Pipeline-collector: A new way to gather performance data for astronomical pipelines

Alexandar P. Mechev[a], Aske Plaat[b], J.B. Raymond Oonk[a,c], Huib T. Intema[a], Huub J.A. Röttgering[a]

[a] *Leiden Observatory, Niels Bohrweg 2, 2333 CA Leiden, the Netherlands*

[b] *Leiden Institute of Advanced Computer Science, Niels Bohrweg 1, 2333 CA Leiden, the Netherlands*

[c] *ASTRON, Oude Hoogeveensedijk 4, 7991 PD Dwingeloo, the Netherlands*

Abstract

Modern astronomical data processing requires complex software pipelines to process ever growing datasets. For radio astronomy, these pipelines have become so large that they need to be distributed across a computational cluster. This makes it difficult to monitor the performance of each pipeline step. To gain insight into the performance of each step, a performance monitoring utility needs to be integrated with the pipeline execution. In this work we have developed such a utility and integrated it with the calibration pipeline of the Low Frequency Array, LOFAR, a leading radio telescope. We tested the tool by running the pipeline on several different compute platforms and collected the performance data. Based on this data, we make well informed recommendations on future hardware and software upgrades. The aim of these upgrades is to accelerate the slowest processing steps for this LOFAR pipeline. The *pipeline_collector* suite is open source and will be incorporated in future LOFAR pipelines to create a performance database for all LOFAR processing.

Keywords: Radio Astronomy, Performance Analysis, Profiling, High Performance Computing

1. Introduction

Astronomical data often requires significant processing before it is considered ready for scientific analysis. This processing is done increasingly by complex and autonomous software pipelines, often consisting of numerous processing steps, which are run without user interaction. It is necessary to collect performance statistics for each pipeline step. Doing so will enable scientists to discover and address software and hardware inefficiencies and produce scientific data at a higher rate. To identify these inefficiencies, we have extended the performance monitoring package *tcollector* (Apache, 2017). The resulting suite, *pipeline_collector*, makes it possible to use *tcollector* to record data for complex pipelines. We have used a leading radio telescope as the test case for the *pipeline_collector* suite. The discoveries made with our software will help remove bottlenecks and suggest hardware requirements for current and future processing clusters. We summarize our findings in Table 2 in Section 3.

Over the past two decades, processing data in radio astronomy has increasingly moved from personal machines to large compute clusters. Over this time, radio telescopes have undergone upgrades in the form of wide band receivers and upgraded correlators (Broekema et al., 2018) (Gupta et al., 2017). In addition, several aperture synthesis arrays such as the Low Frequency Array (LOFAR) (Van Haarlem et al., 2013) , Murchison Widefield Array (MWA) (Lonsdale et al., 2009) and MeerKAT (Jonas, 2009) have begun observing the radio sky, leading to an increase of data rates by up to 3 orders of magnitude (Wu et al., 2013)(Davidson, 2012).

As the data acquisition rate has increased, data size has entered the Petabyte regime, and processing requirements increased to millions of CPU-hours. In order for processing to match the acquisition rate, the data is increasingly processed at large clusters with high-bandwidth connections to the data. An important case where data processing is done at a high throughput cluster is the LOFAR radio telescope.

The LOFAR telescope is a European low frequency aperture synthesis radio telescope based mainly in the Netherlands with stations across Europe. This aperture synthesis telescope requires significant data processing before producing scientific images (Van Weeren et al., 2016) (Williams et al., 2016) (Smirnov and Tasse, 2015). In this work, we will use our performance monitoring utility, *pipeline_collector*[1], to study the first half of the LOFAR processing, the Direction Independent pipeline.

One major project for the LOFAR telescope is the Surveys Key Science Project (SKSP) (Shimwell et al., 2017). This project consists of more than 3000 observations of 8 hours each, 600 of which have been observed. These observations need to be processed by a Direction Independent (DI) pipeline, the results of which are calibrated by

Email address: apmechev@strw.leidenuniv.nl (Alexandar P. Mechev[a])

[1] https://gitlab.com/apmechev/procfs_tcollector.git

a Direction Dependent (DD) pipeline. The DI pipeline is implemented in the software package *prefactor*[2]. The *prefactor* pipeline is itself split into four stages and implemented at the SURFsara Grid location at the Amsterdam e-Science centre (SURF, 2018)(Mechev et al., 2017). The automation and simple parallelization has decreased the run time per dataset from several days to six hours, making it comparable to the observation rate. To better understand the performance of the *prefactor* pipeline, we require detailed performance information for all steps of the processing software. We have developed a utility to gather this information for data processing pipelines running on distributed compute systems.

In this work, we will use the *pipeline_collector* utility to study the LOFAR *prefactor* pipeline and suggest optimization based on our results. To test the software on a diverse set of hardware, we will set up the monitoring package on four different computers and collect data on the pipeline's performance. Using this data, we discuss several aspects of the LOFAR software which we present in Table 2. Finally we discuss the broader context of these optimizations and the LOFAR SKSP project. Finally we will touch on the integration of *pipeline_collector* with the second half of the data processing pipeline, the DD calibration and imaging.

1.1. Related Work

Scientific fields that need to process large data sets employ some type of data processing pipelines. Such pipelines include e.g. solar imaging (Centeno et al., 2014), neuroscience imaging (Strother et al., 2004) and infrared astronomy (Ott and Agency, 2010). While these pipelines often log the start and finishing times of each step (using tools such as pegasus-kickstart (Vöckler et al., 2006)), they do not collect detailed time series performance data throughout the run.

At a typical compute cluster the performance of every node in a distributed systems is monitored using utilities, such as Ganglia (Massie et al., 2004). These tools only monitor the global system performance. If one is interested in specific processes, then the Linux procfs (Bowden, 2009) is used. The procfs system can be used to analyse the performance of individual pipeline steps. Likewise, the Performance API (PAPI) (Mucci et al., 1999) is a tool which collects detailed low level information on the CPU usage per process. Collecting detailed statistics at the process level is required to understand the performance of the LOFAR pipeline.

The Linux procfs system and PAPI record data are already made available by the Linux kernel. This option incurs insignificant overhead as it uses data the kernel and processor already log. Likewise PAPI reads performance counters that the CPU automatically increments during processing. These profiling utilities can run concurrently with the scientific payload without using more than 1-2% of system resources. Their low overhead is why we choose to use them to collect performance data.

Other tools for performance analysis such as Valgrind (Nethercote and Seward, 2007) collect very detailed performance information. This comes at the expense of execution time: running with Valgrind, the processing time slows by up to two orders of magnitude. As such, we do not use Valgrind along the LOFAR software.

2. Measuring LOFAR Pipeline performance

We extended the performance collection package *tcollector*[3] to allow collecting performance data for complex multi-step pipelines. The resulting performance data was recorded in a database and analyzed.

Tcollector is a software package that automatically launches 'collector' scripts. These scripts are typically python scripts responsible for monitoring an aspect of the system performance. They sample the specific system resource and send the data to the main tcollector process. This process then sends the data to the dedicated time series database. We have extended this package by creating custom scripts to monitor processes launched by the LOFAR pipeline (Appendix A.1).

In this work, we have used a sample LOFAR SKSP data set with the LOFAR *prefactor* pipeline as a test case. A particular focus was to understand the effect of hardware on the bottlenecks of the LOFAR data reduction. To gain insight into the effect of hardware on *prefactor* performance, the LOFAR data was processed on four different hardware configurations (Table 1). A typical upgrade cycle for cluster hardware is five years. Our results will be used to select optimal hardware for future clusters tasked with processing LOFAR data.

2.1. Performance suite

Cluster performance is frequently monitored using utilities such as Ganglia (Massie et al., 2004) (Discussed in Section 1.1). These tools cannot access individual processes and thus cannot collect data on a per-process basis. To collect such data, each process launched by the active pipeline step needs to be profiled individually. Our utility is designed to gather such performance data.

Using the performance collection framework *tcollector*, custom performance collectors (Appendix A.1) were added to monitor individual pipeline steps as defined by the user[4]. The tools attach to processes launched by the pipeline and record performance data at a one second interval. This sampling frequency is at high enough resolution to detect trends and anomalies in hardware utilization, and still result in a reasonable database size. The performance data is uploaded to a remote time series database, OpenTSDB (Sigoure, 2012). Details on the data collection can be found in Appendix A.

[2] available at `http://www.github.com/lofar-astron/prefactor`

[3] https://github.com/OpenTSDB/tcollector

[4] https://gitlab.com/apmechev/procfs_tcollector.git

2.1.1. Performance API

The time-series database is also used to collect low-level CPU information for each process. This information is collected by the PAPI interface (discussed in Section 1.1). This was done through the papiex utility[5] (Ahn, 2008). This utility records the CPU's internal performance counters. A CPU's performance counters record information on how efficiently the software uses the CPU's resources. The results from this test are detailed in Section 3.3.

2.2. Prefactor Pipeline

The LOFAR *prefactor* pipeline (Van Weeren et al., 2016) is a software pipeline that performs direction independent calibration using the LOFAR software. The LOFAR software stack is a software package containing commonly used processing software used by LOFAR pipelines (Dijkema, 2017)(Offringa et al., 2013). These tools are built and maintained by ASTRON (ASTRON, 2018).

The *prefactor* pipeline performs a sequence of two stages, namely the calibrator and target calibration. The first stage processes data from a calibration source and the second stage processes a science target. Altogether, this processing takes six hours on a high-throughput cluster. The final result is a data-set ready for creating images of the sky at radio wavelengths. Figure 1 shows a graphical view of the prefactor pipeline's Calibrator and Target stages.

The Calibrator stage consists of the *ndppp_prep_cal* and the *calib_cal* step. The former flags radio interference and averages the data, and the latter performs gain calibration on a bright calibration source. It is followed by the *fitclock* step which fits a clock-TEC model to the calibration solutions (Van Weeren et al., 2016).

The Target stage consists of a *ndppp_prep_targ* step, *predict_ateam* *gsmcal_solve* and *gsmcal_apply* steps. The first two of these steps flag and average the target data and calculate contamination by bright off-axis radio sources. The *gsmcal_solve* step determines phase solutions for each antenna using a model of the target and the results of the *ndppp_prep_targ* step. Finally, the *gsmcal_apply* step applies these solutions to the target data. Figure 2 shows the percentage of time spent by these steps for the four *prefactor* stages.

2.3. Test Hardware

In order to study the effect of different hardware configurations on the performance of LOFAR processing, the *prefactor* pipeline was run on four different sets of hardware. These four machines were located at three computational clusters tasked with processing LOFAR data and a personal computer. The tests were run while the systems were idle to make sure there is no interference of other software with the LOFAR processing. Table 1 details the specifics of the four test machines.

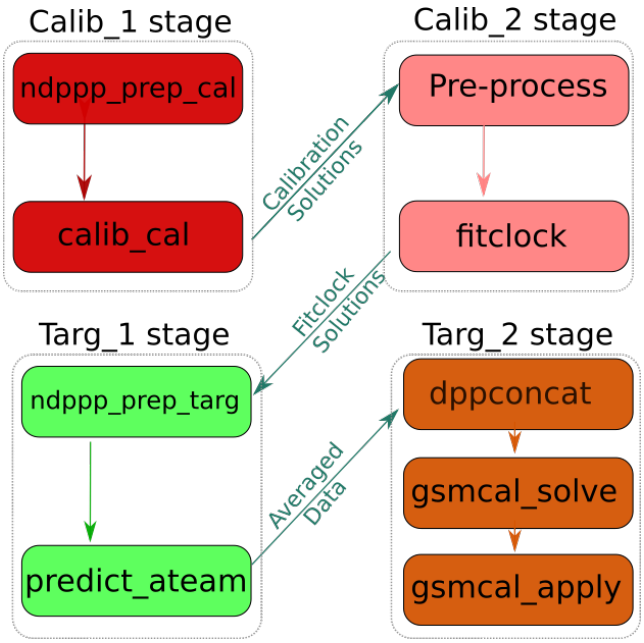


Figure 1: The four processing stages that make up the prefactor pipeline. The Calibrator stages (top) process a known bright calibrator to obtain the gain for the LOFAR antennas. The Target stages (bottom) process the scientific observation to remove Direction Independent effects.

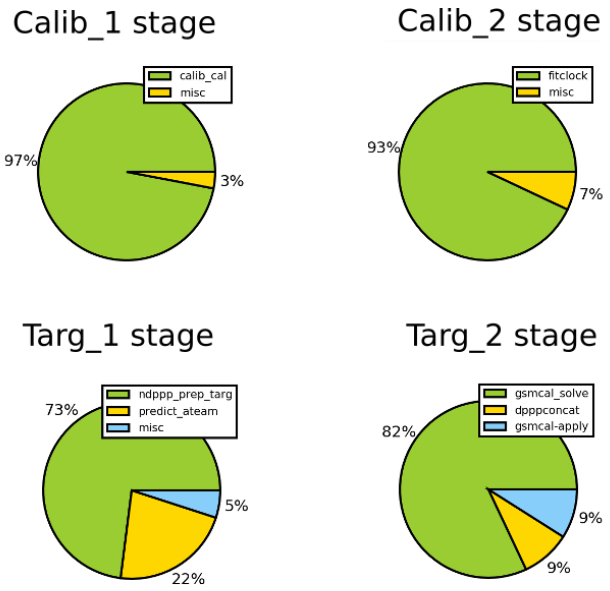


Figure 2: Portion of processing time taken by each step for the four prefactor stages, as reported by the Prefactor software. For each stage, the majority of processing time is spent during one or two steps. This is due to the fact that each prefactor stage also has intermediate book-keeping steps explicitly included in the pipeline.

[5] Available at https://bitbucket.org/minimalmetrics/papiex-oss

Location	CPU Speed (MHz)	Cache Size	RAM Speed[6]	Disk Speed[7]
Leiden	2200	16 MB	1.4 GB/s	99.7 MB/s
Amsterdam	2500	30 MB	2.5 GB/s	65.4 MB/s
Hatfield	2900	20 MB	2.4 GB/s	155 MB/s
Laptop	3300	8 MB	4.7 GB/s	822 MB/s

Table 1: CPU, Cache, RAM and Storage specifications of the four test machines. The tested machines span a factor of 1.5x in CPU speed, 4x in cache and RAM Speed and 10x in Disk speed.

3. LOFAR Prefactor Test Case

With the test set described in Section 2, we aim to understand processing bottlenecks in the *prefactor* pipeline and make informed decisions on future hardware and software upgrades. To do so, we processed a sample observation at institutes that typically process LOFAR data.

From the data collected by processing the sample observation, we determined the slowest pipeline steps. These steps were the *calib_cal* and *gsmcal_solve*, seen in Figure 2. The *calib_cal* step is implemented by the software 'bbs-reducer' (Dijkema, 2017)(Loose, 2008) and the *gsmcal_solve* step is implemented by 'ndppp' (Dijkema, 2017)(Offringa et al., 2013). Both bbs-reducer and ndppp are part of the LOFAR software suite.

We collected performance statistics using the *pipeline_collector* suite as discussed in Section 2. The results discovered using this software are listed in Table 2 and discussed in Section 3.2. Using the PAPI interface (discussed in Section 2.1.1) CPU performance data was collected. The results from this test are detailed in Section 3.3.

We will present a number of insights into the performance of the LOFAR software collected by the profiling suite. The results are presented in Table 2 and are grouped in three main areas. The effect of compilation on the runtime was result **R1**. The set of results (**R2**, **R3**, **R4** and **R5**) were obtained using the *tcollector* package while results **R6**, **R7**, **R8** and **R9** were collected with the PAPI package.

3.1. Pre-compiled vs native compilation

The performance trade-off between pre-compiled and native compilation was studied first. The majority of the processing for the LOFAR SKSP Project (Shimwell et al., 2017) is done at the SURFsara GINA cluster in Amsterdam. This location is part of the European Grid Initiative (EGI)(SURF, 2018). At this location, software is deployed by compiling on a virtual machine and mounting it on all worker nodes through the CernVM FileSystem (CVMFS) service (Aguado Sanchez et al., 2008). The CVMFS server allows any client to mount a fully compiled LOFAR installation, making it easy to distribute and version control the software within and outside of SURFsara. An alternative is to locally compile the LOFAR packages on each cluster. The performance of the natively compiled vs CVMFS installations was compared on the laptop test machine using *pipeline_collector*. In order to validate this result, the two compilations of the same software were also tested at the Data Science Lab at the Leiden Institute of Advanced Computer Science[9].

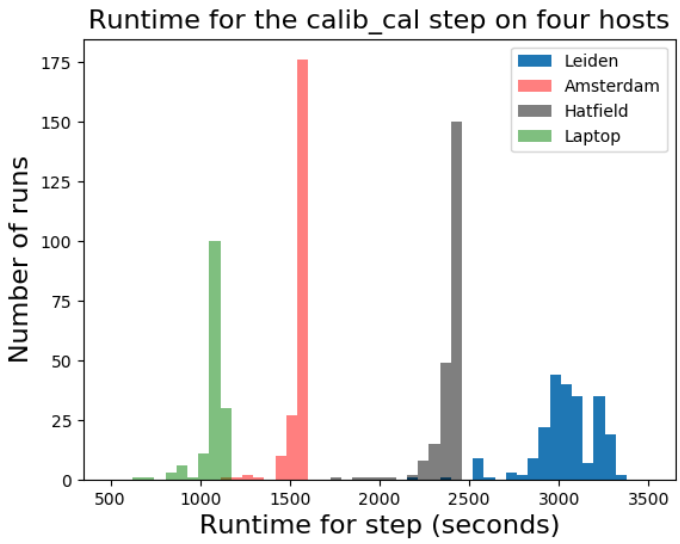
An interesting discovery is that the LOFAR software did not process data faster when compiled natively. This is despite the fact that the local install was compiled with advanced processor instructions available on the host machine. Figure 4 shows a histogram of its processing time with the two different compilation options for the *calib_cal* software running on the sample dataset. The same test was done for the software performing the gain calibration (*gsmcal_solve*), seen in Figure 5. The result of this experiment, shown in Figures 4a and 5a, was that the software for both steps show no noticeable improvement when the software was natively compiled. This is result **R1** in Table 2. The second run at LIACS also confirms this result for both steps (Figures 4b and 5b). This result suggests that the slowest *prefactor* steps are not optimized for modern processors.

3.2. Prefactor Runtime and Hardware Parameters

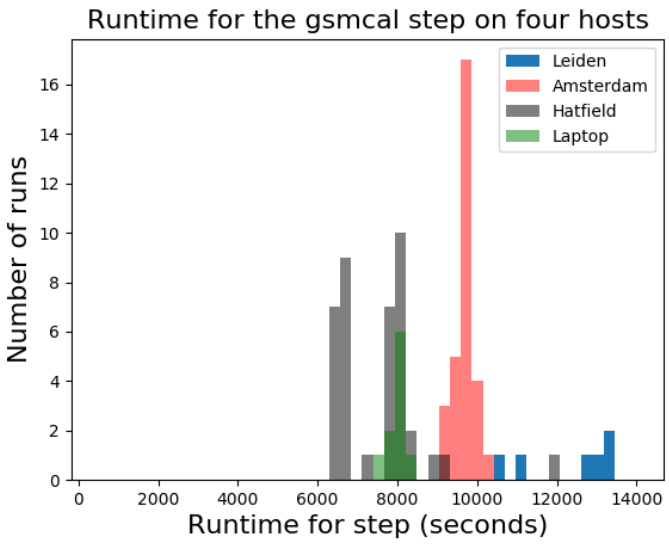
Next, we studied the dependence of runtime on different hardware parameters. With software that collects per-step performance statistics for the LOFAR pipeline, the dependence of the pipeline processing on hardware performance can be easily profiled and studied. Using *pipeline_collector* we determined the pipeline's slowest steps with respect to different hardware parameters.

The system parameters studied here are the CPU speed, memory throughput, cache size and disk speed. Modern computers can have a complex memory hierarchy as demonstrated in Figure 6 (Katz and Patterson, 2001). This is due to the cost trade-off between memory size and memory speed. Because of this trade-off, the full dataset is stored on disk, while the working set is placed in RAM. This is the data that the processor needs to access at the current time (Denning, 1968). The most frequently accessed parts of the data are stored in the CPU cache, which evicts the oldest data when full (Hazelwood and Smith, 2004).

[9] https://www.universiteitleiden.nl/en/science/computer-science/about-us/our-facilities

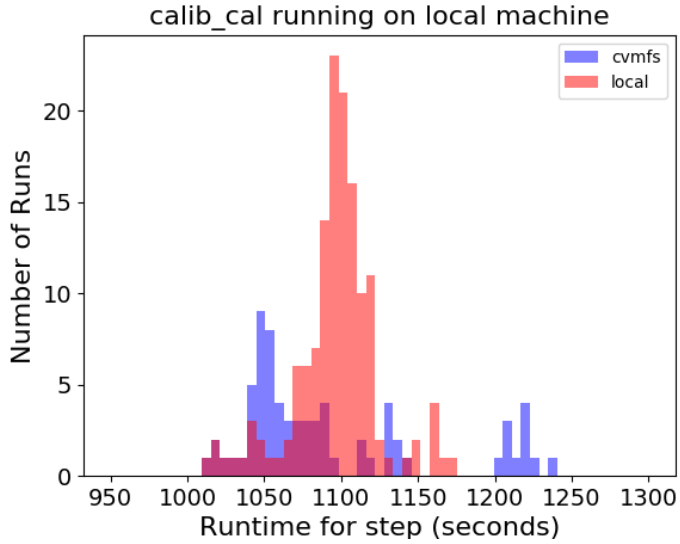


(a) calib_cal

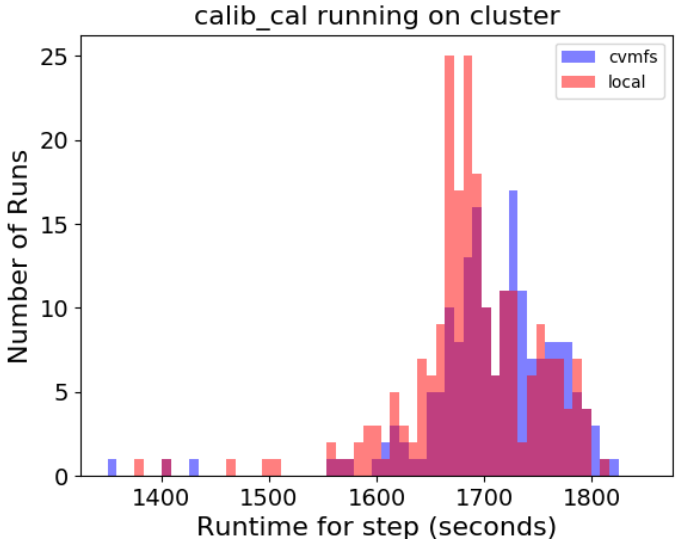


(b) *gsmcal_solve*

Figure 3: Job completion times for *calib_cal* and *gsmcal_solve* steps tested on four hardware setups. The *calib_cal* step ran 244 times. The *gsmcal_solve* ran 24 times as the data is concatenated from 244x1 to 24x10 sets. The step with the longest latency is *gsmcal_solve* while *calib_cal* consumes a comparable number of core-hours over 244 jobs.



(a) Two compilation options on a Laptop



(b) Two compilation options on cluster node

Figure 4: Difference in processing time for *calib_cal* when compiled remotely and natively. *calib_cal* was run 244 times with the native software and 40 times with the CVMFS compilation. Two tests were done, one on the personal laptop (4a) and one on a cluster node at the LIACS Data Science Lab (4b). Both plots show there is no difference in runtime between compilation options.

Result #	Description
R1	Native compilation of the software performs comparably to pre-compiled binaries on two test machines.
R2	The processing steps do not appear to accelerate significantly on a faster processor or with larger cache size.
R3	Both calibration steps[8] show linear correlation between speedup and memory bandwidth.
R4	Disk read/write speed does not affect the completion time of the slowest steps.
R5	Both calibration steps do not use large amounts of RAM despite processing data on the order of Gigabytes.
R6	The *calib_cal* step can suffer up to 20% of Level 1 Instruction Cache misses, while gsmcal only has 5% of these misses.
R7	Both calibration steps are impacted by Level 2 Cache eviction at comparable rates.
R8	The *calib_cal* step stalls on resources often while the gsmcal step less so.
R9	The *calib_cal* step does not execute CPU instructions efficiently.

Table 2: A table of all the results presented in this section.

The CPU processing speed is faster than the RAM latency, so a system of caches exist (Figure 6). Caches store small subsets of the working set and have a fast connection to the processor. The fastest data link is between the CPU and the L1 Cache, with the link to RAM being slower and the disk read speed slower still. The limited memory capacity of the different levels of the memory hierarchy as well as the throughput between them will lead to performance bottlenecks. These bottlenecks will lead to the processor waiting on memory. Such stalls lead to longer processing times.

3.2.1. CPU

Here, we will discuss the effect of CPU speed on the LOFAR pipeline. The CPU speed is usually the primary factor determining how fast computations can be made. In general, a faster CPU will result in faster data processing.

However, Fig. 7a shows that the runtime of the calibration of the calibrator does not strongly depend on the CPU frequency. While the test nodes at Amsterdam and Leiden run at the same CPU frequency, running on a cluster node at Amsterdam takes half the time as on a node at Leiden. Even more surprisingly, the *gsmcal_solve* step does not benefit significantly from a faster CPU, despite being the most computationally heavy prefactor step (**R2** in Table 2). This step does the gain calibration on the target field using the stefcal algorithm (Salvini and Wijnholds, 2014). Figure 7b shows only a slight improvement over faster CPU clock speeds for both steps. The correlation between completion time and CPU speed is similar for both steps.

3.2.2. Cache

Next, we will discuss the effect of cache on performance. The CPU has a hierarchy of caches consisting of Level 1, Level 2 Cache and LLC Cache. For the four processors tested, the Level 1 and 2 caches were all the same size, thus the only difference is the Last Level Cache (LLC or just Cache on Figure 6). This cache stores data needed by the CPU, so the larger it is, the less the processor needs to wait for RAM to return data.

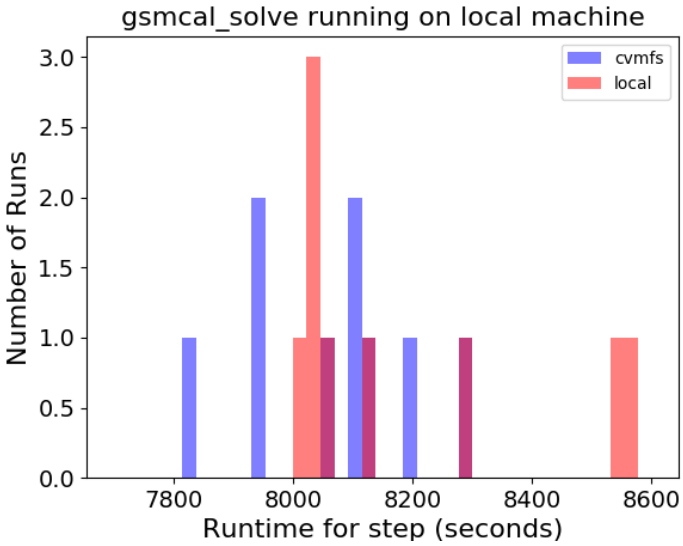
In general, numerical codes benefit from larger cache sizes (Skadron et al., 1999)(Goto and Geijn, 2008). Interestingly, figure 8b suggests that the *gsmcal_solve* step does not exclusively depend on larger cache **R3** (Table 2). On the machines with a larger cache, the *gsmcal_solve* step completed processing as quickly as on the machines with smaller cache, even down to 8MB.

3.2.3. RAM Bandwidth

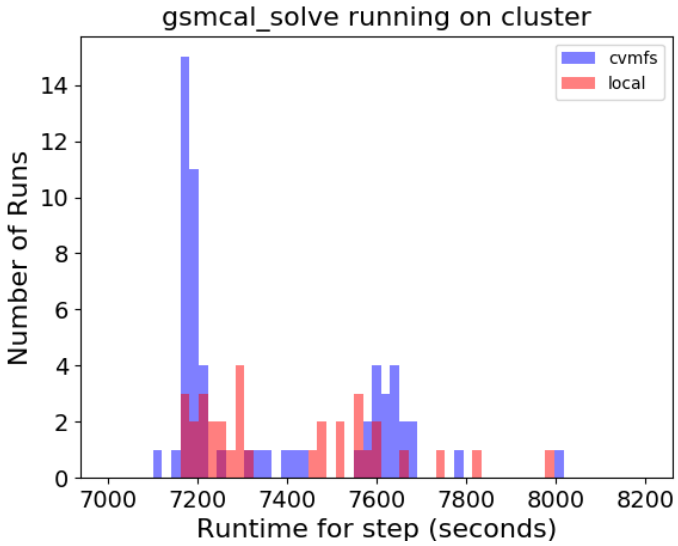
Here we discuss the next step of the memory hierarchy, the RAM bandwidth. If the entire data set does not fit into cache, the software needs to transfer data from RAM to the CPU. In these cases, the *prefactor* step benefits from a fast bandwidth between the cache and RAM. For this study, the RAM throughput was benchmarked[10] using the Linux *dd* utility (Rubin et al., 2010). This command copies dummy data into system memory. As this utility exists on all Unix systems, this is a standardized benchmark of the RAM performance.

Figure 9a showed that higher bandwidth is correlated with a faster completion time for the *calib_cal* and *gsmcal_solve* steps (**R4**). The result is to be expected as the working set of these steps is 200MB and 1.0GB respectively, and cannot fit into cache readily, however it is loaded into RAM within the first 5 seconds of the run (Figure 11), and is streamed from memory throughout the run.

[10]Using the command `$> dd if=/dev/zero of=/dev/shm/test bs=1M count=2048`



(a) Two compilation options on a Laptop



(b) Two compilation options on cluster node

Figure 5: Difference in processing time for *gsmcal_solve* when compiled remotely and natively. *gsmcal_solve* was run 50 times with the native software and 120 times with the CVMFS compilation. Two tests were done, one on the personal laptop (5a) and one on a cluster node at the LIACS Data Science Lab (5b). Just like with the *calib_cal* step, the *gsmcal_solve* step also doesn't accelerate when natively compiled.

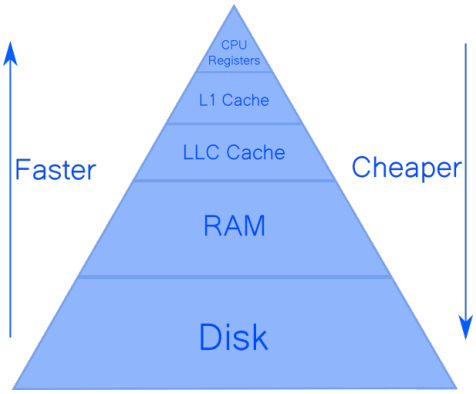


Figure 6: A model of the memory hierarchy. (Goto and Geijn, 2008)

3.2.4. Disk Read speeds

The slowest link in the memory hierarchy is the disk read speed. For the *calib_cal* step, the entire data is loaded into memory during the first few seconds of the run, after which the disk only becomes important when the results need to be written out. The *gsmcal_solve* step streams data from the disk to memory throughout the entire run. The plot of disk read speeds (Fig. 10b) also shows that a faster disk does not speed up the slowest step **R5**. To verify that disk throughput was not the limiting factor, the entire dataset (25 GB) was moved to main memory (using /dev/shm). The resulting runtime for both bottleneck steps did not change.

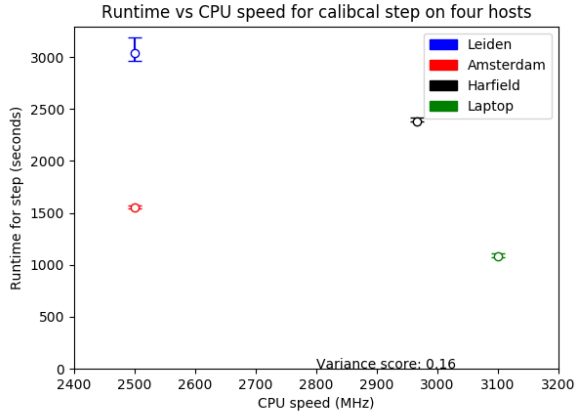
The calibration steps both stored less than 200MB of data in memory throughout their run. Figure 11 shows the time-series of the total memory used by these steps. The *calib_cal* step uses only 200MB of memory and *gsmcal_solve* only 35MB. While the *gsmcal_solve* step works on a 1GB dataset, it streams the data in memory and thus does not require 1GB of RAM. Alternatively, the *calib_cal* step loads the entire (200MB) dataset into memory for the entire duration of the run. The RAM usage time-series in Figure 11 show that the RAM is filled for the first 5 seconds of the run, meaning that the processing is effectively independent from disk speed.

3.3. CPU Utilization

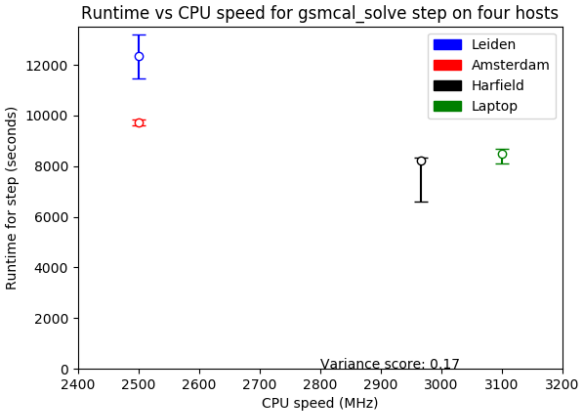
To gain more fine grained data on the CPU utilization, the *calib_cal* and *gsmcal_solve* steps were tested with the PAPI package. We ran this package as a test, and will include it in the *pipeline_collector* suite in the future. PAPI collects data from low-level CPU counters for the a specific thread and can record data such as cache performance, branch prediction rate, fraction of memory/branch instructions and others. This data is complementary to the procfs information, which is collected by the Linux kernel. In the following sections we will discuss the differences in Cache performance between these two steps.

3.3.1. Level 1 Data Misses

The Level 1 Cache is split into cache for instructions and data. For all our test hardware the L1 Data cache is 32 kB, and has a direct link to the processor's computational units (Jain and Agrawal, 2013). The processor collect information logging how many times data requested by the CPU is not located into the L1 Data cache. This counter is called the Level 1 Data Cache Miss rate. To resolve this type of cache miss, the data needs to be fetched from L2 Cache. When this happens, the processor has to wait for

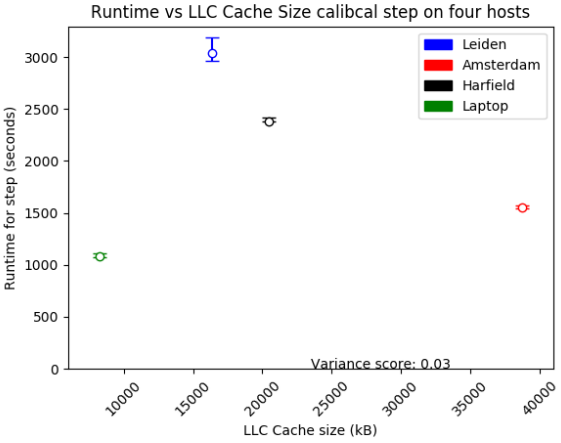


(a) calib_cal

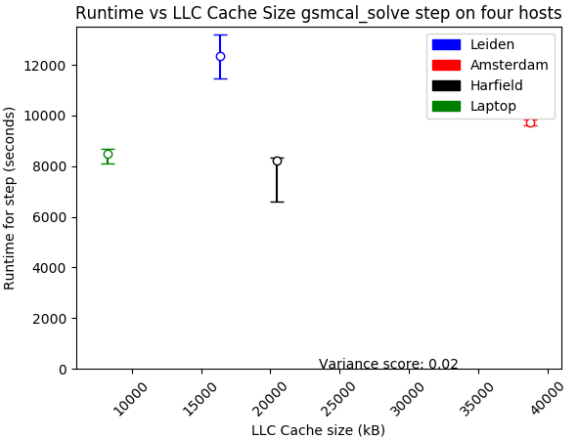


(b) *gsmcal_solve*

Figure 7: Performance of the bottleneck steps compared with the CPU speeds of the four test machines. The values are the mean of 244 runs (Standard prefactor run) and the error bars show the 1-sigma of the distribution of the run time.

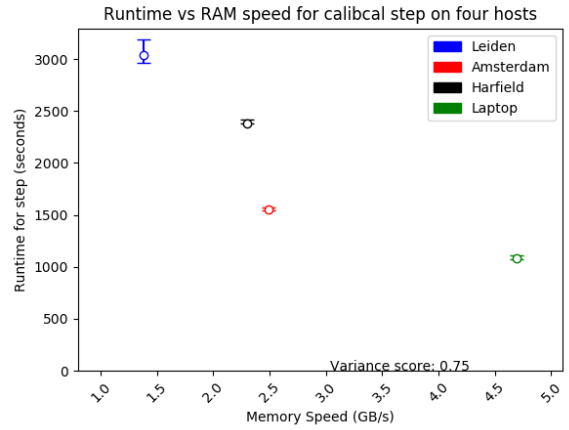


(a) calib_cal

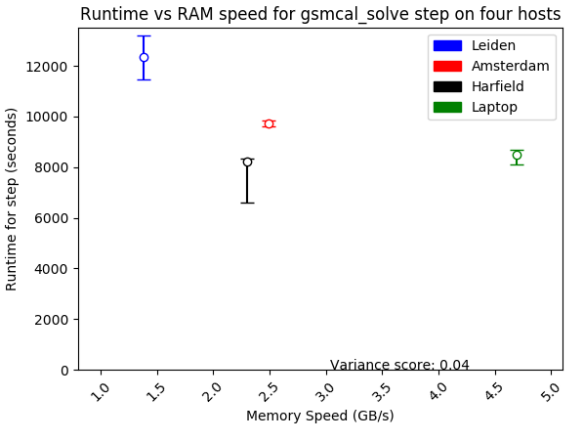


(b) *gsmcal_solve*

Figure 8: Performance of the two bottleneck steps with respect to Last Level Cache size. The *gsmcal_solve* step shows no trend between cache size and completion time. The *calib_cal* step runs the fastest on the machine with the smallest cache.

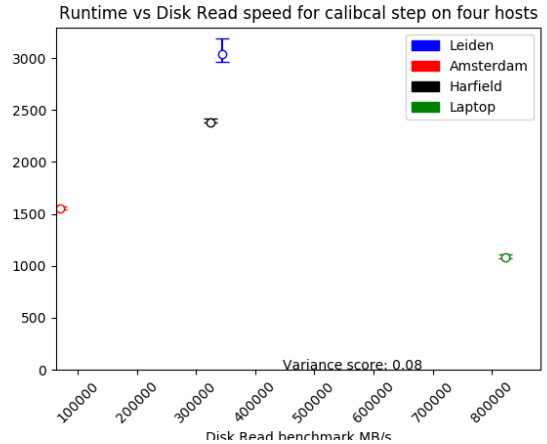


(a) *calib_cal*

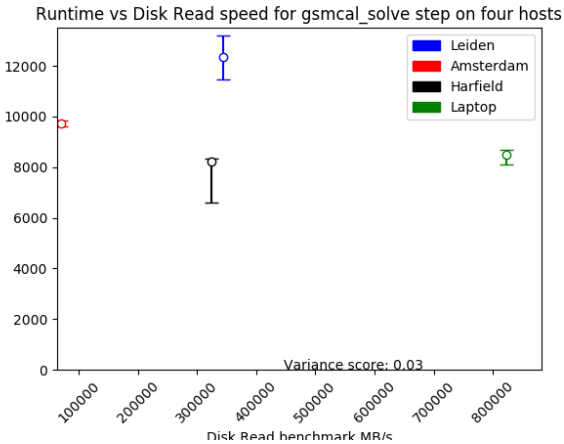


(b) *gsmcal_solve*

Figure 9: Performance of the two bottleneck steps and RAM bandwidth in GB/s. Both the *calib_cal* and *gsmcal_solve* steps show a trend of faster processing times on machines with higher RAM bandwidth.

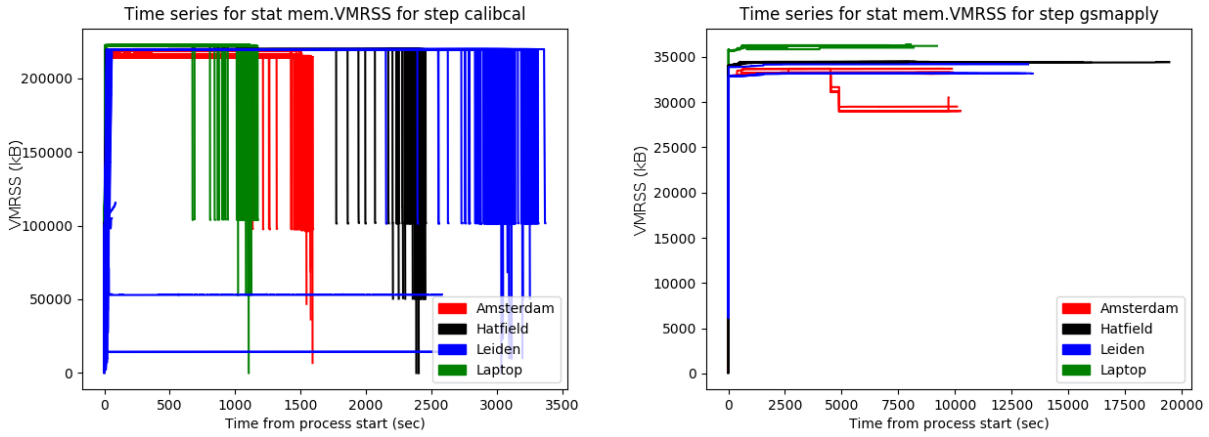


(a) calib_cal



(b) *gsmcal_solve*

Figure 10: Performance of the two bottleneck steps and Disk bandwidth in MB/s. There is no correlation between the Disk read speed and the Runtime of the steps.



(a) *calib_cal* (b) *gsmcal_solve*

Figure 11: Time series of the Virtual Memory Resident Set Size . This is the amount of data stored in RAM (in kB) during the *calib_cal* and *gsmcal_solve* steps. Both steps show the same amount of memory use on all test machines. Additionally, after a brief loading of data, the memory usage remains constant until processing is finished.

the requested data. **R7**: The recorded L1 data misses in Figure 12a, show that the software performing the *calib_cal* step misses 20% of its L1 data cache requests, while the the software implementing the *gsmcal_solve* step misses less than 5% of L1 Cache requests. These cache misses often happens in multi-threaded applications where there are instructions shared by multiple threads on the same cache line (Sarkar and Tullsen, 2008).

3.4. Level 2 Instruction Misses

Unlike the Level 1 cache, Level 2 cache stores data and instructions in the same location. When the cache is full, it evicts the last used element in order to make space for newly requested data. PAPI also counts these eviction events. Figure 12b shows that for both steps, between 50 and 70% of L2 requests for an instruction do not match the contents of L2 Cache. This is significantly more than the applications benchmarked in (Lebeck et al., 2002) (Table 2). Because both steps process data of considerable size, the large amount of data required can evict instructions from the L2 cache (insight number **R7** in table 2).

3.5. Resource Stalls

Modern processors have multiple computational pipelines on chip, in order to process data in parallel (Hu et al., 2006). There are times when the processor's internal pipeline needs to wait for other instructions to finish. When this happens, it flags that it has 'stalled on a resource'. These resource stall cycles are also recorded by PAPI and represented as a percentage of total cycles. From figure 13a, it can be seen that *calib_cal* stalls on 70% of the processor cycles, while *gsmcal_solve* only on 33% of cycles (**R8**).

The Full Issue Cycles counter indicates the percentage of processor cycles, in which the theoretical maximum number of instructions are executed. During these cycles, the software uses the CPU optimally.The full issue cycles counter (Fig. 13b) also shows the difference in efficiency between the *calib_cal* and *gsmcal_solve* step (**R9**), with the former only working at peak efficiency for 10% of the processor cycles.

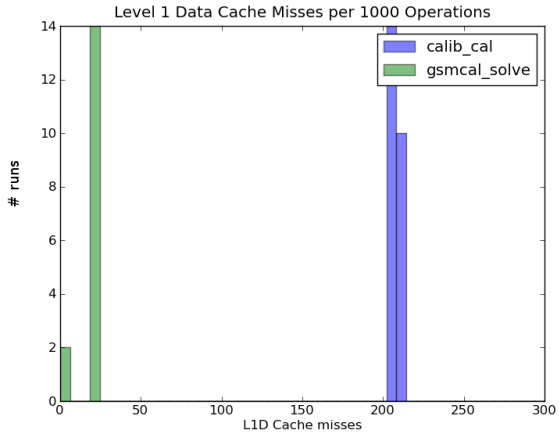
The plots in Figures 13a and 13b indicate that the *calib_cal* step does not use the internal CPU pipelines efficiently leading to waiting on resources and sub-optimal use of the CPU's Computational Units.

4. Discussions and Recommendations

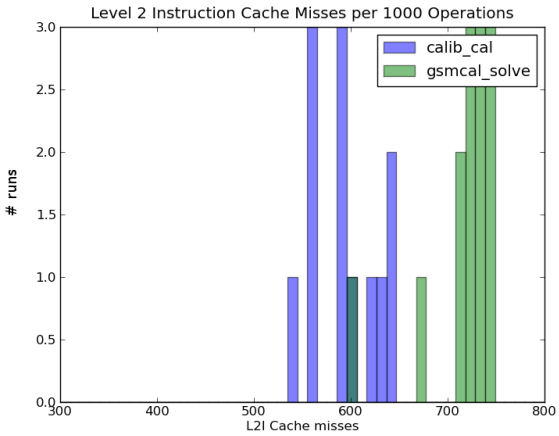
With an increase of data acquisition rates and data complexity in radio astronomy, it is becoming important to thoroughly understand and optimize the performance of processing pipelines. Using our pipeline monitoring package *pipeline_collector*, data can be collected for each pipeline step and stored in a time-series database. This database can be studied to help researchers understand the pipeline performance in different scenarios and on different hardware. The *pipeline_collector* suite is easy to deploy for mature pipelines and has minimal impact on pipeline performance. Typical CPU usage is <0.2% with a memory footprint of $\sim$ 1-10 MB.

Creating a performance model with the collected data will allow to optimize future clusters for LOFAR data processing. This optimization will allow for optimization of future LOFAR clusters. Doing so is necessary given the current data throughput, number of observations and time-line of the SKSP project. Similar issues will be encountered with upcoming radio telescopes (Broekema et al., 2015).

To showcase the power of the pipeline_collector suite, the LOFAR *prefactor* pipeline was run through a single data set on three clusters and a personal machine. A

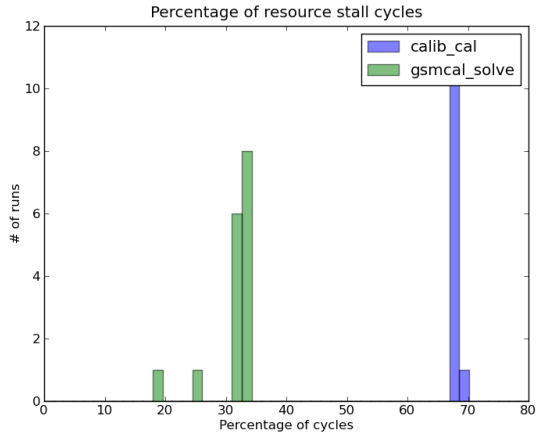


(a) Level 1 Data cache misses

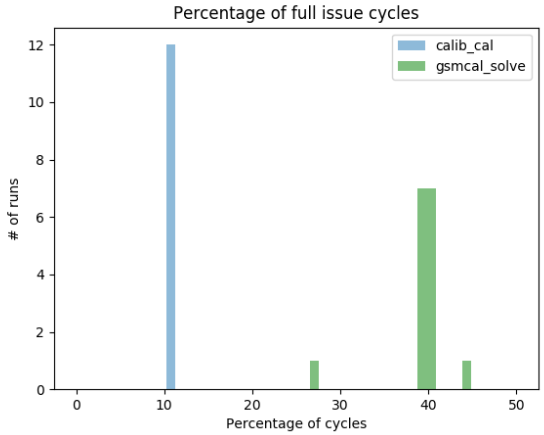


(b) Level 2 Instruction cache misses

Figure 12: Cache miss rates for *calib_cal* and *gsmcal_solve*, executed on the SURFsara gina cluster. The cache is split into instruction and data caches. The figures above show the difference in number of cache misses for both instruction cache and data cache for the slowest *prefactor* steps. *calib_cal* suffers significantly more Data cache misses than *gsmcal_solve* while the two steps undergo similar instruction Cache misses.



(a) Resource Stall Cycles



(b) Full Issue cycles

Figure 13: Resource stall cycles and Full Instruction Issue cycles. The two steps were executed on the SURFsara gina cluster.

number of insights were made using the high resolution timing data collected from this package (such as in Figure 11)and are listed in Table 2. In the future, we'll apply the pipeline_collector software to the more complex LOFAR DD pipeline, *ddf-pipeline*.

The slowest processing steps for the *prefactor* pipeline were identified as the *calib_cal* and *gsmcal_solve* steps. While the data can fit into the RAM for all of the processing machines, it is much larger than the processor's internal cache (Figure 6). The discoveries made concerned the memory hierarchy in Figure 6. Results labeled **R2**, **R8** and **R9** related to the CPU performance; **R2**, **R6** and **R7** related to the Cache performance; **R3** and **R5** related to the Memory usage and **R4** discussed the Disk speed.

Faster processors did not accelerate the *gsmcal_solve* step significantly, as this step streams data between the RAM and CPU. As the CPU speed increases, streaming applications become bottlenecked by the throughput of data into the CPU from RAM. As the *gsmcal_solve* algorithm iteratively calibrates chunks of the data, these chunks need to be loaded from disk once, however they are moved from RAM to CPU multiple times during calibration.

Similarly, the *calib_cal* step is more dependent on memory throughput than on CPU speed as this step moves data to and from memory frequently. This step also does minimization looping over the dataset. As the dataset does not fit in the cache, parts of it need to be constantly moving from memory and back.

4.1. Recommendations

Based on these results, the top hardware recommendation is that *prefactor*'s slowest steps can be accelerated by running on machines with faster memory or upgrading the memory of the current machines. The two slowest *prefactor* steps showed improvements on machines with faster RAM.

One software recommendation is to improve the efficiency of the *calib_cal* step though refactoring or by replacing the software package used. Unfortunately, the software used for the *gsmcal_solve* step cannot be used for the *calib_cal* step as it is not yet able to correct for Faraday Rotation (Salvini and Wijnholds, 2014), making it impossible to currently use the software used by the *gsmcal_solve* step. Faraday Rotation has recently been implemented in a development version of the *prefactor* pipeline and is currently undergoing testing.

Additionally, the large number of data cache misses recorded for the *calib_cal* step suggests that its source code is not optimized for multi-threaded processing. Data cache misses are often encountered when multiple threads have instructions on the same cache line[11], forcing the memory controller to move this cache line between cores (Lebeck et al., 2002). This can also explain the large number of stalled cycles (Fig. 13a) and low number of full issue cycles (Fig. 13b) for the *calib_cal* step.

Finally, we discovered that compiling the software on a virtual machine did not lead to a processing slowdown. This means that the current slowest *prefactor* steps are not optimized to use advanced processor instructions. Nevertheless, the resulting cross-compatibility is an encouraging result as it will allow to easily distribute pre-compiled versions of the software without increasing the processing time.

5. Conclusions

In this paper, we present a novel system for automated collection of performance data for complex software pipelines. We use this method to study the LOFAR *prefactor* pipeline. The results are discussed aiming to understand the effect of different hardware parameters on the data processing. To do so, we run the pipeline on four different machines.

The software automatically collects performance data at the operating system level. Data for each pipeline step can be extracted using the OpenTSDB API, plotted and analyzed. Additionally, the *pipeline_collector* suite is easy to extend with new collectors that record more detailed time-series data for each pipeline step. The performance data is stored in the time series database OpenTSDB. We used this data to find 9 insights into the LOFAR prefactor pipeline listed in Table 2. The implementation details are described in Appendix A.

The *prefactor* pipeline is used to do the initial processing for over 3000 observations that are part of the LOFAR SKSP Tier 1 survey. However, this pipeline is also used for lots of other LOFAR datasets outside the SKSP project. We show that increasing the RAM throughput is the easiest way to speedup *prefactor* processing. Running the *calib_cal* step on hardware with RAM faster than 4 GB/s will save up to 700k CPU hours for the 3000+ unprocessed datasets. This throughput increase will also speedup the *gsmcal_solve* step by 30% saving an additional 400k CPU hours. This is a significant fraction of the estimated 2,400k CPU hours required to process this data with the *prefactor* pipeline.

As shown in this work, we can correlate the performance of the LOFAR software with different hardware specifications. Additionally, the datasets can vary in size and job overheads on the compute cluster can depend on the processing parameters. All of these parameters affect the processing latency for the calibration and imaging pipelines. As such, a thorough parametric model is required to further optimize the end-to-end LOFAR processing pipeline and predict reduction times on future clusters.

The design of this utility makes it easy to apply to future scientific pipelines. In future work, we will use *tcollector* to create a performance model of the full LOFAR

[11] A cache line is a row of cache memory which is loaded into CPU as a single unit (David and John, 2005)

imaging pipeline(Van Weeren et al., 2016), including the DI and DD steps. This model will make it possible to efficiently optimize the LOFAR processing, understand the LOFAR pipeline and suggest for hardware and software improvements.

Appendix A. Performance Collection Implementation Details

The *tcollector* package is a python software suite that can collect system performance data at predetermined intervals. The package is designed to monitor the performance statistics for web-servers and cluster nodes. The tcollector software records time series of the different performance metrics and sends them to a Time Series Database through HTTP. The Time Series Database, OpenTSDB stores the data in an HBase (Apache HBase, 2015) instance at the performance collection server. Users interested in plotting time series can plot real time or historical data through a HTTP interface with OpenTSDB. With a central performance collection server, data from multiple processing sites can be collected and analyzed.

Tcollector formats the time series information in four fields. First is the name of the metric which is measured. Second is the UNIX timestamp. Third is the time series recorded as an integer or a float. Finally, a set of tags (key-value pairs) can be added to the data point. These four fields are discussed below and can be seen on the right side of Figure A.14.

Appendix A.1. The pipeline_collector suite

The tcollector package cannot collect data on individual processes, nor can it associate these processes with specific steps of a data processing pipeline. We've supplemented the software with the pipeline_collector suite[12] using an executable that monitors a pipeline's running processes[13]. When an executable that is part of the LOFAR pipeline launches, a dedicated collector begins reporting information on the individual process. The software determines the current processing step by parsing through the output of the LOFAR pipeline. Running the LOFAR processing concurrently with the tcollector package gives us per-step performance data without changing or slowing down the LOFAR *prefactor* pipeline.

The pipeline_collector suite sends data to the time series database in the same format as the rest of the collectors included in the tcollector package.

Appendix A.2. Setting up for future pipelines

The setup options for pipeline_collector are stored in a configuration file in the root directory of the package. This file holds the sample interval, executables to monitor and the location where pipeline_collector can read the current pipeline step

The pipeline_collector suite reads the current pipeline step from a file, the location of which is specified in the configuration. This file needs to be updated each time the pipeline begins a new step. For LOFAR we have a script running with the pipeline, parsing the output and determining the current step. As each pipeline has a unique sequence of steps, the current step needs to be recorded in a file in order for pipeline_collector to report it to the time series database. The location of the file recoding the current pipeline step is read from the configuration file.

Next, the names of the specific processes need to be included in the configuration file. In the case of LOFAR, we select the 'NDPPP', 'bbs-reducer' and 'losoto' processes. The pipeline_collector searches the running processes for the current user for these process names and launches a collector for each new process launched by the current step.

[12]Located at https://gitlab.com/apmechev/procfs_tcollector.git
[13]Located at https://github.com/apmechev/procfsamp

Acknowledgements

A.P. Mechev would like to acknowledge the support from the NWO/DOME/IBM programme "Big Bang Big Data: Innovating ICT as a Driver For Astronomy", project #628.002.001.

This work was carried out on the Dutch national e-infrastructure with the support of SURF Cooperative through grant e-infra 160022 & 160152.

HJR gratefully acknowledge support from the European Research Council under the European Unions Seventh Framework Programme (FP/2007-2013)/ERC Advanced Grant NEWCLUSTERS- 321271.

References

C. Aguado Sanchez, J. Bloomer, P. Buncic, L. Franco, S. Klemer, and P. Mato. Cvmfs a file system for the cernvm virtual appliance. In *Proceedings of XII Advanced Computing and Analysis Techniques in Physics Research*, volume 1, page 52, 2008.

D. H. Ahn. Measuring flops using hardware performance counter technologies on lc systems. Technical report, Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2008.

S. F. Apache. Tcollector: Opentsdb documentation. Available at `http://opentsdb.net/docs/build/html/user_guide/utilities/tcollector.html`, 2017.

T. Apache HBase. Apache hbase reference guide. *Apache, version*, 2(0), 2015.

ASTRON. Astron: Netherlands institute for radio astronomy, 2018. URL `https://www.astron.nl/`.

T. Bowden. The /proc filesystem v1.3. `https://www.kernel.org/doc/Documentation/filesystems/proc.txt`, 2009.

P. C. Broekema, R. V. van Nieuwpoort, and H. E. Bal. The square kilometre array science data processor. preliminary compute platform design. *Journal of Instrumentation*, 10(07):C07004, 2015.

P. C. Broekema, J. Mol, R. Nijboer, A. S. van Amesfoort, M. A. Brentjens, G. Loose, W. F. Klijn, and J. W. Romein. Cobalt: A gpu-based correlator and beamformer for lofar. *arXiv preprint arXiv:1801.04834*, 2018.

R. Centeno, J. Schou, K. Hayashi, A. Norton, J. Hoeksema, Y. Liu, K. Leka, and G. Barnes. The helioseismic and magnetic imager (hmi) vector magnetic field pipeline: optimization of the spectral line inversion code. *Solar Physics*, 289(9):3531–3547, 2014.

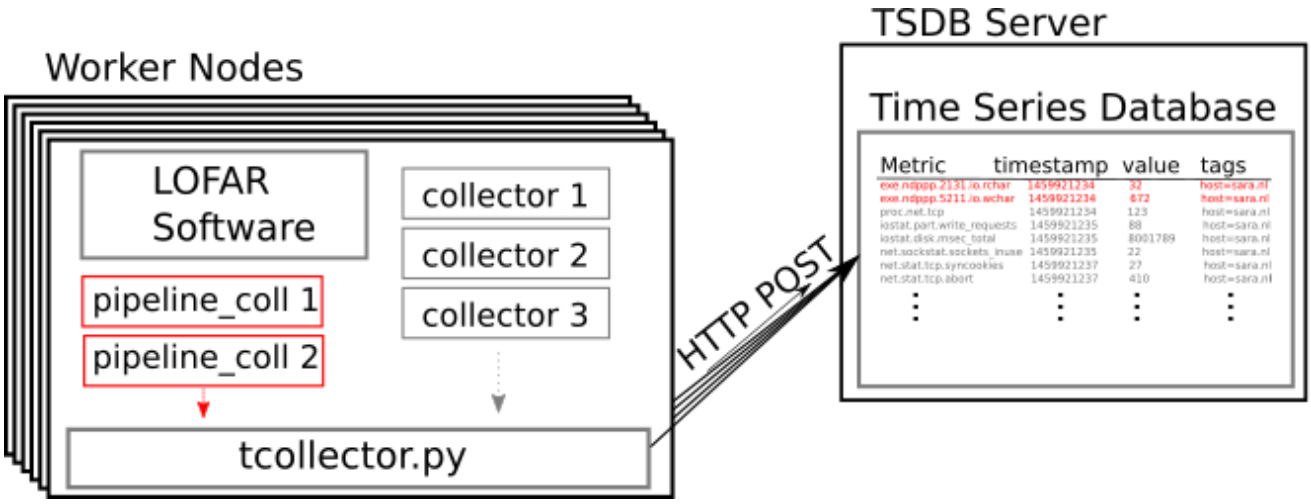


Figure A.14: Communication between worker nodes and the TSDB server, including the pipeline_collector modules (in red). The pipeline_collector suite collects information on the running LOFAR pipelines, while the rest of the tcollector package collects system performance data. The existing tcollector package and its collectors are shown in gray. The collectors in gray only record metrics from the global system.

A. P. David and L. H. John. Computer organization and design: the hardware/software interface. *San mateo, CA: M organ Kaufmann Publishers*, 1:998, 2005.

D. B. Davidson. Potential technological spin-offs from meerkat and the south african square kilometre array bid. *South African Journal of Science*, 108(1-2):01–03, 2012.

P. J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, 1968.

T. J. Dijkema. Lofar imaging cookbook. Available at `http://www.astron.nl/sites/astron.nl/files/cms/lofar_imaging_cookbook_v19.pdf`, 2017.

K. Goto and R. A. Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software (TOMS)*, 34(3):12, 2008.

Y. Gupta, B. Ajithkumar, H. Kale, S. Nayak, S. Sabhapathy, S. Sureshkumar, R. Swami, J. Chengalur, S. Ghosh, C. Ishwara-Chandra, et al. The upgraded gmrt: opening new windows on the radio universe. *Current Science*, 113(4):707, 2017.

K. Hazelwood and J. E. Smith. Exploring code cache eviction granularities in dynamic optimization systems. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 89–99. IEEE, 2004.

S. Hu, I. Kim, M. H. Lipasti, and J. E. Smith. An approach for implementing efficient superscalar cisc processors. In *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pages 41–52. IEEE, 2006.

T. Jain and T. Agrawal. The haswell microarchitecture-4th generation processor. *International Journal of Computer Science and Information Technologies*, 4(3):477–480, 2013.

J. L. Jonas. Meerkatthe south african array with composite dishes and wide-band single pixel feeds. *Proceedings of the IEEE*, 97(8): 1522–1530, 2009.

R. H. Katz and D. A. Patterson. Memory hierarchy, cmput429/cmpe382 winter 2001. university of calgary. Available at `https://webdocs.cs.ualberta.ca/~amaral/courses/429/webslides/Topic4-MemoryHierarchy/sld003.htm`, 2001.

A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. A large, fast instruction window for tolerating cache misses. In *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*, pages 59–70. IEEE, 2002.

C. J. Lonsdale, R. J. Cappallo, M. F. Morales, F. H. Briggs, L. Benkevitch, J. D. Bowman, J. D. Bunton, S. Burns, B. E. Corey, S. S. Doeleman, et al. The murchison widefield array: Design overview. *Proceedings of the IEEE*, 97(8):1497–1506, 2009.

G. Loose. Lofar self-calibration using a blackboard software architecture. In *Astronomical Data Analysis Software and Systems XVII*, volume 394, page 91, 2008.

M. L. Massie, B. N. Chun, and D. E. Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, 2004.

A. Mechev, J. B. R. Oonk, A. Danezi, T. W. Shimwell, C. Schrijvers, H. Intema, A. Plaat, and H. J. A. Rottgering. An Automated Scalable Framework for Distributing Radio Astronomy Processing Across Clusters and Clouds. In *Proceedings of the International Symposium on Grids and Clouds (ISGC) 2017, held 5-10 March, 2017 at Academia Sinica, Taipei, Taiwan (ISGC2017). Online at ¡A href="https://pos.sissa.it/cgi-bin/reader/conf.cgi?confid=293"¿https://pos.sissa.it/cgi-bin/reader/conf.cgi?confid=293¡/A¿, id.2*, page 2, Mar. 2017.

P. J. Mucci, S. Browne, C. Deane, and G. Ho. Papi: A portable interface to hardware performance counters. In *Proceedings of the department of defense HPCMP users group conference*, volume 710, 1999.

N. Nethercote and J. Seward. Valgrind: a framework for heavy-weight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

A. Offringa, A. De Bruyn, S. Zaroubi, G. van Diepen, O. Martinez-Ruby, P. Labropoulos, M. A. Brentjens, B. Ciardi, S. Daiboo, G. Harker, et al. The lofar radio environment. *Astronomy & astrophysics*, 549:A11, 2013.

S. Ott and E. S. Agency. The Herschel data processing system-HIPE and pipelines-up and running since the start of the mission. *arXiv preprint arXiv:1011.1209*, 2010.

P. Rubin, D. MacKenzie, and S. Kemp. dd(1) - linux man page. `https://linux.die.net/man/1/dd`, 2010.

S. Salvini and S. J. Wijnholds. Stefcalan alternating direction implicit method for fast full polarization array calibration. In *General Assembly and Scientific Symposium (URSI GASS), 2014 XXXIth URSI*, pages 1–4. IEEE, 2014.

S. Sarkar and D. Tullsen. Compiler techniques for reducing data cache miss rate on a multithreaded architecture. *High Performance Embedded Architectures and Compilers*, pages 353–368, 2008.

T. Shimwell, H. Röttgering, P. N. Best, W. Williams, T. Dijkema, F. De Gasperin, M. Hardcastle, G. Heald, D. Hoang, A. Horneffer, et al. The lofar two-metre sky survey-i. survey description and preliminary data release. *Astronomy & Astrophysics*, 598:A104, 2017.

B. Sigoure. Opentsdb scalable time series database (tsdb), 2012.

K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark. Branch

prediction, instruction-window size, and cache size: Performance trade-offs and simulation techniques. *IEEE Transactions on Computers*, 48(11):1260–1281, 1999.

O. Smirnov and C. Tasse. Radio interferometric gain calibration as a complex optimization problem. *Monthly Notices of the Royal Astronomical Society*, 449(3):2668–2684, 2015.

S. Strother, S. La Conte, L. K. Hansen, J. Anderson, J. Zhang, S. Pulapura, and D. Rottenberg. Optimizing the fmri data-processing pipeline using prediction and reproducibility performance metrics: I. a preliminary group analysis. *Neuroimage*, 23:S196–S207, 2004.

SURF. Grid at surfsara. `https://www.surf.nl/en/services-and-products/grid/index.html`, 2018.

M. Van Haarlem, M. Wise, A. Gunst, G. Heald, J. McKean, J. Hessels, A. De Bruyn, R. Nijboer, J. Swinbank, R. Fallows, et al. Lofar: The low-frequency array. *Astronomy & astrophysics*, 556: A2, 2013.

R. Van Weeren, W. Williams, M. Hardcastle, T. Shimwell, D. Rafferty, J. Sabater, G. Heald, S. Sridhar, T. Dijkema, G. Brunetti, et al. Lofar facet calibration. *The Astrophysical Journal Supplement Series*, 223(1):2, 2016.

J.-S. Vöckler, G. Mehta, Y. Zhao, E. Deelman, and M. Wilde. Kickstarting remote applications. In *2nd International Workshop on Grid Computing Environments*, pages 1–8, 2006.

W. Williams, R. Van Weeren, H. Röttgering, P. Best, T. Dijkema, F. de Gasperin, M. Hardcastle, G. Heald, I. Prandoni, J. Sabater, et al. Lofar 150-mhz observations of the boötes field: catalogue and source counts. *Monthly Notices of the Royal Astronomical Society*, 460(3):2385–2412, 2016.

C. Wu, A. Wicenec, D. Pallot, and A. Checcucci. Optimising ngas for the mwa archive. *Experimental Astronomy*, 36(3):679–694, 2013.