KTH Computer Science
and Communication

Optimization of parallel transformation of data sets with processing order constraints in Python

Duis autem vel eum iruire dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat null

DEXTER GRAMFORS

Master's Thesis at NADA
Supervisor: Stefano Markidis
Examiner: Erwin Laure

TRITA xxx yyyy-nn

Abstract

This is a skeleton for KTH theses. More documentation regarding the KTH thesis class file can be found in the package documentation.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Mauris purus. Fusce tempor. Nulla facilisi. Sed at turpis. Phasellus eu ipsum. Nam porttitor laoreet nulla. Phasellus massa massa, auctor rutrum, vehicula ut, porttitor a, massa. Pellentesque fringilla. Duis nibh risus, venenatis ac, tempor sed, vestibulum at, tellus. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos.

Referat

Lorem ipsum dolor sit amet, sed diam nonummy nibh eui mod tincidunt ut laoreet dol

Contents

List of Figures

Definitions

IPC - Interprocess communication.

MPI - Message Passing Interface. Standardized interface for message passing between processes.

Embarrassingly parallel - A problem that is embarrassingly parallel can easily be broken down into components that can be run in parallel.

CPU bound - Calculation where the bottleneck is the time it takes for a processor to execute it.

I/O bound - Calculation where the bottleneck is the time it takes for some input/output call, such as file accesses and network operations.

Real time - The total time it takes for a call to finish.

User time - The time a call takes, excluding system overhead; the time the call spends in user mode.

System time - The time in a call that is consumed by system overhead; the time the call spends in kernel mode.

DAG/Directed acyclic graph A directed graph that contains no directed cycles.

Chapter 1

Introduction

1.1 Area of interest

The subject of parallel computing is one that has become highly relevant in recent years. Moore's law, the observed pattern that the number of transistors in a dense integrated circuit doubles approximately every two years [19], has lost its relevance. The increased processor clock speed that the doubling in processors implies is no longer present because of overheating issues [14, p. 1]. Because of this, manufacturers of processors now have largely turned to *multicore* processors. In a multicore architecture, several cores which work as individual processors execute code simultaneously. Using this type of architecture to work on a single task to increase performance is known as *parallelism*.

Efforts to exploit parallelism automatically from a program have been made; however, the benefits of these have reached their limit [17, p. 7-12]. In order to fully utilize the increase in performance that multicore architectures promise, programmers today must instead turn to explicit parallel programming.

Python is one of world's most popular programming languages [16]. It is used extensively both at schools and in the industry, and its benefits include expressiveness, portability, and the fact that it is easy to learn. Python has support for parallel programming, although it has caveats and overheads associated with a concurrency-hampering mechanism called the *Global Interpreter Lock* [6].

This thesis concerns a combination of the areas mentioned above: parallel computing using Python.

1.2 TriOptima

The thesis is conducted at TriOptima, a company that provides different services for the OTC derivatives market. OTC derivatives concern trading directly between two parties, and the customers include large banks. TriOptima's services include portfolio compression, reconciliation, dispute resolution, and risk management. The services deal with substantial amounts of data, and face challenges such as high security demands, availability requirements, and speed optimization for data transformations and risk calculations. In their reconciliation and dispute resolution service, triResolve, customers upload datasets representing trades.

1.3 Problem statement

1.3.1 Dataset standardization

The datasets mentioned in the previous section need to be processed in order to transform them into a standard format which makes comparisons between data from different customers possible. In some cases, the size of the dataset is large enough that this transformation is slow, and could conceivably be sped up through the use of parallelization. The sizes are aptly measured in number of rows, and range between 2 rows to about 1490000 rows. The time it takes to process the datasets range between 0.06 seconds and 15200 seconds 2 (4.2 hours).

1.3.2 Transformation with constraints

The datasets are associated with a file format. The format specifies a set of rules, known as filters, which at times enforce constraints on the processing order in the file when performing the transformation. These constraints reduce the possible benefits of parallelization as they enforce inherently serial parts of the transformation program. Since the size of the datasets as well as the type and number of their associated filters varies, it is plausible that the benefits of parallelization will differ significantly between different datasets. An overhead is associated with creating new threads or processes. This overhead is increased in Python as the data shared between processes needs to serialized. Therefore, it is possible that parallelization of datasets will result in slower execution in some cases. Consequently, it is interesting to find the combinations of dataset sizes, as well as their filters, that result in beneficial parallelization, and which do not. Additionally, the complex nature of the system makes the implementation of the parallelization an interesting problem.

1.4 Research question

Given the size of a dataset and its set of filters, is is possible to the determine if parallelization of the data transformation using Python will be beneficial or not?

The thesis question gives rise to the following subquestions:

- What is the best approach for parallelizing code in Python in order to minimize data races and maintain performance?
- How should the parallel performance be measured?
- What kind of data dependencies exist and how do they affect parallelization?
- What kind of overhead does parallelization introduce?

1.5 Objective

The objective of this thesis is to answer the questions stated above using a literature study and by implementing a working parallelization of the existing dataset processing program.

1.6 Delimitations

The implementation and research in this thesis is limited to the parallelization of an existing program, and no new code for the core problem of processing the datasets will be written. Another delimitation of the thesis is that it does not compare different methods of parallelization, and uses only the Python `multiprocessing` module (chosen with support of related work in the field).

1.7 Contribution

This thesis focuses on parallelization analysis of a file format rather than the more conventional method of analyzing source code. Additionally, it shows how Python can be effectively used for parallelization in a complex system not built for parallelization from the start. The fact that the parallelized system relies on database operations and, consequently, I/O is another aspect of the thesis that may interest other researchers in the field of parallel programming. Similar projects can use the conclusions of this thesis as a foundation when creating a parallelization strategy.

Chapter 2

Related work

2.1 Parallelization of algorithms using python

Ahmad et al. [3] parallelize path planning algorithms such as Dijkstra's algorithm using C/C++ and Python in order to compare the results and evaluate each language's suitability for parallel computing. For the Python implementation, both the `multiprocessing` and `threading` packages are used. The authors identify Python as the preferable choice in application development, due to its safe nature in comparison to C and C++. The implementation using the `threading` module resulted in no speedup over the sequential implementation. Parallelization using the `multithreading` module resulted in a speedup of 2.5x for sparse graphs, and a speedup of 6.5x for dense graphs. The overhead introduced by the interpreted nature of Python, as well as the extra costs associated with Python multiprocessing, was evident as the C/C++ implementations showed both better performance and better scalability. The slowdowns for sparse graph of Python compared to C/C++ ranged between 20x to 700x depending on the graphs. However, the authors note that the parallel Python implementation exhibits scalability in comparison to its sequential implementation. The experiments were conducted on a machine with 4 cores with 2-way hyperthreading.

Cai et al. [8] note that Python is suitable for scientific programming thanks to its richness and power, as well as its interfacing capabilities with legacy software written in other languages. Among other experiments on Python efficiency in scientific computing, its parallel capabilities are investigated. The Python MPI package `Pypar` is used for the parallelization, using typical MPI operations such as send and receive. The calculations, such as wave simulations, are made with the help of the `numpy` package for increased efficiency. The authors conclude that while communication introduces overhead, Python is sufficiently efficient for scientific parallel computing.

Singh et al. [24] present Python as a fitting language for parallel computing, and use the `multiprocessing` module as well as the standalone `Parallel Python` package in their experiments. Because of the communication overhead in Python, the study focuses on embarrassingly parallel problems where little communication is needed. Different means of parallelization are compared: the Pool/Map approach, the Process/Queue approach, and the Parallel Python approach. In the Pool/Map approach, the simple functions of `multiprocessing.Pool` are used to specify a number of processes, a data set, and the function to be executed with each element in the dataset as a parameter. In the Process/Queue approach, a `multiprocessing.Queue` is spawned and filled with chunks of data. Several `multiprocessing.Process` objects are then spawned, which all share the queue and get data to operate on from it while it is not empty. Another shared queue is used for collecting the results. In the Parallel Python approach, the `Parallel Python` abstraction *job server* is used to submit tasks for each data chunk. The tasks are automatically executed in parallel by the job server, and the results are collected when they have finished. The results in general show significant time savings even though the approaches taken are relatively straightforward. The best performance is achieved when the number of processes is equal to the number of physical cores on the computer. The Process/Queue is shown to perform better than both Pool/Map and parallel Python. This comes at the cost of a slightly less straightforward implementation. The impact of load balancing and chunk size is also discussed, with the conclusion that work load should be evenly distributed among cores as computation is limited by the core that takes the longest to finish.

Rey et al. [23] compare `multiprocessing` and `Parallel Python` with the GPU-based parallel module `PyOpenCI` when attempting to parallelize portions of the spatial analysis library PySAL. In particular, different versions of the Fisher-Jenks algorithm for classification are compared. For the smallest sample sizes, the overhead of the different parallel implementations produce slower code, but as the sample sizes grow larger the speedup grows relatively quickly. For the largest of the sample sizes, the speedup curve generally flattens out; the authors state this as counter-intuitive and express an interest in investigating this further. In general, the CPU-based modules `multiprocessing` and `Parallel Python` perform better than the GPU-based PyOpenCI. The `multiprocessing` module produced similar or better results than the `Parallel Python` module. While the parallel versions of the algorithm perform better, the bigger implementation effort associated with it is noted.

In the work above, the code that is parallelized is strictly CPU bound. This differs from this thesis, as a portion of the to be parallelized program is I/O bound due to database interactions. Another difference is the fact that the parallelization analysis conducted in this thesis is mainly done on the file format level rather than at program level, like the work above. However, the works highlight aspects of parallelization using Python that are useful in achieving the thesis objective. These include parallelization patterns, descriptions of overhead associated with parallel programming in Python, and comparisons between different Python modules for parallelization.

2.2 Python I/O performance and general parallel benchmarking

In their proposal for the inclusion of the `multiprocessing` module into the Python standard library, Noller and Oudkerk [20] include several benchmarks where the `multiprocessing` module's performance is compared to that of the `threading` module. They emphasize the fact that the benchmarks are not as applicable on platforms with slow forking time. The benchmarks show that while naturally slower than sequential execution, `multiprocessing` performs better than `threading` when simply spawning workers and executing an empty function. For the CPU-bound task of computing Fibonacci numbers, `multiprocessing` shows significantly better result than `threading` (which is in fact slower than sequential code). For I/O bound calculations, which is an application considered suitable for the `threading` module, the `multiprocessing` module is still shown to have the best performance when 4 or more workers are used.

The benchmarks where performed using the following hardware:

- 4 Core Intel Xeon CPU @ 3.00GHz
- 16 GB of RAM
- Python 2.5.2 compiled on Gentoo Linux (kernel 2.6.18.6)
- pyProcessing 0.52

While this work is a relatively straightforward benchmark under ideal conditions, the fact that `multiprocessing` shows better performance than `threading` for both CPU bound and I/O bound computations contributed to the decision to use `multiprocessing` in this thesis.

2.3 Comparisons of process abstractions

Friborg et al. [11] explore the use of processes, threads and greenlets in their process abstraction library PyCSP. The authors observe the clear performance benefits of using multiprocessing over threads due to the circumvention of the GIL that the `multiprocessing` module allows. Greenlets are user-level threads that execute in the same thread and are unable to utilize several cores. On Microsoft Windows, where the fork() system call is not available, the process creation is observed as significantly slower than on UNIX-based platforms. While serialization and communication has a negative impact on performance when using `multiprocessing`, the authors state that this produces the positive side-effect of processes not being able to modify data received from other processes.

The work above focuses on process abstractions in a library, but comes to conclusions that are helpful in this thesis; `multiprocessing` has performance benefits over the other alternatives, and also introduces safety to a system thanks to less modification of data sent between processes.

2.4 Parallelization in complex systems using Python

Binet et al. [7] present a case study where parts of the ATLAS software used in LHC (Large Hadron Collider) experiments are parallelized. Because of the complexity and sensitivity of the system, one of the goals of the study is to minimize the code changes when implementing the parallelization. The authors highlight several benefits of using multiple processes with IPC instead of traditional multithreading, including ease of implementation, explicit data sharing, and easier error recovery. The Python `multiprocessing` module was used to parallelize the program, and the authors emphasize the decreased burden resulting from not having to implement explicit IPC and synchronization. Finding the parts of the program that are embarrassingly parallel and parallelizing these is identified as the preferred approach in order to avoid an undesirably large increase in complexity while still producing a significant performance boost. The parallel implementation was tested by measuring the user and real time for different numbers of processes. These measurements show a clear increase in user time because of additional overhead, but also a steady decrease in real time.

Implementing parallelization of a component of a large system without introducing excessive complexity is a goal of this thesis, similar to the work above. The above approach to parallelization, identifying embarrassingly parallel parts of the system and focusing on these, were used in this thesis. Again, this thesis differs from the above by having an I/O bound portion and by analysing a file format for parallelizability.

2.5 Summary of related work

Common themes and conclusions in the related work presented above include:

- Python is a suitable language for parallel programming.
- The `multiprocessing` module is successful in circumventing the GIL and consistently shows the same or better performance than other methods, even for I/O bound programs.
- The overhead that IPC introduces when creating parallel Python programs makes it imperative to minimize communication and synchronization. Consequently, embarrassingly parallel programs are preferable when using Python for parallelization.
- For existing larger systems, extensive parallelization may produce undesired complexity.

Chapter 3

Theory

In this chapter, theory related to multicore architecture and parallel programming is explained in order to give the reader the foundation needed to understand these aspects of the thesis.

3.1 Multicore architecture

3.1.1 Processes vs threads

While both threads and processes represent contexts in which a program is run, they have a few differences. A thread is run inside a process, and the threads within the process share memory and state with each other and the parent process [24]. Individual processes do not share memory with each other, and any communication between processes must be done with message passing rather than with shared memory. Consequently, communication between threads is generally faster than between processes. Typically, different threads can be scheduled on different cores, which is also true for different processes.

3.1.2 Multicore communication and caching

Multiple processors communicate with each other through a bus or a network [14, p. 472-476]. Since the means of communication between the processes is a finite resource, too much traffic may result in delays. The processors typically have their own cache. In order to avoid unnecessary reads from the slower main memory, processors may read from another processor that has the requested data cached. In a process called *cache coherence*, shared cached values are kept up to date using one of several protocols. The effect that these different means of communication between processors has on performance in multiprocessor programs should not be ignored.

3.2 Parallel shared memory programming

3.2.1 Data parallelism

Data parallelism denotes code where the parallelism comes from decomposing the data and running it with the same piece of code across several processors or computers [24]. It allows scalability as number of cores and problem sizes increase, since more parallelism can be exploited for larger datasets [17, p. 24].

3.2.2 Task parallelism

In task parallelism, groups of tasks that are independent are run in parallel [10]. Tasks that depend on each other cannot be run in parallel, and must instead be run sequentially. A group of tasks is embarrassingly parallel if none of the tasks in the group depend on each other.

3.2.3 Scheduling

Threads and processes are scheduled by the operating system, and the exact mechanism for choosing what to schedule when differs between platforms and implementations [14, p. 472]. Scheduling may imply running truly parallel on different cores, or on the same core using time-slicing. Threads and processes may be descheduled from running temporarily for several reasons, including issuing a time-consuming memory request.

3.3 Performance models for parallel speedup

3.3.1 Amdahl's law

Amdahl's law [4] states that:

> The effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude.

Amdahl divides programs into two distinct parts: a parallelizable part and an inherently serial part [14, p. 13]. If the time it takes for a single worker (for example, a process) to complete the program is 1, Amdahl's law says that the speedup S of the program with n workers with the parallel fraction of the program p is:

$$S = \frac{1}{1 - p + \frac{p}{n}}$$

The law has the following implication: if the number of workers is infinite, the time it takes for a program to finish is still limited by its inherently serial fraction. This is illustrated below:

$$\lim_{n \to \infty} \frac{1}{1 - p + \frac{p}{n}} = \frac{1}{1 - p}$$

$1 - p$ is the serial fraction which clearly limits the speedup of the program even with an unlimited number of processors.

3.3.2 Extensions of Amdahl's law

Che and Nguyen expand on Amdahl's law and adapts it to modern multicore processors [9]. They find that more factors than the number of workers affect the performance of the parallelizable part of a program, such as if the work is more memory bound or CPU bound. In addition, they find that with core threading (such as hyperthreading), superlinear speedup of a program is achievable and that the parallelizable part of a program is guaranteed to also yield a sequential term due to resource contention.

Yavits et al. come to similar conclusions [27]. They find that it is important to minimize the intensity of synchronization operations even in programs that are highly parallel.

3.3.3 Gustafson's law

Gustafson's law [13] is a result of the observation that problem sizes often grow with the number of processors, an assumption that Amdahl's law dismisses, keeping the problem size fixed. With this premise, a program can be run with a larger problem size in the same time as more workers are added. This view is less pessimistic than Amdahl's law, as it implies that the impact of the serial fraction of a program becomes less significant with many workers and a large problem size [17, p. 61-62].

The speedup S, for n workers, and s and p as the time spent in the serial and parallel parts in the parallel system, respectively, is achieved by:

$$S = n + (1 - n) * s$$

3.3.4 Work-span model

The tasks that need to be performed in a program can be arranged to form a directed acyclic graph, where a task that has to be completed before another precedes it in the graph. The work-span model introduces the following terms [17, p. 62-65]:

- **Work** - The work of a program is the time it takes to complete with a single worker, and equals the total time it takes to complete all of the tasks. The work is denoted T_1.

- **Span** - The span of a program is the time it takes for the program to complete with an infinite number of workers. The span is denoted T_∞.

- **Critical path** - The tasks that are included in the path that has the maximum number of tasks that need to be executed in sequence. The span is equal to the length of the critical path.

An example of a task DAG can be found in figure 3.1.

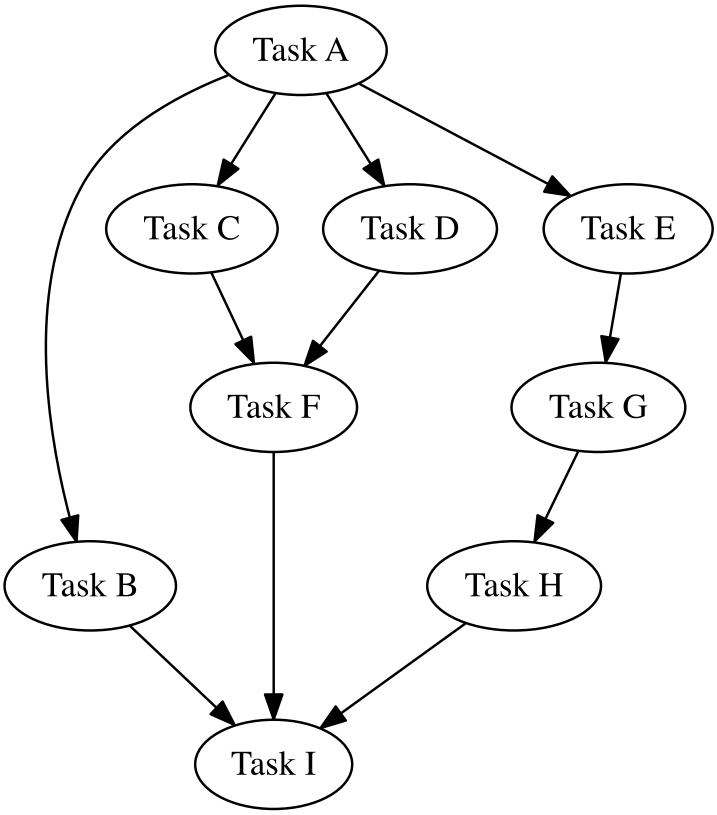


Figure 3.1. An example of a task DAG used in the work-span model. Assuming each task takes time 1 to complete, this DAG has a *work* of 9 and a *span* of 5.

In the work-span model, the following bound on the speedup S holds:

$$S \leq \frac{T_1}{T_\infty}$$

With n workers and running time T_n, the following speedup condition can be derived:

$$S = \frac{T_1}{T_n} \approx P \text{ if } \frac{T_1}{T_\infty} \gg P$$

In essence, this means that linear speedup can be achieved under the condition that the work divided by the span is significantly larger than the number of workers.

The work-span model implies that increasing the work in an excessive manner when parallelizing may result in a disappointing outcome. It also implies that the span of the program should be kept as small as possible in order to utilize parallelization as much as possible.

Chapter 4

Method & Materials

Purpose of chapter, TODO

4.1 Python performance and parallel capabilities

There are several implementations of the Python language. This section will focus on CPython, the canonical and most popular Python implementation [22], and also the one that TriOptima uses.

4.1.1 Performance

The general performance of CPython is slower than other popular languages such as C and Java for several reasons [5]. Overhead is introduced due to the fact that all operations need to dispatched dynamically, and accessing data demands the dereferencing of a pointer to a heap data structure. Also, the fact that late binding is employed for function calls, the automatic memory memory management in the form of reference counting, and the boxing and unboxing of methods contribute to the at times poor performance.

4.1.2 The GIL, Global Interpreter Lock

In order to simplify the implementation and to avoid concurrency related bugs in the CPython interpreter, a mechanism called the Global Interpreter Lock - or the GIL - is employed [21]. The GIL locks the entire CPython interpreter, making it impossible for multiple Python threads to make progress at the same time, thereby removing the benefits of parallel CPU bound calculations [12]. When an I/O operation is started from Python, the GIL is released. Efforts to remove the GIL have been made, but have as of yet been unsuccessful.

4.1.3 Threading

The Python `threading` module provides a multitude of utilities for concurrent programming, such as an object abstraction of threads, locks, semaphores, and condition objects [1]. When using the `threading` module in CPython, the GIL is in effect, disallowing true parallelism and hampering efficient use of multicore machines. When performing I/O bound operations, the `threading` module can be used to improve performance; at times significantly [25, p. 121-124]

4.1.4 Multiprocessing

The `multiprocessing` module has a similar API to the `threading` module, but avoids the negative effects of the GIL by spawning separate processes instead of user threads. This works since the processes have separate GILs, which do not affect each other and enables the processes to utilize true parallelism [25]. The processes are represented by the `multiprocessing.Process` class.

The `multiprocessing` module provides mechanisms for performing IPC. In order for the data to be transferred between processes, it needs to be serializable through the use of the Python `pickle` module [25, p. 143]. When transferring data, it is serialized, sent to another process through a local socket, and then deserialized. These operations, in conjunction with the creation of the processes, gives the `multiprocessing` module a high overhead when communicating between processes.

The two main facilities that the `multiprocessing` module provides for IPC are [21]:

- `multiprocessing.Pipe`, which serves as a way for two processes to communicate using the operations `send()` and `recv()` (receive). The pipe is represented by two connection objects which correspond to each end of the pipe. See figure 4.1 for an example.

- `multiprocessing.Queue`, which closely mimics the behaviour and API of the standard Python `queue.Queue`, but can be used by several processes at the same time without concurrency issues. This `multiprocessing` queue internally synchronizes access by multiple processes using locks, and uses a *feeder thread* to transfer data to other processes. See figure 4.2 for an example.

In addition to the parallel programming utilities mentioned above, the `multiprocessing` module provides the `Pool` abstraction for specifying a number of workers as well as several ways of assigning functions for the workers to be performed in parallel. For example, a programmer can use `Pool.map` to make the workers in the pool execute a specified function on each element in a collection. See figure 4.3 for an example.

```
def worker(conn):
    conn.send("data")
    conn.close()

parent_conn, child_conn = Pipe()
p = Process(target=worker, args=(child_conn,))
p.start()
handle_data(parent_conn.recv())
p.join()
```

Figure 4.1. `multiprocessing.Pipe` example

```
def worker(q):
    q.put("data")

q = Queue()
p = Process(target=worker, args=(q,))
p.start()
handle_data(q.get())
p.join()
```

Figure 4.2. `multiprocessing.Queue` example

```
def worker(data):
    return compute(data)

data = [1, 2, 3...]
pool = Pool(processes=4)
result = pool.map(worker, data)
```

Figure 4.3. `multiprocessing.Pool` example

4.2 Technology

In this section, technologies used in triResolve that will be mentioned throughout this chapter are briefly described.

4.2.1 Django

Django is a Python web development framework [15]. It implements a version of the MVC (Model-View-Controller) pattern, which decouples request routing, data access, and presentation. Django's model layer allows the programmer to retrieve and modify entities in an SQL database through Python code, without writing SQL.

4.2.2 MySQL

MySQL is an open source relational database system [26]. It is used by TriOptima as the database backend for Django.

4.2.3 Cassandra

Cassandra is a column-oriented *NoSQL* database [18, p. 1-9]. It features dynamic schemas, meaning that columns can be added dynamically to a schema as needed, and that the number of columns may vary from row to row. Cassandra is designed to have no single point of failure, and uses a number of nodes in a peer-to-peer structure. This design is employed in order to ensure high availability, with data replicated across the nodes.

4.3 Trade files and datasets

As mentioned briefly in section 1.2, users of the triResolve service upload *trade files*, which contain one or several datasets with rows of trade data such as party id, counterparty id, trade id, notional, and so on. An example of a trade dataset (with some columns omitted) can be seen in figure 4.4.

Party ID	CP ID	Trade ID	Product class	Trade curr	Notional
ABC2	QRS	ddb9c4142205735	Energy - NatGas - Forward	EUR	545940.0
ABC1	QRS	8917cefe8490715	Commodity - Swap	EUR	153438.0
ABC1	KTH1	6fc6ed1474ce42d	Commodity - Swap	EUR	99024.0
ABC2	KTH2	5489cdaab940105	Energy - NatGas - Forward	EUR	286740.0
ABC2	KTH1	119c2d2ec18027b	Energy - NatGas - Forward	EUR	191340.0
ABC1	TTT	556914ab391afb7	Energy - NatGas - Forward	EUR	196560.0
ABC2	KTH2	e6462f8b5f990d6	Commodity - Swap	EUR	105492.0
ABC1	KTH2	a8825933aaba257	Energy - NatGas - Forward	EUR	1269000.0

Figure 4.4. A simplified example of a trade dataset uploaded by the users of triResolve.

4.4 File formats

Different customers may have different ways of formatting their datasets, with different names for headers, varying column orders, extra fields, and special rules. In order to convert these into a standard format that make it possible to use the files in the same contexts, a file format specifying how the dataset in question should be processed is used. The format contains a set of *filters* which should be applied to each row of the dataset. The different filter configurations may affect how parallelizable the processing of the dataset is.

4.5 Verification results

The result of the dataset processing is called a *verification result*, and consists of one row per trade, with correctly modified values, in a Cassandra schema. In addition, a row in the MySQL database consisting of metadata relating to the result as a whole is created. This metadata includes result owner, number of rows, time metrics, and so on.

Note: The verification results are not to be confused with the results of this thesis. They are part of the problem this thesis aims to solve.

4.6 Transformation with constraints

4.6.1 Filters

All filters used to transform a dataset into a verification result are outlined below.

- **Header detection** – There may be a number of initial lines in the dataset which do not contain the header (which specifies the column names). The header detection filter checks if a row is the header, and if it is it saves the column names and corresponding indices for use in subsequent rows. If the row is not the header or the header has already been detected (for example if another header row is encountered in the middle of the dataset), this filter terminates without any effect and the rest of the filters are applied. This filter is included in all file formats.

- **Mapping** – Maps a value from a column in the dataset to a specified output column in the verification result. There is usually a mapping for each of the columns in the input dataset, and the Mapping filter is therefore one of the most common filters. The mappings may have small extra tuning attached to them, such as specifying a date format or extracting only part of the text using regex. One of these extra tunings is attached to the trade id column, and is called *Make unique*. This tuning keeps track of all trade id:s that have been encountered so far, and, if it finds a duplicate, adds a suffix to it in order to ensure that all trade id:s are unique.

- **Dataset translation** – A dataset translation is similar to a mapping, but uses specified columns in an external dataset to map input columns to output columns.

- **Dataset information** – Extracts information about the dataset, such as the name or owner.

- **Tradefile information** – Similar to the dataset information filter, except that it extracts information about the trade file that contains the dataset.

- **Null translation** – In some datasets, other values than `NULL` are used to convey the absence of a value. This filter allows the user to specify which other values should be interpreted as `NULL`.

- **Relation currency** – If the currency that is supposed to be used in a relation (a party and a counterparty) is stored in the database and should be mapped to an output column, this filter retrieves this information.

- **Global variable** – A global variable filter writes a value to a variable that is accessible by subsequent filters on the same row, and by all filters on the rest of the rows in the data set. A global variable can be written several times throughout the processing of a dataset.

- **State variable** – A state variable is similar to a global variable, but is always written to before all other processing of the dataset begins.

- **Temporary variable** – Similar to the other variables, except for the fact that it is only accessible during processing of the row where it was written. When the processing of the row is finished, the variable is cleared.

- **Conditional block** – A conditional block works like the programming construct `if`. It performs a specified filter (which may also be a conditional block) only if a certain condition is fulfilled. Most commonly, the condition takes the form '`field = value`', but may also involve more complex expressions in the form of a subset of Python.

- **Logger** – A logger filter simply logs a given value. Can for instance be used when a user wants to know whenever a conditional block has been entered.

- **Skip row** – Ignores the current row when processing. Usually used in a conditional block.

- **Stop processing** – Stops processing the dataset, ignoring all subsequent rows. Can be used as a subfilter in the Conditional block filter when the footer of the dataset contains information that should not be interpreted as a trade.

- **Third party automapper** – When a customer has uploaded a trade file on behalf of another customer, this filter extracts the information needed to make sure that the data is loaded for the correct customer.
- **Set value** – Simply sets the value of the output column to the value that is entered.
- **RegExp extract** – Extracts text from a column using regex, and writes matching groups to other columns.
- **RegExp replace** – Replaces column text matching some regex with a specified value.

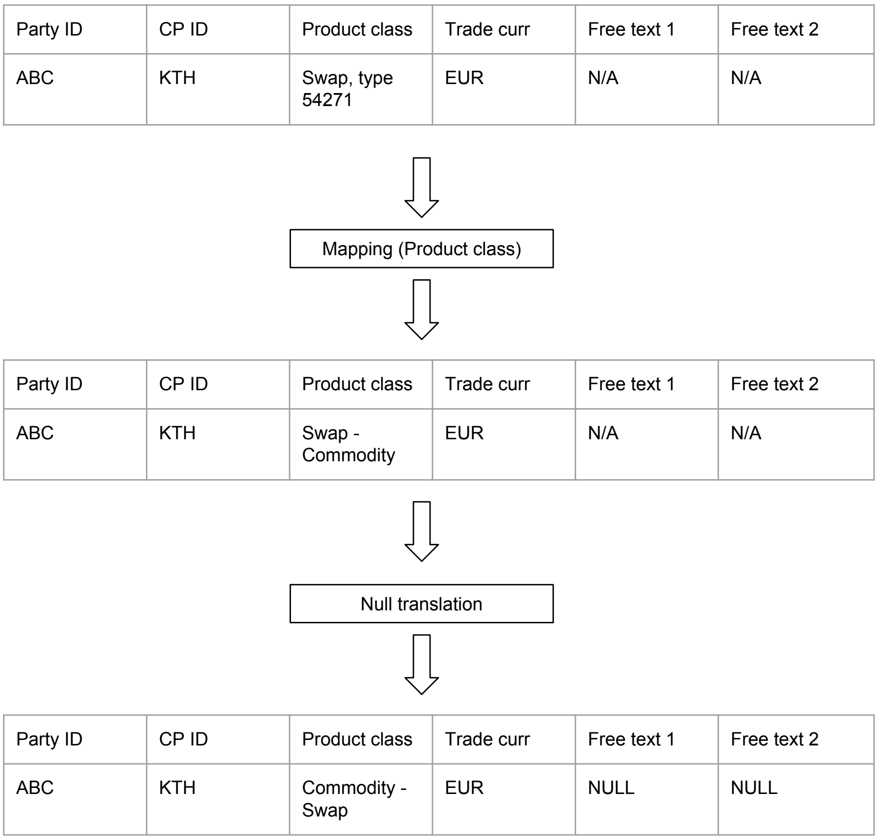
An example of how filter application and dataset row transformation work can be found in figure 4.5.

4.7 Program overview

The general flow of the original, sequential, dataset processing program is the following:

The unprocessed dataset has the rows stored in a Cassandra database, and some metadata and methods stored in a Django object backed by a MySQL database. The file format corresponding to the dataset is looked up, and all of the filters it contains are added to a pipeline that will process the dataset. An empty verification result is then created in both Cassandra and MySQL, containing the row data and result metadata with metrics, respectively. The metrics include processing time, number of trades, timestamp, and similar data. The rows in the dataset are then processed one by one, applying all filters to each row. As soon as a row has finished processing, it is written to the verification result in Cassandra. During this process, the row mappings used in the *Mapping* filter are fetched from the MySQL database, resulting in some I/O waiting time. To mitigate this, the mappings are cached in memory for faster access. After the processing has finished, the result metadata and metrics are saved in the MySQL database.

A simplified overview of the sequential program can be found in figure 4.6.



Party ID	CP ID	Product class	Trade curr	Free text 1	Free text 2
ABC	KTH	Swap, type 54271	EUR	N/A	N/A

Mapping (Product class)

Party ID	CP ID	Product class	Trade curr	Free text 1	Free text 2
ABC	KTH	Swap - Commodity	EUR	N/A	N/A

Null translation

Party ID	CP ID	Product class	Trade curr	Free text 1	Free text 2
ABC	KTH	Commodity - Swap	EUR	NULL	NULL

Figure 4.5. A simplified example of how filter application and dataset transformation works.

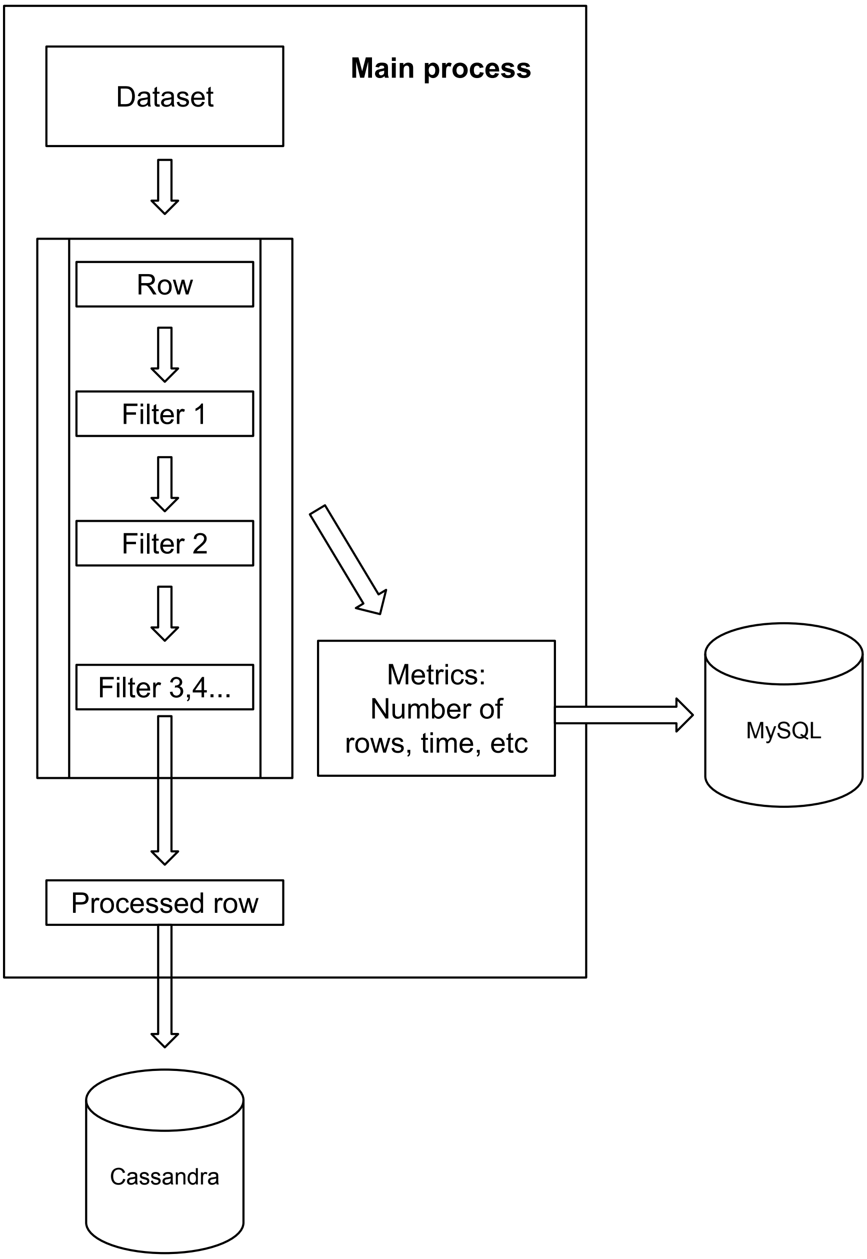


Figure 4.6. Sequential program overview.

Chapter 5

Implementation

5.1 Analysis of filter parallelizability

Since the filters specify what the processing program should do to each row in a dataset, "row by row" or possibly chunks of rows is a suitable granularity when implementing the parallelization of the program. Consequently, the filters of a file format are the prime candidates for parallelization analysis. The analysis made is similar to the methodology used to identify the span in the work-span model described in section 3.3.4. When applying the model to the problem of analyzing filters, a task is the processing of one row. In order to find the tasks that need to be completed before other tasks, the filters that result in state that is accessed by subsequent rows or otherwise affect the total processing of the dataset need to be identified.

Examining the filters, it is apparent that *Dataset translation*, *Null translation*, *Relation currency*, *Third party automapper*, *Set value*, *RegExp extract*, and *RegExp replace* only operate on the current dataset row, with no side effects. This means that they produce no state changes that affect subsequent rows, which means that they do not affect the parallelizability of a dataset.

Additionally, *Dataset information*, *Tradefile information*, *Temporary variable*, *Logger*, and *Skip row* perform operations that either pull information from resources that are available to all rows, or produce an effect that does not affect any other rows. The *Conditional block* filter only produces effects according to its subfilters (a set of the filters already mentioned), and does not affect parallelization by itself.

Hence, the filters that can affect the parallelization of a dataset are:

- *Mapping*, since the trade id mapping may need to keep track of state that can be accessed in subsequent rows in order to make all id:s unique.

- *Header detection*, since all rows beneath the (first) header row depend on the column names for mappings and other values.

- *Global variable*, since the variable may be written and accessed by any subsequent rows. Each rewrite of the variable needs to happen before the next rewrite, in the original, sequential order if the verification result is to be correct.

- *State variable*, for the same reasons as Global variable.

- *Stop processing*, if one thread sees a conditional fulfilled and stops processing, it is possible for another thread to keep processing rows that are intended to be ignored, thereby violating the constraints.

5.2 Code inspection

After an initial code and file format inspection, the following conclusions where made:

- The *Header detection* filter is effectively performed only once, as it is ignored for all rows after the one where the header was found.

- The filters *Global variable* and *State variable* make the processing of every row depend on the previous, as the writing of the variables may happen on each row.

- The process of making an ID unique could possibly be broken out to a post processing step.

- All file formats contain *Header detection*, and many contain the make unique feature of the trade ID mapping.

- There are many file formats that do not have either *Global variable*, *State variable* or *Stop processing* among their filters.

The conclusions above indicate that header detection may be done before creating the parallel processes, sending this data to each process when they are created. If the process of producing unique ID:s is then done as a post processing step, the following task DAGs illustrate how the dependencies when processing different file formats appear: In figure 5.1, the task DAG for a file format without a Global variable or State variable filter is illustrated. In figure 5.1, the task DAG for a file format containing a Global variable is illustrated. Since the span is equal to the work in the file formats containing Global variables or State variables, parallelization of datasets with these formats will result in no speedup according to the work-span model (as $T_1 \leq T_\infty \Rightarrow S \leq 1$). File formats containing *Stop processing* make it unfeasible to produce correct verification results when parallelizing. Determination of whether the result is correct is non-viable if any rows are processed in different processes (as rows that should not be included in the result may be included anyway).

5.3 Filter families

With the help of the findings from the previous sections, families of filters with different characteristics can be identified.

- **Embarrassingly parallel filters** – The filters that do not affect parallelization in any way are: *Dataset translation*, *Null translation*, *Relation currency*, *Third party automapper*, *Set value*, *RegExp extract*, *RegExp replace*, *Dataset information*, *Tradefile information*, *Temporary variable*, *Logger*, *Skip row*, and *Conditional block.* In addition *Mapping* is included among these filters if the make unique feature is disabled.
- **Overhead filters** – Filters that introduce parallelization overhead are: *Mapping* (if the make unique feature is enabled) and *Header detection.*
- **Inherently serial filters** – The filters that enforce serial execution of the entire transformation are: *Global variable*, *State variable*, and *Stop processing.*

5.4 File format families

In addition to the filter families, the fact that the *Header detection* filter is present in all file formats makes it possible to identify the following file format families relevant to this thesis:

- **Embarrassingly parallel file formats** – File formats that with the exception of *Header detection* contain only embarrassingly parallel filters.
- **Extra overhead file formats** – Formats that in addition to *Header detection* and a number of embarrassingly parallel filters also contain *Mapping* with the make unique filter enabled.
- **Inherently serial file formats** – Formats that contain any of the inherently serial filters.

5.5 Parallelization

In accordance with section 2.5, the Python `multiprocessing` module was used to implement the parallelization of the program. Additionally, measures where taken to send as little data as possible between processes and to avoid introducing excessive complexity to the codebase. The `multiprocessing.Queue` facility was chosen for communication between processes due to its noted performance and built-in synchronization [24].

Before deciding to use the parallelized version of program, the list of filters in a file format is examined for *Global variable*, *State variable*, or *Stop processing.* If any of these are found, the program falls back to its sequential version. Otherwise, the program carries on in accordance with figure 5.1. First, before creating any additional processes, the Header detection filter is applied row by row until it produces a result (commonly after a few rows). Next, a (tunable) number of processes, as well as two queues are created. A number of row spans, or chunks, are then created by splitting the rows beneath the header row into equally sized partitions. The first queue is used to transfer the data needed to process a chunk of the dataset, including the header data, the row span, and the result metadata. In order to avoid errors and sending large objects between processes, only the primary key used to retrieve the result metadata object from the MySQL database is sent to the processes. After this, the processes can independently retrieve the data. The second queue is used for sending the partial metrics objects for each chunk, and for indicating if a process is done processing its data or if it encountered an error. Since all other results are written to the Cassandra database, this is the only information that needs to be sent to the main process. The queues can be denoted the 'chunk queue' and the 'message queue', respectively.

In each of the created processes, the rows in the chunk are retrieved from the Cassandra database and a new object containing metrics for the chunk is created. The chunk is then processed as in the sequential program, applying all filters to each row. The metrics object is updated during the processing, as in the original program. If the chunk was processed correctly, the metrics object is put on the message queue. Otherwise, if an exception occurs, an error message is put on the queue instead. When all data in a process has finished processing, a message indicating that the process has finished its work is put on the message queue.

The main process continuously polls the message queue, and merges the partial metrics objects as they are polled from the queue. If an error message is encountered, an exception is raised on the main thread, mimicking the behavior of the original sequential program. It also increments a counter whenever a done message is received from a process. When the counter is equal to the number of subprocesses, the main process stops waiting for messages, and progresses with the post processing step of making the trade ID:s unique. Finally, the main process saves the result object with the corresponding merged metrics to the MySQL database. At this point, the program has produced a finished verification result.

A simplified overview of the parallel program can be found in figure 5.3.

5.6 Sources of overhead

During the implementation of the parallel version of the program, the following possible sources of parallelization overhead where identified:

- The `multiprocessing` module, where creating processes and transferring data between processes is costly.

- Less effective caching. Since the mappings cache is local to each subprocess, caches are built up individually. This results in fewer cache hits than in the sequential program, and more total work looking up values in the MySQL database.

- The process of making trade ID:s unique is added as an extra step after the main data processing pipeline.
- Because they lack built-in support for multiprocessing, the Python connections between both MySQL and Cassandra need to be restarted in the startup of each subprocess.

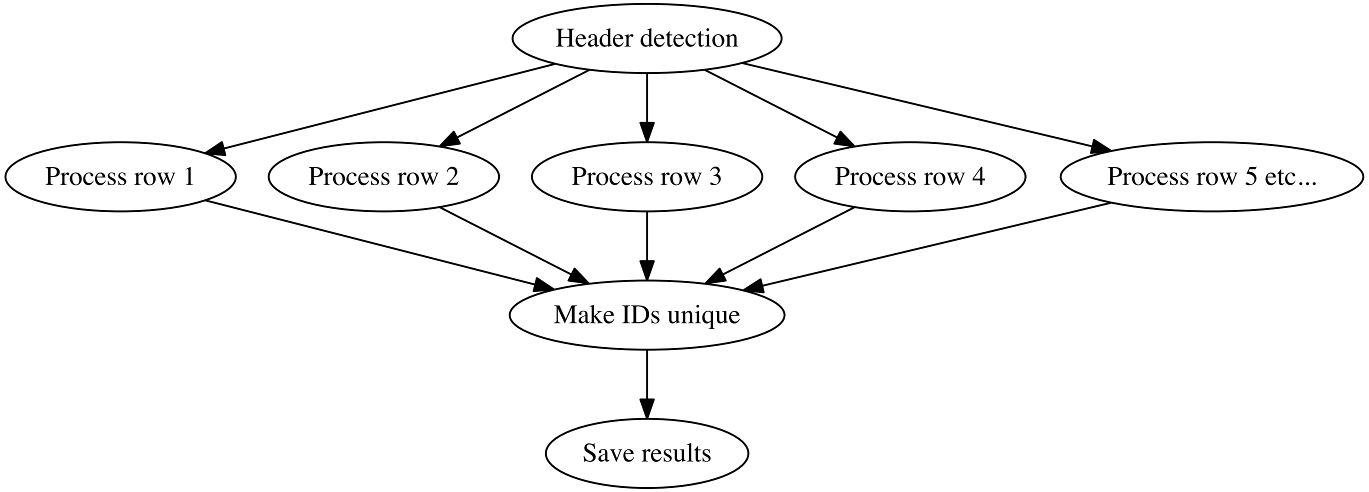


Figure 5.1. An example of a task DAG for a file format that does not contain global variable or state variables. Header detections needs to be performed up front, and making trade IDs unique needs to be performed in a post processing step. The processing of each row does not depend on each other.

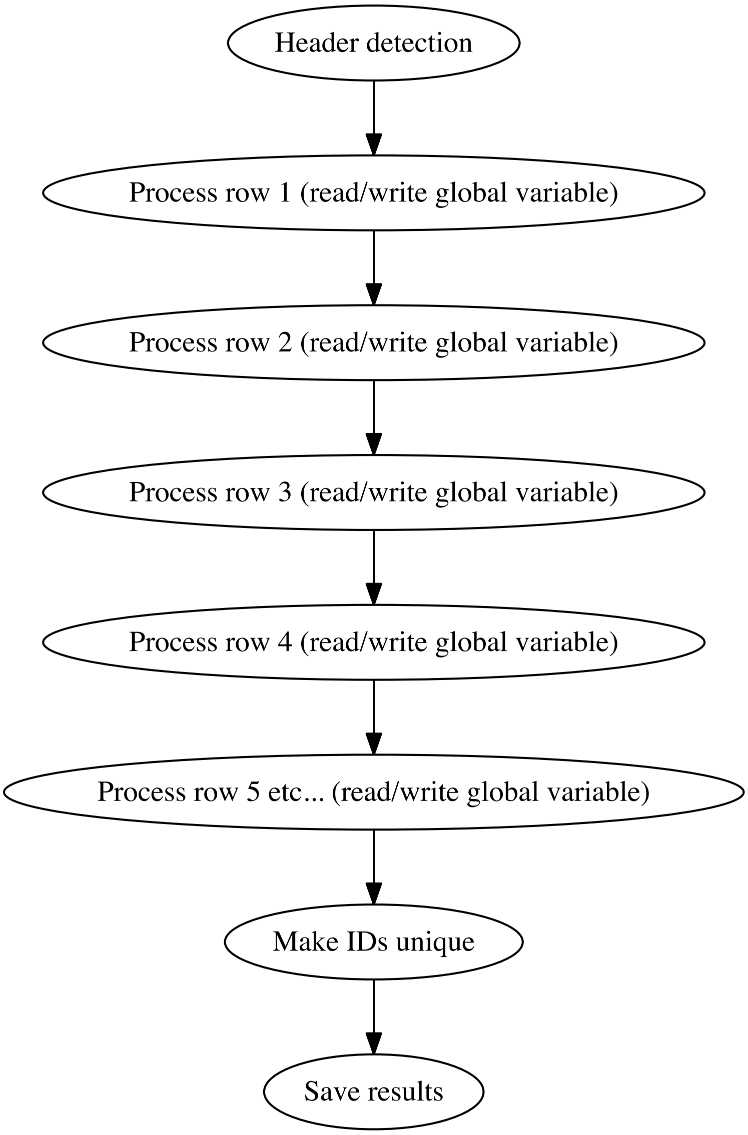

Header detection
Process row 1 (read/write global variable)
Process row 2 (read/write global variable)
Process row 3 (read/write global variable)
Process row 4 (read/write global variable)
Process row 5 etc... (read/write global variable)
Make IDs unique
Save results

Figure 5.2. An example of a task DAG for a file format that contains global or state variables. Since each row may read and write the global (or state) variable, every task depends on the previous task.

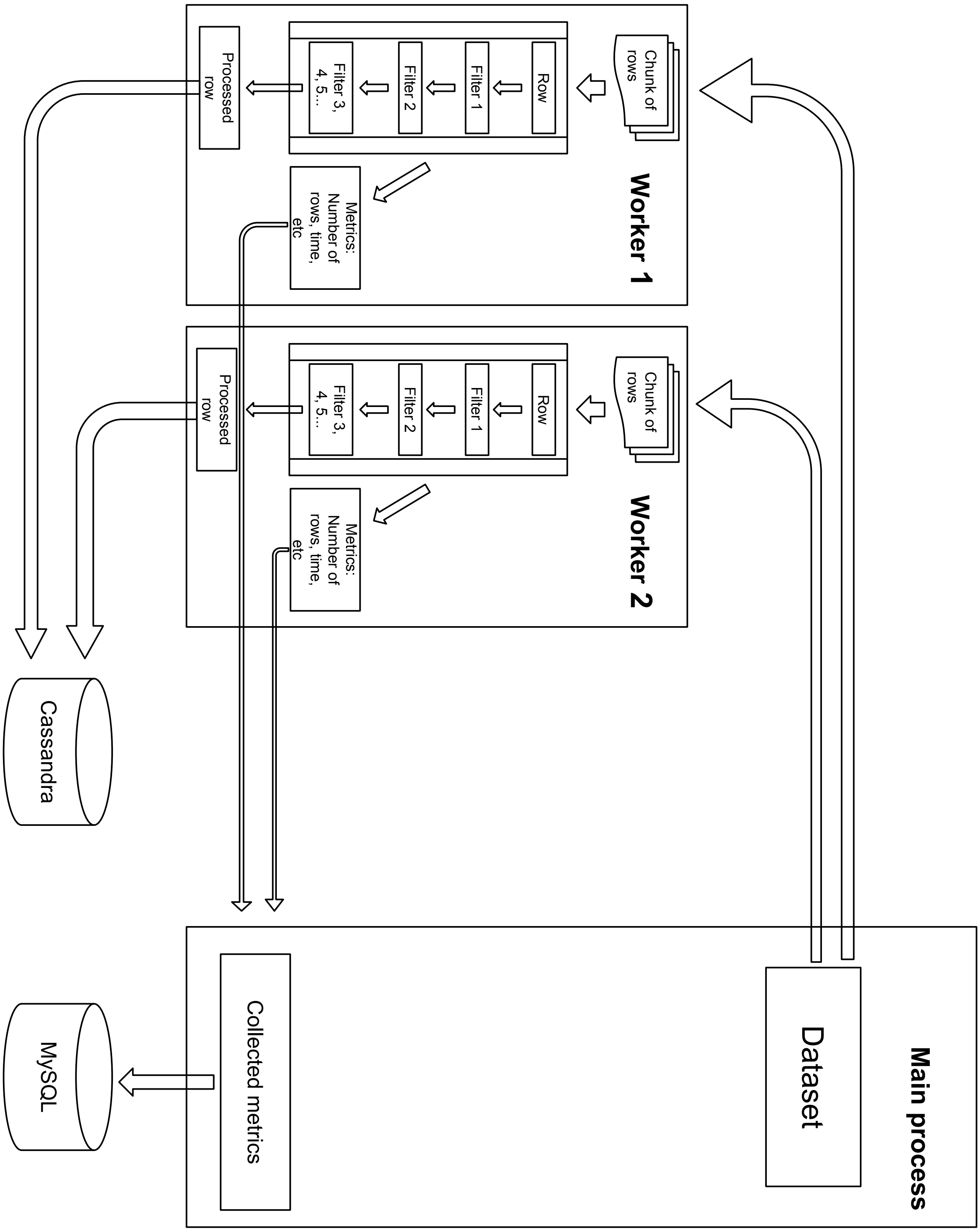


Figure 5.3. Parallel program overview.

Chapter 6

Results

6.1 Transformation benchmarks

Dataset	Dataset rows	Columns	Trade count	Workers	Real (s)	User (s)	System (s)	Memory usage (MB)
isda-first-last-same.csv	27877	46	27876	S	62.85	62.83	0.65	371
isda-first-last-same.csv	27877	46	27876	1	68.77	66.25	1.86	472
isda-first-last-same.csv	27877	46	27876	2	38.58	53.19	1.66	627
isda-first-last-same.csv	27877	46	27876	3	29.63	75.34	2.45	1121
isda-first-last-same.csv	27877	46	27876	4	26.39	83.54	2.85	1444
isda-first-last-same.csv	27877	46	27876	5	26.41	102.08	3.42	1772
isda-first-last-same.csv	27877	46	27876	6	26.66	121.44	3.88	2106
non-coll-duplicates.csv	82	46	81	S	0.67	1.77	0.21	110
non-coll-duplicates.csv	82	46	81	1	1.53	1.94	0.21	204
non-coll-duplicates.csv	82	46	81	2	1.48	2.28	0.29	300
non-coll-duplicates.csv	82	46	81	3	1.53	2.71	0.38	396
non-coll-duplicates.csv	82	46	81	4	1.54	2.98	0.42	491
non-coll-duplicates.csv	82	46	81	5	1.62	3.34	0.50	586
non-coll-duplicates.csv	82	46	81	6	1.59	3.99	0.59	681

Figure 6.1. Initial results for single dataset, on a Macbook Pro with 4 physical and 8 logical cores.

Chapter 7

Discussion

A discussion.

Bibliography

[1] *16.2. threading — Higher-level threading interface — Python 2.7.11 documentation.* URL: https://docs.python.org/2/library/threading.html (visited on 02/16/2016).

[2] *16.6. multiprocessing — Process-based "threading" interface — Python 2.7.11 documentation.* URL: https://docs.python.org/2/library/multiprocessing.html#all-platforms (visited on 02/01/2016).

[3] M. Ahmad, K. Lakshminarasimhan, and O. Khan. "Efficient parallelization of path planning workload on single-chip shared-memory multicores". In: *2015 IEEE High Performance Extreme Computing Conference (HPEC).* 2015 IEEE High Performance Extreme Computing Conference (HPEC). Sept. 2015, pp. 1–6. DOI: 10.1109/HPEC.2015.7322455.

[4] G.M. Amdahl. "Computer Architecture and Amdahl's Law". In: *Computer* 46.12 (Dec. 2013), pp. 38–46. ISSN: 0018-9162. DOI: 10.1109/MC.2013.418.

[5] Gergö Barany. "Python Interpreter Performance Deconstructed". In: *Proceedings of the Workshop on Dynamic Languages and Applications.* Dyla'14. New York, NY, USA: ACM, 2014, 5:1–5:9. ISBN: 978-1-4503-2916-3. DOI: 10.1145/2617548.2617552. URL: http://doi.acm.org/10.1145/2617548.2617552 (visited on 01/27/2016).

[6] David Beazley. "An Introduction to Python Concurrency". 15:07:45 UTC. URL: http://www.slideshare.net/dabeaz/an-introduction-to-python-concurrency?next_slideshow=1 (visited on 02/01/2016).

[7] S. Binet et al. "Harnessing multicores: Strategies and implementations in ATLAS". In: *Journal of Physics: Conference Series* 219.4 (2010), p. 042002. ISSN: 1742-6596. DOI: 10.1088/1742-6596/219/4/042002. URL: http://stacks.iop.org/1742-6596/219/i=4/a=042002 (visited on 01/29/2016).

[8] Xing Cai, Hans Petter Langtangen, and Halvard Moe. "On the Performance of the Python Programming Language for Serial and Parallel Scientific Computations". In: *Scientific Programming* 13.1 (2005), pp. 31–56. ISSN: 1058-9244. DOI: 10.1155/2005/619804. URL: http://www.hindawi.com/journals/sp/2005/619804/abs/ (visited on 01/21/2016).

[9] Hao Che and Minh Nguyen. "Amdahl's law for multithreaded multicore processors". In: *Journal of Parallel and Distributed Computing* 74.10 (Oct. 2014), pp. 3056–3069. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2014.06.012. URL: http://www.sciencedirect.com/science/article/pii/S0743731514001142 (visited on 01/22/2016).

[10] Jonathan Chow, Nasser Giacaman, and Oliver Sinnen. "Pipeline pattern in an object-oriented, task-parallel environment". In: *Concurrency and Computation: Practice and Experience* 27.5 (Apr. 10, 2015), pp. 1273–1291. ISSN: 1532-0634. DOI: 10.1002/cpe.3305. URL: http://onlinelibrary.wiley.com.focus.lib.kth.se/doi/10.1002/cpe.3305/abstract (visited on 02/22/2016).

[11] Rune Møllegaard Friborg, John Markus Bjørndalen, and Brian Vinter. "Three Unique Implementations of Processes for PyCSP." In: *CPA*. 2009, pp. 277–292. URL: http://www.researchgate.net/profile/Brian_Vinter/publication/221004402_Three_Unique_Implementations_of_Processes_for_PyCSP/links/0046352c13f97306f5000000.pdf (visited on 01/29/2016).

[12] *Glossary — Python 2.7.11 documentation.* URL: https://docs.python.org/2/glossary.html#term-global-interpreter-lock (visited on 02/16/2016).

[13] John L. Gustafson. "Reevaluating Amdahl's Law". In: *Commun. ACM* 31.5 (May 1988), pp. 532–533. ISSN: 0001-0782. DOI: 10.1145/42411.42415. URL: http://doi.acm.org/10.1145/42411.42415 (visited on 01/22/2016).

[14] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint.* Morgan Kaufmann, June 25, 2012. 536 pp. ISBN: 978-0-12-397795-3. URL: http://proquest.safaribooksonline.com.focus.lib.kth.se/book/programming/9780123973375 (visited on 01/21/2016).

[15] Adrian Holovaty and Jacob Kaplan-Moss. *Chapter 1: Introduction to Django.* The Django Book. URL: http://www.djangobook.com/en/2.0/chapter01.html (visited on 04/05/2016).

[16] Paul Krill. *Python scales new heights in language popularity.* InfoWorld. 2015-12-08T03:00-05:00. URL: http://www.infoworld.com/article/3012442/application-development/python-scales-new-heights-in-language-popularity.html (visited on 02/12/2016).

[17] Michael McCool, James Reinders, and Arch Robison. *Structured Parallel Programming: Patterns for Efficient Computation.* 1 edition. Amsterdam; Boston: Morgan Kaufmann, July 9, 2012. 432 pp. ISBN: 978-0-12-415993-8.

[18] Vivek Mishra. *Beginning Apache Cassandra Development.* Apress, Dec. 12, 2014. 235 pp. ISBN: 978-1-4842-0142-8.

[19] G.E. Moore. "Cramming More Components Onto Integrated Circuits". In: *Proceedings of the IEEE* 86.1 (Jan. 1998), pp. 82–85. ISSN: 0018-9219. DOI: 10.1109/JPROC.1998.658762.

[20] Jesse Noller and Richard Oudkerk. *PEP 0371*. Python.org. URL: `https://www.python.org/dev/peps/pep-0371/` (visited on 01/29/2016).

[21] Jan Palach. *Parallel Programming with Python*. Packt Publishing, Apr. 24, 2014. 124 pp. ISBN: 978-1-78328-839-7.

[22] *PythonImplementations - Python Wiki*. URL: `https://wiki.python.org/moin/PythonImplementations` (visited on 04/08/2016).

[23] Sergio J. Rey et al. “Parallel optimal choropleth map classification in PySAL”. In: *International Journal of Geographical Information Science* 27.5 (May 1, 2013), pp. 1023–1039. ISSN: 1365-8816. DOI: `10.1080/13658816.2012.752094`. URL: `http://www-tandfonline-com.focus.lib.kth.se/doi/abs/10.1080/13658816.2012.752094` (visited on 02/04/2016).

[24] Navtej Singh, Lisa-Marie Browne, and Ray Butler. “Parallel astronomical data processing with Python: Recipes for multicore machines”. In: *Astronomy and Computing* 2 (Aug. 2013), pp. 1–10. ISSN: 2213-1337. DOI: `10.1016/j.ascom.2013.04.002`. URL: `http://www.sciencedirect.com/science/article/pii/S2213133713000085` (visited on 01/27/2016).

[25] Brett Slatkin. *Effective Python: 59 Specific Ways to Write Better Python*. 1 edition. Addison-Wesley Professional, Mar. 8, 2015. 256 pp. ISBN: 978-0-13-403428-7.

[26] *What is MySQL?* URL: `http://dev.mysql.com/doc/refman/5.1/en/what-is-mysql.html` (visited on 04/05/2016).

[27] L. Yavits, A. Morad, and R. Ginosar. “The effect of communication and synchronization on Amdahl’s law in multicore systems”. In: *Parallel Computing* 40.1 (Jan. 2014), pp. 1–16. ISSN: 0167-8191. DOI: `10.1016/j.parco.2013.11.001`. URL: `http://www.sciencedirect.com/science/article/pii/S0167819113001324` (visited on 01/22/2016).