Digs: Distributed Bioinformatics

Release 1.0a1

Tycho Marinus **Lucas van Dijk**

Jul 27, 2017

Contents

Introduction

In the past few years sequencing the genome has become a lot cheaper, due to next generation sequencing techniques. It is now a lot more viable to sequence the genome of an organism, and for example compare it to a known reference genome. The human genome consists of three billion base pairs, and some plant genomes are sometimes an order of magnitude larger.

This data avalanche provides a lot of opportunities for biological problems. But the amount of data is huge and a lot of operations are computationally expensive. Luckily, these operations are often quite easily executed in parallel. A few examples are discussed in the following sections.

Next Generation Sequencing

A typical NGS pipeline consists of the following steps:

1. A genome sequencing machine produces a lot of independent short reads (around 200 base pair) originating from random locations of a new genome.
2. Try to map these reads to a known reference genome
3. Determine variant genes in the newly sequenced genome

These short reads can be independently mapped to a reference genome, which makes it easy to distribute these reads across multiple workers. Decap et al. built a distributed system based on Hadoop *[decap2015halvade]*.

Multiple Sequence Alignment/Phylogeny

If you have a lot of genome or protein sequences, you can try to match each sequence to each other, a process called alignment. If you have a collection of aligned sequences, you can calculate the distance between each combination of sequences, and from there you can generate a phylogenetic tree.

This process of multiple sequence alignment and distance calculation can also be quite easily performed in parallel, although it requires a merge step at the end.

A Distributed System For Bioinformatics

We would like to design a system which can run bioinformatics software easily on multiple workers. To bound our project a bit, we will focus on the multiple sequence alignment problem.

The program we use for multiple sequence alignment is called MAFFT *[mafft]*, and for this project we want to run this program in a distributed manner. It supports both multiple sequence alignments on genome nucleotide or protein amino acid sequences. We have chosen this tool because it is simple to use, and easily run in parallel.

Requirements

In this section we discuss the following two use cases we have in mind for our distributed system: first a use case where the “life science” group in your organisation has performed a sequencing experiment, and you want to reliably store this data, the second use case being the bioinformatics group wanting to perform some analysis on this data.

Use Case 1: Store Datasets

Wether you collect the genomes of a large number of organisms, or just performed a new sequencing experiment, the data should be stored safely. The biological lab work is often done by a different group than the group performing the actual analysis on the data. The data coming from such experiment should be stored for later analysis.

This brings the following challenges:

- The huge amount of data: a human genome with 60x read coverage depth can occupy easily 200 GB in its compressed FASTQ file format.
- The data needs to be stored persistently and reliably.
- The data needs to be accessible by other teams
- Analysis and other actions need to be performed on this dataset, and the results should be stored too.

Use Case 2: perform analyses on the data

When the data is safely stored in the database, an organisation probably wants to analyse this data. Think of building a new phylogenetic tree based on a multiple sequence alignment of a collection of genomes. In the case of next generation sequencing you can think of mapping individual reads to a reference genome or locally align them, assemble a new genome from the individual reads, or check if this newly sequenced genome has any variant genes compared to the reference.

Most of these operations are computational expensive, but as discussed in the previous section, a lot of these operations can be performed in parallel, on smaller chunks of the data. Building a scalable distributed system for these kinds of pipelines could reduce the computational time significantly.

Requirements Prioritisation

For our system, we focus on computing the multiple sequence alignment of a large collection of genomes or protein sequences.

Must Have

- Built a distributed system that computes the multiple sequence alignment of a large collection of genomes or protein sequences.
- The data must be stored consistently and reliably.
- Fault tolerant, when one of the nodes crashes it should not hinder the final results. Able to resist one node down of any kind at the same time.
- Scalable, must be able to handle large genomes

Should have

- Let the user define a workflow, which specifies with steps to perform, and which steps depends on which previous steps.
- Multi-tenancy: let multiple teams perform different actions simultaneously.

Could Have

- Data-ownership: who can see which datasets

System Design

The general idea is to have a multi cluster architecture with centralized scheduling. There are three kind of nodes:

- Central manager nodes
- Data nodes
- Computational worker nodes

An overview of the system can be seen in Figure 1.

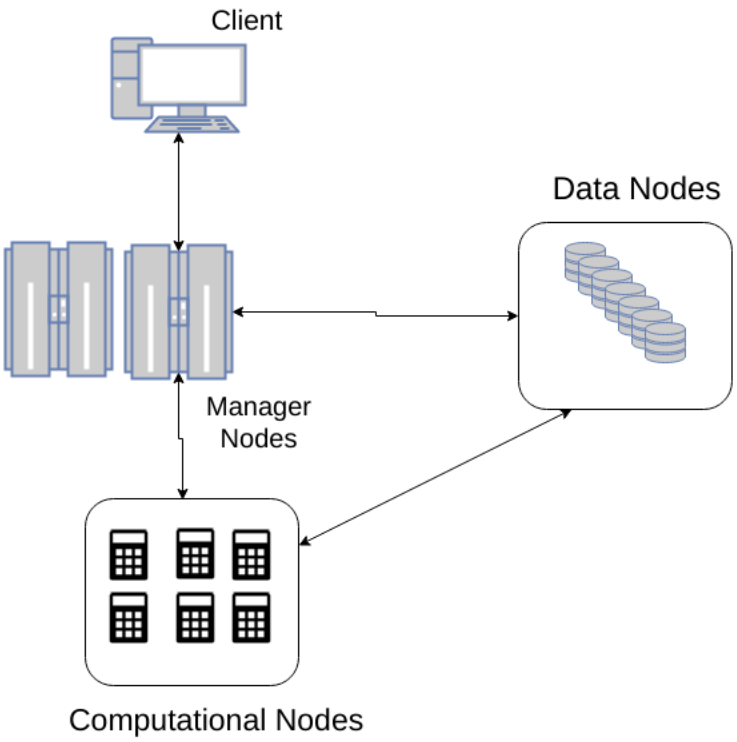

Figure 1: Global system overview

Central Managers

The central managers are the most intelligent nodes of the system and coordinate the whole system. Central managers keep track which datasets are stored on which data nodes, and handle the requests for certain analyses on a dataset. They also take care of splitting a task into multiple subtasks, which can be distributed across workers.

Central managers store a lot of metadata, and the idea is to replicate this data on all central managers. This data is stored in a PostgreSQL database, and using pgPool-II *[pgpool]* it is possible to setup the replication system. The persistent messaging middleware also runs on the central manager, and in our case we use RabbitMQ *[rabbitmq]* as

our message queue. RabbitMQ has built in functionality to share message queues across multiple nodes, and keep those queues consistent.

We assume all nodes will be located in the same data center (it is a distributed system for a single company or research group), and therefore the chance of a network partition is considered low.

Data Nodes

All datasets are stored on dedicated "data nodes". To prevent any data loss, and to retain high availability/access of the data, the central manager makes sure that each dataset is stored on multiple different nodes. Clients communicate with the central manager about where to store initial new data, however the transfer is done directly from client to data node to not unnecessary increase data transfers. We specified in our fault tolerant requirements that the system should be tolerant to at most one unavailable node of each type, and thus each dataset is stored on two data nodes.

Data nodes are a simple distributed file system, but they also provides some more intelligent functionality. They can calculate proper byte offsets to split a file into multiple chunks for a limited set of file types. We do this calculation on the data node, because otherwise you would need to transfer the possibly large dataset to another node.

Computational Worker Nodes

The worker nodes are the power horses in our system: they perform the actual computationally intensive operations on datasets. The central managers build a queue with tasks waiting to be run, and these worker nodes can pick one this queue.

Each task in the queue contains the metadata which program to run and on which dataset (and optionally a chunk start and end byte offset). When a worker node picks a task, it asks the central manager which data node it should contact to retrieve the data, and continues to download that dataset from the given data node. After the program has finished running, the results can be stored on a data node again.

Task Descriptions

In this section we discuss the steps required to perform several tasks.

Upload Dataset

1. Request from a random central manager a data node that is available and has enough diskspace for the new dataset
2. The client uploads the data to the given data node, using rsync. Rsync makes sure the data uploaded is actually correct.
3. The client notifies a central manager when the upload has finished
4. The central manager coordinates the data duplication, and makes sure the dataset is stored on two data nodes.

A schematic overview can be seen in Figure 2. All this communication is transient: it only makes sense to transfer the data if the data node is available.

Job Request

1. Client sends a job request to the manager: which kind of program, on which dataset

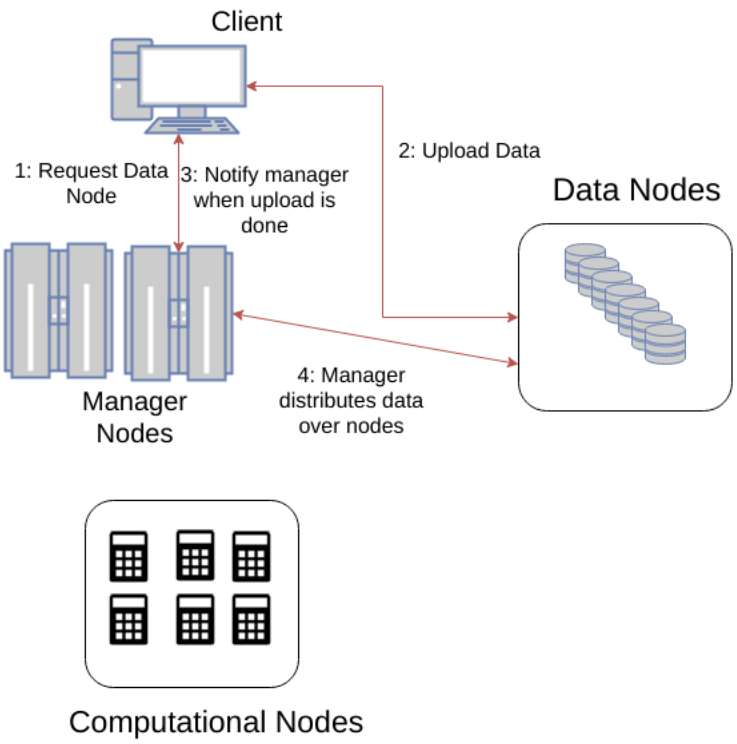

Figure 2: Schematic overview of all steps required to upload a dataset.

2. Central manager divides the job in subtasks, by asking the corresponding data node to parse the dataset, and return byte offsets where a dataset can be split (not shown in the image). Each subtask is put on the queue, and available workers can pick these subtasks from this queue.
3. Worker nodes download the corresponding datasets or chunks from the data nodes, and start performing the task.
4. Results can be stored on data nodes again.

A schematic overview can be found in Figure 3. Most of this communication is persistent: clients can send a persistent message to the central manager, and can periodically check if their job has finished afterwards.

Performing a subtask

1. *[persistent]* An available worker picks a job from the subtask queue. This task contains the following metadata: a dataset/file ID, which program, and the chunk start and end byte offsets.
2. *[transient]* A worker asks the central manager which data node to contact to retrieve the dataset. If the data node appears offline, the worker notifies the central manager, and the manager will send an alternate data node.
3. *[transient]* The worker downloads the data chunk from the given data node.
4. The worker starts the program, and when finished stores the results back on the data node.

Currently, the program is always MAFFT *[mafft]*, which can quickly calculate a multiple sequence alignment for large collections of genomes. It also supports merging independent alignments to a single alignment, which is useful to merge all results calculated by workers to a single alignment result.

If something goes wrong, and the worker can gracefully handle this error, we notify the RabbitMQ server to requeue

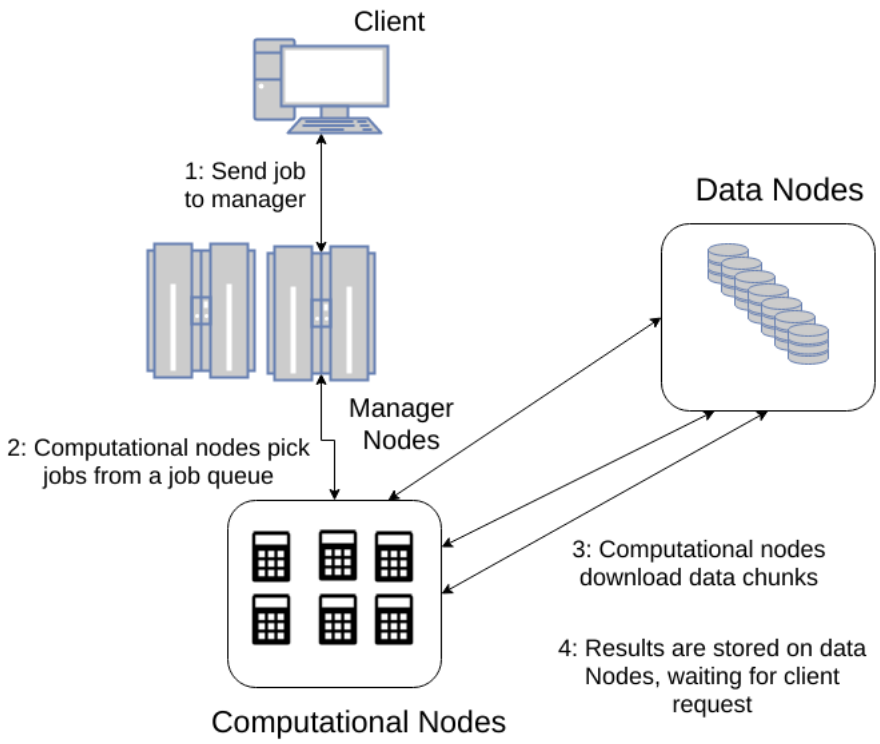

Figure 3: Steps to perform a large job

the subtask. The RabbitMQ server also checks if each worker is still alive, and when one worker dies without acknowledging the completion of a subtask, this subtask is again requeued.

When MAFFT finishes successfully the worker uploads the results to a data node, and acknowledges the completion of a subtask to the RabbitMQ server.

Experiments

To test our system we have performed several experiments. These are described in the sections below.

Real World Dataset And Scalability

We want to test if our system could handle a real world dataset. To test this we compiled a dataset with 550960 sequences from different organisms containing the amino acid sequence of a "ribosome": the cellular machine that converts RNA to a protein. Practically all life forms have this machine. This dataset is downloaded from Uniprot *[uniprot]*, a collaboration between different institutions providing annotated protein sequences. The 550960 sequences result in a dataset of 253MB. To see how our system scales with an even larger dataset, we duplicate the data and create a few more "artificial" datasets of 506 MB, 1012 and 2024 MB.

We measure the running time as follows: from the moment the manager puts the first subtask on the queue until all subtasks have finished. This does not include the preprocessing stage (to determine proper byte splitting offsets), but does include the transfer of data chunks from the data nodes to the worker nodes.

The experiment is run on the trial version of Google Compute Engine, which limits the number of VM instances. We measured the running time with one, two, three and four available worker nodes. The running times can be seen in

Figure 4. To see how the performance is affected with even more worker nodes is recommended for future work.

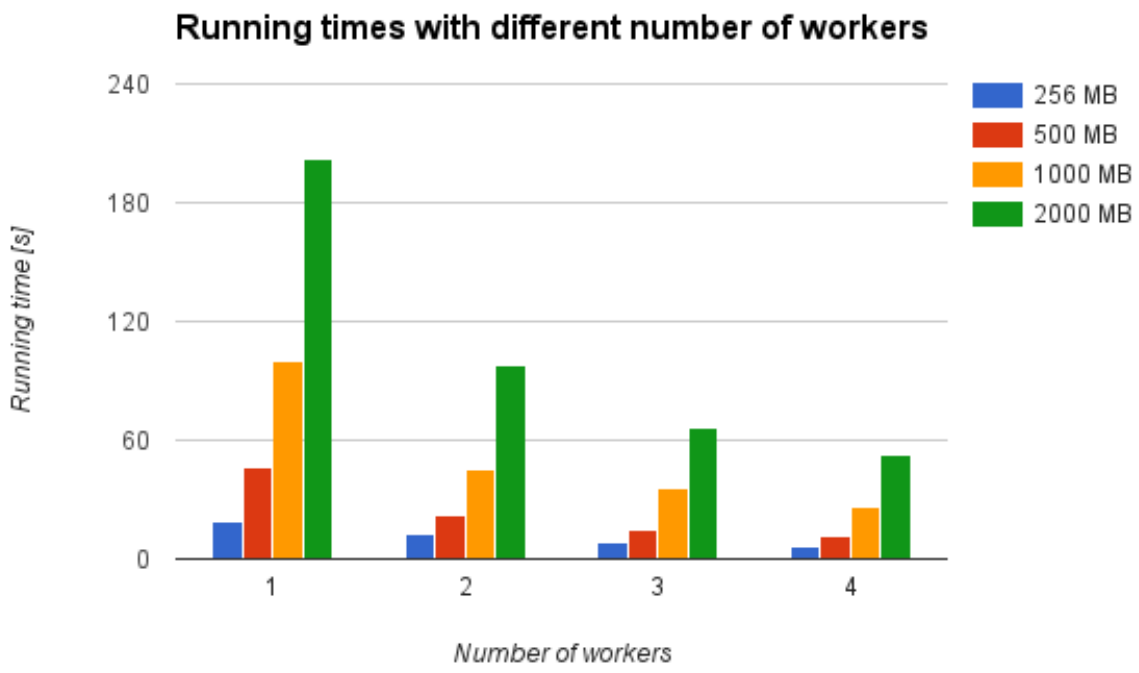

Figure 4: The runtime of each job with different number of available workers.

As you can see in the figure, the running time decreases a lot when more worker nodes are added.

Fault Tolerance

Our requirements state that our system should be able to handle at most one node down of each kind at the same time. Furthermore, when a worker node dies while processing a subtask, this subtask should not be forgotten.

Worker Nodes

When we manually kill a worker while processing a task, we see that RabbitMQ notices this, and automatically requeues the corresponding subtask. An other available worker can pick this task at a later time.

Data Nodes

All datasets are stored on at least two data nodes. We manually shutdown one data node. When the worker receives the location of a dataset, but notices that this data node is not available, the worker notifies the manager, which in turn will mark the data node as unavailable and return the alternate data node.

Managers

Unfortunately we did not have time to properly test the fault tolerance of the managers. We could not find the time to properly setup and configure pgPool-II for PostgreSQL replication, and a RabbitMQ cluster which shares the queues.

But any client wanting to connect to a manager, tries connect to a random IP selected from a configuration file where all manager IP addresses are stored, until one responds. So when multiple managers are available, it should not be a problem when one of them goes down.

Discussion

We have built a quite complex distributed system:

- A simple distributed file system for storing datasets
- A computational grid for running bioinformatics software
- A custom communication protocol between all nodes

The size of this project plus a few uncertainties at the beginning of this project (what will be our project exactly? Focus on which bioinformatics software package?) are a few reasons this project took a bit more time than expected. As a result we unfortunately could not do all experiments we would have wanted to do.

Scalability

We think our system scales well: it's easy to add a new data node or computational node. Data nodes need to register with the manager, computational nodes do not need to do anything: just pick a subtask from the job queue.

One concern for the scalability could be the consistency management of all managers, the data needs to be replicated on each manager. We do not think this will become an issue very quickly, because most actions on the manager are read operations (where is this dataset located?), and if there is a write operation, it is often a small one (only metadata). Furthermore, we think it is not likely that one would need more than three managers, and therefore the replication slowdown is probably relatively small. This should be confirmed with further experiments.

Performance

As seen in the real world experiment the performance is quite good. With multiple workers a large multiple sequence alignment is only a matter of minutes. There are however a few concerns: the preprocessing step can take a lot of time, and although the resulting byte offsets can be cached, this is not really convenient.

Furthermore, we would like to recommend more experiments with more workers and multiple managers. More workers means more result datasets written to the data nodes (and the manager for metadata), which could hurt the performance a bit, especially when you have multiple managers (and therefore the required replication).

Fault Tolerance

We think our system meets our fault tolerance requirements quite well. We ensure there is no single point of failure, and the system can handle one node down of each kind at the same time. Furthermore, when a worker dies while processing a subtask, this subtask is automatically requeued. This indirectly moves this task eventually to another available worker.

Some improvements could be adding error correcting codes to each dataset, to fix the data in case of a partial harddisk failure.

Conclusion

This report discusses the design of a distributed system for running bioinformatics software. In this report we focus on running MAFFT in a distributed manner. The system has three different nodes: manager nodes, data nodes and computational worker nodes, which means our system has both a simple distributed file system and a computational grid.

Our system is fault tolerant: the system is guaranteed to function correctly when at most one node of each kind is down. Running an experiment with a real world dataset on Google Compute Engine showed a significant decrease in runtime with more nodes, but more experiments are needed to properly see the overhead effects of the data transfer between worker, data and manager nodes.

The system also easily scales, and it is almost no effort to add another worker or data node. One concern could be the managers: they need to replicate and share the metadata stored in a PostgreSQL databases and the queues in the RabbitMQ server. We think this will not become an issue quickly because the write operations are small and quick, and more than three managers is not likely to happen. The slowdown due to replication will therefore probably be small.

All our must-have requirements have therefore been met.

References

[decap2015halvade] Decap, D., Reumers, J., Herzeel, C., Costanza, P., & Fostier, J. (2015). Halvade: scalable sequence analysis with MapReduce. *Bioinformatics*, 31(15), 2482-2488.

[pgpool] pgPool-II, a PostgreSQL middleware. Available: http://pgpool.net

[rabbitmq] RabbitMQ: robust messaging for applications. Available: https://rabbitmq.com

[mafft] Katoh, Kazutaka, and Daron M. Standley. "MAFFT multiple sequence alignment software version 7: improvements in performance and usability." *Molecular biology and evolution* 30.4 (2013): 772-780.

[uniprot] UniProt Consortium. "The universal protein resource (UniProt)." *Nucleic acids research* 36.suppl 1 (2008): D190-D195.