def apply(superStepNumber: Int, thisVertex: VertexInfo, oldVal: Colour, total: Option[Int]): Colour =
    if (superStepNumber == 0) Colour.Blank
    else if (oldVal != Colour.Blank) oldVal
    else if (total == None || total.get < thisVertex.id) Colour(superStepNumber - 1)
    else Colour.Blank

  override def scatter(superStepNumber: Int, thisVertex: VertexInfo, oldVal: Colour, newVal: Colour): Option[Int] =
    if (newVal == Colour.Blank) Some(thisVertex.id)
    else None

  override def deactivateSelf(superStepNumber: Int, oldVal: Colour, newVal: Colour): Boolean = (newVal != Colour.Blank)

  override val defaultVertexValue: Colour = Colour.Blank

  override val defaultActivationStatus: Boolean = true

  override val mode = VertexProgram.Bidirectional
}
```

Fig. 3. Local Maxima Colouring, implemented with our `VertexProgram` API.

through `EntityManagers` and manage entity addressing by overriding. Our system currently configures cluster sharding to have a single node and shard region per machine, disables rebalancing, and separates the initialization (including partitioning) and execution to simplify usage.

This module also has numerous fault-tolerance mechanisms and scalability capabilities, such as fine-tuned reallocation of entities (on changes or failures of any element in the system). Actor systems including Akka typically provide at-most-once delivery, meaning that messages can be lost. However, Akka additionally provides ordering between sender-receiver pairs and also claims that message loss in normal operation is rare [17]. Since our requirements do not include heightened message guarantees, we implement our TLAV messaging patterns above actor messaging as-is. In the future, we can improve upon Akka's message guarantees or improve the configuration controls of cluster fault tolerance and scalability mechanisms, in response to particular evaluation results and needs.

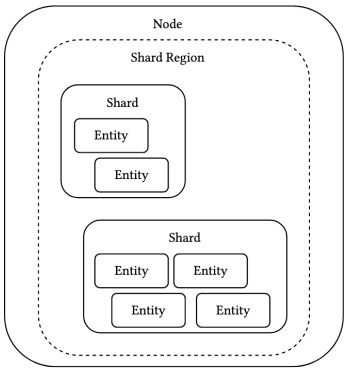

Fig. 4. Basic hierarchy of cluster sharding components. A cluster configuration can have any number of nodes, shard regions within each node, shards, and entities as needed.

4.2 Actor Messaging Patterns

To implement Akka-GPS using actors, we had to determine: what actors exist in the system, what types of messages the actors send to each other (and their contents), and what actors do upon receiving messages.

In Akka-GPS, each main and mirror copy of a vertex is represented by an actor, in addition to the global and partition coordinators. Most of the GAS architecture translates naturally into message types. For example, we defined a `NeighbourMessage` message type that corresponds to the TLAV neighbour messages, a `MirrorTotal` message type for mirrors to send their sums to their main copies, and an `ApplyResult` message type for mains to send their updated state to mirrors. There are also message types for auxiliary operations such as initialization.

As an example for message receipt behaviour, consider the `NeighbourMessage` message type. Upon receiving a `NeighbourMessage`, a vertex copy uses the `gather` and `sum` functions to convert the message contents into an accumulator value and combine it with the copy's current total. If the copy is a mirror, it checks to see if it has received all expected messages from neighbours, and if so then sends a `MirrorTotal` message with its accumulated sum to its main copy. If the copy is a main, it checks to see if it has received all expected `NeighbourMessage` messages and `MirrorTotal` messages, and if so then proceeds to the Apply phase.

In this implementation, vertex copies check that all expected messages from in-neighbours have arrived before continuing computation. Since vertex copies do not know which in-neighbours will send a message during Scatter, vertices that send nothing during Scatter must inform their out-neighbours of this intention with null `NeighbourMessages`. We discuss an alternative implementation in the next subsection, and why we rejected it.

While it was generally straightforward to map GAS to actor messages and behaviours, small subtleties were uncovered. For example, mirrors with zero in-degree receive no `NeighbourMessages`, but as a result their check to see if all expected messages have arrived never triggers, causing a difficult-to-debug non-termination issue. Another issue is that it is possible for a vertex u to be on the Scatter phase of superstep n while it has an out-neighbour v still in the Gather phase of superstep n. When u sends a `NeighbourMessage` to v, it is intended to be read in superstep $n + 1$, so v should not add this message to its accumulator total for superstep n. To solve this, actors associate sums and counters with superstep numbers, and most messages are tagged with superstep numbers.

4.3 Superstep Synchronization

Akka-GPS has a blocking mechanism to ensure that ensures that vertices may not run the Apply phase of superstep $n + 1$ until all vertices have finished the Apply phase of superstep n (recall however that vertices may be processing incoming neighbour messages for superstep $n + 1$ while other vertices, or even itself, have not finished the Apply phase for superstep n). Without this blocking, we only have the following guarantee: if v is an out-neighbour of u, then u must be finished the Apply phase of superstep n before v can begin the Apply phase of superstep $n + 1$ (because otherwise v would be missing a neighbour message from u). But vertices in different components could become out-of-sync.

To implement blocking, the following mechanism is used. When main either executes or skips its Apply phase (depending on activation status), it sends a `DONE` vote or `TERMINATE` vote respectively to its partition coordinator. The aggregated partition votes are forwarded to the global coordinator, which decides either to terminate computation or begin the next superstep. In the latter case, a `BEGIN` message is sent to each partition coordinator to disseminate to the mains. Mains do not perform the next Apply phase until receiving this message. This blocking mechanism also provides a convenient way for checking the termination condition and triggering computation in vertices of zero in-degree. We see our current approach as simple but somewhat heavy-handed, and believe that it is a prime candidate for future experimentation and improvement.

We considered an alternative approach where vertices do not vote until they confirm that their outgoing Scatter messages have arrived. This would also eliminate the need to have null `NeighbourMessages`, since once all votes have been cast, all neighbour messages must have been processed. We ultimately rejected this solution because it requires waiting for acknowledgment messages between many actors.

5 EVALUATION

5.1 Deployment and Scalability

Akka-GPS has been successfully deployed on AWS clusters, and is able to correctly process moderately large graphs. We evaluated the scalability of Akka-GPS by measuring the following metrics as a function of the number of nodes in an Akka cluster:

- **Replication Factor**. $\text{RF} := \frac{\text{Number of main vertices + Number of mirror vertices}}{\text{Number of vertices}}$
- **Execution Time**. The time to complete 10 iterations of PageRank or SSSP starting from source vertex 0.

We used the following graphs:

- `web-Google` [15]: A directed web graph. Nodes represent web pages and edges represent hyperlinks between web pages. Contains 875,713 nodes and 5,105,039 edges.
- `web-NotreDame` [1]: A directed web graph. Nodes represent web pages from the University of Notre Dame and edges represent hyperlinks between those web pages. Contains 325,729 nodes and 1,497,134 edges.

and the following configurations:

- An EC2 cluster with 2, 4, 8, 12 `m4.large` nodes. Each node in the cluster had 2 vCPUs and 8 GiB of memory.
- An EC2 cluster with 2, 4 `m4.xlarge` nodes. Each node in the cluster had 4 vCPUs and 16 GiB of memory.

For each of the cluster configurations, we partition the graph using both 1D and Hybrid Partitioning. We did not evaluate the performance of 2D Partitioning since some of the cluster configurations contain numbers of partitions that were not perfect squares (e.g. 2, 8, 12). In addition, we've instrumented each node in the cluster to report the following metrics:

- `mem-used-percent`: The percentage of memory used.
- `net-bytes-recv`: The number of bytes received by by a node on all of its network interfaces.
- `net-bytes-sent`: The number of bytes sent by a node on all of its network interfaces.
- `cpu-usage-system`: The percentage of time that the CPU was in system mode.
- `cpu-usage-user`: The percentage of time that the CPU was in user mode.

During execution, we recorded the average per second for each of the metrics.

Fig. 5 shows that as we increase the number of nodes in the cluster, the Replication Factor correspondingly increases. This increase is due to the fact that as we add more nodes to the cluster, more vertices will have to be mirrored across the additional nodes, which in turn increases the replication factor. We note a decrease in execution time for both PageRank and SSSP from 2 to 4 nodes, but, unexpectedly, an increase in execution time for 8 and 12 nodes in the cluster. We hypothesize why this may be below.

Fig. 5 shows a slightly improved replication factor between 1D and Hybrid partitioning. We believe these improvements directly correspond to the improved execution time for both SSSP and PR. However, Fig. 6 shows no substantial difference in the replication factor 1D and Hybrid Partitioning. This is due to the small number of Hub Vertices (vertices whose outdegree is greater than 100) in `web-Google`. This graph contains 171 hub vertices (0.07% of the total number of vertices). As a result, the 1D and hybrid partitionings of `web-Google` are essentially equivalent. Akka-GPS allows the user to input the degree threshold that defines a hub vertex. Modifying this hyperparameter may produce better Replication Factors for hybrid partitionings of different graphs.

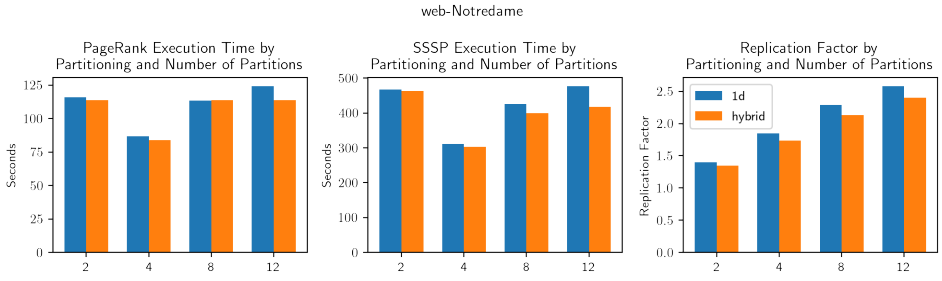

Fig. 5. *Left*: Runtime of 10 iterations of PageRank on `web-Notredame` using 2, 4, 8, 12 nodes. *Centre:* Runtime of SSSP from source vertex 0. *Right:* Replication factor of `web-Notredame` using 2, 4, 8, 12 nodes.

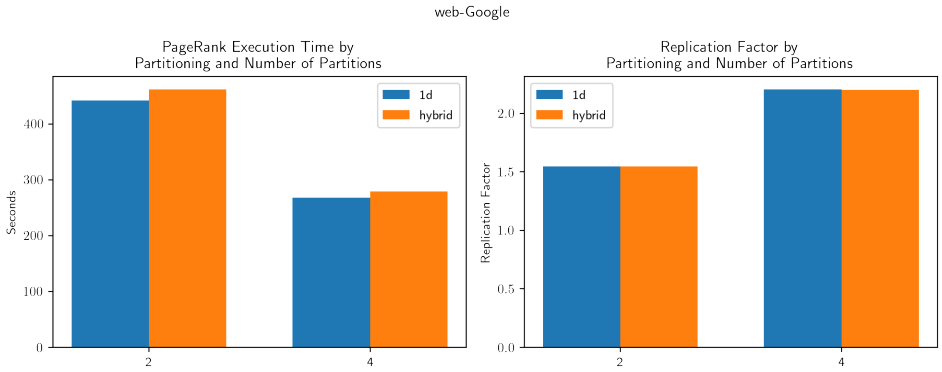

Fig. 6. *Left*: Runtime of 10 iterations of PageRank on `web-Google` using 2, 4 nodes. *Right:* Replication factor of `web-Google` using 2, 4 nodes.

Network usage (omitted due to space constraints) illustrates a useful feature of Akka cluster sharding. Specifically, the routing of messages using `ShardRegion` actors. In the first Superstep, all partition coordinators message all their assigned main vertices to begin computation. This is seen by a peak in the bytes sent and received over the network for Superstep 1. However, once all main and mirror addresses have been registered with the Partition Coordinators, each subsequent Superstep incurs a reduced and consistent network overhead. CPU usage data (omitted due to space constraints) shows another unexpected behaviour of Akka-GPS. In 12 partitions, 0-3 exhibit notable usage of their CPUs, while 4-11 all are underutilizing their CPUs. We hypothesize that this behaviour may have played a part in producing the unexpected scalability results seen in Fig. 5. Given the time constraints of our project, we were unable to substantiate these claims, but will continue debug and instrument the performance of Akka-GPS at scale.

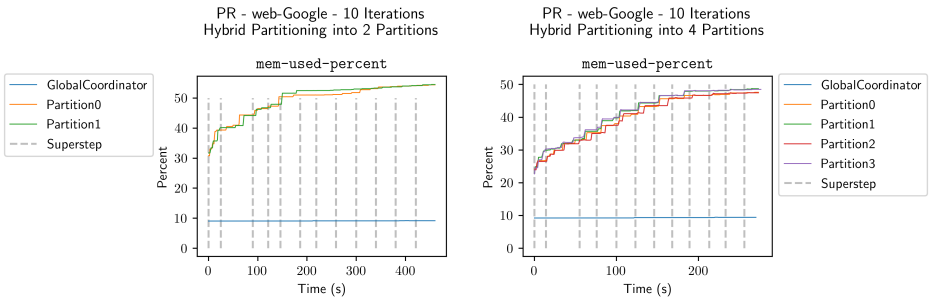

Fig. 7. The percentage of memory used by each node in the cluster during PageRank on the hybrid-partitioned `web-Google` graph with 2, 4 partitions.

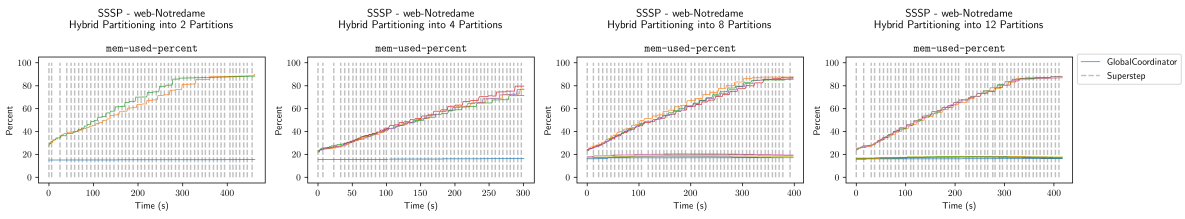

Fig. 8. The percentage of memory used by each node in the cluster during SSSP on the hybrid-partitioned `web-Notredame` graph with 2, 4, 8, 12 partitions.

Fig. 7 showcases an advantage of using Akka-GPS and, in general, distributed graph processing. As we increase the number of nodes in the cluster, the baseline memory percentage required to start a vertex program decreases from ~ 30% to ~ 25%. Note that this reduction is not perfectly linear in the number of partitions, since additional partitions will tend to introduce larger number of vertex copies. At the same time, we note that as execution progresses, the memory used by the nodes in the cluster monotonically increases, with no apparent Garbage Collection. In the future, we hope to modify Akka-GPS' garbage collection as a possible optimization. Finally, it should be noted that Fig. 8 exhibits a similar pattern of memory utilization for 8, 12 node cluster as we see in our CPU utilization.

5.2 Simplicity

Simplicity was another major goal of Akka-GPS, as we wanted it to serve as a starting point for newcomers to graph processing systems, including developers and budding researchers. It is difficult to evaluate this thoroughly and objectively, since we did not have time for a user study. However, there are indicators that reflect positively on the simplicity of Akka-GPS.

Table 1 shows the lines of code in which we implemented various algorithms using our vertex program API. We did not make these intentionally terse; we believe our implementations to be readable and straightforward (see Figure 3 for our full implementation of local maxima colouring as an example) and yet the algorithms were able to implemented with very little code. We believe this is evidence of a strong API that is able to facilitate simple algorithm implementations.

Algorithm	Lines of Code
Local Maxima Colouring	15
PageRank	17
Single-Source Shortest Paths	15
Weakly Connected Components	15

Table 1. Algorithm implementation lengths using API. Lines of code excludes whitespace, braces, and comments. Class and function signatures take up 9 lines (included in the count).

Additionally, the code for our execution engine which implements the API is relatively small. The core part of the engine, consisting of the actors for the global and partition coordinators as well as actors for vertices, consists of less than 750 lines of code altogether (excluding whitespace, comments, and braces), with room to optimize and make significantly more compact (admittedly however, there is quite a bit more code for setting up the Akka cluster). Moreover, we are pleased with the abstraction within our system. The vertex program API does not expose details of the execution engine or partitioning system (beyond the use of GAS), there is little dependency between the execution engine and partitioning system. The execution engine makes no assumptions about the type of edge partitioning used, and in theory an external system could provide the partitioning. We believe the short length of the execution engine and the clear separation of internal components facilitate maintainability and easy modification of Akka-GPS, thus enabling it to serve its function as an approachable system for newcomers to graph processing who are looking to experiment.

6 RELATED WORK

Pregel introduced the stateful, TLAV vertex-based programming model, which is the basis for our system and many other large scale successful GPS' [16]. It also guided our decision to represent our vertices as actors, since actors' properties closely aligned with those of vertices while drastically simplifying reasoning about distributed graph programs. To the best of our knowledge, there are no other systems with this actor-based approach and we believe this produces an approachable and modular GPS.

PowerGraph [9] is another distributed graph processing system. PowerGraph uses edge partitioning, giving it an advantage over Pregel in partitioning power-law distributed graphs. It also introduced the Gather, Apply, Scatter model for TLAV graph computation. PowerGraph served as the main inspiration for the high-level design of Akka-GPS, though we used the activation and termination model of Pregel instead.

PowerLyra [4] builds off of Pregel, PowerGraph and other GPS contributions by adapting to vertices of varying degree in computation and partitioning [4]. This is the source of our hybrid partitioning strateg and argues for adaptable, locality-conscious GPS', which we also identify as extensions of our work we hope to make accessible with our modular, customizable system.

Spark provides a dataflow model for distributed computation, different from the original graph-specific model proposed by Pregel [21]. It can be used for graph computation but is adjacent to the TLAV system we want to provide for users and adds significant underlying complexity if the GAS model was implemented overtop.

MiniSat [6], short for Minimal SAT Solver, is a solver for the boolean satisfiability (SAT) problem. The creators of MiniSat observed that there was a large gap between the high-level description of SAT solvers and their actual implementation, and that creating new SAT solvers or even modifying existing systems had become difficult and obtuse. MiniSat was designed to prioritize accessibility and readability, with the stated goal being: "we hope it is possible for *you* to implement a fresh SAT-solver in your favorite language, or to grab the C++ version of MiniSat from the net and start modifying it to include new and interesting ideas". We were inspired to make Akka-GPS following this philosophy, but for distributed graph processing systems.

7 FUTURE WORK

We recognize there is room for optimization in the execution engine of Akka-GPS. As mentioned previously, the synchronous blocking mechanism is somewhat heavy. We believe the blocking barrier may be able to be relaxed with tweaks to Akka's scheduling, becoming closer to an asynchronous system without giving up the logical behaviour of supersteps. Even if different parts of the graph are somewhat out-of-sync, tagging messages with supersteps may enable us to behave equivalently to a synchronous system. We also want to investigate implementing a separate, truly asynchronous execution engine. Additionally, PowerGraph describes a method for caching and triggering vertex computations only as needed, called *delta caching*, but this requires centralized coordination of caches and only works on certain algorithms.

There are additions including fault tolerance, scalability, and dynamic repartitioning possible on the underlying Akka implementation. These can be added by modifying configurations to use Akka's built-in implementations or by modifying the whole cluster sharding component of Akka-GPS. Traditional snapshot fault tolerance can also be added without modifying any of the underlying system, similar to Pregel and PowerGraph [9, 16]. Extensions such as these are highly encouraged, as our objectives include enabling extensions and modifications at the discretion of the Akka-GPS user. Measuring extensibility and usability is something we hope to be done in the future as well.

8 CONCLUSION

In this project, we built and documented the design behind Akka-GPS, a distributed graph processing system. Akka-GPS partitions the edges of a graph across different computational nodes, and uses the Think-Like-a-Vertex paradigm for distributed graph algorithms. It exposes a simple API to allow users to implement their own algorithms in the GAS model of vertex programs. Akka-GPS implements the GAS model using an execution engine based on a system of actors, using the open-source Akka library. Akka-GPS is able to execute correctly in a distributed setting with multiple computational nodes on moderately large graphs. Many of the optimizations and features of a modern GPS come with tradeoffs in simplicity, approachability, and modularity. We succeeded in producing a functioning distributed GPS, using an approachable actor-based system, modular components (e.g. partitioning, sharding control, or execution), and an expressive API. While its execution engine requires some optimization to become more scalable, we believe that through its simple and approachable design, Akka-GPS will be able to fulfill its goal as an accessible tool to serve as a base for reference and experimentation in graph processing systems.

REFERENCES

[1] Réka Albert, Hawoong Jeong, and Albert-László Barabási. 1999. Diameter of the world-wide web. *nature* 401, 6749 (1999), 130–131.

[2] Apache Software Foundation. Accessed: 2021-10-05. Giraph. https://giraph.apache.org

[3] Serina Chang, Emma Pierson, Pang Wei Koh, Jaline Gerardin, Beth Redbird, David Grusky, and Jure Leskovec. 2021. Mobility network models of COVID-19 explain inequities and inform reopening. *Nature* 589, 7840 (2021), 82–87. https://doi.org/10.1038/s41586-020-2923-3

[4] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. 2015. PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. In *Proceedings of the Tenth European Conference on Computer Systems* (Bordeaux, France) *(EuroSys '15)*. Association for Computing Machinery, New York, NY, USA, Article 1, 15 pages. https://doi.org/10.1145/2741948.2741970

[5] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. 2015. One Trillion Edges: Graph Processing at Facebook-Scale. *Proc. VLDB Endow.* 8, 12 (aug 2015), 1804–1815. https://doi.org/10.14778/2824032.2824077

[6] Niklas Eén and Niklas Sörensson. 2004. An Extensible SAT-solver. In *Giunchiglia E., Tacchella A. (eds) Theory and Applications of Satisfiability Testing. Lecture Notes in Computer Science (SAT 2003, Vol. 2919)*. Springer, Berlin, Heidelberg, 502–518. https://doi.org/10.1007/978-3-540-24605-3_37

[7] Nishant M Gandhi and Rajiv Misra. 2015. Performance comparison of parallel graph coloring algorithms on bsp model using hadoop. In *2015 International Conference on Computing, Networking and Communications (ICNC)*. IEEE, 110–116.

[8] Gurbinder Gill, Roshan Dathathri, Loc Hoang, and Keshav Pingali. 2018. A Study of Partitioning Policies for Graph Analytics on Large-Scale Distributed Platforms. *Proc. VLDB Endow.* 12, 4 (dec 2018), 321–334. https://doi.org/10.14778/3297753.3297754

[9] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, USA) *(OSDI'12)*. USENIX Association, USA, 17–30. https://doi.org/10.5555/2387880.2387883

[10] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 599–613. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/gonzalez

[11] Irene Grief. 1975. *Semantics of Communicating Parallel Processes.* Ph. D. Dissertation. Massachusetts Institute of Technology, Cambridge, MA.

[12] Masatoshi Hanai, Toyotaro Suzumura, Wen Jun Tan, Elvis Liu, Georgios Theodoropoulos, and Wentong Cai. 2019. Distributed Edge Partitioning for Trillion-Edge Graphs. *Proc. VLDB Endow.* 12, 13 (sep 2019), 2379–2392. https://doi.org/10.14778/3358701.3358706

[13] Safiollah Heidari, Yogesh Simmhan, Rodrigo N. Calheiros, and Rajkumar Buyya. 2018. Scalable Graph Processing Frameworks: A Taxonomy and Open Challenges. *ACM Comput. Surv.* 51, 3, Article 60 (jun 2018), 53 pages. https://doi.org/10.1145/3199523

[14] Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence* (Stanford, USA) *(IJCAI'73)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 235–245.

[15] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. 2009. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics* 6, 1 (2009), 29–123.

[16] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-Scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (Indianapolis, Indiana, USA) *(SIGMOD '10)*. Association for Computing Machinery, New York, NY, USA, 135–146. https://doi.org/10.1145/1807167.1807184

[17] The Lightbend Project. Accessed: 2021-12-15. Akka. https://akka.io/

[18] Shiv Verma, Luke M. Leslie, Yosub Shin, and Indranil Gupta. 2017. An Experimental Comparison of Partitioning Strategies in Distributed Graph Processing. *Proc. VLDB Endow.* 10, 5 (jan 2017), 493–504. https://doi.org/10.14778/3055540.3055543

[19] Ming Wu, Fan Yang, Jilong Xue, Wencong Xiao, Youshan Miao, Lan Wei, Haoxiang Lin, Yafei Dai, and Lidong Zhou. 2015. GraM: Scaling Graph Computation to the Trillions. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (Kohala Coast, Hawaii) *(SoCC '15)*. Association for Computing Machinery, New York, NY, USA, 408–421. https://doi.org/10.1145/2806777.2806849

[20] Yan Yan, Shenggui Zhang, and Fang-Xiang Wu. 2011. Applications of graph theory in protein structure identification. *Proteome science* 9 Suppl 1 (10 2011), S17. https://doi.org/10.1186/1477-5956-9-S1-S17

[21] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing* (Boston, MA) *(HotCloud'10)*. USENIX Association, USA, 10.