DFF: A Distributed Forest Fire driven Graph Scalar

Graph Scalar Group 1

Tom Ebergen
Vrije Universiteit Amsterdam
Amsterdam, Netherlands
t.g.ebergen@student.vu.nl

Dennis Wind
Vrije Universiteit Amsterdam
Amsterdam, Netherlands
d.g.wind@student.vu.nl

Okke van Eck
Vrije Universiteit Amsterdam
Amsterdam, Netherlands
o.l.van.eck@student.vu.nl

Yancheng Zhuang
Vrije Universiteit Amsterdam
Amsterdam, Netherlands
y.zhuang@student.vu.nl

ABSTRACT

Graph sampling based on the Forest Fire model is an effective and intuitive way of handling a massive amount of graph data. In this report, we present DFF: a distributed graph scalar based on forest fire sampling (FFS). Two kinds of distributed forest fire sampling algorithms are implemented: *halted FFS* and *wild FFS*. The major difference is that wild FFS allows a fire to propogate from one compute node to another, which is not possible for halted FFS. DFF supports downscaling and upscaling with different stitching topologies and connectivity rates. Several experiments are conducted on the system using the DAS-5 supercomputer and graphs from the Graphalytics datasets. Results show that both halted and wild FFS are scalable for all scenarios, but do not perform well on the preservation of graph properties.

1 INTRODUCTION

With the wide spread of the internet, understanding networks and their representation as graphs is an important research topic. The amount of connected devices is increasing, which arises two important questions. Firstly, what happens when the size of a network scales above what a single machine can hold in memory? Secondly, can a network sustain the increase in devices? Both of these questions can be handled by graph scaling, whether it is scaling down a graph to one that fits in memory, or scaling a graph up to see if certain properties are still present.

Multiple single device graph sampling algorithms exist [6]. Simple graph sampling can be used as a method for downscaling. Upscaling is most commonly handled by taking multiple samples of the graphs and stitching those together to end up with a graph larger than the original.

Random Walk (RW) is a graph sampling algorithm, where a graph is created by a walker [6]. The walker starts at a random initial vertex and travels to a neighboring vertex during each iteration of the algorithm. All traveled vertices form the resulting sampled graph. One advantage of RW is that it is stateless, which means that the algorithm does not need to remember what vertices were previously visited, as vertex are allowed to be repeated.

When using Forest Fire Sampling (FFS), an initial vertex is set on fire [6]. For each neighbor of the burning vertex, a dice is rolled to determine whether or not the neighbor will start burning as well. This is done iteratively using the newly lit vertices in order to form the resulting sample, which will consists of the burned vertices and edges. FFS was originally designed as a graph generation model that focuses on capturing observations of real social networks. This includes observations as densification laws, shrinking diameters, and community guided attachments. Nowadays, it is also adapted as a graph sampling algorithm focusing on those properties.

However, these algorithms only work when the original graph and resulting graph can be held in memory on a single machine. We present a distributed graph scaler based on Forest Fire sampling for when a graph is too large. This system is revered to as Distributed Forest Fire, or DFF for short.

In the upcoming sections, we will first specify the functional and non-functional requirements of DFF. Then, we outline the system's design and evaluate the performance and scalability of its features. We conclude with a discussion on the usefulness and feasibility of distributed graph scalers.

2 BACKGROUND

A Distributed Graph Scaler (DGS) allows researchers with access to a distributed supercomputer to utilise the full resources of the system to create a downscaled or upscaled sample of a graph when the memory of a single machine is insufficient. This can be achieved by partitioning the input graph and storing the partitions on seperate machines. The functional requirements of DFS can be covered in the following four points.

Functional Requirement 1. The system needs to be able to downscale a graph to a variable smaller size.

Functional Requirement 2. The system needs to be able to upscale a graph to a variable larger size.

Functional Requirement 3. The system needs to be able to run on multiple machines and divide the work between them.

Functional Requirement 4. The system needs to be able to read graphs in a standardized format.

This report has been written as the lab assignment of the Vrije Universiteit Distributed Systems master course (Dec 2020) taught by prof. dr. ir. Alexandru Iosup (a.iosup@vu.nl). This project was supervised by T. Hegeman (T.Hegeman@atlarge-research.com). The system can be found at https://github.com/OkkeVanEck/distributed_systems_lab.

Next to these functional requirements there are also some non-functional requirements that a DGS would benefit from. These non-functional requirements are meant to justify the development, the increase in resources, and the increase in monetary and energy cost of running a DGS.

Non-Functional Requirement 1. The system achieves speedup when running on more machines.

Non-Functional Requirement 2. The system should distribute the graph between machines and each machine should only need to access its own partition.

Non-Functional Requirement 3. The system needs to be able to handle graphs that are too large to fit in the RAM of a single machine.

Non-Functional Requirement 4. The system should exist out of free software. Users are allowed to run, copy, distribute, study, change and improve the software in any way they see fit [4].

3 SYSTEM DESIGN

3.1 Overview

DFF is implemented with 4 variable components: the simulation type, the input graph and its partitions, the number of utilized nodes, and the desired scale of the output graph relative to the input. It is important that the input graph is provided with partition files that are equal to the available number of compute nodes when running DFF. The partition files should also be in the Graphalytics format[5] or Metis format [7]. There are two types of forest fire simulations that can be run: halted forest fires and wild forest fires. In a halted forest fire simulation, a fire is ignited on each partition. If a fire spreads along an edge where the incident vertices are in two separate partitions, the fire does not burn to the neighboring partition (illustrated in figure 1). In the wild forest fire simulations, fires are allowed to burn across these edges, meaning compute nodes must communicate when a fire has spread to other partitions (illustrated in figure 2). This design choice was made to show the effects of compute node communication on total system performance and scalability.

3.2 Compute Node

In both simulations the compute nodes start by loading their assigned partition file and igniting one random vertex, which is placed into a *burning_vertexes* queue. Next, there is a fire phase and a communication phase. During the fire phase, each compute node pops the first vertex from the *burning_vertexes* queue and determines the unburned neighbors. 70% of the unburned neighbors are randomly chosen to be burned and are placed at the end of the *burning_vertexes* queue. For wild simulations, any unburned neighbors that are not on the same partition as the current burning vertex are placed into a *remote_burn_requests* list. This list is later used during the communication phase of the system. This fire spread operation is performed up to 40 times per compute node during the fire phase. After the fire phase is completed, the simulation enters a communication phase. In wild forest fire simulations all the compute nodes communicate with each other by collectively communicating with the lowest rank machine to the highest rank.

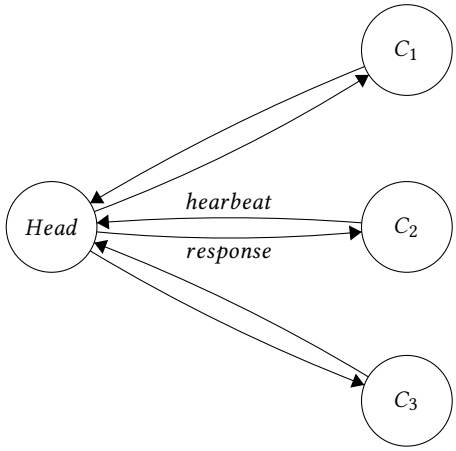

Figure 1: Communication between nodes when running in halted fire mode. Heartbeat contains the nodes that have burned, and response is whether the compute node should continue, reset for new sample, or shutdown.

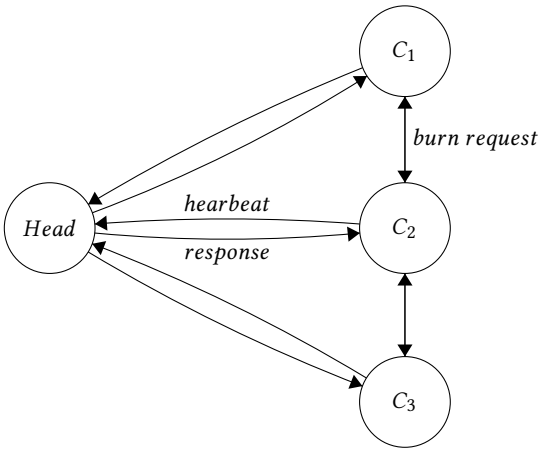

Figure 2: Communication between nodes when running in wild fire mode. Burn request entails the nodes that started burning but are on the other node.

This is implemented using *sendrecv* calls that are blocking to avoid complexity during the communication phase (illustrated by the double pointed arrows in figure 2). After the compute nodes have communicated with each other, they each send a heartbeat to the head node. In a halted simulation, the communication phase only involves heartbeats. A heartbeat consists of a list of edges that the compute node has burned from its assigned partition graph since the last heartbeat message. After sending a heartbeat, the compute nodes wait until the head node response is received. This response indicates whether they should continue sampling, shutdown, or reset and start again on a new sample. The reset and start response is used for upscaling simulations (illustrated by the single arrows in figures 1 and 2). If a compute node receives a reset and start request, all vertices in the partitioned graph are labeled unburned, and the compute node starts a new fire at a random vertex. Every

subsequent heartbeat will contain sampled edges from the new fire until either another reset message is received, or a shutdown message is received.

Compute nodes also have the ability to relight their fire. If a fire is started on a small component of the graph, it may have burned all of the vertices in just a few spread steps. To avoid having an idle compute node sending empty heartbeats, the compute node can choose to ignite unburned vertices. Naturally, these vertices can burn out again, therefore, the amount of vertices that are ignited is determined with an exponential back off, using the spread step as time. When a relight happens, n vertices are re-ignited. For our experiments, n=8. After relighting n vertices, the fire must burn an additional n vertices. If this does not happen, the compute node will relight $2n$ vertices the next time the fire burns out. If the fire does burn n new vertices, n decreases to $n/2$ for the next time relighting is needed.

3.3 Head Node

The head node is responsible for the scaling and stitching of the graph. It starts by determining the number of vertices in the original graph and computes the number of vertices needed to produce the scaled graph. This number of vertices is referred to as the cutoff vertex amount. Then, the head node determines how many samples need to be collected by dividing the scale factor by 4 and multiplying it by the number of vertices in the original graph. If a graph is scaled up by 2, the algorithm collects 8 samples of size 0.25. This strategy is also used to downscale a graph to anywhere above 0.5 times the original graph size.

After calculating run-time variables, the head node starts waiting for heartbeats from all the compute nodes (illustrated as heartbeat in figures 1 and 2). For every edge in a heartbeat, it will add the received edge to a Python set in order to remove any chance of duplicate edges. After receiving a heartbeat from every compute node, the head node checks whether it reached the cutoff vertex amount. It will send all compute nodes a response message to either continue burning, reset and move to next sample, or shutdown because sampling is done (illustrated as response in Figures 1 and 2). After finishing one sampling round, the head node performs a memory-saving operation, where it writes the currently obtained edges to a file and clears the edges list. Note that in this implementation, the difference between halted and wild simulations is completely handled by the compute nodes and that the head node does not differentiate between the two sampling methods.

When the head node has collected enough samples, it will start stitching. The stitching has 3 optional parameters. The first parameter tells the head node whether it should perform any stitching or return the graph as a group of (potentially) connected samples. The second parameter tells the head node which stitching topology to use, with the options being a random stitch or ring stitch. In a ring stitch, a sample vertex n is stitched to sample vertex $n + 1$ with a number of edges determined by the connectivity parameter. If the stitching topology is a random stitch, a number of random edges are made between all vertices. The last optional parameter determines the connectivity, which controls how many edges are added during stitching. It was added because an issue occurred with our upscaling implementation. Adding smaller samples together without stitching caused the number of edges to scale in the same additive way as the vertices, while the number of potential edges in a graphs increased by (N-1) when adding the Nth vertex. This caused a disconnect, which can be fixed by tuning the connectivity parameter.

3.4 Libraries

DFF uses the mpi4py library to provide its distributed functionality. The mpi4py library is built on top of MPI and implements a number of native MPI features into Python. We chose to use mpi4py mostly for ease of programming, since Python has was a language we were all familiar with and thus we only needed to learn the new mpi4py functionality. We also considered using threading in our implementation, to make sending and receiving message asynchronous, but we came to the realization that this would introduce unnecessary complexities for what we were trying to achieve.

4 EXPERIMENTAL RESULTS

4.1 Experimental setup

In order to test the system on large scale, the experiments will be executed on the DAS-5 supercomputer. The DAS-5 is a six-cluster wide-area distributed system designed by the Advanced School of Computing and Imageing (ASCI)[1]. It was an easy choice to use the DAS-5, as it is made for parallel and distributed work, and the Vrije Universiteit (VU) owns one of the clusters.

The DAS-5 uses the SLURM workload manager, which is an open source, fault-tolerant, and highly scalable cluster management and job scheduling system [9]. It satisfies our need to test our designed system at different scales, and it matches our wish to keep the entire setup open source. SLURM uses a job-based system, which implies that all the experiments need to be pre-defined and stored in SBATCH job scripts. In order to have an easy and consistent way to manage these jobs for the experiments, a bash script called *manage.sh* is created. In addition to managing jobs, *manage.sh* also takes care of the datasets that are used by the jobs.

For each experiment, a job is created via the *manage.sh* script, which allows for customizing multiple aspects. First, a simulation can be specified which defines the halted or wild version of the algorithm. Then, the scale factor, dataset, number of nodes, and job time are specified. After these, the three optional parameters for the stitching process can be specified. The first defines whether to stitch the resulting subgraphs or not. If the subgraphs will be stitched together, the second argument specifies whether the ring topology or the random topology is used. The last variable defines the connectivity that is used. Altering all these variables allow for many different experiments that will test multiple aspect of the designed system.

Table 1: Number of vertices and edges per dataset

Dataset	#Vertices	#Edges	Avg vertex degree
KGS	832.247	17.891.698	42.996
wiki-Talk	2.394.385	4.659.565	3.892
cit-Patents	3.774.768	16.518.947	8.752

The experiments down below will be conducted on three different datasets from Graphalytics [5]. The *KGS*, *wiki-Talk* and *cit-Patents* datasets are used because they offer different densities and have different ratio of vertices to edges. They are also of reasonable size, which is in accordance to the limited available time. The system converts the datasets into undirected graphs before scaling them. During the conversion, a directed edge is translated into an undirected edge. This way, the system can work with directed and undirected datasets. Table 1 contains the number of vertices, the number of edges, and the average vertex degree for each dataset after the conversions have been done. Note that the number of edges of *wiki-Talk* is lower than the number described by Graphalytics. This is due to the graph being converted from directed into undirected.

Before an experiment can be run, the used datasets need to be partitioned for the required number of compute nodes. In order to minimize the communication overhead between compute nodes thus minimizing the edges between partitions, the system uses an adopted version of the parallel graph partitioning framework ParHIP[8]. ParHIP is designed for networks with a hierarchical cluster structure and adopts a parallel label propagation technique from graph clustering field. This overcomes shortcomings of other parallel graph partitioning frameworks that only work well for regular mesh-like networks. ParHIP achieves better results than many other graph partitioning frameworks, like ParMetis [7] and PT-Scotch [2]. The system also incorporates a graph format converter to convert the LDBC Graphalytics graph data format into the Metis graph data format, which most graph partitioning frameworks support. After partitioning, the system will store one edge file and one partition file for each compute node. The compute nodes only need to load local edges and vertices, hence decreasing the time to load the graph data into memory.

During the experiments, multiple sections are timed in order to know what the bottlenecks are for the scalability of the system. The time it took to load a partition is stored as *init_partition*. The time spent computing what vertices are burned is called *do_spread_steps*. After every heartbeat, the compute nodes wait for a response of the head node. The waiting time is named *receive_from_headnode*. And lastly, the time it takes for sending the heartbeats is timed and named *send_heartbeat*. All given numbers in the results are averages across the compute nodes.

After the simulations, the resulting graphs can be analyzed as well. The Python package NetworkX is used to compute the number of vertices, the number of edges, the density, and the average vertex degree of the resulting graph. These numbers will be compared to the converted original graphs in order to show what effect the design choices of the system have.

4.2 Experiments

Multiple experiments have been conducted to show the scalability of the system. Both the halted forest fire and wild forest fire algorithms were tested in equal scenarios. The scenarios consist of downscaling and upscaling by a scale factor of 0.4 and 2. In order to show the scalability of the system for all scenarios, all combinations of these two parameters have been tested using different number of compute nodes. The tested number of compute nodes are 2, 4, 8 and 16.

Besides altering the simulation type and scale factor, it is also possible to change the stitching policy. The experiments test the effect of turning the stitching off, switching between the ring and random topology, and using a connectivity of 0.1 or 0.01. Changing the stitching policy should also be reflected in the resulting graph properties. Therefore, these will also be presented in the results.

All of the described experiments are executed for the three datasets. Each dataset has produced different interesting results, thus only those will be highlighted in this section. All graphs that are not discussed can be found in Appendix B. The full results for the graph properties can be found in Appendix C.

Figure 3 shows the results of downscaling the cit-Patents graph with a factor of 0.4. This is done using 2, 4, 8, and 16 compute nodes with the wild forest fire simulation.

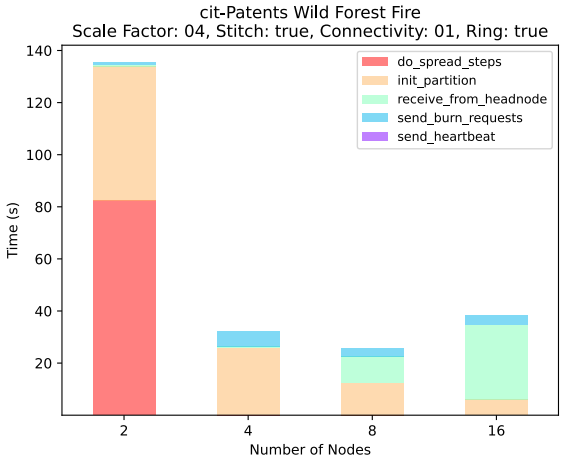

Figure 3: Result of downscaling wild forest fire using cit-Patents

The results from this graph show that DFF can have super linear speedups. We believe super linear speedups were achieved because, while the number of fires increases in a theoretically linear fashion, the number of burned edges sent in heartbeats can increase exponentially. This can occur if a well connected vertex is selected as the starting point of the fire.

Figure 4 shows the same downscaling by a factor of 0.4 on 2, 4, 8, and 16 compute nodes, but when using the halted forest fire simulation. This test is also performed on the same dataset as used in Figure 3.

In both Figure 4 and Figure 3, you can see that the time to read graph partitions decreases linearly with the amount of nodes available. Figure 4 also shows that the communication time between the head node and compute nodes increases as the number of compute nodes increases. This is also expected as the head node must process every heartbeat from a compute node before sending a message back to all compute nodes. This leads to a communication overhead that scales with the number of compute nodes.

Figure 5 and Figure 6 show the scalability results of upscaling a graph using wild forest fires and halted forest fires respectively. These experiments were run on 2, 4, 8, and 16 compute nodes using the wiki-Talk dataset.

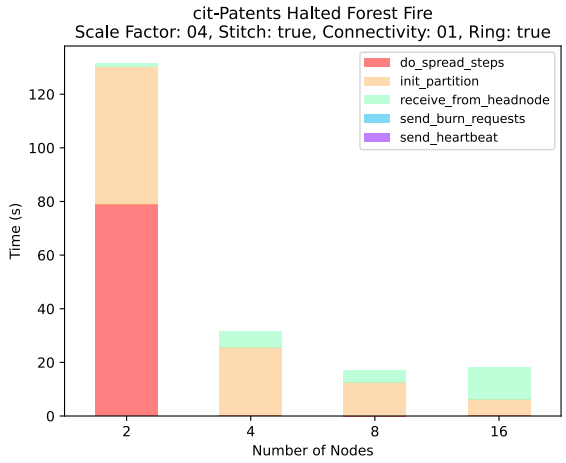

Figure 4: Result of downscaling halted forest fire using cit-Patents

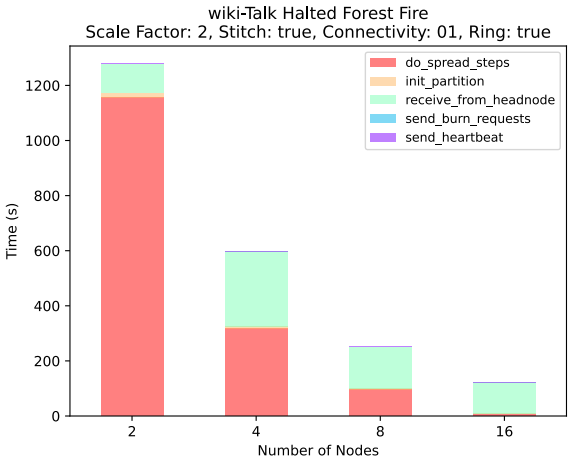

Figure 6: Result of upscaling halted forest fire using wiki-Talk

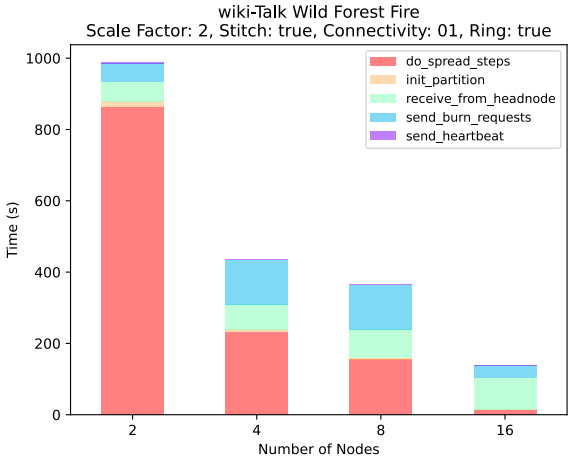

Figure 5: Result of upscaling wild forest fire using wiki-Talk

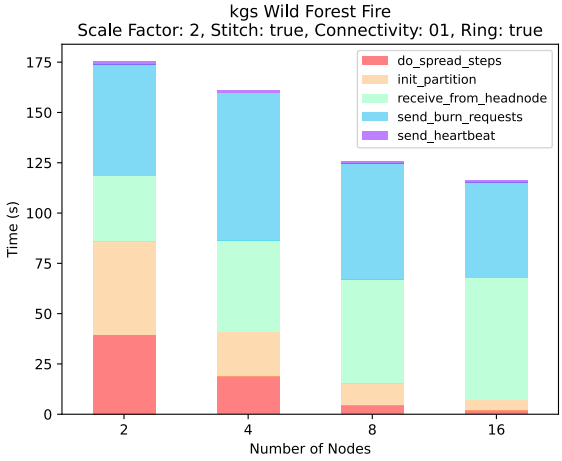

Figure 7: Result of upscaling wild forest fire using KGS

Both figures show super linear speedups from 2 to 4 compute nodes. The figures also show a speedup in computation time taken by the compute nodes when the number of compute nodes increases. It is also important to note that time compute nodes spend communicating starts to dominate the total time of the simulation as the number of compute nodes increases.

Figures 7 and 8 show the timing of compute nodes while upscaling the KGS dataset.

Interestingly enough, these results do not show the same super linear speedup results as the results from the wiki-Talk and cit-Patents datasets did. We believe this is because the KGS graph is a much more connected graph. KGS has 21.5 edges per vertex while wiki-Talk and cit-Patents have 2.1 and 4.37 edges per vertex respectively. In DFF, when the edge count is much greater than the vertex count, the number of edges sent to the head will also increase. This results in a higher number of vertices in the *remote_burn_requests* messages as well. Message processing time is counted as communication time and Figure 8 clearly shows that communication time overtakes computation time when graphs are well connected. In addition, communication time remains constant as the number of nodes increases. Something we have not yet investigated is why the time spent waiting to receive from the head node is much larger in the halted forest fires results than in the wild forest fire results. In these simulations, the head node should be processing the same amount of data.

Table 2 shows how the graph properties of our scaled graphs compare to the original. As you can see, many of the graph properties do not hold. We believe this is because of how the graphs were sampled and how they were stitched. For example, the average vertex degrees for the halted DFF is smaller than the wild DFF. Moreover, the number of components is always 1 when DFF

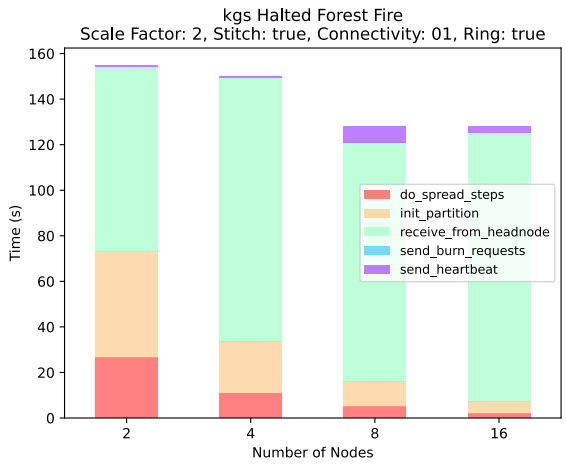

Figure 8: Result of upscaling halted forest fire using KGS

performs downscaling (the only 2 in the table is regarded as an outlier).

5 DISCUSSION

In the head node, our decision to iterate on graph samples of size 0.25 was made while performing initial tests. During these tests, we noticed that a size of 50% was a bit too slow for our taste and thus we lowered it to 25%. This resulted in faster scaling times. It also kept the number of samples within a comfortable amount. These observations were made using an early version of DFF where a geometric distribution was used for selecting the number of neighbours to burn. The latest version of DFF uses a linear scale, which is a lot quicker than the geometric distribution. Due to time constraints, we have not searched for the optimal values to run the latest version of DFF on, which could possibly speed up the whole process.

Our experiments demonstrated that DFF is scalable. As the number of compute nodes increased, we observed an overall decrease in the processing time of our compute nodes. In almost all simulations, the compute time (measured as *do_spread_tasks*) decreased linearly with the amount of compute nodes, and sometimes super linear speedup was achieved. Between the halted and wild forest fires, the results were mixed. Although sending burn requests between compute nodes increases the communication time of a compute node, the number of extra edges computed from the burn requests will sometimes compensate for it. In figures 5 and 6 the wild forest fire simulation is faster, while in figures 7 and 8 the halted forest fire simulations were either faster or had the same performance. Although these results were mixed, we believe with more experiments and larger input graphs, the communication overheads for wild forest fires will become more pronounced, and a more conclusive result can be achieved.

For workloads that are of magnitudes higher than those in our experiments, we think there will be an optimal number of compute nodes to use for some input graph. Our results have shown that time spent waiting for a heartbeat response increases with the number of compute nodes used in the simulation. This leads us to believe that for each graph, there will be a point where the additional wait time of adding a node will overtake the computational benefits provided by that node. However, the location of this optimum dependents on the size and properties of the input graph. We would need more research to predict where this optimum would be.

A problem with magnitude larger input graph is head node exhaustion. In DFF the head node keeps at most one graph sample (of size 0.25) in memory, as well as a list of all vertices of the scaled graph. The design can however be augmented to postpone this exhaustion. Firstly, the size of a single sample can be decreased, which increases the number of samples needed to produce the scaled graph. Secondly, the list of vertices could be optimized but it is a bit more difficult as the whole list is needed for stitching. By limiting the amount of vertices a stitched edge can be added between we could reduce the size of the number of vertices that needs to be kept in memory when moving from 1 sample to the next.

While fault tolerance was left out of our system design due to time constraints, we did consider it in our original threaded solution and it can still be found in some design decisions. Currently, the system does not care about how many vertices are submitted by each of the compute nodes. This decision was originally made for two reasons: to equalize the run time of the compute nodes, and because the threaded design used non-blocking communication and thus for most scale factors the system could still finish when a single compute node failed. However, the current single thread design uses blocking receives and so this is no longer possible in the system.

Our scaling algorithm performs poorly when comparing the graph properties of a scaled graph with the original. Only the vertex count behaves as desired. This may be due to the fact that we base the cutoff criteria on the amount of vertices. This criteria is also the reason why the edges are not scaled exactly according to the scale factor. When adding the Nth vertex, $N - 1$ potential edges can be added. Therefore, it would not make sense to scale the edges with the same factor. It also explains why the average vertex degree is off. However, we still believe we could improve these factors by expanding the cutoff criteria by adding a second scaling factor for the edges. This will allow the system to keep adding edges between existing vertices if the vertices cutoff is already met but more edges are desired.

The number of components in our output graphs do not scale with the original graphs by design. As explained in subsection 3.3, our algorithm contains stitching. During stitching, a number of edges are added to the output graph based on 3 input parameters. By configuring these parameters the user gets some control over the properties of the output graph. Users can indirectly control the component count, the density, average vertex degree, and number of edges. While all 4 properties could be controlled by the stitching parameters, stitching itself has especially great effect on the number components. By stitching, we add edges between unconnected vertices of different partitions. Therefore, the resulting graph combines the two components into one.

Finally, we would like to discuss our design decisions and whether or not we chose the right programming model for creating a DGS. We explained before that we made the choice to use mpi4py based

	Scale Factor	Compute Nodes	Edge Count	Density	Components	AVG vertex degree
Original	/	/	17,891,698	5.17e-05	6,099	43.0
Wild	2	2	3,209,551	2.32e-06	159	3.86
Wild	2	4	4,536,076	3.27e-06	102	5.45
Wild	2	8	5,367,224	3.87e-06	62	6.45
Wild	2	16	6,592,684	4.76e-06	87	7.92
Halted	2	2	3,720,941	2.69e-06	204	4.47
Halted	2	4	2,895,207	2.09e-06	285	3.48
Halted	2	8	5,093,709	3.68e-06	346	6.12
Halted	2	16	5,879,984	4.24e-06	335	7.07
Wild	0.4	2	1,310,133	2.36e-05	1	7.87
Wild	0.4	4	1,615,033	2.91e-05	1	9.70
Wild	0.4	8	1,954,964	3.53e-05	1	11.75
Wild	0.4	16	2,203,102	3.98e-05	1	13.24
Halted	0.4	2	807,938	1.46e-05	1	4.85
Halted	0.4	4	962,404	1.74e-05	1	5.78
Halted	0.4	8	1,509,344	2.72e-05	1	9.07
Halted	0.4	16	1,728,529	3.12e-05	2	10.38

Table 2: KGS scaled graph properties table

on ease of programming. However, after we committed to mpi4py we ran into some problems. First off, during the receive heartbeat there is a chance of receiving pickling errors because of an invalid load key. Our implementation only sends Numpy arrays, which is the datatype mpi4py was designed for. This was amplified by our second problem, the mpi4py documentation[3] is severely lacking and does not mention any possibility of pickling errors. Lastly, the blocking send was not blocking in our testing.

While working with MPI and learning more of its strengths and weaknesses, we came to the conclusion that MPI works best for large homogeneous problems where only simple messages are sent. Mpi4py limited our communication options greatly, we had to combine every send with a receive to keep the compute nodes and head node synchronized each iteration. This lead to a larger than necessary communication overhead.

We did try reducing this communication overhead by using different threads for receiving and sending messages. However, the communication threads needed to access the same lists as the main thread of each node, which would lead to complicated thread synchronization. Sadly, we decided this would not fit in the scope of our project. In the end, if we would remake our system we would change our programming model to one that better fits the requirements of a DGS, and one that gives us more opportunities to reduce overheads.

6 CONCLUSION

In this report a distributed graph scalar based on forest fire sampling algorithm is described. The conducted experiments confirm that the design is scalable. However, the experiments also show that the properties obtained from the system are not similar to the input graph. As discussed in section 5, we think many aspects of our system can be improved, including performance, property preservation, even the choice of programming model.

REFERENCES

[1] Henri Bal, Dick Epema, Cees de Laat, Rob van Nieuwpoort, John Romein, Frank Seinstra, Cees Snoek, and Harry Wijshoff. 2016. A medium-scale distributed system for computer science research: Infrastructure for the long term. *Computer* 49, 5 (2016), 54–63.

[2] Cédric Chevalier and François Pellegrini. 2008. PT-Scotch: A tool for efficient parallel graph ordering. *Parallel computing* 34, 6-8 (2008), 318–331.

[3] Lisandro Dalcin. Visited Dec 2020. *MPI for Pythonn.* https://mpi4py.readthedocs.io/en/stable/

[4] Free Software Foundation. Visited Dec 2020. *What is free software?* https://www.gnu.org/philosophy/free-sw.html

[5] At Large Research Group. Visited Dec 2020. *Graphalytics Datasets.* https://graphalytics.org/datasets

[6] Pili Hu and Wing Cheong Lau. 2013. A survey and taxonomy of graph sampling. *arXiv preprint arXiv:1308.5865* (2013).

[7] George Karypis, Kirk Schloegel, and Vipin Kumar. 1997. Parmetis: Parallel graph partitioning and sparse matrix ordering library. (1997).

[8] Henning Meyerhenke, Peter Sanders, and Christian Schulz. 2017. Parallel Graph Partitioning for Complex Networks. *IEEE Trans. Parallel Distrib. Syst.* 28, 9 (2017), 2625–2638.

[9] Andy B Yoo, Morris A Jette, and Mark Grondona. 2003. Slurm: Simple linux utility for resource management. In *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 44–60.

A TIME SHEETS

A.1 Dennis Wind

total-time	67h+
think-time	8h
dev-time	26h
xp-time	2h
analysis-time	4h
write-time	25h
wasted-time	undecided, with working at the same place as relaxing its hard to not think about assignment when trying to have fun.

Table 3: Time sheet Dennis Wind, times are rounded to hours

A.2 Tom Ebergen

total-time	70h
think-time	4h
dev-time	23h
xp-time	8h
analysis-time	3h
write-time	17h
wasted-time	15h. Our first iteration of our code wasn't usable, so some of these hours are double counted in dev-time.

Table 4: Time sheet Tom Ebergen

A.3 Okke van Eck

total-time	67h
think-time	6h
dev-time	29h
xp-time	8h
analysis-time	1h
write-time	19
wasted-time	4h

Table 5: Time sheet Okke van Eck

A.4 Yancheng Zhuang

total-time	53h
think-time	4h
dev-time	15h
xp-time	4h
analysis-time	1h
write-time	6h
wasted-time	25h. Waste a lot of time on developing and debugging on the first iteration

Table 6: Time sheet Yancheng Zhuang

B MISSING GRAPHS

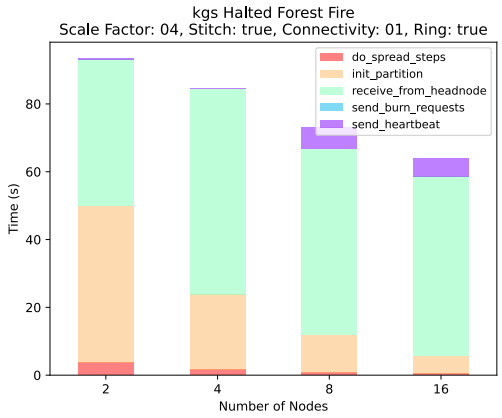

Figure 9: Result of downscaling halted forest fire using KGS

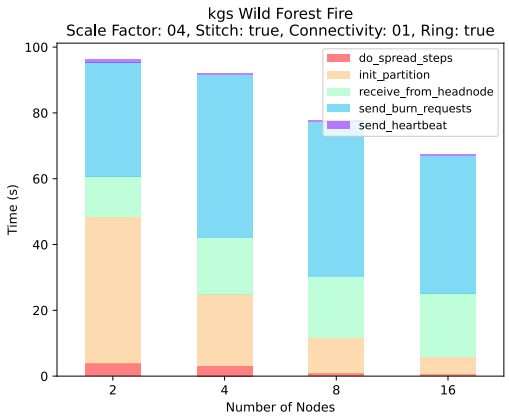

Figure 10: Result of downscaling wild forest fire using KGS

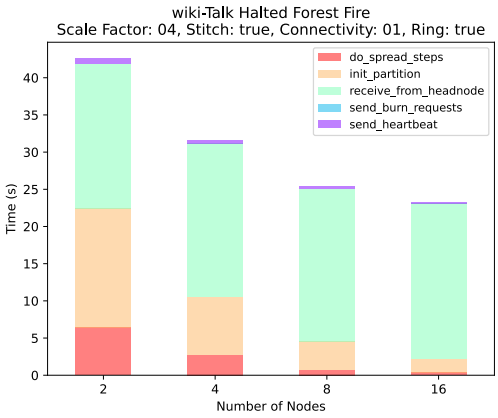

Figure 11: Result of downscaling halted forest fire using wiki-Talk

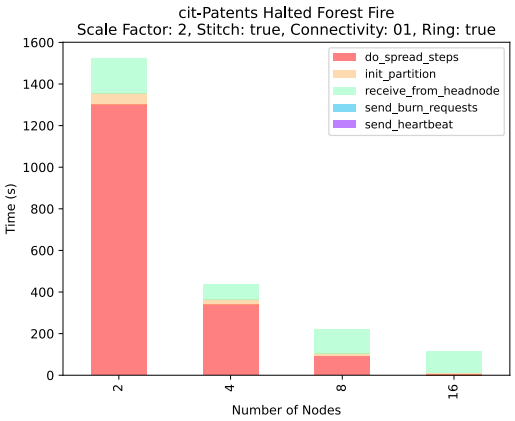

Figure 13: Result of upscaling halted forest fire using cit-Patents

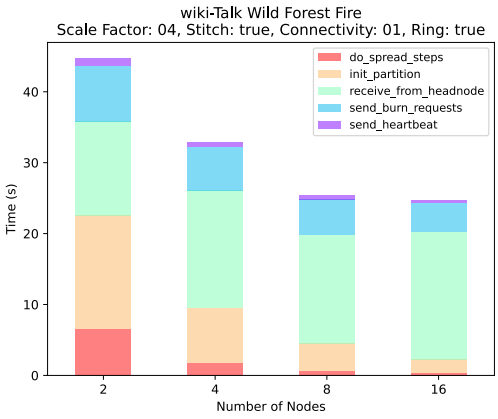

Figure 12: Result of downscaling wild forest fire using wiki-Talk

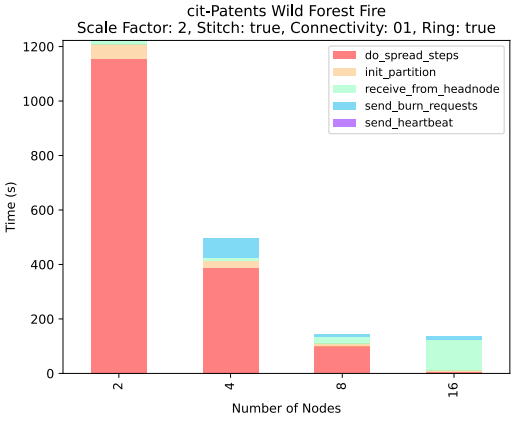

Figure 14: Result of upscaling wild forest fire using cit-Patents

C PROPERTIES TABLES

	Scale Factor	Compute Nodes	Stitching	Connectivity	Edge Count	Density	Components	AVG vertex degree
Original	/	/	/	/	4,659,565	1.63e-06	2,555	3.89
Halted	3	4	Random	0.1	11,170,252	4.33e-07	4,335	3.11
Halted	3	4	Random	0.01	10,443,817	4.04e-07	7,364	2.91
Halted	3	4	/	/	11,166,436	4.33e-07	4,800	3.11
Halted	3	4	Ring	0.1	11,162,712	4.33e-07	4,691	3.11
Wild	3	4	/	/	10,691,425	4.14e-07	6,989	2.98
Wild	3	4	Ring	0.1	11,400,246	4.42e-07	4,804	3.17
Wild	3	4	Random	0.1	11,474,547	4.45e-07	4,562	3.19
Wild	2	2	Ring	0.1	6,500,857	5.67e-07	3,851	2.72
Wild	2	4	Ring	0.1	7,377,359	6.43e-07	2,182	3.08
Wild	2	8	Ring	0.1	8,431,897	7.35e-07	1,876	3.52
Wild	2	16	Ring	0.1	8,659,677	7.55e-07	1,292	3.62
Halted	2	2	Ring	0.1	7,257,270	6.33e-07	2,776	3.03
Halted	2	4	Ring	0.1	6,381,116	5.57e-07	4,327	2.67
Halted	2	8	Ring	0.1	8,199,833	7.15e-07	2,102	3.42
Halted	2	16	Ring	0.1	8,295,627	7.23e-07	1,709	3.46
Wild	0.4	2	Ring	0.1	1,404,790	3.06e-06	1	2.93
Wild	0.4	4	Ring	0.1	1,625,347	3.54e-06	1	3.39
Wild	0.4	8	Ring	0.1	1,652,579	3.60e-06	1	3.45
Wild	0.4	16	Ring	0.1	1,924,783	4.20e-06	1	4.02
Halted	0.4	2	Ring	0.1	1,274,143	2.78e-06	1	2.66
Halted	0.4	4	Ring	0.1	1,473,013	3.21e-06	1	3.08
Halted	0.4	8	Ring	0.1	1,682,038	3.67e-06	1	3.51
Halted	0.4	16	Ring	0.1	1,801,950	3.93e-06	1	3.76

Table 8: wiki-Talk scaled graph properties table

	Scale Factor	Compute Nodes	Stitching	Connectivity	Edge Count	Density	Components	AVG vertex degree
Original	/	/	/	/	16,518,947	2.32e-06	3,627	8.75
Halted	3	4	Random	0.1	12,727,850	2.05e-07	122,130	2.28
Halted	3	4	Random	0.01	11,795,773	1.90e-07	266,565	2.11
Halted	3	4	/	/	12,638,691	2.03e-07	130,227	2.27
Halted	3	4	Ring	0.1	12,834,124	2.06e-07	118,605	2.30
Wild	3	4	/	/	12,128,118	1.94e-07	272,192	2.17
Wild	3	4	Ring	0.1	13.238.567	2.12e-07	100,912	2.37
Wild	3	4	Random	0.1	13.276.306	2.13e-07	103,275	2.38
Wild	2	2	Ring	0.1	8,504,779	3.07e-07	75,995	2.28
Wild	2	4	Ring	0.1	8,967,922	3.23e-07	70,883	2.41
Wild	2	8	Ring	0.1	9,359744	3.37e-07	53,477	2.51
Wild	2	16	Ring	0.1	9,581,204	3.45e-07	48,298	2.57
Halted	2	2	Ring	0.1	8,579,382	3.09e-07	81,040	2.30
Halted	2	4	Ring	0.1	8,199,815	2.96e-07	90,938	2.20
Halted	2	8	Ring	0.1	8,626,590	3.11e-07	80,786	2.32
Halted	2	16	Ring	0.1	8,930,198	3.22e-07	70,611	2.40
Wild	0.4	2	Ring	0.1	1,909698	1.68e-06	1	2.53
Wild	0.4	4	Ring	0.1	2,184,745	1.92e-06	1	2.89
Wild	0.4	8	Ring	0.1	2,355,331	2.07e-06	1	3.12
Wild	0.4	16	Ring	0.1	2,412,449	2.12e-06	1	3.20
Halted	0.4	2	Ring	0.1	1,797,653	1.58e-06	1	2.38
Halted	0.4	4	Ring	0.1	1,949,292	1.71e-06	1	2.58
Halted	0.4	8	Ring	0.1	1,953,494	1.71e-06	1	2.59
Halted	0.4	16	Ring	0.1	2,069,056	1.82e-06	1	2.74

Table 9: cit-Patents scaled graph properties table