Amstel: a framework for processing large-scale graphs

Claudiu Dan Gheorghe, c.d.gheorghe@vu.nl

Coordinators
Elzbieta Krepska, ekr@cs.vu.nl
Henri Bal, bal@cs.vu.nl

1. Introduction

Amstel is a framework for processing large-scale graphs implemented in Java by following the model published by Google, called Pregel (Grzegorz Malewicz 2010). This framework should ease the task of processing large-scale graphs by hiding the costs for running the computation in a distributed environment, similar how Map-Reduce framework does for large-scale arrays.

The computational model is based on Bulk Synchronous Parallel (BSP), a programming model proposed by Leslie G. Valiant for hardware-level computation (Valiant 1990). Programs are running in a sequence of iterations called Super-Steps, in which each vertex can send messages to other vertices or modify its own state. The user has to supply a Compute method that is run by the framework on each vertex at every super-step.

2. Implementation

Amstel is implemented entirely in Java on top of Ibis[1] communication library and is meant to run on commodity PCs like the ones from DAS3 and DAS4 clusters of Vrije Universiteit Amsterdam.

The distributed architecture is a centralized master-worker scheme with one dedicated master node that mainly does synchronization and multiple worker nodes that keep the vertices in memory and are performing the actual calling of the Compute method and sending messages.

Before describing the detailed architecture of each master and worker node, we describe the general phases involved a running instance of Amstel. There are four distinct phases:

1. Registration
2. Input
3. Computation
4. Output

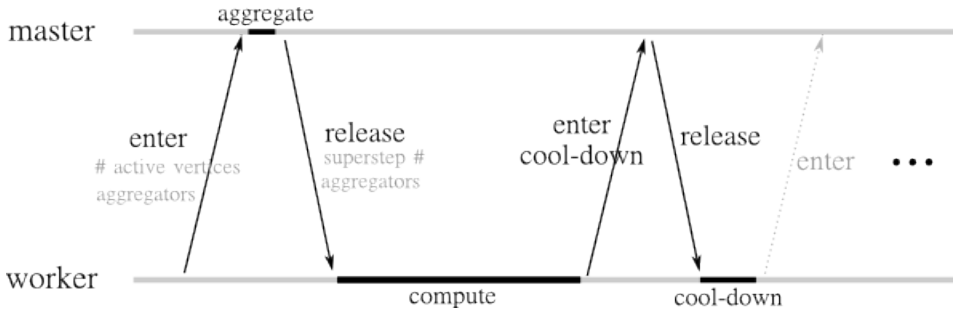
In the **registration** phase, the master node waits for a fixed number of workers to join the computation. Knowing the number of workers that have to join is also useful for partitioning the input data among the workers. As a response for a registration message, each worker gets a partition of the input data and the graph partitions as a response.

After all the workers register, they synchronize in a barrier and they proceed with the **input** phase. During this phase, each worker reads its part of the input data and distributes the vertices to the appropriate worker node, based on the graph-partitioning scheme.

The main phase is the **computation** phase that consists in running the actual algorithm on the loaded graph. The computation is performed in iterations called *super-steps*. All workers synchronize in a barrier at the beginning of a super-step and they

[1] http://www.cs.vu.nl/ibis/

start by running the Compute method on each vertex, buffering and eventually sending the messages for the next super-step. When all messages have been sent and confirmed each worker enters in a barrier and after all reach this point they enter in a phase called cool-down phase, in which it is safe to switch the message inboxes and to determine which vertices will be active in the next super-step. The number of active vertices is reported to the master, which takes the decision. If there is no active vertex means that the algorithm finished and it sends the exit message to all workers. Otherwise, the super-step number is incremented and another iteration takes place.



After there is no active vertex and before exiting, the master node may instruct the workers to write their vertices or other information in persistent storage. This makes the **output** phase, but it is not implemented yet, since we don't have a distributed storage system that can handle such a huge amount of information efficiently.

Master node implementation

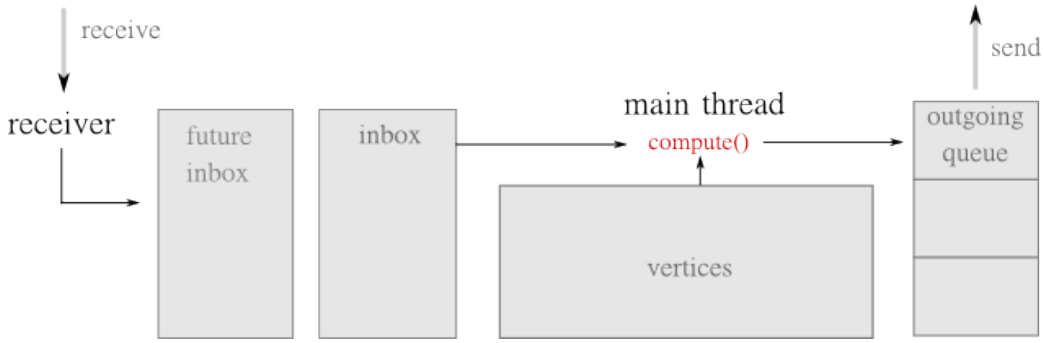
The master node is simple and its main role is to synchronize worker nodes in a barrier at each super-step. The master node is elected using the IPL election mechanism before the registration phase takes place.

The master node is also responsible for splitting the input and for assigning the graph partitions to workers. After it gets a register message it replies with one of the input partition and with the entire map of the graph partitions. For now a simple graph partitioning scheme is used; the graph is divided into a number of parts equal with the number of workers.

The barrier is implemented using simple message passing by using the blocking receive primitive.

Worker node implementation

The worker node contains the core functionality of the framework. It is responsible for keeping in memory the vertex state for its own partitions of the graph, running the Compute method on each super-step and sending all the messages before the next super-step starts. As it will be noticed, the most difficult task is delivering the messages efficiently.



The figure above depicts the architecture of a worker node. It contains two threads, one for receiving messages from the other worker nodes and a main thread that is responsible for executing the compute method on each vertex, sending messages to other threads and synchronizing with the master node. During a super-step, the main thread runs the Compute method on each vertex by loading the messages from the inbox and offering an iterator.

One of the key design features of Amstel was to avoid creating objects especially in the critical path (compute method and sending messages), as the object instantiation is a quite expensive operation in Java. Therefore for running the Compute method on each vertex we change only the state of the vertex that switches the context completely.

Messages are buffered in the outgoing queue grouped by their destination so each worker has a message buffer for each other worker and their vertices. The outgoing buffers are raw byte arrays and the messages are serialized into these buffers in the moment they are sent, thus minimizing memory consumption. When the outgoing queue reaches a certain threshold, the few biggest buffers are flushed and sent to their destination. When they reach the destination, the receiver thread unpacks the target vertices and keeps the bulk messages in serialized format. These messages will be deserialized on demand, in the moment when the vertex accesses the messages in the Compute method, using the message iterator.

The main limitation of the framework is physical memory, as we have to store in memory all vertex states. For each vertex we have to store the following fields:

- vertex ID: a string, say around 32 bytes;
- vertex internal index: int, 4 bytes;
- vertex value: an object, 8 bytes;
- edges: array of String, E x 32 bytes
- edge values: array of objects, E x 8 bytes

If E is the average number of edges per vertex, the total amount of memory for storing graph state is

$$Mem = V \times (40 + E \times 40)B \cong V \times E \times 40\ B$$

so for a graph with 1 million vertices and 200 edges per vertex we need at least 8GB of memory, which means at least 3 worker nodes from DAS3.

Storing vertex IDs

One of the optimizations used to improve memory consumption was the management of vertex IDs. A vertex ID is a unique String ID of the vertex and it's used

for partitioning the graph based on the hash code of the String. The same vertex IDs appear also in the targets of the edges, the dominant factor of memory complexity.

Having the fact that the vertex IDs are immutable and with read-only usage, we can avoid storing the String multiple times on the same worker node. The solution is to create a static pool of String objects indexed by the String content itself, so every time we add a new vertex or a new edge, we check the vertex ID in this global pool, reusing the existing object if this is the case. Even this pool grows with the number of vertices and can become huge, it will be queried only in the input phase, when workers read input and exchange vertices.

This optimization has reduced the memory footprint with 2 to 3 times and the execution time with around 10%.

Combiners

Combiners are commutative and associative operators that can be applied to messages to combine them and to reduce memory consumption and network bandwidth. There is no way to find a combiner for a type of message automatically, because it depends on the algorithm itself, so the user must specify one before the computation starts. For example, in the Single Source Shortest Path (SSSP) algorithm, a *min* combiner can be used for the distance carried by the messages.

Using combiners in Amstel drastically changes how the messages are managed. The outgoing queue implementation is much simpler and uses a single message object in place of serialized buffers. Every time a message is sent it is combined with the existing one, if the case. The flushing policy is also different and based only on the number of destinations in the queue. The inbound queue (inbox) is also simpler by keeping just one message for each vertex, so the vertex will always have at most one message in the inbox that will contain the combined messages from all workers. Therefore we apply the combiner both at the sending and at the receiving part, reducing memory footprint and execution time.

Aggregators

Aggregators are mechanisms for global communication, synchronization and data. They contain a commutative and associative operator similar with a combiner and the user has to specify it. Vertices can output a value to an aggregator, which ends up being aggregated by each worker and finally there are collected by the master node, which will aggregate again and will make the result available in the next super-step.

The aggregator values are encoded along with the barrier synchronization messages and they can be observed in the figure that illustrates all the phases involved in a running instance of Amstel.

3. User API

The API exposed to the user is focused on specifying the Compute method that will be run on each vertex and is very similar with the API described by Pregel.

Firstly, the user has to extend the abstract Vertex class and it has to specify the parameterized types for vertex value, edge value and message value. It is mandatory to

implement the abstract method Compute that takes as argument the message iterator and implements the desired algorithm.

The message and edge values have to extend a custom Value class that forces the user to implement object's serialization. This might be inconvenient but it was necessary in order to implement message passing simple and to avoid keeping objects in place of byte arrays. There are predefined values for integer values, which is the most common option.

Secondly, a runner class that contains a *main* method must be implemented. Here is where the user configures, instantiates and launches the execution of Amstel. Besides the number of nodes that will be used (including the master node), the user has to specify Class objects for each generic type due to erasure in Java Generics. The user must also specify a Reader object that will be used for partitioning the input and reading the graph. For now there is just one format implemented, a simple text file format, but the interface is decoupled so the user may freely implement its own storage and encoding.

Optionally, the user may specify a combiner class or use aggregators. There are predefined combiners and aggregators with min/max/sum operators for integer values.

Another important note is that the user can specify a NullValue value for edge values, in case the edges values are not used in the algorithm. Using this custom value will reduce the memory size a lot because in this case the edge values arrays will not be instantiated at all.

4. Experiments and Performance Analysis

To test the performance of the framework we have conducted a series of experiments on the DAS3 cluster using 4 to 64 nodes. The tests were performed on the SSSP algorithm by using a simplified version that does not uses edge values and considers an implicit unitary cost. Also a Min combiner is used for the messages.

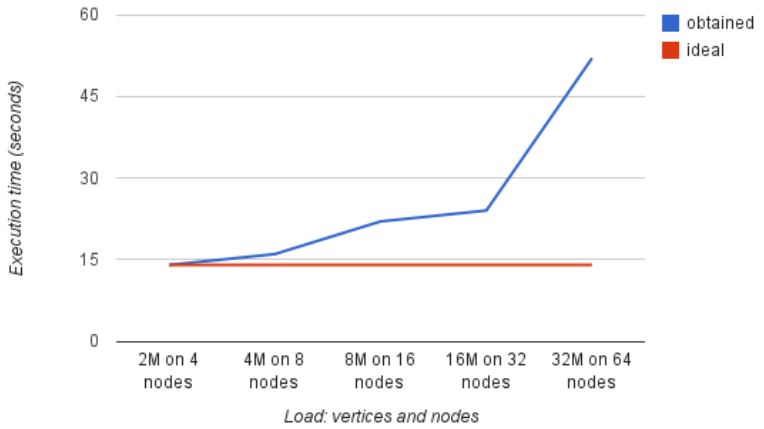
The graph used is a generated graph with a simple scheme as depicted above. Each node is linked to the E next nodes. Almost all tests have E set to 127 and the number of vertices ranges from 2 million to 32 million. The generation of the graph it is done by implementing a custom Reader interface, because reading the vertices from a file would take too much space and time. For example, a text file that stores a graph with 1 millions vertices and 200 edges takes 2 GB on disk and leads to around 2 minutes loading using 4 nodes. More than that, the generator is implemented such that all vertices are generated directly on their host machines so we also avoid sending data over the network in the input phase. Even so, the input phase is still dominant and takes at least 30 seconds to complete.

The biggest limitation for Amstel is memory. The parameters for the JVM were to use a 2.6GB heap (maximum achieved on DAS3) and to use the Concurrent Mark Sweep Garbage Collector (the best performing for Amstel). With this configuration we managed to fit a maximum of 2 million vertices per one machine.

All execution times refer only to the compute phase and they exclude the input phase, which takes minutes even with the most efficient generator.

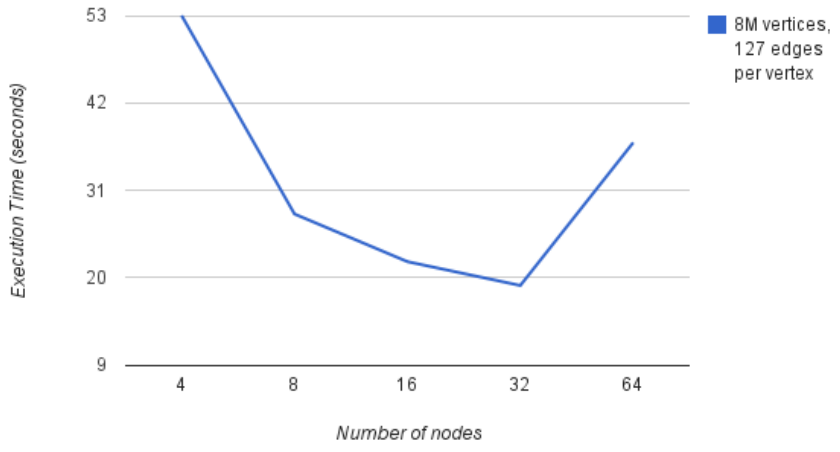
Another important note is that we use a fixed number of super-steps in the tests, because this number can vary based on the input graph on the algorithm. We considered a limit of 1000 super-steps for the SSSP algorithm.

Scalability



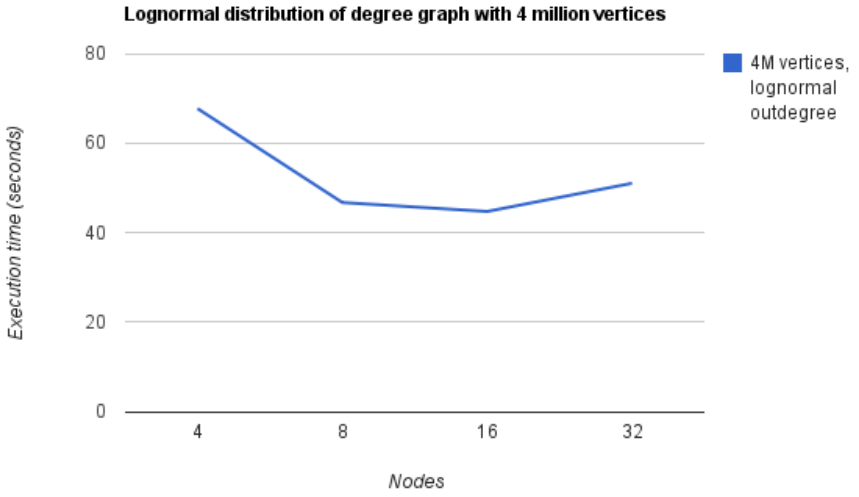
	2M on 4 nodes	4M on 8 nodes	8M on 16 nodes	16M on 32 nodes	32M on 64 nodes
execution time (seconds)	14	16	22	24	52

The first test for scalability was to double both the problem size and number of nodes. Ideally, the execution time should remain the same. The first test uses 2 million vertices and runs on 4 nodes, where the largest uses 32 million vertices and runs on 64 nodes; all vertices have 127 edges. As it can be observed, the execution time increases significantly when starting from 32 nodes, as Amstel performs global synchronization at each super-step. The need for synchronization is the biggest noticed bottleneck, as it will be shown below. Even if synchronization time cannot be avoided, increasing the problem size can mask it. Anyway, we cannot increase the problem size too much as all vertices must fit in memory.



nodes	4	8	16	32	64
8M vertices, 127 edges per vertex	53	28	22	19	37

The second test considers a fixed problem with 8 million vertices and it is run on 4 to 64 machines. As it can be observed, the execution time decreases up to 32 nodes, but when using 64 nodes it takes even more time to execute than using 16 nodes, which was expected due to global synchronization on each super-step. When using 64 nodes, each node will manage around 125 thousand vertices, which can be done instantly and in practice execution time is mostly about synchronizing at each super-step.



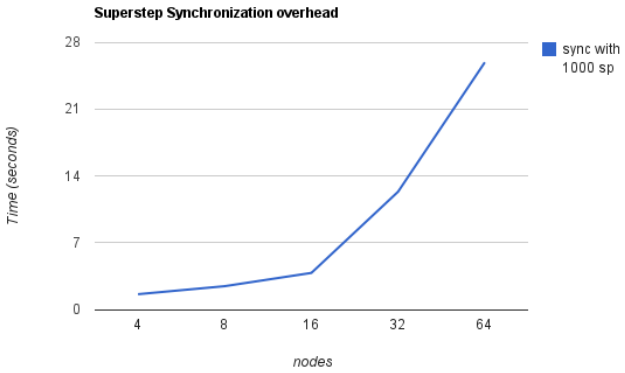
	4	8	16	32
4M vertices, lognormal outdegree	67.7	46.7	44.7	51

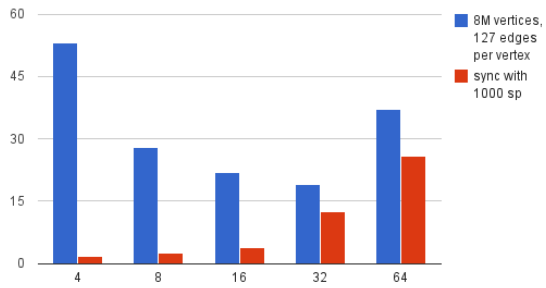
Another test, which is depicted in the figure above, is performed on a random graph with lognormal distribution of the out-degree, with a mean value of 127 edges. The execution times are bigger compared with the wheel graph because it involves much more communication.

For this framework the execution time and speedup are not a viable metric for showing the system scalability, which is a similar aspect with the Map-Reduce paradigm. By increasing the number of nodes, one cannot expect that will run infinitely faster, as the synchronization time increases linearly with the number of nodes. The purpose of this framework is that it makes possible to solve problems that cannot be solved on a single machine. Actually the number of nodes used is not something that should concern the user; for a given graph, the system should be able to find the optimum number of nodes for processing the graph.

Synchronization overhead

To compute the synchronization overhead a suite of tests with very small graphs have been used, with 10 thousands vertices and 5 edges per vertex. With such a small graph, the execution time will show only the time needed to synchronize at each super-step; a number of 1000 super-steps was used, as previously.



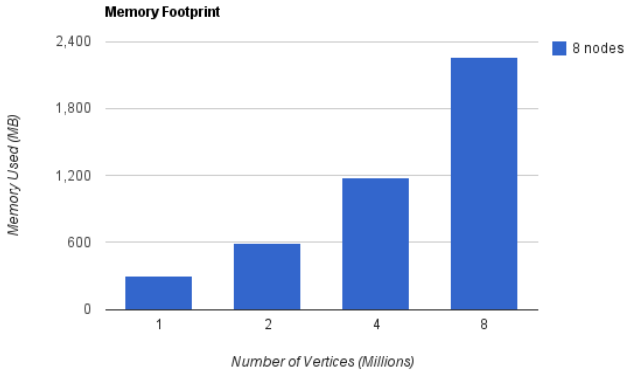


In the left graph we can observe how the synchronization time increases with the number of nodes. For these test the evolution tends to be linear; to be noticed that the number of nodes increases exponentially. This synchronization time is not dependent on the input data and it is considered an invariant; it is not influenced by input data nor by the nature of the algorithm and it is mainly influenced by network speed.

In the right graph the execution times from the previous test (with blue), using 8M vertices, are placed together with the estimated synchronization time (in red). As it can be observed, the major source of the scalability issue is the synchronization time at the end of each super-step.

Memory

The main purpose of Amstel is to fit as many vertices as possible in one single machine. This improves the system performance because it increases utilization level on each super-step and it can mask synchronization time. Still, Java is not the best environment for optimizing memory consumption and this is one of the biggest limitations of Amstel.

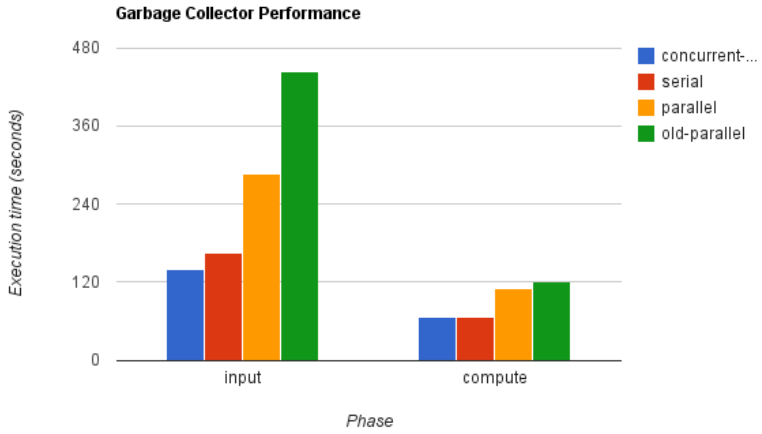


As it can be shown in the graph, the memory usage increases linearly with the input size. This information was collected using the garbage collector log on the worker nodes, while processing a graph with 1 million to 8 million vertices (E is fixed to 127). The indicated bar shows how much memory one worker uses.

Not only vertices state influence the size of memory needed. On each worker node we have to keep buffers for each outgoing destination, in the worst case for each node in the graph, so the memory space evolves faster than linear with the number of vertices.

It should be noticed that these tests are making the lowest memory consumption possible by having a combiner for messages and by using implicit unitary cost (1) on each edge, therefore using NullValue as edge value type. In a normal scenario with serialized output buffers and explicit edge values, the memory needed would be much higher.

Another concern regarding memory is the option of the garbage collector. Amstel has been tested with all available garbage collectors (Serial, Parallel, Old-Parallel, Concurrent-MS) and the best one is seems to be the Concurrent-MS, since this one offers low-latency and avoids stop-world garbage collecting for the old generation. To support this conclusion you can see the results for running a SSSP with lognormal distribution out-degree on 4 nodes:



5. Future work

This is the first version of the framework that covers the basic functionality needed to run simple algorithms. Nevertheless there are still ways of improving the framework, and the most significant are:

a) Multithreaded workers: currently only the main thread processes the vertices, but having multiple threads doing this will make use of the multi-core CPUs from DAS3. This means that the master node has to distribute multiple graph partitions to the same worker node, and that's also benefic for load balancing.

b) Distributed filesystem dedicated for graph storage: I/O is the biggest bottleneck that the framework currently has. Running algorithms for huge graphs means also reading efficiently this huge information and on the current version the input phase takes even more than the entire algorithm computation. Also due to limited I/O capabilities (text files for now) we cannot implement Writers because we cannot handle and coordinate efficiently such a huge amount of information.

c) Optimum number of nodes: given an input graph and knowing in advance the memory requirements and the synchronization time, Amstel can find out the optimum number of nodes to use for a certain computation. Different parameters can be optimized, like resource utilization or execution time.

d) Topology mutations: doing the topology mutation operations like adding or removing edges should be straightforward since the vertex state is entirely in memory. The main problem is that if we do topology mutations, we would eventually want to write the new graph in a persistent storage, and we reach the Writers problem mentioned before.

6. Bibliography

Valiant, Leslie G. *A Bridging Model for Parallel Computation.* Comm. ACM 33(8), 1990.
Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. *Pregel: A System for Large-Scale Graph Processing.* Google, 2010.