Project CS
Uppsala University

LoPEC
Low Power Erlang-based Cluster

Product Report

Fredrik Andersson
Axel Andrén
Niclas Axelsson
Fabian Bergström
Björn Dahlman
Christofer Ferm
Henrik Nordh
Vasilij Savin
Gustav Simonsson
Henrik Thalin

January 14, 2010

Abstract

This document describes the result produced by ten Computer Science students at Uppsala University. The goal was to develop a energy-efficient cluster entirely in Erlang that could utilize the processing power of GPUs. We have developed a heterogeneous map-reduce framework that runs arbitrary code.

Contents

Nomenclature

CUDA (Compute Unified Device Architecture), a parallel computig framework for NVIDIA devices, very similar to OpenCL

Embarrassingly parallel problems Problems for which little or no effort is required to separate the problem into a number of parallel tasks.

FUSE (Filesystem in Userspace), kernel module for Unix-like operating systems that enables non-privileged users to create file systems

GPU Graphical Processing Unit

GUI Graphical User Interface

NFS Network File System, a centralized network storage

OTP Open Telecom Platform

OpenCL Open Computing Language

PVFS Parallel Virtual File System. Distributed storage, transparent to the file system

SDK Software Development Kit

Chapter 1

Introduction

The goal of the LoPEC project was to create a General Purpose ***GPU*** cluster system using the Erlang programming language. It was planned to run on Mac Mini computers, due to their cheap price and low power consumption compared to other hardware at the time of the project. Hence the acronym, Low-Power Erlang-based Cluster. By also utilizing ***OpenCL*** (*See sec. 1.1.3*), a new standard for heterogeneous computing, we would be able to perform parallel computing on both CPUs and GPUs of nodes in our cluster environment. A custom control system was required to divide and distribute a computational job to multiple nodes, monitoring the computation, and merging the intermediate results of the sub-problems.

1.1 Languages and frameworks

1.1.1 Erlang

Erlang [1] is a programming language developed by Ericsson that recently has seen a big surge in popularity, mostly because it allows one to develop fault-tolerant, concurrent and distributed systems easier than other programming languages. Erlang was used to write the cluster control and distribution system. It has been a very convenient language to write this type of application in, due to its process-driven design and its convenient and mostly transparent way of distributing work over a network.

1.1.2 OTP

OTP is a framework included in Erlang that extends Erlang with many features such as supervision trees and abstraction of standard ways of writing code by so called behaviours. All code written in the project is OTP compliant. This made the project scale very well, and it helped us immensely,

[1] `http://erlang.org`

together with the built-in tools for working with OTP (e.g. Appmon). It allowed us to write less boilerplate code, focus on delivering functionality and simplify deployment.

1.1.3 OpenCL

Most of our test programs for the cluster were written in OpenCL[2] which is a framework for writing code that runs on heterogeneous platforms consisting of different computing devices, for example CPUs and GPUs. The main reason to pick OpenCL was its capability to utilize GPUs for computations since they are more suitable for some computations compared to CPUs. OpenCL comes with a built-in language based on C99 for writing kernels that run on the different computing devices. We have tested different ***SDK***s from NVIDIA, AMD and Apple. In the beginning of the project NVIDIA's SDK was not available for Linux and they had no OpenCL drivers for the graphics cards. Therefore there was no alternative to AMD's OpenCL SDK. Since our workstations did not have ATI graphics cards, OpenCL programs were run only on CPUs, which was enough to get the hang of the language. About a month after the project had begun, the NVIDIA SDK and drivers were released for Linux, so we could run OpenCL programs on the graphics cards as well. We also have had access to two Mac Mini Computers during the project which ran their own implementation of OpenCL. Minor tweaks were made to source code to make it work on both NVIDIA and Apple machines.

1.1.4 Nitrogen

Nitrogen[3] is a web framework that was used to create the ***GUI*** for our cluster. It has helped us to make the UI simple but powerful. The user is able to add jobs, lookup stats, etc. with ease.

1.1.5 Distributed filesystems

Our initial design was based on a shared filesystem that is accessible by all cluster nodes. We used ***NFS*** in the beginning, because it was easy to setup and use. It worked pretty well on a small scale but became a real bottleneck when more than just a few nodes worked simultaneously, because of the I/O limitations of the single hard drive and excess amount of network communication traffic going in and out from one server. Later we tested our cluster with ***PVFS***[4] which relieved the system of the stress of only using a central storage.

[2]`http://www.khronos.org/opencl`
[3]`http://nitrogen-erlang.tumblr.com`
[4]`http://www.pvfs.org`

1.1.6 Riak

Riak[5] is a distributed key-value store. It is written completely in Erlang, so it is easily accessible from Erlang applications. Riak was chosen as the storage solution alternative to the existing filesystem. The main reason to pick Riak was the capability to store all data in memory, which allows fast data access and transparent replication. Each physical node can thus act as both a computational node and a storage node.

It was very easy to setup and run Riak and the learning curve for the simple requirements we had in the project were quite shallow.

[5]`http://riak.basho.com`

Chapter 2

Architecture

2.1 Overview

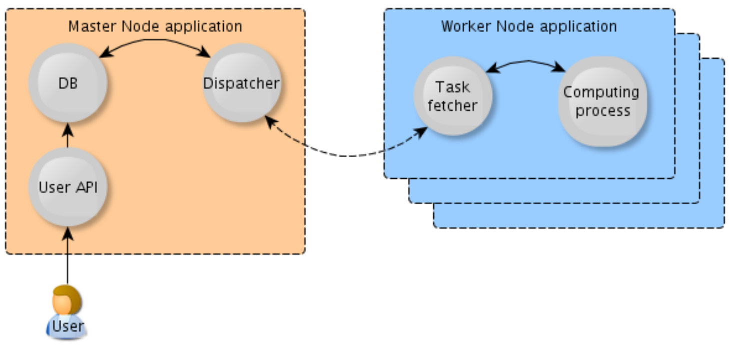


Figure 2.1: Architecture Overview

The LoPEC cluster consists of one master nodes, that distribute work to numerous slave nodes. Work is divided according to the map-reduce implementation the user has written for his program.

2.2 Map-reduce

Map-reduce is a framework developed by Google for processing problems with large amounts of data. It assumes there is a large number of computers working together as a cluster to solve the problem.

The advantage of MapReduce is that it allows for distributed processing of the map and reduction operations. Provided that each mapping operation is independent, all maps and reduces can be performed in parallel. The drawback of this is that you can only handle ***Embarrassingly parallel problems***.

Our implementation of MapReduce

Split step: The input data is chopped up into smaller pieces according to a split function provided by the user's program. One map task is created for each created piece.

Map step: The pieces of the input are taken on by the slave nodes, and they apply the map function given by the user's program on the input data. Each map task can generate several input files for the reduce step, but several maps can generate input to one file as well.

Reduce step: This step will "merge" the data that was created by the map step.

Finalize step: A step unique to our cluster; anything that is left to do is done here, like merging of data, moving an output file, etc.

2.3 Master node

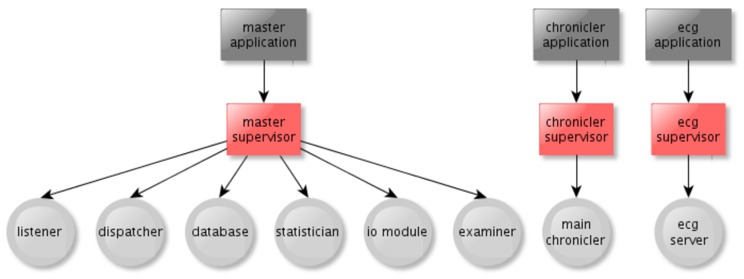


Figure 2.2: Master Node Architecture

The modules in the master node handle calls from users and slave nodes, with the User API being handled by the Listener module, and the slave node communication by the Dispatcher module. These two will redirect calls to various backends, like the ECG module, which keeps track of whether nodes are alive or not, or the database module, which stores metadata concerning jobs and tasks.

2.3.1 The Database

The database is currently designed with the map-reduce structure in mind, and runs Mnesia for the transactional backend. It stores metadata for all of the jobs running on the cluster, as well as the metadata of all the subtasks of these jobs.

Input and output data is handled by the IO_module (see 2.5), so when a slave node receives a task from the database, it will contain pointers to the program and the data the node should process, rather than the data itself.

All jobs and tasks are assigned an unique id. This id is actually nothing more than a time stamp, based on Erlang's *now()*-function, but it is sufficient for our purposes.

Job data is put in one table while all the subtasks are moved between twelve different tables, depending on their type and current state. These twelve tables are named *type_state*, where type = [*split, map, reduce, finalize*], and state = [*free, assigned, done*]. An *assigned* task of type *reduce* would therefore belong in the *reduce_assigned* task table. These all refer to what part of the map-reduce algorithm the task handles.

Although the jobs have states similar to the tasks, they always stay in the same table. States a job can have are: *free, stopped, paused*, and *no_tasks.* Each job also has a flag attached to it called *is_bg* indicating whether it is a background job or not. Tasks of non-background jobs are always done before tasks of any background job.

The database also contains separate tables for the relations between jobs, tasks, nodes and users, called *assigned_tasks* and *task_relations.* These tables are mostly used by internal queries to find specific items from the database.

Our API performs all operations needed to maintain a coherent structural workflow in the cluster. These operations include selecting which tasks to return to available nodes, scheduling which jobs to run and handling users. It may be worth noting that the database server currently runs on only one master node even though Mnesia provides support for distributed databases. It should not be too difficult to extend it to support multiple master nodes.

When a job is completed the database clears all metadata associated with the job, to prevent the tables from growing forever and causing issues.

2.3.2 ECG - The cluster heartbeat monitor

The ECG [1] module monitors the slave nodes in the cluster for crashes. Whenever a new node asks the dispatcher for a task, the ECG is notified and adds that node to its list of nodes to monitor. When a slave node crashes, the ECG will get a *nodedown* message and notify the master node about node going down, so that any tasks the node was working on can be unassigned and be completed by some other node.

2.3.3 Listener

The listener module acts as an interface to the cluster. It is mostly a wrapper for some calls to the database and the dispatcher to create and control jobs.

2.3.4 Examiner

The examiner module keeps track of tasks that are created and assigned. This way, it can track the progress of jobs and answer queries about progress. This module was created because we thought letting the database handle queries about progress would be too much of a performance bottleneck. In hindsight, distributing the state of jobs lead to some complications and race conditions and the database is now handling the progress queries from the web interface.

2.3.5 Dispatcher

The dispatcher is a module bridging the database to the rest of the cluster. It thus acts as a second layer for many different database activities, such as adding new jobs, creating new tasks when some tasks have finished and notifying processes when tasks fail. When slave nodes request more work to be done, their requests are handled by the dispatcher on the master node.

By using the dispatcher as an additional layer there is no need to connect the slave nodes to all different database nodes in case the database would be distributed.

2.4 Slave Node

An idle slave node in the cluster will attempt to pull a task from the master node at regular intervals via the Task fetcher module. When a task is received, the Computing process module will spawn a new OS process to run the task.

[1]as in electrocardiograph

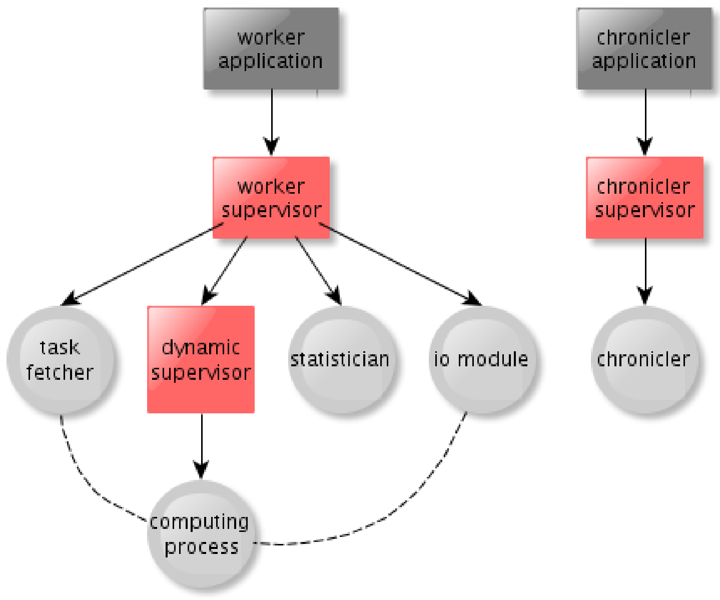


Figure 2.3: Slave Node Architecture

2.4.1 Task Fetcher

The Task fetcher module is responsible for acquiring new tasks from the master node and adding newly produced tasks from the computation. It does this by polling the Dispatcher module on the master node for work. When a task is done the Task fetcher collects and calculates all the necessary statistics about the task and reports it to the Statistician. It does this with two helper modules, net monitor and power check. New tasks are reported to the master as the user program reports them to the slave.

2.4.2 Computing Process

When the Task fetcher on the slave nodes has received a task, it starts a computing process. The Computing Process acts as a wrapper for the user program. The computing process spawns the application and communicates with it on the standard input and output streams, fetching input data when the application requests it and saving output data when it is given. It also tracks processes that the slave spawns so that they can all be killed if the slave node is told to stop working on the task. Since it is spawned on the fly with different arguments every time it is supervised by a dynamic supervisor to make everything OTP compliant.

2.5 Common Modules

Some modules are used by both the master and slaves, like the Statistician which collects various statistics from the cluster, or the Chronicler, our logging system.

2.5.1 IO Module

The IO Module works as a frontend for fetching and retrieving data. It is a key value store that requires a backend to function. There are two predefined backends, one for Riak and one for any Erlang-compliant filesystems. These filesystems have to be distributed in some manner though, like NFS for example.

The cluster provides different storage backends that take care of the results. Data is stored using a two-level key, the first part of the key is called "bucket", and is used to differentiate input for unique tasks. The second part of the key is called "key" and together with the bucket, it defines one unique input data entry to a task. A task can have several input data entries. The cluster will ensure that the given bucket does not interfere with buckets in other jobs, or other tasks in the same job.

A backend only needs to implement three callbacks;

init(*Args*) Should set up everything necessary, *Args* is backend-specific stuff.

put(*Bucket, Key, Value*) Should put *Value* in a place identified by *Bucket, Key.*

get(*Bucket, Key*) Should return the value corresponding to *Bucket, Key.*

2.5.2 Chronicler - The System logger

The Chronicler system is based on two parts, the slave chronicler is responsible for collecting logs and sending them to the main chronicler and the main chronicler just receives the logs and saves them. The system will log to a local file and can as an alternative also log to the screen.

2.5.2.1 Slave Chronicler

The slave chronicler is responsible for collecting and sending the log messages from the local system to the master chronicler, it will, however, not fail if the Master Chronicler is not present and will still produce a local log file to allow for better debugging of a running system. There are five available logging levels, see section B.1.10.2 for more info.

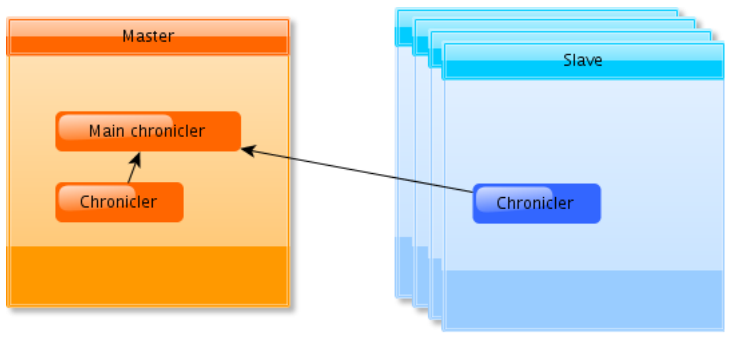


Figure 2.4: Chronicler Architecture

2.5.2.2 Main Chronicler

The Main Chronicler has a local hash table to store and allow lookups of log messages. This was done so the user interface could look up log messages and it allows for filtering of message by either a direct API call for some queries or custom queries to allow for a dynamic system. If the Main Chronicler is present it will produce a log for the entire system (all messages it receives) and write this file to disk for further reading. Nothing is saved in the hash table between restarts or crashes. Currently the hash table is not cleaned up periodically, although support for this is planned.

2.5.3 Statistician

Information on power usage, network traffic, disk usage etc. in the cluster is stored by the Statistician module. A statistician process runs on every node, periodically checking the network traffic, memory usage, and disk usage. Power consuption is at the moment estimated using the processor load and numbers from external measurements on the computers done prior. The statistician also tracks how much time is spent working on each task, and how many tasks have been restarted.

The statistics are then sent periodically from the slave nodes to the master node so that it can be queried for the collected statistics. Queries can be made on e.g. statistics for individual jobs, nodes, and users.

2.6 Workflow

2.6.1 The Workflow in the Cluster

The user adds a job and uploads its input data through the web interface. The master node creates the initial split task for the job. The slave nodes request tasks, work on them and submit request to create new tasks as needed. When new tasks are added, they become available for nodes to work on. Reduce tasks can only be assigned when all map tasks related to the job are completed, while finalize tasks only get assigned when all reduce tasks related to the job are done. The web interface shows job progress so the user knows when a job is done and can collect the results.

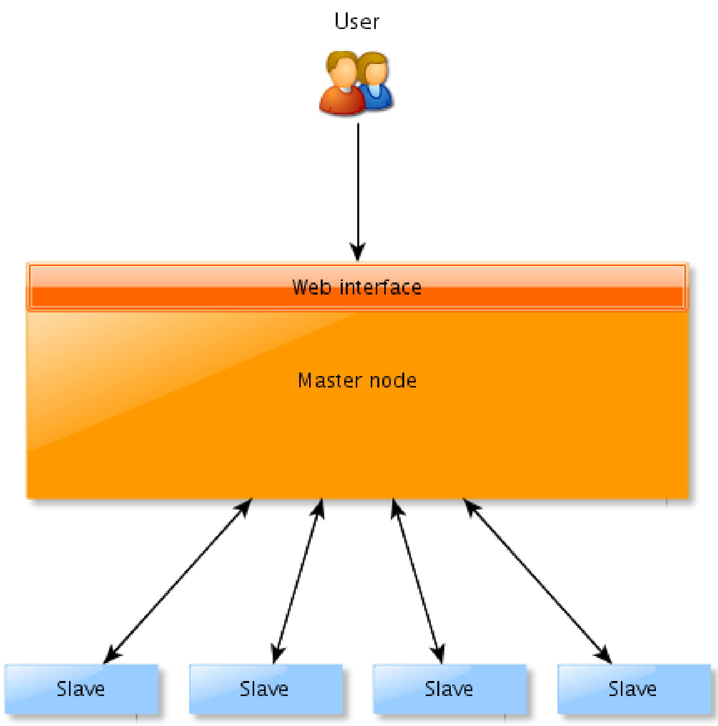


Figure 2.5: Workflow Overview

2.6.2 The Workflow in the Master Node

A job is added from the web interface (Step 1). This interface talks with the listener (Step 2) which acts as an interface to the cluster. When jobs are added via Listener module, the listener calls the dispatcher (Step 3) which tells the database (Step 4) to insert a new job and then creates the first split task.

The dispatcher is the interface between the database and the slave nodes, it waits for task requests (Step 5) from the slaves and distributes tasks from the database on request (Step 6). When the slaves complete tasks, they report back to the dispatcher, which tells the database about the task that was completed and whether new tasks need to be created. The dispatcher then tells the database to create any new tasks, if needed. The dispatcher keeps the examiner informed on task creations, assignments, completions and failures.

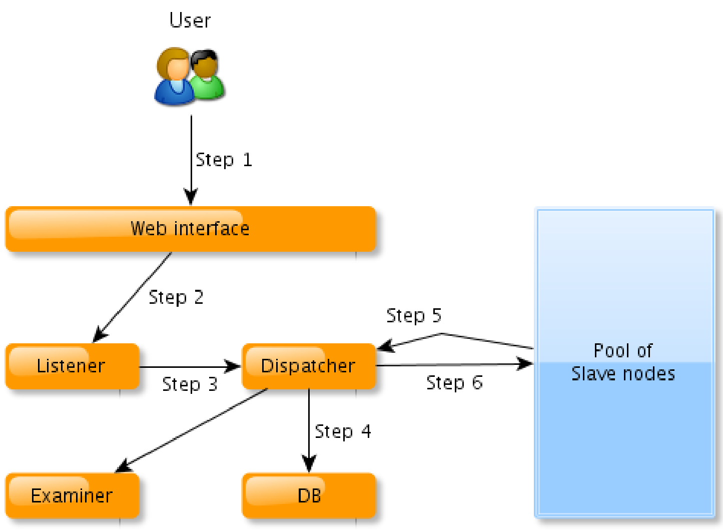


Figure 2.6: Master Node workflow

2.6.3 The Workflow in the Slave Nodes

The task fetcher requests tasks from the master node. When a task is received, a computing process, wrapping the user application, is started. The computing process gets input data from the IO module to the application slave. When the user application completed its work and wants to store results, the computing process stores them using the IO module and informs the master node about the completed task and a new task is requested by the Task fetcher. If the application crashes or sends an error message, the task is reported as failed to the master.

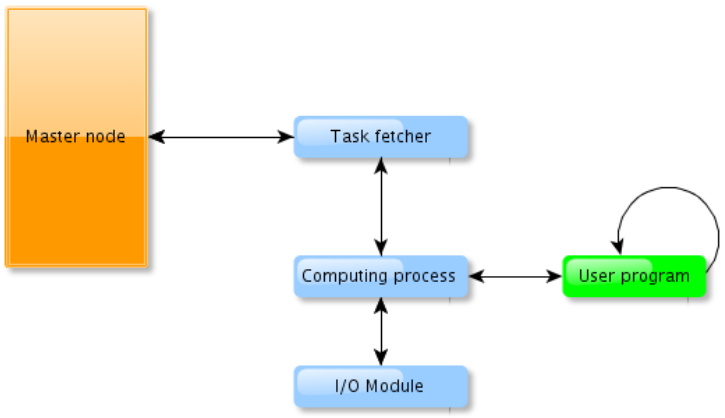


Figure 2.7: Slave Node workflow

Chapter 3

Example Programs

3.1 Example Programs

3.1.1 RayTracer

The raytracer is written using C++ and OpenCL. It is fairly basic and can as of now only render spheres of different (as of now predefined) colors from a simple, user-defined scene file. This scene file, together with the resolution of the image, and how large a part of the scene it should do (i.e. how many rows of the final image it should calculate) forms the input for the program. This is very useful when the raytracer is run on the cluster as each node can do a different part of the image. The tracing is done by an OpenCL kernel. The kernel is run once for each pixel and it can be run for several pixels in parallel, but at most run calculations for one row of an image in parallel depending on the image size. The output format is PPM (Portable Pixmap) and is printed to standard out as ASCII.

3.1.2 Image Filters

We have one C++ application that we have used as the test base for different image filters. Among the filters we have implemented are sharpness, gaussian blur, and an emboss filter. We also have an OpenCL version which at the moment only handles the sharpness filter. The filters all work by applying a weighted filter matrix on each pixel and those surrounding it. The image format we use is PPM, the reason for this is it is a simple file format that is easy to read and write, however it is not optimal, since PPM files tend to be very large. For the implementation of filter applications we used a tutorial at GameDev[1] as our base.

If one would like to implement any of the other filters in OpenCL, it is possible to use the sharpness filter application as a reference implementation. It should be fairly straightforward to modify it to do any of the other filters.

[1]The GameDev Tutorial is available at http://www.gamedev.net/reference/articles/article1994.asp

3.1.3 Audio Filters

Our main audio filter is an echo filter, which adds a single echo to a .wav file. It's a very simple implementation in that the algorithm works by copying the entire audio file, reducing its amplitude (volume) and then overlaying it with an offset on top of the original file. At the moment it is only implemented in C++ and does not use OpenCL, the plan was to port it to OpenCL as well but due to time constraints we decided to leave it be as is.

3.1.4 Wordcount

The simplest of the example applications, the wordcount is implemented in Ruby and counts the occurrences of all the words in a .txt file given as input argument. The program outputs its results to a .txt file.

3.1.5 How To Run

For instructions how to run these sample programs, you can refer to the user manual in Appendix B

3.2 Languages

The section covers the languages that were used to develop the example programs for the system.

3.2.1 C++

The "Host" programs for our OpenCL code has been written in C++[2]. C++ was used as the OpenCL SDK is written to interface with C++. The purpose of these programs is to allocate memory for the data to compute and assign where and how the data should be processed.

3.2.2 Ruby

All of our programs has a Ruby[3] script which communicates with Erlang. In a few cases it is the main program itself but mostly it just starts a C++ application. This was just something we chose because it seemed convenient and fast so there is no kind of requirement that the external application has to be a Ruby application.

[2]http://www.cplusplus.com

[3]http://www.ruby-lang.org/en

Chapter 4

Results

4.1 Cluster Software

We have produced a heterogeneous cluster for general computation of embarrassingly parallel problems. The computation of a job is distributed, and performed in parallel over a set of Erlang nodes residing on multiple physical nodes. These nodes can be Mac Minis, in line with the original aim of the project in terms of low price and low power consumption, or regular PCs or other forms of hardware. The cluster is not limited to any specific hardware, though, it can in theory run on everything Erlang runs on. The distributed nature of the cluster pertains to nodes being mostly autonomous, requesting computational tasks when they have resources available.

The cluster works in parallel as tasks are computed simultaneously. However, as jobs must be divided into tasks prior to calculation, and merged into a result after, the cluster is also sequential in part, with one or more slave nodes computing sequential tasks. Nevertheless, most of the nodes are performing the actual calculations in parallel, and can run on any network, local, external or across the Internet. Running the cluster on a closed, local network is recommended, as the cluster doesn't implement any form of security or access-control.

The cluster is agnostic to the implementation language of the user program, thus limits on hardware are in the strict sense only related to if the program is compiled for the preferred OS. The program only has to conform to our protocol. More practical limits on the choice of hardware to run the cluster on arise if the user chooses to make use of OpenCL, as OpenCL is limited in the number of devices (i.e. graphic cards) that support it.

Nodes can be added to the cluster as it is executing jobs, and they will immediately pick up the next free task. When nodes for some reason leave the cluster, any task they were working on will be marked as free again, and picked up by some other node later. Programs can be patched as the cluster is executing them, though we do not recommend doing so.

The cluster has support for different types of storage solutions, we have written backends for as riak and file systems, but more can be added. It comes with a few example applications, such as a ray tracer and an image filter – both OpenCL programs – and a word count program written in ruby. There is also a graphical user interface supporting addition of jobs, management of users, viewing of statistics and presentation of logs.

4.2 Performance Testing

Due to unfortunate planning, we have not had much time to benchmark our cluster. We did make time for some performance testing half-way through the project, however. The system has since had some major changes done to it, so the results presented here do not indicate the current status of the cluster.

The tests were carried out by running a ray tracer application we have written (See 3.1.1), and comparing the time taken to render an image. We compared the time taken when using one, two, four, and eight nodes. We also tried rendering images of different sizes for comparison. The images were square, with a side of 1024, 2048, 4096, and 8192 pixels. We rendered each image size ten times sequentially for each number of nodes, and compared the mean time taken from when the split task was started to when the finalize task was done.

From these numbers we derived the factor of the speedup when adding nodes to the cluster as the ratio of the time taken on one node to the time taken on two, four, and eight nodes.

We also measured the speedup factor when rendering eight images concurrently on the cluster.

As can be seen, the speedup factor is higher in this case. The reason for this is that putting the different splitted image pieces together after they have been rendered is a sequential operation, so only one node can do it. While one node is putting pieces together, the others can start rendering the next image.

As was mentioned above, these results are for an older incarnation of our cluster and does not reflect the current situation. Since the performance tests, we have substituted the storage of the cluster for an abstract storage system with interchangeable backends. The storage overhaul led to breaking compatibility with the raytracer and we have not yet had time to mend it. We did do some informal performance measurements by running the pathologically task-creating word count application. These informal measurements suggested that, using the riak backend for the abstract storage, the cluster worked faster.

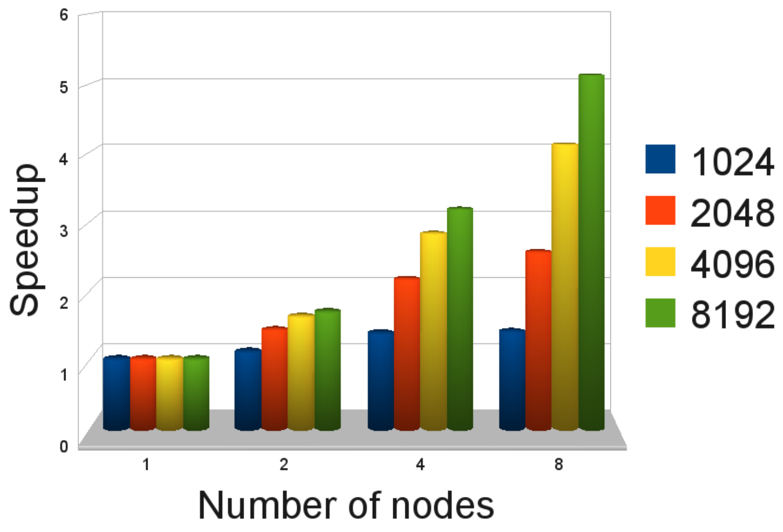


Table 4.1: Seconds taken to render images

Image side	One node	Two nodes	Four nodes	Eight nodes
1024	8.74	7.88	6.3	6.21
2048	23.58	16.43	10.9	9.24
4096	81.59	50.11	28.93	19.89
8192	311.21	183.56	98.16	60.84

Table 4.2: Speedup factor when using more nodes

Image side	One node	Two nodes	Four nodes	Eight nodes
1024	1	1.11	1.39	1.41
2048	1	1.43	2.16	2.55
4096	1	1.63	2.82	4.1
8192	1	1.7	3.17	5.11

Table 4.3: Speedup factor when rendering eight concurrent images

Image side	One node	Eight nodes
4096	1	5.43
8192	1	5.71

Chapter 5

Problems

5.1 NVIDIA OpenCL Compiler

OpenCL code is most often compiled during runtime just as it is about to be executed. This is due to the fact that the code is required to be compiled for a specific GPU. This, however makes it harder to find warnings and errors. This is especially the case for NVidias compiler which adds an extra step to the compilation in that it first translates the OpenCL code to ***CUDA*** code. This is where really tricky problems are introduced, if the CUDA code fails to compile you will get error messages that are very hard to understand and almost impossible to find the cause of. These errors might not even exist in the original OpenCL code, having been introduced in the translation step.

5.2 PVFS

When we were trying to get PVFS up and running on our computers the ***FUSE*** module refused to compile. This turned out to be the PVFS code missing a semicolon. A minor fault in the code which somewhat delayed our project.

5.3 Power Monitoring

We wanted to use built-in sensors to monitor power consumption in the Mac Minis but they were too few and required third-party software to work so instead we put a power meter on the wall socket to see how much a Mac Mini spent during idle and high load. We have then used these values to estimate the power consumption on a task basis.

5.4 OTP Documentation

When we had to familiarize ourselves with Erlang and OTP we obviously had to read the existing documentation. However, as it turned out this documentation was not entirely exhaustive or fully descriptive. In some cases you would have to spend several hours trying to find the exact usage description of some built-in-function.

Chapter 6

Known Issues

6.1 Storage

NFS has the problem of using just one harddrive for all I/O, making the read/write speed of the drive a bottleneck. PVFS and Riak help alleviate this problem to some degree, but PVFS is difficult to set up, and Riak gets slowed down by the large amount of temporary files created during our computations.

6.2 Logging

There is currently no way to find out which levels are active for chronicler, other than deriving it from from what messages are being printed.

We never remove old messages from the table of messages in the Master Chronicler and there is no functionality to do so.

6.3 Modules

Our statistician process receives its updates from each node once per second, which has a side-effect; When a job is finished and its stats are dumped to file, an update from a node may theoretically be delayed so long it'll arrive after the dump. This is unlikely to occur unless the cluster is distributed over the internet.

The values given for network load may not be accurate as it measures total traffic, not just the traffic generated by the cluster software.

The statistics for power usage is an interpolation based on values we measured manually beforehand at both high and low processor load. This is not an accurate method of measuring power consumption. To do this accurate hardware sensors is needed.

6.4 Example Programs

6.4.1 The Raytracer

The raytracer produces some pixels that are off, due to some failed hit check. This is more noticeable when producing smaller images.

6.4.2 Audio filters

The echo filter currently introduces some background noise to the output file. We have not yet determined the cause of this.

Chapter 7

Future Work

7.1 Implement Problem-dependent Storage

As it is now, our cluster can not switch storage back-end on-the-fly. We believe it could be interesting if a user could pick the back-end the cluster would use for that job, in case one type is more efficient for a specific job. For example, imagine there are two jobs running in the cluster, one of them handling large files (like a video encoder program), and the other numerous small files. Then it could be a good idea to let the video encoder use a disk-based storage and let the other one use a RAM-based storage since that would not flood the memory on the nodes.

7.2 Chronicler Maintenance

The hash table that keep tracks of the logs in the Master Chronicler is never cleaned and will grow indefinitely. It needs to be cleaned up, either at regular intervals or when it reaches some threshold.

7.3 Internode Communication

The cluster currently only supports embarrassingly parallel problems, as there is no internode communication. Functionality to handle such problems should be added to the cluster, though it will require a substantial rewrite.

7.4 Statistician

7.4.1 Additional Features

It might be interesting to measure uptime for a specific node, or the whole cluster. This could be the total uptime of the computer or the uptime of the LoPEC applications.

7.4.2 Code Split Up

Currently the same statistician is used in both master and slave, but parts of the code are only used on one of the two. It would make our code more clean if we separated the module into three different modules, one for the master code, one for the slave code, and one with the shared code.

7.5 Remove Pulling

We currently use a pull-system, where the slaves ask the master for work to do with regular intervals. Modifying this behaviour so the master node remembers the nodes that have requested work would remove the need for continuous pulling and thus reduce some network traffic.

7.6 Improving the GUI

The web interface currently does not allow control over the entire cluster, just some basic administration abilities like controlling jobs and managing users. We could add more controls to the web interface so that the user does not need to use the Erlang shell or edit configuration files by hand to get the desired results. OTP has support for such abilities, but we would probably need to include user information with each task rather than just each job.

7.7 Master Failover

Mnesia supports distribution across several nodes and is critical for implementing failover on the master node. With this in mind the master node info could go to a standby computer that can take over the responsibility from the old master if it failed somehow.

Chapter 8

Conclusion

In the initial design stages of the project concepts related to the design were poorly defined in terms of how things were to function. While we saw the possiblity of going forth with more advanced design structures, we also realized more advanced designs would lead to a more difficult development phase of the product.

We thus set out for a pretty simple solution and have been improving it ever since our first design discussions of the requirements set by the customer. The design ended up being quite simple at the highest level, but with some dependencies between different modules not expected in early design phases.

The cluster is currently only able to solve embarrassingly parallel problems. If implemented, internode communication would allow new types of programs to be run on the cluster, such as synchronous or loosely synchronous algorithms. While this would definitely have been a great feature, it would have added greatly to the complexity of the cluster, and it was regarded early in the design discussions too great a feat to fit inside the scope of the course.

Another conclusion reached is the interesting aspects of having a failover for the master node. Currently not implemented, master failover would prove vital in a running large-scale system employing the current cluster setup with the addition of several master nodes.

We also lack a comparison between our cluster, taking the specific hardware in terms of the mac minis into account, and other clusters. Compared to other contemporary clusters, it is reasonable to expect our cluster not to be exceptionally interesting if we confine ourselves to total computational power and scalability. However, we hope that when looking at hardware cost per theoretical giga-flop performance, our cluster should provide an interesting way to go about when choosing hardware for distributed, parallel computation.

Appendix A

Install Guide

A.1 Storage

Storage should be set up before starting the cluster. Currently we provide support for distributed filesystem (like NFS and PVFS) and riak. There should be a file "lopec.conf" in the trunk directory, describing among other things where the cluster "root" should be, and what filesystem to use. Change as necessary and move the file to "/etc/lopec.conf", the config file contains comments that describe the settings. Riak needs to be configured to use the same cookie as your erlang nodes, as well as having the long name that contains the IP address for the interface that communicates with the cluster.

A.2 Dependencies

You will also need Nitrogen for the cluster web GUI. You can get it at `http://nitrogenproject.com/`. The master node can not start without it.

A.2.1 Other Dependencies

The programs have their own dependencies. Our wordcount script for example requires Ruby, while OpenCL programs will require OpenCL-compatible hardware and drivers.

A.3 LoPEC

Setting up the cluster software on the master and slave nodes is described below.

A.3.1 Master Node

Obtain our tarball and extract it somewhere, e.g. your home folder:

```
pushd ~
tar xvf lopec.tar.gz
```

Change directory to "lopec" and make the script for starting the master:

```
pushd lopec
make master_script
```

Copy the config file to "/etc/" and edit it to suit your system, it has comments explaining the different settings.

```
cp lopec.conf /etc/lopec.conf
popd
popd
vi /etc/lopec.conf
```

A.3.2 Slave Node

Obtain our tarball and extract it somewhere, e.g. your home folder:

```
pushd ~
tar xvf lopec.tar.gz
```

Change directory to "lopec" and make the script for starting the slave:

```
pushd lopec
make slave_script
```

Copy the config file to "/etc/" and edit it to suit your system, it has comments explaining the different settings.

```
cp lopec.conf /etc/lopec.conf
popd
popd
vi /etc/lopec.conf
```

Appendix B

User Manual

B.1 User Manual Appendix

B.1.1 Starting the Master

The master is started by running the start_master boot script, which will start all the applications in the right order. If you use the riak backend, the riak node has to be started as well. The IP address of the interface communicating with the cluster needs to be exported to the environment variable "MYIP", and the riak node must be called "riak@$MYIP" for the master node to find it. Be sure to export the path to riak to the environment variable "RIAK".

To start the Riak node, run:

```
$RIAK/rel/riak/bin/riak start
```

To start erlang and boot the master, run:

```
erl -name master@MASTERS_IP_ADRESS -boot releases/master/start_master\
    -pa $RIAK/apps/riak/ebin
```

B.1.2 Web Interface

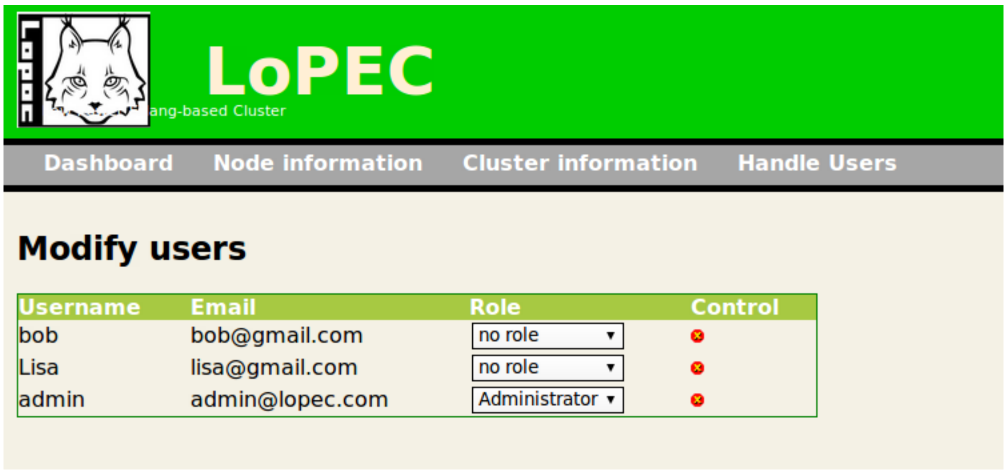
When the master is started you can visit *http://localhost:8000* to access the web interface.

If mnesia fails because tables already exist, you need to remove the disc copies of the tables first:

```
rm -fr Mnesia*
```

B.1.2.1 User Management

There are two different types of users within the web interface; *admin* and *user*.



Modify users

Username	Email	Role	Control
bob	bob@gmail.com	no role	✕
Lisa	lisa@gmail.com	no role	✕
admin	admin@lopec.com	Administrator	✕

B.1.3 Access the Cluster Through Command Line

There is no user management when using the command line since it is implemented in the web-layer, so use the command line with caution.

B.1.4 Tune the Config File

The configuration file for LoPEC is located in */etc/lopec.conf*. This file contains some tuples with one identifier and one value.

B.1.5 Starting a Slave Node

The slave is started by running the start_slave boot script, which will start all the applications in the correct order. If you use the riak backend, the riak node has to be started as well. The IP address of the interface communicating with the cluster needs to be exported to the environment variable "MYIP", and the riak node must be called "riak@$MYIP" for the slave node to find it. Be sure to export the path to riak as $RIAK.

To start the Riak node and connect it to the Riak cluster, run:

```
$RIAK/rel/riak/bin/riak start
$RIAK/rel/riak/bin/riak-admin join riak@$MASTER_IP
```

To start erlang and boot the slave, run:

```
erl -name slave@SLAVES_IP_ADRESS -boot releases/slave/start_slave\
    -pa $RIAK/apps/riak/ebin
```

Important! The name of a node must be unique.

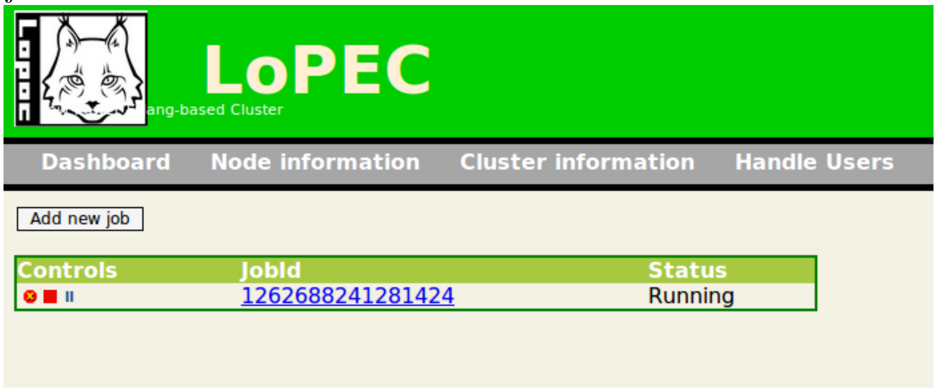
Connect the slave node to the master node:

```
(slave@$MYIP)1> net_adm:ping('$MASTER_NAME@$MASTER_IP').
```

If the connection was successful the *net_adm:ping*-function should return *pong*. If something went wrong a *pang* message will be returned.

B.1.6 Handling a Job in the Cluster

All job related informartion is located under *Dashboard* → *Jobs*.

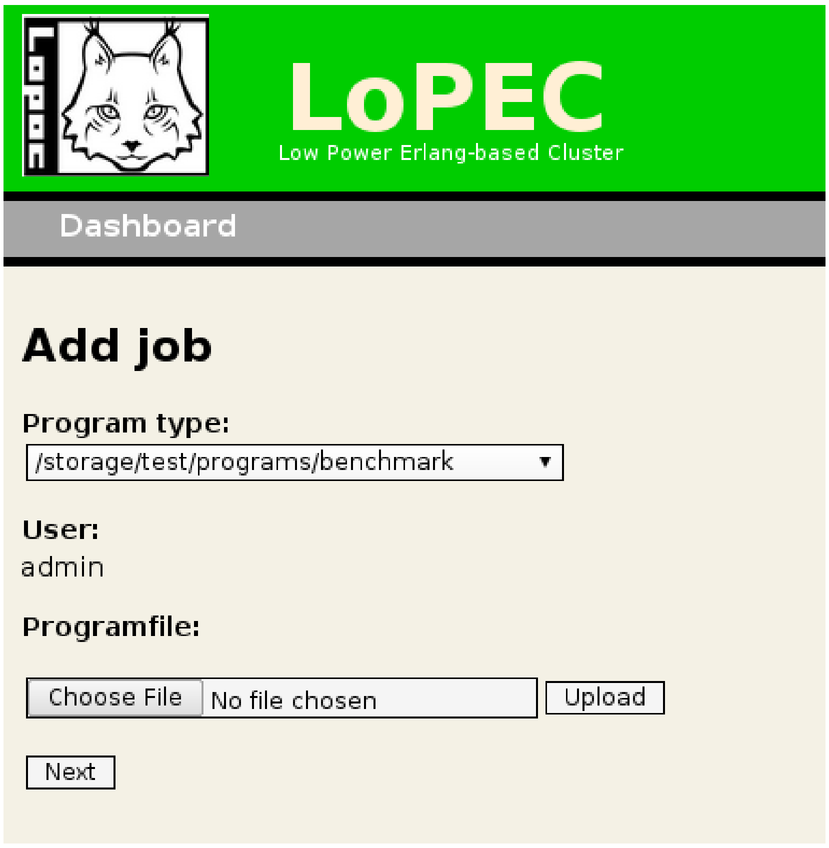


Controls	JobId	Status
	1262688241281424	Running

In the picture above one could see that there is currently one job running. Every job have some controls attached to it (on the left side of the jobid). the user can use them to stop, pause, resume and cancel a job. There's also a button of which the user can press to add a new job.

B.1.6.1 Adding a Job

To reach the add job page the user must go through *Dashboard* → *Jobs* and click on the *Add job* button.



Add job

Program type:
/storage/test/programs/benchmark

User:
admin

Programfile:

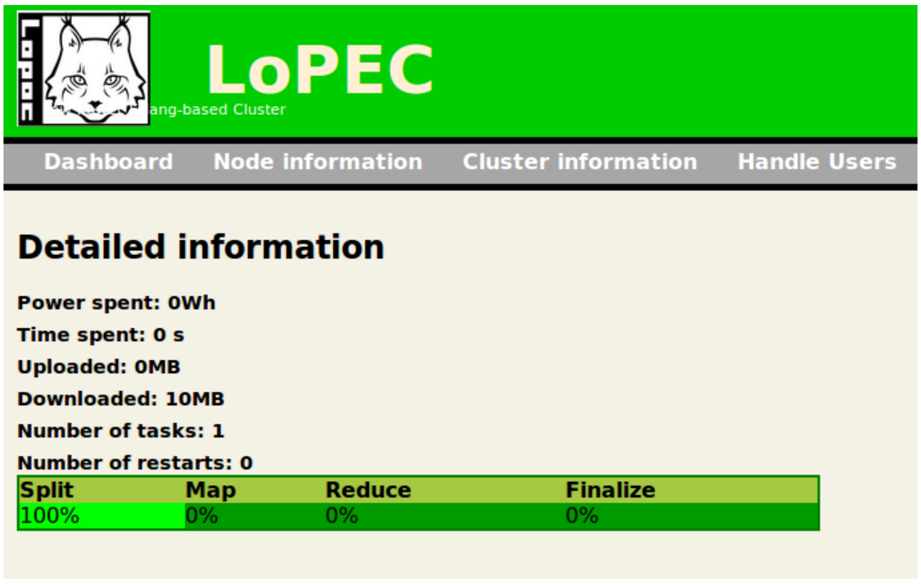
There are three different variables the user has to input before adding a new job:

- Program type - What kind of program this job should run
- Programfile - The input file to the program

When a job is added the user will be redirected to the jobs-screen.

B.1.6.2 Get Detailed Information About a Job

When looking at the job page, the user can select one of the jobs in the list to get detailed information about that specific job.



Detailed information

Power spent: 0Wh

Time spent: 0 s

Uploaded: 0MB

Downloaded: 10MB

Number of tasks: 1

Number of restarts: 0

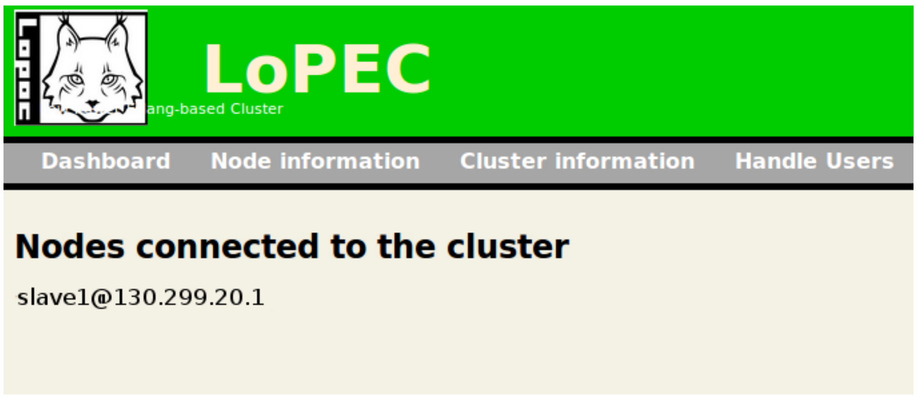
Split	Map	Reduce	Finalize
100%	0%	0%	0%

On this page the user can see progress, power consumptions and other information abut the job.

B.1.7 Getting Information About the Cluster

If the user have admin privileges there are two more items in the menu, *Node information* and *Cluster information*. In the *Node information* section one can find information about how many and which nodes that are connected to the cluster.

B.1.7.1 Node Information



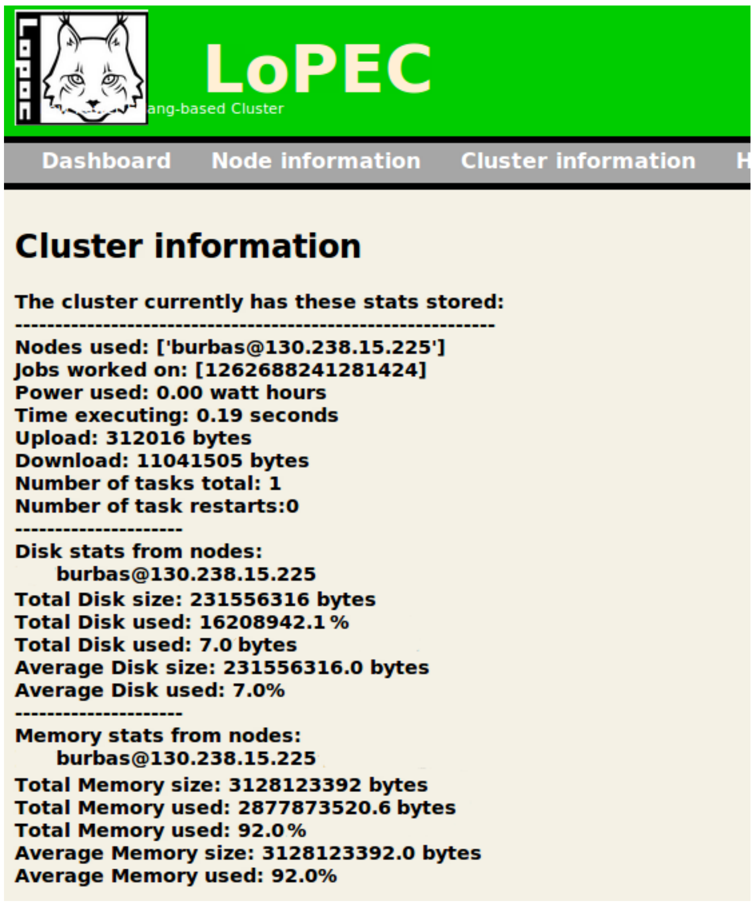
Nodes connected to the cluster

slave1@130.299.20.1

Figure B.1: Cluster has one node 'slave1' connected to the cluster

B.1.7.2 Cluster Information

Under the *Cluster information*, the user can find detailed information about the whole cluster.



```
Cluster information

The cluster currently has these stats stored:
------------------------------------------------------------
Nodes used: ['burbas@130.238.15.225']
Jobs worked on: [1262688241281424]
Power used: 0.00 watt hours
Time executing: 0.19 seconds
Upload: 312016 bytes
Download: 11041505 bytes
Number of tasks total: 1
Number of task restarts:0
--------------------
Disk stats from nodes:
     burbas@130.238.15.225
Total Disk size: 231556316 bytes
Total Disk used: 16208942.1%
Total Disk used: 7.0 bytes
Average Disk size: 231556316.0 bytes
Average Disk used: 7.0%
--------------------
Memory stats from nodes:
     burbas@130.238.15.225
Total Memory size: 3128123392 bytes
Total Memory used: 2877873520.6 bytes
Total Memory used: 92.0%
Average Memory size: 3128123392.0 bytes
Average Memory used: 92.0%
```

Figure B.2: This page shows all gathered data from the whole cluster

B.1.8 Adding a New Program to the Cluster

B.1.8.1 User protocol

The user program implements the different steps in map reduce algorithm. For each task in the job, the user application is started and the function to be executed is given in the argument vector. Input data is fed through the standard input by the cluster when the user program asks for it. Results from the different steps are written to the standard output and picked up by the cluster.

For the cluster to be able to stop and preempt jobs, OS pids must be reported for every process that is spawned during the task execution, even the process that is started first and given the task type.

All messages sent to and from the cluster will be prefixed by four bytes representing an integer with the most significant byte first (big endian) that defines the size of the message in bytes.

Input Data Input data for tasks is requested by printing the following to standard output:

```
GET_DATA
```

The cluster will print either:

```
SOME\n
<key>\n
<binary data>
```

if there is more data to send, or

```
NONE\n
```

if there is no more input data items.

Split

The split command will be invoked as follows:

```
<computing_program_name> split <pid_path> <number_of_nodes>
```

While the program is splitting, each new successful split should be reported to the standard output stream following this format:

```
NEW_MAP <bucket> <key>\n
<binary_data>
```

For each unique bucket, a map task will be started and it will be given data from every split that is in that bucket.

Map

Each map task will be started by starting:

```
<computing_program_name> map <pid_path>
```

Each map task produces data for reduce tasks. Data is stored in the cluster by writing the following to standard output:

```
NEW_REDUCE <bucket> <key>\n
<binary_data>
```

For each unique bucket, a reduce task will be started and it will be given data from every split that is in that bucket. So different map tasks can pass input to the same reduce task.

Reduce
Each reduce task will be started by running:

```
<computing_program_name> reduce <pid_path>
```

Each reduce task produces results that can be finalized if the user so desires (see below for finalizing). Data is stored in the cluster by writing the following to standard output stream:

```
NEW_RESULT <key>\n
<binary_data>
```

The reduce results will all be stored in the bucket "results" and will be unique to the finalize task.

Finalize
When all reducing is done for a job, the finalize task can be used if the results need to be collected somehow, it will be started by running:

```
<computing_program_name> finalize <pid_path>
```

If no finalizing is desired, the user program just writes the following to standard output:

```
NO_FINALIZING_PLZ
```

If finalizing is desired, just get data as per above. Input data to be worked on is given on standard input. Each input data entry is prefixed with a header saying how many bytes to expect before the next header. Final results are written to standard output according to the following format:

```
NEW_FINAL_RESULT <key>\n
<binary_data>
```

When the finalizing is done, the results will be available to the user.

User Program Logging
The computing program communicates with the cluster by printing messages to standard output, see below. In all types of tasks, the program can log whatever the user wants by printing:

```
LOG <some message>
```

and it will be shown to the user and not just internally in the cluster.

Error Handling
If something goes wrong, errors can be reported by writing the following to standard output stream:

```
ERROR <reason>
```

A task that terminates abnormally will be restarted a few times to see if it works. If too many restarts fail, the job will be cancelled.

PID Reporting
When a job is preempted or stopped, programs working on it must be stopped. Since nodes start arbitrary processes, that in turn can start arbitrary processes, the cluster needs to know what processes to kill.
Pids are reported to the cluster by writing the following to standard output stream:

```
NEW_PID <pid>
```

If writing to the standard output is not an option, Pids can be written to files in the directory given to the user program as ¡pid_path¿. All files in the directory will be read and each line will be interpreted as a pid to be killed.

B.1.9 Add Program to the Cluster

In the */etc/clusterbusters.conf*, there's an entry *cluster_root* which tells the cluster where to look for program files.

B.1.10 Important Module APIs

B.1.10.1 Database API

The database API will mostly be used by other modules, thus providing the user with cleaner access to it. However, some functionality may be vital knowing about.

To start the database, the command *db:start_link()* or *db:start_link(test)* can be used. The latter command starts the database in a test environment, with the schema and tables stored only in memory, and thus all data will be lost in between sessions. If the former command is used, the tables must be manually created, using the command *db:create_tables(StorageType)*. *StorageType* denotes where the tables should be created, this is supplied directly to the Mnesia API, and should thus be *disc_copies*, for example. To stop the database server, the command *db:stop()* is used.

B.1.10.2 Chronicler API - Logging Levels

We have five levels of printouts from the logging process.

lopec_debug Debugging messages, not to be used in a live enviroment

lopec_info what developers might want to see

lopec_user_info the user may want to see progress and such, but not the inner workings. Also the level the jobs run in the cluster print to.

lopec_error something went very wrong, entire node or cluster may fail

lopec_warning something probably went wrong, but not catastrophically.

We thought warning, error and user_info could be interesting to the end user as they take a userid as an argument, allowing us to filter the messages in the interface depending on which user is logged in.

To change, use *chronicler:set_logging_level(List of desired levels)* once the cluster is running. The list may also just contain the atom "all" which will turn on all logging levels, this will produce alot of messages, use with caution.

The default is lopec_error, lopec_warning, lopec_user_info and lopec_info.

Note of caution setting the logging level to [all, lopec_user_info] will only activate user_info level

B.1.11 Running the Example Programs

They each contain a README file, which contain how to compile and try them out.

The example program are not final in any way. They can contian flaws and/or be poorly documented. They are to be considered as samples to get you started in writing your own programs using OpenCL and creating programs for our cluster.

B.1.11.1 Current Example Programs

Raytracer in OpenCL
A simple raytracer using OpenCL.

Image Filter
An image filter application that can do sharpness, gaussian blur, grayscale and emboss.

Image Filter in OpenCL
An image filter using OpenCL can only do sharpness at the moment.

Audio Filter
An application that adds echo to a wave file.

Appendix C

Edoc

Module chronicler

- Description
- Function Index
- Function Details

logger holds an API for logging messages on the server.

Behaviours: `gen_server`.

Authors: Fredrik Andersson (`sedrik@consbox.se`).

Description

logger holds an API for logging messages on the server. It uses error_logger for info, warning and error messages. Don't use it for debugging messages, if needed a debugging function can be added to the API later on. Currently no nice formatting of the message is done it's simply treated as single whole message and will be printed that way. See also http://www.erlang.org/doc/man /error_logger.html

Function Index

debug/1	Logs a debug message.
debug/2	Equivalent to `debug(io_lib:format(Format, Args))`.
error/2	Logs a error message.
error/3	Equivalent to `error(UserId, io_lib:format(Format, Args))`.
info/1	Logs a info message.
info/2	Equivalent to `info(io_lib:format(Format, Args))`.
set_logging_level/1	Changes the logging level of the logger, available levels are info, user_info, error, warning and debug.
set_tty/1	Turns on tty logging.
start_link/0	Starts the server.
user_info/2	Logs a user info message.
user_info/3	Equivalent to `user_info(UserId, io_lib:format(Format, Args))`.
warning/2	Logs a warning message.

warning/3	Equivalent to `warning(UserId, io_lib:format(Format, Args))`.

Function Details

debug/1

```
debug(Msg) -> ok
```

Logs a debug message

debug/2

```
debug(Format, Args) -> ok
```

Equivalent to `debug(io_lib:format(Format, Args))`.

error/2

```
error(UserId, Msg) -> ok
```

Logs a error message

error/3

```
error(UserId, Format, Args) -> ok
```

Equivalent to `error(UserId, io_lib:format(Format, Args))`.

info/1

```
info(Msg) -> ok
```

Logs a info message

info/2

```
info(Format, Args) -> ok
```

Equivalent to `info(io_lib:format(Format, Args))`.

set_logging_level/1

```
set_logging_level(NewLevel) -> ok
```

- `NewLevel = list()`

Changes the logging level of the logger, available levels are info, user_info, error, warning and debug

set_tty/1

```
set_tty(X1::on) -> ok
```

Turns on tty logging

start_link/0

```
start_link() -> {ok, Pid} | ignore | {error, Error}
```

Starts the server

user_info/2

```
user_info(UserId, Msg) -> ok
```

Logs a user info message

user_info/3

```
user_info(UserId, Format, Args) -> ok
```

Equivalent to `user_info(UserId, io_lib:format(Format, Args))`.

warning/2

```
warning(UserId, Msg) -> ok
```

Logs a warning message

warning/3

```
warning(UserId, Format, Args) -> ok
```

Equivalent to `warning(UserId, io_lib:format(Format, Args))`.

Generated by EDoc, Dec 17 2009, 09:55:02.

Module chronicler_sup

The logger supervisor Supervises the logging supervision tree.

Behaviours: supervisor.

Authors: Fredrik Andersson (sedrik@consbox.se).

Description

The logger supervisor Supervises the logging supervision tree

Function Index

start_link/0	Starts the supervisor.

Function Details

start_link/0

```
start_link() -> {ok, Pid} | ignore | {error, Error}
```

Starts the supervisor

Generated by EDoc, Dec 17 2009, 09:55:02.

Module computingProcess

The erlang process that communicates with the external process on the node.

Version: 0.0.2

Behaviours: `gen_server`.

Authors: Bjorn Dahlman (`bjorn.dahlman@gmail.com`).

Description

The erlang process that communicates with the external process on the node.

Function Index

code_change/3	
start_link/5	Starts the server.
stop/0	Stops the server.
stop_job/0	Stops the currently running task on the node.

Function Details

code_change/3

```
code_change(OldVsn, State, Extra) -> any()
```

start_link/5

```
start_link(ProgName, TaskType, JobId, StorageKeys, TaskId) -> {ok, Pid} | ignore | {error, Error}
```

Starts the server. Path is the path to the external program, Op is the first argument, Arg1 is the second and Arg2 is the third argument. So the os call will look like "Path Op Arg1 Arg2". The TaskId is there for the statistician.

stop/0

```
stop() -> void()
```

Stops the server.

stop_job/0

```
stop_job() -> ok
```

Stops the currently running task on the node.

Generated by EDoc, Dec 17 2009, 09:55:00.

Module configparser

Function Index

parse/2	Go throu the List and looks if there exist a Key.
read_config/2	

Function Details

parse/2

```
parse(Key, Config::List) -> {ok, Value} | {error, not_found}
```

Go throu the List and looks if there exist a Key. If so it returns the value of that key.

read_config/2

```
read_config(File, Key) -> any()
```

Generated by EDoc, Dec 17 2009, 09:55:05.

Module db

.

Behaviours: `gen_server`.

Authors: Henkan (`henkethalin@hotmail.com`), Nordh.

Description

db.erl contains the database API for the cluster. The API handles everything the user needs to work the Job and Task tables.

The database can also be started in test mode by using db:start(test). This will only create RAM copies of the db tabels for easy testing.

Function Index

add_bg_job/1	Adds a background job to the database.
add_job/1	Adds a job to the database.
add_task/1	Adds a task to the database.
add_user/3	Adds a user to the database.
cancel_job/1	Sets the state of the specified job to stopped.
code_change/3	
create_tables/1	Creates the tables and the schema used for keeping track of the jobs, tasks and the assigned tasks.
delete_user/1	Removes a user.
exist_user/1	Checks whether a user exists in the database.
fetch_task/1	Finds the task which is the next to be worked on and sets it as assigned to the specified node.
free_tasks/1	Marks all assigned tasks of the specified node as free.
get_job/1	Returns the whole job record from the database given a valid id.

get_task/1	Returns the whole task from the database given a valid id.
get_user/1	Gets a user from the database.
get_user_from_job/1	Returns the user of the given JobId.
get_user_jobs/1	Returns a list of JobIds belonging to the specified user.
job_status/1	Returns the status of a given job.
list/1	Lists all items in the specified table.
list_active_jobs/0	Returns a list of all jobs that currently have their states set as 'free'.
list_keys/1	Lists all keys pertaining to Bucket in the db.
list_users/0	Returns a list of all users with some of their info.
mark_done/1	Sets the state of the specified task to done.
pause_job/1	Sets the state of the specified job to paused.
remove_job/1	Removes a job and all its associated tasks.
resume_job/1	Set the state of the job to free so it can resume execution.
set_email/2	Changes the email address of a specific user.
set_email_notification/2	Changes the email notification field of a specific user.
set_job_path/2	Sets the path of the specified job.
set_job_state/2	Sets the state of the specified job.
set_password/3	Changes the password of a specific user.
set_role/2	Changes the role (and thus the rights) of a specific user.
start_link/1	Starts the database gen_server in a test environment, with all tables as ram copies only.
stop/0	Stops the database gen_server.
stop_job/1	Sets the state of the specified job to stopped.
task_info_from_job/2	Returns a tuple containing information of all tasks of a specific type belonging to the given JobId.
validate_user/2	Validates a user's name and password to the database.

Function Details

add_bg_job/1

```
add_bg_job(X1::{ProgramName::atom(), ProblemType::atom(), Owner::atom(), Priority::integer()}) ->
JobId | {error, Error}
```

Adds a background job to the database. ProgramName is the name of the program to be run, ProblemType is how the problem is run (by default map/reduce for now), Owner is the user who submitted the job and Priority is the priority of the job.

add_job/1

```
add_job(X1::{ProgramName::atom(), ProblemType::atom(), Owner::atom(), Priority::integer()}) ->
JobId | {error, Error}
```

Adds a job to the database. ProgramName is the name of the program to be run, ProblemType is how the problem is run (by default map/reduce for now), Owner is the user who submitted the job and Priority is the priority of the job.

add_task/1

```
add_task(X1::{JobId::integer(), ProgramName::atom(), Type::atom(), {Bucket::binary(),
Key::binary()}}) -> {ok, TaskId::integer()} | {error, Reason} | {ok, task_exists}
```

- `Type = split | map | reduce | finalize`

Adds a task to the database. The JobId is the id of the job the task belongs to, ProgramName denotes what kind of program the task runs, Type is the task type and Path is the path to the input relative to the NFS root.

add_user/3

```
add_user(Username::atom(), Email::string(), Password::string()) -> {ok, user_added} | {error,
Error}
```

Adds a user to the database.

cancel_job/1

```
cancel_job(JobId::integer()) -> TaskList | {error, Error}
```

Sets the state of the specified job to stopped. then removes the job from the job tale

code_change/3

```
code_change(OldVsn, State, Extra) -> any()
```

create_tables/1

```
create_tables(StorageType::atom()) -> ok | ignore | {error, Error}
```

- `StorageType = ram_copies | disc_copies | disc_only_copies`

Creates the tables and the schema used for keeping track of the jobs, tasks and the assigned tasks. When creating the tables in a stable environment, use disc_copies as argument. In test environments ram_copies is preferrably supplied as the argument.

delete_user/1

```
delete_user(UserName::string()) -> {ok} | {error, Error}
```

Removes a user.

exist_user/1

```
exist_user(Username::atom()) -> {ok, yes} | {ok, no}
```

Checks whether a user exists in the database.

fetch_task/1

```
fetch_task(NodeId::atom()) -> Task::record()
```

Finds the task which is the next to be worked on and sets it as assigned to the specified node.

free_tasks/1

```
free_tasks(NodeId::atom()) -> List | {error, Error}
```

- `List = [{JobId::integer(), TaskType::atom()}]`

Marks all assigned tasks of the specified node as free.

get_job/1

```
get_job(JobId::integer()) -> Job::record() | {error, Error}
```

Returns the whole job record from the database given a valid id.

get_task/1

`get_task(TaskId::integer()) -> Task::record() | {error, Error}`

Returns the whole task from the database given a valid id.

get_user/1

`get_user(Username::atom()) -> User | {error, Error}`

Gets a user from the database.

get_user_from_job/1

`get_user_from_job(JobId::integer()) -> User::atom() | {error, Error}`

Returns the user of the given JobId.

get_user_jobs/1

`get_user_jobs(User::atom()) -> List | {error, Error}`

- `List = [JobId::integer()]`

Returns a list of JobIds belonging to the specified user.

job_status/1

`job_status(JobId::integer()) -> {ok, active} | {ok, paused} | {ok, stopped} | {error, Error}`

Returns the status of a given job.

list/1

`list(TableName::atom()) -> List | {error, Error}`

Lists all items in the specified table.

list_active_jobs/0

`list_active_jobs() -> List`

- `List = [JobId::integer()]`

Returns a list of all jobs that currently have their states set as 'free'.

list_keys/1

```
list_keys(Bucket::binary()) -> Keys::[binary()]
```

Lists all keys pertaining to Bucket in the db.

list_users/0

```
list_users() -> {ok, List} | {error, Error}
```

- `List = [{Username::string(), Email::string(), Role::atom()}]`

Returns a list of all users with some of their info.

mark_done/1

```
mark_done(TaskId::integer()) -> ok | {error, Error}
```

Sets the state of the specified task to done.

pause_job/1

```
pause_job(JobId::integer()) -> ok | {error, Error}
```

Sets the state of the specified job to paused.

remove_job/1

```
remove_job(JobId) -> ok | {error, Error}
```

Removes a job and all its associated tasks.

resume_job/1

```
resume_job(JobId::integer()) -> ok | {error, Error}
```

Set the state of the job to free so it can resume execution.

set_email/2

```
set_email(Username::atom(), NewEmail::atom()) -> {ok, email_set} | {error, Error}
```

Changes the email address of a specific user.

set_email_notification/2

`set_email_notification(Username::atom(), EmailNotify::boolean()) -> {ok, email_notice_set} | {error, Error}`

Changes the email notification field of a specific user.

set_job_path/2

`set_job_path(JobId::integer(), NewPath::string()) -> ok | {error, Error}`

Sets the path of the specified job.

set_job_state/2

`set_job_state(JobId::integer(), NewState::atom()) -> ok | {error, Error}`

- `NewState = free | paused | stopped | done`

Sets the state of the specified job.

set_password/3

`set_password(Username::atom(), OldPassword::atom(), NewPassword::atom()) -> {ok, password_set} | {error, Error}`

Changes the password of a specific user.

set_role/2

`set_role(Username::atom(), NewRole::atom()) -> {ok, role_set} | {error, Error}`

Changes the role (and thus the rights) of a specific user.

start_link/1

`start_link(X1::test:atom()) -> ok | {error, Error}`

Starts the database gen_server in a test environment, with all tables as ram copies only.

stop/0

`stop() -> ok | {error, Error}`

Stops the database gen_server.

stop_job/1

```
stop_job(JobId::integer()) -> TaskList | {error, Error}
```

Sets the state of the specified job to stopped.

task_info_from_job/2

```
task_info_from_job(Type::atom(), JobId::integer()) -> {ok, Info} | {error, Error}
```

Returns a tuple containing information of all tasks of a specific type belonging to the given JobId.

validate_user/2

```
validate_user(Username::atom(), Password::string()) -> {ok, user_validated} | {error, Error}
```

Validates a user's name and password to the database.

Generated by EDoc, Dec 17 2009, 09:54:59.

Module diskMemHandler

Custom event handler, adds itself to SASL event manager 'alarm_handler' and sends any alarm event receieved to global statistician, as a cast in log_alarm.

Authors: Henkan (`henkethalin@hotmail.com`), Gustav Simonsson (`gusi7871@student.uu.se`).

Description

Custom event handler, adds itself to SASL event manager 'alarm_handler' and sends any alarm event receieved to global statistician, as a cast in log_alarm.

Function Index

handle_call/2
handle_event/2
handle_info/2
init/1
start/0
stop/0
terminate/2

Function Details

handle_call/2

```
handle_call(Query, Alarms) -> any()
```

handle_event/2

```
handle_event(X1, Alarms) -> any()
```

handle_info/2

```
handle_info(X1, Alarms) -> any()
```

init/1

```
init(X1) -> any()
```

start/0

```
start() -> any()
```

stop/0

```
stop() -> any()
```

terminate/2

```
terminate(Reason, Alarms) -> any()
```

Generated by EDoc, Dec 17 2009, 09:55:05.

Module dispatcher

Interfaces with the database.

Behaviours: gen_server.

Authors: Axel Andren (axelandren@gmail.com), Vasilij Savin (vasilij.savin@gmail.com).

Description

Interfaces with the database. Can take requests for tasks, marking a task as done, adding tasks or jobs, and free tasks assigned to nodes (by un-assigning them).

Function Index

add_job/2	Adds specified job to the database job list, add it to the bg job list instead iff IsBGJob.
add_task/1	Adds specified task to the database task list.
cancel_job/1	Cancels a job.
fetch_task/2	Sends a message to the caller with the first available task.
free_tasks/1	Frees all tasks assigned to Node in master task list.
get_split_amount/0	Returns the amount of splits to be done.
get_user_from_job/1	Returns the user associated with the job.
handle_call/3	Marks a specified task as done in the database.
handle_cast/2	Expects task requests from nodes, and passes such requests to the find_task function.
report_task_done/1	Marks the task as being completely done.
report_task_done/2	Like report_task_done/1 except the node can ask to generate another task by providing a TaskSpec.
start_link/0	Starts the server.

stop_job/1	Stops a job.
task_failed/2	Increases the task restart counter for the job and makes the task free.

Function Details

add_job/2

```
add_job(JobSpec, X2::IsBGJob) -> JobID
```

Adds specified job to the database job list, add it to the bg job list instead iff IsBGJob.

```
  JobSpec is a tuple:
  {
       ProgramName,
       ProblemType (map reduce only accepted now)
       Owner
       Priority - not implemented at the moment
  }
```

add_task/1

```
add_task(TaskSpec) -> TaskID
```

Adds specified task to the database task list.

```
  TaskSpec is a tuple:
  {
       JobId,
       ProgramName,
       Type - atoms 'map', 'reduce', 'finalize' or 'split' are
                  accepted at the moment (without quote marks '')
       Path - input file name
  }
```

cancel_job/1

```
cancel_job(JobId) -> ok
```

Cancels a job. Its the same as for stop_job/1 but the job will also be removed from the database.

fetch_task/2

```
fetch_task(NodeId::NodeID, PID) -> ok
```

Sends a message to the caller with the first available task. If no task is available a {task_response, no_task} message is returned

free_tasks/1

```
free_tasks(NodeId::NodeID) -> ok
```

Frees all tasks assigned to Node in master task list

get_split_amount/0

```
get_split_amount() -> Amount::integer()
```

Returns the amount of splits to be done.

get_user_from_job/1

```
get_user_from_job(JobId) -> User
```

Returns the user associated with the job

handle_call/3

```
handle_call(Msg::{task_done, TaskId, no_task}, From, State) -> {reply, ok, State}
```

Marks a specified task as done in the database.

handle_cast/2

```
handle_cast(Msg::{task_request, NodeId, From}, State) -> {noreply, State}
```

Expects task requests from nodes, and passes such requests to the find_task function.

report_task_done/1

```
report_task_done(TaskId::TaskID) -> ok
```

Marks the task as being completely done. The results should be posted on storage before calling this method.

report_task_done/2

```
report_task_done(TaskId::TaskID, TaskSpec) -> ok
```

Like report_task_done/1 except the node can ask to generate another task by providing a TaskSpec

start_link/0

```
start_link() -> {ok, Pid} | ignore | {error, Error}
```

Starts the server.

stop_job/1

```
stop_job(JobId) -> ok
```

Stops a job. A stopping job is halted before completion and stays in that state until its resumed

task_failed/2

```
task_failed(JobId, TaskType) -> ok | {ok, stopped}
```

Increases the task restart counter for the job and makes the task free. If the threshold for the max task restarts is reached for the job the job will be stopped.

Generated by EDoc, Dec 17 2009, 09:55:05.

Module dynamicSupervisor

A supervisor for dynamic processes spawning, called with externally defined child specifications.

Behaviours: `supervisor`.

Authors: Bjorn Dahlman (`bjorn.dahlman@gmail.com`).

Description

A supervisor for dynamic processes spawning, called with externally defined child specifications. Currently only called by taskFetcher to spawn computingProcess.

Function Index

start_link/0	Starts the supervisor.

Function Details

start_link/0

```
start_link() -> {ok, Pid} | ignore | {error, Error}
```

Starts the supervisor

Generated by EDoc, Dec 17 2009, 09:55:00.

Module ecg

This module handles start of electrocardiogram application.

Behaviours: `application`.

Authors: Vasilij Savin (`vasilij.savin@gmail.com`).

Description

This module handles start of electrocardiogram application. Currently it is not using any parameters.

Function Index

start/2	
stop/1	

Function Details

start/2

```
start(Type, StartArgs) -> any()
```

stop/1

```
stop(State) -> any()
```

Generated by EDoc, Dec 17 2009, 09:55:03.

Module ecg_server

ElectroCardioGram - process that keeps track of all alive computational nodes.

Behaviours: `gen_server`.

Authors: Gustav Simonsson (`gusi7871@student.uu.se`), Vasilij Savin (`vasilij.savin@gmail.com`).

Description

ElectroCardioGram - process that keeps track of all alive computational nodes

Function Index

accept_message/1	Main interface with ECG.
handle_call/3	
handle_cast/2	
handle_info/2	
init/1	Boots up ECG - cluster heartbeat listener.
start_link/0	gen_server callback function.

Function Details

accept_message/1

```
accept_message(Msg) -> any()
```

Main interface with ECG. ECG waits for 3 types of messages: {nodeup} and {nodedown} are generated by net_kernel. {new_node} notifies that potential new node arrived. ECG then checks if this process is already known and establish connection in case there is no prior connection. Everything else is passed to logger and ignored.

handle_call/3

```
handle_call(Request, From, State) -> any()
```

handle_cast/2

```
handle_cast(Msg, State) -> any()
```

handle_info/2

```
handle_info(Info, State) -> any()
```

init/1

```
init(X1) -> any()
```

Boots up ECG - cluster heartbeat listener.

start_link/0

```
start_link() -> any()
```

gen_server callback function. Starting ECG server and initialise it to listen to network messages

Generated by EDoc, Dec 17 2009, 09:55:03.

Module ecg_sup

ECG supervisor - watches a single worker, ECG server.

Behaviours: `supervisor`.

Authors: Vasilij Savin (`vasilij.savin@gmail.com`).

Description

ECG supervisor - watches a single worker, ECG server.

Function Index

init/1
start_link/0

Function Details

init/1

```
init(X1) -> any()
```

start_link/0

```
start_link() -> any()
```

Generated by EDoc, Dec 17 2009, 09:55:03.

Module examiner

- Function Index
- Function Details

Behaviours: `gen_server`.

Function Index

get_progress/1	Returns the current information about all tasks created by given JobId.
get_promising_job/0	Returns the jobid of the job that is closest to be completed.
insert/1	Insert a new job to be tracked by examiner.
remove/1	Removes the job with JobId from the examiner.
report_assigned/2	Report that a task of type TaskType in the job with the id JobId was assigned.
report_created/2	Report that a task of type TaskType in the job with the id JobId was created.
report_done/2	Report that a task of type TaskType in the job with the id JobId was done.
report_free/1	Report that all tasks ({JobId, TaskType}) in Tasks were freed.
start_link/0	Starts the server.
stop/0	Stops the server and cleans up.

Function Details

get_progress/1

```
get_progress(JobId) -> {job_stats, JobId, {FreeSplits, AssignedSplits, DoneSplits}, {FreeMaps,
AssignedMaps, DoneMaps}, {FreeReduces, AssignedReduces, DoneReduces}, {FreeFinalizes,
AssignedFinalizes, DoneFinalizes}}
```

Returns the current information about all tasks created by given JobId.

get_promising_job/0

```
get_promising_job() -> {ok, JobId} | {error, Reason}
```

Returns the jobid of the job that is closest to be completed.

insert/1

```
insert(JobId) -> ok
```

Insert a new job to be tracked by examiner. IMPORTANT: if job is not inserted before usage, it will crash examiner.

remove/1

```
remove(JobId) -> ok
```

Removes the job with JobId from the examiner.

report_assigned/2

```
report_assigned(JobId, TaskType) -> ok
```

Report that a task of type TaskType in the job with the id JobId was assigned.

report_created/2

```
report_created(JobId, TaskType) -> ok
```

Report that a task of type TaskType in the job with the id JobId was created.

report_done/2

```
report_done(JobId, TaskType) -> ok
```

Report that a task of type TaskType in the job with the id JobId was done.

report_free/1

```
report_free(Tasks) -> ok
```

Report that all tasks ({JobId, TaskType}) in Tasks were freed.

start_link/0

```
start_link() -> {ok, Pid} | ignore | {error, Error}
```

Starts the server.

stop/0

```
stop() -> ok
```

Stops the server and cleans up.

Generated by EDoc, Dec 17 2009, 09:54:59.

Module fs_io_module

Deals with the temporary storage in the cluster.

Authors: Vasilij Savin.

Description

Deals with the temporary storage in the cluster. Gets a binary stream of data to write or returns the binary stream of data.

Function Index

get/3	Gets the value associated with the bucket and the key.
put/4	Puts a value to the storage, either the file system or riak depending on how the server was started.

Function Details

get/3

```
get(Bucket, Key, State) -> {ok, binary()} | {error, Reason}
```

Gets the value associated with the bucket and the key.

put/4

```
put(Bucket, Key, Value::Val, State) -> ok | {error, Reason}
```

Puts a value to the storage, either the file system or riak depending on how the server was started.

Generated by EDoc, Dec 17 2009, 09:55:05.

Module io_module

- Description
- Function Index
- Function Details

Deals with the temporary storage in the cluster.

Behaviours: `gen_server`.

Authors: Bjorn Dahlman (`bjorn.dahlman@gmail.com`), Vasilij Savin.

Description

Deals with the temporary storage in the cluster. Gets a binary stream of data to write or returns the binary stream of data.

Function Index

get/2	Gets the value associated with the bucket and the key.
put/3	Puts a value to the storage, either the file system or riak depending on how the server was started.
start_link/2	Starts the server Currently supported storage types = fs_io_module | riak_io_module.
stop/0	Stops the server.

Function Details

get/2

```
get(Bucket, Key) -> binary() | {error, Reason}
```

Gets the value associated with the bucket and the key.

put/3

```
put(Bucket, Key, Val) -> ok | {error, Reason}
```

Puts a value to the storage, either the file system or riak depending on how the server was started.

start_link/2

```
start_link(ModuleName::atom(), Args::list()) -> {ok, Pid} | ignore | {error, Error}
```

Starts the server Currently supported storage types = fs_io_module | riak_io_module

stop/0

```
stop() -> ok
```

Stops the server

Generated by EDoc, Dec 17 2009, 09:55:05.

Module janitor

- Description
- Function Index
- Function Details

Our garbagecollector.

Behaviours: `gen_server`.

Authors: Burbas (`niclas@burbas.se`).

Description

Our garbagecollector

Function Index

cleanup_finalize/1	Clean up storage from finalize-files beloning to a specific job.
cleanup_job/1	Clean up storage after a job have finished.
cleanup_map/1	Clean up storage from map-files beloning to a specific job.
cleanup_reduce/1	Clean up storage from reduce-files beloning to a specific job.
cleanup_split/1	Clean up storage from split-files belonging to a specific job.
start_link/0	Starts the server.

Function Details

cleanup_finalize/1

```
cleanup_finalize(JobId) -> ok | {error, Reason}
```

Clean up storage from finalize-files beloning to a specific job.

cleanup_job/1

```
cleanup_job(JobId) -> ok | {error, Reason}
```

Clean up storage after a job have finished

cleanup_map/1

```
cleanup_map(JobId) -> ok | {error, Reason}
```

Clean up storage from map-files beloning to a specific job.

cleanup_reduce/1

```
cleanup_reduce(JobId) -> ok | {error, Reason}
```

Clean up storage from reduce-files beloning to a specific job.

cleanup_split/1

```
cleanup_split(JobId) -> ok | {error, Reason}
```

Clean up storage from split-files belonging to a specific job.

start_link/0

```
start_link() -> {ok, Pid}
```

Starts the server

Generated by EDoc, Dec 17 2009, 09:54:59.

Module listener

Listener - The link between our cluster and the user.

Behaviours: `gen_server`.

Authors: Burbas (`niclas@burbas.se`).

Description

Listener - The link between our cluster and the user. Process that listens for new jobs from user.

Function Index

add_bg_job/5	This is the same as add_job/6 but without the 'Name'-variable.
add_bg_job/6	When a new job is reported a series of new directories will be created and the input file will be moved to this new structure.
add_job/5	This is the same as add_job/6 but without the 'Name'-variable.
add_job/6	When a new job is reported a series of new directories will be created and the input file will be moved to this new structure.
cancel_job/1	cancel a job Does the same as stop/1 but it also removes the job from the database.
get_job_id/1	Returns the JobId of the job with name JobName, or {error, Reason} if there was no job with JobName.
get_job_name/1	Returns the name the user gave JobId when starting it.
pause_job/1	Pauses a job.
remove_job_name/1	Disassociates JobId with any saved name.
resume_job/1	Resumes a paused or stopped job.
start_link/0	Starts the server.

stop_job/1	Stops a job Hard-stops a job.

Function Details

add_bg_job/5

```
add_bg_job(ProgramType, ProblemType, Owner, Priority, InputData) -> JobID
```

This is the same as add_job/6 but without the 'Name'-variable.

add_bg_job/6

```
add_bg_job(ProgramType, ProblemType, Owner, Priority, InputData, Name) -> JobID
```

When a new job is reported a series of new directories will be created and the input file will be moved to this new structure. When this is done a new split-task is created. Name is the atom the user wants to associate with the job id, or no_name if no association is wanted.

add_job/5

```
add_job(ProgramType, ProblemType, Owner, Priority, InputData) -> JobID
```

This is the same as add_job/6 but without the 'Name'-variable.

add_job/6

```
add_job(ProgramType, ProblemType, Owner, Priority, InputData, Name) -> JobID
```

When a new job is reported a series of new directories will be created and the input file will be moved to this new structure. When this is done a new split-task is created. Name is the atom the user wants to associate with the job id, or no_name if no association is wanted.

cancel_job/1

```
cancel_job(JobId) -> ok
```

cancel a job Does the same as stop/1 but it also removes the job from the database.

get_job_id/1

```
get_job_id(JobName) -> {ok, JobId} | {error, Reason}
```

Returns the JobId of the job with name JobName, or {error, Reason} if there was no job with JobName.

get_job_name/1

```
get_job_name(JobId) -> {name, Name} | anonymous
```

Returns the name the user gave JobId when starting it. If no name was given, anonymous is returned

pause_job/1

```
pause_job(JobId) -> ok
```

Pauses a job.

remove_job_name/1

```
remove_job_name(JobId) -> ok
```

Disassociates JobId with any saved name.

resume_job/1

```
resume_job(JobId) -> ok
```

Resumes a paused or stopped job.

start_link/0

```
start_link() -> {ok, Pid}
```

Starts the server

stop_job/1

```
stop_job(JobId) -> ok
```

Stops a job Hard-stops a job. The job will be stopped without finishing current tasks.

Generated by EDoc, Dec 17 2009, 09:54:59.

Module main_chronicler

main_chronicler is responsible for keeping a database over the logging messages passed to the system.

Behaviours: `gen_server`.

Authors: Fredrik Andersson (`sedrik@consbox.se`).

Description

main_chronicler is responsible for keeping a database over the logging messages passed to the system. It runs on the node logger only and should only be runned once since it is globaly registered.

Function Index

get_all_logs/0	Returns all the log messages in the database.
get_custom_logs/1	Returns all the log messages that matches the record Record, It takes a match record that corresponds to the log_message record.
get_node_logs/1	Returns all the log messages in the database from node Node.
get_type_logs/1	Returns all the log messages in the database with type Type.
get_user_logs/1	Returns all the log messages in the database from user User.
print_it/1	
start_link/0	Starts the master chronicler that holds a database over the log messages in the system.

Function Details

get_all_logs/0

`get_all_logs() -> {ok, Match}`

Returns all the log messages in the database

get_custom_logs/1

`get_custom_logs(Record::Type) -> {ok, Match}`

Returns all the log messages that matches the record Record, It takes a match record that corresponds to the log_message record

get_node_logs/1

`get_node_logs(Node::User) -> {ok, Match}`

Returns all the log messages in the database from node Node

get_type_logs/1

`get_type_logs(Type) -> {ok, Match}`

Returns all the log messages in the database with type Type

get_user_logs/1

`get_user_logs(User) -> {ok, Match}`

Returns all the log messages in the database from user User

print_it/1

`print_it(X) -> any()`

start_link/0

`start_link() -> {ok, Pid} | ignore | {error, Error}`

Starts the master chronicler that holds a database over the log messages in the system.

Generated by EDoc, Dec 17 2009, 09:55:02.

Module master_node

This module handles start of master node application.

Behaviours: `application`.

Authors: Vasilij Savin (`vasilij.savin@gmail.com`).

Description

This module handles start of master node application. Currently it is not using any parameters.

Function Index

start/2	
stop/1	All arguments are ignored by now.

Function Details

start/2

```
start(Type, StartArgs) -> any()
```

stop/1

```
stop(X1) -> any()
```

All arguments are ignored by now. Stops application and logs shutdown.

Generated by EDoc, Dec 17 2009, 09:54:59.

Module master_sup

Master supervisor supervises WPM processes.

Behaviours: `supervisor`.

Authors: Vasilij Savin (`vasilij.savin@gmail.com`), Gustav Simonsson (`gusi7871@student.uu.se`).

Description

Master supervisor supervises WPM processes. Currently there are 3 processes to monitor: Listener - listen to job submissions from users DbDaemon - interaction with DB Dispatcher - listens to task requests from nodes

Function Index

init/1
start_link/0

Function Details

init/1

```
init(X1) -> any()
```

start_link/0

```
start_link() -> any()
```

Generated by EDoc, Dec 17 2009, 09:54:59.

Module netMonitor

Fetches information about the network statistics on the system.

Authors: Bjoern Dahlman.

Description

Fetches information about the network statistics on the system.

Function Index

get_net_stats/0	Asks the system how much data has been sent/received on a network interface card (hardcoded as en0/eth0).

Function Details

get_net_stats/0

```
get_net_stats() -> {Up, Down}
```

Asks the system how much data has been sent/received on a network interface card (hardcoded as en0/eth0).

Generated by EDoc, Dec 17 2009, 09:55:00.

Module riak_io_module

Deals with the temporary storage in the cluster.

Authors: Vasilij Savin.

Description

Deals with the temporary storage in the cluster. Gets a binary stream of data to write or returns the binary stream of data.

Function Index

get/3	Gets the value associated with the bucket and the key.
put/4	Puts a value to the storage, either the file system or riak depending on how the server was started.

Function Details

get/3

```
get(Bucket, Key, X3::State) -> binary() | {error, Reason}
```

Gets the value associated with the bucket and the key.

put/4

```
put(Bucket, Key, Value::Val, X4::State) -> ok | {error, Reason}
```

Puts a value to the storage, either the file system or riak depending on how the server was started.

Generated by EDoc, Dec 17 2009, 09:55:05.

Module statistician

Collects various statistics about the cluster nodes and jobs put in the cluster.

Behaviours: gen_server.

Authors: Axel "Align" Andren (axelandren@gmail.com), Bjorn "norno" Dahlman (bjorn.dahlman@gmail.com), Gustav "azariah" Simonsson (gusi7871@student.uu.se), Vasilij "Chabbrik" Savin (vasilij.savin@gmail.com).

Description

Collects various statistics about the cluster nodes and jobs put in the cluster.

```
 Cluster global statistics include:
    * Jobs executed (also those that are done or cancelled)
    * Power consumed (estimation)
    * Time spent executing tasks (sum total for all nodes)
    * Upload network traffic (total unfortunately, not just ours)
    * Download network traffic (ditto)
    * Number of tasks processed
    * Number of task restarts
    * Total amount of diskspace in cluster
    * Total amount of diskspace used in cluster
    * % of diskspace in cluster that is used
    * Total amount of primary memory in cluster
    * Total amount of primary memory used in cluster
    * % of primary memory used in cluster
```

Function Index

get_cluster_disk_usage/1	Returns average disk usage over all nodes.
get_cluster_mem_usage/1	Returns average primary memory usage over all nodes.
get_cluster_stats/1	Returns stats for the entire cluster.
get_job_stats/2	Returns stats for JobId.
get_node_disk_usage/1	Returns disk usage on a node.

get_node_job_stats/3	Returns stats the node NodeId has for the job JobId, like how many JobId tasks NodeId has worked on, or how long.
get_node_mem_usage/1	Returns memory usage on a node.
get_node_stats/2	Returns stats for NodeId.
get_user_stats/2	Returns stats for the given user.
job_finished/1	Jobs, once finished in our cluster, have their stats dumped to file and their entry cleared out of the ets table.
remove_node/1	Remove a node from the global stats.
start_link/1	Starts the server.
stop/0	Stops the statistician and all related applications and modules.
update/1	Updates local (node) ets table with statistics, adding the job and its stats to the table if it doesn't already exist, otherwise updating the existing entry.

Function Details

get_cluster_disk_usage/1

```
get_cluster_disk_usage(Flag) -> String | {Total::Integer, Percentage::Integer}
```

Returns average disk usage over all nodes.

```
  Flag:
   raw - gives internal representation (Tuples, lists, whatnot)
   string - gives nicely formatted string
```

get_cluster_mem_usage/1

```
get_cluster_mem_usage(Flag) -> String | {Total::Integer, Percentage::Integer}
```

Returns average primary memory usage over all nodes.

```
  Flag:
   raw - gives internal representation (Tuples, lists, whatnot)
   string - gives nicely formatted string
```

get_cluster_stats/1

```
get_cluster_stats(Flag) -> String
```

Returns stats for the entire cluster.

```
  Flag:
   raw - gives internal representation (Tuples, lists, whatnot)
   string - gives nicely formatted string
```

get_job_stats/2

```
get_job_stats(JobId, X2::Flag) -> String
```

Returns stats for JobId.

```
  Flag:
   raw - gives internal representation (a list of the total stats)
   string - gives nicely formatted string with stats for each tasktype
```

get_node_disk_usage/1

```
get_node_disk_usage(Flag) -> String | {Total::Integer, Percentage::Integer}
```

Returns disk usage on a node.

```
  Flag:
   raw - gives internal representation (Tuples, lists, whatnot)
   string - gives nicely formatted string
```

get_node_job_stats/3

```
get_node_job_stats(NodeId, JobId, X3::Flag) -> String
```

Returns stats the node NodeId has for the job JobId, like how many JobId tasks NodeId has worked on, or how long.

```
  Flag:
       raw - gives internal representation (Tuples, lists, whatnot)
       string - gives nicely formatted string
```

get_node_mem_usage/1

```
get_node_mem_usage(Flag) -> String | {Total::Integer, Percentage::Integer}
```

Returns memory usage on a node.

```
  Flag:
   raw - gives internal representation (Tuples, lists, whatnot)
   string - gives nicely formatted string
```

get_node_stats/2

`get_node_stats(NodeId, X2::Flag) -> String`

Returns stats for NodeId.

```
  Flag:
       raw - gives internal representation (Tuples, lists, whatnot)
       string - gives nicely formatted string
```

get_user_stats/2

`get_user_stats(User, X2::Flag) -> String`

Returns stats for the given user.

```
  Flag:
       raw - gives internal representation (Tuples, lists, whatnot)
       string - gives nicely formatted string
```

job_finished/1

`job_finished(JobId) -> please_wait_a_few_seconds`

Jobs, once finished in our cluster, have their stats dumped to file and their entry cleared out of the ets table. However, we have to wait to make sure that all slaves have sent their stats updates - we hope that waiting two update intervals will be sufficient, but if a node is stalled for more than that long, we're out of luck.

This wait is done using timer:send_after/3, which sends a regular Erlang message, meaning we have to use handle_info/2 to catch it. After the message is catched we pass the command onto handle_cast/2 though.

remove_node/1

`remove_node(NodeId) -> ok`

Remove a node from the global stats. Probably called when a node drops from the cluster for some reason.

start_link/1

```
start_link(Type) -> {ok, Pid} | ignore | {error, Error}
```

Starts the server.

```
  Type:
   slave  - start a slave node statistician. It intermittently flushes
            collected stats to the master.
   master - start a master node statistician. It keeps track of node
            (global) stats as well as job stats.
```

stop/0

```
stop() -> ok
```

Stops the statistician and all related applications and modules.

update/1

```
update(Data) -> ok
```

Updates local (node) ets table with statistics, adding the job and its stats to the table if it doesn't already exist, otherwise updating the existing entry.

```
  The Data variable should look like this tuple:
  {{NodeId, JobId, TaskType},
       Power, Time, Upload, Download, NumTasks, Restarts, Disk, Mem}
  where Disk and Mem are formatted like calls to
  get_node_disk/mem_stats(raw)
```

Generated by EDoc, Dec 17 2009, 09:55:05.

Module taskFetcher

The taskFetcher is responsible for fetching and adding tasks.

Behaviours: `gen_server`.

Authors: Bjorn Dahlman (`bjorn.dahlman@gmail.com`), Fredrik Andersson (`sedrik@consbox.se`).

Description

The taskFetcher is responsible for fetching and adding tasks.

Function Index

error/1	
new_task/5	Queries the dispatcher to create a new task.
start_link/0	Starts the server.
task_done/1	

Function Details

error/1

```
error(Data) -> any()
```

new_task/5

```
new_task(JobId, ProgName, Type, Bucket, Key) -> Task | {error, Error}
```

Queries the dispatcher to create a new task.

start_link/0

```
start_link() -> {ok, Pid} | ignore | {error, Error}
```

Starts the server

task_done/1

```
task_done(Data) -> any()
```

Generated by EDoc, Dec 17 2009, 09:55:00.

Module utils

Library containing convenient things for the admin to use.

Authors: Bjorn Dahlman (`bjorn.dahlman@gmail.com`).

Description

Library containing convenient things for the admin to use.

Function Index

get_cluster_path/0	Gets the path to the cluster from the config file defined in env.hrl.
get_program_executables/1	Checks what executables are available for a specific program.
get_programs/0	Fetches a list of programs in the cluster.

Function Details

get_cluster_path/0

```
get_cluster_path() -> Path::string()
```

Gets the path to the cluster from the config file defined in env.hrl

get_program_executables/1

```
get_program_executables(Program::term()) -> Executables::list()
```

Checks what executables are available for a specific program.

get_programs/0

```
get_programs() -> Programs::list()
```

Fetches a list of programs in the cluster.

Generated by EDoc, Dec 17 2009, 09:55:05.