KTH Computer Science
and Communication

Parallelization of dataset transformation with processing order constraints in Python

DEXTER GRAMFORS

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

Abstract

Financial data is often represented with rows of values, contained in a dataset. This data needs to be transformed into a common format in order for comparison and matching to be made, which can take a long time for larger datasets. The main goal of this master's thesis is speeding up these transformation through parallelization using Python multiprocessing. The datasets in question consist of several rows representing trades, and are transformed into a common format using rules known as filters. In order to devise a parallelization strategy, the filters were analyzed in order to find ordering constraints, and the Python profiler cProfile was used to find bottlenecks and potential parallelization points. This analysis resulted in the use of a task-based approach for the implementation, in which the transformation was divided into an initial sequential pre-processing step, a parallel step where chunks of several trade rows were distributed among workers, and a sequential post processing step.

The implementation was tested by transforming four datasets of differing sizes using up to 16 workers, and execution time and memory consumption was measured. The results for the tiny, small, medium, and large datasets showed a speedup of 0.5, 2.1, 3.8, and 4.81. They also showed linearly increasing memory consumption for all datasets. The test transformations were also profiled in order to understand the parallel program's behaviour for the different datasets. The experiments gave way to the conclusion that dataset size heavily influence the speedup, partly because of the fact that the sequential parts become less significant. In addition, the large memory increase for larger amount of workers is noted as a major downside of multiprocessing when using caching mechanisms, as data is duplicated instead of shared.

This thesis shows that it is possible to speed up the dataset transformations using chunks of rows as tasks, though the speedup is relatively low.

Referat

Parallelisering av datamängdstransformation med ordningsbegränsningar i Python

Finansiell data representeras ofta med rader av värden, samlade i en datamängd. Denna data måste transformeras till ett standardformat för att möjliggöra jämförelser och matchning. Detta kan ta lång tid för stora datamängder. Huvudmålet för detta examensarbete är att snabba upp dessa transformationer genom parallellisering med hjälp av Python-modulen multiprocessing. Datamängderna omvandlas med hjälp av regler, kallade filter. Dessa filter analyserades för att identifiera begränsningar på ordningen i vilken datamängden kan behandlas, och därigenom finna en parallelliseringsstrategi. Python-profileraren cProfile användes även för att hitta potentiella parallelliseringspunkter i koden. Denna analys resulterade i användandet av ett "task"-baserat tillvägagångssätt, där transformationen delades in i ett sekventiellt pre-processingsteg, ett parallelt steg där grupper av rader distribuerades ut bland arbetarprocesser, och ett sekventiellt post-processingsteg.

Implementationen testades genom transformation av fyra datamängder av olika storlekar, med upp till 16 arbetarprocesser. Resultaten för de fyra datamängderna var en speedup på 0.5, 2.1, 3.8 respektive 4.81. En linjär ökning i minnesanvändning uppvisades även. Experimenten resulterade i slutsatsen att datamängdens storlek var en betydande faktor i hur mycket speedup som uppvisades, delvis på grund av faktumet att de sekventiella delarna tar upp en mindre del av programmet. Den stora minnesåtgången noterades som en nackdel med att använda multiprocessing i kombination med cachning, på grund av duplicerad data.

Detta examensarbete visar att det är möjligt att snabba upp datamängdstransformation genom att använda radgrupper som tasks, även om en relativt låg speedup uppvisades.

Contents

List of Figures

List of Tables

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; "wall clock" time.

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

In this chapter, an introduction to the parallel computing problem domain and the specific problem of dataset transformation are described in order to give the reader an initial view of what this thesis entails.

1.1 Dataset transformation

In financial applications concerning trading, it is common for customers to upload datasets containing several rows describing trades, which may be in different formats. One such application is triResolve, an application maintained by TriOptima, where this thesis is conducted. In triResolve, customers resolve trade disputes in the OTC (Over-The-Counter) derivatives market, which may arise due to for example differences in valuation methods. The triResolve service is built in Python, to be able to resolve the trade disputes, customers upload the previously mentioned datasets to the service.

The datasets 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. Out of the possible techniques for optimizing the transformation code, this thesis will focus on parallelization. Since Python is the language used in triResolve, the parallelization of the existing program will be implemented using parallel programming tools available in the language.

When parallelizing a program, the workload is divided amongst multiple cores of a system, which execute the program in parallel. For the dataset transformation problem, this means dividing the dataset, conceivably into chunks of rows, and performing the transformation of each of these chunks on separate cores.

This thesis presents the challenges associated with this parallelization problem, and how to solve them.

The datasets are associated with a file format. The format specifies a set of rules, known as filters, which at times enforce implicit constraints on the processing order in the file when performing the transformation. This thesis aims to identify these

constraints, which may affect how parallelizable a dataset is, and find a suitable parallelization strategy. Another aim is to identify the impact dataset size has on any potential speedup. In addition, how using the Python `multiprocessing` module and its process-over-thread with message passing approach affects implementation and performance will be investigated.

1.2 Parallel computing

In this thesis, a task-based approach is used to parallelize the dataset transformation [12]. A task is a single unit of computation, often represented as a function and run on different threads or processes. Tasks are executed by the operating system's scheduler, and can be executed on different cores. When tasks are scheduled on different cores, they are able to run at the same time, resulting in parallelism and possible speedup of a program. If there are more tasks than cores, the tasks are scheduled using time-slicing, where tasks share cores.

1.3 Hardware

The parallelization in this thesis is conducted on a shared memory computer. In this setup, several computing units (cores) share one memory. Examples of shared memory systems are common laptops and workstations.

1.4 Motivation

The motivation of this thesis is to answer the following questions.

Given the size of a dataset and its set of filters, is it 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 Objectives

The objectives of this thesis are to:

- Analyze parallelizability of dataset file formats.

- Use a Python profiler to analyze `multiprocessing` performance for dataset transformation.
- Implement a working parallelization of the dataset transformation program, for the applicable datasets.
- Evaluate the parallel performance of transformation of different datasets by measuring execution time, speedup, and memory consumption.

1.6 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

Background

In this section, multicore architecture, shared memory programming, and Python parallel capabilities are explained in order to give the reader a foundation in these relevant areas.

2.1 Multicore architecture

2.1.1 Multicore processors

In a typical multicore processor, several cores (which are similar to regular processors) work together in the same system [19, p. 44-50]. The cores consist of several *functional units*, which are the core components that perform arithmetic calculations. These functional units are able to perform multiple calculation instructions in parallel if these are not dependent on each other. This is known as *instruction level parallelism*. In the multicore model, a hierarchical cache memory architecture is used. The small, fast cache memories closest to the functional units are called *registers*. The next caches in the hierarchy are the data and instruction caches which are attached to each core. Subsequent, higher level caches that follow these are usually an order of magnitude slower for each cache level.

2.1.2 Multicore communication

Multiple cores communicate with each other through a bus or a network [16, p. 472-476]. Since the means of communication between the cores is a finite resource, too much traffic may result in delays. As previously mentioned, the processors typically have their own cache. In order to avoid unnecessary reads from the slower main memory when a cache miss is encountered for the core's own cache, cores may read from another core 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 multicore programs should not be ignored.

2.2 Parallel shared memory programming

2.2.1 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 [26]. 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.

2.2.2 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 [26]. It allows scalability as number of cores and problem sizes increase, since more parallelism can be exploited for larger datasets [19, p. 24].

2.2.3 Task parallelism

In task parallelism, groups of tasks that are independent are run in parallel [12]. 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.

2.2.4 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 [16, 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.

2.3 Performance models for parallel speedup

2.3.1 Amdahl's law

Amdahl's law [6] 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 [16, 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.

2.3.2 Extensions of Amdahl's law

Che and Nguyen expand on Amdahl's law and adapts it to modern multicore processors [11]. 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 [29]. They find that it is important to minimize the intensity of synchronization operations even in programs that are highly parallel.

2.3.3 Gustafson's law

Gustafson's law [15] 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 [19, p. 61-62].

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

$$S = n + (1 - n) \cdot s$$

2.3.4 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 [19, 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 2.1.
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.

2.4 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 [24]. This thesis uses CPython 2.7.

2.4.1 Performance

The general performance of CPython is slower than other popular languages such as C and Java for several reasons [7]. 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

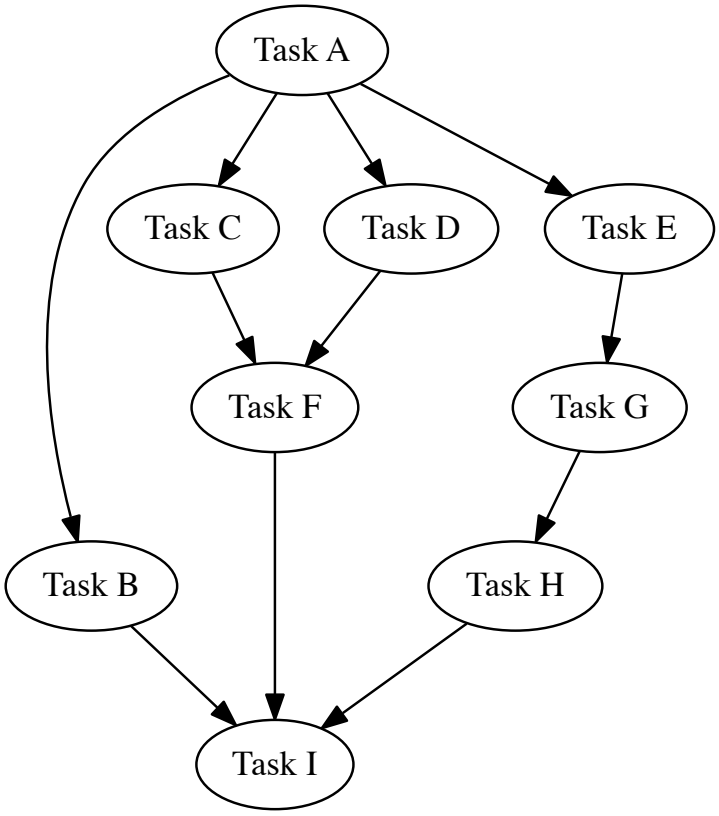

Figure 2.1. An example of a task DAG used in the work-span model. Assuming each task takes time 1 to complete, this DAG has a *work* (T_1) of 9 and a *span* (T_∞) of 5. The upper bound for parallel speedup for this dag is $\frac{9}{5} = 1.8$.

form of reference counting, and the boxing and unboxing of methods contribute to the at times poor performance.

2.4.2 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 [23]. 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 [14]. 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.

2.4.3 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 [27, p. 121-124]

2.4.4 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 [27]. 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 [27, 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 [23]:

- `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 2.2 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 2.3 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 2.4 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 2.2. `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 2.3. `multiprocessing.Queue` example

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

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

Figure 2.4. `multiprocessing.Pool` example

Chapter 3

Related work

In this section, work related to that of this thesis is summarized and discussed, in order to utilize conclusions made by others when deciding upon the method to use, and also to highlight differences between earlier works and this thesis.

3.1 Parallelization of algorithms using Python

Ahmad et al. [5] 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. [10] 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. [26] 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. [25] 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.

3.2 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 [22] 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.

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.

3.3 Comparisons of process abstractions

Friborg et al. [13] 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.

3.4 Parallelization in complex systems using Python

Binet et al. [9] 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.

3.5 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 4

Dataset Transformation

In this section, the problem of dataset transformation is thoroughly described. Initially, technology used in the transformation is described. Then, performance analysis tools used to analyse the code and its performance are given an overview. The overall problem and its different parts are then individually described, after which an overview of the transformation program is given. After this, a profiling session of the program is shown, performance models are used to calculate potential speedup, and a parallelizability analysis is conducted. Finally, the implementation of the parallelization is described.

4.1 Technology

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

4.1.1 Django

Django is a Python web development framework [17]. 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.

4.1.2 MySQL

MySQL is an open source relational database system [28]. It is used by TriOptima as the database backend for Django.

4.1.3 Cassandra

Cassandra is a column-oriented *NoSQL* database [20, 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.

4.2 Performance analysis tools

4.2.1 cProfile

A Python profiler with a relatively low overhead, which can be invoked both directly in a Python program and from the command line [3]. An example of the way cProfile is used in this thesis can be found in figure 4.1. The output of a profiling session, `ncalls` is the number of times the function was called, `tottime` is total time spent in the function (excluding subfunctions), `cumtime` is the total time spent in the function including its subfunctions, and `percall` is the quotient of `cumtime` divided by primitive calls.

```
from cProfile import Profile
from pstats import Stats


def profile(func, file_path):
    pr = Profile()
    pr.enable()
    func()
    pr.disable()
    s = open(file_path, 'w')
    sortby = 'cumulative'
    ps = Stats(pr, stream=s).sort_stats(sortby)
    ps.print_stats()
```

Figure 4.1. `cProfile` usage example. In this example, the input function `func` is profiled, and the output is printed to the file in `file_path`.

4.2.2 resource

`resource` is a Python module used for measuring resources used by a Python program [4]. It can be used for finding the user time, system time, and the maximum memory used by the process. An example of how to use `resource` can be found in figure 4.2.

```
import resource


def get_resource_usage(func):
    func()
    usage = resource.getrusage(resource.RUSAGE_SELF)
    print usage.ru_maxrss  # maximum memory usage
    print usage.ru_utime  # time in user mode
    print usage.ru_stime  # time in system mode
```

Figure 4.2. `resource` usage example. In this example, the memory usage, user time, and system time after executing `func` is printed.

4.3 Trade files and datasets

As mentioned briefly in section 1.1, 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 4.1.

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

Table 4.1. 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 Filters

The filters are implemented as Python classes, which implement the interface of `prepare`, `pre_process_record`, `process_record`, and `post_process_record`. `prepare` is called only once, before the processing of any rows has started, and may include caching values from the database in order to avoid an excessive amount of round trips to it. The records that the method names reference are rows in the dataset, represented as a Python class containing the fields in the dataset row as a Python dict. In the methods above, the filter has access to the current processing pipeline state, and the current record that is being processed. If a filter does not implement one of the listed methods, it falls back to a default, empty implementation.

After applying `pre_process_record`, `process_record`, and `post_process_record` on a row, the result is a modified row, a modified pipeline state, or both.

An example of how filter application and dataset row transformation work can be found in figure 4.7.

4.6 Verification results

The result of the dataset processing is called a *verification result*[1], and consists of one row per trade, with correctly modified values, in a Cassandra schema. After all filters have been applied to a row, it is written to the 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.

4.7 Transformation with constraints

4.7.1 Filter list

All filters used to transform a dataset into a verification result are outlined below. The code for the base Filter can be found in figure 4.3. Simplified code for *Null*

[1]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.

translation, *Global variable*, and *Regexp extract* are provided in figure 4.4, 4.5, and 4.6, in order to give the reader an understanding of how the filters are implemented.

```
class Filter(object):
    def prepare(self, config, pipeline):
        pass

    def pre_process_record(self, record, pipeline):
        pass

    def process_record(self, record, pipeline):
        pass

    def post_process_record(self, record, pipeline):
        pass
```

Figure 4.3. The base `Filter` implementation.

```
class NullTranslationFilter(Filter):
    def prepare(self, config, pipeline):
        self.null_tokens = config.null_tokens

    def process_record(self, record, pipeline):
        for field, value in record.fields.items():
            if value in self.null_tokens:
                record.fields[field] = None
```

Figure 4.4. Null translation filter implementation.

```
class GlobalVariableFilter(Filter):
    def prepare(self, config, pipeline):
        self.variable_name = config.variable_name
        pipeline.globals.init(self.variable_name)
```

Figure 4.5. Global variable filter implementation.

```
class RegexpExtractFilter(Filter):
    def prepare(self, config, pipeline):
        self.regex = config.regex
        self.field = config.field
        self.destination = config.destination
        self.destination_field = config.destination_field

    def process_record(self, record, pipeline):
        value = record.fields[self.field]
        match = self.regex.search(value)
        if match:
            if self.destination == 'record':
                record.fields[self.destination_field] = match
            elif self.destination == 'variable':
                pipeline.globals[self.destination_field] = match
```

Figure 4.6. Regexp extract filter implementation

- **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 initiates 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. The variable may then be set by the *Set value* filter.
- **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** – Sets the value of an output column or a variable to the value that is entered.

- **RegExp extract** – Extracts text from a column using regex, and writes matching groups to other columns or variables.
- **RegExp replace** – Replaces column text matching some regex with a specified value.
- **Post process** – A filter that is active in every file format. Performs final adjustments to the row in a similar manner to the mappings filters, and initially caches these mappings in order to avoid round trips to the database.

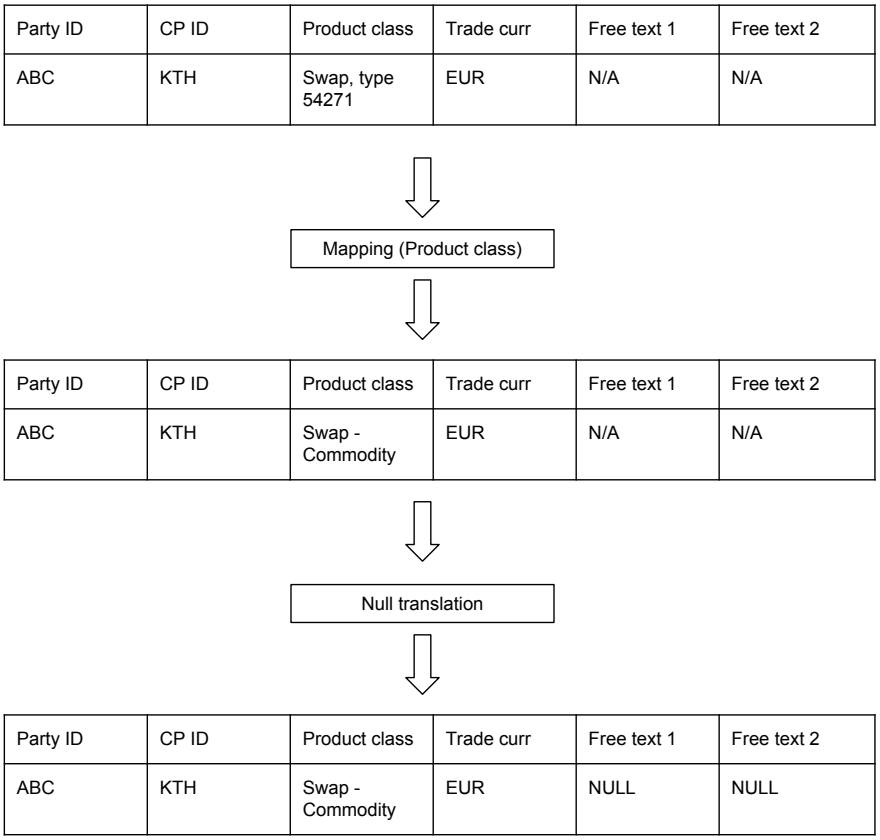

Figure 4.7. A simplified example of how filter application and dataset transformation work. The mapping filter for the product class column is applied, transforming "Swap, type 54271" to the standardized "Swap - Commodity ". In the file format used for this example, "N/A" is used to denote the absence of a value, making the null translation filter translate all columns containing "N/A" to "NULL".

4.8 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 4.8.

4.9 Sequential program profiler analysis

The result of running `cProfile` on the sequential transformation program can be found in figure 4.9. The dataset used had 27877 rows, 46 columns, and belonged to the extra overhead file format family. Function calls with very low cumulative time have been omitted.

From the profiling information above, it is clear that some function calls take significantly more time than others, and are therefore interesting targets for parallelization analysis. The `process` method is the one that launches the main pipeline that applies all filters and performs the transformation of the dataset. The fact that it takes 66.280 seconds out of 66.567 is therefore expected. Among the functions that `process` calls, `process_record`, `post_process_record`, `consume_record`, and `_prepare` are the most interesting. Other functions with relatively high cumulative time are called from these functions.

- `process_record` – In the profiling information, `process_record` appears twice, once in the file `pipeline.py` and once in the file `mappings.py`. The version in `pipeline.py` is abstract, with an implementation in each of the filters. It is evident that the implementation that is most common resides in `mappings.py` as it is the only one that shows up among the function calls that take up a significant amount of time. This is expected, as *Mapping* is the most common filter and represented in `mappings.py`. The function has a low `percall` and a high `ncalls`, indicating that the reason it takes up a large portion of the total time is the fact that it is called many times due to the

many filters and dataset rows. `process_record` takes up 35.0% of the total execution time.

- `post_process_record` – Similarly to `process_record`, `post_process_record` is an abstract method and is implemented in all filters. It also has a low `percall` and a high `ncalls`. `post_process_record` takes up 33.1% of the total time.
- `consume_record` – This method calls `_write_record`, which is the method that writes rows to Cassandra after they have been transformed. `consume_record` is called once per row and takes 8.4% of the total time, and `_write_record` is responsible for 7.9% of these.
- `_prepare` – Called once before the program starts iterating over all rows, performing setup needed to perform the transformations properly. It has a relatively high `percall` and takes up 3.3% of the total execution time.

The time spent in the functions above is 87.7% of the total time, and the majority of the rest of the code in `process` is contained in the body of the loop that iterates over and performs actions on every row. Since only `_prepare` runs before the loop, and a small portion of code is run after the loop, about 4% of the code is run outside the loop. This means that around 96% of the code is conceivably parallelizable, depending on the filters in the dataset's file format.

The following conclusions can be made from the analysis above:

- A majority of the functions responsible for most of the time consumption have low `percall` and high `ncalls`, indicating that no single function is a significant bottleneck, and that the major reason these functions take up large portions of the total time is that they are called a high number of times.
- A relatively small portion of the code is spent in functions that perform I/O, indicating that the program is CPU bound and suitable for speedup using `multiprocessing`.
- Close to all of the code in `process` is run for each row, indicating that performing the transformation of different rows in different tasks is a suitable granularity when implementing the parallelization.
- The fact that `_prepare` takes up a non-negligible part of the program and is called before the processing of each row, it may introduce extra overhead when parallelizing, since it may need to be called for each worker.

4.10 Performance model calculations

In this section, different performance calculation models are used to find a preliminary indication of possible speedup.

4.10.1 Amdahl's law

In the sequential profiling session, it is suggested that around 96% of the code is parallelizable. The computer used for testing has 8 cores, which is the value that will be used for the n value when applying Amdahl's law to the profiling session run:

$$S = \frac{1}{1 - 0.96 + \frac{0.96}{8}} = 6.25$$

This potential speedup of 6.25 does not take into account overhead associated with parallelization.

4.10.2 Gustafson's law

With Gustafson's law, the speedup is calculated as the following:

$$S = 8 + (1 - 8) * 0.04 = 7.72$$

With the more optimistic Gustafson's law, the speedup is higher. Overhead is not taken into account in the above calculation.

4.10.3 Work-span model

As mentioned in section 2.3.4, the increase in work when parallelizing a program should be kept to a minimum. In addition, the span should be kept as small as possible. In the implementation made in this thesis, both the work and the span is increased. The work is increased since the code that is run before the loop over each row has to be run once for each worker, and because the caching of column mappings is done for each worker. The span is increased because of the added post processing needed when transforming datasets from the extra overhead file format family. This suggests that parallelization cannot be utilized at its fullest, which may impact the speedup.

4.11 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 2.3.4. 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.

4.12 Code inspection

After an initial code and file format inspection, the following conclusions were 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 4.11, the task DAG for a file format without a Global variable or State variable filter is illustrated. In figure 4.10, 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).

4.13 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*.

4.14 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.

4.15 Parallelization

In accordance with section 3.5, 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 [26].

Before deciding to use the parallelized version of the program, the list of filters in a file format is examined for any of the inherently serial filters *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 4.10. 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 4.12.

4.16 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 of mappings. 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. Hardware cache may also be affected in a similar manner.
- 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.
- `_prepare` has to be called once for each of the workers, compared to only one call for the sequential program.

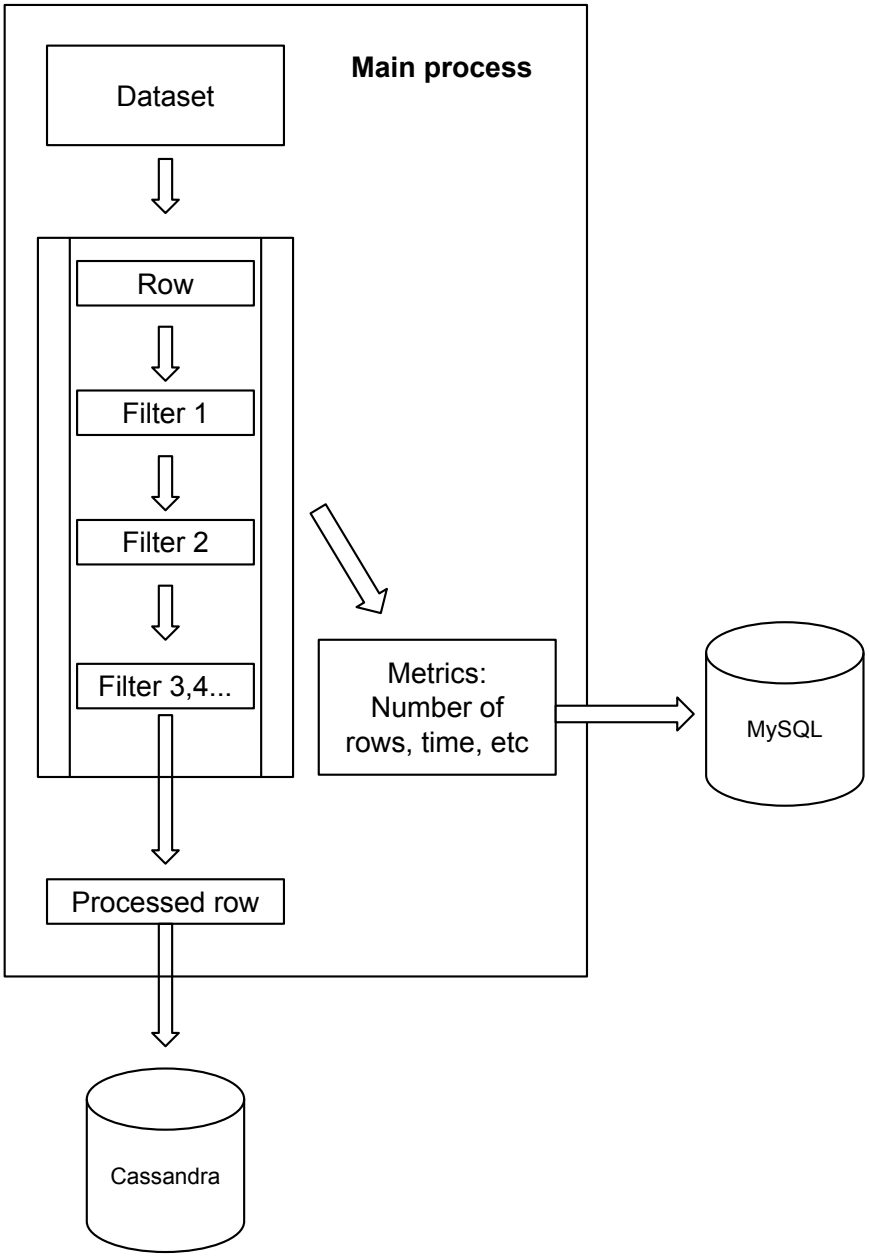

Figure 4.8. Sequential program overview.

```
         76069831 function calls (75553060 primitive calls) in 66.567 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000   66.567   66.567 importer.py:433(verify)
        1    0.000    0.000   66.552   66.552 importer.py:399(_do_verify)
        1    0.000    0.000   66.545   66.545 verifier.py:441(verify)
        1    0.000    0.000   66.522   66.522 verifier.py:626(do_verify)
        1    5.916    5.916   66.280   66.280 pipeline.py:806(process)
  1477428    0.815    0.000   23.326    0.000 pipeline.py:834(process_record)
   111504    0.129    0.000   22.036    0.000 pipeline.py:838(post_process_record)
  1365924    2.654    0.000   18.756    0.000 mappings.py:570(process_record)
    27876    1.094    0.000   10.401    0.000 post_process.py:119(post_process_record)
    27876    4.272    0.000    6.832    0.000 verification.py:188(post_process_record)
  1393800    1.068    0.000    5.789    0.000 mappings.py:555(get_value)
    27876    0.118    0.000    5.612    0.000 verifier.py:330(consume_record)
  2759725    2.789    0.000    5.541    0.000 pipeline.py:636(__getitem__)
    27876    0.190    0.000    5.282    0.000 verifier.py:268(_write_record)
  1449553    0.829    0.000    5.108    0.000 pipeline.py:655(get)
    27876    0.542    0.000    4.412    0.000 statistics.py:151(post_process_record)
   527669    0.453    0.000    4.314    0.000 translation_manager.py:68(hooked)
    27878    0.078    0.000    3.459    0.000 pipeline.py:322(info)
    28006    0.498    0.000    3.394    0.000 pipeline.py:262(add)
    55752    0.380    0.000    3.165    0.000 mappings.py:112(get_transformed)
   111504    0.148    0.000    3.128    0.000 mappings.py:281(get_transformed)
336076/84022    1.855    0.000    3.102    0.000 struct/__init__.py:15(__hash__)
    72075    0.119    0.000    2.955    0.000 dateext.py:56(parse_date)
   446016    1.056    0.000    2.921    0.000 mappings.py:69(get_transformed)
    72075    0.304    0.000    2.836    0.000 dateext.py:21(parse_date_base)
 17343845    2.539    0.000    2.646    0.000 method 'get' of 'dict' objects
    27876    1.297    0.000    2.616    0.000 row_skipper.py:27(process_record)
   143975    0.150    0.000    2.507    0.000 time.strptime
    27876    0.173    0.000    2.501    0.000 verifier.py:247(do_relation_dependent_transformations)
921603/669537    0.392    0.000    2.418    0.000 hash
   328147    0.920    0.000    2.391    0.000 db.py:63(get_any)
   143975    0.096    0.000    2.355    0.000 python2.7/_strptime.py:466(_strptime_time)
  299/294    0.012    0.000    2.312    0.008 django/db/models/query.py:972(_fetch_all)
   143975    1.246    0.000    2.259    0.000 python2.7/_strptime.py:295(_strptime)
        1    0.000    0.000    2.244    2.244 pipeline.py:794(_prepare)
        1    0.001    0.001    2.240    2.240 post_process.py:45(prepare)
  2759725    1.569    0.000    2.228    0.000 pipeline.py:580(resolve_alias)
        1    0.000    0.000    2.227    2.227 post_process.py:664(build_underlying_lookup_dicts)
    45966    0.043    0.000    2.210    0.000 django/db/models/query.py:229(iterator)
   166708    0.705    0.000    2.183    0.000 metrics.py:136(inc)
  122/119    0.000    0.000    2.151    0.018 django/db/models/query.py:147(__iter__)
    27879    0.108    0.000    2.118    0.000 pipeline.py:241(flush)
    27876    0.389    0.000    2.004    0.000 cassandradataset.py:248(write)
```

Figure 4.9. Sequential program `cProfile` output. The most interesting functions (`process`, `process_record`, `post_process_record`, `consume_record`, and `prepare`) are highlighted in green.

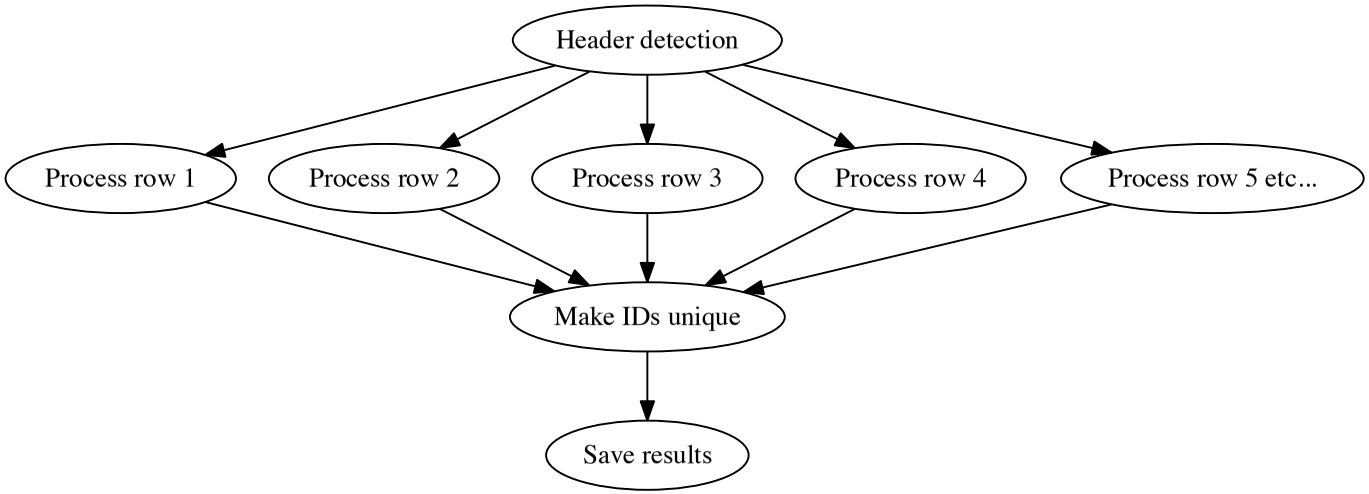

Figure 4.10. An example of a task DAG for a file format that does not contain global variable or state variables. Header detections needs to be performed up front, and making trade IDs unique needs to be performed in a post processing step. The processing of each row does not depend on each other.

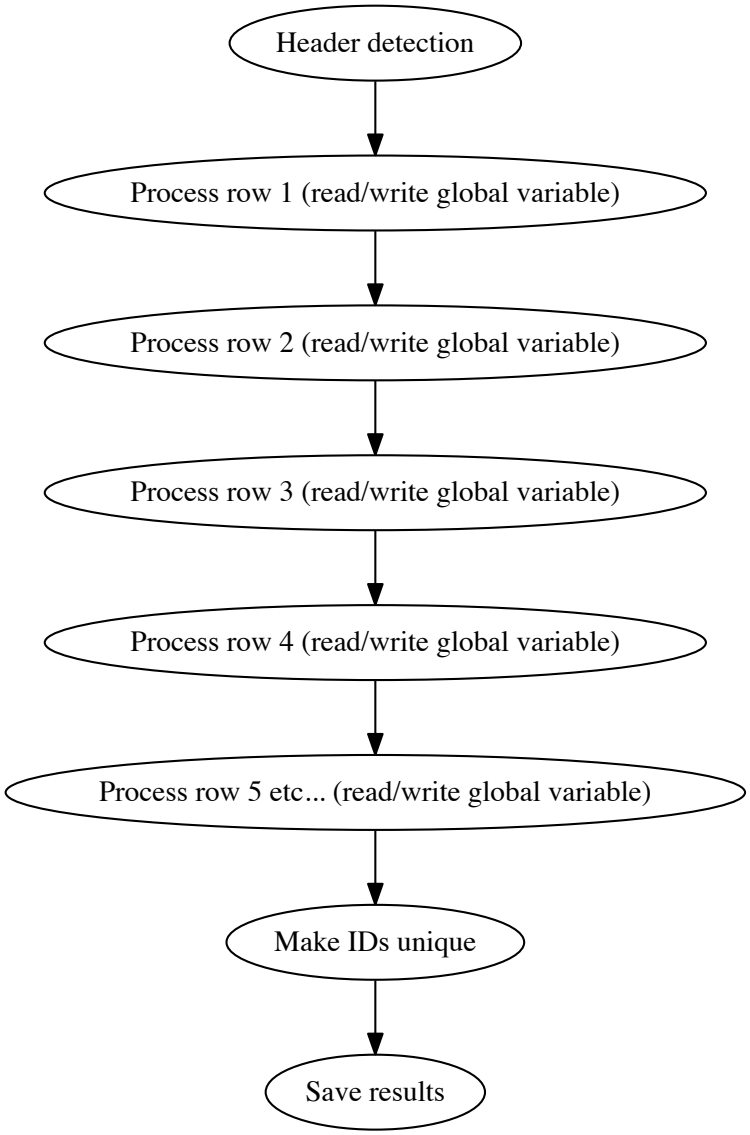

Figure 4.11. An example of a task DAG for a file format that contains global or state variables. Since each row may read and write the global (or state) variable, every task depends on the previous task.

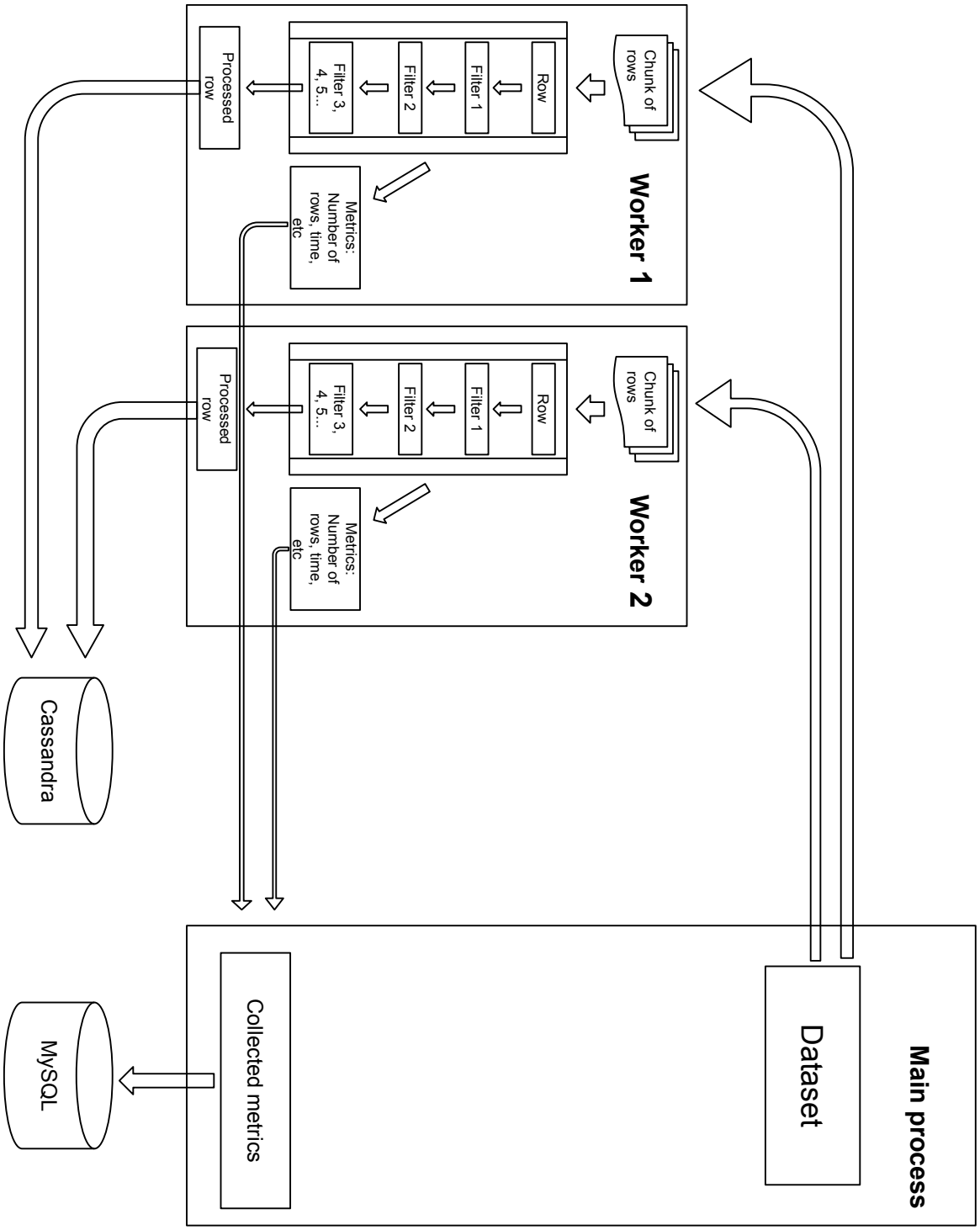

Figure 4.12. Parallel program overview.

Chapter 5

Benchmark Environment

In this section, the experiments and the hardware used to conduct them are described.

5.1 Hardware

For the main evaluation, TriOptima's acceptance test environment with hardware similar to what is used in production, was used. This is a stationary computer running Linux CentOS, with a dual processor architecture. The processors are 2 Intel Xeon E5504 processors with 4 cores each, making a total of 8 cores. The cores do not support hyperthreading. The machine has 32 GB of memory.

5.2 Test datasets

For all datasets, the number of filters is limited to a few more than the number of columns, as the majority of the filters are column mapping filters. This is typical for most datasets, making this a suitable sample of datasets. No datasets with inherently serial file formats where used, as these cannot be parallelized and would provide no interesting data. Instead, the focus of the experiments are on the *Extra overhead* file formats, since these are the most common (since making trade ID:s unique is a useful feature). They also give the fairest indication of how successful the parallelization is, as they are both the worst case (apart from inherently serial file formats) and the common case. Another reason why this set of datasets was chosen is their differing sizes, with potentially differing parallelization benefits.

The characteristics of the test datasets are outlined in figure 5.2.

Dataset	Rows	Columns	File format family
Tiny	102	46	Extra overhead
Small	2,890	46	Extra overhead
Medium	23,763	46	Extra overhead
Large	349309	41	Extra overhead

Table 5.1. Test datasets.

5.3 Experiments

5.3.1 Benchmarks

For each dataset, experiments were run to gather information about time and memory consumption for different configurations. The experiments entailed performing a dataset transformation using the parallel program with each number of workers in the range 1 to 2 times the number of cores: 1–16. In addition to running the parallel program, a serial transformation was also performed as a reference point. In each of the experiments, the Python `resource` module was used in each worker process to find the user time, system time, and maximum memory consumption. These values were then summed in order to find the total resource use. In the case of maximum memory consumption, this means that the number found is a worst case number. It is possible that the processes in total have a smaller maximum memory consumption since they may reach their maximum memory consumption at different points in time.

In order to provide more accurate results, the experiments were run 10 times for each dataset–worker number configuration. These 10 runs were then used to calculate the average value as well as the standard deviation for each metric value. In addition, the speedup was calculated by dividing the sequential execution time with the parallel execution time for each worker value.

5.3.2 Profiling

For each dataset, a single run with 8 workers was profiled using `cProfile` in order to find where in the program the most time is spent. This was done in order to better understand the benchmark results.

Chapter 6

Results

The results of the benchmarking and profiling experiments are outlined in this chapter.

6.1 Transformation benchmarks

The benchmarks for the experiments are outlined below. For each dataset, the results consist of a table containing speedup, real time, user time, system time, and memory usage for each worker number. The standard deviation is represented as a value following the ± sign. The value S in worker column signifies the original sequential program, while 1 is the parallel program with a single worker. In addition to the tables, the real time, speedup, and memory usage are illustrated using plots in order to visually demonstrate how the values change with the number of workers. In the plots for real time and memory usage, the standard deviation is illustrated using error bars above and below each data point[1].

6.2 Benchmark tables

6.2.1 Tiny dataset benchmark table

The benchmark for the tiny dataset can be found in figure 6.1. The table displays the number of workers, the speedup (sequential run real time divided by real time), real time, user time, system time, and memory usage. The standard deviation is displayed with a value following the ± sign.

6.2.2 Small dataset benchmark table

The benchmark for the small dataset can be found in figure 6.2. The table displays the number of workers, the speedup (sequential run real time divided by real time),

[1] The standard deviation is shown for all plots of real time and memory usage, but it may be too small to be seen for some of the plots.

Workers	Speedup	Real (s)	User (s)	System (s)	Memory usage (MB)
S	1.00	1.11 ± 0.13	2.32 ± 0.02	0.18 ± 0.00	102 ± 0.27
1	0.70	1.58 ± 0.04	2.47 ± 0.02	0.21 ± 0.01	199 ± 0.52
2	0.72	1.54 ± 0.03	2.72 ± 0.03	0.23 ± 0.00	296 ± 0.80
3	0.73	1.52 ± 0.03	3.04 ± 0.03	0.25 ± 0.01	392 ± 1.07
4	0.69	1.61 ± 0.11	3.24 ± 0.04	0.29 ± 0.01	489 ± 1.34
5	0.71	1.56 ± 0.03	3.47 ± 0.03	0.30 ± 0.00	586 ± 1.70
6	0.66	1.67 ± 0.06	3.72 ± 0.03	0.32 ± 0.01	683 ± 1.99
7	0.62	1.79 ± 0.10	3.93 ± 0.04	0.34 ± 0.01	780 ± 2.25
8	0.64	1.74 ± 0.04	4.20 ± 0.03	0.37 ± 0.01	877 ± 2.55
9	0.62	1.80 ± 0.03	4.41 ± 0.03	0.39 ± 0.01	973 ± 2.82
10	0.48	2.31 ± 0.38	4.55 ± 0.05	0.41 ± 0.01	1070 ± 3.12
11	0.55	2.02 ± 0.05	4.81 ± 0.04	0.43 ± 0.01	1167 ± 3.38
12	0.52	2.15 ± 0.04	5.10 ± 0.04	0.46 ± 0.01	1264 ± 3.65
13	0.42	2.67 ± 0.43	5.24 ± 0.04	0.49 ± 0.01	1361 ± 3.92
14	0.47	2.34 ± 0.08	5.37 ± 0.05	0.48 ± 0.01	1361 ± 3.91
15	0.45	2.45 ± 0.08	5.64 ± 0.04	0.51 ± 0.01	1545 ± 7.77
16	0.47	2.34 ± 0.06	5.79 ± 0.07	0.54 ± 0.01	1554 ± 4.50

Table 6.1. Tiny dataset benchmark table. The table displays the number of workers, the speedup (sequential run real time divided by real time), real time, user time, system time, and memory usage. The standard deviation is displayed with a value following the ± sign.

real time, user time, system time, and memory usage. The standard deviation is displayed with a value following the ± sign.

6.2.3 Medium dataset benchmark table

The benchmark for the medium dataset can be found in figure 6.3. The table displays the number of workers, the speedup (sequential run real time divided by real time), real time, user time, system time, and memory usage. The standard deviation is displayed with a value following the ± sign.

6.2.4 Large dataset benchmark table

The benchmark for the large dataset can be found in figure 6.4. The table displays the number of workers, the speedup (sequential run real time divided by real time), real time, user time, system time, and memory usage. The standard deviation is displayed with a value following the ± sign.

Workers	Speedup	Real (s)	User (s)	System (s)	Memory usage (MB)
S	1.00	15.28 ± 0.27	13.89 ± 0.11	0.33 ± 0.01	137 ± 0.30
1	0.87	17.48 ± 0.39	14.80 ± 0.09	0.40 ± 0.01	239 ± 0.50
2	1.17	13.11 ± 0.15	16.04 ± 0.12	0.52 ± 0.01	361 ± 0.81
3	1.47	10.37 ± 0.10	16.53 ± 0.49	0.59 ± 0.02	467 ± 11.14
4	1.71	8.96 ± 0.07	17.55 ± 0.34	0.67 ± 0.01	588 ± 11.08
5	1.84	8.31 ± 0.15	18.51 ± 0.24	0.76 ± 0.01	720 ± 1.67
6	1.99	7.66 ± 0.07	19.45 ± 0.09	0.83 ± 0.01	837 ± 1.96
7	2.06	7.43 ± 0.09	20.38 ± 0.12	0.92 ± 0.01	956 ± 2.29
8	2.12	7.21 ± 0.11	21.36 ± 0.14	1.02 ± 0.02	1075 ± 2.38
9	2.09	7.30 ± 0.11	21.82 ± 0.12	1.10 ± 0.01	1194 ± 2.73
10	2.01	7.59 ± 0.31	22.63 ± 0.09	1.17 ± 0.01	1313 ± 3.18
11	2.14	7.14 ± 0.11	22.96 ± 0.20	1.23 ± 0.02	1432 ± 3.30
12	2.04	7.48 ± 0.13	23.90 ± 0.17	1.32 ± 0.01	1550 ± 3.57
13	2.01	7.59 ± 0.14	24.84 ± 0.10	1.42 ± 0.02	1669 ± 3.82
14	2.05	7.45 ± 0.11	25.58 ± 0.10	1.49 ± 0.01	1788 ± 4.09
15	2.06	7.42 ± 0.09	26.16 ± 0.15	1.56 ± 0.01	1876 ± 12.17
16	2.01	7.59 ± 0.07	27.29 ± 0.15	1.62 ± 0.02	2006 ± 4.62

Table 6.2. Small dataset benchmark table. The table displays the number of workers, the speedup (sequential run real time divided by real time), real time, user time, system time, and memory usage. The standard deviation is displayed with a value following the ± sign.

Workers	Speedup	Real (s)	User (s)	System (s)	Memory usage (MB)
S	1.00	75.06 ± 0.73	74.41 ± 0.57	0.57 ± 0.02	393 ± 0.35
1	0.97	77.37 ± 0.38	75.21 ± 0.36	0.67 ± 0.02	484 ± 0.62
2	1.73	43.29 ± 0.14	80.75 ± 0.20	0.83 ± 0.01	840 ± 0.82
3	2.29	32.81 ± 0.10	88.02 ± 0.18	1.09 ± 0.01	1199 ± 1.23
4	2.77	27.08 ± 0.05	91.16 ± 2.01	1.30 ± 0.03	1523 ± 31.69
5	3.16	23.74 ± 0.10	97.66 ± 0.23	1.53 ± 0.02	1910 ± 1.70
6	3.44	21.84 ± 0.23	102.44 ± 0.36	1.79 ± 0.03	2266 ± 1.87
7	3.65	20.57 ± 0.36	106.39 ± 1.35	2.01 ± 0.04	2589 ± 31.67
8	3.76	19.94 ± 0.31	111.90 ± 0.40	2.34 ± 0.02	2979 ± 2.46
9	3.82	19.65 ± 0.42	115.61 ± 0.27	2.55 ± 0.03	3331 ± 2.74
10	3.78	19.87 ± 0.30	118.06 ± 1.21	2.79 ± 0.03	3621 ± 41.47
11	3.74	20.07 ± 0.19	122.24 ± 1.45	2.98 ± 0.04	3980 ± 41.25
12	3.70	20.31 ± 0.20	128.46 ± 0.29	3.35 ± 0.04	4402 ± 3.54
13	3.67	20.48 ± 0.25	130.92 ± 1.33	3.53 ± 0.05	4693 ± 43.07
14	3.60	20.83 ± 0.23	136.32 ± 0.96	3.79 ± 0.04	5083 ± 32.24
15	3.48	21.60 ± 0.19	141.99 ± 0.99	4.10 ± 0.05	5469 ± 4.39
16	3.41	22.01 ± 0.20	144.60 ± 0.61	4.30 ± 0.04	5831 ± 4.73

Table 6.3. Medium dataset benchmark table. The table displays the number of workers, the speedup (sequential run real time divided by real time), real time, user time, system time, and memory usage. The standard deviation is displayed with a value following the ± sign.

Workers	Speedup	Real (s)	User (s)	System (s)	Memory usage (MB)
S	1.00	2577.81 ± 27.57	2037.83 ± 17.43	420.68 ± 7.50	1056 ± 26.46
1	1.01	2542.28 ± 43.96	2006.77 ± 31.70	383.28 ± 4.19	1192 ± 23.67
2	1.85	1391.80 ± 25.73	2069.22 ± 24.28	468.81 ± 4.00	1577 ± 28.97
3	2.52	1022.66 ± 16.26	2114.43 ± 19.11	515.31 ± 5.21	1993 ± 27.19
4	3.06	842.63 ± 29.41	2153.54 ± 19.97	548.10 ± 4.97	2423 ± 13.39
5	3.63	709.95 ± 23.81	2186.59 ± 10.82	580.02 ± 4.97	2843 ± 4.26
6	3.84	671.77 ± 26.63	2210.57 ± 7.21	612.45 ± 4.49	3292 ± 2.16
7	4.18	617.39 ± 28.28	2245.29 ± 10.66	635.07 ± 3.79	3756 ± 1.31
8	4.30	599.12 ± 24.50	2249.47 ± 9.82	666.14 ± 7.19	4225 ± 3.65
9	4.45	579.12 ± 21.91	2290.80 ± 16.17	674.96 ± 7.12	4640 ± 0.69
10	4.51	571.42 ± 14.36	2299.32 ± 10.67	681.82 ± 5.69	5105 ± 1.30
11	4.66	552.94 ± 14.92	2316.17 ± 9.88	695.41 ± 6.23	5535 ± 2.66
12	4.65	554.22 ± 17.36	2333.11 ± 15.31	700.38 ± 6.73	5986 ± 1.38
13	4.72	545.68 ± 19.41	2358.86 ± 15.85	715.91 ± 7.41	6454 ± 1.38
14	4.81	535.96 ± 15.07	2382.01 ± 13.12	712.07 ± 6.96	6878 ± 1.27
15	4.79	537.71 ± 13.91	2397.88 ± 10.56	717.12 ± 6.61	7313 ± 1.68
16	4.78	539.12 ± 14.88	2416.33 ± 16.06	722.39 ± 7.86	7761 ± 1.03

Table 6.4. Large dataset benchmark table. The table displays the number of workers, the speedup (sequential run real time divided by real time), real time, user time, system time, and memory usage. The standard deviation is displayed with a value following the ± sign.

6.3 Execution time

6.3.1 Tiny dataset execution time

The real execution time for the tiny dataset can be found in figure 6.1. The X axis shows the number of workers, where 0 signifies the sequential program run. The Y axis shows the real execution time for each worker value. The standard deviation is displayed with black bars around each data point. Real time increases as number of workers increase. The data points at 10 and 13 workers show high standard deviation.

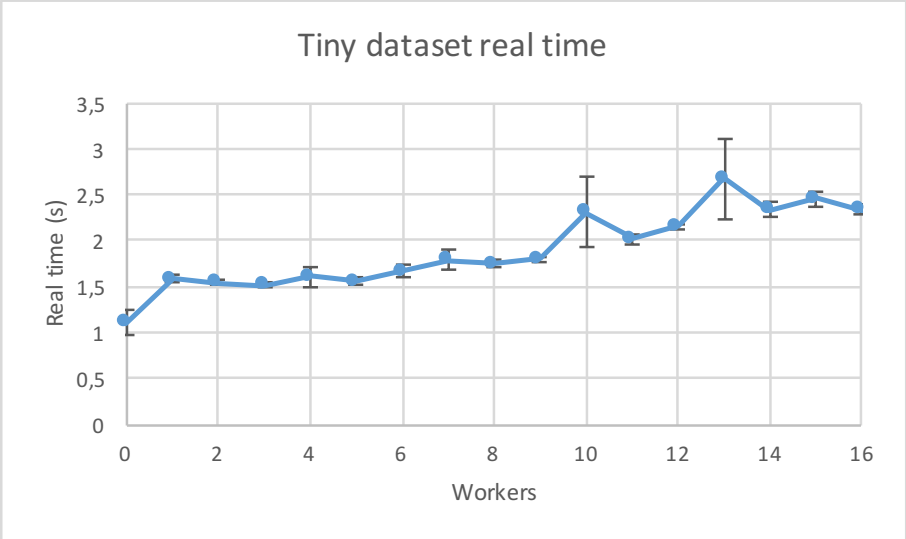

Figure 6.1. Real time plot for the tiny dataset. The X axis shows the number of workers, where 0 signifies the sequential program run. The Y axis shows the real execution time for each worker value. The standard deviation is displayed with black bars around each data point. Real time increases as number of workers increase. The data points at 10 and 13 workers show high standard deviation.

6.3.2 Small dataset execution time

The real execution time for the small dataset can be found in figure 6.2. The X axis shows the number of workers, where 0 signifies the sequential program run. The Y axis shows the real execution time for each worker value. The standard deviation is displayed with black bars around each data point. The real time decreases with each added worker up to 8 workers, where it evens out. The decrease in real time for each added worker becomes smaller as the number increases.

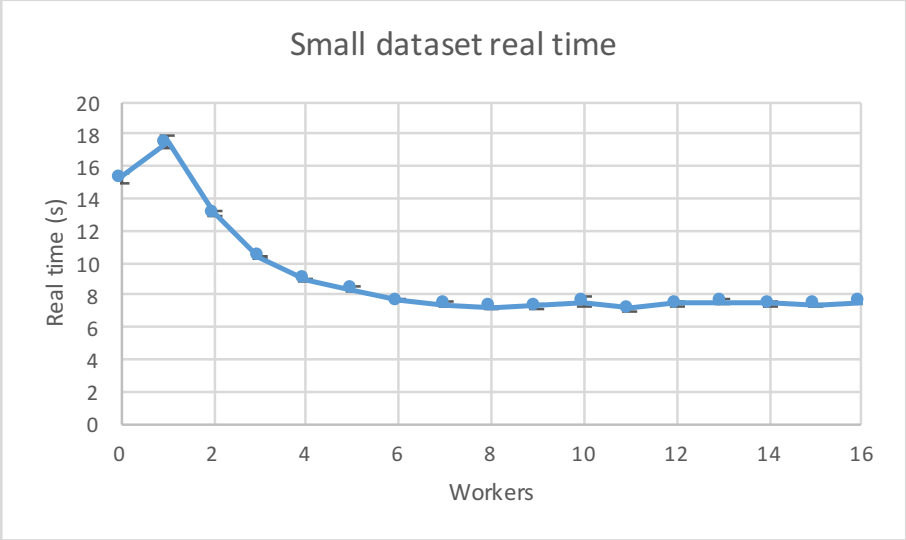

Figure 6.2. Real time plot for the small dataset. The X axis shows the number of workers, where 0 signifies the sequential program run. The Y axis shows the real execution time for each worker value. The standard deviation is displayed with black bars around each data point. The real time decreases with each added worker up to 8 workers, where it evens out. The decrease in real time for each added worker becomes smaller as the number increases.

6.3.3 Medium dataset execution time

The real execution time for the medium dataset can be found in figure 6.3. The X axis shows the number of workers, where 0 signifies the sequential program run. The Y axis shows the real execution time for each worker value. The standard deviation is displayed with black bars around each data point. The real time decreases with each added worker up to 8 workers, where it evens out. The decrease in real time for each added worker becomes smaller as the number increases.

6.3.4 Large dataset execution time

The real execution time for the large dataset can be found in figure 6.4. The X axis shows the number of workers, where 0 signifies the sequential program run. The Y axis shows the real execution time for each worker value. The standard deviation is displayed with black bars around each data point. The real time decreases with each added worker up to 11 workers, where it evens out. The decrease in real time for each added worker becomes smaller as the number increases.

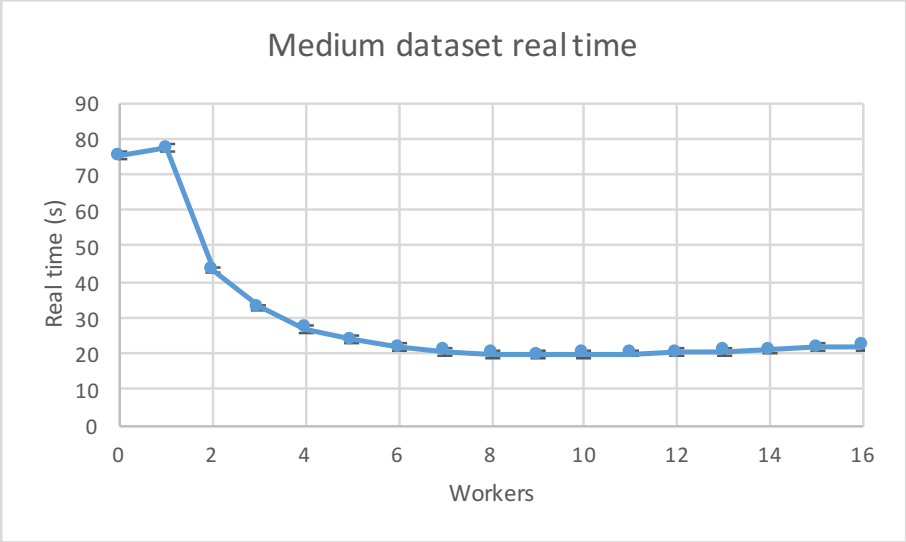

Figure 6.3. Real time plot for the medium dataset. The X axis shows the number of workers, where 0 signifies the sequential program run. The Y axis shows the real execution time for each worker value. The standard deviation is displayed with black bars around each data point. The real time decreases with each added worker up to 8 workers, where it evens out. The decrease in real time for each added worker becomes smaller as the number increases.

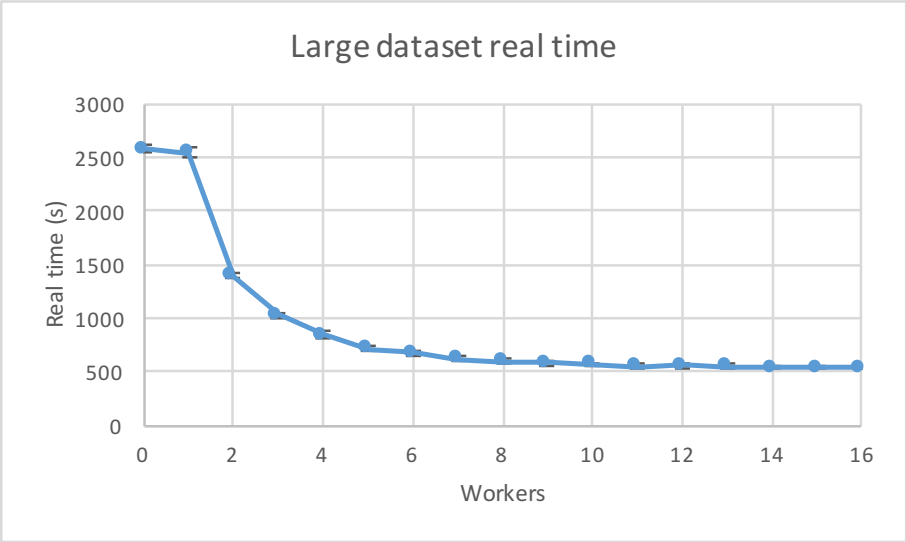

Figure 6.4. Real time plot for the large dataset. The X axis shows the number of workers, where 0 signifies the sequential program run. The Y axis shows the real execution time for each worker value. The standard deviation is displayed with black bars around each data point. The real time decreases with each added worker up to 11 workers, where it evens out. The decrease in real time for each added worker becomes smaller as the number increases.

6.4 Speedup

6.4.1 Tiny dataset speedup

The speedup for the tiny dataset can be found in figure 6.5. The X axis shows the number of workers, and the Y axis shows a scalar value signifying the speedup as "number of times faster than sequential execution". Speedup decreases with each added worker, down to about half the speed of the sequential execution.

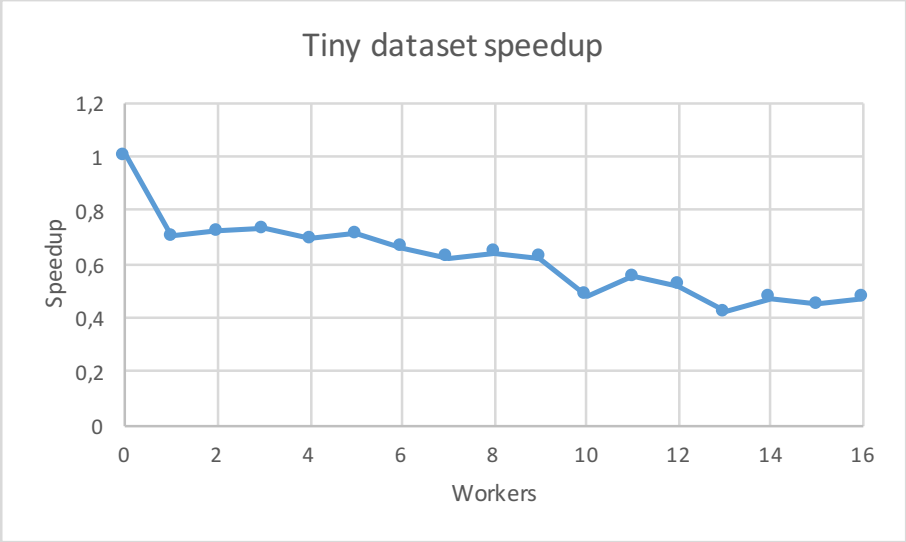

Figure 6.5. Speedup plot for the tiny dataset. The X axis shows the number of workers, and the Y axis shows a scalar value signifying the speedup as "number of times faster than sequential execution". Speedup decreases with each added worker, down to about half the speed of the sequential execution.

6.4.2 Small dataset speedup

The speedup for the small dataset can be found in figure 6.6. The X axis shows the number of workers, and the Y axis shows a scalar value signifying the speedup as "number of times faster than sequential execution". Speedup increases with each worker, evening out around 8 workers and 2.1X speedup.

6.4.3 Medium dataset speedup

The speedup for the medium dataset can be found in figure 6.7. The X axis shows the number of workers, and the Y axis shows a scalar value signifying the speedup as "number of times faster than sequential execution". Speedup increases with each worker, evening out around 8-9 workers and 3.8X speedup.

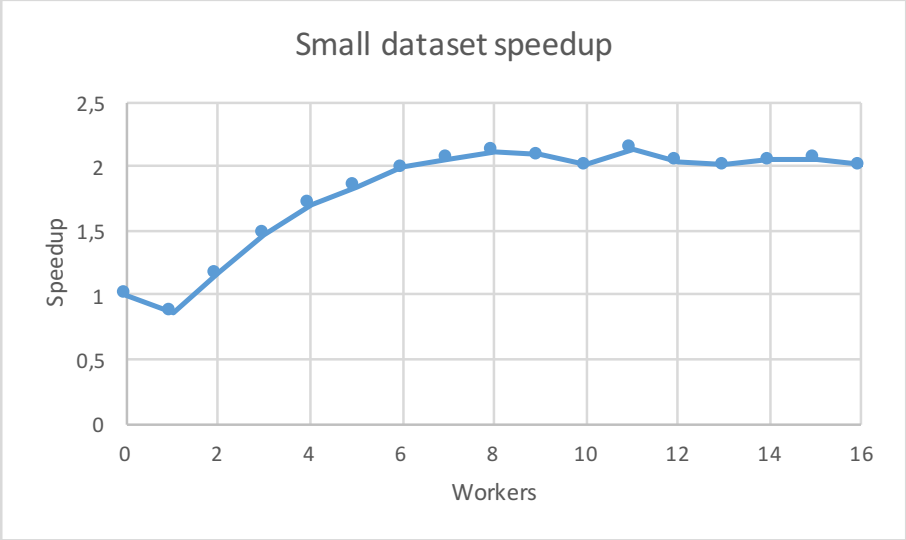

Figure 6.6. Speedup plot for the small dataset. The X axis shows the number of workers, and the Y axis shows a scalar value signifying the speedup as "number of times faster than sequential execution". Speedup increases with each worker, evening out around 8 workers and 2.1X speedup.

6.4.4 Large dataset speedup

The speedup for the large dataset can be found in figure 6.8. The X axis shows the number of workers, and the Y axis shows a scalar value signifying the speedup as "number of times faster than sequential execution". Speedup increases with each worker, evening out around 11 workers and with 4.81X as the highest speedup.

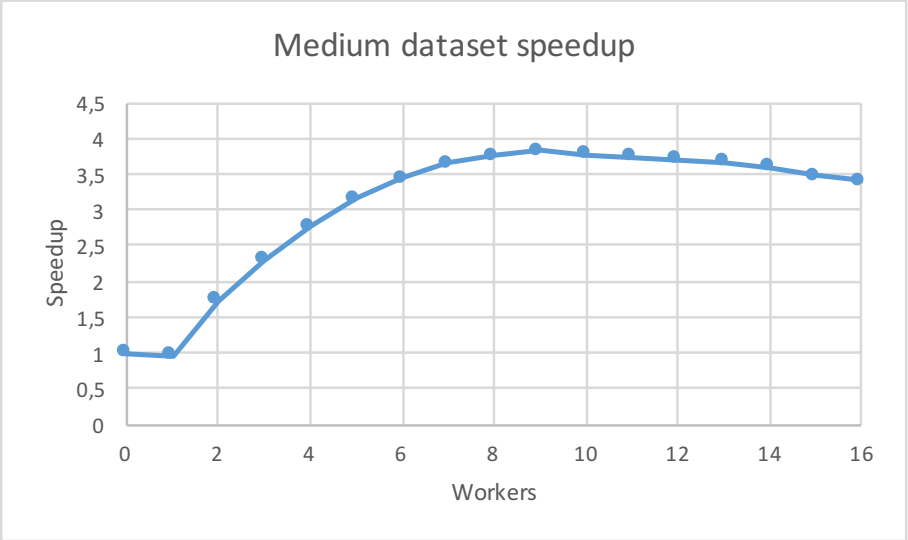

Figure 6.7. Speedup plot for the medium dataset. The X axis shows the number of workers, and the Y axis shows a scalar value signifying the speedup as "number of times faster than sequential execution". Speedup increases with each worker, evening out around 8-9 workers and 3.8X speedup.

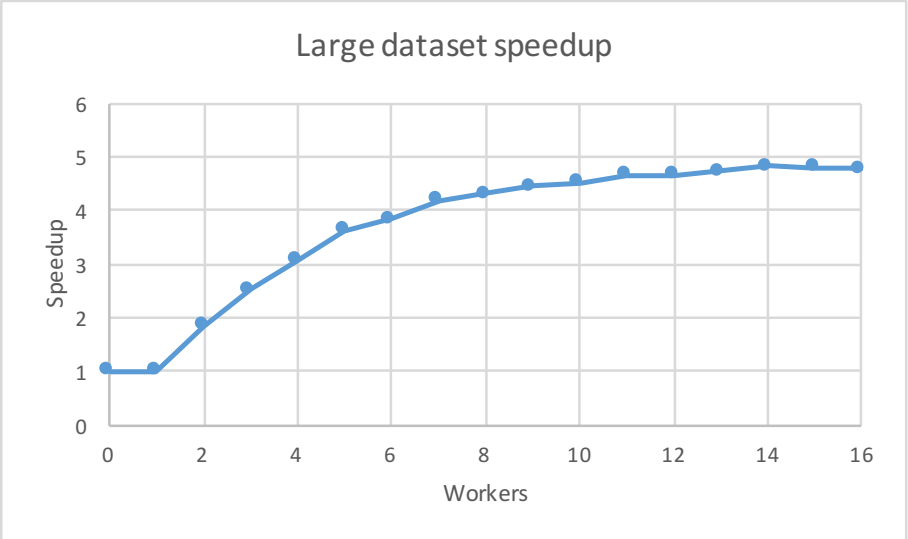

Figure 6.8. Speedup plot for the large dataset. The X axis shows the number of workers, and the Y axis shows a scalar value signifying the speedup as "number of times faster than sequential execution". Speedup increases with each worker, evening out around 11 workers and with 4.81X as the highest speedup.

6.5 Memory consumption

6.5.1 Tiny dataset memory consumption

The memory consumption for the tiny dataset can be found in figure 6.9. The X axis shows the number of workers, and the Y axis shows the total memory usage as a sum of the highest memory usage for each worker process, in addition to the main process. Memory usage increases close to linearly with each added worker, up to about 1.6 GB for 16 workers.

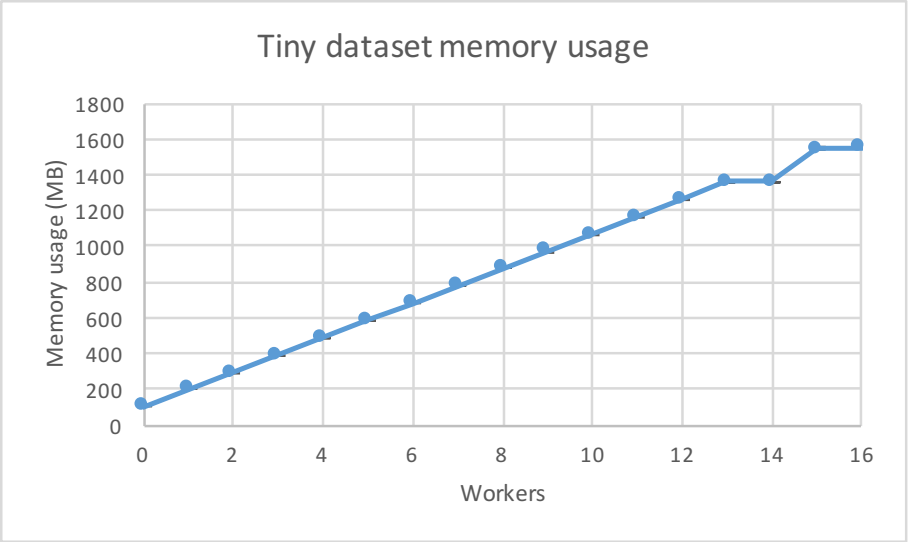

Figure 6.9. Memory usage plot for the tiny dataset. The X axis shows the number of workers, and the Y axis shows the total memory usage as a sum of the highest memory usage for each worker process, in addition to the main process. Memory usage increases close to linearly with each added worker, up to about 1.6 GB for 16 workers.

6.5.2 Small dataset memory consumption

The memory consumption for the tiny dataset can be found in figure 6.10. The X axis shows the number of workers, and the Y axis shows the total memory usage as a sum of the highest memory usage for each worker process, in addition to the main process. Memory usage increases close to linearly with each added worker, up to about 2 GB for 16 workers.

6.5.3 Medium dataset memory consumption

The memory consumption for the tiny dataset can be found in figure 6.11. The X axis shows the number of workers, and the Y axis shows the total memory usage as a sum of the highest memory usage for each worker process, in addition to the

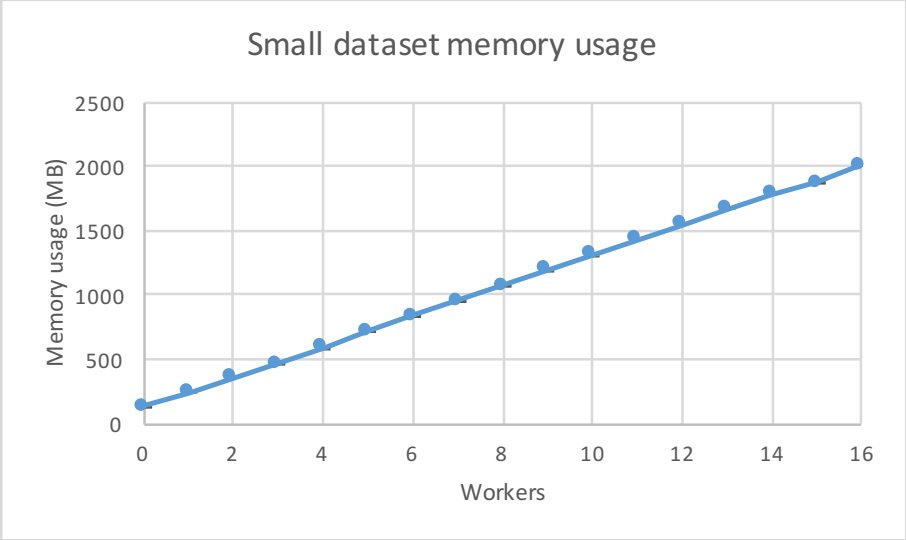

Figure 6.10. Memory usage plot for the small dataset. The X axis shows the number of workers, and the Y axis shows the total memory usage as a sum of the highest memory usage for each worker process, in addition to the main process. Memory usage increases close to linearly with each added worker, up to about 2 GB for 16 workers.

main process. Memory usage increases close to linearly with each added worker, up to about 6 GB for 16 workers.

6.5.4 Large dataset memory consumption

The memory consumption for the tiny dataset can be found in figure 6.12. The X axis shows the number of workers, and the Y axis shows the total memory usage as a sum of the highest memory usage for each worker process, in addition to the main process. Memory usage increases close to linearly with each added worker, up to about 8.8 GB for 16 workers.

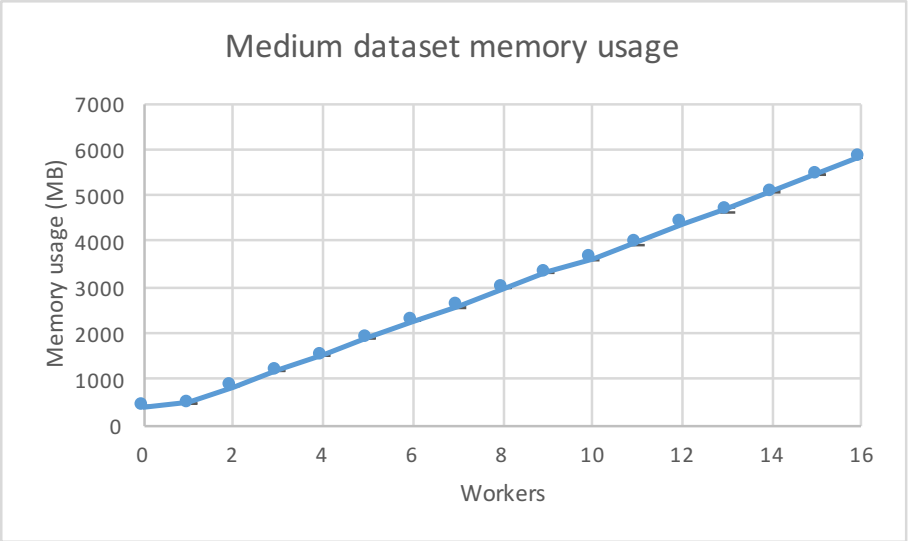

Figure 6.11. Memory usage plot for the medium dataset. The X axis shows the number of workers, and the Y axis shows the total memory usage as a sum of the highest memory usage for each worker process, in addition to the main process. Memory usage increases close to linearly with each added worker, up to about 6 GB for 16 workers.

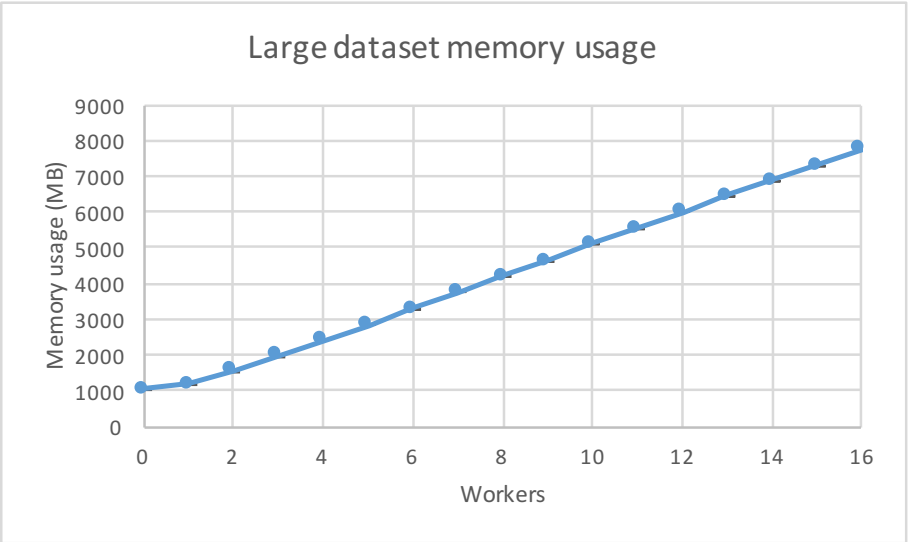

Figure 6.12. Memory usage plot for the large dataset. The X axis shows the number of workers, and the Y axis shows the total memory usage as a sum of the highest memory usage for each worker process, in addition to the main process. Memory usage increases close to linearly with each added worker, up to about 7.7 GB for 16 workers.

6.6 Parallel profiler analysis

For each dataset, a profiling session of the parallel program was conducted. The same hardware as for the benchmarks was used, and the profiling was done for a transformation run with 8 workers. The functions highlighted and discussed are the ones with the highest cumulative time which are at the top level. These are the most interesting functions to examine as they take up a large part of the running time, and other functions with high cumulative time are also called from these.

A function that is present in each of the analyses is `aggregate_result`, which contains the code that creates the other processes and waits for these to produce their partial results, aggregating them as they are produced. It is therefore a large part of the execution time for each main process run. Another function seen in each main process is `post_process_parallel`, which is responsible for the sequential post processing step of making trade ID:s unique (which happens after `aggregate_result`.

For the worker processes, `process` is the main dataset processing function, explaining its prominence in each profiling session.

6.6.1 Tiny dataset profiler analysis

The parallel program `cProfile` output for the tiny dataset can be found in figure 6.13 (main process) and figure 6.14 (worker process). Functions with low cumulative time have been omitted.

For the main process, the most prominent functions are `aggregate_result` (70% of the execution time), `perform_header_detection` (11.5% of the execution time), and `post_process_parallel` (7.8% of the execution time). `perform_header_detection` finds the header of the dataset, which the other processes needs to process their dataset chunks.

Only one worker process is shown, as the results where very similar. For the worker process, apart from `process`, `readMessageBegin` (44.9%) of the worker execution time is one of the functions that take the most time. This function is part of the communication protocol between the program and the database, indicating that communication takes up a large part of the worker process for the tiny dataset, taking up larger portions of the total time than row processing functions such as `process_record` (9.0% of the execution time).

6.6.2 Small dataset profiler analysis

The parallel program `cProfile` output for the small dataset can be found in figure 6.15 (main process) and figure 6.16 (worker process). Functions with low cumulative time have been omitted.

For the main process, the major functions are `aggregate_result` (71.7%) and `post_process_parallel` (22.5%).

```
         168902 function calls (163541 primitive calls) in 2.540 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    2.540    2.540 importer.py:433(verify)
        1    0.000    0.000    2.507    2.507 importer.py:399(_do_verify)
        1    0.000    0.000    2.492    2.492 verifier.py:441(verify)
        1    0.000    0.000    2.312    2.312 verifier.py:546(do_verify_parallel)
        1    0.000    0.000    1.789    1.789 verifier.py:565(aggregate_result)
       16    0.000    0.000    1.786    0.112 queues.py:113(get)
       16    1.783    0.111    1.786    0.112 method 'recv' of '_multiprocessing.Connection' objects
        1    0.000    0.000    0.294    0.294 verifier.py:611(perform_header_detection)
       62    0.000    0.000    0.284    0.005 TBinaryProtocol.py:125(readMessageBegin)
      186    0.000    0.000    0.284    0.002 TBinaryProtocol.py:205(readI32)
  372/248    0.001    0.000    0.284    0.001 TTransport.py:54(readAll)
      248    0.001    0.000    0.283    0.001 TTransport.py:271(read)
       62    0.000    0.000    0.282    0.005 TTransport.py:279(readFrame)
      127    0.001    0.000    0.281    0.002 TSocket.py:103(read)
      127    0.280    0.002    0.280    0.002 method 'recv' of '_socket.socket' objects
       17    0.000    0.000    0.245    0.014 cassandradataset.py:320(open)
        1    0.000    0.000    0.210    0.210 pipeline.py:806(process)
        3    0.000    0.000    0.202    0.067 models.py:694(__iter__)
        1    0.000    0.000    0.199    0.199 parallel_post_process.py:8(post_process_parallel)
```

Figure 6.13. Parallel program `cProfile` output for the main process of the tiny dataset. The most interesting functions are highlighted in green. For the main process, the most prominent functions are `aggregate_result` (70% of the execution time), `perform_header_detection` (11.5% of the execution time), and `post_process_parallel` (7.8% of the execution time). `perform_header_detection` finds the header of the dataset, which the other processes needs to process their dataset chunks.

Only one worker process is shown, as the results where very similar. For the worker process `_fetch_all` (38%) is responisble for getting the dataset rows. The row processing functions take up a larger portion of the code than for the tiny dataset, 33.6% for `post_process_record`, 21.5% for `consume_record`, and 14.6% for `process_record`.

6.6.3 Medium dataset profiler analysis

The parallel program `cProfile` output for the medium dataset can be found in figure 6.17 (main process) and figure 6.18 (worker process). Functions with low cumulative time have been omitted.

For the main process, the major functions are `aggregate_result` (91.1%) and `post_process_parallel` (7.0%).

Only one worker process is shown, as the results where very similar. For the worker process, the functions related to row processing are more prominent: `process_record` (26%), `post_process_record` (24.8%), `consume_record` (8.7%). `_prepare`, called before the main row processing loop, takes up 22% of the total

```
         208905 function calls (204849 primitive calls) in 1.447 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    1.389    1.389 verifier_parallel.py:61(start_pipeline)
        1    0.000    0.000    1.226    1.226 verifier_parallel.py:117(process)
        1    0.000    0.000    1.198    1.198 verifier_parallel.py:148(pipeline_process)
        1    0.007    0.007    1.176    1.176 pipeline.py:806(process)
       16    0.000    0.000    0.624    0.039 TBinaryProtocol.py:125(readMessageBegin)
        2    0.000    0.000    0.473    0.236 models.py:701(get_rows)
        2    0.000    0.000    0.473    0.236 models.py:688(get_ds_manager)
        2    0.000    0.000    0.472    0.236 cassandradataset.py:320(open)
  173/168    0.002    0.000    0.449    0.003 django/db/models/query.py:972(_fetch_all)
      100    0.001    0.000    0.306    0.003 django/db/models/sql/compiler.py:814(execute_sql)
      991    0.005    0.000    0.300    0.000 django/db/models/query.py:229(iterator)
       13    0.000    0.000    0.254    0.020 verifier.py:330(consume_record)
       13    0.000    0.000    0.254    0.020 verifier.py:268(_write_record)
       52    0.000    0.000    0.202    0.004 pipeline.py:838(post_process_record)
      102    0.001    0.000    0.195    0.002 MySQLdb/cursors.py:288(_do_query)
      102    0.188    0.002    0.188    0.002 method 'query' of '_mysql.connection' objects
   102/99    0.000    0.000    0.187    0.002 query.py:147(__iter__)
      237    0.001    0.000    0.157    0.001 translation_manager.py:68(hooked)
      650    0.001    0.000    0.157    0.000 pipeline.py:834(process_record)
        1    0.000    0.000    0.157    0.157 verifier_parallel.py:82(build_pipeline)
        1    0.000    0.000    0.152    0.152 cassandradataset.py:323(create)
```

Figure 6.14. Parallel program `cProfile` output for a worker process of the tiny dataset. The most interesting functions are highlighted in green. For the worker process, apart from `process`, `readMessageBegin` (44.9%) of the worker execution time is one of the functions that take the most time. This function is part of the communication protocol between the program and the database, indicating that communication takes up a large part of the worker process for the tiny dataset, taking up larger portions of the total time than row processing functions such as `process_record` (9.0% of the execution time).

time.

6.6.4 Large dataset profiler analysis

The parallel program `cProfile` output for the large dataset can be found in figure 6.19 (main process), figure 6.20 (slow worker process), and 6.21. Functions with low cumulative time have been omitted.

For the main process, the major functions are `aggregate_result` (91.1%) and `post_process_parallel` (7.0%).

As the workers showed different timings, two workers, a slow one and a fast one are shown. For the slow worker process, the functions related to row processing are prominent: `process_record` (40.7%), `post_process_record` (30%), `consume_record` (8.7%). `_prepare`, called before the main row processing loop, takes up 1.3% of the total time.

For the fast worker process, the functions related to row processing are promi-

```
      540012 function calls (533520 primitive calls) in 10.057 seconds

Ordered by: cumulative time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000   10.066   10.066 importer.py:433(verify)
     1    0.000    0.000   10.019   10.019 importer.py:399(_do_verify)
     1    0.000    0.000   10.002   10.002 verifier.py:441(verify)
     1    0.000    0.000    9.827    9.827 verifier.py:546(do_verify_parallel)
     1    0.000    0.000    7.224    7.224 verifier.py:565(aggregate_result)
    16    0.000    0.000    7.219    0.451 multiprocessing/queues.py:113(get)
    16    7.216    0.451    7.219    0.451 method 'recv' of '_multiprocessing.Connection' objects
     1    0.001    0.001    2.260    2.260 parallel_post_process.py:8(post_process_parallel)
     1    0.003    0.003    1.447    1.447 parallel_post_process.py:70(make_unique)
    32    0.023    0.001    1.439    0.045 parallel_post_process.py:76(write_relation_unique)
   230    0.001    0.000    0.951    0.004 TBinaryProtocol.py:125(readMessageBegin)
```

Figure 6.15. Parallel program `cProfile` output for the main process of the small dataset. The most interesting functions are highlighted in green. For the main process, the major functions are `aggregate_result` (71.7%) and `post_process_parallel` (22.5%).

nent: `process_record` (48.3%), `post_process_record` (21.8%), `consume_record` (5.4%). `_prepare`, called before the main row processing loop, takes up 1.6% of the total time.

```
         1718055 function calls (1701979 primitive calls) in 6.942 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    6.886    6.886 verifier_parallel.py:61(start_pipeline)
        1    0.000    0.000    5.868    5.868 verifier_parallel.py:117(process)
        1    0.000    0.000    5.730    5.730 verifier_parallel.py:148(pipeline_process)
        1    0.201    0.201    5.729    5.729 pipeline.py:806(process)
  581/576    0.055    0.000    2.621    0.005 query.py:972(_fetch_all)
     1448    0.004    0.000    2.314    0.002 pipeline.py:838(post_process_record)
      444    0.005    0.000    2.122    0.005 compiler.py:814(execute_sql)
  350/347    0.001    0.000    2.017    0.006 query.py:147(__iter__)
      472    0.004    0.000    1.846    0.004 utils.py:58(execute)
      472    0.001    0.000    1.842    0.004 base.py:137(execute)
      472    0.007    0.000    1.840    0.004 MySQLdb/cursors.py:141(execute)
      472    0.001    0.000    1.819    0.004 MySQLdb/cursors.py:326(_query)
     1918    0.017    0.000    1.583    0.001 query.py:229(iterator)
      362    0.004    0.000    1.478    0.004 verifier.py:330(consume_record)
      516    0.010    0.000    1.464    0.003 verifier.py:268(_write_record)
       52    0.000    0.000    1.445    0.028 TBinaryProtocol.py:125(readMessageBegin)
        1    0.000    0.000    1.014    1.014 verifier_parallel.py:82(build_pipeline)
    19548    0.028    0.000    1.002    0.000 pipeline.py:834(process_record)
```

Figure 6.16. Parallel program `cProfile` output for a worker process of the small dataset. The most interesting functions are highlighted in green. For the worker process `_fetch_all` (38%) is responisble for getting the dataset rows. The row processing functions take up a larger portion of the code than for the tiny dataset, 33.6% for `post_process_record`, 21.5% for `consume_record`, and 14.6% for `process_record`.

```
         793459 function calls (786921 primitive calls) in 33.076 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000   33.097   33.097 importer.py:433(verify)
        1    0.000    0.000   33.068   33.068 importer.py:399(_do_verify)
        1    0.000    0.000   33.047   33.047 verifier.py:441(verify)
        1    0.000    0.000   32.873   32.873 verifier.py:546(do_verify_parallel)
        1    0.000    0.000   30.138   30.138 verifier.py:565(aggregate_result)
       16    0.000    0.000   30.134    1.883 multiprocessing/queues.py:113(get)
       16   30.132    1.883   30.134    1.883 method 'recv' of '_multiprocessing.Connection' objects
        1    0.013    0.013    2.334    2.334 parallel_post_process.py:8(post_process_parallel)
        1    0.000    0.000    2.306    2.306 parallel_post_process.py:26(find_duplicates)
       15    0.114    0.008    2.292    0.153 parallel_post_process.py:34(update_relation_duplicates)
    23780    0.123    0.000    1.633    0.000 cassandradataset.py:107(_get_many)
    23762    0.662    0.000    0.821    0.000 cassandradataset.py:123(_convert_record)
```

Figure 6.17. Parallel program `cProfile` output for the main process of the medium dataset. The most interesting functions are highlighted in green. For the main process, the major functions are `aggregate_result` (91.1%) and `post_process_parallel` (7.0%).

```
       11047815 function calls (10958607 primitive calls) in 30.103 seconds

  Ordered by: cumulative time

  ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       1    0.000    0.000   30.050   30.050 verifier_parallel.py:61(start_pipeline)
       1    0.000    0.000   28.908   28.908 verifier_parallel.py:117(process)
       1    0.000    0.000   28.803   28.803 verifier_parallel.py:148(pipeline_process)
       1    1.637    1.637   28.802   28.802 pipeline.py:806(process)
 279/274    0.115    0.000    8.349    0.030 query.py:972(_fetch_all)
  157463    0.247    0.000    7.804    0.000 pipeline.py:834(process_record)
   11884    0.032    0.000    7.440    0.001 pipeline.py:838(post_process_record)
 117/114    0.000    0.000    7.389    0.065 query.py:147(__iter__)
   48948    0.166    0.000    7.132    0.000 query.py:229(iterator)
       1    0.000    0.000    6.648    6.648 pipeline.py:794(_prepare)
       1    0.001    0.001    6.635    6.635 post_process.py:45(prepare)
       1    0.000    0.000    6.539    6.539 post_process.py:664(build_underlying_lookup_dicts)
  145579    0.734    0.000    6.455    0.000 mappings.py:570(process_record)
       1    0.252    0.252    4.909    4.909 post_process.py:433(add_to_ins_lookup_dicts)
   48758    0.192    0.000    3.771    0.000 django/db/models/base.py:484(from_db)
    2971    0.256    0.000    3.742    0.001 post_process.py:119(post_process_record)
     289    0.003    0.000    3.627    0.013 django/db/backends/utils.py:58(execute)
     289    0.001    0.000    3.624    0.013 django/db/backends/mysql/base.py:137(execute)
     289    0.015    0.000    3.623    0.013 MySQLdb/cursors.py:141(execute)
     289    0.001    0.000    3.597    0.012 MySQLdb/cursors.py:326(_query)
   48764    2.317    0.000    3.581    0.000 base.py:388(__init__)
     217    0.003    0.000    3.445    0.016 compiler.py:814(execute_sql)
    2971    0.027    0.000    2.623    0.001 verifier.py:330(consume_record)
    2971    0.062    0.000    2.535    0.001 verifier.py:268(_write_record)
```

Figure 6.18. Parallel program `cProfile` output for a worker process of the medium dataset. The most interesting functions are highlighted in green. For the worker process, the functions related to row processing are more prominent: `process_record` (26%), `post_process_record` (24.8%), `consume_record` (8.7%). `_prepare`, called before the main row processing loop, takes up 22% of the total time.

```
         11506438 function calls (11444360 primitive calls) in 540.778 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000  541.087  541.087 importer.py:433(verify)
        1    0.000    0.000  541.051  541.051 importer.py:399(_do_verify)
        1    0.000    0.000  541.035  541.035 verifier.py:441(verify)
        1    0.007    0.007  540.650  540.650 verifier.py:546(do_verify_parallel)
        1    0.001    0.001  503.996  503.996 verifier.py:565(aggregate_result)
       16    0.000    0.000  503.910   31.494 multiprocessing/queues.py:113(get)
       16  503.907   31.494  503.910   31.494 method 'recv' of '_multiprocessing.Connection' objects
        1    0.293    0.293   35.875   35.875 parallel_post_process.py:8(post_process_parallel)
        1    0.005    0.005   35.554   35.554 parallel_post_process.py:26(find_duplicates)
      599    1.077    0.002   35.498    0.059 parallel_post_process.py:34(update_relation_duplicates)
   313427    1.485    0.000   18.981    0.000 cassandradataset.py:107(_get_many)
   312825    1.745    0.000   13.373    0.000 parallel_post_process.py:43(update_duplicates)
   312825    8.393    0.000    9.825    0.000 cassandradataset.py:123(_convert_record)
```

Figure 6.19. Parallel program `cProfile` output for the main process of the large dataset. The most interesting functions are highlighted in green. For the main process, the major functions are `aggregate_result` (93.0%) and `post_process_parallel` (6.5%).

```
         167668851 function calls (166172011 primitive calls) in 503.941 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000  503.866  503.866 verifier_parallel.py:61(start_pipeline)
        1    0.000    0.000  500.859  500.859 verifier_parallel.py:117(process)
        1    0.000    0.000  499.527  499.527 verifier_parallel.py:148(pipeline_process)
        1   42.686   42.686  499.526  499.526 pipeline.py:806(process)
  2770364    4.742    0.000  205.878    0.000 pipeline.py:834(process_record)
   164648    0.582    0.000  151.028    0.001 pipeline.py:838(post_process_record)
1611993/1610322    4.060    0.000  125.265    0.000 conditional.py:69(process_record)
  1961305    8.721    0.000  104.723    0.000 conditional.py:84(should_apply)
  1199331    7.147    0.000   91.551    0.000 mappings.py:570(process_record)
  1621266    6.765    0.000   64.816    0.000 conditional.py:376(apply)
    41162    4.293    0.000   60.395    0.001 post_process.py:119(post_process_record)
    41162    2.311    0.000   59.456    0.001 statistics.py:151(post_process_record)
  4885438   11.431    0.000   57.648    0.000 conditional.py:150(resolve_value)
  3684120    8.338    0.000   53.896    0.000 pipeline.py:655(get)
  5559170   22.559    0.000   51.305    0.000 pipeline.py:636(__getitem__)
6418/6413    0.248    0.000   45.797    0.007 query.py:972(_fetch_all)
    41162    0.398    0.000   43.858    0.001 verifier.py:330(consume_record)
      ...
        1    0.000    0.000    6.575    6.575 pipeline.py:794(_prepare)
```

Figure 6.20. Parallel program `cProfile` output for a slow worker process of the large dataset. The most interesting functions are highlighted in green. For the slow worker process, the functions related to row processing are prominent: `process_record` (40.7%), `post_process_record` (30%), `consume_record` (8.7%). `_prepare`, called before the main row processing loop, takes up 1.3% of the total time.

```
         156741758 function calls (155711119 primitive calls) in 413.059 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000  412.995  412.995 verifier_parallel.py:61(start_pipeline)
        1    0.000    0.000  409.467  409.467 verifier_parallel.py:117(process)
        1    0.000    0.000  409.270  409.270 verifier_parallel.py:148(pipeline_process)
        1   43.140   43.140  409.270  409.270 pipeline.py:806(process)
  2786468    4.883    0.000  199.673    0.000 pipeline.py:834(process_record)
1595097/1580961    4.254    0.000  115.514    0.000 conditional.py:69(process_record)
  1944409    8.840    0.000  106.480    0.000 conditional.py:84(should_apply)
   153836    0.496    0.000   90.622    0.001 pipeline.py:838(post_process_record)
  1257155    7.573    0.000   81.331    0.000 mappings.py:570(process_record)
  1653324    7.294    0.000   66.632    0.000 conditional.py:376(apply)
  4948465   11.699    0.000   59.407    0.000 conditional.py:150(resolve_value)
  3778004    8.740    0.000   56.046    0.000 pipeline.py:655(get)
  5659323   23.439    0.000   53.161    0.000 pipeline.py:636(__getitem__)
    38459    4.040    0.000   43.115    0.001 post_process.py:119(post_process_record)
    38459   15.826    0.000   27.382    0.001 verification.py:188(post_process_record)
   726861    3.687    0.000   24.766    0.000 conditional.py:246(apply)
   215845    0.826    0.000   23.247    0.000 mappings.py:281(get_transformed)
    38459    0.383    0.000   22.469    0.001 verifier.py:330(consume_record)
        ...
        1    0.000    0.000    6.712    6.712 pipeline.py:794(_prepare)
```

Figure 6.21. Parallel program `cProfile` output for a fast worker process of the large dataset. The most interesting functions are highlighted in green. For the fast worker process, the functions related to row processing are prominent: `process_record` (48.3%), `post_process_record` (21.8%), `consume_record` (5.4%). `_prepare`, called before the main row processing loop, takes up 1.6% of the total time.

6.7 Performance without sequential post processing

As the sequential profiling adds overhead, it is interesting to see what the speedup would have been if this extra step was not needed. These results are based in the profiling session above, conducted with 8 workers.

For the tiny dataset, the post processing adds 7.8% overhead. Subtracting 7.8% percent from the run with 8 workers results in a decrease from 1.74 s to 1.6 s. This means the "speedup" goes from 0.6X to 0.69X.

For the small dataset, the post processing adds 22% overhead. Subtracting 22% percent from the run with 8 workers results in a decrease from 7.21 s to 5.62 s. This means the speedup goes from 2.11X to 2.72X.

For the medium dataset, the post processing adds 7% overhead. Subtracting 7% percent from the run with 8 workers results in a decrease from 19.94 s to 18.54 s. This means the speedup goes from 3.76X to 4.04X.

For the large dataset, the post processing adds 6.5% overhead. Subtracting 6.5% percent from the run with 8 workers results in a decrease from 599.12 s to 560.065 s. This means the speedup goes from 4.3X to 4.6X.

6.8 Targeted memory profiling

In order to find the source of memory overhead, a simple targeted memory profiling was conducted by finding (using `resource`) and printing the current highest memory usage before and after some of the major functions in the program.

The targeted memory profiling for one of the worker processes in the medium dataset can be found in figure 6.22. This profiling shows that the function `build_underlying_lookup_dicts` is responsible for a clear majority of the memory usage. The function caches several mappings from the database in order to avoid round trips to the database every time one of these mappings is needed.

```
80995: Memory usage before filter.prepare for ServiceMapping: 19.193856
80995: Memory usage after filter.prepare for ServiceMapping: 19.193856
80995: Memory usage before filter.prepare for PostProcessFilter: 19.197952
80995: Memory usage before build_underlying_lookup_dicts: 20.779008
80995: Memory usage after build_underlying_lookup_dicts: 321.376256
80995: Memory usage after filter.prepare for PostProcessFilter: 321.376256
80995: Memory usage before filter.prepare for VerificationFilter: 321.376256
80995: Memory usage after filter.prepare for VerificationFilter: 321.376256
80995: Memory usage before filter.prepare for DeactivatedPartiesFilter: 321.376256
80995: Memory usage after filter.prepare for DeactivatedPartiesFilter: 321.376256
80995: Memory usage before filter.prepare for StatisticsCollectorFilter: 321.376256
80995: Memory usage after filter.prepare for StatisticsCollectorFilter: 321.376256
80995: Memory usage after prepare filters: 321.376256
80995: Memory usage after _prepare: 321.376256
80995: Memory usage after pipeline.process: 345.112576
```

Figure 6.22. Targeted profiling for the medium dataset. The interesting rows are highlighted in green. The number at the start of each row is the process id. This profiling shows that the function `build_underlying_lookup_dicts` is responsible for a clear majority of the memory usage. The function caches several mappings from the database in order to avoid round trips to the database every time one of these mappings is needed.

Chapter 7

Discussion & Conclusions

The results in the previous chapter are discussed below, in an effort to explain them. In addition, final conclusions about the thesis as a whole are drawn.

7.1 Dataset benchmarks discussion

7.1.1 Tiny dataset discussion

The tiny dataset shows poor results when parallelizing. For every worker number, only slowdown can be observed. The real time values for 10 and 12 workers show high standard deviations when comparing with the average value. It is conceivable that tasks such as creating processes have a more noticable impact for the low execution time.

The profiling session for the main process for the tiny dataset showed that `aggregate_result` takes up 70% of the time. Since this is the function that performs the main parallel processing of the dataset, about 30% of the execution time is overhead due to for example header detection and making trade ID:s unique. Since `aggregate_result` takes 0.4 seconds (15.7% of execution time) more to execute than it takes for a single worker to finish, it appears as though parallelization and aggregating the partial results result in noticable overhead for the tiny dataset. It is conceivable that the above results in the fact that only slowdown can be observed when parallelizing the tiny dataset.

Even without the parallel aggregation step, the tiny dataset still results in slowdown. A steady, close to linear increase in memory usage can be observed, resulting in total memory usage several times above what is used for the sequential program.

7.1.2 Small dataset discussion

For the small dataset, some speedup scaling can be observed, with the maximum value around 2.1. As more workers are added, speedup is increased, flattening out around 8 workers. This fits with the fact that the testing machine has 8 cores, effectively making it able to utilize true parallelism for a maximum of 8 workers.

The impact of adding more workers decreases with each one that is added. In the profiling session `post_process_parallel` is shown to take up a relatively large portion of the execution time (22.5%). Since this function is a separate, sequential and constant step, it is not affected by parallelization and therefore takes up varying percentages of the total time. Since more parallelization leads to faster execution of the parallel portion of the code, the post processing becomes more significant as workers are added, contributing to the fact that adding more workers results in less and less increase in speedup.

Without the post processing step, the small dataset gets slightly better speedup for 8 workers, 2.7X instead of 2.11X. This is still not close to the performance model calculations, which can be seen as disappointing. The memory usage grows in a similar manner to the tiny dataset, though the values are greater. Real time shows relatively low standard deviations compared to the total time, indicating that real time is fairly accurate for each worker value.

7.1.3 Medium dataset discussion

The medium shows greater speedup than the small dataset, with a similar shape to the speedup by worker curve. The maximum speedup is around 3.8, evening out around 8 workers, once again showing the best results around the machine's core number. This further suggests that larger individual workload, this time a result of the larger dataset size, results in better speedup. Real time again shows low standard deviation. Memory usage for the worker numbers with the largest speedup values is around 3 GB for dataset 3, demonstrating that a large price in memory usage is paid for parallelization, which may impact performance negatively for large dataset sizes and worker numbers.

The profiling session for the main process shows that `aggregate_result` (30.138 s) takes a similar amount of time to the execution of the worker processes (30.050 s). This suggests that parallelization overhead is not a major part of the execution time for the medium dataset. However, in the worker process the `process` function (handling the main dataset processing) takes up 28.802 s of the total 30.050 s that the worker takes to execute. This suggests a small overhead in the form of worker startup (4%). The post processing step takes up about 7.0% of the execution time for 8 workers. This is less than for the small dataset, but may still contribute to the fact that adding workers results in less increase in speedup.

Without the parallel processing, the medium dataset shows slightly better speedup for 8 workers. However, this increase is only from 3.76X to 4.04X speedup, meaning that it still has relatively low efficiency.

7.1.4 Large dataset discussion

The large dataset shows an even greater speedup, maintaining the trend of larger parallelization gains for larger datasets. Speedup for the large dataset increases past 8 workers, flattening out around 11 workers. From the profiling session, it is evident

that workers for the larger dataset may receive different workloads (taking between 412 seconds and 503 seconds) even though the chunks are the same size. This means that for 8 workers (the same as the number of cores), several cores are unused as the faster workers terminate and the program waits for the slow workers to terminate. By having more workers than cores, work can be more evenly scheduled among cores, resulting in better utilization of the processing power of the cores.

The profiling session also shows that `_prepare` takes up less than 2% of the processing time for the workers, meaning that workers for the large dataset spend more time in the main row processing loop, likely contributing to the fact that the larger dataset shows greater speedup.

The post processing step takes up about 6.5% of the execution time for 8 workers, close to the medium dataset. Without the parallel processing, the large dataset shows slighlty better speedup for 8 workers. However, this increase is only from 4.3X to 4.6X speedup, meaning that it still has relatively low efficiency.

7.1.5 General benchmark trends

In general, user time increases significantly with added workers, as is expected due to the greater total CPU utilization. Together with the fact that memory usage increases linearly with worker number, this indicates that a noteworthy amount of system resources is required for parallelization as dataset sizes and number of workers grow large. The highest amount of memory used by the program is 7.7 GB. While this is a substantial amount of memory, the benchmarking machine has 32 GB of memory, meaning that the memory usage should not impact performance in an all too significant way.

In general, larger datasets show greater speedup than smaller datasets.

7.1.6 Memory usage and caching discussion

The fact that memory usage increases as workers are added (though the overall problem size stays the same) can be explained by the fact that the `multiprocessing` module creates separate, entirely new processes. For each of these, the filters, mappings and other data relating to the transformation has to be stored in addition to the base memory footprint of the process.

As shown in section 6.8, the mappings cached by the post process filter take up a clear majority of the memory, and is substantial in size. The cache is in place in order to avoid many round trips to the database in order to get faster execution. However, due to the fact that the worker processes cannot share any data when using `multiprocessing`, the data is duplicated across the processes, resulting in a net increase in memory usage. For larger datasets, this increase in memory usage is very noticable, and therefore a significant price to pay for speedup.

7.1.7 Ethics and sustainable development

Faster processing of financial data could have a small but positive effect on society. Since faster processing facilitates faster response cycles for the customers (banks), errors can be spotted sooner and bottlenecks in the customer's organization avoided. This can also reduce frustration for the customer. As the banks perform services that greatly affect society, society can benefit (at least slightly) from these positive effects of faster processing. In addition, if the datasets are processed faster through the use of multiple cores, the time until the cores are put in the power-saving sleep mode is smaller. In today's world, where resources are depleted faster than they are replenished, saving power is an important goal.

7.2 Conclusions

This section outlines the final conclusions drawn from this thesis.

7.2.1 Main conclusions

Using Python `multiprocessing` for parallelization resulted in true parallel speedup, but was not without issues. Sharing data using only message passing results in relatively safe and readable code since excessive sharing of data is avoided. However, in the transformation program parallelized in this thesis, the fact that the processing pipeline including column mappings needed to be stored for each worker resulted in more overhead both regarding worker startup and memory consumption. This leads to the conclusion that users of multiprocessing need to be wary not only of communication and creation overhead associated with processes (as opposed to threads), but also of overhead from worker startup and data duplication as a result of the message passing model.

The transformation problem in this thesis, and parallel programs with expensive worker startup, are heavily influenced by the size of the data. This means that developers faced with implementing parallelization of similar systems should examine the data in their problem domain in order to find an initial indication of whether the size of the datasets are, in general, large enough to benefit from parallelization.

While workload may appear to be the same when splitting a dataset into equal parts, it may still be beneficial to examine if the workers take different times to finish, in order to avoid underused cores. Using more tasks than cores for larger datasets can increase speedup in these cases.

When parallelizing a complex system, as few assumptions as possible should be made before starting. In this thesis, the problem involves I/O and database communication, suggesting that the problem may be I/O bound. However, further investigations proved that the problem was largely CPU bound, making it suitable for `multiprocessing` in combination with a multicore machine. Investigation, rather than assumption, are helpful when parallelizing a complex program involving both CPU and I/O tasks.

The method used in this thesis for finding parallelizability was relatively successful. When analyzing file formats for parallelizability, inherently serial, extra overhead, and embarrassingly parallel formats were found. The implementation then focused on parallelizing the extra overhead and embarrassingly parallel formats. This method could possibly be generalized to other parallelization problems in complex systems, by identifying a subsystem that is easily parallelizable and focusing on that part rather than the system as a whole. The subsystem might be either a subset of the code or of the datasets in the problem domain. In this thesis, the subsystem proved to be large enough that the effort of parallelization was worthwhile, which is an analysis that should be made also in the general case of parallelizing complex systems.

Developers should be wary of aggregation when parallelizing systems. In this thesis, aggregation manifested itself as making trade ID:s unique, which was done by storing encountered ID:s. While storing state in this manner may appear as making rows dependant on each other, it proved possible to extract the aggregation into a post-processing step. This extra step introduced overhead, but parallel speedup could still be observed for larger datasets. In conclusion, implementers of parallelization should look for aggregation in their system, conclude if this can be extracted into a post processing step, and if this post processing step is small enough for parallelization to be beneficial. Examining if the post processing step takes up a smaller part of the total execution time for larger datasets can also be of interest.

7.2.2 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 was written. Another delimitation of the thesis is that it does not compare different methods of parallelization, and uses only the Python `multiprocessing` module.

7.2.3 Future work

This thesis focused on parallelization on a single computer. Since `multiprocessing` uses a message passing approach with serialized data, it is conceivable that a future work in a distributed approach is interesting for the dataset transformation problem for larger datasets. Also, conducting experiments on even larger datasets may be of value as the trend points to greater speedup the larger the dataset. Another interesting future work would be to implement a mechanism that selects a suitable parallelization strategy given the size of a dataset and its file format, with the help of the results in this thesis.

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] *26.4. The Python Profilers — Python 2.7.11 documentation*. URL: https://docs.python.org/2/library/profile.html (visited on 05/18/2016).

[4] *36.13. resource — Resource usage information — Python 2.7.11 documentation*. URL: https://docs.python.org/2/library/resource.html (visited on 05/18/2016).

[5] 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.

[6] 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.

[7] 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).

[8] 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).

[9] 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).

[10] 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).

[11] 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).

[12] 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).

[13] 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).

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

[15] 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).

[16] 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).

[17] 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).

[18] 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).

[19] 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.

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

[21] 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.

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

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

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

[25] 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).

[26] 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).

[27] 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.

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

[29] 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).