Astronomy and Computing

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH

(Dr. sc. ETH Zurich)

presented by

STEFAN CHRISTIAN MÜLLER

Master of Science ETH in Computer Science, ETH Zurich

born on 18.03.1982
citizen of Wädenswil, Zürich

accepted on the recommendation of

Prof. Dr. Gustavo Alonso, examiner
Prof. Dr. André Csillaghy, co-examiner
Prof. Dr. Timothy Roscoe, co-examiner

2016
The data volume created in data heavy sciences such as astronomy is growing at an exponential rate. Drawing conclusions from data requires extensive processing, which requires high performance computing infrastructures.

In astronomy, the processing needs are diverse. Each instrument is novel and unique, producing new data in new data types. The required processing is as diverse as the research questions the data scientists are interested in. As we learn more about our universe, scientists move on to new questions, requiring new data and new ways of processing. As a result, a large portion of the processing and analysis code is short lived. In exploratory data science, code is written not with the purpose of creating a software product, but as a means to better understand the data. Such code typically changes for each execution run since new information is gained each time, with leads to immediate changes to the code. This development process is comparable to software prototyping, only that a small test dataset is rarely sufficient and large data volumes need to be processed for each iteration.

This calls for systems that can utilize high performance infrastructures without demanding significant development efforts to achieve parallelization. If the process has to be formulated in a constrictive pattern, such as MapReduce, or rewritten in a special language, it is often not efficient to employ such a system, as these efforts cannot be amortized over a long lifetime of the software, as it is possible in many business use-cases for big data. For example, most data warehouse systems assume that the data can be represented as OLAP cubes and that queries can be answered by filtering and aggregating. Since this is the case for many practical applications across many companies, vendors can invest significantly into the development of such systems. Astronomy data processing is lacking these economies of scale.

This dissertation presents Pydron, a system that takes the volatile nature of the code
Abstract

into account. Instead of focusing entirely on minimizing the execution time, Pydron is
designed to allow for a fast development cycles, which includes the time to set up the
system and write code, while scaling to large high performance infrastructures, such as
clouds. This is achieved by semi-automatically parallelizing sequential, single-threaded,
Python code. This approach significantly reduces the economic barriers to utilize highly
parallel infrastructures, as it avoids the cost of reimplementation in a different language
and is flexible enough to handle arbitrary Python code, avoiding a constraining framework.
The system is designed to be quickly deployable on a wide range of infrastructures to ensure
that existing resources can be employed.

Python is currently the language of choice in astronomy. It has proven to be an effective
and convenient tool to write astronomy data processing and analysis code. With Pydron
the data scientists can keep using their language of choice, while profiting from the per-
formance of modern infrastructures. With a small API consisting of only two decorators,
Pydron is easy to use, even with existing code. Behind the scenes, Pydron handles paral-
lelization, scheduling, and resource management automatically, allowing the data scientists
to focus on their research.
Die Datenvolumen welche in Wissenschaften mit einem Schwerpunkt auf Daten gesammelt werden, wächst exponentiell. Um Schlussfolgerungen aus den Daten zu ziehen, müssen diese umfangreich verarbeitet werden. Dazu sind Hochleistungsrechnerinfrastrukturen notwendig.


Dies verlangt nach einem System das Hochleistungsinfrastrukturen verwenden kann, ohne erheblichen Entwicklungsaufwand zu verlangen, um parallele Ausführung zu ermöglichen. Wenn die Datenverarbeitung in einem einengendem System wie MapReduce formuliert werden muss, oder gar in einer speziellen Sprache neu geschrieben werden muss, ist es oft nicht effizient so ein System einzusetzen, da dieser Aufwand nicht über eine lange Lebensdauer der Software amortisiert werden kann, wie es für kommerzielle Anwendungen im Big Data Bereich möglich wäre. Die meisten Data-Warehouse Systeme nehmen an, dass die Daten als OLAP-Würfel dargestellt werden können, und dass Filtern und
Zusammenfassung

Aggregation ausreicht um die Anfragen zu beantworten. Da dies in vielen kommerziellen Anwendungsfällen in vielen Firmen der Fall ist, können die Hersteller solcher Systeme erhebliche Ressourcen in die Entwicklung investieren. Der astronomischen Datenverarbeitung fehlt dieser wirtschaftliche Größenvorteil.


Python ist im Moment die Sprache der Wahl in der Astronomie. Sie hat sich bewährt als effektives und praktisches Werkzeug um Datenverarbeitungs- und Datenanalysecode für Astronomie zu schreiben. Pydron erlaubt den Datenwissenschaftlern und Wissenschaftlerinnen sowohl ihre Wahlsprache zu verwenden als auch die Leistung moderner Infrastrukturen zu benützen. Mit einer kleinen API aus nur aus zwei Dekoratoren ist Pydron einfach zu verwenden, auch mit bestehendem Code. Hinter den Kulissen kümmert sich Pydron automatisch um die Parallelisierung, Disposition und Ressourcenmanagement, damit die Datenwissenschaftler und Wissenschaftlerinnen sich auf die Forschung konzentrieren können.
I would like to express my most sincere gratitude to my advisors. Without their help, this thesis would not exist. Building a bridge between two science domains, it is all too easy to get lost on the other side, or to remain close to the own comfort zone. Prof. Dr. Gustavo Alonso gave me the freedom to explore while keeping me on the path to this thesis. When I got tripped, he had my back. Prof. Dr. André Csillaghy’s experience in both science domains was irreplaceable. He convinced me of pursuing a PhD in the first place, opened the doors to astronomy, and guided me in this, for me, hitherto unknown domain. It was a great pleasure and a privilege to work with them.

My heartfelt thanks to all the astronomers that welcomed a computer science PhD student in their midst and took the time to explain their trade. In particular the wonderful people of the Euclid Mission and Dark Energy Survey who let me join their projects, despite my first suggestion to port *everything* to java. It was a very exiting experience that has opened my eyes.

I had the privilege to be part of two groups, the Systems Group of ETH Zürich and the Institute of 4D Technologies at the University of Applied Sciences Northwestern Switzerland. The colleges of both always had an open ear and supported me with feedback and suggestions. I would particularly like to thank Prof. Dr. Martin Melchior, for his friendship and hard work in both Euclid and the Dark Energy Survey that gave me time to focus on my research.

Last but not least, I lack the words to express how grateful I am for my parents and my sister. Without their love and support throughout my entire life I would not be where I am today.
## Contents

1 Computer science and astronomy ............................................... 1
  1.1 Interactive Data Exploration ........................................... 2
  1.2 The need for data .......................................................... 5
  1.3 Consortium organization .................................................. 6
  1.4 Changing environment ..................................................... 7
  1.5 Project lifecycle ............................................................ 7
  1.6 Operations ................................................................. 9
  1.7 Scientific Data Analysis ................................................... 11
  1.8 Requirements for data processing systems ............................ 13
  1.9 Pydron ................................................................. 16

2 Related work ........................................................................... 19
  2.1 Science Data Archives ......................................................... 19
  2.2 On-site computation .......................................................... 21
  2.3 Interactive Data Exploration .............................................. 21
  2.4 Explicit Parallelization ....................................................... 23
  2.5 Delayed Evaluation ........................................................... 25
  2.6 Just-in-time compilation ..................................................... 27
  2.7 Domain Specific Languages ............................................... 30
  2.8 Streaming ................................................................. 32
2.9 Static Compiler Optimization .................................................. 34
2.10 Resource Management .......................................................... 34
2.11 Summary ................................................................................. 36

3 Pydron frontend ......................................................................... 37
3.1 Using Pydron ............................................................................ 37
   3.1.1 Exposure Calibration ......................................................... 38
   3.1.2 Coaddition ......................................................................... 40
   3.1.3 Pydron API ......................................................................... 42
   3.1.4 @functional contract .......................................................... 43
   3.1.5 Configuration ....................................................................... 44
3.2 System Overview ....................................................................... 44
3.3 Language Translation ............................................................... 46
3.4 Language Reduction ............................................................... 47
   3.4.1 Reduction Steps ................................................................... 47
   3.4.2 Reduced Language .............................................................. 53
3.5 Data-flow Graph generation ..................................................... 54
3.6 Static Single Assignment Form ................................................. 56
3.7 Attribute and Subscript ............................................................ 57
3.8 Function Calls ........................................................................... 57
3.9 Conditional Statements ........................................................... 58
3.10 Loops ....................................................................................... 58
3.11 Exception Handling .............................................................. 59
3.12 yield Statement ........................................................................ 60
3.13 @functional Decorator ........................................................... 60
3.14 Graph Attributes ..................................................................... 60
   3.14.1 Tick ................................................................................... 60
   3.14.2 sync-points ...................................................................... 61
4 Runtime system

4.1 Graph Traversal ............................................. 66
  4.1.1 Conditions to be ready for execution ................. 66
  4.1.2 Conditions for graph refinement ..................... 68
  4.1.3 Conditions to end traversal ......................... 69
  4.1.4 Purging data products ................................. 70

4.2 Graph Refinement ........................................... 71
  4.2.1 Rules for refinement operations ..................... 71
  4.2.2 sync-point Removal .................................. 72
  4.2.3 Inline Substitution .................................. 73
  4.2.4 Loop run-rolling ..................................... 74
  4.2.5 if-tasks ................................................. 76

4.3 Scheduling ...................................................... 76
  4.3.1 Worker Process Model ................................ 77
  4.3.2 Trivial Strategy ....................................... 78
  4.3.3 Scheduler Relocation .................................. 79
  4.3.4 Fault Tolerance ....................................... 79

4.4 Backend ....................................................... 80
  4.4.1 API towards the scheduler ............................ 80
  4.4.2 Code Deployment ...................................... 81
  4.4.3 Pydron Deployment .................................... 82
  4.4.4 Remote Procedure Calls .............................. 83
  4.4.5 Message Passing ...................................... 84
  4.4.6 Watchdog timer ........................................ 85

3.14.3 no-transfer .............................................. 62
3.14.4 quick ..................................................... 62
3.15 Summary .................................................... 63
# Conclusions

6.1 Future Extensions ................................................. 120

6.1.1 External data sources ......................................... 121
6.1.2 Deep scheduling ............................................... 122
6.1.3 Globally distributed computation ............................. 124
6.1.4 Simulated execution runs ...................................... 125
6.1.5 Simulations on hardware ....................................... 126
6.1.6 Streaming ....................................................... 127
6.1.7 Automatic granularity control ................................. 128
6.1.8 Tooling .......................................................... 129
6.1.9 Metrics .......................................................... 130
6.1.10 Fault tolerance .................................................. 131

6.2 Towards interactive data exploration ............................. 131

A Appendix – Open source releases ................................. 135
1

Computer science and astronomy

This thesis presents Pydron, a system to semi-automatically parallelize sequential code written for astronomical data analysis, and to dynamically schedule the execution of that code on infrastructures such as multi-core machines, clusters, and clouds.

Astronomers are facing an exponential growth of data collected by their instruments. As in other fields, managing, processing, and analyzing big data is a major challenge. In recent years computer science has come up with many systems to handle large data volumes. Most of the existing systems have been designed with business data in mind. In business as well as in data science applications extracting insights from large data sets is often an explorative process where precise questions are not known in advance but result from studying the data. However, while the business domain shares the need for data exploration, astronomy and related science domains face unique challenges that limit the applicability of existing technologies. The goal of this chapter is to explain the need for iterative data exploration, give insight on how modern astronomy operates, and how this influences their data processing needs.

This chapter focuses on the aspects of astronomy that are related to their data processing needs. It is based on my experiences from working in Euclid [1] and the Dark Energy Survey (DES) [2]. In Euclid, I was part of the ground segment system group and helped in the early design and prototyping of the system to orchestrate the processing in the distributed data centers. I also participated in the data management group of DES where I helped with the software management, continuous integration, and software deployment
systems. I use these projects to exemplify the data processing aspects which are relevant to this dissertation.

1.1 Interactive Data Exploration

Data science is the art of extracting knowledge from data. This is inherently an explorative process as it lies in the nature of science that we do not yet know the insights that we hope to gain. Due to the massive data volumes, data exploration requires massive computational resources. Nevertheless, the human element cannot be neglected. It is essential that the exploration process includes human interaction to guide the search and draw conclusions. Since the amount of data humans can process is limited, we use this critical human resource efficiently by providing tools that prepare and present the data carefully. Ideally the scientist can explore the data interactively [3].

We can make an analogy to online maps, such as those provided by Google and Microsoft. These allow the users to quickly get a big-picture overview of the world and then zoom in, move, and back-track to explore their area of interest. They even provide photographs of the streets. A collection of paper maps and a file archive of these pictures would, in theory, provide the same information. The difference is not just that online maps are faster to work with, but that they allow the user to explore the world and develop a familiarity with an area that would be impossible otherwise.

Unfortunately, to explore the universe it is not sufficient to provide a zoomable map of the sky. Much of the astronomy science requires aggregation of statistics over large regions of the sky and detailed analysis of individual light sources. The space to explore might be also the parameters of an algorithm, or a set of algorithms for a particular problem.

If, for example, the data scientist can be provided with an interactive way of manipulating the parameters of an algorithm using sliders, and is given the result quickly, the scientist can “play” with the parameters and develop an intuition for the correlation of parameters and result. This can then guide the scientist in the further exploration of the search space. If parameters need to be typed in every time and the result is delayed significantly, it is nearly impossible to develop the same “feel” for the data.

The lack of interactivity makes it challenging to find insights in data unless the scientist already has a clear idea of what to look for. While there are certainly specific research questions astronomers are looking to answer with a data set, there are unexpected insights
hidden in the data that are overlooked, which is why astronomers often come back to datasets years after their creation, with new ideas what to look for.

The positive effect of interactivity has been studied in the context of learning theory. An experiment performed by Optiz et al. [4] asked participants to classify non-word strings according to rules unknown to them. Those participants that were provided with immediate feedback performed significantly better at inferring the grammatical rules from the samples shown to them, compared to the control group that received the same feedback after a short delay.

Interactive data exploration poses many challenging problems, including data management, data visualization, user interface design, and high performance computing. Interactive data exploration has been studied in various contexts, such as business data analysis and evidence based medicine [5]. This lead, for example, to data warehouses where information from various sources within a company is aggregated in multi-dimensional data cubes. A graphical interface can be provided to the users to allow interactive exploration using filtering and aggregation along the predefined dimensions of the cubes. Such solutions can be designed when the structure of the data and the kind of questions the users will ask are predictable. The success of data warehouses is possible since the data of most companies fits well into a data cube format and since data exploration occurs mostly along the axes of the cubes. This makes it possible to deploy the same data warehouse system in many companies, resulting in economies of scale that allow significant investment into the development of such a system.

Astronomy lacks this predictability of future data model, processing, and analysis needs. While it would be possible to take a set of current and past projects as reference for the development of a system with a data model and operators that covers a large portion of the project requirements, the chances that the resulting system would also meet the requirements of future projects are slim. In Astronomy, data scientists are not just consuming data processing and analysis methods provided to them. Instead, research and development of such methods is an integral part of data science in this field. For a data analytics product to be successful, it must not only allow integration of novel methods but should actively support the development of these, as this is a major part of the work of an astronomy data scientist. The degree of flexibility required in such an environment implies that the user interface needs to be turing complete, allowing the users to write their own programs.

Offering the flexibility and power of programming languages to users that are not primarily
interested in software development, but wish use it as a tool to advance in their own field of interest, is challenging. Spreadsheets are an example from the business domain. Effectively a cell-based functional language, they are used extensively to perform calculations that would otherwise require tedious labor or hiring of a software developer. The latter is often not economically possible, especially for one time calculations. Spreadsheet calculations are often used as a fallback when, for example, the functionality provided by the enterprise resource planning software is found insufficient. But even the flexibility of spreadsheets cannot always satisfy the needs due to the lack of turing completeness. For these cases, most spreadsheet products have scripting languages built in. Scripting interfaces like that are common in applications that target expert users, such as in computer aided design (CAD) software. The scripting functionality exists since the users might face challenges not predicted by the developers of the product. All these solutions have in common that they use scripting as a “contingency plan” should a rare operation not be efficiently doable via the primary user interface. The performance of these scripts is of secondary importance since they are seen as an alternative to manual execution of the scripted steps and even a single-threaded script will be orders of magnitude faster than manual labor. The ease of use is considerably more important, but since use of the scripting functionality is rare, it is not a primary concern for the application either.

This thesis focuses on a major component required for an interactive data processing system: A turing complete interface for the users that does not force them to become software development experts. Such languages exist. In particular, Python is well established in the astronomy community for this purpose. However, interactivity also requires high performance computing to give feedback to the user with minimal delay. Given the data sizes and complexity of computations required by astronomy data scientists, using parallel computation is unavoidable for all but the most basic use-cases. Python’s sequential execution model prevents its use for an interactive data exploration system. This thesis shows how the gap between a single-threaded, sequential Python code and parallel high performance computing infrastructure can be closed for many practical problems. Pydron achieves this by semi-automatically parallelizing Python code, replacing the sequential execution model with a parallel one, without changing the language.

We have designed Pydron to be more than a component for a future interactive astronomy data exploration system, which will require significant further research, as discussed in Section 6.2. By analyzing the needs and constraints in the astronomy environment, we were able to make Pydron into a tool that fits into the astronomer’s workflow as it is today,
where it can significantly reduce the round trip time between results in the iterative data exploration process.

The following sections give the reader an understanding on how data is handled in astronomy and what the constraints are that have to be considered for a tool like Pydron to be useful in the current environment. They conclude in a set of requirements in Section 1.8 and a short overview of how Pydron is addressing these in Section 1.9.

\section{1.2 The need for data}

The nature of the objects we can see in the sky has always spurned the curiosity of mankind. Only in the last decades we were able to investigate the nearest objects directly using landers and manned moon missions. But even Voyager 1\cite{6}, the spacecraft that has gone farthest, traveled merely 0.021 light years on its 38 year journey. A tiny distance compared to the edge of the observable universe, 46.6 billion light years away.

With the vast majority of the universe outside our reach, direct experimentation is not possible. This limits us to observation. Archaeologists have found signs of systematic recording of celestial objects as far back as ten thousand years ago\cite{7}. Astronomy is not only limited by the instruments, but also by our ability to record and analyze observations. Historically, this was done by hand. A star catalog released in 1802 by Wilhelm Herschel contained about 2500 entries\cite{8}. With computers, it has become possible to handle far larger datasets. Such data volumes are necessary to answer questions that require statistical methods as the signals are too weak or not directly observable at all.

One such example is the detection of exo-planets, planets outside our solar system. They are fainter than the stars they orbit by orders of magnitude. Both the stars and the planets are significantly smaller from this distance than the resolution of the telescopes, reducing them to point sources. Close to their star, the faint signal of the planet is dwarfed by the far stronger signal of the star close to it.

There are several methods that have been used to detect exo-planets. One measures the perturbations of the star’s position due to the gravitational force of the planet\cite{9}. Another uses fluctuations in the star’s brightness as a planet crosses in front of the star\cite{10}. Yet another measures the point spread function of the star with a precision that allows subtraction of the star’s signal, making the faint signal of the planet detectable\cite{11}. All
of these methods require hundreds of observations to confirm the existence of a single
exo-planet and considerable processing of the resulting data.

A particular elusive subject is dark matter which, as the name indicates, is not directly
visible. We know it is there due to the gravitational effects of its mass. Weak lensing \[12\]
is an approach to detect dark matter by the gravitational distortion of light traveling from
galaxies to us, distorting the image of the galaxy. The effect is weak, and galaxies have
a wide spectrum of shapes and are randomly oriented in space. The gravitational lensing
effects can only be measured using statistical methods over large sets of galaxies, again
requiring complex computation over large data collections.

All these methods inherently require significant processing to calibrate and analyze the
data. A individual research group might focus their study on a single exo-planet, requiring
only a few gigabytes of data, but if such methods are applied to an entire survey, the
data volume and processing needs increase accordingly. Other methods, such as weak
lensing, cannot be applied on a small scale at all and inherently require big data. For the
Dark Energy Survey (DES), the data volume of the final data products is estimated to
accumulate to four petabytes \[13\] over the five year period of observations with the four
meter telescope on Cerro Tololo, Chile. The square kilometre array (SKA) is estimated to
produce 500-7000 petabytes of data products each year \[14\] from the signal of thousands of
antennas. The following four sections focus on the needs and constraints of large consortia,
and Section 1.7 looks at the issues of smaller projects from individual research groups.

1.3 Consortium organization

Projects such as DES, Euclid, or SKA are not a product of a single institution. They
are a collaboration of hundreds scientists and engineers from all over the world. The
organization differs from project to project. However, there are some common elements.

Such large projects require expertise in many areas. Research groups join the consortium
to support the project in their area of expertise \[15, 16\]. These groups are usually expected
to have their own funding. A consortium has a organizational structure, but other than in
commercial projects, the funding does not flow along the paths of the consortium hierarchy.

While this organization has proven effective in many projects, it has some implications
relevant to data processing.
For example, it can be difficult to enforce consortium wide policies about technologies, such as programming languages, libraries, or frameworks. If two groups have strong opposing opinions, then the result can be that both technologies will be used, leading to heterogeneity in the technology stack.

There are also political constraints affecting the data processing needs. The groups are usually funded through their institutions and/or national research funds. This has implications for the data-processing infrastructure. A university will be reluctant to fund infrastructure located outside at their university and a national fund will be reluctant to finance infrastructure to be installed in another country. This leads to geographical fragmentation of the infrastructure available to the project. These data centers usually use different hardware. Some groups may have an interest that the data processing steps related to their area of expertise are executed on their infrastructure.

When designing data processing systems for the astronomy community, it is important to take such non-technical constraints into consideration. Sound technical solutions, such as keeping all data and processing at one location, for example in a cloud, may not be feasible. Instead, support of a large variety of infrastructures is required.

1.4 Changing environment

A consortium is a dynamic construct. During its lifetime of well over twenty years, while the project progresses through the different project phases, the consortium changes too. Personal fluctuations are significant. Over time, research groups will leave and join the project. More fundamental changes occur too. For example, the two projects dark universe explorer and spectroscopic all-sky cosmology explorer merged to form Euclid, an ESA mission. In 2013, NASA joined the Euclid project as well [17].

1.5 Project lifecycle

Projects such as Euclid or DES take years to plan and prepare. They represent a large financial investment and several thousand person-years of work. Such projects are carefully selected. For example, ESA’s Cosmic Vision program [18] is a long term space science mission program that started in 2004. Over several years ideas were collected and the key science questions to be addressed were decided upon. This lead to a call for missions
in 2007. ESA selected Euclid as a medium class mission (500 million euros price cap) in 2011 [19].

This process is competitive and only the most promising projects are selected. In this early phase the consortium does not only need to convince the committee of the expected scientific impact, it also needs to provide evidence that the mission can reach its science goals [15]. On the data processing side, this implies requirement engineering, feasibility studies, and budgeting of infrastructure and human resources. This is complicated by the fact that at this stage there is neither data nor code available. For some processing steps there might be known algorithms, but since the mission involves novel science, there is processing required that is still in research. With operation still over a decade away, infrastructure estimates are made under the assumption of Moore’s Law [20]. Estimates are constantly refined as the work on the project continuous and more information is gained.

Technologies are evaluated on a technology readiness scale with levels from “basic principles observed and reported” to “flight proven” in previous missions [21]. Mission critical software must be maintained for the duration of the project. This is difficult to ensure for third-party software, therefore simpler, custom made software that can be maintained by the consortium maybe preferred.

Several groups contribute on the data processing system, working on different processing steps and providing the framework to integrate these steps [15].

Code is written to create simulations which produce synthetic data that mimics the data produced by the planned instrument [22]. This simulated data is then used by the other teams to test their processing codes and to test the system that orchestrates the data processing. These simulations are a challenging problem by themselves and are constantly refined, which then helps to further develop and verify the data processing.

Once the instrument is operating, there is a calibration phase in which the instrument is tested and calibrated. This is the first time that real data is available for processing. For Euclid this is planned to happen in 2020. This is an intense phase where many assumptions are put to the test. The data processing software is adapted to account for the new insights.

During operation, which usually lasts several years, large data volumes are collected and processed. This puts the data management systems to the test which before handled only simulation data, usually on a smaller scale. Throughout the entire operation period, and
Beyond, the data processing software evolves as the scientists gain a deeper understanding of the instrument. Data is processed as it arrives, but to account for changes in the software, the complete dataset collected so far is reprocessed at regular intervals [15, 23]. The lifetime of an instrument is limited. The collected data has a far longer lifetime however. All data is archived [15]. Both raw data and processed products at multiple stages of processing are archived if feasible. This makes it possible to come back and reprocess the data if research finds better algorithms or new questions can be answered based on this dataset. This way the data remains a valuable asset for the scientific community for many years. For some projects the data size is prohibitively large and can only be stored after significant processing [14].

1.6 Operations

Once the instrument is operational, observations begin. Since observation time is limited, there are processes in place to decide which observations to make. For surveys, there might be a fixed observation plan to scan the universe in a predefined pattern [25]. For other instruments, scientists may submit proposals to be granted observation time.

Even for ground-based telescopes, the scientists usually don’t travel to the telescope to perform their observations. Instead, an observation team operates the instrument. This team is the first one to see the data. Specialized processing and analysis is performed to check the correct functioning of the instrument and help with operations, such as positioning and focusing.

The raw data from the instrument is archived so that future improvements to the data processing software can be applied to it. However, for satellites with a limited throughput to ground or for instruments with a prohibitively large data output, this may not be possible.

This raw-data is usually not ready for scientific analysis and first goes through several steps of processing to remove instrument effects and extract information that the scientists may be interested in.

For example, the camera used by DES consists of 62 CCD sensors each with 2048 x 4096 pixels [15]. One exposure creates 62 images. Between the sensors there are gaps. Each sensor, each amplifier, and each pixel have slightly different behavior. The atmospheric conditions vary with each exposure. Figure 1.1 shows parts of an unprocessed image from
Figure 1.1: Raw exposure from one sensor of DECam (cropped). The bright round light sources are stars. The fainter oval shaped ones are galaxies. A number of artifacts are visible [24]. For example, each side of the sensor is using a different read-out amplifier with different black currents. The star at the bottom right is overexposed. The charge has leaked into neighboring pixels, in particular along the columns. The four smaller diagonal rays originating from the same star are reflections of the frame that holds the camera in front of the four meter primary mirror. The bright hair-like lines on the left are traces of high energy particles. Some noise can also be seen. There are also other artifacts present that difficult to spot in this image, including manufacturing imperfections, causing a ringing effect across the sensor, and marks left by double sided tape used in the assembly process (top left). Source: Dark Energy Survey, exposure 009949, sensor 25S.

DES. The data is processed to remove dark current effects and to calibrate the sensitivity of each pixel. Defect pixels are detected and masked. To account for the gaps and the missing pixels, overlapping exposures are made. These images have to be aligned with sub-pixel accuracy, taking into account any distortion. They are then merged to compensate for the missing parts of the image and to increase the signal to noise ratio. There are many other factors to take into consideration, such as the point spread function and unwanted
reflections on the structure that holds the instrument in front of the primary mirror. Light sources are detected to form catalogs of objects. Especially for faint objects that are close to the noise floor, this is a challenging problem. Machine learning methods are applied to categorize these objects by their type, such as stars and galaxies. Properties, including the sub-pixel accurate position, the brightness and shape are measured for each object. DES makes exposures in five different wave-length bands, giving insight on the spectrum of each object which can be used to estimate the distance [25].

The data products that result from this process are made available to the scientists of the consortium at different stages of processing. The science groups then process and analyze this data further to extract scientific insights. Some projects have a proprietary period during which the data is only available to consortium members or the team which was granted the observation time. After that time, for DES it is one year, the data becomes publicly available [23]. This puts significant pressure on the data processing and science teams to produce results in that time period.

For some projects there can be stringent constraints on the processing time. For example, DES is also used to detect supernova events (the very energetic explosions of high-mass stars at the end of their lifetime) [23]. These events are short lived and the astronomers need to be informed on short notice to be able to study the events.

The science teams and the data processing teams are in constant contact, exchanging findings of the instruments behavior, and improving the data processing to produce more accurate data. As a result, the data processing software is changing frequently.

### 1.7 Scientific Data Analysis

The data processing made as part of operations is driven by the goal of delivering science-ready products to the consortium. These products are well defined and changes occur primarily to improve the data quality and the robustness of the system.

The codes used by the individual research teams to study these data products are far more agile. They try to answer a specific research question. Writing such software is primarily a learning process in which the results of the previous execution give the data scientists information that results in a better understanding of the data, the research question, as well as the method attempted to answer that question.
This is an iterative process where significant code changes occur between all runs, comparable to prototyping in software engineering. There are some key differences though.

Prototyping would typically be done with only small test dataset. If possible, this is done in astronomy research too, but often large datasets have to be processed as the goal is not to develop software, but to understand the data and study the scientific results produced from it.

Every attempt is made by the researchers to reduce the data volume required for their project. How well this works depends on the problem at hand. If the required data is limited to a small region of the sky, the data can be selected easily. If the problem requires statistics over the entire sky, it might be unavoidable to process the entire dataset of a survey. Since most data science research groups focus on projects with smaller datasets, one could get the impression that big data processing is not needed for individual research groups. However, with the current technology analyzing large datasets is infeasible for a single research group. This directly limits the research questions a group can attempt to answer. Such big-data science can currently only be attempted on a consortium level, which is a slow and expensive process, limited to a few research questions carefully selected by the process mentioned in Section 1.5. This typically precludes truly exploratory processes as the outcome is too uncertain. Improving access to parallel infrastructures for individual data scientists and research teams is not just an effort to make their work more efficient, it will also open the door for science that is currently out of reach entirely.

In commercial software development, prototyping is usually a first step for a envisioned software product. It represents a comparatively small part of the project’s budget. Parts of the prototype may later be incorporated into the main project. For data science this is not the case. The iterative process continues until the research question is answered and the results are published or the interest shifts to other questions. At this point there is no incentive to continue the development into a fully-fledged software product, thus this prototyping-like phase constitutes the entire life-span of the software.

Compared to the software used in operations of a consortium, which has to be maintained for decades, robustness and longevity is less of an issue for code written by a small team for a specific research question. This makes new and experimental systems more attractive in such an environment.

For commercial software, the cost of deploying a system, such as porting code to a different language or re-writing it to fit the structure of a framework can be amortized over the
lifetime of the software. This is not the case here. To be practical in astronomy, a system should be non-intrusive and allow for fast development cycles while being sufficiently flexible to allow for the novel processing methods the scientist is developing.

In practice, two patterns often occur. If the smallest usable dataset fits onto a single machine, and the processing time is not much more than about 24 hours, the data scientists often use their workstations and write the code themselves. This is different than, for example, in biology, where few scientists write code. In astronomy data scientists often prefer the direct control over the data processing. Python is currently popular, replacing C and Fortran. Using numeric packages such as NumPy and SciPy [26], the performance limitations of Python can often be avoided.

If the problem size makes use of clusters and clouds unavoidable, the software development is handed to professional software developers for parallelization. Using thousands of CPU cores drastically reduces processing time. This requires that the astronomers communicate their needs to the developer. This needs close collaboration and both parties need to have some degree of knowledge of the other ones profession. Maintaining the parallelized code is also costly, considering that significant changes are required each time. Typically, communication and development are the limiting factors, not execution time. An experienced team may achieve one development cycle per week, of which only a few hours are spent processing the data.

1.8 Requirements for data processing systems

Data processing in astronomy faces a unique combination of challenges that results in a fundamentally different requirement set compared to industry applications. Data science is performed in groups of various sizes, from individual researches to large international consortia. Many of the requirements are common across all project sizes, although the reasons can differ.

The exploratory nature of astronomy expresses itself in a demand for interactive data analytics. The business domain desires interactive data analytics on large datasets as well, but the systems designed for that domain can make assumptions on the data structure and queries that allows such solutions to be used across many companies. Similarly, other science domains, such as biology, have more predictable data processing and analysis needs which allow for significant development effort to be put into these methods. Astronomy is
lacking these economies of scale, requiring significant code specific for a particular project. This leads to a more ad-hoc style of software development, similar to software prototyping, where the life expectancy of code is short, with the key difference that for astronomy data analytics, large data volumes have to be processed, as the goal is not the development of a software but extract insights from the data. While the need for fast development cycles with big data can also occur occasionally in other fields, in astronomical data processing it represents the majority of software development. It is worth noting that the short life-expectancy of the code is not just limited to data science in small groups. The code in large projects is changing too, in a constant effort to improve the quality of the data products. Both cases are driven too, in a constant effort to improve the quality of the data products.

Since writing code is an integral part of data exploration in astronomy, most data scientists in astronomy prefer to write their own code. Delegation to software developers is done if needed, but it implies that the round trip time of the iterative process now involves explaining the ideas to a different person, that may or may not have an astronomy background. This slows the development cycle down to days per iteration. In other fields, data exploration does not require code changes for each iteration. The user might, for example, use a graphical query editor to interact with the system. This decoupling between data scientist and software developer does not exist in astronomy in this form. A parallel computing system for astronomy cannot expect that the developers are computer science experts.

Political constraints dictate the infrastructure acquisition and location of deployment. For large projects, this leads to a fragmented and heterogeneous infrastructure distributed across many countries. Strategies that companies might follow, such as centralizing the data processing and analysis in a single location, are not available to astronomy. For individual research teams the situation is similar. Unlike companies, which process mostly their own data, astronomy requires data from instruments that are rarely located close to the scientist’s institution. This gives the team the choice between large data transfers and running the software on remote infrastructure closer to the data. This is makes portability a major concern, requiring not just portable software, but a parallel computing system that makes few assumptions on the infrastructure. In particular, such a system must be deployable without administration rights and cooperate well with existing HPC management systems as we cannot assume that the hardware will be dedicated to it. The system should serve as an abstraction layer which decouples the scientist’s code from the underlying infrastructure, making it possible to run code efficiently in a different
environment from where it was originally developed. Such an abstraction layer can also help in operations when transitioning from one generation of hardware to the next. Due to the long lifetime of astronomy projects such transitions are unavoidable and will occur several times.

With short-lived code developed by individual research teams, it is important that a system is quick to deploy, or the setup efforts might be prohibitive. This is less of a problem in software prototyping and ad-hoc software written in the business domain, as these typically do not require parallel computing, therefore deploying a parallel computing system would typically only be done for longer running projects that process big data.

Scientific data analysis is agile and hard to predict. This makes it difficult to judge which processing system will fit the need at the beginning of the development. It might not even be apparent at an early stage that parallel data processing will be required at all. If a system is intrusive, it might not be profitable to use it once the development has progressed. It is therefore beneficial if a system can support existing code. In particular, systems can be difficult to adopt if they require code to be written in a specific programming language, or require that code is written within a framework that dictates the structure of the code. This is an issue outside astronomy as well, as software prototypes and ad-hoc solutions often evolve into larger projects.

A significant amount of code is written for a specific mission or for a specific research question. However, there are also reusable software components. These can be utility libraries, for example to read and write certain file formats, implementations of common processing steps, such as sextractor [27], or interactive data visualization tools, such as DS9 [28]. Existing software can form part of the data processing and analysis. The system must allow for external applications and legacy libraries, written in Fortran or C, as well as legacy data formats, if not to significantly limit its applicability. This is a common issue in software development. Such existing components often represent a great value and would be prohibitively expensive to redo with current technologies, in particular for a short-lived research project. A non-intrusive system, capable of handling existing code and libraries, also lends itself to short development cycles, as existing code can be used when available. The absence of constraints that would force a complete or partial rewrite of the software implies that frequent code changes do not require expensive adaptation either.
1.9 Pydron

There are two options to handle the requirements and constraints summarized in the previous section when designing a system for astronomy data processing. The system design can take them into account, or expect that astronomers adapt to the ways dictated by the system. Most existing systems have not been designed specifically for astronomers and therefore fall into the second category. None has found widespread use for astronomy data processing. Pydron adapts to the way astronomy operates today, taking into account the requirements and constraints, rather than expecting astronomers to adapt to Pydron. The fast changing code is the result of the exploratory nature of data science and the current state of astronomy research which has not converged on a stable set of processing and analysis methods, and may never do so. Pydron is designed specifically for prototyping with large data. It keeps the development overhead to a minimum with semi-automatic parallelization and it uses Python, the language that is successfully used in astronomy data science today. After each execution, the results are studied, conclusions are drawn, and the code is changed according to the new insights. The efficiency of this process is not determined by the execution time alone, but by the time to complete an entire development cycle. Pydron gives the developer access to parallel infrastructure to reduce the execution time without significantly increasing the time spent writing code. This is different from most existing systems which focus primarily on the optimal performance during execution.

An attempt could be made to convince astronomers to delegate software development to computer scientists more often, which would make advanced parallelization APIs more attractive. But the skill and willingness of astronomers to write code is a strength that should rather be exploited instead. Few fields have the luxury of users with this skill. In business administration, for example, employees with excel programming skills are rare and highly sought after. Pydron is designed for ease-of-use, having a minimalistic API of only two decorators. This allows the data scientists to remain in charge of code and data themselves, avoiding the slow development cycles that result from delegation to a software developer.

The distributed and heterogeneous nature of the infrastructure is mostly due to financial and political constraints. Attempts at centralizing the infrastructure will fail unless the political positions change. Some attempts in this direction are being made, for example by
the European Open Science Cloud [29] initiative. Despite these efforts, the infrastructure situation is unlikely to change in the near future. We address this in Pydron in two ways. A flexible and modular back-end allows Pydron to utilize every infrastructure that can run a Python interpreter, often with a simple configuration file, or with a custom back-end module for more exotic cases. On top of this, Pydron adapts its scheduling to the infrastructure, so that the developer does not need to tweak the code for a particular environment.

Support for legacy software and data formats can be challenging to fit into a new system design. However, personal experience from the technology selection process for Euclid’s data processing has shown that a lack of legacy support leads to immediate rejection. The pressure to use the existing solutions for the few common processing steps is too strong. Pydron is able to support arbitrary Python code. This is not just important to handle legacy Python code, but it also lets Pydron inherit Python’s strengths in interacting with external legacy applications and native libraries which are among of the main contributors to Python’s success in astronomy.

What Pydron cannot offer at this stage is true interactive data exploration. This would require development cycles in the sub-second to a few seconds range. Interactivity also requires a number of additional requirements, such as interactive graphical data visualization and incremental computations in the presence of continuous code changes. It would require fine granular parallelization that Pydron currently does not offer. We see Pydron as an important step towards interactivity in astronomy data exploration, as such a system will require a turing complete, easy-to-use interface to the highly parallel infrastructure that will be required to achieve such short execution runs. This is discussed in more detail in Section 6.2. Since other components are still missing to achieve this goal, we designed Pydron for coarse granular parallelization, where it can be a significant benefit to the data scientists today.

The following chapter discusses how existing systems relate to astronomy data processing. Chapter 3 then shows Pydron from the user’s perspective and explains how Python code is translated into an intermediate data-flow graph, that the runtime system can execute in parallel, as described in Chapter 4. Chapter 5 introduces the location-aware scheduler which reduces data transfers and makes the execution more independent of the infrastructure by adapting to it automatically. The thesis concludes in Chapter 6 and provides an outlook for future research that could benefit from the semi-automatic parallelization and the dynamic data-flow graph structure which is behind it.
In this chapter we look at a selection of existing systems designed to aid the developer in using parallel infrastructures. These systems make different tradeoffs between simplicity for the developer and efficiency in their use of the infrastructure. They also differ in the use-cases they are designed for and the kind of hardware they can support.

We have grouped these systems into categories to highlight some of the design choices that have been made by existing systems. This assignment is not entirely black and white, as a parallel processing framework is not a matter of a single design choice. For example, Spark Streaming [30] is combining data streaming with a delayed evaluation frontend. We exemplify each group by discussing a number of systems to demonstrate how the highlighted aspect is used in existing systems.

### 2.1 Science Data Archives

Astronomy projects require high quality data products to produce high quality science. These data products are a valuable asset well beyond the lifetime of the instrument. Science data archives are a research topic by themselves. Since this thesis focuses on the data processing, this section gives only a short overview of the aspects particularly important from a data processing point of view.
Chapter 2. Related work

Due to the large data volumes, it is not realistic to expect scientists to download the complete dataset for study. Researchers therefore need means for selecting the data from science data archives which is relevant for their work. This is an inherently difficult problem as it depends on the research question the data scientists are trying to answer by analyzing the data, which results in complex, hard to foresee, selection criteria. It is further complicated by the wide range of data types that exists in different projects and even among the products of one project.

For data that can be represented by a schema, relational databases are typically used. These can be meta-data or actual data products. For example, surveys such as the Sloan Digital Sky Survey [31], Euclid [1], or DES [2] create catalogs of celestial objects, such as galaxies and stars. For each catalog entry, several characteristics are measured, including position, brightness and other properties to describe, for example, the spectrum and shape of the object. Data scientists can formulate SQL queries to select those entries required for their research. The large variety of queries makes it challenging to select the right set of database indexes. These queries are often expensive, selecting large subsets of the objects, which can make specialized extensions to the database system necessary, such as specialized indices [32].

Not all data fit well into a relational database model. Image and spectrum data are usually stored as raster data. In DES, these products are stored as files. A relational database is used to keep track of the files and make them searchable by meta-data. This has the drawback that the actual data cannot be used in the query. Database systems such as SciDB [33] can handle multidimensional array data directly to address some of these issues.

In cases where the scientists requires fairly small datasets and the database allows for sufficiently expressive queries to select the data these approaches work well and are used either with direct database access or through web frontends [24]. For example, it might be enough to select data for a specific galaxy, or for the galaxies in a small region of the sky. But when the research question requires statistics over a large dataset, then it becomes important to perform as much of the processing within the archive to avoid transferring the entire dataset.

Trying to delegate parts of the processing to a relational database leads to complex queries that process large data volumes. This is a kind of workload most databases are not optimized for. Databases are very efficient at handling large numbers of comparatively small queries for online transaction processing (OLTP) or few large queries for online analytical processing (OLAP), with a schema and index structures carefully prepared for
2.2. On-site computation

Due to the limitations of databases, scientists are either forced to download a large dataset and perform the processing themselves or restrict themselves to science that does not require statistics over large data volumes. Alternatively, a high performance computing infrastructure can be made available at the site of the archive to allow scientists to run their own codes on a cluster near the data.

In such a shared infrastructure, the individual data scientists will only have limited access rights, which forces the user to use the cluster management system put in place by administrators. It typically will not be possible to install custom file systems, database systems, queuing systems, such as the ones from Hadoop [34]. Such systems have to be put into place by the administrators.

2.3 Interactive Data Exploration

The importance of interactivity in discovery oriented applications is well known. Machine learning methods can help in extracting interesting insights from data, but the human cannot be replaced to make sense of the data and provide domain knowledge to guide the search for interesting aspects of the data. This leads to an iterative process where the user closely interacts with the system. This section focuses in on interactive data exploration systems that are of particular interest for astronomy use-cases.

Query Steering Automatic Interactive Data Exploration [35] is an example of a database centric exploration system. It is designed for use-cases where the user wishes to navigate a data space to find objects of interest, for example in a catalog of light sources from an
astronomy survey. The system presents a selection of samples to the user which then classifies these objects as relevant or irrelevant. It then uses this information to find similar objects that might also be of interest. The process is iterative with the goal of refining the selection criteria to steer the user towards the relevant data subset. This is closely related to machine learning online classification methods, where a classification of objects is learned from incremental feedback [36]. Query steering has the additional challenge that it must also choose which objects to present to the user next. These should be of high relevance and provide good samples to improve the classification in the next iteration.

This method is limited to data that can be represented as a collection of objects and to use-cases where the user’s interest corresponds to a sub-set of objects. If the relevant information requires processing, the entire dataset has to be processed beforehand. If the sought after insight is not in individual objects, but, for example, in statistics gathered over the entire dataset, such methods cannot be applied.

**VIZDOM**  
VIZDOM [37] is an interactive data analysis tool for pen and touch interfaces. It combines machine learning with complex statistics and user defined workflows with the goal to give an intuitive, easy to use, interface that can extract deeper insights than the high level summaries that can be expected from OLAP reports. The frontend is based on PanoramicData [38] which allows visual construction of SQL queries. This interface is extended with support for machine learning and statistics operators. The backend is delivered by Tupleware [39], a parallel execution system designed for small clusters. Tupleware can handle user defined functions which are translated into native code for performance and augmented with the necessary synchronization constructs for parallelism. VIZDOM recognizes the importance of giving the user more flexibility than can be offered through SQL alone. In particular, the ability for parallel execution of user defined code is an important feature for scientific data exploration. It is, however inherently limited to data that can be represented as tuples by to the expressions supported within the user defined functions.

**IPython and Jupyter**  
Combined IPython [40] and Jupyter [41] form a web-based interactive Python environment. They use the concept of a digital notebook where code, text, and data visualization can be combined. The notebook becomes a document inside which code can be executed to provide parts of the content. Integration of libraries for graphical user interfaces allows the user to create on-demand interfaces. For example,
to interactively modify parameters using sliders. IPython has some support for parallel execution through an explicit parallelization API.

## 2.4 Explicit Parallelization

Developers can achieve parallel computing by explicitly parallelizing their code. At the lowest level of abstraction, the developers can use the API provided by the operating system for multi-threading, multi-processing, and network communication [42]. This gives the developer the highest degree of control at the cost of complexity. An experienced developer can achieve the highest possible performance using low-level APIs. In situations where the high performance benefit outweighs the high development cost, this is a reasonable approach. For example, algorithms implemented in database systems are used intensively by the thousands of companies which use this product for daily operation. This justifies the significant investment required to make this code fast, robust, and compatible with a wide range of hardware.

The systems described in this section provide an abstraction layer on top of the low-level APIs and hardware to simplify development by offering the developer a set of well tested and optimized tools at the cost of restricting the flexibility. MPI, for example, is based on a message passing paradigm, whereas Python’s multiprocessing module can only utilize cores on one machine.

These APIs have in common that the developer has to explicitly formulate how the program has to be parallelized. It is the developer’s responsibility to split the work up into tasks. Unless the API is limited to shared memory systems, communication needs to be handled explicitly by the developer too.

Since these systems still leave a significant amount of work to the developer, other systems have been created that internally use these systems, and take over some of that responsibility. Some of them are discussed in later sections of this chapter.

**OpenMP** OpenMP [43] simplifies parallelizing loops with multiple threads. The C code is augmented with `pragma` directives. During the build process, these are detected and the necessary code for parallelization is injected. An example is the `parallel for` directive that marks a loop and causes the iterations to be split among the threads. This approach saves the developer from writing boilerplate code for thread management and synchronization.
Communication is done through shared memory. This limits OpenMP to parallelization on a single machine. While OpenMP makes multi-threaded parallelism significantly easier, it does not ensure that the results are equal to those of a single threaded execution of the same code. In particular, it does not check if there are any conflicts in memory access that would constrain the execution order of the loop iterations. The developer is responsible to ensure correct execution which is supported by various directives for thread synchronization, atomic updates, and thread private variables. By delegating this responsibility to the developer, OpenMP can avoid code analysis beyond detection of the directives. This implies that the developer must be well versed in multi-threaded programming.

**MPI** MPI [44] is a communication API for parallel computing. The participating nodes can send messages to each other to exchange information. MPI serves as an abstraction layer above the underlying communication hardware. Implementations exist for Ethernet, Infiniband, shared memory, and others [45, 46, 47]. Since MPI does not rely on shared memory, it can be used to parallelize over many machines.

Only the communication is handled by MPI. The application is started on all nodes as individual processes. Each node is given a unique number which allows the developer to differentiate the different processes running in parallel. The developer is responsible for splitting the computation into parallel tasks. All communication is done explicitly through MPI’s API.

MPI and OpenMP can be combined. In this setup, one process is started per machine. MPI handles the communication between machines, while OpenMP uses multi-threading and shared memory to utilize all cores of the machine.

**CUDA** APIs, such as CUDA [48], are required to access GPUs. A GPU architecture is different from a multi-core CPU architecture. To profit from the GPU performance, the developer requires knowledge of the hardware architecture, such as the thread-group hierarchy, shared memory hierarchy, and barrier synchronization. Other than with CPUs, where each core can execute independently of the others, GPU cores use single instruction, multiple thread (SIMT) execution model where each core executes the same instruction in the same clock cycle. GPUs are therefore useful if the same program has to be performed many times on different data. This is a similar scenario as for single instruction, multiple-data (SIMD) instructions of modern CPUs.
2.5. Delayed Evaluation

**MapReduce** MapReduce [49] is a computational pattern that has proven useful in many data parallel applications. In the first phase, a user-defined map function is applied to all data items. These operations produce a number of key-value pairs. The MapReduce system then groups these pairs by key and applies the user-defined reduce function to each group. This structure is fixed, there is no communication between individual map or reduce functions other than through the key-value pairs.

Since MapReduce systems, such as Hadoop [34], don’t have to support arbitrary execution and communication patterns, they can optimize for this specific one. For example, HDFS is a distributed file system specifically designed to be used with MapReduce. It reduces the required data transfers by executing the map operations on the node where the data is stored. These systems also contain optimized shuffle stages to group the items by key [50], and advanced load balancer strategies [51].

While a MapReduce system is handling distribution and communication, the developer is responsible to bring the problem into a shape where the MapReduce pattern can be applied.

### 2.5 Delayed Evaluation

Systems that take over some of the burden of explicit parallelization need a way for the developer to specify the processing that is to be executed in parallel. Many of the languages currently in widespread use were not designed with parallelization in mind but were designed for sequential execution.

Delayed evaluation is a method to transform sequential execution into a formulation of the program that a system can then help to parallelize. While executing the program, an operation not executed directly. Instead, it returns a place-holder object for the result that contains the instructions on how to perform the actual calculation. Those objects can then participate in other operations, resulting in an expression graph. This graph representation is used by the system to perform the actual computation in a parallel fashion. This is also known as lazy evaluation, as the evaluation is delayed until the data is actually accessed, at which point the evaluation is performed.

Since delayed evaluation is typically not supported for the operations from the programming language’s standard library, such systems introduce their own data types and operators. The close control of the system over both data and operators allows for efficient
parallelization and data management. The developer has to formulate the code with the data types and operators provided by the system. For the scientific applications we are interested in, this is a significant limitation as those applications often use legacy code and data formats.

Some languages include language features that allow for tighter integration of systems that use delayed evaluation. For example C# support LINQ [52] which is a delayed evaluation query language build directly into the language. This can be used to execute these queries in parallel, for example with PLINQ [53].

**Spark** Spark [54] provides a delayed evaluation API for several languages, including Python and Scala. It uses the concept of a *resilient distributed dataset* (RDD), a read-only collection of objects distributed across several machines. The data is either stored or can be rebuilt if data is lost. The developer writes code that applies high-level operators to RDDs, which are then applied per object (*foreach*), or to combined objects (*reduce*) of the RDD. The API provides functions to create RDDs from files and provides the operators that can be applied to them.

The targeted use-cases are similar to those of MapReduce, unlike PigLatin (see Section 2.6), Spark does not use MapReduce internally, but comes with its own runtime system. This gives Spark more freedom to implement optimizations, such as storing intermediate results in memory.

**Weaver** Like Spark, Weaver [55] uses the concept of datasets that represent collections of objects. There are different types of datasets. In a file dataset each object of the dataset is the path to a file. Weaver also supports datasets that are backed by SQL databases. Weaver targets scientific use-cases where filter operations are common. For this purpose Weaver includes a query API with SQL-like operators which can be applied to datasets. These queries are translated into SQL queries if the dataset is based on a database, or evaluated directly.

The actual computations are performed by external applications. Data is passed to the executables by passing the file names stored in the datasets as command line arguments. Weaver provides functions such as *map*, *reduce*, *all-pairs* that can be applied to datasets.
FlumeJava FlumeJava [56] also uses special collection types and operators. Before it executes the operations, it performs a series of optimizations. For example, it may combine several operations into a single MapReduce step. This is possible as delayed execution does not enforce the same execution order as in the code. The entire expression graph is available before the execution starts.

FlumeJava employs runtime information about the actual data, to decide where to execute an operation. For smaller data sizes it will run locally in a thread. This helps to avoid the overhead of starting remote parallel jobs. For larger datasets it will start a remote MapReduce operation and uses heuristics to estimate the output sizes and computation requirements to choose the number of parallel workers.

2.6 Just-in-time compilation

Python is a dynamic language, as such it is typically significantly slower than statically typed languages such as C. Due to the global interpreter lock, it is also not very well suited for parallelization. One approach to address this issue is to translate the source code into another language, such as CUDA or C++, more suited for the targeted parallel infrastructure. The translation and subsequent compilation typically happens just-in-time during execution as opposed to a separate compilation step. The performance improvement comes from a more efficient target language, and / or from parallel execution on hardware such as graphics processor units.

These systems may place constraints on the code they can translate. There is usually mismatch between the data types and language features of both languages, especially if hardware such as GPUs are targeted. There is also a mismatch between the available libraries. This approach is limited to a subset of data types, language features and libraries unless all Python libraries can be ported as well. This makes such systems attractive to speed up inner loops, where the amount of translated code is relatively small, and it is easier to comply with these constraints.

The target language is typically statically typed, which gives the compiler a significant advantage for optimization, leading to faster code. The transition from dynamic typing to static typing poses a challenge. A just-in-time (JIT) compilation system must compensate for the lack of type information.
Chapter 2. Related work

JIT compilation may be performed for parallelization but even with single threaded execution it can be a valuable tool. Python can be orders of magnitude slower than native code. For algorithms that lend themselves to vectorization, NumPy [26] is commonly used to avoid implementing the innermost loops directly in Python. If that is not possible, JIT systems can translate these innermost loops to native code avoiding the overhead of Python. This is convenient for the developers as they can continue to develop in Python and the constraints imposed to make the translation possible affect only a small portion of the code.

Numba   Numba [57] is a selective JIT compiler for Python. The developer is responsible to decide which functions are to be translated by adding a `@jit` decorator. Once the method is invoked for the first time, Numba will translate the Python code into the intermediate language of the LLVM compiler which then creates native code. This code is cached for further invocations.

Numba uses type inference. Based on the types of the arguments passed to the function, the types of all variables and intermediate results are inferred. If this fails, for example because the resulting type of an operation is unknown, the code cannot be translated. Not all data types are supported. Numba supports primitive types, such as integers, and certain complex data types, notably NumPy arrays. Other types, such as user provided classes, are not supported. If the function is invoked with different combinations of argument types, the process is repeated for each combination and separate native code variants are generated.

Since NumPy is a Python library, there is no direct equivalent available for the generated native code. Numba provides implementations for a subset of NumPy functions.

HOPE   HOPE [58] is similar to Numba, with a focus on astrophysical calculations. It too is selective using a decorator and employs type inference.

Other than Numba, it translates to C++ code, which is then compiled. In addition to the optimizations implemented in the compiler, HOPE has its own optimizations that are applied before the C++ code is generated. Using the symbolic math library SymPy [59], mathematical expressions are simplified and common sub-expressions are factored out. `pow` with integer exponents are replaced with multiplications as well.
SEJITS  SEJITS [60] is a framework for embedded domain specific languages. It translates code specifically selected by the user. However, while systems such as Numba attempt to support arbitrary computations, SEJITS targets specific use-cases, for example stencil operations on multidimensional arrays. These are called specializations. The flexibility of SEJITS comes from the simplicity with which such new specializations can be added to the framework. Since each specialization is targeted at a specific computational pattern, domain specific knowledge can be used to produce highly efficient code, for example by loop fusing and elimination of extraneous copies between CPU and GPU [61].

SEJITS can be used to generate code for C/OpenMP [43] and CUDA [48].

Copperhead  Copperhead [62] compiles Python code to CUDA. GPUs have a significantly different architectures compared to CPUs, for which Python is designed. Mapping an imperative language to a data parallel architecture is non-trivial. SEJITS solves this issue on a case-by-case basis using specializations for each computational pattern that is to be supported. Copperhead is more generic. The developer has to write the translated code using Python built-in functions such as `map` and `reduce`, as well as language features such as `lambda` expressions, that are recognized by Copperhead and allow it to generate data parallel code.

PyPy  PyPy [63] is a complete Python interpreter. In all above JIT systems, the developer explicitly selects the code that is to be compiled. PyPy uses heuristics to select which parts of the code should be compiled to native code. It does not support alternative platforms, such as GPUs, and it does not automatically parallelize the code. This is the same approach used, for example, by HotSpot [64] for Java.

PyPy has the advantage of being completely transparent to the developer. There is no need to mark functions with decorators, nor are there any constraints for data types or operations since PyPy supports the entire Python language and includes a near complete standard library. Limitations mostly exist with third party libraries if they are not written purely in Python. For example, NumPy is only partially supported.

As with all JIT systems, PyPy will not always be able to compile the code to native, for example, type inference can still fail. This will not cause the application to fail however, rather will PyPy be forced to execute this code as an interpreter.
The ease-of-use comes at the cost of performance. There is an overhead introduced as PyPy will require a certain warm-up period to detect often executed code and compile it.

**Cython** Cython [65] is an extension to the Python language and an optimizing static compiler. Like PyPy, it serves as a replacement for the Python interpreter. It adds language constructs to optionally declare types, making Cython partially statically typed. This gives this approach an advantage compared to the others as it does not have to rely on type inference only. This comes at the cost of a higher development effort. Typically, the developer would add type information to functions that are known to be performance critical. As such, Cython can be placed between PyPy and Numba, as the developer gives hints, but does not limit compilation to certain methods. Since CPython makes changes to the Python language, compatibility is limited.

Cpython requires a compilation step in the build procedure, as such it is not a just-in-time system. We list it here for comparison as it serves the same purpose.

### 2.7 Domain Specific Languages

When parallelizing at a coarse granular level, near the outer most loops, the distributed tasks will use a significant amount of the application’s code. Just-in-time compilation approaches are less suited here, as they work best with small quantities of code, where methods such as type inference are successful. To avoid the limitations of Python in respect to parallelization, programming languages can be designed that are better suited for this task.

Parallelization of the actual computations and the inner loops are typically not addressed in those languages. Instead, such systems provide convenient ways to call code written in other languages. Such orchestration languages can be functional or have other means to avoid side-effects which would hinder automatic parallelization. Those constraints are less restrictive than in the source-to-source translation approach since they only apply to the outer loops and not to the code called from within.

A separate language enforces a strict separation between orchestration and computation. This can be an issue if the application has no natural boundary at which this separation should occur. Which parts of a program are implemented in which language has a significant effect on parallelizability and performance. However, moving code from one language
to the other requires costly reimplementation. This makes such solutions intrusive as the software architecture is greatly affected by them.

**SWIFT** SWIFT [66] is a scripting language designed for coarse granular parallelization. It is statically typed and has a syntax similar to Java. The code is translated into a data-flow graph which is then traversed at runtime. The language constructs, such as loops, are designed to allow for parallelism. SWIFT is an orchestration language. The actual computations are performed by *apps*, which are a special type of function which invoke an external application. The function’s arguments are mapped to command line arguments. SWIFT is built on top of Karajan [67], a runtime system for grid computation. Karajan handles scheduling, execution, and file transfers using replaceable *providers*, such as GridFTP [68].

**CIEL** CIEL [69] uses a programming language called Skywriter. It has language constructs such as *spawn* and loops to create parallel tasks. It supports data dependent control flow. This is important for applications where the required operations depend on the calculated data. For example, an operation may have to repeat until convergence has been reached. For this purpose, CIEL separates objects in two spaces. The data space for large objects that are passed between tasks and a coordination space with small objects which affect the control flow. The data space objects are read and written by the tasks only, which are typically applications written in native code. CIEL keeps track of these objects through references to their location. Objects of the coordination space are managed by CIEL directly and can be used for control flow purposes, for example as the criteria of a *while* loop. Skywriter has a *-operator that moves objects from data space to coordination space by loading the object and interpreting its content as a JSON file.

Internally, CIEL uses a directed acyclic graph representation of the program. Data dependent control flow modifies this graph at runtime. Regular tasks perform an operation, for example by executing an external application, but a task may also decide to spawn new tasks instead to create the outputs. This decision can depend on objects of the coordination space. An *if*-statement can be implemented by a task that will spawn either tasks of the *body* or the *else* block.

**PigLatin** PigLatin[70] is a language designed to make it easier to use MapReduce [49]. Its syntax is inspired by SQL, placing PigLatin between high-level declarative languages
and low-level procedural MapReduce. The developer specifies a series of high level data transformation operations using primitives such as `filter` and `group`. This avoids much the tedious writing of individual `map` and `reduce` functions.

PigLatin is executed as a series of MapReduce jobs on Hadoop [34].

**Taverna** Taverna [71] is a graphical workflow system. In a graphical user interface the developer arranges tasks and connects them to form a workflow graph. Edges can represent data flows or generic constraints on the execution order.

The tasks can be web services using SOAP or REST, local Java code, or external applications. The goal is to give the developer a tool to quickly assemble various tools and data sources available online. Originally targeted for molecular biology, it has found uses in many sciences, including astronomy.

The use of web services makes it possible to provide data or a processing step to the public without forcing the user to install any software. It does, however, limit its use for larger data volumes.

### 2.8 Streaming

Data streaming systems use a data-flow graph representation. Records are passed along the edges, and vertices represent operations on them. Several other systems also use graph representations internally. In those systems vertices are executed once and data that flows along the edges typically represent larger units of data (for example sets of records). Data streaming can achieve finer parallelization, on the granularity of individual records, while keeping the graph at a manageable size since there is no need to have a vertex per record.

Streaming systems have the advantage that they can begin processing before the entire dataset is available. Such a system may be running continuously, receiving and processing elements of their input data streams. Accordingly, the result is not a single data product, but a stream by itself that is updated as new information is ingested. To avoid storing potentially large volumes of intermediate results, streaming systems calculate results incrementally, avoiding to reprocess the entire dataset if new data is added to it. This makes streaming systems particularly attractive for real-time applications where results are required with short delay which makes it unpractical to process data in large batches.
2.8. Streaming

Real-time applications are rare in astronomy use-cases as the observed phenomena progress slowly compared to a human life time. Supernovae (exploding stars) are considered particularly fast events and may take over a week to reach peak luminosity.

For iterative data exploration fast results are important, but it are primarily the algorithms and parameters that change frequently during exploration.

For use-cases that can be formulated as record streams, such systems can scale well to many nodes. The developer has to provide the implementation of the vertices and has to provide the structure of the graph.

**Spark Streaming** Spark Streaming [30] adds streaming features to Spark. It introduces a new data-type called *DStream*, that represents a series of data elements. These *DStreams* can then be manipulated with various transformations, such as *map* and *filter*, using a delayed evaluation API. Spark Streaming splits the data items of the streams into batches which are then processed through the Spark parallel processing system. Stateful transformations are possible through specialized operators. The data-flow graph cannot contain cycles, this limits its applicability in use-cases where data has to be processed repeatedly, for example until a convergence criteria is reached. The batch size limits the latency that can be achieved, as incoming data is not processed immediately.

**Apache Storm** Like Spark Streaming, Storm [72] uses an acyclic data-flow graph. Records are processed immediately without batching which decreases the latency of the system. Each processing node in the graph, called *bolt*, may run several *tasks* on the cluster. These tasks run in their own thread. The records processed by a *bolt* are distributed among the *tasks* according to a strategy selectable by the user.

Tasks report back if a record has been processed successfully. The source nodes, where records enter the system, will trigger a re-processing of a record if successful processing is not acknowledged within a time limit.

**Naiad** Naiad [73] uses a computational model that allows for loops in the data-flow graph. This model allows iterative computation of cyclic computations, that is, if new data arrives, the result of a loop can be updated without running the entire loop again. This allows results to be ready within milliseconds on computations that would not be possible with an acyclic model.
Chapter 2. Related work

2.9 Static Compiler Optimization

Modern processors support parallelism in various ways, with multiple cores, multiple threads, and vectorized instructions. Compile-time optimizations to make use of those features have been studied extensively [74]. The main advantage is that they are completely automated, requiring no additional action from the developer which makes them particularly convenient to use.

Those approaches are limited by the large search space of such optimizations. Like source-to-source translation, reasoning over large sections of code is very challenging and often provably impossible in the general case [75]. As a result, such optimizations are typically only applied to relatively simple innermost loops and not at higher levels.

2.10 Resource Management

Independent of how the parallel computation is formulated by the developer, the system has to handle the scheduling and resource management. There has been significant research into allocating resources for distributed computing applications.

These systems address the issue of assigning tasks to workers in an efficient way. Since their focus is on the resource side, they assume that the problem is already formulated in a certain way, for example as a series of MapReduce operations. These systems assume a known fixed data-flow graph and given estimates for execution cost, data transfers, and infrastructure.

Quincy Quincy [76] does data location aware scheduling for Dryad [77]. To perform a scheduling operation, Quincy forms a graph consisting of all tasks that are ready for execution as well as all workers. Tasks, for which the inputs are not yet available, are not included in the graph. Edges represent possible assignments of tasks to workers. To reduce the number of edges, a data locality threshold is applied that removes edges for assignments that would require more than 90% of the data to be transferred. For each edge a cost for the scheduling decision is calculated, based on the required data transfer volume. The graph is designed such that a min-cost solver can be applied to find task to worker assignments.
The method has several tuning parameters, in particular to compromise between reducing data transfers and fair use in cases where the cluster is shared with other users.

**LATE**  The LATE scheduler [78] militates straggler issues in MapReduce [49] for heterogeneous infrastructures. In cloud infrastructure, the performance of the nodes can vary significantly due to visualization and sharing of the underlying hardware as well as faults, such as hardware defects. Tasks scheduled to slow workers can become the dominating factor for the total execution time. LATE scheduler identifies such workers and speculatively runs a copy of the task on a different machine, in the hope that the copy will finish sooner. Once one of the tasks finishes, the other can be stopped.

To decide which tasks to speculatively execute twice, LATE estimates the remaining execution time of a task based on the time this task has already been running and the fraction of input data it has processed. These tasks have potentially the greatest impact on the total execution time. To avoid running the speculative copy on another slow worker, LATE also estimates the performance of each worker and ensures that speculative executions are only scheduled on fast workers.

These estimates assume that the *map* and *reduce* functions process data at a constant rate, otherwise the heuristic may decide to run a task twice which is slower since it actually performs more computation. To avoid introducing too much overhead, the number of speculative tasks running at a given time is limited.

**Conductor**  Traditionally, the goal of schedulers is to minimize the execution time of a given set of jobs on a given infrastructure. With cloud computing however, the infrastructure is variable, as the user can choose how many instances and of which type should be employed. There are also various choices for storage and different pricing options, such as spot markets which allow bidding for resources. This adds another dimension to the scheduling problem.

Conductor [79] makes it easier for the users to use this additional flexibility to their advantage. Instead of letting the user decide how to use the cloud services, the user specifies high-level goals. Conductor can be instructed to optimize for monetary cost or execution time. The user can set a limit for the execution time will attempt reach this goal with minimal cost by automatically selecting cloud services.
Chapter 2. Related work

A linear programming model is used to model the computation and the cloud services. This is made possible by restricting the computations to MapReduce jobs, and by the linear nature of the pricing model of Amazon AWS services. The model requires estimates for the data volumes, computing requirements, as well as for the performance of the cloud services. As it is challenging to make accurate estimates for these values, Conductor observes the performance at runtime and dynamically updates the model to produce a new execution plan.

2.11 Summary

The systems differ in how the user interacts with them, the infrastructure they can run on, and the use-cases they support.

There are a number of different execution models used by existing systems, including message passing, batch processing, and streaming. These all have their strengths and weaknesses, depending on the use-case. Pydron is using a model similar to batch processing. This fits many astronomy use-cases well, where data is processed, for example, in nightly batches. Section 6.1.6 discusses how Pydron’s execution model could be extended to combine both batch and streaming. Pydron is using message passing for communication.

While there are systems that support for interactive data exploration, these either neglect the need for big-data processing or are limited in their use-cases as they don’t have a turing complete interface. There is a missing key component to combine interactivity with high performance computing: A programming interface that is flexible, easy-to-use, and scalable, so that the iterations are fast enough for iterative work. Pydron shows that this component can be provided with semi-automatic parallelization.

There is a trade-off in existing systems between simplicity and parallelization. Methods that can handle sequential code, such as just-in-time compilation and static compiler optimizations, can only parallelize on shared memory hardware, if they support parallelization at all. Solutions that can benefit from clusters and clouds require significantly more effort from the developer, be it through explicit parallelization APIs, framework specific data types that allow delayed evaluation, or entirely new programming languages. Our approach, however, can parallelize sequential Python code without shared memory, combining the advantages of both. The next two chapters describe how this can be achieved.
3

Pydron frontend

We can make parallel infrastructures available to users that are not necessarily familiar with parallel computing by providing a frontend that allows the developer to write sequential code, which is translated into a form that allows parallel execution.

The Pydron system consists of two main components. The frontend is the part that the developers have direct contact with. It is responsible for analyzing the Python code and translating it into the initial data-flow graph. The runtime part is then evaluating this data-flow graph to achieve parallel execution. This chapter focuses on the former part, with Chapter 4 and 5 focusing on the runtime system. This chapter first describes how the developer interacts with Pydron using Python code and Pydron’s API of two decorators. It then describes in detail, starting with Section 3.3, how the system is using the code and decorators provided by the developer to create the graph representation.

A partial description of the language translation method has been previously published in OSDI 2014 [80].

3.1 Using Pydron

Pydron targets applications where parallelization is possible at a coarse granular level. In the data processing software of the astronomy projects I collaborated with, there are many cases that can be parallelized on such a coarse granular level. A common pattern
that can be seen is that of an “orchestration” function which controls the overall process. This function contains the outer-most loops, for example over individual exposures of the camera, or over regions of the sky. The actual processing is then done in separate functions which typically follow a functional pattern. That is, they take data as arguments, process it, and produce data as output.

When there is one or more functions that contain coarse granular parallelizable loops, and the functions that perform the actual data processing do not have side-effects, then Pydron can be applied with minimal effort. Since this is convenient way to structure data processing software, the code will often already be in this form, or can be easily adapted for it.

The next two sections introduce two examples that commonly appear in astronomy data reduction pipelines to exemplify how Pydron is used.

### 3.1.1 Exposure Calibration

Figure 3.1 shows a simplified example of a calibration pipeline as it might be found in projects that use image data. The raw data from the image sensor goes through a number of processing steps to remove some of the instrument effects as well as environmental effects, for example due to the atmosphere. This would be a typical process that has to be applied before overlapping images can be coadded to increase the signal to noise ratio and to compensate for defect pixels, which is shown in the next section.

The Dark Energy Survey makes several hundred exposures per night. Each raw image is around 300 MB. The three steps would typically take a few minutes each. When reprocessing several years’ worth of observations, even this small part of the entire data processing requires significant resources.

Without the decorators, the code of Figure 3.1 is regular Python code and can be used as long as the dataset is small enough to be processed on a single CPU core, which is sufficient if the data scientist is only interested in a few select exposures.

To use Pydron with this code, the developer has to add two kind of decorators. The \texttt{@schedule} decorator marks the “orchestration” function. This instructs Pydron to translate the function into a data-flow graph representation and to evaluate the graph on a configured infrastructure. The second decorator, \texttt{@functional}, informs Pydron that this
@schedule
def main(raw_images):
    calibrated_files = []
    for raw_image in raw_images:
        mask = defect_pixel_mask(raw_image)
        dk = measure_darkcurrent(raw_image)
        cal_file = calibrate(raw_image, mask, dk)
        calibrated_files.append(cal_file)

@functional
def defect_pixel_mask(raw_image):
    ...

@functional
def measure_darkcurrent(raw_image):
    ...

@functional
def calibrate(raw_image, mask, dk):
    ...

Figure 3.1: Applying Pydron to Python code

function can be executed in parallel, potentially on a different machine. It defines a contract between Pydron and the developer described in Section 3.1.4.

With the decorators in place, Pydron can parallelize the loop without any additional changes to the code. Only the part that has the @schedule decorator will be treated specially by Pydron. The application is still a regular Python application which can be run with a regular Python interpreter. This allows the developer to decide which part of the code should be parallelized.

Pydron will distribute calls to the three functions across the available CPU cores where they are executed inside regular Python interpreters.

This example demonstrates how little effort is required to utilize parallel infrastructures
with Pydron. This is particularly important in data exploration, where it might not be clear from the beginning that parallelization is required at all. If the single threaded execution takes only a few hours, setting up Hadoop and reformulating the problem into a MapReduce form is not very attractive. Pydron can be employed quickly, making it interesting not just for very large problems, but also for small and medium sized ones. The same holds for delayed evaluation systems such as Spark. These could parallelize such a case, but since they require the developer to use framework specific data types and operators, it would require a rewrite of the code, similar to what would be needed to employ a domain specific language. Just-in-time compilation systems like Numba are designed to handle regular Python code, but would most likely fail in this scenario as they would need to translate the entire code within the \texttt{@functional} methods as well, including all functions called from them.

### 3.1.2 Coaddition

The quality of many data products can be increased by combining multiple exposures of the same region of the sky. The basic principle is simple, by registering the exposures with sub-pixel accuracy, the pixel values of the exposures can be added together. This increases the signal to noise ratio as the noise is independent of the exposure. It also compensates for gaps between detectors and defect pixels. The product of the coaddition process is a large image, covering a significant part of the sky. Due to its size, it is split into a mosaic of tiles, keeping the size of a single image file manageable. These tiles can still be several hundred MB each and the mosaic may consist of thousands of tiles.

The overall process can be seen in Figure 3.2. Before the exposures can be added, significant preprocessing is required, including the steps shown in the previous section. In addition, the exposures need to be projected into a common coordinate system, which requires resampling. The coaddition is performed per tile. First, all resampled exposures that overlap with the tile are identified and collected. How this is performed depends on the project’s data organization. It might involve spatial queries to a meta-data database or search data structures built on demand. The final coaddition is then performed on this stack of exposures.
3.1. Using Pydron

```python
@schedule
def coadd_images(mosaic, exposures, coordinatesystem):
    exposure_byid = {}
    for exposure in exposures:
        exposure = resampling(exposure, coordinatesystem)
        exposure_byid[exposure.id] = exposure

    results = []
    for tile in mosaic:
        ids = query_overlapping_exposures(tile.area)
        stack = [byid[id] for id in ids]
        res = coadd_tile(stack)
        results += [res]

    return results
```

```python
@functional
def resampling(exposure):
    ...
```

```python
@functional
def query_overlapping_exposures(cell):
    ...
```

```python
@functional
def coadd_tile(cell, stack):
    ...
```

Figure 3.2: Exposure Coadding

The first part of the process is trivially parallelizable and is mostly CPU bound due to the resampling step, which makes it an ideal candidate for parallelization. The coadding step is a fast operation and bound by the data transfers required to aggregate the overlapping
exposures. The resulting data-flow graph can become quite complex. While surveys typically take exposures in a regular pattern, the observation plan is subject to changes. This might be due to optimizations or due to unplanned events, such as faulty sensors. As a result, there is no fixed pattern of exposure to tile assignments.

While it is possible to implement this process efficiently using MapReduce [81], the irregular nature of the problem makes it challenging to find a good assignment of computation tasks to computation resources that avoids excessive data transfers.

Pydron is well suited for this kind of problem. The process can be implemented in a straightforward way. It makes it convenient and efficient for the developer, not just for the initial development, but also for future changes and code reviews. As in the previous example, two decorators are sufficient to parallelize this code. Pydron can dynamically react to the actual data-flow pattern at runtime. No information about the spatial distribution of the exposures or tiles has to be provided to Pydron. This is a significant benefit in the second phase in particular. A manual strategy might be more efficient than Pydron’s automatic scheduling strategy, but with increasing complexity of the data-flow, the cost of manually finding the ideal distribution pattern becomes prohibitively expensive, especially since it would have to be redone every time there is a change to the code, or the observation plan.

3.1.3 Pydron API

The two decorators \texttt{@schedule} and \texttt{@functional} are the only Pydron specific code that has to be added in order to make regular Python code parallelizable. These two decorators are the entire API of Pydron.

The \texttt{@schedule} decorator informs Pydron that this is the function should be parallelized. Pydron will translate the code of this function into a data-flow graph and execute it on the configured parallel infrastructure.

\texttt{@functional} is a contract between the developer and Pydron. The developer guarantees that the decorated function follows a set of rules that allow Pydron to send calls to this function to other machines (see Section 3.1.4).

There are no such rules for \texttt{@schedule} since Pydron can handle arbitrary Python code. However, Pydron cannot guarantee that the code can be parallelized in a useful way. For example, the algorithm may be inherently sequential, or the functions which perform most of the work may not be marked as \texttt{@functional}. 

42
3.1.4 \texttt{@functional} contract

Pydron parallelizes at a coarse granular level. This implies that the unit of work which is sent to a CPU core for execution may include a potentially large amount of Python code. Python is not well suited for static analysis. It is generally not possible to decide if a certain function call may be parallelizable in a completely automated fashion. Pydron therefore needs additional information from the developer.

With the \texttt{@functional} decorator the developer informs Pydron that a function has a certain set of properties:

- Arguments and return values are serializable.
- No assignments to global variables.
- No interactions with the environment.
- Objects passed as arguments are not modified.

The first one is necessary to be able to send an invocation to a different Python interpreter. This is not just required for parallelization in infrastructures without shared memory, but also for utilization of cores on the same machine, since Python has restrictions for multithreading (global interpreter lock) and lacks support for shared memory between Python interpreters.

The next two ensure that Pydron can track the information flow. Violations of these rules may result in a different evaluation order compared to a single threaded execution, or make differences between the machines visible.

It is up to the developer to decide how strictly these are to be interpreted. For example, if the processing code produces log messages, the developer may accept that these will be in order of actual execution. The log files may also end up on different machines and would need to be collected by other means.

This room for interpretation can work for the developer’s advantage. For example, a data processing pipeline may consist of a series of external applications that take files as input and produce files as output. Python may then be used to orchestrate this process. In the code shown in Figure 3.1, the variables would store only file names, not actual images, and the three \texttt{@functional} functions would invoke the external applications. As long as inputs
and outputs are tracked through their file names, Pydron can still be used to parallelize such code on an infrastructure with a shared file system.

The last rule is required since an in-place change to an object may not affect the original object passed to the function. Serialization for transfer effectively creates a copy of the arguments.

In some cases these rules could be verified automatically, either statically or at runtime. This would reduce the risk of accidental violations. The current prototype will inform the developer if the first rule has been violated, but does not currently include any code analysis for the other criteria.

### 3.1.5 Configuration

Pydron uses a small configuration file to store the necessary information to access the target infrastructure, such as IP addresses and credentials. The configuration options depend on the selected backend which is described in Section 4.4.

The configuration file has to be written only once and may be provided by a system administrator.

### 3.2 System Overview

Pydron becomes active when a function with the `@schedule` decorator is invoked. This triggers the process shown in Figure 3.3. At the first invocation, the function’s code is translated into an initial data-flow graph. This graph is cached for further invocations.

The data-flow graph is a directed acyclic graph consisting of value-nodes and task-nodes. Each expression and statement in the code becomes a task-node, each variable or intermediate result becomes a value-node. There are different task nodes for the various types of expressions and statements. For example, there is a task for binary operations and one for function calls. The data-flow graph and the translation process are described in this chapter from Section 3.3 onwards.

Python is a dynamic language. The absence of static type information, visibility scopes and language properties such as late-binding, make Python a particularly challenging language to analyze.
3.2. System Overview

Pydron uses a dynamic data-flow graph to deal with the dynamic nature of Python. Since the conclusions that can be drawn from static analysis are limited, Pydron is using runtime information as well. At runtime, tasks are executed and the values represented by value-nodes become known. This information is used to dynamically refine the graph. CIEL [69] uses a similar approach to handle data dependent control flow (branches and loops). Pydron also utilizes other information about the data, such as its type and size. This allows Pydron to keep the constraints for the developer to a minimum.

The graph is kept on the main worker which is responsible for graph traversal and scheduling. The graph refinement is driven by the tasks. When inputs to a task become available, the task may decide to change the graph. This may happen with partial inputs only. For example, the call-task requires only the function object to be available to detect the \texttt{functional} decorator. A for-task will require the iterator object for unrolling. This usually allows Pydron to preform most refinements early. For example, if a called function is not the result of another operation, the check for \texttt{functional} can be performed immediately.

\texttt{for x in a+b:
    r += [x*1]}

\textbf{Figure 3.3: Pydron Overview}
A task is considered ready for execution once all the inputs are available and all graph refinements for that task have been performed. At this point the task is passed on to the scheduler.

The scheduler has to assign tasks to workers. We implemented two scheduling strategies, a trivial one that creates an arbitrary assignment and an advanced strategy that can significantly reduce the required data-transfers by assigning tasks to workers that already contain the data products when possible. These are described in Chapter 4 and 5 respectively.

The workers are Python processes. They run for the duration of the entire process. Tasks are sent to these processes by the scheduler. The modular backend is responsible for starting these processes and to establish communication. Since the workers run a Python interpreter, the code that is sent to the workers does not need to be translated but can be executed as is.

As the processing is running, we collect metrics about the task execution time, the data size, and network throughput. These metrics are included in the graph data structure so that the scheduler can improve future decisions. Tasks may also request additional information about their inputs before the actual execution of the task for graph refinement purposes. For example, a call task will inquire if the called function has the \texttt{@functional} decorator, and a \texttt{for}-task needs to know if the iterator has reached its end. These information is fed back to the master node without actually transferring the data products to it.

### 3.3 Language Translation

Functions decorated with \texttt{@schedule} are translated into a data-flow graph. These functions may contain arbitrary Python code.

The translation process parses the code, using Python’s own parser, into an abstract syntax tree. Then a series of operations are performed to reduce the number of language features (see Section 3.4).

The resulting code which is using only subset of the Python language is then translated into the data-flow graph (see Section 3.5).

To avoid translating the same function multiple times, the graph is cached. The graph refinements that will occur while evaluating the graph, are performed on a copy of the
3.4 Language Reduction

Python is designed to be a concise, easy-to-read, language [82]. This reflects in language design choices, such as dynamic typing and syntax constructs such as list comprehensions. Unfortunately, simplicity for the developer does not imply in simplicity for automatic language translation systems.

Python has some redundancy in its language features. For example, list comprehensions could also be implemented with regular loops. We use this to our advantage and perform a series of preparation steps to reduce the number of syntax constructs and language features. These steps do not limit the features Pydron can support, as each step replaces one language construct with code that performs the same operation using only a sub-set of the Python language.

These steps translate from Python code to Python code, avoiding additional intermediate languages. After each step we get valid, executable, Python code. It lies in the nature of these simplifications that they do not make the code easier to read for humans as they typically produce more verbose code.

After all reduction steps have been applied, we receive code with identical behavior to the original code of the developer, but that uses only a small subset of the language (Section 3.4.2). This greatly simplifies the reminder of the Pydron system, since only a minimalistic version of Python has to be supported. Earlier prototypes of Pydron lacked this preparation step which made the translation and the data-flow graph prohibitively complex, and the developer had to be constrained to a subset of the language.

With this reduction steps, the system can remain simple and yet the developers are not restricted in their use of the language.

3.4.1 Reduction Steps

Some of the reduction steps introduce new variable names. We use name mangling to avoid conflicts with variables in the original code, this is not shown in the examples here.
for clarity. For some steps Pydron provides helper functions and values. For clarity, we do not show their \_pydron\_ prefix.

**Python decorators** A good example for a reduction step are Python decorators. A decorator can be applied to a function or class using the @ syntax. The decorator expression must evaluate to a callable object. Instead of assigning the declared function to a variable with the function’s name, the function object is passed to the decorator and the decorator’s return value is assigned to the variable. We can remove the decorator by making this process explicit.

```python
@abc
def sum():
  ...
  sum = abc(sum)
```

Since Python allows arbitrary code in any block where functions can be defined, such as within the body of a class. The simplified code is still valid Python code and will perform identically.

**Import statements** Python has two import constructs with rather sophisticated rules regarding relative imports and assigned names. We simplify these to calls to the built-in \_import\_ function and explicit assignments.

```python
import math
from util import *
  math = __import__('math', globals(), None, None)
  t1 = __import__('util', globals(), None, ('*',))
  for t2 in getattr(t1, '__all__', dir(t1)):
    globals()[t2] = getattr(t1, t2)
```

Essentially, we replace the import statement by a function call. This is a pattern that is used by other steps as well. If Python does not offer a suitable built-in function, we wrap the statement in a function that becomes part of the Pydron library, allowing us to remove the statement from the translated code.

**List / Set / Dict comprehensions** We replace these with loops that build the collection explicitly. Multiple for statements can be combined within a comprehension and
if statements can be used to filter elements. Generators use a very similar syntax (with round brackets), but these are fundamentally different and are treated separately.

\[
r = []
\]

\[
r = [x+1 \text{ for } x \text{ in } y] \Rightarrow \text{ for } x \text{ in } y:
\]

\[
r += [x+1]
\]

Note that the generated for-loop will itself be replaced by a while-loop in a later reduction step.

Generators Generators are replaced by nested functions and yield statements. This preserves the correct variable binding, as variables used within generators become closure variables in Python.

\[
\text{def } x():
\]

\[
x = (v+1 \text{ for } v \text{ in } lst) \Rightarrow \text{ for } v \text{ in } lst:
\]

\[
yield v+1
\]

Generators are a fairly commonly used feature in Python. Since they are effectively syntactic sugar for nested functions, they emphasize the importance to support nested functions and closure variables.

break / continue / return These statements interrupt the regular program flow. Explicit program flow control does map naturally to a data-flow representation. For each of these statements we introduce a variable that stores a flag, indicating if the statement was called. Code that could be potentially skipped over, is wrapped inside an if statement. For return we also keep track of the return value.

\[
\text{for } x \text{ in } y:
\]

\[
\text{if } x == 0:
\]

\[
\text{continue;}
\]

\[
f(x)
\]

\[
\Rightarrow
\]

\[
\text{if } x == 0:
\]

\[
\text{continue_flag = True}
\]

\[
\text{if not continue_flag:}
\]

\[
f(x)
\]

For break this is not sufficient, as it also aborts the loop. This simplification step ensures that the only location break occurs is within an if statement at the very end of the loop body. The code that creates the data-flow graph is aware of this rule and extracts the
condition expression of the final if statement. The created if-task in the graph receives this condition as an additional input which is considered when unrolling the loop at runtime.

return statements within for loops result in break statements in all surrounding loops. Similarly to break statements, we place a return statement as the very last statement in any function body, so that the return value is easily found during the final translation into a data-flow graph.

**Unbound variables** Variables that have not yet been assigned to are unbound. They exist respect to name binding, but any attempt to access their content results in an error. We initialize all variables with a special object that represents unbound variables. This gives us a way to explicitly check if a variable is currently bound or not. Any regular access to a variable is wrapped in a helper method that checks against the marker object and raises an exception if required.

```python
def f():
    print x
    x = 1
```

```python
def f():
x = __unbound__
print __check__(x)
x = 1
```

During translation into the final data-flow graph these checks don’t have to be translated, as the tasks perform this check internally. We introduce them to the Python code to ensure that the behavior of the code does not change if the reduced code is executed as regular Python code. Keeping the behavior strictly identical is a key for testability for all reduction steps, even if the later translation to a data-flow graph would restore identical behavior.

**locals()** Most built-in functions can be invoked like regular functions with a call-task. `local()` is an exception as it inspects the current stack-frame. It returns a dictionary with all the variables in the function from where it is invoked. If we were to invoke it from a call-task, it would return variables local to an internal function of Pydron. We therefore have to collect the required information before we translate the code into the data-flow graph.
3.4. Language Reduction

```python
def f():
x = 1
l = locals ()
y = 1
⇒
def f():
x = 1
y = __unbound__
l = __locals__(x=x, y=y)
y = 1
```

The helper function `__locals__` checks the variables against the unbound marker to build the same dictionary that the built-in function would return.

**Closures** Python supports nested functions that can access variables from enclosing functions. Python replaces these variables with cell objects that store the content of the variable. We use the same approach to remove closures. Cells are created explicitly and passed to the function as parameters. Every access to such a variable is changed to access the cell’s content instead. This turns these closure variables into regular local variables.

```python
def outer():
    x = 0
def inner():
    print x
⇒
def outer():
    x = __new_cell__('x')
    x.cell_content = 1
def inner(x):
    print x.cell_content
    inner = __cellwrapper__(inner, (x,))
```

Adding these additional parameters changes the function’s signature, which would require a change to any code that is invoking this function. This is not possible, as these locations are not known and might not even be within the code that is translated. To avoid this issue, we wrap the function object in a callable helper object that takes the extra parameters in the constructor and passes them to the actual function together with the regular parameters. This way, the signature remains the same from the perspective of the caller, while the implementation receives the cell objects as additional arguments.

If the outermost translated function has such variables, a slightly modified process is used where we access the cell-objects provided by the Python interpreter.

**Classes** Unlike most languages, where the body of a class allows only declarations, Python allows arbitrary code directly within the class which is executed when the class
is declared. Variables local to those code blocks become members of the class. We can use this to replace class bodies by function bodies and create the resulting class object explicitly.

```python
def t1():
x = 1
class MyClass(base):
x = 1
def f():
    pass
    return {'x': x, 'f': f}
MyClass = type('MyClass', (base,), t1())
```

Due to differences in name binding between code in classes and functions, it is important that closures have been replaced at this stage.

**Global variables** From a data-flow perspective, global variables are treated differently from local variables. While local variables are translated to edges between the tasks, accesses to global variables have to be performed on the actual Python modules those variables belong to. In Python code, however, there is no difference in the way those variables are accessed. We prepare for the later translation into the data-flow graph by making all accesses to global variables explicit. During the final translation into the graph, calls to `__global__` and `__assign_global__` are replaced by corresponding task nodes. This is important as assignments to global variables introduce synchronization points.

```python
def f():
    print x
    return {'x': x, 'f': f}
```

**Other reductions** There are a number of further reduction steps that remove language features which can be replaced by simpler language features:

- `print` statements become calls to the `print` built-in function.
- Slice notation for array indexing are replaced by `slice` objects.
- `exec` statements are replaced with a function calls to a helper function.
- `assert` statements are replaced with `if` and `raise` statements.
3.4. Language Reduction

- *with* statements are replaced by *try catch* constructs.
- Expressions for default values are explicitly evaluated in the parent block and the functions are wrapped with a helper that injects the default values if the caller does not provide these arguments.
- *lambda* expressions are replaced by functions.
- *if* expressions are replaced by *if* statements.
- Conditional evaluation of boolean expressions is modeled with nested *if* statements.
- Assignments with multiple targets are split into separate assignments.
- *for* loops are replaced with *while* loops, using an explicit iterator.

**Reduction step execution order**  Special care must be taken to execute the steps in the correct order, as the produced code must not include features that have already been removed.

Certain reductions are particularly difficult as they have circular dependencies. For example, a *lambda* expression cannot be replaced by a function definition if the *lambda* expression occurs within a *list* comprehension since only expressions, but not statements, may occur within a comprehension. For the same reason, we cannot replace a comprehension with a loop if it occurs within a *lambda* expression. Reducing such nested structures must occur from the outermost block towards the innermost block.

### 3.4.2 Reduced Language

After the reduction is completed, we are left with code that contains a small set of language features:

**Statements:** function definitions without default values or decorators, assignment to local variables, *while*, *if*, *yield*, *try*, *catch*, *finally*, and *raise*.

**Expressions:** *unary*, *binary*, function invocations, attribute access, subscript access, collection initializers, and literals.
Chapter 3. Pydron frontend

There are also `break` and `return` statements, but these can only occur at designated places.

The reduction has been designed with the translation to the data-flow graph in mind, but the resulting feature set is similarly to the language feature set source-to-source translators limit their users to. While no attempt has been made to verify this, it seems reasonable to assume that source-to-source translators could support additional language features, such as closures, with a similar approach. However, translation of Python code into a statically typed language, such as C, also implies other constraints, such as successful type interference, that cannot be avoided by reducing the number of language features.

Figure 3.4 shows the completely reduced code from these three lines of Python code:

```python
def demo(data = []):
    tmp = sort(data, key=lambda i: i.v)
    return [x for x in tmp if x.w > 0]
```

Clearly, the reduction of language features comes at the cost of increased verbosity, as it makes many of the operations explicit that the Python hides from the developer. Since this code is not shown to the user, but only used for translation, the verbosity does not affect usability. It also emphasizes the complexity that results from supporting the Python language in all its aspects.

### 3.5 Data-flow Graph generation

This section and the following sections in this chapter are partially taken from [80].

The intermediate data-flow graph used by Pydron is directional, acyclic, and bipartite. There are two types of nodes: Value-nodes which represent immutable data and tasks which represent operations on data.

Task-nodes have input and output ports to which value-nodes are connected. All tracked data required by an operation are inputs to the operation, all data produced by an operation are outputs. Each task is only executed once.

Figure 3.5 shows the data-flow graph for a simple expression. In general, variables become value-nodes. Expressions and statements become task-nodes. Nested blocks are translated into sub-graphs. Intermediate values in expressions are also represented by value-nodes. Those have no direct correspondence with a Python variable, but behave no differently otherwise.
3.5. Data-flow Graph generation

```python
list$U0 = []
def demo$U0(data):
tmp = __unbound__
compare$U0 = __unbound__
iterator$U0 = __unbound__
lambda$U0 = __unbound__
list$U1 = __unbound__
attribute$U1 = __unbound__
listcomp$U0 = __unbound__
x = __unbound__
returnvalue$U0 = __unbound__
interrupted$U0 = __unbound__
def lambda$U0(i):
    attribute$U0 = __unbound__
    attribute$U0 = __check__(i).v
    return __check__(attribute$U0)
tmp = __read_global__('sort')(__check__(data),
             key=__check__(lambda$U0))
listcomp$U0 = []
iterator$U0 = __iter__(__check__(tmp))
while __hasnext__(__check__(iterator$U0)):
    (x, iterator$U0) = __next__(__check__(iterator$U0))
    attribute$U1 = __check__(x).w
    compare$U0 = (__check__(attribute$U1) > 0)
    if __check__(compare$U0):
        list$U1 = [__check__(x)]
        listcomp$U0 += __check__(list$U1)
    returnvalue$U0 = __check__(listcomp$U0)
    interrupted$U0 = '3_return'
    return __check__(returnvalue$U0)
tuple$U0 = ('data',)
tuple$U1 = (__check__(list$U0),)
demo = __defaults__(__check__(demo$U0),
             __check__(tuple$U0), __check__(tuple$U1))
```

Figure 3.4: Reduced code example
In workflow systems such as Taverna [71], there are also other types of edges. In Pydron dependencies between tasks can only result from data dependencies.

Functions with the @schedule decorator are translated the first time they are invoked. Pydron uses Python’s built-in parser to create an abstract syntax tree of the function’s code. This tree is then traversed several times for the language reduction steps. The generation into a data-flow graph occurs in a final traversal.

Due to the previous reduction, the translation only needs to support a small set of the Python language, without containing the developers in the language features they may use.

### 3.6 Static Single Assignment Form

Python variables can be reassigned. This conflicts with the property that value-nodes represent immutable data. Therefore a one-to-one relationship between variables and value-nodes is not possible. We translate the Python code into a static single assignment form [83]. A Python variable is represented by a series of value-nodes, each representing the content the variable would hold for a period of the time in a sequential execution of the code.

Figure 3.6 shows an example. The variable $x$ is assigned twice. The value node $x_1$ represents the content before the $+=$ operator is executed, $x_2$ represents the content after.
Once this operator has been executed, both $x_1$ and $x_2$ are known and the scheduler (see Chapter 4) which will be able to schedule both calls to $f$ for parallel execution. We don’t show the subscripts explicitly in the other figures, as the order can be derived easily from the graph structure.

### 3.7 Attribute and Subscript

When used as a right-hand-side expression, both attributes and subscripts are translated to a task which receives the object as an input. For attribute access, the name of the attribute is stored in the task. In case of a subscript, the indices are also passed as inputs. Pydron supports all indexing constructs, including slicing. Figure 3.7 shows a simple example with both attribute and subscript used as a right-hand-side expression.

Both attributes and subscripts can be used as a left-hand-side expression as well. Those tasks have the assigned value as an additional input.

### 3.8 Function Calls

Functions are first-class objects in Python. The invoked function is represented by a value-node since the function may itself be the result of an operation. This value-node is an
input to a call-task, together with the arguments passed to the function. The return value is again a value-node. Figure 3.8 shows a simple example.

Pydron supports all of Python’s language features for function calls, such as keyword arguments and argument lists.

To improve the readability of the data-flow graphs, we show a slightly compacted form. Instead of showing the function calls as in Figure 3.8, we hide the input for the function object and name the call task-node by the function. We will also hide intermediate value-nodes, as shown in Figure 3.5, from expressions. Instead we directly connect the two tasks.

### 3.9 Conditional Statements

The translation of the if statement makes use of the dynamic data-flow graph. The complete if statement is initially translated into a single task. The condition is an input to this task. Both the if and the else section are translated individually into sub-graphs. During translation of the body and else blocks, the variables read and assigned are kept track of. They too become inputs and outputs of the if-task.

A variable in Python can be unbound if it is assigned in only in the if-block but not in the else-block, or vice versa. In the data-flow graph each value-node must be the output of a task. For such situations a special task is added to the graph which produces a special value to represent unbound variables. The scheduler is aware of value-nodes with an unbound content and will produce the same exceptions on an attempt to use the value-node as Python would when using an unbound variable.

### 3.10 Loops

The translation of while loops uses the same techniques as for conditional statements. The loop body and the optional else block are translated into sub-graphs. The complete loop construct is translated initially into a single task. The condition, in case of a while loop is an input to this task.

At the end of the body sub-graph the loop task itself is added to form a tail-recursive pattern. This may seem to enforce sequential execution, and indeed it will do so, unless
3.11. Exception Handling

Exception handling statements such as `try-except` or the `with` statement can be translated as well.

At first, exceptions seem to forbid any parallelism as every operation could potentially throw an exception. The decision if an operation is to be executed can only be made once the previous operation has finished. We can still achieve parallelism by using speculative execution [84]. An operation is executed even if it is not clear if an exception in a previous operation may occur. This is possible when we execute a task without side-effects or in-place modifications and discard its outputs. This has the same observable behavior as if we would not have executed the task at all.

If a task does have side effects, this translates into a synchronization point which forces all previous operations to complete before it. In this situation, we know if any of the previous operations raised an exception.

The cost of this method is that we potentially waste significant resources on speculatively executed tasks should an exception occur. If we assume that exceptions are used in rare...
scenarios and not for regular control flow, then exception handing has little impact on the potential parallelism.

### 3.12 yield Statement

The `yield` statement is special since it transforms the function in which it appears into a generator. When the function is invoked it pauses and returns an iterator to the caller. When elements are consumed from the iterator, the execution of the function proceeds from `yield` statement to `yield` statement.

The `yield` statement can be translated into a data-flow equivalent which is treated specially by the scheduler. Between reaching a `yield` statement and the next consumption of an element on the iterator, only tasks can be executed which are free of side-effects and perform no in-place modifications. This is similar to exception handing as we cannot say for sure if another element will be consumed by the iterator, making the execution of any operation after a `yield` statement speculative.

### 3.13 @functional Decorator

Functions decorated with `@functional` are not changed at all. The only effect is that the decorated function is marked as functional.

At runtime, we use this mark to remove the sync-point of call-tasks (see Section 4.2.2).

### 3.14 Graph Attributes

Task- and value-nodes have a number of attributes assigned to them that store additional information for the scheduler.

#### 3.14.1 Tick

All tasks receive a tick. A tick is a tuple of integers that represents the point in time where this operation would be executed if run sequentially as regular Python code.
The statements and expressions are assigned an integer in the order they appear in the code. Statements that contain their own block, such as loops, get a number for the location of the loop, a number that is incremented at runtime with the loop counter, and a number for each statement within the body. These numbers are concatenated into a tuple (see Figure 3.10).

Lexicographic comparison of ticks gives the sequential execution order of all operations.

### 3.14.2 *sync-points*

The *sync-point* property is assigned during translation to each task-node that is not known to be free of side effects. A side-effect is any change to the state of the system that is not explicitly tracked in the data-flow graph.

For example, a call to an unknown function may change global variables internally, later another function could read this value. This results in an information flow which is not visible in the graph. This could lead to wrong results should the scheduler decide to execute these calls in a different order than they would if executed sequentially.

If a task is marked as a *sync-point*, the scheduler will delay the execution of this task until all tasks that would run before it in a sequential execution have finished. Tasks that come after it in sequential execution order are delayed until the marked task has finished. This ensures correct execution order for tasks that may have untracked dependencies. Tasks with this mark must also run on the main node, as these hidden information flows cannot be transferred over a network connection. The scheduler uses the ticks to ensure correct ordering.

Tasks with the *sync-point* mark can result either from Python expressions that are directly recognizable as having side-effects, such as an assignment to a global variable, or from
expressions where static information is not sufficient to ensure that the operation is free of such behavior. An example for the later are all function calls, since the invoked callable is typically not identifiable with certainty from the code alone, making it impossible to check for the @functional annotation.

Sync-points prohibit parallelism. Therefore they need to be avoided if possible. At runtime, when the invoked function object becomes known, the sync-point mark is removed if the function is decorated with @functional.

3.14.3 no-transfer

By contract, all functions with @functional decorator receive arguments, and only produce return values, that can be serialized for transfer over a network connection. All others run exclusively on the main worker, and therefore do not require data transfer.

However, internally we require a third case to handle iterators. Iterators are common objects as they result from for loops. Unfortunately, iterators are not serializable. We could treat all functions that operate on iterators as sync-points, but this would defeat parallelism in most practical use-cases.

We use a no-transfer flag for value nodes that cannot be serialized. This flag forces all tasks that have this value as an input to run on the worker that stores this value. However, it does not imply that these tasks have side-effects and must run with strict sequential execution ordering.

This mark is only used for value-nodes that result from the reduction steps that reduce for loops to while loops, and not for code directly written by the developer. This gives us sufficient control to guarantee that we do not run into the issue where a task may have two no-transfer inputs located on different workers.

3.14.4 quick

While some tasks execute arbitrary code provided by the user, many tasks perform small operations, such as unpacking a tuple, or reading a global variable from a module.

During translation, these tasks are marked as quick. This is a hint for the scheduler that these tasks take no significant time to execute, compared to the overhead introduced by Pydron.
The scheduler will then try to avoid any costly scheduling operations by executing these tasks immediately, bypassing the scheduling algorithm. In most cases, all tasks in the critical path for loop unrolling can be marked as *quick*. This significantly speeds up loop unrolling, which may expose parallelism. Especially with a large number of workers it becomes important to unroll loops fast enough to keep them all occupied.

### 3.15 Summary

The simple form of an acyclic directed graph makes the data-flow representation a good choice for automated systems, as it makes the communication between the tasks explicit, allowing the system to reason about the parallelizability of the process. Conditional control flow, such as loops, are modeled as sub-graphs which allows us to retain the acyclic nature of the graph representation. The only constraints not directly modeled in the graph structure are *sync-points* and nontransferable data products. These require special treatment by the runtime system which is a small price given that they play an important role in Pydron’s support for arbitrary Python code.

All language features of Python can be translated and the resulting data-flow graph preserves the semantics. The test set that is part of the CPython interpreter could be used in the future to test the correctness of Pydron. Some deviations are known, for example Pydron’s internal structure is visible when the call stack is inspected or the code of translated methods is accessed through the `inspect` module. These features are not commonly used in regular Python code and often only used for debugging purposes. It is unclear if transparency of Pydron in this respect would even be in the user’s interest. There is no known hard constraint had would prevent transparency of Pydron even for inspection, but it would likely require Pydron specific changes to the interpreter that we were able to avoid so far.

The creation of the data-flow graph from Python code represents the static analysis part of the language analysis. The next chapter describes how runtime information is utilized while evaluating the graph to refine the initial graph, augmenting the static analysis to account for the dynamic nature of Python. Only then is the graph sufficiently detailed that parallel tasks can be identified and scheduled to workers provided by the backend.
The static analysis described in the previous chapter translates the Python code into an initial data-flow graph that encodes all the relevant information we are able to extract from the code. Since Python is a dynamic language, this is not sufficient for parallelization by itself. This chapter explains how we can get the required additional information at runtime, incorporate it into the data-flow graph, and use it to parallelize the process.

The runtime component of Pydron takes the data-flow the graph that results from the translation phase and executes it on the configured infrastructure. Since the initial data-flow graph includes all the information Pydron can extract from the source code, the runtime system does not need to concern itself with the code of the \texttt{@schedule} function anymore. This decouples the runtime system from the frontend.

The system operates by traversing the graph and identifying tasks that can be scheduled for execution (Section 4.1). The information that can be extracted from the source code alone is limited, therefore the runtime system is using knowledge about the actual data products that are passed between tasks to refine the graph, resulting in a more detailed description of the processing (Section 4.2). Tasks that are ready for execution are assigned to workers for execution. In Section 4.3, we present the scheduling process in general as well as a first simple scheduling strategy. A more advanced scheduler is described in Chapter 5. Pydron uses a modular backend that provides a Python interpreter for each core available in the configured infrastructure and the means for communication between them. The supported backends and the communication system are described in Section 4.4. Finally, we show
results in Section 4.5 to demonstrate Pydron’s ability to scale on multi-core, cluster, and cloud infrastructures using two real-world use-cases.

A partial description of the runtime system and the results of Section 4.5 have been previously published in OSDI 2014 [80].

4.1 Graph Traversal

As the graph is evaluated, the data products represented by the value-nodes become available, either because they are passed as arguments to the function with the `@schedule` decorator, or because the task that produces the data product has been executed. This information is stored in the graph.

When tasks finish execution, and data products become available, two kinds of events are triggered:

- A downstream task may become ready for execution if all inputs are now available.
- A downstream task may want to perform a graph refinement operation based on this data.

The exact conditions that need to be fulfilled for a task to be allowed to execute or for a graph refinement operation to happen are critical to ensure that we get the same results as if the code would have been executed sequentially. These rules decide which tasks may run in parallel.

4.1.1 Conditions to be ready for execution

The main condition for a task to be ready for execution is that all its inputs are available, but this is not sufficient.

There might be tasks in the graph that got flagged as `sync-points` during translation (see Section 3.14.2). These tasks may have information flows that are not represented by a graph edge. For example, they may write to a global variable that another task may later read. Since this is not represented by an edge, the graph structure alone does not ensure the correct order of execution. Regular tasks have only explicit data-flows. `sync-points`
are the exception in case the structure of the graph may not be sufficient to ensure the correct order of execution.

This leads to an additional condition that tasks must fulfill before they may be scheduled for execution.

Let $T$ be the set of all tasks, $T_r \subset T$ the set of regular tasks not marked sync-points, and $T_s \subset T$ the set of tasks marked as sync-points. A regular task $r \in T_r$ may only run if

$$\forall s \in T_s: r < s \text{ or } s \text{ has been executed.}$$

A sync-point task $s \in T_s$ may only run if

$$\forall t \in T: s \leq t \text{ or } t \text{ has been executed.}$$

Where tasks are compared by their tick. The tick establishes an order where $t_1 < t_2$ implies that $t_1$ would run before $t_2$ if the code would run as regular sequential Python code, and not through the Pydron system.

The rules force sync-points to run in the order equivalent to sequential execution. All tasks that are sequentially before a sync-point must have been completed. Tasks that are sequentially after a sync-point may only start executing once the sync-point has been executed. Therefore sync-points can be used to ensure correct execution ordering when the graph structure might have incomplete knowledge of the actual information flow.

This comes at the cost of prohibiting parallelism. Due to the strict order enforcement, a sync-point cannot be run in parallel with any other task, which implies that only one worker will be busy while all others have to wait for it to finish.

If a sync-point appears in the body of a loop, the loop cannot be parallelized. After unrolling the loop iterations, there will be one sync-point task per iteration. Each forces all previous iterations to complete before it, and forces all later iterations to wait. This makes it impossible to parallelize over loop iterations. Statements within an iteration, such as nested loops, may still be parallelizable.

Clearly, sync-points have to be avoided. One of the goals of refining the graph is to remove the sync-point mark of tasks using the available runtime information (see Section 4.2.2). Unfortunately, this is not always possible and it is ultimately the developers responsibility to ensure that all operations within relevant loops follow the @functional contract and are marked as such.
If the sync-points are not within the data processing loops, the effects are less dramatic, as loops before and after a sync-point can still be parallelized. This could happen, for example, if progress reports are written into a database between major steps of the processing. As long as these phases without parallelization are short, the idle time of the workers is neglectable.

Sync-points allow the developer to mix parallelizable code with non-parallelizable code transparently. This is a convenience for the developers at one side, but also implies that the developer needs to be aware of what code can be parallelized. Pydron does not limit the developer to code that is guaranteed to be parallelizable. Due to the limitations of static analysis, this would be very constraining. This reflects in other aspects of Pydron as well, for example, Pydron does not prevent the developer from implementing inherently sequential algorithms.

The traverser is maintaining all sync-point ticks in a sorted tree structure and maintains a counter of inputs for each task that counts down as inputs become available. This results in a $O(\log n)$ update cost after a task has finished.

### 4.1.2 Conditions for graph refinement

Refining the graph is the way Pydron learns more about the developer’s code and the problem at hand based on runtime information. The more information we have encoded in the graph, the better Pydron can parallelize. For example, by having fewer prohibitive sync-points or by having the iterations of a loop unrolled. It is therefore in our interest to perform refinements as soon as possible.

Each task declares if it would like to perform a refinement operation. The refinement operations depend on one or more of the tasks input values, which is how the runtime information is entering the refinement process. These inputs are a subset of the task’s regular inputs. The tasks declare which inputs they need for the refinement operation. The traverser is using similar rules as to decide when a task is ready for execution to decide when a task is ready for refining, but applies them to the requested subset of inputs. The condition regarding sync-points is different. As before, let $T$ be the set of all tasks and $T_s \subset T$ the set of tasks marked as sync-points. A task $t \in T$ may only be refined if

$$\forall s \in T_s : t \leq s \text{ or } s \text{ has been executed}.$$
This is similar to the rules for task execution, but *sync-point* tasks may refine as soon as previous *sync-points* have finished and must not wait for regular tasks to finish.

We can allow this because graph refinement operations, independent of the presence of the *sync-point* flag, do not change any state other than the graph. The graph refinement operations are part of Pydron and are not written by the user. Execution of *sync-point* tasks that come earlier may still have side-effects which affect the inputs required for refinement. Therefore we cannot refine until all earlier *sync-points* have completed. Since the refinement itself has no side-effects, we can run it out-of-order relative to regular tasks.

This change in rule for refinement is critical. Many tasks, including all function invocations, will initially have a *sync-point* flag. If we would use the same rules as for execution, then we could only perform the refinement operation that removes this flag once all tasks that are sequentially before it have completed. This would annihilate the main benefit from removing the *sync-point* in the first place, which is to remove the execution ordering constraint.

### 4.1.3 Conditions to end traversal

The usual way for a translated function to produce a result is by a returning a value. The data-flow graph contains a value-node that represents the function’s return value. Until we have the corresponding data product, traversing must continue.

This is not sufficient, however. The *schedule* function which is translated and traversed is not bound by the rules of the *functional* contract. In particular, the translated function may use side-effects, such as assigning to global variables or writing files. When ending the graph traversing, we have to ensure that these operations have been completed, which implies that traversing may only finish once the return value is known and all *sync-point* tasks have run.

One way to achieve this would be to continue until all tasks of the graph have finished. But this may lead to more task executions than strictly required. The minimum set of tasks that need to be executed are all *sync-point* tasks as well as all tasks from which there is a path to either the return value-node or to an input of a *sync-point* task. This avoids executing tasks where we know that the result is not used.

While correct, in practice this rule has shown to be counterproductive. Since Python is an imperative language, it executes code independent of the usage of the result. Developers
have developed a habit of wrapping code in appropriate if-statements to avoid unnecessary computations. Actual data processing code therefore rarely results in a graph where some tasks are unnecessary. However, when developers are exploring Pydron this occurs frequently as developers tend to use simple loops to familiarize themselves with Pydron. The return values of the functions invoked within the loop body are typically not collected for this purpose. The invoked functions usually contain print statements to check the operation of Pydron. Using the above rules to terminate traversal, Pydron detects that none of these calls is required, as their return value is ignored, and does not execute them. Which then tricks the developer into believing that the system is not working. Strictly speaking, print statements violate the @functional contract, as they are a side-effect, but for convenience Pydron collects the standard output from all workers and forwards it to the developer’s machine.

Based on this observation, it is actually beneficial to continue traversing the graph until all tasks have been completed, even if their outputs are not used. It makes the behavior more intuitive, as it is closer to the behavior of Python itself. For actual data processing the advanced termination rules rarely encounter tasks that do not need execution, as the code does not include unnecessary calculations.

Despite this, there is still need to abort traversal early in failure scenarios that cannot be recovered. Since there might be substantial cost associated with the used infrastructure, it is important to terminate traversal and not continue wasting resources on further processing. The traverser keeps track of all tasks that have been handed to the scheduler and have not yet finished. The scheduler is informed that these task executions are no longer required. The scheduler can then remove them from the queue if processing has not yet started. If the tasks are currently running, the worker is informed. Unfortunately, Python has no feature that would allow terminating an arbitrary long-running Python call, short of forcefully terminating the worker process.

### 4.1.4 Purging data products

Given the targeted applications, the data volume handled by Pydron can be significantly larger than the memory of a single machine. The traverser identifies intermediate data products that are not going to be required anymore. When all tasks that have a certain value-node as input have completed their execution, the data product for that value-node
Graph refinement is the process of changing the graph based on runtime information. Each task may refine the graph for its purpose, although not all tasks do.

The refinement operations extend the initial translation of Python code into a data-flow graph, described in Chapter 3. The translation is limited to static analysis and can only use information that can be inferred from the code alone. Whenever this is not sufficient, the translator generates task-nodes that contain refinement operations. The part of the analysis that could not be done statically is therefore delegated to refinement operations. We describe them here as they are part of the runtime system, conceptually they can also be seen as part of modeling the Python language in a data-flow graph.

Unlike the translation process, the refinement operations have access to the actual data products. This makes them inherently more powerful, but since they are performed at execution time, and with high frequency, they are also a main contributor to the overhead introduced by Pydron. Only operations that strictly require runtime information are performed with refinement operations.

4.2.1 Rules for refinement operations

All graph refinement operations must follow a set of rules that ensure that the graph remains consistent and that the refinement operations of different tasks cannot negatively
effect each other. These rules allow the scheduler, and other components of the runtime system, to reason about the entire graph and not just the part that has been fully refined since they can make assumptions on the changes that may or may not occur in the future.

- No task node and no value node in the graph may be changed with the exception of the task node which is performing the refinement operation. This limits the graph changes to the vicinity of the acting task node and ensures that the operations of separate tasks cannot affect each other.

- Input dependencies of a task may be removed, but not added. If edges to inputs could be added, the purging described in Section 4.1.4 would not work as it could not make the assumption that a certain data product is no longer required. It would also pose a risk adding inconsistencies such as circular dependencies.

- Outputs must remain connected. Since a data-value may be used by other tasks as an input, we must always preserve a way to produce this value. See Section 4.1.3 for a discussion on tasks that produce unused data products.

In particular, the following operations are allowed:

- Change a property of the task node, such as removing the sync-point flag.

- Remove inputs that are no longer required for the execution of the task.

- Remove the task node and insert a sub-graph in its place. The sub-graph may have an arbitrary valid structure, but the way it is connected to the existing graph is constrained: It may only use input connections to value nodes that the original task also used as inputs. It must have an output connection to each output of the original task. The two operations above can be seen as special cases of this operation.

### 4.2.2 sync-point Removal

Refinement of a call-task attempts to remove the sync-point flag from the task-node. For that it requires the invoked callable, which is an input to the call-task.

The refinement operation checks if the callable has a @functional decorator, in which case the sync-point flag can be removed. It also checks the callable against a white-list that contains functions that are known to follow the @functional contract but lack the
4.2. Graph Refinement

**Figure 4.1: Inline Substitution**

```
@schedule
def do(a, b):
    return g(a) / b
```

decorator. The white-list is needed as it is not possible for the developer to add a decorator to a Python built-in function or a function of a library.

*Call*-tasks are frequent since the data processing code is typically in separate methods, and because many language features are mapped to function calls during language reduction. All *call*-tasks are initially *sync-points*. Without this refinement parallelization would be impossible.

There are no changes to the structure of the graph.

### 4.2.3 Inline Substitution

If, during the evaluation of the graph, an invoked function is found to be decorated with `@schedule`, then this function can be translated to a data-flow graph as well.

Instead of invoking the original Python method, the *call*-task is replaced by the function’s graph. This corresponds to the inline substitution optimization performed by compilers [83]. Inline substitution can expose additional parallelism as shown in Figure 4.1. The call to `do` is inline substituted, allowing for parallel execution of `f` and `g`, even though `f` is required to calculate an argument of the call. This works since the substitution can be performed as soon as the invoked method is known, even before the arguments are calculated.

Inline substitution is optional and the scheduling strategy may decide not to inline a call and instead run the original, untranslated, function to control the granularity of the parallelization. Only functions that are marked as `@schedule` can be inlined as only those
are translated into a data-flow graph, which allows the developer to control which functions may be inlined.

There are also other ways in which the granularity could be controlled, for example in case of nested loops the inner loops could be executed as a single task. Inline substitution is a component of a larger optimization effort for automatic granularity control, and detailed study of it is left as future work (see Section 6.1.7). Some early experimentation showed significant potential in reducing the overhead of Pydron when the granularity is too fine. The experiments shown in this thesis do not make use of inline substitution.

4.2.4 Loop run-rolling

During the reduction phase (Section 3.4) all loop constructs are replaced with while-loops. These become while-tasks (see Section 3.5) in the data-flow graph. Each such task contains two sub-graphs, one for the body of the loop and one for the else block. Since we use tail recursion, the body sub-graph contains a copy of the while-task itself.

A while-task is never executed, it only performs graph refinements. The while-task replaces itself with one of the sub-graphs. The sub-graph is selected based on two inputs of the task, the condition and the break-flag. The condition is the evaluated expression of the while-statement, and the break-flag indicates if the loop was aborted due to a break or return statement within the body. Once the loop is completed, the while-task will replace itself with either the else sub-graph, which does not contain another copy of itself, or an empty sub-graph.

Python does not demand that the condition value of a while loop is a boolean type. It may be an arbitrary Python object and its truth value is decided according to the rules of the Python language specification. A data transfer is required since the refinement happens on the main worker, where the graph structure is maintained, but the object may be located on a different machine. Since the object could be arbitrary large, Pydron evaluates the truth value of the object remotely and transfers only a boolean value.

The sub-graph is selected as follows:

- If the condition is true, there is another iteration. The while-task is replaced with the body sub-graph.

- If the condition is false, there are no more iterations. The while-task is replaced with the else sub-graph.
• If the break-flag is set, independent of the condition, the while-task is replaced with an empty sub-graph.

The sub-graph is then placed in the graph and the while-task is removed.

All the inputs of the sub-graph are connected to the inputs of the while-task. The translator has made sure that the while-task has all inputs of the sub-graphs, therefore we can make all connections without violating the refinement rules.

All outputs of the while-task need to be connected. If the sub-graph has an output for it, this output will be used. This may not always be the case. For example, an output may be the result of a variable assignment within the body of the loop. However, if no loop iteration occurs, then this assignment does not occur either, and the content of the variable remains unchanged. For each output of the while-task there is also a corresponding input. This input represents the value before the while-task is executed. If the selected sub-graph does not provide this output, then the output is connected to the corresponding input. If that variable has not been assigned before, the translator makes sure that it is initialized to the special value that represents an unbound variable. Any attempt to use this value will cause an exception.

The sub-graph of the body contains at its end the tasks to evaluate the condition expression as well as another while-task that represents the tail of the loop. This is prepared during translation and does not require further handling during refinement.

Figure 4.2 shows how the graph from a while loop looks initially and after three replacements. This method can still allow for parallelism, even if the loop iterations are not completely independent of each other. If \(g\) executes faster than \(h\), the while loop will unroll faster than a single \(h\) executes, allowing for several parallel executions of \(h\). Even if \(g\) is slow, \(g\) can run in parallel with the \(h\) call of the previous iteration.

The summation is still executed sequentially, without making any associativity assumptions on the potentially overloaded plus operator.

While Pydron can detect parallelization between all statements that do not have data dependencies, most use-cases will gain parallelization from loops. This makes loop unrolling a particular important refinement for the operation of Pydron.
4.2.5 if-tasks

The procedure to refine if-tasks is similar to that of while-tasks. As with while-tasks, there is a body and an else sub-graph. The if-task replaces itself with either of them. The rules for connecting inputs and outputs are identical, as the same issues regarding outputs can occur if a variable is assigned in only one of the two code blocks.

Since if-tasks do not need to be aware of the break-flag, the refinement operation only requires the condition value as an input.

4.3 Scheduling

The traverser hands tasks to the scheduler once they are ready for execution (see Section 4.1.1). The backend is responsible for the acquisition of resources which are provided to the scheduler as workers (see Section 4.4). It is the scheduler’s job to assign the tasks to the workers.
4.3. Scheduling

4.3.1 Worker Process Model

There are two common models to schedule tasks.

One that is commonly employed on clusters and grids is to split the work into units that can run in parallel and then hand these to the management system of the infrastructure. These management systems have a queue of tasks, and then use various strategies to assign them to physical infrastructure. The goal is to minimize the time each task spends in the queue and to maximize the utilisation of the hardware. Some of these systems, such as OpenLava [85], allow dependencies to be formulated between tasks to ensure correct order of execution. They can also ensure that a group of tasks run at the same time, which can be used if they require communication among themselves.

This seemingly maps nicely to Pydron, as it is aware of the dependencies between tasks due to the graph structure. However, while these queueing systems handle dependencies between tasks, they may not handle data transfer.

Moreover, while Pydron is designed for coarse granular problems, where tasks are long running, there are also numerous smaller tasks, that handle meta-data. For example, there might be code that performs string concatenation operations to create the configuration for a larger processing task. It is one of the strengths of Pydron that it does not require the developer to strictly separate the actual computation from code that may only assemble a file name. As long as there is a sufficient number of long running tasks to keep the workers busy, the overhead introduced by the smaller tasks is compensated for. If we were forced to go through the queueing system for each single Python expression, the overhead would become prohibitively large, especially since small tasks are in the critical path for loop unrolling. If these small tasks cannot be executed quickly, loops cannot be unrolled fast enough to uncover large tasks at a sufficiently fast rate to keep all workers busy.

Lastly, while there are such systems available on clusters and grids, Pydron also supports computation on the local machine and on cloud infrastructures. Neither commonly has such a system in place.

Pydron therefore uses long running worker processes. Instead of acquiring resources for each task individually, Pydron allocates resources once at the beginning. It starts one Python interpreter for each core it has available for processing. These processes remain running until the entire processing run has completed. Pydron then sends tasks to those worker processes for execution.
This implies that Pydron does its own scheduling. Since we have more information available about the tasks and the data, this allows Pydron to implement advanced scheduling strategies, such as the one described in Chapter 5, that go beyond what could be achieved by the schedulers of a cluster of grid system. In particular, it allows the scheduler to favor small tasks, therefore ensuring that loops are unrolled as quickly as possible which in turn helps to keep all the workers occupied.

### 4.3.2 Trivial Strategy

The trivial scheduling strategy assigns tasks to workers arbitrarily. A more advanced strategy is introduced in Chapter 5.

All tasks that are ready for execution are placed in a queue. All currently idle workers are placed in a queue as well. The scheduling operates by receiving one entry from each queue to assign a task to a worker, until one or both queues are empty. The tasks are sent to their respective worker for execution.

This strategy stores all data products on the main worker. For each execution the inputs are transferred to the worker and once the execution has finished, the outputs are transferred back to the main worker. This requires a potentially large number of data transfers and it limits the total size of data that can be handled by the system by the memory available on the main worker.

This strategy is therefore mostly suited for smaller problems, or large problems where the main data transfers happen over an alternative medium, for example over a shared file system. This is a fairly common scenario in astronomy use-cases, where most applications are file-based. In such a situation there is no advantage from a scheduler that attempts to reduce data transfers, as only file names are transferred through Pydron. Since the main worker contains the scheduler and the traverser logic, some communication is required to keep the status in the main worker up-to-date. The network cost will be determined by latency and not throughput.

In such cases this strategy has certain advantages. Due to its simplicity, there are no costly calculations required to assign tasks to workers and since all data is available on the main worker, certain tasks can be executed there. During translation from Python code to the data-flow graph, tasks that are known to execute quickly are marked (see Section 3.14.4). These are run in a background thread on the main worker. Due to the
global interpreter lock this does not give us actual parallelization, but it allows these tasks
to run without blocking the graph traversal, refinement, and scheduling operations. Since
the data is available in the same Python interpreter, no serialization or data transfer is
required, reducing the overhead to state keeping.

4.3.3 Scheduler Relocation

The main worker can run on the workstation of the user, but it can also be relocated
to a remote Python process managed by the backend. If the latency between the user’s
workstation and the remote machines is substantial (such as when executing on a cloud),
this will greatly reduce the communication overhead. The data transferred from and to
the workstation is reduced to the arguments passed to the \(\texttt{@schedule}\) function, the return
value, and the data-flow graph of the function.

Figure 3.4 shows the complexity that a simple loop produces internally. Each expression
results in a task and each task requires at least two network round-trips. If the scheduler
has to communicate with the workers via an internet connection, the latency can add up
to more than a second per iteration. This is can dominate the execution time if the tasks
within the loop execute faster than that. With the trivial scheduling strategy described in
the previous section there is the additional issue that all data products are transferred back
to the main worker running the scheduler at the end of each task execution. Therefore not
just latency but also bandwidth can severely limit the scalability. The scheduler introduced
in the following chapter avoids this issue by transferring data products directly from worker
to worker. Still, with both schedulers it is advantageous to relocate the scheduler closer
to the workers.

4.3.4 Fault Tolerance

With increased number of machines, the probability of a single machines failing is in-
creased. If a machine fails, the scheduler is informed that the workers from that node
are no longer available. The scheduler takes the worker from the list of available workers.
Since a data products are only deleted once all tasks using them have completed, we can
re-run any task that was executing at the time of the failure on a different worker.
Chapter 4. Runtime system

Figure 4.3: Layers of the backend system

4.4 Backend

The backend is responsible for acquiring computing resources from the infrastructure. It serves as an abstraction layer from the infrastructure. This allows Pydron to run in a wide range of different environments. This in turn, makes the user’s code portable which is especially important if the development teams and data centers are not centralized, as it is often the case in large astronomy projects.

The backend consists of several layers. Each layer is modular and can be replaced if needed. Figure 4.3 shows the components of the backend. Most infrastructures it will only require a dedicated implementation of the lowest layer. We implemented three such implementations to support a wide range of infrastructures.

4.4.1 API towards the scheduler

The backend provides resources to the scheduler. There is one worker provided to the scheduler per CPU core available for computation. The scheduler performs a number of operations on the workers:

- Transfers of data products from one worker to another.
- Delete cached data products.
- Inquire information about a cached data product required for refinement.
4.4. Backend

- Send tasks to the worker for execution.
- Free the resources allocated by the worker.

The API is kept to a minimum to decouple the infrastructure from the remaining Pydron system. Even for advanced scheduling algorithms, such as the one described in Chapter 5, this API can be kept small due to Pydron’s approach of using runtime information. Therefore no API is required to describe infrastructure details, such as the network topology or performance information for the machines.

4.4.2 Code Deployment

To execute tasks on remote nodes, the application’s code has to be available on the nodes. Manual code deployment is tedious. One option is a shared file system to which both the developer’s workstation and all the machines available for computing have access. This is often not the case, however. Especially if the developer teams and the infrastructures are not centralized. If the code is changing infrequently, the overhead introduced by manual installation, say through a package management system, may be neglectable, but if the developer is working interactively with the code, changes are not the exception but the norm and code deployment becomes a significant part of the development cycle.

Pydron combines two methods to automate this process. The goal is to make it entirely transparent to the user. Running the code on a parallel infrastructure should ideally be no different than running the same code on the local machine.

Pydron achieves this goal by using a Python import hook [86] on the worker nodes. When a Python module is imported which is not available on the worker node, the import hook loads the source code from the user’s workstation over the established communication channel. Python’s internal caching of loaded modules ensures that this has to be done only once per module and Python process. This ensures that all workers are using the up-to-date version of the developer’s code. There is no manual intervention required by the user, such as specifying source directories or package dependencies, as this method will automatically identify the required Python modules. This method will also transfer third party libraries from the workstation to the workers if they are not installed there already. The only constraints are libraries that contain native code, as these are platform specific and may not work on the compute nodes. In the future, Python wheels [87] and PyPi [88] could be employed to automatically install third party dependencies in a virtualenv [89].
One specific scenario where native code is involved is when source-to-source translators such as Numba [57] are used to speed up innermost loops. These libraries compile Python code into native code. In our advantage, these systems are compiling the code just-in-time [90]. Pydron will transfer the Python module containing the Python code to the worker. Upon first invocation this code will be translated to native code. This happens on the worker, therefore the correct platform is targeted. As a result, Pydron can be combined well with such systems. We have used this in the use-case shown in Section 5.3.1, where Pydron handles the parallelization of the outer-most loops and Numba ensures efficient execution of the innermost loops. Numba itself, however, cannot be transferred automatically through the import hook, as it includes the binaries for the LLVM compiler [91].

4.4.3 Pydron Deployment

One of the drawbacks of systems such as Hadoop [34] is that they require significant effort to set up on a given infrastructure. In an agile development scenario, where the life-time of the parallelized code is short, such an initial effort can be a major hindrance. Pydron is designed to be quickly deployable, so that it may be used even for a short prototyping effort.

Part of the deployment process is to get the code of Pydron itself installed on the infrastructure. Before a Python worker process is started, Pydron creates a zip file containing Pydron itself and its key dependencies. This file is transferred to the target machine. Pydron also creates a small bootstrap script which is passed to the Python interpreter as an argument. The bootstrap code then adds this file to the search path. This ensures that all workers receive Pydron from the machine that initiates the process, for example the developer’s workstation. The compute nodes do not need to have Pydron installed at all. All that is required is a method to start Python interpreters, which is discussed in Section 4.4.7 and a few prerequisite Python libraries. This method avoids potential issues with version conflicts as all workers use the same Pydron version. As a welcome side-effect, it is also a convenience when developing and testing Pydron itself.

If the developer so wishes, it is possible to specify additional packages that will be transferred by means of the zip file. This can be handy to avoid the overhead of the import hook based deployment at the cost of explicit declaration. This can be beneficial if the execution runs are particularly short. For most use-cases, the overhead of individually transferring Python modules on demand is negligible.
4.4.4 Remote Procedure Calls

The communication layer consists of a remote procedure call system that forwards the scheduler’s requests of the worker API (Section 4.4.1) to the worker process. It operates on top of the message passing API described in Section 4.4.1.

Remote Procedure Calls (RPC) [92] is a well-established concept, however for the purpose of Pydron, there were several challenges that had to be addressed.

Most RPC libraries are blocking. It lies in the nature of parallel computing that multiple calls will occur at a given time. With a blocking API, it is necessary to use threads. However, with potentially thousands of workers, contention of the global interpreter lock becomes severe to the point where connections time out and watchdog timers (see Section 4.4.6) trigger the termination of the workers as they seemingly lost contact to the main worker. To avoid these issues, the RPC system is asynchronous, using the twisted library [93].

The RPC system has to serialize and deserialize the parameters and return values sent of the message passing system. While this works well for small objects, Pydron is designed to handle significant data volumes. Therefore, the objects can be large. The copy operations take time, but the larger issue comes from memory usage. For example, if a ten GB NumPy [26] array is passed to a task for processing, it has to be serialized first. The serialization library of Python [94] will ask NumPy to provide a byte-string representation of this object, which will consume another ten GB of memory. The byte-strings of all serialized objects are then concatenated into the final serialized form, creating another copy of the data. The RPC system is then adding its own headers, creating yet another copy. These copy operations consume memory as there will be at least three copies of the object in memory at the same time. This considerably reduces the amount of data that can be processed on a worker.

Our RPC system avoids these issues by streaming the data to the message passing system. The sent messages are not assembled in memory before being passed on. Instead, our RPC system streams the data to message passing system. If the output buffers are full, since the network cannot keep up with the serialization rate, the serialization is paused. Similarly, data is received and deserialized immediately to avoid storing the entire object in its serialized form. Python objects consists of references to other objects, which eventually point to native types, such as primitives. This allows us to stream the serialized data in small units. The exception are NumPy arrays, which use a continuous memory range for
storage. Using the regular serialization API, these objects become a single byte string. This implies that at least one copy will be in memory, which can be an issue as these arrays can be arbitrarily large. However, NumPy provides alternative ways to access the arrays content and the copy can be avoided by treating NumPy arrays specially. Treating NumPy array efficiently is particularly important, as they are the developer’s tool of choice to store larger amounts of data in a compact form, as regular Python data structures have a significant memory overhead.

Before this streaming feature was added, Pydron was not able to scale the use-case later shown in Section 5.3.1 beyond about 250 cores, as it would either run out of memory or the main worker would slow down to the point where the worker’s watchdog timers (see Section 4.4.6) were no longer reset in time.

### 4.4.5 Message Passing

Pydron uses a message passing API to hide the underlying communication methods. We choose message passing as it lends itself well to implement a RPC system on top of it. It can also be efficiently implemented on most communication infrastructures.

For our experiments we have implemented message passing on top of TCP. Depending on the infrastructure, the API could also be implemented on top of MPI [44], or using shared memory to improve communication between workers on the same machine.

Like the RPC system, the message passing system is asynchronous to avoid lock contention. Twisted [93] is used communicate over TCP. To avoid the memory issues addressed in Section 4.4.4, the message passing system avoids accumulating entire messages and instead streams data directly to and from the underlying sockets. Both producers and consumers can be paused to avoid buffers growing too large.

While some messages will be large and mostly bound by throughput, others are latency critical. For example, information about data products has to be collected for graph refinement operations. These are often very small messages. For example, the information required to refine an if-task is a single boolean. If refinement is progressing slowly, it may cause a situation in which loops are unrolled slower than tasks are executed. To avoid the graph refinement to become a bottleneck, small messages must be able to pass quickly, even in the presence of large messages. Our implementation handles this by opening additional TCP connections on demand so that there is one connection available at all
times to send small messages and no delay occurs due to messages having to wait until larger messages release the connection. Since the scheduler does not schedule several large tasks to the same worker simultaneously, the number of large messages in transit from or to a particular worker is limited. In practice two connections are usually sufficient.

Firewalls and network address translation can interfere with the communication. Pydron attempts to open connections in both directions, which can help to avoid some of these issues. More advanced methods, such as hole punching [95] could be applied in the future.

4.4.6 Watchdog timer

Whenever starting processes automatically, there is a risk that these processes are not terminated automatically and remain running until manually terminated. Since these workers can allocate costly infrastructure, and since manual termination is a significant effort with thousands of processes, Pydron includes save guards to reduce the risk of processes surviving the main worker.

Each worker runs a timer in a background thread. This timer is reset from the main worker every thirty seconds. If the timer is not reset within one minute, a time out occurs and the worker assumes it has lost contact to the main worker. The worker then proceeds to free any resources it has allocated. This depends on actual infrastructure it is running on. The timer reset is made via the RPC system. This ensures that any failures of the RPC system, or the underlying systems, cause a timeout. This may happen, for example, if the thread that runs the event loop encounters a dead-lock.

The main worker resets the timers on all workers. The workers have to acknowledge that the timer was reset. If this acknowledgment is not received within one minute, the main worker assumes that communication has failed and tries to free the resources allocated by that worker.

There is no guarantee that these watchdogs will succeed in all cases. For example, if a bug within the Python interpreter, or a library containing native code, does not release the global interpreter lock, the entire interpreter will stall, and the watchdog cannot perform the clean-up routine.
import sys
import os.path
import traceback
import logging
import signal
import site

if hasattr(signal, "SIGHUP"):
    signal.signal(signal.SIGHUP, lambda *args: os._exit(1))
logging.basicConfig(level=logging.WARN, stream=sys.stdout)
codepath = os.path.abspath({zip_filename})
sys.path.insert(0, codepath)
reload(site)
import {kill_module}

try:
    import {this_module}
    from twisted.internet import reactor
    {this_module}._boot(
        {myaddr},
        {init_encoded},
        {kill_module}.{kill_name},
        {comm_setup_timeout},
        {watchdog_timeout})
    reactor.run()
except:
    traceback.print_exc()
    {kill_module}.{kill_name}()

Figure 4.4: Boot-strap script template

4.4.7 Starting Python Interpreters

For the reasons elaborated in Section 4.3.1, we start one Python interpreter per core. There are different starter implementations for different infrastructures, but the general process is the same for all of them.
A zip file containing Pydron itself is placed in a temporary directory (see Section 4.4.3). Then the Python interpreter is invoked and a boot-strap script is passed to it. The script is generated from a template shown in Figure 4.4. It performs the first steps of the process to create a new worker:

- Registers SIGHUP event handler that exits the interpreter. This is a safety mechanism that helps to ensure that the Process will terminate if something out of the ordinary happens.
- If possible, the standard output of the interpreter is intercepted and forwarded to the user. We also configure the logging framework as early as possible to help with debugging.
- The zip-file containing the Pydron library, which was placed in a temporary directory, is added to the search path.
- A starter specific module is loaded and the control is handed to the starter specific boot process.
- The twisted reactor loop is initiated.
- In case of an error that went unhandled, the boot script attempts to report the error and uses a starter specific method to clear up any resources associated with this worker.

4.4.8 Local multi-core starter

Since Python has a global interpreter lock, multi-threading cannot be used effectively for parallel computing. Instead, Pydron starts one Python interpreter per core on the local machine. For smaller problem sizes, utilizing the cores of the local machines might be sufficient. This is by far the simplest starter implementation and is well suited for testing and evaluating Pydron as no high performance infrastructure is required.

While communication could be done using shared memory, the current prototype of Pydron is using TCP connections to localhost as a basis for the message passing system.

The main worker can forcefully terminate the processes if communication has been lost.

The user has the option to configure how many processes are to be started. By default Pydron starts one process per core.
4.4.9 SSH starter

A flexible method to utilize the processing power of remote machines is through SSH. This starter opens an SSH channel and uses it to transfer the zip file (via SFTP) and to execute the Python interpreter.

Note that the communication between the workers is using TCP through the message passing layer and is not passed over the SSH channel. Since the SSH protocol has support to route network traffic through SSH, future versions of Pydron may change this which would have the added benefit of encryption. It would also avoid issues with firewalls, as the communication would pass over a channel that is explicitly allowed by the firewall rules.

If the message passing communication fails for some reason, the SSH protocol offers the option to forcefully terminate a remote process. Unfortunately, some SSH servers, such as OpenSSH, choose not to implement this feature [96]. Instead, Pydron closes the SSH connection in such a situation. This causes the Process to receive a SIGHUP. The boot script registers a handler for this signal that terminates the process.

The remote machines must have a Python 2.7 interpreter installed, as well as number of required Python packages that Pydron does not automatically distribute as part of its deployment process since they contain native code. There are no further constraints.

The user has to configure the host names and credentials as well as the number of processes to start on each machine.

4.4.10 Cloud starter

Pydron is using Apache libcloud [97] which provides a unified API to access a large number of cloud service providers.

The cloud instances require a disk image with the operating system to boot from. As described in Section 4.4.9, this image must contain a Python interpreter and must have the required libraries installed. There are images readily available that have been designed for scientific computing in the cloud, such as StarCluster [98], that contain Python and a large selection of pre-installed packages.

To use the cloud starter, the user configures the starter with image, instance type and credentials. The starter will then use libcloud to create and boot cloud instances. Once
the machines are running, and their IPs have been received, the method described in Section 4.4.9 is used to open SSH connections to the machines and start a Python interpreter per core.

The entire process typically takes less than five minutes. When working on large jobs, this delay will have little impact on the total execution time. For for interactive work, instances can be started and stopped explicitly by the user. This way, the instances can be started once, and then used for several execution runs before being shut down again.

Should the process on the developer’s machine lose connection with a machine, it will use the cloud API to forcefully terminate the instance. The Python processes on the cloud in turn notice if communication with the developer’s machine has failed, and will shut down the machines they are running on (see Section 4.4.6). Despite these efforts, it is the user’s responsibility to check that all machines have been terminated once the processing has finished.

### 4.4.11 Cluster starter

Many institutions provide high performance computing clusters for their members. How these clusters are organized, and which software product they use manage the queue, differs. The cluster starter is a universal starter that executes a bash script provided by the user in the configuration file. The bash script is provided with arguments that have to be passed to the Python interpreter, in particular the boot-script. It is then up to the user to invoke the necessary commands to request resources from the queuing system and ensure that the Python processes are started. SSH can be used to execute the script remotely on a login node.

Starters could be written to target specific cluster management systems, which would simplify the setup for the user.

### 4.4.12 Combining Starters

Starters can be combined in different ways. For example, the SSH, Cloud, and Cluster starters are combined with the multi-core starter to start one worker per core on each remote machine. It is also possible to use multiple starters to combine resources from different infrastructures.
Chapter 4. Runtime system

@schedule
def train_forest(data, labels, count):
    forest = []
    for i in range(count):
        tree = train_tree(i, data, labels);
        forest += [tree]

def predict(sample):
    predictions = [tree.predict(sample)[0] for tree in forest]
    return Counter(predictions).most_common(1)
return predict

Figure 4.5: Random Forest Implementation

4.5 Scalability

In this section, we demonstrate the scalability of Pydron for multi-core, Cluster and Cloud infrastructures. We also provide insights through several experiments on how Pydron operates. All measurements were taken with CPython 2.7.6. We use the scheduling strategy discussed in Section 4.3.2 to demonstrate Pydron’s ability to extract parallelism from regular Python code.

4.5.1 Multi-core

We use a machine learning example for the multi-core and cluster measurements. The Random Forest method [99] trains several decision trees on a random sub-set of the training samples. Predictions are made by majority vote among the predictions made by the individual decision trees. We used 50% of the samples in the MNIST hand-written digits dataset for training [100] (approx. 27 MB).

The code is shown in Figure 4.5. The train_forest function is annotated with @schedule. The for-loop can be unrolled completely in the beginning of the execution since train_tree is annotated with @functional. train_forest returns a nested function to make predictions, using a closure variable to access the forest. If predictions were expensive, then annotating the nested function with @schedule would parallelize the list-comprehension as well.
4.5. Scalability

Figure 4.6: Random Forest Training on Multi-core

The implementation of \texttt{train\_tree} is using scikit-learn from SciPy [26] internally. Pydron handles calls to third-party libraries as any other function call. Since SciPy uses native code, the library cannot be automatically sent to the worker, but has to be installed on all participating machines (see Section 4.4.2).

Figure 4.6 shows the learning time on a single machine with 64 cores (AMD Opteron 6276) when running the code using regular Python and when using Pydron with an increasing number of cores. It scales nearly linearly and only flattens once the communication overhead becomes noticeable. This is to be expected as Pydron currently makes no use of shared memory for communication. The horizontal line marks the execution time with regular, single threaded Python, which is about 90s (~1%) faster than Pydron with a single core.

4.5.2 Cluster

Figure 4.7 shows the result of the same machine learning code when Pydron is instructed to execute it on a cluster (Intel Xeon L5520). We use the combination of the cluster
and the multi-core backend to utilize the 8 cores of each node. We run the experiment with up to 16 cluster nodes with a total of 128 cores, at which point the scalability starts to degrade due to communication overheads. The training data has to be distributed, and the resulting trees are transferred back to the main worker. The execution time of regular, single threaded, Python is not shown in the figure as it would be about eight times slower than a single node. When running Pydron with only one of the node’s cores, the difference is comparable to the one shown for the 64-core machine. The numbers of the two experiments differ slightly due to the different CPUs.

4.5.3 Cloud

Running the machine learning code on the cloud produces results comparable to those on the cluster, we therefore use cloud computing to demonstrate Pydron on an astronomy use-case.

PynPoint [11] is a method for detection of planets outside the solar system. The challenge of exo-planet detection lies in the extreme contrast between the bright host star and the
4.5. Scalability

```python
@schedule
def parameter_sweep(in_file):
    images = []
    basis = []
    for center in np.linspace(0.01, 0.1, 6):
        for edge in np.linspace(0.7, 1.0, 6):
            images += [create_images(in_file, center, edge)]
            basis += [create_basis(in_file, center, edge)]
```

**Figure 4.8: Parameter Sweep Code**

faint planet. Optical effects and atmospheric distortions spread the light of the star over an area larger than the orbit of the planet. PynPoint models the point-spread function of the star with a principal component analysis (PCA) to remove the spread-out light from the star, leaving the planet visible in the residue.

We use a high-contrast imaging dataset of $\beta$ Pictoris [101] and the massive exo-planet orbiting it. The dataset was taken with the Very Large Telescope. The raw data is publicly available from the European Southern Observatory (ESO) archive (Program ID: 084.C-0739(A)). Some data reduction steps [102] have already been applied to the data. The dataset consists of 24000 individual exposures, totaling to 3.8 GB.

PynPoint operates in two main phases. In the first phase, the images are prepared and the basis of the PCA are calculated. In the second phase, the modeled point-spread-function of the star is removed from the exposures. The exposures are then rotated to compensate for earth’s rotation and aggregated into the final result. The second phase is fast enough to be used interactively by the scientists to study the effect of the method’s parameters. However, some parameters affect the first phase which takes about 15 minutes to execute. We have used Pydron to scale the parameter sweep over the two main parameters used in the first phase.

Six values are used for each parameter, resulting in a total of 36 executions. The code is shown in Figure 4.8. `in_file` contains the path to the input data file in HDF5 format. The two functions `create_images` and `create_basis` are both decorated with `@functional`. Since they are independent of each other, all 72 calls can be run in parallel. The implementation of those methods use numerical routines from SciPy [26] which use multi-threading inter-
nally to utilize multiple cores. Pydron can be used together with such libraries. We use Pydron’s cloud backend to parallelize across multiple cloud instances, but to get a clearer performance analysis we configure Pydron to start only one Python process per node.

We use Amazon EC2 with *m2.large* instances, with two CPU cores each. Adding more cores per node would not scale well with this workload, as the routines only profit from the parallel SciPy library for a part of their execution. The cloud instances are connected to a shared file system used as a scratch space. This file system is provided by two separate EC2 instances (*c2.2xlarge*) which provide the storage from a total of four solid state drives. The file system is clustered with glusterFS [103]. The file system initially contains the input data.

Figure 4.9 shows the execution time for up to 32 instances (64 cores). The execution time includes the time required to start the instances, which takes about one minute.

Other than in the machine learning use-case, the actual data is transmitted over a shared file system, while Pydron only handles the paths, as described in Section 3.1.4. The Pydron induced overhead is therefore very small, about 7 seconds. With a large number of instances, the throughput of the scratch file system becomes a bottleneck, as each

**Figure 4.9: Scalability on Amazon EC2**
parameter combination produces approximately 4 GB of data. This bottleneck could be easily addressed by increasing the number of nodes of the clustered file system.

The overhead introduced by Pydron is neglectable in this use-case. The translation of the Python code into the initial data-flow takes five milliseconds. Figure 4.10 shows that less than a second is spent for all dynamic changes in the data-flow graph and that less than eight seconds are required for communication, including serialization. Both can partially run in parallel, reducing the impact. With 32 instances the workers are limited by the shared file system, the lower CPU utilization speeds up serialization.

4.6 Summary

The runtime system completes static code analysis of Chapter 3 with runtime information. The resulting dynamic data-flow graph is sufficiently expressive that the runtime system can execute it parallel, resulting in non-intrusive system that can parallelize sequential single-threaded Python code with minimal help from the developer.
Chapter 4. Runtime system

The results show that even with a simple scheduling strategy, the system is able to utilize multi-core, cluster, and cloud infrastructures with an API of only two decorators. While not every code is parallelizable, Pydron can deal with legacy Python code and third-party libraries, including such that use native code.

Since the focus of Pydron is on non-intrusiveness and a low learning curve, achieving the best performance on a given infrastructure is not the primary goal. We have found that, despite this focus on simplicity for the user, Pydron’s approach of using runtime information in addition to static analysis has significant potential to further improve the performance.

In the next chapter we describe such an extension that provides a location aware scheduler capable of reducing the required data transfers while increasing Pydron’s scalability to a thousand cores.
Location aware scheduling

The previous chapter used a simple scheduler to demonstrate the feasibility of parallelizing sequential Python code. In this chapter, we describe a more advanced scheduler. This scheduler takes the location of the data products into account to assign tasks to workers in a way that balances utilization of the available cores with the amount of data that has to be transported.

Parallel infrastructure is complex. A typical cluster may consist of nodes with multiple sockets, each with several cores that can run multiple hardware threads. The communication between two nodes depends on their location in the interconnect’s topology. On the other hand, the communication pattern between the tasks of the application can be complex as well.

Making the right scheduling decisions depends not only on the infrastructure but also requires application specific information. This makes it difficult to provide a generic scheduler in parallel systems that work for a large number of use-cases on a large variety of infrastructures.

Systems such as MPI [44] leave the problem to the developer. MPI provides fast communication, but makes no attempt to position tasks on the infrastructure in such a way as to reduce the communication cost. Systems that include scheduling, such as MapReduce [49], limit the communication patterns that the application is allowed to use. This makes the scheduling problem easier at the cost of flexibility. If such systems are too constraining for
an application, the developer has to implement scheduling, which is time consuming and leads to solutions tailored towards a particular application and infrastructure, reducing portability and reusability of the code.

This is particularly an issue for science consortia. For example, in astronomy where globally distributed research groups contribute to the algorithm and data processing. Each group may have access to high performance computing infrastructure, but the infrastructure used by the groups can differ significantly. Even in projects of a single group, the increasing interest in cloud computing makes portability important.

In the previous chapters we focused on the aspects of Pydron that allow Python code to be represented as a dynamic data-flow graph to allow semi-automatic parallelization. This chapter takes the idea of using runtime information a step further by collecting information about the application and the infrastructure to improve the scheduling decisions without requiring the user to specify any additional information about either the application or the infrastructure.

Such a scheduler does not just improve the performance by reducing the data transfers, it also provides decoupling from the application and the scheduler by adapting to both the application and the infrastructure automatically.

The scheduler, which is described in Section 4.3, does not require any extensions to Pydron’s API since it can collect the required information automatically (see Section 5.1).

We demonstrate the effectiveness of this scheduler by showing results of a real-world and of a synthetic use-case on up to 1008 CPU cores on Amazon EC2 in Section 5.3.

5.1 Metrics

We collect three metrics during the execution that provide additional information for the scheduler.

5.1.1 Execution time

The execution time of a job is important as it puts the cost to data transfer to a node into relation with the time gained from running that job on that node. It also gives us an estimate when a currently running job will be finished and the worker will be idle again.
5.1. Metrics

We measure the execution time of each completed job and store this information in the graph. This information is then used to provide an estimate for jobs that have not yet been executed or have not yet finished.

To estimate the execution time of a job we use a simple strategy which looks for a “similar” job in the graph that has already finished. Since parallelization usually happens over loops, we use for the job that corresponds to the same code statement, but from a previous loop iteration. This is possible using the tick property of task nodes described in Section 3.14.1. The tick tuple has elements that encode loop iteration counters while the others identify the statement within the code.

Initially there are no such jobs that have completed, in which case we set the estimate to $+\infty$. This causes the scheduler to prefer to parallelization of such tasks, as the estimated execution time outweighs the cost of data transfer.

More elaborate strategies would be possible. In data processing applications, the execution time is often related to the input data size. For jobs that are ready for execution, the input data sizes are known. Machine learning methods could be applied to model the relation between input data and execution time for each function. Developers may be willing to provide additional information, such as the complexity class, for their code, which could further improve the estimates at additional development cost.

Other scheduling systems, such as the ones discussed in Section 2.10, assume that the processing time per data item does not vary significantly. For example, in MapReduce a typical assumption is that the each map operation takes about the same time. In Pydron, this would roughly correspond to taking the average execution time of previously executed tasks of the same statement.

In an ideal scheduler, if small variations of an estimated execution time effect the resulting scheduling decision, both scheduling plans have a similar estimated total execution time. This suggests that small variations in the estimate have only a small effect on the efficiency of the system. Since Pydron uses a greedy scheduler, and therefore might not find the global minimum of the total execution time, this cannot be guaranteed. In our experiments we found that even with a simple estimator the scheduler can make good decisions.

Some tasks are known to execute quickly, for example extracting an element from a tuple. The translator marks such tasks as quick (see Section 3.14.4). If the mark is present, the execution time estimate is set to zero. While this is not true, the actual execution time is dwarfed by the overhead of managing the task.
Chapter 5. Location aware scheduling

5.1.2 Data volume

For the data volume, we don’t need a prediction since the scheduler is only considering jobs that are ready for execution. For those jobs, all input data products have been produced already.

Measuring the size of those products is non-trivial, however. We send data from node to node by using Python’s serialization system called pickle. Since the data can be any arbitrary Python object, the size is known only after serialization. This poses a problem as serialization can, depending on the data, be the bottleneck in the data transfer. If we require that every data product is serialized to measure its size, even before we make the decision if to transfer it or not, we lose a significant part of the advantage we want to gain by avoiding data transfers.

To address this issue we have developed a separate library picklesize which traverses an object structure and calculates the size of an object in its serialized form, without the actual data copying operations performed by serialization. Especially for applications that make heavy use of array data structures, for example using NumPy, our approach is orders of magnitudes faster and has minimal memory overhead.

For data structures consisting of many small Python objects, picklesize still has to traverse the entire object graph. For scientific applications this is rarely the case, since Python objects have a significant memory overhead and can only be accessed through the Python interpreter. Even Just-in-time compilers (see Section 2.6) typically don’t support arbitrary Python objects. Therefore, NumPy arrays are used almost exclusively to store the bulk of the data. This allows more compact storage in memory and makes it possible to run the innermost loops without the Python interpreter, either through NumPy operators, or using just-in-time compilation. Regular Python data structures are still used, for example to maintain meta-data. Picklesize can handle arbitrary Python objects. It can calculate the byte accurate serialized size for any object that can be serialized by Python. This is important, as we do not want to limit the data types the developer may use.

5.1.3 Network

To calculate the transfer cost of a data product, we need an estimation for the throughput between the worker nodes.
Due to Python’s global interpreter lock, we use a separate process for each core. Ideally shared memory would be used to avoid serialization and transfer of data products within a machine, but the memory model of Python does not allow sharing of arbitrary objects. Therefore, Pydron is using sockets for communication between all workers.

Ideally, we would estimate the throughput between each pair of workers, but this is impractical. There might be significantly more worker pairs than there are data transfers during the entire execution run. We apply a simpler model. Workers are assigned to groups. For each group there is an estimate for inter-group communication. In addition, we estimate the throughput between groups. For the experiments presented in this dissertation, we estimate only two throughputs: Between workers on the same machine, and between workers on different machines. This model could easily be extended to be rack-aware.

A more advanced model could be used to better represent the actual network topology which would also allow to model network congestion. We found this unnecessary for our use-cases.

Initially the estimator is set to a configurable default value. As soon as transfers occur, the estimator is updated using a least-square approximation for both throughput and latency. These estimations can be stored to initialize the estimator and in future runs.

5.2 Scheduler

Transferring the input data products to a worker node can take a significant amount of time. To reduce this effect, we’d ideally run each task on a worker that already contains those products, such as the worker on which the product was previously created. Since tasks can have several inputs, and since we’d like to distribute the tasks across many workers, we cannot always avoid data transfers, but careful assignments of task to worker can significantly reduce the data transfer over an arbitrary assignment.

We applied a greedy scheduling strategy. It only takes tasks into consideration if they are ready for execution, that is, if all input data products have been produced.

Using the metrics, the scheduler can estimate the time it would take to execute a particular task on a particular worker. This includes the execution time and the time for all required data transfers. This total duration is calculated for all possible task and worker combinations. Since the metrics are constantly refined, this process has to be repeated for each scheduling run.
Chapter 5. Location aware scheduling

Figure 5.1: Worker’s task queues. Tasks 1 and 2 are assigned, task 3 is not yet assigned.

![Worker’s task queues. Tasks 1 and 2 are assigned, task 3 is not yet assigned.](image)

Figure 5.2: Worker’s task queues. Tasks 1 to 3 are assigned, task 4 is not yet assigned.

![Worker’s task queues. Tasks 1 to 3 are assigned, task 4 is not yet assigned.](image)

If we’d just assign each task to the worker where the estimated execution time is minimal, we run into a risk of assigning many tasks to the same worker. If all the input data is initially located on a single worker, we’d effectively run the complete execution on this worker alone. To account for that, we take the time it takes for a worker to become idle into consideration too, and assign a task to the worker where the total time until that task has finished is minimal.

To the estimate of the execution time and data transfer, we add the time until the worker will be ready to start processing the task. For this we keep a queue for each worker of the tasks that have already been assigned to this worker. Initially, these queues contain the tasks the worker is currently processing. As we assign tasks to workers, these queues get filled. In Figure 5.1, Worker A and B are currently processing a task each. To decide where the third task should run, we check how long it would take for task 3 to complete
5.2. Scheduler

on each worker. In this example worker B requires the least data transfers. Even though
worker C is idle, it is beneficial to wait until worker B has finished task 2 to avoid the
data transfers.

Once the decision has been made to run task 3 on worker B, the task is added to that
queue.

Figure 5.2 shows the situation for the next task. Again, worker B would require the fewest
data transfers, but since that worker will be occupied with two tasks for some time, this
worker is no longer the best choice. Instead worker A is selected, even though this requires
some additional transfers.

The order in which tasks are scheduled can greatly impact the final result. Assuming we
have two tasks ready for execution, one is CPU bound and takes about the same time to
finish on all workers, while the other is IO bound and can run in seconds on one specific
worker but will take minutes on any other worker. If the CPU bound task gets scheduled
first, it will get assigned to the worker with the smallest queue. If that worker happens to
be the one which contains the data required by the IO bound task, this might force the
IO bound task to another worker.

To militate this effect, we schedule the tasks in the order of their expected time until
finished. The process is as follows:

1. For each non-scheduled and ready-to-run task and each worker, estimate the time,
based on the current queue contents, until the task would finish on that worker.

2. From all these, choose the task and worker combination that can finish soonest.

3. Add the task to the queue the worker. If the worker has been idle, send the task to
   the worker for execution.

4. Since one of the queues has changed, go back to step 1 to re-calculate the estimations.

A straight forward implementation of this procedure would be very expensive. Step 1 is
$O(nm)$, where $n$ is the number of tasks, and $m$ is the number of workers. This has to be
repeated for each scheduled task, resulting in $O(n^2m)$ for the scheduling run. It also has
to be restarted for each scheduling run since the metrics will have been refined. We have
found an algorithm to perform a scheduling run in $O(nm)$.
5.2.1 Transfer time calculation

The majority of tasks will have a small number of inputs, as they result from operands and function arguments. If a task has two inputs, there are only four possible combinations of inputs that are already present on a worker or not. Since the scheduling is actively avoiding transferring large data products, the common pattern that emerges is that inputs with large data volumes are stored on only one worker, while inputs for small objects are likely to have been transferred to a large number of workers, if not all. Therefore, the required data transfers will be the similar or equal for most workers, with a few that store the large input products. These few will have a significantly lower data transfer cost. For example, if a series of map operations are applied to a set of data items, there will be some initial data transfer required to distribute the items across the workers. Afterwards only the code of the map-functions is transferred, while the processed data remains on the worker.

We can use this observation to speed up the first step. Instead of calculating the estimation for the data transfer time directly, we pre-calculate the size of the required transfers. Since we do not have individual throughput and latency measurements for each pair of workers, we can sum the transfers up in groups. For our experiments this results in only four numbers per task.

- Number of transfers from the same machine.
- Total bytes transferred from the same machine.
- Number of transfers from other machines.
- Total bytes transferred from other machine.

To calculate the transfer time, we can multiply these numbers with the estimated latencies and throughputs. By keeping a $n \times 4$ matrix of all tasks to be scheduled, we can calculate the data transfer time of all tasks with a single matrix-vector product each time the network estimations change. This can be done efficiently with NumPy in $O(n)$.

However, the matrix itself is not static. As data products are transferred, they become available on other workers. Therefore data transfer from another machine might later be possible locally or not be required at all. We incrementally update the matrix accordingly. Since most data products either stay on a single worker or are present on all workers, such
5.2. Scheduler

updates become rarer once the execution has warmed up. In our experiments we found that the cost of these updates is neglectable.

Tasks that become ready for execution are added to the matrix and tasks that have been sent to workers for execution are removed. We keep a list of empty slots in the matrix to avoid resizing the matrix if possible.

Since the matrix is build and maintained incrementally, it does not need to be created per scheduling run. Ignoring the cost of updating the matrix due to data transfers triggered by other tasks, the total cost is $O(nm)$ where $n$ is the total number of tasks that will pass through the scheduler. Since this effort is spread over the entire duration of the execution run, which may take hours, it is not a limiting factor.

5.2.2 Task-worker assignment

With the data transfer time estimated, we can get the estimation for task duration by adding the estimated execution time, but to get the total time until completion, we also have to take the current content of the worker’s queues into account. The time until a worker is able to start processing a new task, which we refer to as the queue-time, is given by the duration of all tasks already assigned to that worker plus the remaining time for the task the worker is currently processing. The latter is calculated from the task’s estimated duration and the time that has passed since it was sent to the worker for execution.

For each worker we maintain a min-heap with contains all tasks that have not yet been assigned to a worker. Each min-heap is filled with the same tasks, but the duration differ from worker to worker. These heaps are sorted by task duration. This gives us quick access to the task with the shortest duration on a given worker.

An additional min-heap is filled with one entry per worker, referred to as the worker heap, sorted by the sum of the worker’s queue-time and the duration of the first task in that worker’s min-heap. The minimum entry of the worker heap therefore gives us the worker and task combination that can finish soonest of all possible combinations.

The advantage of these data structures is that they can be updated incrementally. Once the heaps are built, the process is as follows:
1. Extract the minimum worker from the worker heap and extract the minimum task from this worker’s task heap.

2. Increase the queue-time of the worker by the task’s duration. If the worker is idle, send the task to the worker for execution.

3. Peek at the next task in the worker’s task heap to get its duration.

4. If there are more tasks in the worker’s task heap, insert the worker into the worker heap again. Since the queue-time and duration of the first task in the worker’s heap have changed, the position in the worker heap may change too.

5. Repeat until all workers are busy or the worker heap is empty.

Since each task is initially added to each worker’s task queue, this process could assign the same task to several workers. We could solve this issue by removing the task from all other heaps once an assignment has been made, but we found it simpler to check against a hash set of all tasks that have been assigned already. For tasks that we have processed already, the queues are updated as above with the difference that the task is not sent to the worker and the queue-time is not increased.

The cost for building task heaps with \( n \) tasks for \( m \) workers is \( O(nm) \) as heaps can be built in linear time. The heaps will not be the same due to the differences in data transfer time.

The assignment process removes one task from one task heap in each iteration, we therefore have an upper bound of \( nm \) iterations. Each iteration requires a task heap extract and a worker heap extract operation, therefore worst-case is \( O(nm(\log n + \log m)) \).

In practice, in particular if \( n \gg m \), we can abort early as all \( m \) workers will be busy at some point. A more realistic approximation that takes this into consideration can be given by \( O(cm(\log n + \log m)) \), where \( c \) is a use-case specific factor, that can be as low as one in ideal cases. The dominating cost of the entire process becomes building the heaps with \( O(nm) \).

Besides the better asymptotic behavior compared to \( O(n^2m) \), this implementation can use native code for all the innermost loops, such as the matrix multiplication and the heap operations, avoiding the overhead of the Python interpreter.
5.2.3 Fast by-pass

While the goal is to parallelize the coarse granular tasks, typically there are many smaller tasks required to prepare inputs and collect the outputs of the main tasks. Such small tasks may do nothing more than to check if an iterator has reached the end, or concatenate strings to form a file path. It is in the interest of the system to execute such task as soon as possible as each task that we’ve executed gives us more information. In particular, such tasks are required to unroll loops. Since these tasks can be numerous, they also slow down the scheduling process described above.

For those reasons, we identify tasks that can execute quickly and without significant data transfer and allow them to bypass the scheduling algorithm described earlier. There are two configurable thresholds, one for the amount of data to be transferred and one for the evaluation time. For the experiments the first was set to 32 KB. For such small data products the latency and the overhead of Pydron dominate and the actual transfer time is negligible. For the evaluation time the threshold was set to zero. This limits the by-pass to tasks that are marked as quick by the translator (see Section 3.14.4). They are guaranteed to execute fast, eliminating the risk of expensive tasks taking the by-pass.

If a task fulfills the criteria for the by-pass on multiple workers, the master worker is preferred. Since this is the worker running the scheduler, no network communication might be required at all.

A task that takes the by-pass is executed immediately, even if the worker is currently processing another task. The task is run in a separate thread as not to interrupt the other task. The multi-threaded execution does not lead to a better utilization of the hardware directly, due to Python’s global interpreter lock, but it does allow us to process these tasks without having to wait for a potentially long-running task.

Besides reducing the load of the main scheduling algorithm, the by-pass reduces data-transfers. Take for example a task that reads the number of elements a list object has. This task can be executed quickly on the worker that stores the list, but will require significant data transfer on any other worker. Without the by-pass, the scheduler can only choose between delaying the task until the worker is idle again, which may delay loop unrolling, or to make an expensive transfer for a trivial operation. The fuller the worker’s queue, the more attractive the transfer becomes for the scheduler, especially if there are no other tasks to occupy other workers. The by-pass offers a better solution that allows such small tasks to execute close to the data is without delay.
5.2.4 Special Cases

Since Pydron supports arbitrary Python code, this imposes some additional challenges. Some tasks are marked as *sync-points* by the translator as they may have untracked side-effects (see Section 3.14.2). Graph refinement attempts to remove these marks whenever possible (see Section 4.2.2), but the developer is free to use Pydron with code where this is not always possible. The scheduler forces such tasks to be executed on the master worker only. This ensures that any untracked data-flow remains consistent. For example, one such task might write to a global variable that another one reads. The graph traverser ensures that these tasks execute in the right order (see Section 4.1.1). By running them within the same Python interpreter, we ensure that the change to the variable will be visible.

Another issue are data products that cannot be serialized and therefore cannot be transferred to another worker. Since the *@functional* contract forbids this explicitly for both arguments and return value, such products can only be produced in two ways:

- Calls to functions that do not follow the *@functional* contract.
- Special tasks created by the translator for iterator handing and marked with no-transfer (see Section 3.14.3).

In the first case, the resulting task will have a *sync-point* flag, assuming that the developer did not wrongly declare the function as *@functional*. According to the rule above, all such tasks will run on the master node and therefore no transfer is required.

This rule does not work in the second case, as the nontransferable product can be located on an arbitrary worker. This is necessary since iterator objects need to be in the same Python interpreter as the underlying collection object. Since these can be large, we want to avoid transferring them to the master worker. All these tasks have been designed to have only a single *no-transfer* input to avoid the unresolvable situation where two unserializable inputs are located on different workers.

Despite the transfer limitations, these tasks are still subject to the scheduling process, unless they are eligible for the fast by-pass. The scheduler is informed of the constraint, and when the initial task heaps are built, these tasks are only inserted in the heap of the worker on which they are allowed to run.
5.3 Results

Since the master worker is responsible for maintaining the data-flow graph and the scheduling, it is usually beneficial not to schedule tasks to it. This gives the master worker more CPU time which allows it to dispatch tasks faster, decreasing idle time of the other workers. However, due to the special cases described above, this is not always possible. The scheduler can, however, be instructed to limit the tasks run on the master worker to these special cases. Regular tasks will then not be inserted into the task heap of the master.

5.3 Results

We show here that our strategy is successful even if simple. Further optimizations and using more complex strategies and analysis is part of future work.

The experiments shown in this section have been supported by an “AstroCompute in the Cloud” grant by the Square Kilometre Array and Amazon Web Services.

5.3.1 Scalability

To test the performance with a real-world use-case, we applied a compressed sensing algorithm [104] to RHESSI data. The RHESSI spacecraft observes the sun in the x-ray spectrum. Since x-rays cannot easily be focused by lenses or mirrors, the RHESSI instrument does not record images directly. Instead it uses rotating collimators, which effectively results in the recording of fourier components of the image. These images have to be reconstructed from the fourier components captured by the instrument. The instrument does not measure all fourier components, which makes this reconstruction challenging. Here we have applied a compressed sensing method to several thousand solar flares. For each flare several images are reconstructed to show the development of the flare over time and in different energy bands.

It takes about nine minutes on average to process a flare, depending on the strength and duration of the event. For our experiments, we processed about 10'000 flares, which is a small subset of the data RHESSI has collected so far.

We’ve run our experiments on Amazon EC2 with up to 1008 cores of the c4-type (which are optimized for computation).

Figure 5.3 shows the utilization over time to reconstruct the images of 10455 solar flares on 28 machines with 36 cores each. There is a short warm-up phase initially where the
loop is unrolled and the first tasks are distributed to the workers. There is significant variation in the execution time of the individual tasks. Small flares may finish in less than a minute, while large flares can take over 20 minutes to compute. This causes the roll-off effect at the end where no longer all workers can be kept occupied. We can also notice a slight increase in the utilization just before the roll-off begins. As the number of tasks in the queue is reducing, the scheduling algorithm becomes faster and workers get a new job assigned sooner.

We have run with a smaller number of flares varying the number of cores to demonstrate scalability. Figure 5.4 shows the duration to process 2613 flares on one to 28 machines with 36 cores each. The performance flattens out slightly since the search space for the scheduler increases with the number of workers.

Figure 5.5 shows a scale-out experiment where we varied the number of flares and number of cores together at a ratio of 10 jobs per core. Increasing the number of cores increases the overhead slightly. For the runs with very few cores the results are dominated by the variance of the randomly selected flares. This demonstrates how Pydron can be used efficiently from small problem sizes that fit on a single workstation to large data volumes.
5.3. Results

Figure 5.4: Scale-up. Processing 2613 flares on one to 28 machines.

processed on 1008 cores without any increased code complexity.

5.3.2 Locality

To test the decisions made by the scheduler we use a synthetic use-case with a well defined data transfer pattern for which we can manually find the optimal strategy. This gives us a base-line to compare our algorithm to.

Figure 5.6 shows the data-flow graph. The problem consists of a loop. Within each iteration, one task produces eight data products which are processed by eight tasks. A final task collects those results. If we were to process 5 loop iterations, a total of 50 tasks would be executed. Besides the edges shown in the figure, there are other data transfers occurring, for example, the function object is an input too. These are neglectable in size and are not shown. The scheduler is, of course, aware of them and does take them into consideration.

For the experiment we used 250 quad-core machines for a total of 1000 cores. We varied the number of loop iterations.
5.3.2.1 Ideal Scheduling

If we set the number of loop iterations to 1000, the ideal scheduling is trivial: Assign each of the 1000 loop iterations to one of the cores. The data products therefore stay within a core and the transfer is done by passing the object reference. Figure 5.7 shows this pattern (core to core transfers are shown in green).

This is ideal, but only as long as we have at least 1000 loop iterations. If we have fewer, than this distribution pattern would not allow us to keep all cores busy.

If we have less than 1000 iterations we can apply the pattern shown in Figure 5.8. Here each
5.3. Results

Figure 5.7: Scheduling pattern with number of loop iterations greater or equal to number of cores. No actual data transfers are required as the data stays within the same core.

Figure 5.8: Scheduling pattern with number of loop iterations greater or equal to number of machines. Data transfers are confined to cores of the same machine.

loop iteration is assigned to a single machine. Transfers happen only between cores of the machine. Ideally, such transfers would use shared memory to avoid any copy operation, but Python does not allow this. Therefore, these transfers have some cost due to the required serialization and deserialization. Since the fork and join tasks run on these cores
Figure 5.9: Scheduling pattern with fewer loop iterations than number of machines. Each iteration is split among two machines, requiring machine to machine transfers.

too, four of the transfers are still within the same core.

This pattern avoids any network transfers, but can only utilize all machines if there is at least one loop iteration per machine. If we have fewer iterations, transfers over the network become unavoidable. Figure 5.9 shows the pattern that minimizes the number of transfers between machines. Since we have eight tasks in parallel for each iteration and since we are using quad-core machines, we expect that each iteration will be split onto two machines. This requires a minimum of eight data transfers over the network (shown in red).

If we were to reduce the number of iterations further, the granularity of the problem would be the limiting factor, as each iteration has eight tasks that can run in parallel, we need at least 125 iterations to keep 1000 cores busy.

5.3.2.2 Measurements

We have run the above scenario with different numbers of loop iterations and observed the data transfers that resulted from our scheduler’s decisions.

Figure 5.10 shows the results for 125 up to 1000 loop iterations. Since the data-flow graph is given, we know that a total of eight transfers are required per loop. The figure shows
5.4 Limits of Scalability

The scheduling algorithm is able to find the correct distribution pattern reliably without any user intervention. At 1000 iterations, each iteration is assigned to a core, at 250 iterations each iteration is assigned to a machine, and at 125 iterations two machines share the work of each iteration.

For 500 iterations, the scheduler uses a combination of the two patterns shown in Figure 5.7 and 5.8. This would be rather challenging to do manually.

**Figure 5.10:** *Number and type of transfers for different number of loop iterations.*

With this improved scheduler, Pydron can scale up to about 1’000 cores as we have shown with the results in the previous section.

However, the constraining factor is not so much the number of CPU cores, but the number of tasks that the main worker has to schedule and dispatch each second. This depends on the granularity of the process as well as the number of cores.
Chapter 5. Location aware scheduling

The first experiment shown in Section 5.3.1 processed about 10'000 jobs in 120 minutes. In that experiment a total of 62'740 tasks were executed, or about 8.7 tasks per second. The number of tasks is higher than the number of jobs, since there are also tasks to handle the loop iteration, prepare arguments for the call to the function that performs the processing, and to collect the results. For simple loops the number of tasks will be between five to ten times the number of loop iterations. Each of these tasks is individually maintained in the data-flow graph, scheduled, and sent to a worker for execution.

The current Pydron implementation can handle about 100 tasks per second, but it is important to consider that the actual rate is not constant. At the start, all 1'008 cores need to get a task assigned. If iterations take an approximately equal amount of CPU time, the workers will also finish about at the same time, causing repeated periods where many tasks need to be dispatched quickly. To avoid lengthy periods in which many workers are idle, the peak rate of task evaluations the system can handle should be significantly higher than the average rate.

Ten tasks per second is a reasonable average with the current implementation, this gives us a processing rate of about one to two loop iterations each second. In the experiment, the 10'000 iterations complete in 120 minutes, which corresponds to a rate of about 1.3 iterations per second.

This observation allows us to formulate a relationship between the number of CPU cores and the minimal required granularity. With 1’000 cores, each iteration should require at least 500-1’000 seconds of CPU time, about 8 to 16 minutes. In this particular experiment the average was about 20 minutes. If we were to scale up to 10’000 cores, ignoring overheads proportional to the number of cores that would eventually become the dominant factor, the problem would require a significantly coarser granularity of 80-120 minutes per iteration.

We can also use this relationship to scale down. With 100 cores, 50-100 seconds per iteration is still sufficiently coarse granular, with 10 cores 5-10 seconds suffice. Since there is a component to the overhead that is proportional to the number of cores, the actual granularity requirements are somewhat lower. Runs with a small number of cores typically run on a single multi-core machine which lowers the overhead further. In practice, jobs of about a second work well in a multi-core setup.

So some degree, these granularity constraints are due to the implementation of the runtime system with single-threaded Python code. A more optimized implementation in a different language should be able process about an order of magnitude more tasks per second.
Beyond that, the runtime system could be parallelized itself by splitting the graph between several scheduler nodes, each feeding a set of cores. Work stealing methods [105] could be used to balance work between the scheduler nodes.

The reason for the granularity constraint is that each task execution is handled individually. With increasingly fine granularity and increasing number of cores, this becomes the limiting factor. If we want to achieve a truly interactive system, where each execution run takes a few seconds at most, we need a system that can handle granularity in the milliseconds range. This will require more fundamental changes than improving the implementation of the runtime system. Sections 6.1.6 and 6.1.7 describe two potential solutions for this problem. Automated granularity control reduces the number of tasks in the graph by combining small tasks into larger tasks, coarsening the granularity. The idea of streaming is to extend the data-flow graph model to allow for tasks that can run multiple times and edges that transfer multiple data items, avoiding the overhead of modeling each execution individually. This would allow the scheduler to reason over all executions of a task together, without preventing parallelization at a fine granularity.

5.5 Summary

Pydron, used with the simple scheduler of Chapter 4, allows scientists to focus on the data and the processing, freeing them from the burden of parallelizing the code manually in many use-cases. However, for larger problems this is not sufficient as the execution of the parallel task needs to be adapted to the infrastructure as well to avoid unnecessary data transfers. The scheduling strategy introduced in this chapter can perform this task automatically as well. We don’t target the perfect solution, but trade simplicity of use for performance. Manual assignment of tasks to workers would be possible, but it is non-trivial and typically a short-lived effort as the code and the infrastructure change over time. Our scheduler can make good decisions fully automatically. The language frontend with only two decorators does not need to be extended, nor is it necessary to introduce additional constraints for the developer.

In Chapter 6, a number of ideas further improvements are sketched out. For example to use the advantages of a location-aware scheduler even if the developer chooses to use a file-system as the main means of communication and the Python code is used only to keep track of book-keeping data.
Skills in software development allow us to pursue our own ideas beyond what software developers have prepared for us. This is particularly important in sciences where computers have become an irreplaceable tool. Answers to novel questions lie outside the well-trodden paths for which of-the-shelf software exists, forcing data scientists to write their own code. Single threaded programming used to be sufficient for most tasks, but around 2004 the performance of a single CPU core stagnated and the exponential performance growth could only be continued by utilizing more cores [106]. Even on a laptop, we can only expect a fraction of the performance if the software is not parallelized. In the foreseeable future this trend will continue and the gap between parallelized and unparallelized software will widen further, making it more and more important to utilize multiple cores. Parallel computation is no longer limited to a small percentage of high performance applications but is becoming commonplace.

This trend calls for parallelization tools that can be used by others than professional software developers. In Astronomy, this need is particularly pressing due to the data large volumes which demand high performance infrastructures for processing and analysis. If data scientists can write their own code and utilize parallel infrastructures, it does not just increase their productivity, but also opens the door for new research that requires interactive data exploration. When each development iteration requires meetings with hired software developers and implementation in high-performance languages using a complex parallel computing system, then such research cannot be done in a reasonable amount of time, no matter what hardware is available.
In this dissertation we explored a new approach to make parallel computing accessible. Pydron automatically parallelizes single threaded Python code. While this is impossible in the general case, Pydron succeeds by asking for the developer’s help where needed, and by using a novel combination of static and dynamic analysis based on runtime information. The later proves so powerful that it compensates for the absence of language features in Python, such as static typing, which makes the language difficult to analyze. It allows us keep the help needed from the developer to a minimum, and also enables many other desirable features, such as decoupling the code from the infrastructure. In the course of this dissertation we discovered that we can support arbitrary Python code and provide a system that can run on a large variety of infrastructures, scaling from a single laptop to cloud computing. In Chapter 5 we explored how runtime information can be utilized to make the system more efficient by reducing data transfers. This new approach has more potential than we were able to explore in this dissertation. A number of promising ideas can be found in the next section. None of these require additional help from the developer, keeping the ease-of-use of Pydron intact.

With Pydron, the domain scientists can focus on their research rather than spending their time becoming parallel computing experts or handing the code over to a different person. In astronomy in particular, the data scientists would like to remain in control of their data and code. Our system allows the scientists to utilize their programming skills to explore big data efficiently and independently.

The source code for the prototype implementation of Pydron and all other software components developed in this dissertation are released under open source licenses (see Appendix A).

6.1 Future Extensions

Pydron converts arbitrary, difficult to analyze, Python code into a data-flow representation that makes it easy to reason about the structure of the program. This is achieved by combining static language analysis with dynamic analysis at runtime which provides the system with a wealth of knowledge about the job at hand. Pydron is using this information to parallelize the computation on a coarse granular level. This novel approach has potential beyond what was explored in this dissertation.
6.1. Future Extensions

It could be used with other languages than Python as well. In many ways, Python is a particularly challenging language for such a system. Languages that use static typing would give the system even more information.

Perhaps more interestingly, we have not fully explored what can be achieved with the knowledge gained from using runtime information. The detailed, up-to-date, and accurate data-flow graph lends itself well not just to the coarse granular parallelization studied in this dissertation, but has potential that could significantly extend the scenarios in which Pydron can be used as well as improve the performance and scalability. This dissertation shows that with our approach this powerful graph representation can be generated from arbitrary Python code, thereby opening the door to provide advanced and highly automated features to developers with a minimal API and minimal constraints.

This section sketches some our ideas for future development and research to show the potential the approach of Pydron has.

6.1.1 External data sources

Data can enter the Pydron system through arguments passed to the @schedule function or by invoking operations within that function that load data from an external source, such as a file-system or a database. The first method is limited in size, as the initial call is made from a single Python interpreter, typically on the developer’s workstation. Therefore, when processing large data volumes, the data is typically loaded from within the @schedule function. This can happen either directly in the Python code, or indirectly by external applications that are invoked by the Python code to process the data.

There are numerous APIs available for Python to access possible external data sources, and since Pydron supports arbitrary Python code, the developer is free to use them. There is potential, however, to support these more actively. In particular, if the scheduler is made aware of the location of the data. The location aware scheduler as described in this dissertation can often avoid transfers of data products produced by one task and consumed by the next. Tasks that load data from external sources have no inputs in the data-flow graph other then, for example, a file name. If the system could be made aware of the location of external data, these load operations could already be placed on a worker close to the data. For some applications, particularly if the data volume is reduced early in the processing, this has significant potential to avoid a large portion of the required data transfer.
This is a common optimization in systems such as Hadoop [34], where the distributed file system and the processing are using the same machines. File systems such as HDFS [107] are designed for this purpose and provide the API required to retrieve the information on how a file is split across machines. This allows Hadoop to assign the map operators to the machines that already store the data.

Pydron could use runtime information to detect if a file operation is accessing a file system for which such information is available. This would allow the scheduler to make similar optimizations transparently, without any additional effort from the developer.

In grid infrastructures [108] there is no shared file-system available to all workers. All file-based operations require explicit file transfers. While Pydron can be used on a grid, it provides no support for such file transfers, limiting grid applications to work on Python objects only, as these are transferred by Pydron’s own communication system. Pydron could be extended to detect accesses to files not available on the machine to trigger file transfers. This would be useful beyond loading the initial data since tasks could also use files as a means of communication while the scheduler can minimize the required data transfers as the file locations would be tracked by Pydron. It would also be interesting to explore partial file transfers in cases where only a fraction of the file content is accessed. Condor [109] uses Chirp [110] in such scenarios which effectively forwards open, read, write, close operations to a remote machine, resulting in fine granular transfers. Pydron might be able to improve on this due to the additional information about the application’s code that is available to it. For example, it could use a heuristic to decide if the file should be transferred completely and cached for future use. Since file-based data processing is common in astronomy, such features would make it easier to adopt Pydron in many real-world applications.

6.1.2 Deep scheduling

Pydron’s scheduler is fed with tasks that are ready for execution, which are tasks for which all inputs are available. The scheduler then distributes these tasks preferably to workers that store some or all of the inputs. This approach is also used by other scheduling strategies, such as Quincy [76] which is briefly discussed in Section 2.10. The data-flow graph representation used by Pydron gives us more information that we could employ to make good scheduling decisions.
6.1. Future Extensions

The data-flow graph shown in Figure 6.1 shows an example of a data-flow graph where the scheduling strategy discussed in Chapter 5 is unlikely to result in the optimal distribution of the tasks. Let’s assume that the A tasks execute comparatively fast, but produce a large output. This could be the case if the A tasks load data from a fast external source, such as a distributed file-system. Since the current scheduler is only looking at tasks that are ready for execution, the scheduler only considers the A tasks. Since they have no inputs, there is no data locality advantage visible and they will be distributed arbitrarily. Clearly this is not ideal. Depending on the actual metrics, it might make sense to run two A tasks in sequence on the same worker, even if that means to leave some workers idle, in order to reduce the data transfers later required to execute the B tasks. At the very least the pairs should run on the same machine.

To achieve such behavior, the scheduler would have to look deeper into the graph than just the part of it which is ready for execution. Scheduling then becomes a graph cut optimization problem. There are various options for the optimization criteria. For example, the finish time of the last task would minimize the total execution time. But other goals could be formulated. This could also be combined with automated resource selection, similar to Conductor [79] which would allow to set the goal to finish the computation within a configured time limit at minimal monetary cost.

For Pydron there will be additional challenges since only the tasks that are ready for execution are guaranteed to remain static in the graph. The remainder of the data-flow graph might still change due to graph refinements. Despite these graph changes, the graph is a complete description of the entire execution run at all times. The rules for graph refinements (see Section 4.2.1) allow a deep scheduling strategy to make assumptions on the future structure of the graph. Such a strategy needs to be dynamic so that it can update the scheduling plan as new knowledge about the process is gained. The metrics will need to be extended as well. Currently no estimates for data sizes are required as only products are considered that have been produced already.
6.1.3 Globally distributed computation

The data produced by astronomy projects is archived to that it may serve the community beyond the lifetime of the project. It is customary in astronomy to make such archives publicly available. This allows data analysis across multiple data sources. For example, observations of the same galaxy made by different instruments may be combined to get data in different wavelengths.

If each project uses different tools, APIs, and conventions then significant effort is required by the data scientists to familiarize them with each data source which prevents efficient exploratory data science and thereby reduces the scientific yield gained from astronomy projects. Great efforts have gone into defining standards for data formats and software tools to the extent possible given that each instrument is unique. This has led to the concept of Virtual Observatories (VA) that allows scientists from all over the world to access and analyze data in a user-friendly way. Many countries have VO sites. The International Virtual Observatory Alliance serves as a standard body [111].

One aspect that standardization cannot solve is that the data is distributed across the world. Due to the political constrains (see Section 1.3) this is unlikely to change. As a result, scientists are forced to transfer large volumes of data. To avoid this, VOs can provide infrastructure for computation on-site, but this is of limited help when data from multiple VO sites is required. Ideally, the data processing would be split into parts such that those processing steps that require data from only one VO are run close to the data, reducing the data volume that needs to be transferred for later processing steps that combine results from different VOs. This is tedious to do manually and several questions arise that might be difficult to answer for the developer, such as where to split the application, and on which infrastructure to perform the final phases of the computation.

Pydron has the potential to do this automatically. The location-aware scheduler is designed to find good answers to exactly these questions and the backend is designed to give transparent access to different infrastructures. The scheduler has to be aware of external data sources (see Section 6.1.1) as the initial location of the data plays a central role in such a scenario. Deep scheduling (see Section 6.1.2) is important to allow the scheduler to predict the right point in the process were data should leave the VO. The slow data transfer over internet connections compared to the throughput within a data center, combined with the large data sizes, amplifies the effects of the scheduler’s decisions for good or bad. Future scheduling strategies might need to differentiate between decisions that
effect only data transfers within a data center and decisions that have a potentially larger impact, so that a tradeoff can be reached between fast decisions to keep the workers busy, and decisions that justify more extensive scheduling optimization.

Even the advanced scheduling strategy cannot help if the problem at hand inherently requires large datasets to be processed together. The developer has to be provided with the necessary tools to identify such situations and might desire the possibility to manually intervene with the scheduler’s decision.

### 6.1.4 Simulated execution runs

Astronomy projects have to write proposals to receive the required funds. These proposals require estimates on the cost of the infrastructure needed to process the data. This depends on the data and the processing done on the data. At the time the proposals have to be written the software has not been developed yet, nor does the data exist. For some processing steps the required algorithms might be known, for others there is still research required. The data itself will only become available once the instrument is operational, which might be years in the future. By that time, the hardware will have evolved as well. As the instrument is still in design, the expected raw data changes too.

Making convincing predictions in this phase is challenging. Information from the participating groups that work on parts of the data processing system has to be collected and aggregated into plausible predictions of the entire system. In general, as the project progresses more information is gained and the estimates become more accurate.

Pydron could be modified to provide a valuable tool to make such predictions by simulating execution runs.

The user would sketch the processing as a Python script, resulting in a data-flow graph representing the current best guess of the data processing procedure. The actual processing codes will still be missing at that point, and so is the data, therefore no actual computation is possible. The idea is that the teams would provide estimates for execution time and data sizes for the processing steps they work on. Since the hardware is not available either, Pydron would be provided with estimates on the infrastructure’s capabilities, such as number of cores per machine, number of machines, network bandwidth, etc.

With these estimates, Pydron could simulate an execution and produce a prediction on the execution time. These estimates would be as accurate as the given models, taking into
account how well the process can be parallelized, and to which degree data transfers can be avoided, which is challenging to predict manually.

Most components of Pydron can remain unchanged to add this feature. For example, the scheduler would still operate as normal. Changes are required to simulate task executions without actually spending any time or producing any data, ensuring that the collected metrics report the correct numbers back to the scheduler. Such simulation runs would be very fast. As no time for actual computation is spent, it would be bound by the speed scheduler only.

With such a tool, one can produce estimates quickly and keep them up-to-date when estimates change. It could also be used to decide which hardware to acquire, as it offers a quick way to check the effect it would have to, for example, replace many machines with few cores, with few machines with many cores.

Another application would be for globally distributed computing described in Section 6.1.3. It would give the developer a tool to quickly check to which degree Pydron can avoid costly transfers between VOs without using any actual resources.

### 6.1.5 Simulations on hardware

As the project progresses, there is a point at which infrastructure becomes available. This might be hardware acquired for the specific project, or an existing infrastructure. In both cases there is a need to know if this infrastructure is up to the task. Since the software development is still in progress, some processing steps will be missing. Also, simulation data might not yet be available in sufficient quantity. It is usually not possible to perform actual data processing runs at this stage.

Using estimates for the execution time of the processing steps, and estimates for the size of the data products, Pydron could still put the hardware to the test by replacing the tasks by mock implementations that consume the estimated amount of CPU cycles and by transferring mock data.

This allows intensive hardware tests using the actual data transfer patterns, and might uncover bottlenecks in the infrastructure that can then be addressed before the actual data processing begins. It also serves as an integration test to check if Pydron can correctly and efficiently interact with the infrastructure.
Some work on simulations of this kind has been made in a student project [112] in the context of Euclid [1].

Such a feature could be added to Pydron by replacing the task implementations with mocks. This is already possible, using mock implementations for the \texttt{@functional} functions instead of real ones. To be effective as a tool, Pydron should be able to provide these mocks and provide with a convenient way to configure the estimates for the execution time and data sizes, taking into account that these will be provided not by a single developer but by several separate teams.

### 6.1.6 Streaming

Pydron parallelizes on a coarse granular level. This sets Pydron apart from many other systems, such as those discussed in Section 2.6, which are limited to fine grained parallelism. It does limit the applicability of Pydron for problems that cannot be parallelized at the outer most loops. This limitation results from the overhead Pydron introduces.

For each iteration of a loop, there are a number of task executions and graph changes required to unroll the loop. The scheduler also treats each task individually, which adds to the overhead. For coarse granular parallelism, these overheads are small compared to the execution time, but at a finer granularity they become a bottleneck.

One approach to exploit parallelism at a finer granularity in Pydron would be to add data streaming features. Currently, each task node is executed exactly once, and exactly one data product traverses an edge in the data-flow graph. The graph model of Pydron could be extended to support edges that represent a sequence of data products. Tasks would then consume and produce such streams. The advantage is that not each individual data product and task execution has to be represented explicitly in the graph, avoiding much of the graph refining required for loop unrolling. It would also allow the scheduler to to make decisions not on individual tasks, but on entire sequences of task executions. Section 2.8 discusses several systems that use this approach successfully for fine grained parallelism.

The challenge is that arbitrary Python code often does not have stream semantic and cannot be translated easily into a streaming model. An interesting solution would be a hybrid data-flow graph that allows both stream and non-stream edges. This would allow Pydron to combine the advantages of both worlds.

Python has some language features that have streaming semantic, in particular iterators and generators. An iterator represents a sequence of objects that can be consumed one by
one without the ability of random access or to move backwards. That makes iterators ideal to represent data streams. A generator is a special type of iterator that is produced by a Python function using the \texttt{yield} expression. Every time an element is consumed from the generator, the function runs until the next \texttt{yield} statement, at which point the execution is paused until another element is requested. Generators might themselves consume elements from other iterators. This makes generators very similar to tasks in a streaming system.

The translator could model these language features as streams which would give the developer a way to access the advantages of a streaming system through regular Python code without any additional APIs. What’s more, the developer would not be limited to language features that translate into streams, as the existing functionality of Pydron, including its support for arbitrary Python code, would be preserved.

Research is required to define how such a hybrid data-flow graph would work and what additional constraints are necessary when mixing streaming with non-streaming tasks.

Streaming would not be limited to functions provided by the user. Since generators and iterators are a regular language features, they are used in many libraries that could be combined with user written code. For example, functions such as the built-in \texttt{map} and \texttt{filter} functions could be used on streams.

\subsection{Automatic granularity control}

Streaming support will allow Pydron to efficiently work with fine granular use-cases in situations where the data streams can be detected in the code, for example through use of generators. This might not always be the case though. Explicit loops can also result in a large numbers of small tasks. The overhead of maintaining the data-flow graph and scheduling can quickly become the bottleneck in such scenarios.

The data-flow graph structure allows us to combine tasks to make the granularity coarser. A sub-graph can be replaced by a single task as long as the sub-graph is closed in the sense that all tasks that produce intermediate data products of the sub-graph are within the sub-graph. This sub-graph could then be translated back to Python code and executed as a regular function, thereby avoiding the overhead of Pydron. Ideal candidates are groups of tasks with dependencies between them that force a sequential execution anyway. The scheduler could use the metrics to decide how to balance the granularity with the utilization of the workers. If the execution time of a task is less than the overhead, merging tasks
6.1. Future Extensions

is beneficial even when not all workers are occupied. If the tasks are significantly larger than the overhead, there is little to be gained. In the area in between the scheduler could use heuristics to make decisions. These do not have to final, as the scheduler always has the option to unpack the tasks again.

The inline substitution feature explained in Section 4.2.3 could also participate in automatic granularity control. If the developer has structured the translated code into multiple functions, these are interesting candidates to be executed as single tasks as they are likely to represent a good structure of the processing. The way the code is structured can therefore provide hints for automatic granularity control. The same is true with loops. Loop bodies are also candidates for merging, in particular if they include nested loops.

The very first prototype of Pydron included the ability to translate the data-flow graph back to Python code. This feature later dropped out due to the added complexity in favor of other features. But some early experiments have shown that this approach is feasible and has potential.

6.1.8 Tooling

A drawback of a highly automated system such as Pydron is that it can be difficult for users to understand what is happening inside the system. This makes it hard to predict and how their code affects the performance. If Pydron fails to parallelize a loop, the execution time can be several orders of magnitude longer. If the system is a black-box, the user might only realize that something is wrong once the computation has run for an unexpectedly long time. Since even a parallelized execution run can take hours, and since there can be significant monetary costs associated with the infrastructure, this has to be avoided.

Monitoring tools could be developed that give feedback on the progress and the utilization of the infrastructure. This could include a graphical representation of the data-flow graph which updates as the graph is refined. The information required for such a tool is easily available on the master worker that keeps track of the graph and the status of all workers. Such a tool could also give the user the ability to make changes to the system on-the-fly, such as starting or stopping workers.

If provided as an plug-in to the development environment, syntax highlighting could be used to give feedback to the developer as well. This could be as simple highlighting the
lines of code that are currently being processed or as advanced as debugging support that would allow to pause the process and inspect the content of the variables.

Tools could also include code checkers that attempt to identify possible issues with the code, such as calls to non-functional methods within a loop, that might hinder parallelization, or violations of the \texttt{functional} contract. The usefulness of static analysis in Python is limited due to its dynamic nature, but if used to produce warnings, the analysis must not be perfectly accurate and some false warnings are acceptable. Such code checkers are already in use for Python code [113] and the same approach could be used to check against common problems of code used with Pydron.

Once the execution is running, runtime information can be used as well to further improve these hints.

Simulating execution runs, as described in Section 6.1.4, would also provide a helpful tool to the developer. The model of the hardware would not have to be very accurate to give the developer an idea on how well Pydron is able to parallelize the code.

### 6.1.9 Metrics

The metrics described in Section 5.1 are very simple. There is significant room for improvement. For example, some of the systems discussed in Section 2.10 make the assumption that the required processing time is proportional to the input data size. Since applications parallelized with Pydron might not be as regular as MapReduce jobs, this might not be the right strategy for all tasks. A more generic solution would be to apply machine learning methods to predict the execution time based on the operation and the input sizes.

As mentioned in Section 5.1.3, a topology aware model of the network would be a possible extension of the current network metrics. This would give the scheduler the information needed to avoid bottlenecks in the network topology resulting from too much throughput to a single link or node. There has been significant research into this topic, as topology awareness can greatly increase the efficiency of a system [114, 115]. These systems improve the overall throughput by optimizing the order and paths of the individual transfers. Pydron could employ such methods as well to optimize the transfers required for the tasks that have been assigned to workers. We could go beyond this, however. Pydron can also manipulate which transfers are required in the first place. The scheduler might have significantly more tasks that are ready for execution than there are cores, allowing
the scheduler to choose those that won’t cause network congestion. It could also decide on which workers these tasks are placed and thereby changing the path the data takes through the network. This additional flexibility might be exploitable to further increase utilization of the network.

6.1.10 Fault tolerance

With the trivial scheduling strategy described in Section 4.3.2, fault tolerance is achieved easily by re-scheduling the task which was running on a failed worker. This provides fault tolerance for all but the master worker. This works since all created data products are always transferred to the master. For a location aware strategy, such as the one described in Chapter 5, a more advanced solution is required. Since data products are stored on the workers, we lose data if a worker fails.

One approach is replication of the data products, but this requires a significant amount of data transfer. The alternative is to recalculate lost data products. The data-flow graph includes all the required information necessary to identify which tasks have to be run again. To save memory, Pydron deletes intermediate data products once all the tasks have completed that depend on it. This implies that a failed worker might require a significant amount of reprocessing since intermediate data products need to be reproduced as well, even if they were produced on different workers.

When storing the execution times of tasks, and the data sizes of products, even after they’ve been freed, the system can accurately calculate the cost of a potential failure. This information could be used to identify and select data products in key positions for replication. This would give a trade-off between replication and recovery time. The scheduler could assist in this process as well. The assignments of tasks to workers define how the data products are distributed among the workers. A transfer of a data product required as a task input is also a replication of that product. Reducing the cost of a potential failure can be modeled as another optimization goal of the scheduling strategy.

6.2 Towards interactive data exploration

Pydron is a contribution towards the greater goal of interactive data exploration in astronomy. It successfully shows a way to give the scientists a turing complete and easy
to use interface to formulate their data processing needs that can scale to large parallel infrastructures. While this is an important step towards an interactive system, and can by itself significantly increase productivity, other components, as well as further research in the parallelization system is required to achieve true interactivity.

Interactivity is mostly interesting for small and medium size problems without inherently sequential processing requirements, as we cannot realistically assume to achieve response times of a few seconds for arbitrarily large and complex use-cases. The coarse granular nature of Pydron becomes a limiting constraint in such a scenario. If we wish the results to appear in less than a second, the granularity at which the computation is parallelized has to be smaller than that. The coarse granularity at which Pydron operates is not an inherently a result of semi-automatic parallelization, but primarily a design decision made to make Pydron applicable in the current, non-interactive, work processes of astronomy.

While the coarse granularity does simplify the system, support for finer granularity could be achieved, for example, using the streaming extension suggested in Section 6.1.6.

Processing large data volumes with short response times puts additional pressure on data handling aspects as well. Careful scheduling to avoid data transfers (see Section 6.1.2), and close interaction with the underlying data storage (see Section 6.1.1) are unavoidable. Adapting to an existing data storage solution will only help to a certain degree. To improve performance further, data management has to be an integrated part of the system, so that not only the scheduling adapts to the storage but the storage adapts to the needs of the computation. Cloud infrastructures give us new options to dynamically adapt the infrastructure to needs of throughput, latency, and capacity.

The majority of data processing in astronomy is currently done with batch processing. Where a continuous stream of data exists, for example from an active instrument, the data is often batched together for nightly processing. Pydron is designed to fit well into such a workflow. For an interactive system, this would have to be extended to support incremental computation [116], saving time by only recomputing the parts of the process required to account for the change performed by the user. The data-flow representation of the program puts Pydron in a good position to provide such a feature.

Incremental computation can be seen as a part of a larger change from the current system that operates on a per-run basis, to one where long running sessions allow continuous interaction between the user and the system. This would allow Pydron to invest resources into, for example, building of search index structures, as these efforts can be amortized over the duration of the entire session. The knowledge of Pydron about the computation at
6.2. Towards interactive data exploration

hand can guide these decisions. Other than traditional database systems, our system does not just see the current operation, but has information about the entire process, including information about future operations on the data. This further stresses the need for a tighter integration of data management aspects. Interactive data exploration has been primarily studied from the perspective of database systems. Such extensions to Pydron would reduce the gap between Pydron and databases, attacking the problem of interactive data exploration from a programming direction.

Performance is only one required component of interactivity. The human–computer interface is just as important. The data has to be presented in a form that makes it interpretable by the scientists, and the user needs the ability to instruct the system on what information to process and display next. Human–computer interaction and scientific data visualization are entire research areas by themselves. A large number of libraries and tools exist for data visualization. The large and ever changing variety of use-cases in astronomy makes it difficult to provide tools that are sufficiently generic. Flexible plotting libraries [26] have emerged that are intensively used to create the required visualizations on-demand. These work well in a batch processing workflow, but for interactivity just displaying data is not sufficient, the user has to manipulate the visualization as well.

The visualization system cannot assume that the entire dataset is preprocessed and only requires filtering and displaying. This implies communication between the visualization system and the processing system, as one provides the instructions and the other provides the data. The complexity of the problems addressed in astronomy today requires a turing complete programming language as a frontend. This stands in contrast to the need for interactivity, as program code is not well suited for interactive manipulation, in particular with the touch based interfaces of today. Currently, visualization and code are mostly treated separately. This might have to change. Workbook-style systems such as IPython (see Section 2.3) have made first steps towards a closer integration of visualization and code. For an interactive system, changes to the code should immediately be reflected in the visualization and vice versa.
Appendix – Open source releases

A prototype of Pydron is released as open source under the MIT license which allows use of Pydron without having the release the code it is used with. The prototype is functional but not yet at a stage where it could be considered a product ready for use, mostly due to a lack of supporting tools (see Section 6.1.8). The source code is available under https://github.com/pydron/pydron it is also available in the PyPI package repository under the name pydron.

To support Pydron, a number of additional libraries were developed. There are available as independent products as they may prove useful outside the context of Pydron.

**remoot** This library implements the backend functionality described in Section 4.4 by providing an API to start Python interpreters on various infrastructures and the means to establish communication with them. The released version includes support for multi-core, cluster, and cloud computing. Source code: https://github.com/pydron/remoot, PyPI Package: remoot.

**ifaddr** Python 2.7 lacks an API to find out what network interfaces the local machine has, and what IP addresses are bound to them. This is a small library that provides this functionality by accessing the operating system’s API directly. It has support for Windows, Linux, and OSX. Other than similar libraries, ifaddr does not need to be compiled which makes installation on some systems considerably easier. Source code: https://github.com/pydron/ifaddr, PyPI Package: ifaddr.
**picklesize** This library can calculate the size of an object in its serialized form without actually serializing the data which can be significantly faster, particularly if large NumPy arrays are involved. *Picklesize* is essentially a complete re-implementation of the *pickle* serialization system, but instead of aggregating the data the sizes are summed up. The use of this library is described in Section 5.1.2. Source code: [https://github.com/pydron/picklesize](https://github.com/pydron/picklesize), PyPI Package: *picklesize*.

**sourblossom** We use *twisted* as the main library to perform network operations as it is asynchronous. Unfortunately, there is no remote procedure call (RPC) library available on top of twisted. *Sourblossom* is the second RPC system developed as part of this dissertation. The first one, named *anycall*, is available online too. *Sourblossom* has a unique feature, it allows for very large objects to be passed as function arguments or return values. It does this by streaming the data from serialization to network and back to serialization without ever storing the entire object. The library includes a message passing layer with the same feature. The communication backend is replaceable, the current implementation supports TCP/IP. Source code: [https://github.com/pydron/sourblossom](https://github.com/pydron/sourblossom), PyPI Package: *sourblossom*.

**twistit** *Twistit* is a collection of utility functions that have proven useful when working with the *twisted* library. Some notable features include a replacement for *inlineCallback* that supports cancellation, decorator for timeouts, and a detector that can help find the cause if the reactor thread is blocked due to a bug. Source code: [https://github.com/pydron/twistit](https://github.com/pydron/twistit), PyPI Package: *twistit*.

**utwist** Testing applications written with twisted is complicated by the fact that *twisted* requires the main IO thread to run for many operations. Twisted provides its own unit-testing framework that takes care of this issue. Unfortunately, it is based on an older version of *unittest*, lacking many of its newer features. It is also a invasive solution as it requires all unit tests of the project to be tested with this framework, even if they test parts of the code that does not use *twisted*. Finally, it lacks integration into development environments. *utwist* is an alternative solution that makes it possible to use twisted with any unit-testing frameworks by adding a decorator to those methods that require twisted. Source code: [https://github.com/pydron/utwist](https://github.com/pydron/utwist), PyPI Package: *utwist*.


Curriculum Vitae

Stefan C. Müller

Education

2010-2016  Ph.D. in Computer Science
           Systems Group, Departement of Computer Science, ETH Zürich
           Advisor: Prof. Dr. Gustavo Alonso

2008-2010  Master of Science in Computer Science
           Computational Science track
           Departement of Computer Science, ETH Zürich

2003-2006  Engineer’s Degree in Computer Science
           Enterprise Computing track
           University of Applied Sciences Northwestern Switzerland
Curriculum Vitae

Professional Activities

2015-2016  Research Assistant
            ETH Zürich
            Research and teaching in the Systems Group

June-September  Intern
2014            Amazon Inc.
            Development and research in the ElastiCache group
            of Amazon Web Services.

2007-2015  Research Assistant
            University of Applied Sciences Northwestern Switzerland
            Institute of 4D Technologies
            Research and development in projects with industry partners.

2000-2004  Your-Web GmbH
            Co-owner

Publications

2014  Müller, S. C., Alonso, G. and Csillaghy A.
      Scaling Astroinformatics: Python + Automatic Parallelization.
      IEEE Computer Sept. 2014.

      Pydron: semi-automatic parallelization for multi-core and the cloud.
      OSDI 2014.