Parallel Query Evaluation in Streaming Environments

Murray Steele

Master of Informatics
School of Informatics
University of Edinburgh

Academic Year 2019/2020

Abstract

With single-threaded performance stagnating, processor manufacturers in recent years have opted to increase the number of cores for their devices. To improve the performance of existing software, developers can no longer rely on improvements in single-thread performance and instead must focus on parallel computing instead.

Data-parallelism – the processing of data chunks in parallel – has seen significant growth in automation over the past two decades and research in the area as produced novel and performant techniques available through a variety of compilers and tools. Task-parallelism – the execution of separate code sections in parallel – has seen less exploration on the subject of automation, and practical implementations are lacking. Automatic task-parallelization presents many difficult to solve problems, which are amplified by modern programming language features such as aliasing and recursion.

This report discusses existing literature in the area of automatic parallelization, and discusses methods for automatic task-parallelization and optimisation of a simple language – M3 – produced by DBToaster – a novel SQL-to-native-code compiler. This work extends DBToaster to provide an interface for automatic parallelization using OpenMP – a popular parallel programming API – and evaluates the affect of parallelization on the runtime of generated code.

Acknowledgements

I would like to thank my supervisor Milos Nikolic for all his help and support throughout the duration of the project. Without his insight the project would have been significantly harder.

Table of Contents

Chapter 1

Introduction

This report discusses existing literature in the area of automatic parallelization and presents methodologies and optimisations for automatic task-parallelization of M3, the intermediate representation language for DBToaster [8], a novel SQL-to-native-code compiler. The implementation of the presented methods and optimisations extends DBToaster with an interface for automatic parallelization of emitted C++ code using the API provided by OpenMP. The report evaluates the effect of automatic parallelization and optimisation on the runtime of the program produced from a varied query set.

1.1 Motivation

For several decades, Moore's law – a prediction that the number of transistors on integrated circuits will double every two years – has continued, and has been correlated with unparalleled growth in single-threaded performance. In recent years, Moore's law has continued, yet single threaded performance has stagnated. Shown by figure 1.1, the number of logical cores supported by microprocessors in the past decade has increased rapidly rather than the performance of single core processors.

Instead of relying on higher clock speeds and greater instruction parallelism, multi-core computers rely on the programmer to split their programs into separate computations which may be executed in parallel. This introduces many new types of bugs and errors which are difficult to diagnose, as well as performance trade-offs which involve choices to be made about whether code can, or should, be parallelized.

For these reasons it is important to build tools which automate the parallelization process. Furthermore, DBToaster – a compiler – benefits from automatic parallelization as it allows generated code to better utilise present and future hardware, potentially improving performance of emitted programs without requiring additional work by the user.

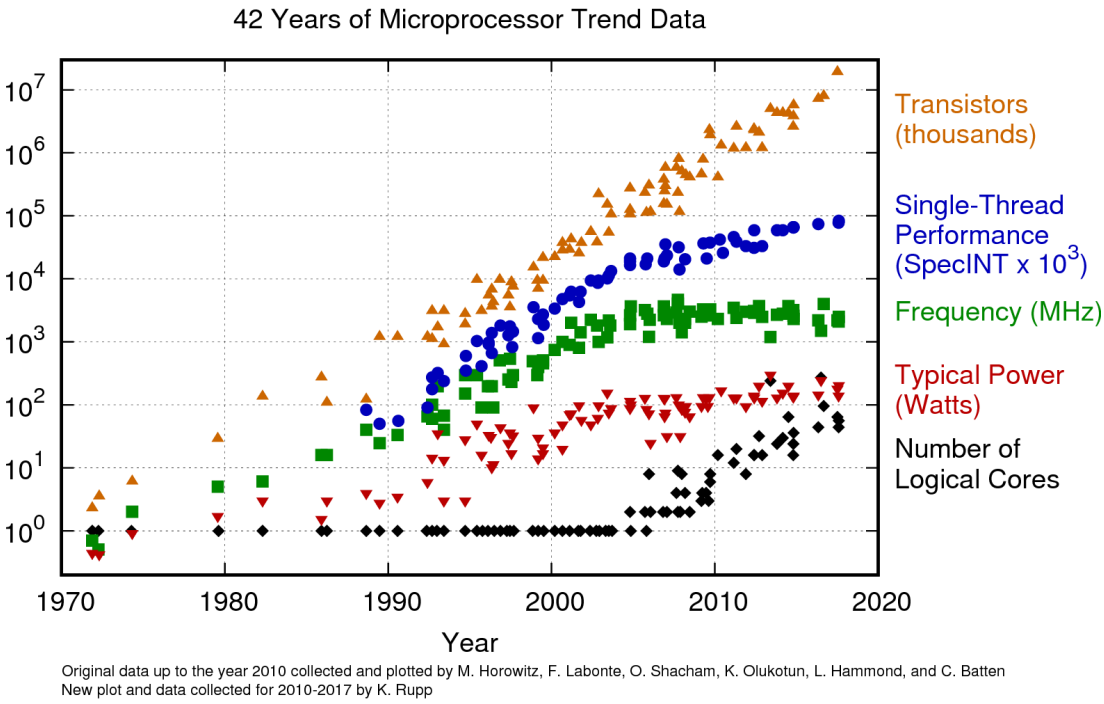


Figure 1.1: Trends in microprocessor design [20]

1.2 Goals and contributions

The end goal of this project was to enable automatic parallel code generation of query evaluation for DBToaster programs using task-based parallelization methods, and to show the effect of parallelization for a varied query set, as to evaluate its applicability.

The core sub-goals of this project were:

1. To present approaches for automated task-based parallelization of M3
2. To present optimisation methods for parallelized M3
3. To implement the presented parallelization and optimisation methods in DBToaster, and provide a user interface for automatic parallelization
4. To evaluate effect of parallelization and optimisations on the runtime of input queries

All goals have been achieved over the course of the project, during which time extensions to the project have been identified for the coming year.

1.3 Overview and organisation

Excluding the introduction, this report is organised into 5 chapters:

2. Literature and Technology Review: This chapter reviews and discusses literature and technology – and a lack thereof – related to the project. This chapter also discusses prerequisite knowledge, definitions, and terminology used throughout

the report.

3. Methods: This chapter presents assumptions for task-based parallelization, strategies for parallelization, and optimisations of an intermediate representation to aid parallelization.
4. Implementation: This chapter discusses the implementation of the strategies and optimisations presented in the previous chapter, and discusses choices made for the implementation.
5. Evaluation: This chapter evaluates the performance of sequential and parallel code generated by DBToaster and the implementation of automation of task-based parallelization. The evaluation focuses on runtime performance as well as the effectiveness of optimisations. It discusses the merits of automatic parallelization, and its shortcomings.
6. Conclusion: This chapter discusses the findings presented in this report, and future extensions to the project which may help to solve the shortcomings of the presented parallelization strategies and optimisations found during evaluation.

Chapter 2

Literature and Technology Review

This chapter discusses literature in the subject area, and the existing technologies relevant to, or used within, this project.

2.1 Overview

Parallelization is an extensive and active area of research which has produced well known, and impactful techniques for performance. There are two core parallelism paradigms: data-parallelism, and task-parallelism. Data-parallelism has been a popular topic of compiler research, and many varied techniques for automatic data-parallelization of code are now available. Research on task-parallelism is not as popular, and is mostly limited to related or prerequisite problems.

2.2 Data Parallelism, Parallelization, and Applications

Data-parallelism is a paradigm which aims to split data into discrete chunks which are processed identically in parallel. For programs which process a large amount of data, this paradigm is extremely effective, the time taken to process data can simply be divided by the number of threads used to process the data, and often results in predictable performance improvements. Applications of data-parallelism are often related to big-data, machine learning, and distributed computing [16] [21].

This concept has seen applications in hardware, software, and distributed systems. At the hardware level, single-instruction-multiple-data (SIMD) architectures provide instructions for operating on large chunks of data of a fixed size, enabling data-parallelism for single-threaded applications [13]. At the software level, parallel programming APIs and libraries such as OpenMP [18] and Intel's Threading Building Blocks (TBB) [10], enable programmers to parallelize code using compiler supported – or library provided – constructs. For distributed systems, techniques such as MapReduce [11] have enabled fault tolerant data-parallelism for distributed systems.

Automatic data-parallelization is a feature supported by several compilers and tools, predominantly for C++ and FORTRAN. Intel's commericial C++ and FORTRAN compilers support automatic data-parallelization for data processing loops [9], which may be used in conjunction with manually written parallel code, or be used to guide the writing of parallel code. More recent research in the area of the polyhedral compilation – an approach to compilation which relies on the representation of programs to exploit combinatorial and geometrical optimizations [3] – has lead to projects such as APOLLO which enables sopisticated parallelization of complex nested loops [22] [5].

In addition to automatic parallelization, several tools exist to assist parallelization by suggesting changes to source code. One such tool is iPat/OMP [15], a tool which highlights areas of source code which can be exploited for data-parallelism and can automatically parallelize highlighted areas using OpenMP constructs.

2.3 Task Parallelism, Parallelization, and Applications

Task-parallelism is a more general paradigm of parallelization which aims to split code into discrete tasks which are executed in parallel. In recent years, parallel programming APIs such as OpenMP and TBB have improved support for task-based parallelism, but practical implementations of automatic task-parallelization are still lacking.

Task-parallelism presents very different problems than data-parallelism, many of which are still being worked on. One such problem is the extraction of data dependencies from source code. Many modern programming languages support features such as aliasing which makes practical analysis of data dependencies extremely difficult. A large variety of approached have been applied to the area, but the problem is not yet fully solved. Another problem which plagues task-parallelization is that parallel constructs have overheads, which means that in some cases it is not worthwhile to parallelize parts of a program. This is a difficult problem as it relies on runtime characteristics of programs which are not available at compile-time.

Automatic task-parallelization does not currently have any popular real-world implementations, but manual task-parallelism is used in several real world systems such as Apache Storm [14] which lets users design pipelines for data which may result in distinct tasks executing in parallel to process data.

In addition, the term *task* is often used in literature to refer to the separate but identical computations on distinct chunks of data which are a feature of data-parallelism, which can make the subject difficult to navigate. For this report, the term *task* will be used to refer to explicitly defined code sections which may be executed in parallel.

2.4 DBToaster

DBToaster is a novel SQL-to-native-code compiler, capable of producing C++ or Scala programs which implement SQL queries and schema [8]. DBToaster generates lightweight

query engines which may be embedded into existing programs, or which new programs may be built around. Unlike typical database management systems, DBToaster-generated programs feature incremental updates which allows for only partial evaluation of input queries to reduce latency with which their answers are availiable.

2.5 Updates

Programs generated by DBToaster incrementally update the answers to their associated queries – also known as views – over time with each update to schema relations. Updates may either insert or delete tuples from relations, and may be handled using one of two methods: single-tuple updates, or batch updates. For single-tuple updates, queries are reevaluated for each individual update to schema relations. For batch mode, queries are evaluated for each batch of updates of a specified size. Single-tuple updates and batch updates result in different programs which may have different runtimes and resource usage. Both update modes will be considered throughout this report, but only insertion updates will be considered, though deletion updates can be parallelized using the same methods.‘

2.6 M3

DBToaster converts input SQL queries into an internal representation language called M3. An M3 system consists of 5 parts: type definitions, sources, maps, queries, and triggers.

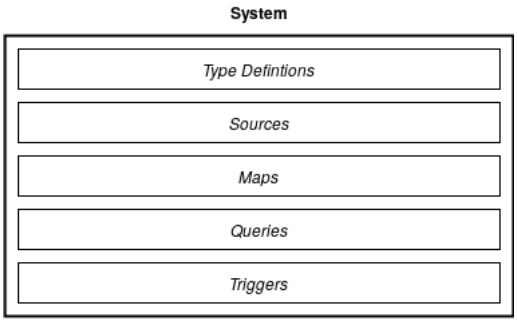


Figure 2.1: M3 system structure

This project only considers the *triggers* of M3 systems, which describe computations required for query evaluation. Each trigger is associated with a relation specified in the schema used in the generation of their M3 system, and handles query evaluation for a specific update on that relation. For single-tuple updates, triggers are generated from insertions and deletions on schema relations, and for batch updates, triggers are only generated for handling batches of mixed insertion and deletions.

How a trigger evaluates each query is described by an associated ordered list of statements. Each statement is associated with an aggregate or relation which they make

some modification to. These relations are either the result of a query, or they are an intermediary result which are used to compute the result of a query. M3 does not feature any kind of recursion, aliasing, or control statements which make task-parallelization difficult, making it a good environment for demonstrating task-parallelization techniques.

Rather than discussing M3 with respect to its specific syntax, expressions, and statements, this report will discuss the structure and data dependencies of M3, which may be exploited for automatic parallelization.

2.7 Dependency Graphs

Dependency graphs are a representation of computer programs which do not specify a strict order of operations, but instead partially order statements using two types of dependencies: *in-dependencies*, *out-dependencies* – also known as read-dependencies and write-dependencies. An *in-dependency* is a dependency on data as a result of the associated statement using, but not modifying, the data. An *out-dependency* is a dependency on data as a result of the associated statement modifying the data. For a pair of statements `(A, B)`. `B` is said to be *dependent* on `A`, and `A` is said to be a *dependency* of `B` if `B` comes after `A` in the original statement execution order, and one of the following conditions is true for some data `C`:

1. One statement declares `C` as an *in-dependency*, and the other statement declares `C` as an *out-dependency*.
2. Both statements declare `C` as an *out-dependency*.

Statement `B` is referred to as a *direct dependent* of statement `A` if there does not exist a statement `D` which is a *dependency* of statement `B` and a *dependent* of statement `A`. Statements `A` and `B` are said to be independent if neither is a *dependent* of the other.

Given these dependencies, a program can be executed in any order which allows every statement to be executed prior to their *dependants*. This is referred to as a partial order, as multiple orders of execution may be derived from it.

Representing a program as a dependency graph is already a technique used in automatic parallelization [1]; their merit is that they identify *independent* statements which may be executed in parallel. For this reason, dependency graphs are the representaton used by methods and optimisations presented in later chapters.

2.8 OpenMP

OpenMP is a parallel programming API which is supported by a variety of compilers [19]. OpenMP makes parallelization of existing C++ code easy by providing its interface through pragma directives which act as signals to a pre-processor. This means that existing code often does not need to be changed drastically to enable paralleliza-

tion [17]. For these reasons, it was decided that OpenMP would be used in the implementation of this project. Because of this, assumptions relating to behaviours and overheads of parallel execution are based on the behaviour and overheads of OpenMP. It is also important to understand the model and constructs OpenMP supports to understand some of the methods and implementation details discussed in this report.

2.8.1 Fork-Join Model

OpenMP implements a fork-join model. This is a model of parallel computation which assumes that code is sequentially executed by a *main* thread until it meets a region which is designated as being executed in parallel. When the main thread meets such a region, it *forks* into several separate threads which independently execute the region. Once all threads have completed the region, they *join* into the main thread again, which continues to execute the program sequentially.

The fork-join model is extended further by OpenMP's additional constructs which allow for fully fledged task-parallelism.

2.8.2 Relevant Constructs

OpenMP supports a large number of constructs for implementing data-parallelism or task parallelism. Only a few constructs are important for this project, and this section will cover these in sufficient detail. Far greater detail is available in the specifications published by OpenMP [4].

The `parallel` construct defines a parallel execution region where multiple threads will execute the code inside the region. When the main thread encounters a parallel region it creates a team – a group of other threads which may contribute to execution of the parallel region and other OpenMP constructs inside. By default all threads in the team will execute the code inside the region independently. A parallel region implements an implicit barrier at its end, where all threads part of the associated team are synchronised.

The `single` construct defines a region inside a parallel region which only a single thread may execute. The single construct also implements a barrier at its end, which may be disabled by adding a `nowait` clause to the construct.

The `task` construct defines a code region which is allocated to a thread in the team for execution, depending on whether there are any threads which are not busy. If no threads are available the task is deferred. Since version 4 of OpenMP, tasks support a `depend` clause in which data dependencies can be declared. Data dependencies are either in-dependencies, which refer to data which the task uses but does not modify, or out-dependencies, which refer to data which the task modifies. A task's in-dependencies are satisfied when all previous tasks which declare the data as an out-dependency are complete, and a task's out-dependencies are satisfied when all previous tasks which declare the data as an in-dependency or out-dependency are complete. Once all of a

task's dependency are satisfied, it may be executed. In what order tasks with shared dependencies are executed is determined by their ordering in a parallel region.

A combination of the `parallel`, `single`, and `task` constructs allow us to create a region where a single thread collects tasks and allocates them to a team of threads. This is the method of task-parallelization that this project utilises.

The `default` clause may be added to the `parallel` and `task` to specify a default method of data-sharing. This project only makes use of the `shared` method to allow separate tasks to share the same data, and to avoid copying relations between tasks.

2.9 Overheads

OpenMP constructs have associated overheads which are unavoidable for task-parallelization. These overheads can depend on hardware, the number of threads allocated to parallel regions, and the specific constructs used within them. Using the micro-benchmark suite provided by EPCC [12], we found that the overhead for a single OpenMP task is not trivial, and scales non-linearly with the number of threads allocated. The results from the micro-benchmark is shown by figure 2.2, was compiled using GCC, and executed on a system with a 40-core Intel Xeon E5-2690 processor.

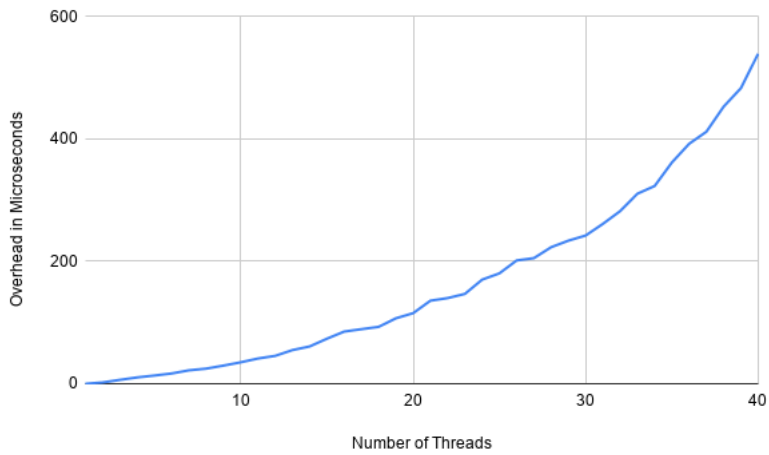


Figure 2.2: Overhead for a single OpenMP task with different thread allocations.

Chapter 3

Methods

Being a topic with a lack of existing technologies, automatic task-parallelization lacks documentation of practical approaches. This chapter presents two possible approaches to automatic task-parallelization of M3, and explores optimisations based on the structure of data-dependencies between M3 statements, and an approximation of statement runtime complexity.

3.1 Task Overhead Assumption

The methods presented in this chapter base the behaviour and attributes of parallel constructs off of OpenMP. The core assumption of the presented methods is that tasks each have some significant associated overhead. This is a reasonable assumption based on the results of the EPCC micro-benchmarks from earlier, and means that for some statements, task overhead will be greater than runtime of the actual statement.

3.2 Task-Parallelization Strategies

By default, M3 statements are translated into native code which is executed sequentially. However, by first transforming a trigger's M3 statements into a dependency graph, we can identify statements which are obviously independent, and could be executed in parallel. An example of a possible dependency graph is shown in figure 3.1 3.1.

In figure 3.1, a directed edge from statement `1` to statement `2` means that statement `2` is dependent on statement `1` and therefore statement `1` must be executed prior to statement `2`. The partial ordering given by the dependency graph also features several pairs of statements which may reordered or executed in parallel. Statement `2` and/or statement `4` may be executed in parallel with statement `3`, or statements `1` through `4` could be executed in parallel with statements `5` and `6`. These pairing expose two task-parallelization strategies: executing independent single statements in parallel, and

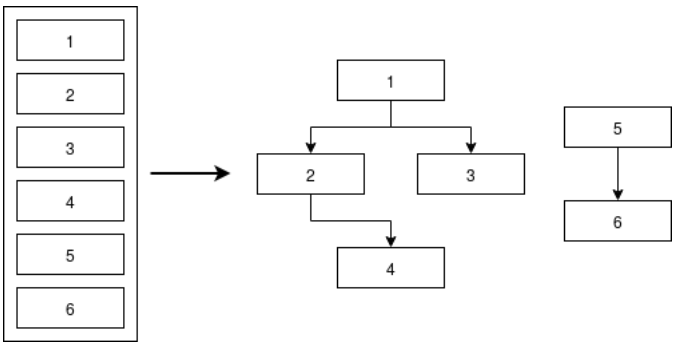


Figure 3.1: Example of dependency graph extracted from M3 statements.

execution of independent groups of statements in parallel. These are referred to as fine-grained parallelization and coarse-grained parallelization respectively.

3.2.1 Fine-Grained Parallelization

Fine-grained parallelization is a strategy for task-parallelization which maximises parallelization by assigning individual statements to tasks which may be executed as soon as soon as their dependencies are satisfied. An example of task allocation for a dependency graph is shown in figure 3.2.

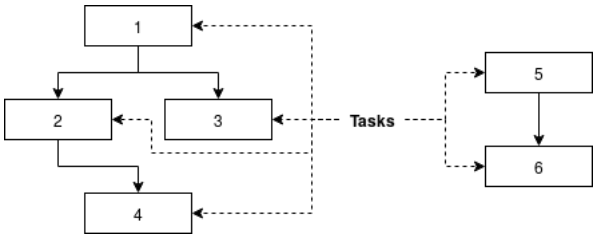


Figure 3.2: Fine-grained parallelization of an example dependency graph.

Fine-grained parallelization creates many small tasks, which should result in good thread utilisation as a computation will be more evenly shared between tasks than a strategy that opts to assign multiple statements to the same task. However, because each task is assumed to add some overhead, having a large number of tasks may be a disadvantage if individual statements are trivial to compute. Also, because every statements is not necessarily independent of all other statements, it is reasonable to assume that additional overhead will be introduced through tasks dependency management. Optimisations to mitigate overheads are discussed later in section 3.3.

3.2.2 Coarse-Grained Parallelization

Coarse-grained parallelization is a strategy for task-parallelization which maximises the size of tasks by assigning independent groups of statements to single tasks such that each task is executed independently of one another. Independent statement groups are identified as disconnected sub-graphs of a dependency graph.

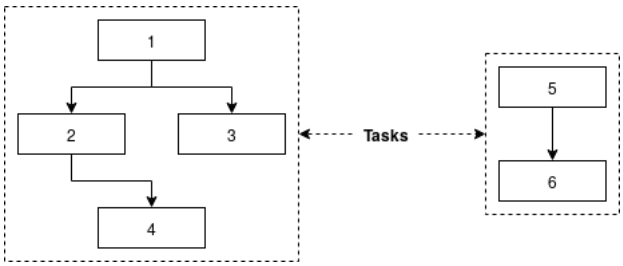


Figure 3.3: Coarse-grained parallelization of an example dependency graph.

Coarse-grained parallelization results in a few large tasks, rather than many smaller tasks, which may mean that threads are not utilised well if the workload of each task is not well balanced. However, because tasks are necessarily independent, coarse-grain parallelization guarantees that there are no dependencies between tasks. A lack of dependencies and a lower number of tasks should result in lower overhead incurred by tasks.

One major disadvantage of this approach is that M3 dependency graphs may not consist of several disconnected sub-graphs, but rather a single connected graph. Optimisations capable to removing statements from dependency graphs, potentially mitigating this problem, are discussed later in section 3.3.

3.3 Optimisations

Already, several potential problems have been highlighted for the parallelization methods presented. This section presents optimisations for dependency graphs which aim to reduce task overhead by fusing statements or by simply moving statements outside of the dependency graph to improve or limit parallelization.

3.3.1 Optimised Code Structure

Until now, we have assumed that all statements will be involved in parallelization, or no statements will be parallelized. Optimisations introduce a hybrid approach in which some statements may be executed sequentially before or after tasks are executed. Statements executed sequentially prior to parallelized code are referred to as *pre-statements*, and statements executed sequentially following parallelized code are referred to as *post-statements*. This terminology will help better explain how optimisations work, and the resulting structure of after optimisations are applied.

3.3.2 Structural Optimisations

Dependency graphs sometimes include structural features which mean that extra parallelization overheads are incurred, or, in the case of coarse-parallelization, parallelization cannot occur. This section discusses optimisation methods based on the structure of dependency graphs, for reducing the number of tasks involved in a parallel computation, therefore reducing the total number of dependencies between tasks, and the

total overhead introduced by tasks.

3.3.2.1 Trim-Sequential

The trim-sequential optimisation targets statements which constrain the start or end of a parallel region because they are cannot be executed in parallel with any other statements and therefore add additional task overhead when using fine-grain parallelization. The same statements may constrain a dependency graph such that removing it will allow coarse-grain parallelization to identify more disconnected sub-graphs.

This optimisation iterates over statements in a dependency graph; if a statement is a dependency of all other statements, it is removed from the dependency graph and appended to the list of pre-statements. Otherwise, if the statement is a dependent of all other statements, it is removed from the dependency graph and prepended to the list of post-statements.

An example of the trim-sequential optimisation can be seen in figure 3.4.

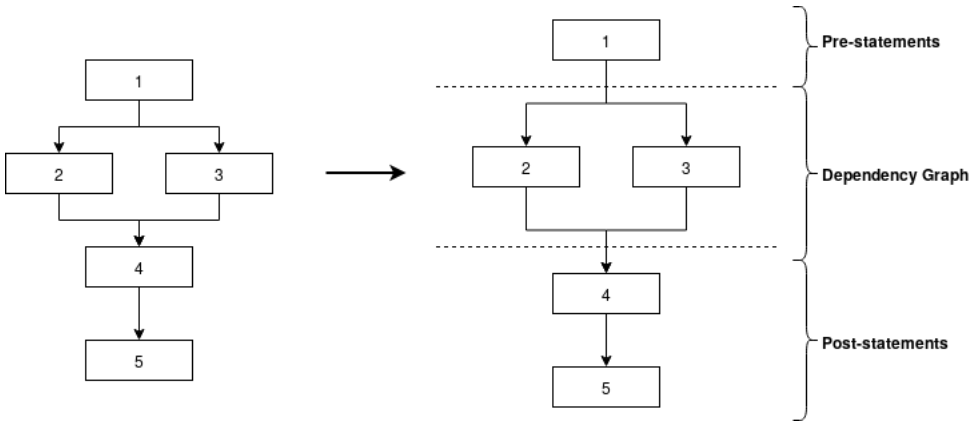


Figure 3.4: Trim-sequential optimisation applied to an example dependency graph.

3.3.2.2 Fuse-Sequential

The fuse-sequential optimisation targets pairs on statements with can be fused into a single statement to reduce the number of statements without changing the order of execution for the two statements. This reduces task-incurred overhead for fine-grained parallelization.

This optimisation iterates over pairs of statements in the dependency graph. For a pair of statements, `(A, B)`, if statement `A` is the only direct dependency of statement `(B)`, and statement `B` is the only direct dependent of statement `A`, then statements `A` and `B` can be fused into a single statement, which replaces the two statements, and inherits the dependencies and dependants of both statements. An example of the fuse-sequential optimisation is shown by figure 3.5.

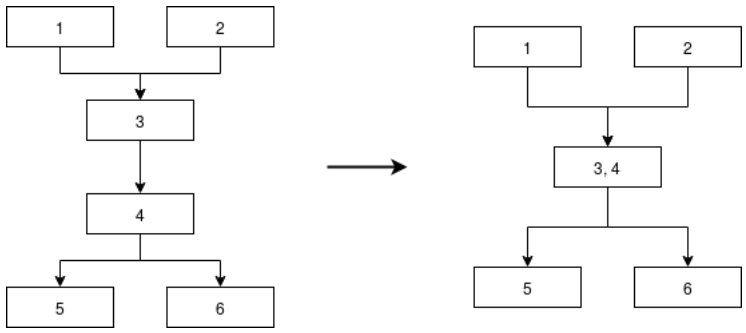


Figure 3.5: Fuse-sequential optimisation applied to an example dependency graph.

3.3.3 Complexity Optimisations

Because of the overhead which tasks add to a computation, it is not always worthwhile to parallelize all statements. Some statements result in native code whose runtime is less than that of the overhead added by executing it as a task. This section presents a heuristic for approximating the *complexity* of M3 statements , enabling the optimisations presented in this section.

3.3.3.1 Complexity Heuristic

M3 supports a wide variety of expressions, used to implement statements, which range widely in runtime complexity. Many expressions generate trivial code which runs in constant, or near-constant time. Such code may check conditions, or the existence of tuples in relations. A few expressions result in native code which runs in linear or polynomial time. These expressions' runtime are related to the size of relations, which tend to grow over time during execution.

A statement which contains a nontrivial expression is referred to as a nontrivial statement, and is a prime candidate for parallel execution with other nontrivial statements whose runtime can increase greatly over the course of a program's execution. Statements which do not contain a nontrivial expression are known as trivial statements, and do not benefit from parallelization as their sequential runtime is assumed to be less than, or equal to the overheads incurred by executing them as a task.

The AggSum expression – an M3 expressions which represents iterations over relations – is a nontrivial expression for which DBToaster generates loops over their associated relation. Our heuristic takes this into account and defines nontrivial statements as those that include an AggSum expression, and all other statements are classified as trivial. This heuristic highlights statements which are highly likely to benefit from parallelization, but this is not always the case. This is discussed at length in chapter 5.

3.3.3.2 Trim-Trivials

The trim-trivials optimisation removes trivial statements with either no dependencies or no dependants from a M3 statement dependency graph, and moves them into the pre-statement and post-statements lists instead where they are sequentially executed. This

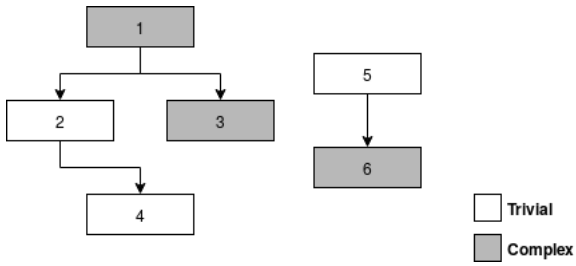


Figure 3.6: Example of classification of statements using complexity heuristic

reduces task-incurred overhead for fine-grained parallelization, and can potentially remove statements which prevent a dependency graph from being split into disconnected sub-graphs, which is useful for coarse-grained parallelization.

The optimisation iterates over trivial statements in the dependency graph. If the chosen statement has no dependants, then it is removed from the dependency graph and prepended to the list of post-statements. Otherwise, if the chosen statement has no dependencies, then it is removed from the dependency graph and append to the list of pre-statements. An example of the trim-trivials optimisation on a dependency graph is shown in figure 3.7.

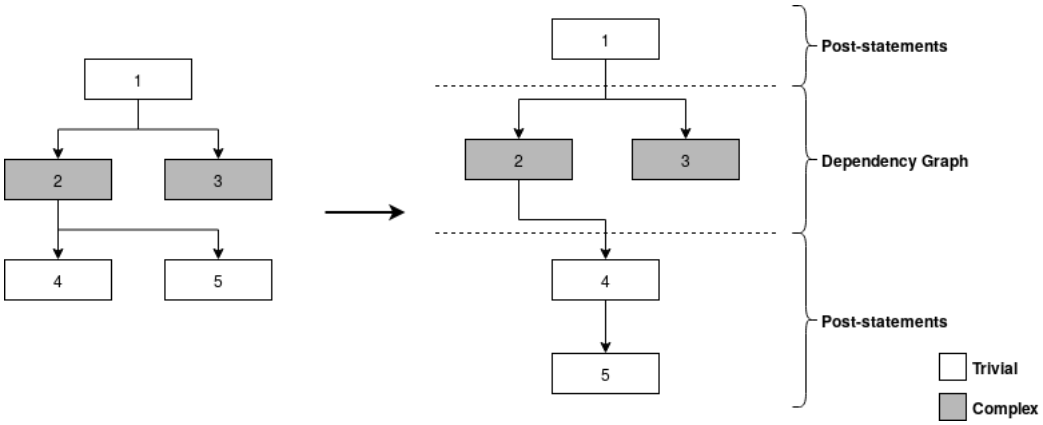


Figure 3.7: Trim-trivials optimisation applied to an example dependency graph.

3.3.3.3 Fuse-Trivials

The fuse-trivials optimisation targets trivial statements, and fuses them into other statements. This reduces task-incurred overhead for fine-grained parallelization, and may enable other optimisations. Fuse-trivials reduces the total number of statements in a dependency graph, and reduces the complexity of a dependency graph's structure which may allow other optimisations to be applied. This optimisation also have the advantage of reducing the complexity of a dependency graph, and the number of statements, without removing any statements from the computation.

This optimisation exploits the assumption of trivial statements – that their sequential runtime is less than or equal to the overhead incurred by executing them as task. Given this assumption, there is little performance difference in executing two independent

trivial tasks in parallel, or sequentially, as task overhead negates the benefits of parallel execution. Additionally, for the same reasons, there is little performance difference in executing a pair of independent nontrivial and trivial tasks in parallel, or sequentially. With this information, a set of trivial statements which share a single direct dependent or dependency, can be restructured into a sequential list of statements which can then be merged into their direct dependency or dependent.

Fuse-trivials iterates over pairs of statements in the dependency graph. For a pair of statements, `(A, B)`, if statement `A` is trivial, and statement `B` is the only dependent of statement `A`, then fuse the two statements into a new statement which inherits the dependencies of the two statements. Otherwise, if statement `B` is trivial, and statement `A` is the only dependency of statement `B`, then fuse the two statements into a new statement which inherits the dependencies of the two statements. An example of this optimisation on a dependency graph is shown in figure 3.8

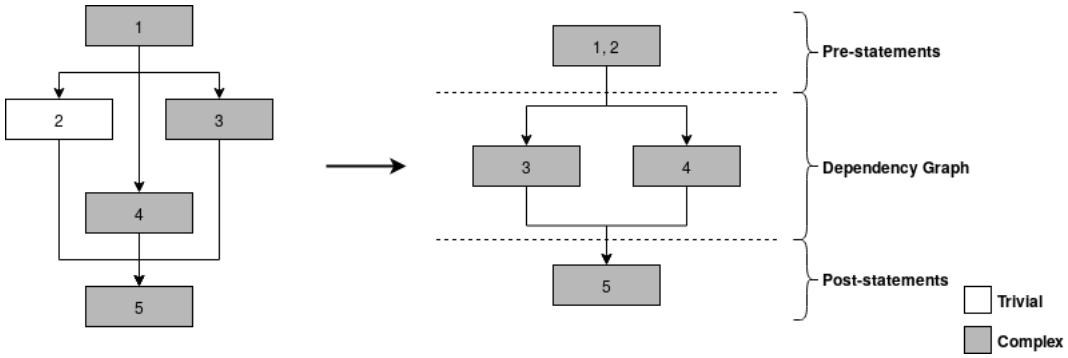


Figure 3.8: Fuse trivials optimisation on example dependency graph

Chapter 4

Implementation

This chapter discusses the implementation of the automatic parallelization strategies and optimisations discussed previously within DBToaster's architecture. It also discusses the extension of DBToaster's command-line interface, and extra features which are available to aid with parallelization.

4.1 Architecture

DBToaster consists of two parts: a frontend, and a backend. The frontend is responsible for translating SQL into an internal calculus and applying various optimisations. Once optimised, the calculus is transformed into M3 which is given to the backend as input; then the backend takes M3 and translates it to a language specified by the user [6]. The full architecture without additional parallelization stages is shown by figure 4.1.

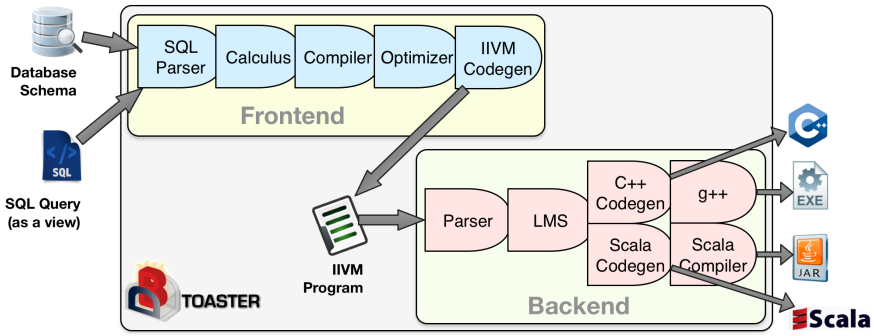


Figure 4.1: Original DBToaster Architecture [6].

This project modifies DBToaster's backend with several parallelization stages prior to C++ code generation which is enabled by extensions to DBToaster's command-line interface. Firstly, each trigger function's statements are transformed into a dependency graph, then this graph is passed through an optimisation stage which applies optimisations presented earlier, if enabled. The optimised dependency graph is passed

to a parallelizer for coarse-grained or fine-grained parallelization, where the dependency graph is converted into a parallelized M3 program. The new architecture for DBToaster's backend is shown in figure 4.2.

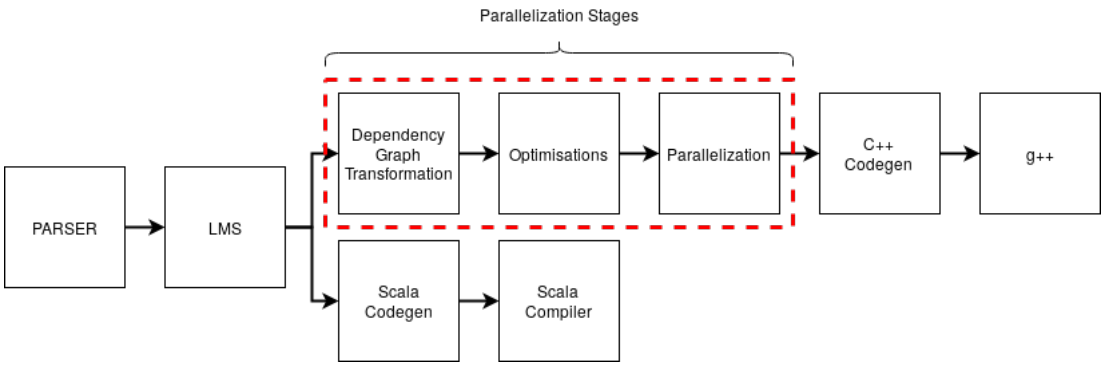


Figure 4.2: Modified DBToaster backend architecture.

4.2 Parallelization

Parallelization is implemented using two classes which implement the two parallelization strategies presented in chapter 3, and a super-class of the two which contains shared code for dependency graph construction, optimisations, and construction of supporting M3 statements.

The implementation uses the following operation pipeline for parallelization, with M3 emitted as input:

1. Each trigger's statements are converted into a dependency graph.
2. User-selected optimisations are applied to each dependency graph.
3. The optimised dependency graphs are parallelized using the strategy selected by the user, resulting in a new list of parallelized statements.

When the user selects the fine-grained parallelization strategy, each statement in a dependency graph is transformed into a distinct task. Tasks are then ordered such that each task is executed prior to its dependants.

When the user selects the coarse-grained parallelization strategy, disconnected subgraphs are extracted from a dependency graph, then each is fused into a single statement and transformed into a distinct task.

Finally, to avoid redundant parallelization, if a parallelization strategy only creates a single task for a given trigger, then that trigger is not parallelized.

4.3 Supporting Statements

The implementation of methods presented in chapter 3 require the addition of three new M3 statements: ParallelTask, ParallelBlock, and SerialBlock. ParallelTask and

ParallelBlock are used to implement parallelism, whilst SerialBlock is used in the implementation of optimisations which require the fusion of statements. This section discusses each in more depth, and presents OpenMP examples of code generated for them.

4.3.1 ParallelTask

A ParallelTask is a direct translation of tasks. ParallelTasks enclose other statements, indicating that they may be executed as a task in parallel with other ParallelTasks so long as their dependencies are satisfied.

The ParallelTask statement is implemented using OpenMP's `task` construct with `depend` clauses for the implementation of in-dependencies and out-dependencies. Because it is possible to add OpenMP constructs without changing the contained code, a statement contained within a ParallelTask does not need to be emitted differently than a normal statement. This provides a direct translation of ParallelTasks to OpenMP tasks. For a ParallelTask enclosing a statement `A`, the following code is generated:

```
#pragma omp task default(shared) depend(in: ...) depend(out: ...)
{
  // Statement A
}
```

Where `...` is representative of statement `A`'s in and out dependencies.

4.3.2 ParallelBlock

A ParallelBlock statement encloses an ordered list of ParallelTasks, indicating that the enclosed tasks' dependencies must be handled in order presented. ParallelBlock is mainly an aid to code generation which gives an order for dependency handling, and makes code generation easier for the implementation.

ParallelBlock is implemented using a combination of OpenMP's `parallel` and `single` construct with a `nowait` clause. A combination of the two allows a single thread to execute the region, collect all the tasks inside, and allocate them to the other threads spawned by the `parallel` construct.

When a ParallelBlock is encountered during code generation, references to relations and aggregate used by the encapsulated tasks are emitted. This allows these relations and aggregates to be used within OpenMP `depend` clauses, as they are generated as class member variables which cannot be used within `depend` clauses normally. The references use the same identifiers as the relations and aggregates, which shadow the original identifiers, and allow for code generation to remain unchanged.

For a ParallelBlock enclosing tasks `A`, `B`, and `C` which enclose statements of the same name; the following code is generated:

```
// References to relations and aggregates used within the parallel
    block
// auto& ... = ...;
#pragma omp parallel default(shared)
{
#pragma omp single nowait
{

// Task A
#pragma omp task default(shared) depend(in: ...) depend(out: ...)
{
  // Statement A
}

// Task B
#pragma omp task default(shared) depend(in: ...) depend(out: ...)
{
  // Statement B
}

// Task C
#pragma omp task default(shared) depend(in: ...) depend(out: ...)
{
  // Statement C
}

}
}
```

4.3.3 SerialBlock

The SerialBlock enables the fusion of several statements into a single statement, as utilised for several of the presented optimisations. The SerialBlock encloses an ordered list of statements, indicating that the enclosed statements must be executed in the given order.

The SerialBlock statement does not translate to any OpenMP code. Instead, the statement is implemented by emitting each statement in the order they are listed. This utilises existing DBToaster code generation and requires little added functionality to code generation.

4.4 Command-Line Interface

To allow users of DBToaster to parallelize their own code, DBToaster's command-line interface was extended to provide comprehensive control over the parallelization strategies and optimisations presented in chapter 3.

Parallelization is enabled by the `--parallel <strategy>` option, which is also used to specify a parallelization strategy. `--parallel coarse` is used to select coarse-grain parallelization, and `--parallel fine` is used to select fine-grain parallelization.

Optimisations are enabled by the `-P<number>` and `-P <optimisation>` options. When `-P0` is specified, no optimisation are applied. When `-P1` is specified, only structural optimisations are applied. When `-P2` is specified, all optimisations are applied. Individual optimisations may also be set manually using their designated `-P` parameter. The options for enabling each optimisation are shown below:

- `-P fusesequential`: Enables the fuse-sequential optimisation.
- `-P fusetrivials`: Enables the fuse-trivials optimisation.
- `-P trimtrivials`: Enables the trim-trivials optimisation.
- `-P trimsequential`: Enables the trim-sequential optimisation.

When no optimisation options are specified, the `-P1` option is implicitly set as default, as it avoids complexity optimisations which can sometimes have unintended effects on performance for coarse-parallelization. This is discussed at length in chapter 5.

4.5 Order of Optimisations

Optimisations are applied in the following order:

1. Fuse-sequential
2. Fuse-trivials
3. Trim-trivials
4. Trim-sequential

Then, optimisations are applied until no difference in observed between dependency graphs. This ordering prioritises optimisations which fuse statements together, resulting in a larger workload for tasks which can help to mitigate overheads. Optimisations which remove statements are prioritised last to avoid removing statements which could be fused as much as possible.

4.6 Additional Features

During the implementation of parallelization strategies and optimisations, some additional features were implemented to aid parallelization and inspection of M3 programs.

4.6.1 Automatic Thread Allocation

Task-parallelization results in a finite number of tasks which can be executed in parallel at any one time. However, as observed in chapter 2, the overhead of individual tasks increases with the number of threads allocated to the execution of a parallel region. For this reason, it is important to limit the number of threads which a parallel region is allocated. By default, the `OMP_NUM_THREADS` environment variable may be used to set

the number of threads allocated, but this may require some optimisation by the user. For this reason, an additional optimisation was added which strictly limits OpenMP `parallel` regions to using only as many threads as there are tasks which may execute in parallel at any one time. For each parallelization strategy, a different calculation is made.

For the coarse-grained parallelization strategy, the number of threads is limited to the number of tasks, as all tasks are necessarily independent and therefore can be executed at the same time.

For the fine-grained parallelization method, each statement in a dependency graph is iterated over, and the maximum set of statements which are independent of each other is found. The size of the largest set over all statements defines the number of threads allocated to a `parallel` region.

Automatic thread allocation is enabled by using the `-P automaticthreadcount` option.

4.6.2 M3 Program Visualisation

As a debugging and visualisation tool, an additional language option was added for DBToaster code emission – DOT. DOT is a language which is used for plain text descriptions of graphs [2]. DOT is supported by a variety of tools such as the command-line *dot* tool, available for UNIX systems.

Visualisation can be enabled by specifying `-l dot` which results in a visualisation of an M3 dependency graph in DOT format with colouring for trivial and nontrivial statements. This representation can than be visualised using external tools.

Chapter 5

Evaluation

In this chapter, we discuss the evaluation of the implementation of the automatic parallelization strategies and optimisations presented in chapter 3. The evaluation compiles and runs programs for a query set containing a variety of SQL aggregates, designed to show the effects of automatic parallelization of M3 on program runtime. Using our findings, we discuss the merits of task-parallelization of M3, and its shortcomings.

Programs in both single-tuple and batch update modes will be tested, along with programs with each individual optimisation, all optimisations, and no optimisations. A batch size of 1000 was used for batch mode programs. Full tabulations of results are available under appendices C through L

5.1 Query Set

The query set chosen for evaluation was constructed to include several queries which utilise distinct SQL aggregates, and workarounds for the MIN and MAX aggregates which DBToaster does not currently support [7]. The query set produces a mixture of trivial and nontrivial M3 statements, which presents a challenge for optimisations.

The queries are tested against chronological stock market index data sets which associated prices of the stock market index with dates and minutes within that date. Each query in the query set has three versions; one which is limited to evaluation of data which falls within the first week, another one which is limited to evaluation of data which falls within the first month, and finally one which is limited to the evaluation of data which falls within the first 6 months. This is to simulate the different time categories which a dashboard user may be interested in plotting.

The full query set can be found under appendix A.

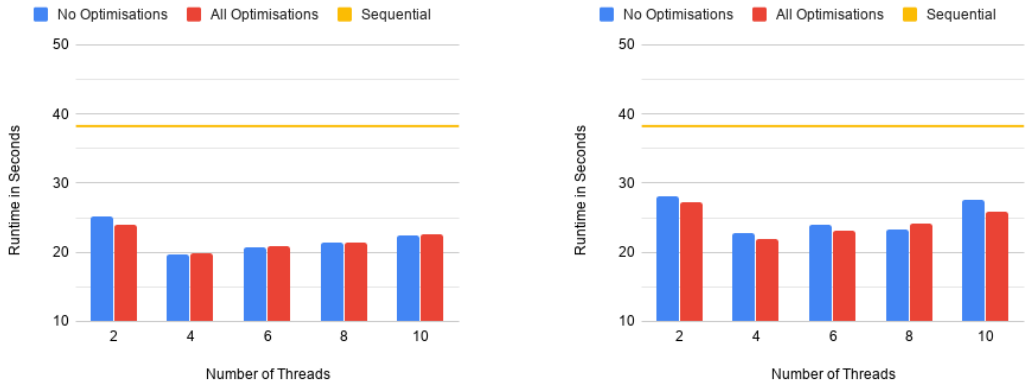
5.2 Data Sets

All programs are evaluated for two separate datasets: a dataset of 65160 entries covering 6 months of stock market index data, and a dataset of 262800 entries covering 24 months of stock market index data.

The 6 month data set aims to evaluate programs for data which falls within the conditions of the queries within the query set, whilst the 24 month data set aims to evaluate programs for a mixture of data which may fall within and outwith the conditions of query sets, which is a common occurrence for real-world systems. Both datasets are stochastically generated using the python script under appendix B, which uses additive increases an multiplicative decreases in prices to generate somewhat volatile stock data.

5.3 Fine-Grain Parallelization

This sections covers the evaluation of programs generated using the fine-grain parallelization strategy presented in chapter 3. Figure 5.1 shows runtimes of programs parallelized using the fine-grain parallelized programs for the 6 month dataset.



(a) Runtimes for single-tuple updates.

(b) Runtimes for batch updates.

Figure 5.1: Comparison of runtimes for fine-grain parallelization strategy for single-tuple and batch updates. Sequential baseline is shown in yellow. Lower is better.

Fine-grain parallelization performs well for the 6 month dataset with and without optimisations. A significant improvement is attained by fine-grain parallelization of both single-tuple and batch update programs, however, the difference between unoptimised and optimised code is insignificant. The optimal number of threads for the parallelized programs is 4 threads, though there is little variance in runtime for between 4 and 10 threads, and all show significant improvement over sequential execution.

Tables 5.1 and 5.2 show that most optimisations are been effective at reducing the number of statements involved in both single-tuple and batch update programs' depen-

Optimisations	Tasks Generated
No optimisations	43
Fuse-trivials	25
Trim-sequential	43
Fuse-sequential	37
Trim-trivials	18
All optimisations	18

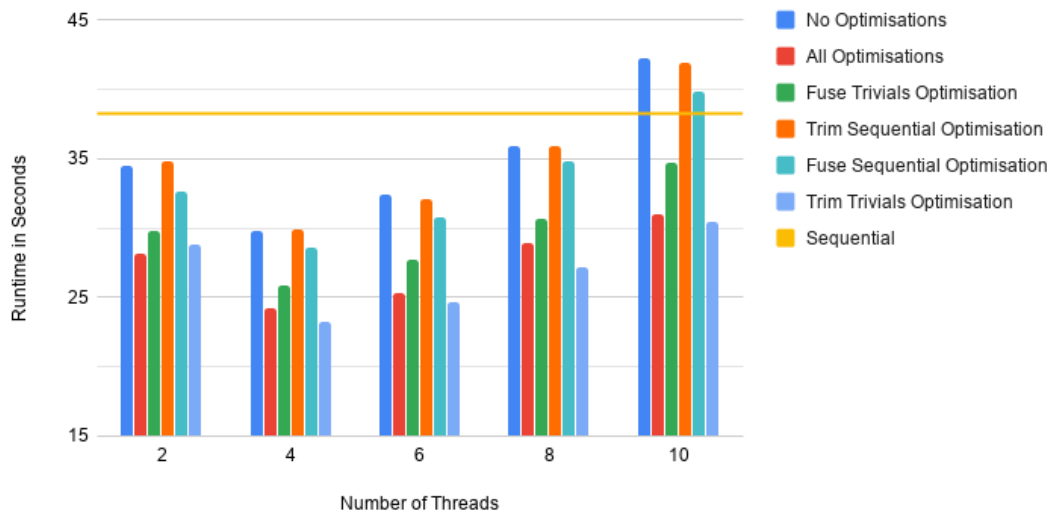
Table 5.1: Tasks generated with optimisations applied for single-tuple update programs.

Optimisations	Tasks Generated
No optimisations	71
Fuse-trivials	49
Trim-sequential	71
Fuse-sequential	61
Trim-trivials	46
All optimisations	46

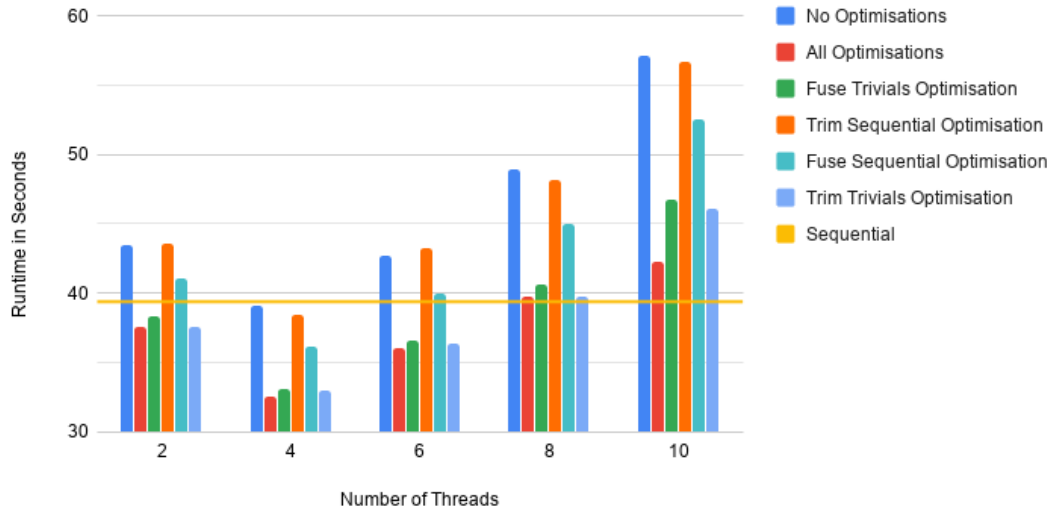
Table 5.2: Tasks generated with optimisations applied for batch update programs.

dency graphs, but for the 6 month data set, no performance improvements have come as a result of this.

Figure 5.2 shows runtimes of programs parallelized using the fine-grain strategy for the 24 month dataset.



(a) Runtimes for single-tuple updates.



(b) Runtimes for batch updates.

Figure 5.2: Comparison of runtimes for programs parallelized using the fine-grain parallelization strategy for single-tuple and batch updates, for the 24 month dataset. Sequential baseline is shown in yellow. Lower is better.

For the larger dataset, only optimised programs show a significant improvement in runtime, and unoptimised – or partially optimised – programs don't perform as well. This is due to the code which is emitted due to the queries' `WHERE` clauses.

SQL's `WHERE` clauses are often translated by DBToaster into conditions in native code which filter input tuples. This allows statements to short-circuit their execution on input tuples to skip unnecessary work for query updates. This is an important feature for efficiency and correctness, but does mean that the runtime of individual M3 statements can depend on input data.

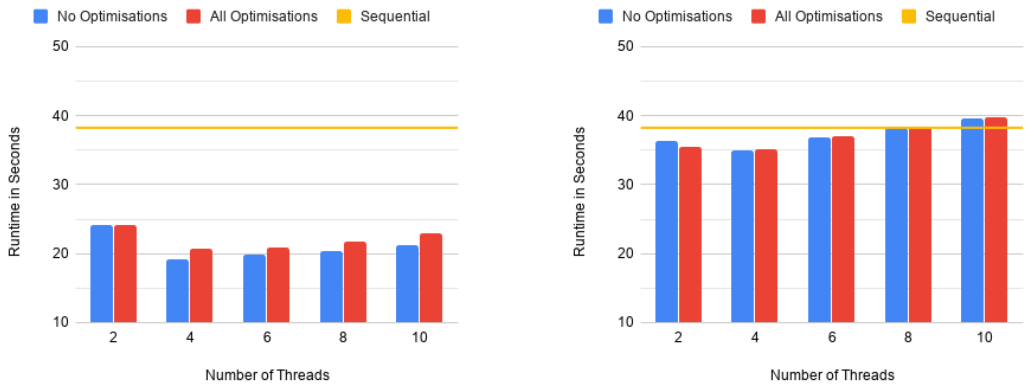
Because the query set only considers the first 6 months of data, data for the following 18 months results in a lot of short-circuiting and therefore reduced work for individual statements. The number of threads allocated is also an import factor in runtime for the 24 month dataset. As discussed earlier, task overhead increases with threads allocated; since individual statements do not have as much work to do for most data, the increase in task overhead is much more prominent, and – for the all but the 4 thread case – results in a worse runtime than sequential execution.

Because task overhead is more prominent here, optimisations have a much more significant effect on runtime, and the effect on runtime by each optimisation is correlated with the number of tasks generated after their application. Fewer tasks result in a shorter runtime.

With optimisations and careful thread allocation, fine-grained parallelization is still capable of significant improvements in program runtime.

5.4 Coarse-Grained Parallelization

This sections covers the evaluation of programs generated using the coarse-grain parallelization strategy presented in chapter 3. Figure 5.3 shows runtimes of programs parallelized using the coarse-grain parallelized programs for the 6 month dataset.



(a) Runtimes for single-tuple updates.

(b) Runtimes for batch updates.

Figure 5.3: Comparison of runtimes for coarse-grain parallelization strategy for single-tuple and batch updates. Sequential baseline is shown in yellow. Lower is better.

Like fine-grain parallelization, coarse-grain parallelization results in good performance using the 6 month dataset with and without optimisations for single-tuple updates. However, it does not perform well for batch updates. Like fine-grain parallelization, optimisation makes little difference to runtime, and 4 threads remains to be the optimal number of threads allocated to all programs, though there is little variance in performance between 4 and 10 threads.

One possible reason that batch-update programs are less performant than their single-tuple counterparts, is that they lack parallelization of some significant statements which contribute largely to runtime. These statements may belong to the same disconnected sub-graph, resulting in them being fused into a single task.

Optimisations	Tasks Generated
No optimisations	13
Fuse-trivials	13
Trim-sequential	13
Fuse-sequential	13
Trim-trivials	18
All optimisations	13

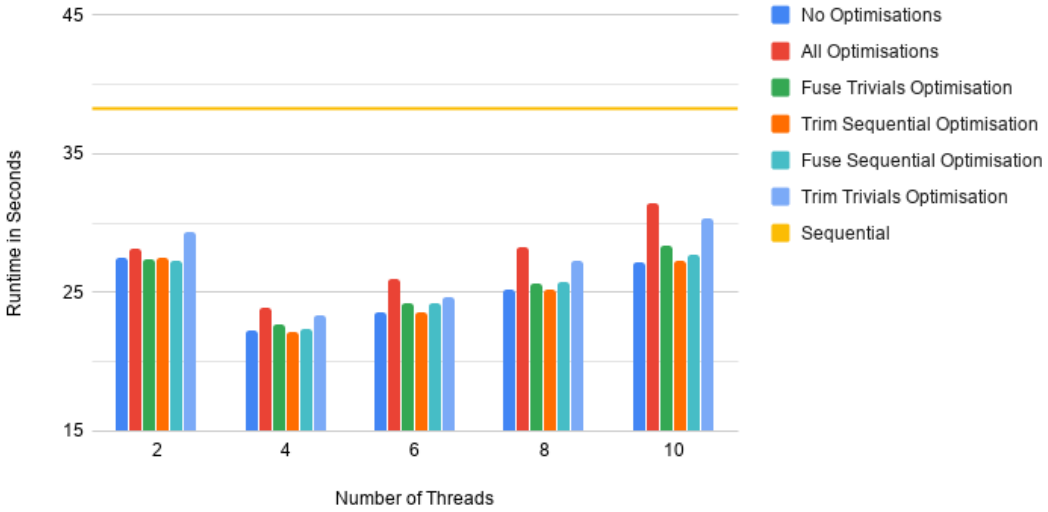
Table 5.3: Tasks generated with optimisations applied for single-tuple updates.

Optimisations	Tasks Generated
No optimisations	7
Fuse-trivials	7
Trim-sequential	7
Fuse-sequential	7
Trim-trivials	7
All optimisations	7

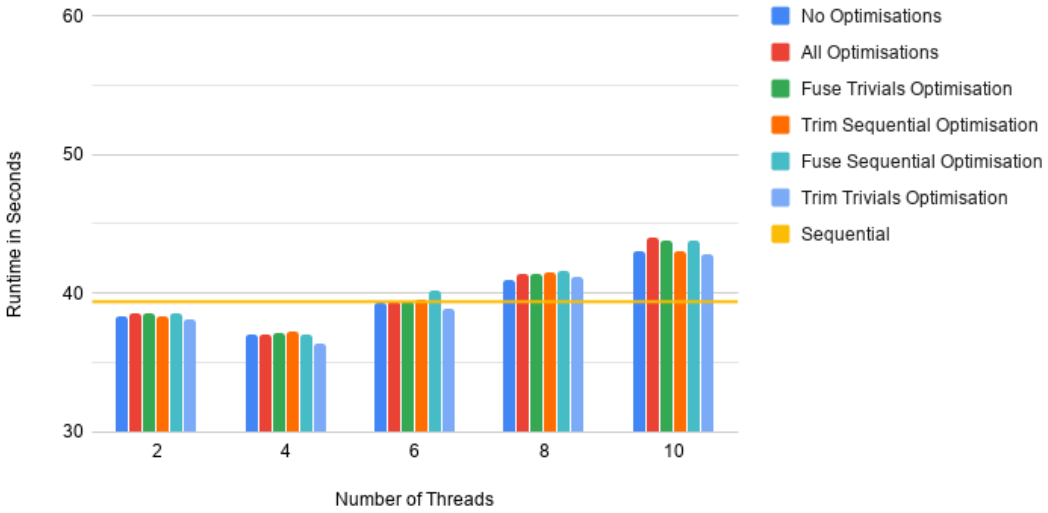
Table 5.4: Tasks generated with optimisations applied for batch updates.

Tables 5.3 and 5.4 show that optimizations are not particularly effective for either single-tuple or batch update programs. All optimizations except trim-trivials alone have no effect on the number of tasks generated by coarse-grain parallelization.

Figure 5.4 shows runtimes of programs parallelized using the coarse-grain strategy for the 24 month dataset.



(a) Runtimes for single-tuple updates.



(b) Runtimes for batch updates.

Figure 5.4: Comparison of runtimes for programs parallelized using the coarse-grain parallelization strategy for single-tuple and batch updates, for the 24 month dataset. Sequential baseline is shown in yellow. Lower is better.

For the larger dataset, optimisations still do not have a measurable effect which is expected as optimisations did not reduce the number to tasks, and therefore did not reduce task-associated overhead. However, unlike the fine-grain parallelization strategy, coarse-grain parallelization for single-tuple updates does not rely on updates to achieve a significant performance improvement over sequential execution. This is likely because tasks are formed of several statements which increases the work done by each task, thus helping to mitigate overheads.

5.5 Summary

Evaluation exposed behaviours of both parallelization strategies which work to their advantage, or are a shortcomings which stop a strategy from being as effective as it could be. This section aims to discuss these merits and shortcomings.

5.5.1 Merits

It has been shown that both parallelization strategies are effective in certain circumstances, and that optimisations can be of great benefit, particularly to the fine-grain parallelization strategy. Both parallelization strategies, with the aid of optimizations are effective for parallelization of single-tuple update programs, and the fine-parallelization strategy is particularly effective for parallelization of batch update programs. Also, optimisations were shown to be very effective at reducing the number of tasks generated by fine-grain parallelization, at times reducing them by over 50%.

5.5.2 Shortcomings

Despite the success of the parallelization strategies for most of the test cases, there are still shortcomings to be solved in future.

Firstly, optimisations did not have a measurable effect on the number of task generated by the coarse-grain parallelization strategy. This can be blamed on coarse-parallelization's method of task allocation, which allocates groups of statements to single tasks. This means that the work done for each task can be imbalanced, and parallelization of non-trivial statements is combined into a single task. The present optimisations don't account for this, and aren't always effective in splitting up the independent statement groups. New optimisations – with more aggressive strategies for splitting these statement groups, or strategies for combining smaller statement groups – could be a solution to this problem.

Secondly, task overhead can be amplified depending on input data, because of conditions present in input queries which cause short-circuiting behaviour. To solve this problem, conditional parallelization is required, which can depend on this short-circuiting behaviour to decide which statements are executed in parallel.

Chapter 6

Conclusion

This chapter concludes the report, and discusses the merits of the presented methods, and system implementation. It discusses some problems which constrained the project and potential directions for project extensions.

6.1 Achievements

Despite the shortcomings discussed during evaluation, the project achieves the its stated aims. The project presented methods for parallelization, optimisations for parallelization, an implementation of the presented methods in DBToaster, and an evaluation of the implementation against a query set which exhibits its advantages and disadvantages.

The project has successfully explored new methods of parallelization, and identified new problems to tackle for improvements and extensions of these methods in the future.

6.2 Future Work

Despite achieving the stated aims, it is clear that there is still work to do to make automatic parallelization of M3 practical. During evaluation it was discovered that many factors can affect the effectiveness of parallelized M3, some of which are outside the control of compile-time parallelization. This section discusses extensions to this project to tackle these issues, or expand on ideas and concepts which were presented earlier. Each extension discussed here could be a project of its own, and are intended to provide possible directions for the next year of the project.

6.2.1 Runtime Task-Parallelization

As discussed during evaluation, compile-time parallelization can be ineffective due to properties of input data which cannot be known until runtime. Whilst optimisations were shown to be effective at mitigating this effect, the difference between parallel execution runtime and sequential execution runtime still decreased, making parallelization less effective overall.

To mitigate this effect, we propose the implementation of runtime task scheduler capable of making choices of parallelization at runtime. Such a system would require a runtime equivalent of the complexity heuristic presented earlier which is able to reason about the complexity of nontrivial code at runtime based upon the size of relations which they iterate over.

A naïve approach to this problem is thresholding, which was implemented but not evaluated during the course of this project. This approach compares a user defined threshold to the size of relations which a statement iterates over, and will only execute the statement as a task if the relation size is larger than the threshold. The implementation of thresholding can be enabled using the `--pthreshold <threshold>` option with the specified threshold.

It is also possible that a runtime parallelization system could allow for applications of the optimisations presented earlier in real-time based on an extended complexity heuristic. This could be particularly effective at removing runtime overheads, and reordering statements to exploit the complexity of statements on per-update basis.

6.2.2 Thread-Safe Hashmap Implementation

The core constraining elements of parallelization are data dependencies between statements. The current hashmap implementations used to implement relations in DBToaster generated programs are not thread-safe and thus can result in data races if multiple reads and writes, or just multiple writes, on a relation are performed at the same time. This results in dependencies between statements which could be eliminated by the introduction of a thread-safe hashmap implementation with support for the special operations required by DBToaster such as slicing.

A new hashmap implementation is also an opportunity to implement OpenMP friendly iteration methods which would mean that automatic data-parallelization could be realistically implemented. This would increase the number of threads a parallelized DBToaster program can utilise, and make the parallel workload more granular overall, resulting in better load balancing.

This extension may be key to improving the coarse-grain parallelization method, which, as we discovered, sometimes has difficulty parallelizing DBToaster programs.

6.2.3 Further Exploration and Evaluation of Optimisations

This project only covered a small number of optimisation methods, and evaluated them against a single varied query set. There is clearly much more space for optimisations and evaluation thereof, which could lead to even better performing parallel programs.

This project did not explore optimisations which are more aggressive in their approach to removing statements from a parallel workload, or fusing statements. These kinds of optimisations are already common place in compilers for sequential code and can show significant improvements in code generation [23]. In the future these could be added to the roster of parallelization optimisations available to DBToaster and evaluated for a variety of queries.

Furthermore, evaluation was limited in this project as exploration of parallelization methods was the main focus, however future extensions could explore more varied input queries and focus on isolating the effects of optimisations more so than was done during this project. No matter the direction of the project in future, it is key for extended methods of parallelization and optimisation to better understand their effects on more specific inputs.

Bibliography

[1] Alexander Aiken and Alexandru Nicolau. "Optimal loop parallelization". In: *ACM SIGPLAN Notices* 23.7 (1988), pp. 308–317.

[2] Tony Ballantyne. *Drawing Graphs using DOT and Graphviz*. 2018. URL: https://www.tonyballantyne.com/graphs.html (visited on 04/16/2020).

[3] Mohamed-Walid Benabderrahmane et al. "The polyhedral model is more widely applicable than you think". In: *International Conference on Compiler Construction*. Springer. 2010, pp. 283–303.

[4] OpenMP Architecture Review Board. *OpenMP Application Program Interface Version 4*. 2013. URL: https://www.openmp.org/specifications/ (visited on 04/16/2020).

[5] Juan Manuel Martinez Caamaño et al. "APOLLO: Automatic speculative polyhedral loop optimizer". In: 2017.

[6] The DBToaster Consortium. *DBToaster - Architecture*. 2014. URL: https://dbtoaster.github.io/docs_architecture.html (visited on 04/13/2020).

[7] The DBToaster Consortium. *DBToaster - DBToaster SQL Reference*. 2014. URL: https://dbtoaster.github.io/docs_sql.html (visited on 04/14/2020).

[8] The DBToaster Consortium. *DBToaster - Welcome to DBToaster.org*. 2017. URL: https://dbtoaster.github.io/ (visited on 04/13/2020).

[9] Intel Corporation. *Automatic Parallelization with Intel® Compilers — Intel® Software*. 2018. URL: https://software.intel.com/en-us/articles/automatic-parallelization-with-intel-compilers/ (visited on 04/14/2020).

[10] Intel Corporation. *Intel® Threading Building Blocks Documentation*. 2020. URL: https://software.intel.com/en-us/tbb-documentation (visited on 04/14/2020).

[11] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *OSDI'04: Sixth Symposium on Operating System Design and Implementation*. San Francisco, CA, 2004, pp. 137–150.

[12] EPCC. *EPCC OpenMP micro-benchmark suite — EPCC at The University of Edinburgh*. 2015. URL: https://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/epcc-openmp-micro-benchmark-suite (visited on 04/16/2020).

[13] Michael J. Flynn. "Some Computer Organizations and Their Effectiveness". In: *IEEE Transactions on Computers* (1972), pp. 948–960. DOI: 10.1145/1671970.1921702.

[14] Apache Software Foundation. *Apache Storm*. 2019. URL: https://storm.apache.org/ (visited on 04/16/2020).

[15] Makoto Ishihara, Hiroki Honda, and Mitsuhisa Sato. "Development and implementation of an interactive parallelization assistance tool for OpenMP: iPat/OMP". In: *IEICE transactions on information and systems* 89.2 (2006), pp. 399–407.

[16] Ling Liu. "Computing infrastructure for big data processing". In: *Frontiers of Computer Science* 7.2 (2013), pp. 165–170.

[17] Tim Mattson and Larry Meadows. "A "Hands-on" Introduction to OpenMP". In: *Intel Corporation* (2014).

[18] OpenMP. *Home - OpenMP*. 2019. URL: https://www.openmp.org/ (visited on 04/14/2020).

[19] OpenMP. *OpenMP Compilers & Tools - OpenMP*. 2019. URL: https://www.openmp.org/resources/openmp-compilers-tools/ (visited on 04/14/2020).

[20] K. Rupp et al. *42 Years of Microprocessor Trend Data*. 2018. URL: https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/ (visited on 04/13/2020).

[21] Christopher J Shallue et al. "Measuring the effects of data parallelism on neural network training". In: *arXiv preprint arXiv:1811.03600* (2018).

[22] University of Strasbourg. *APOLLO Automatic speculative POLyhedral Loop Optimizer*. 2016. URL: http://apollo.gforge.inria.fr/about (visited on 04/16/2020).

[23] Jingling Xue. "Aggressive loop fusion for improving locality and parallelism". In: *International Symposium on Parallel and Distributed Processing and Applications*. Springer. 2005, pp. 224–238.

Appendices

Appendix A

Stocks Query Set and Schema

```sql
-- Written by Murray Steele -- S1655225
-- Assumptions:
--   The tick field starts at 1
--   There are 360 distinct ticks per date, one for each minute of a 6-hour day

-- Define input stream schema
CREATE STREAM S(tick INT, timestamp DATE, open DOUBLE, close DOUBLE, high DOUBLE, low DOUBLE)
  FROM FILE 'data/stocks/stock.csv' LINE DELIMITED CSV (fields := ',');



-- Count number of ticks
SELECT COUNT(*) AS n
FROM S;



-- Count number ticks where the close is higher than the open
-- For the first week
SELECT COUNT(*) AS n
FROM S AS S1
WHERE S1.close > S1.open
  AND S1.tick <= 2520;

-- For the first month
SELECT COUNT(*) AS n
FROM S AS S1
WHERE S1.close > S1.open
  AND S1.tick <= 11160;

-- For the first six months
SELECT COUNT(*) AS n
FROM S AS S1
WHERE S1.close > S1.open
  AND S1.tick <= 65160;
```

```sql
-- Daily average prices
-- For the first week
SELECT S1.timestamp AS timestamp,
       AVG(S1.open) AS open,
       AVG(S1.close) AS close,
       AVG(S1.high) AS high,
       AVG(S1.low) AS low
FROM S AS S1
WHERE S1.tick <= 2520
GROUP BY S1.timestamp;

-- For the first month
SELECT S1.timestamp AS timestamp,
       AVG(S1.open) AS open,
       AVG(S1.close) AS close,
       AVG(S1.high) AS high,
       AVG(S1.low) AS low
FROM S AS S1
WHERE S1.tick <= 11160
GROUP BY S1.timestamp;

-- For the first 6 months
SELECT S1.timestamp as timestamp,
       AVG(S1.open) AS open,
       AVG(S1.close) AS close,
       AVG(S1.high) AS high,
       AVG(S1.low) AS low
FROM S AS S1
WHERE S1.tick <= 65160
GROUP BY timestamp;



-- Max prices per day
-- for the first week
SELECT S1.timestamp AS timestamp,
       S1.high AS high
FROM S AS S1
WHERE S1.tick <= 2520
  AND S1.high >= ALL (SELECT S2.high
                      FROM S AS S2
                      WHERE S2.tick <= 2520
                        AND S1.timestamp = S2.timestamp);

-- For the first month
SELECT S1.timestamp AS timestamp,
       S1.high AS high
FROM S AS S1
WHERE S1.tick <= 11160
  AND S1.high >= ALL (SELECT S2.high
                      FROM S AS S2
                      WHERE S2.tick <= 11160
                        AND S1.timestamp = S2.timestamp);
```

```sql

-- For the first six months
SELECT S1.timestamp AS timestamp,
       S1.high AS high
FROM S AS S1
WHERE S1.tick <= 65160
  AND S1.high >= ALL (SELECT S2.high
                      FROM S AS S2
                      WHERE S2.tick <= 65160
                        AND S1.timestamp = S2.timestamp);



-- Min prices per day
-- For the first week
SELECT S1.timestamp AS timestamp,
       S1.low AS low
FROM S AS S1
WHERE S1.tick <= 2520
  AND S1.low <= ALL (SELECT S2.low
                     FROM S AS S2
                     WHERE S2.tick <= 2520
                       AND S1.timestamp = S2.timestamp);


-- For the first month
SELECT S1.timestamp AS timestamp,
       S1.low AS low
FROM S AS S1
WHERE S1.tick <= 11160
  AND S1.low <= ALL (SELECT S2.low
                     FROM S AS S2
                     WHERE S2.tick <= 11160
                       AND S1.timestamp = S2.timestamp);


-- For the first six months
SELECT S1.timestamp AS timestamp,
       S1.low AS low
FROM S AS S1
WHERE S1.tick <= 65160
  AND S1.low <= ALL (SELECT S2.low
                     FROM S AS S2
                     WHERE S2.tick <= 65160
                       AND S1.timestamp = S2.timestamp);
```

Appendix B

Stocks Data Generation Script

```python
#!/usr/bin/env python3

# Written by Murray Steele -- S1655225

from datetime import datetime
from datetime import timedelta
from dateutil.relativedelta import relativedelta
import random
from statistics import mean, stdev
import sys

# Get arguments
MONTHS = int(sys.argv[1])
FILE_PATH = sys.argv[2]

# Initial tick
START_TICK = 1

# Start and end date periods
START_DATE = datetime(3000, 1, 1)
END_DATE = START_DATE + relativedelta(months=MONTHS)

# Do one tick per minute, representing 6 hours of data
DAY_TICKS = 360
INIT_PRICE = 1000

# Set max increment of price on any given day
MAX_INC = 10

# Randomise seed
random.seed()

# Set state variables
tick = START_TICK
date = START_DATE
open_price = INIT_PRICE
close_price = INIT_PRICE
high_price = INIT_PRICE
```

```
low_price = INIT_PRICE

# Write randomised stock data to file
# Stock data is generated with additive increases, and multiplicative decreases
# This results in spikey data similiar to that of a real stock
with open(FILE_PATH, 'w+') as f:
  while date < END_DATE:
    for _ in range(DAY_TICKS):
      date_str = date.strftime('%Y-%m-%d')
      open_price = close_price
      delta = random.uniform(0.0, 1.1)
      if delta <= 1.0:
        close_price = delta * MAX_INC + open_price
      else:
        close_price = open_price * random.uniform(0.9, 1.0)
      high_delta = random.uniform(1.0, 1.01)
      high_price = max(open_price, close_price) * high_delta
      low_delta = random.uniform(0.99, 1.0)
      low_price = min(open_price, close_price) * low_delta
      f.write(f'{tick},{date_str},{open_price},{close_price},{high_price},{low_price}\n')
      tick += 1
    date += timedelta(days=1)
print(f'TOTAL TICKS: {tick - START_TICK}')
```

Appendix C

Runtimes for Sequentially Executed DBToaster Programs for the 6 Month Dataset

Runtimes for sequential execution

Update Mode	Runtime	
Single-tuple	38.254	
Batch	39.254	

Appendix D

Runtimes for Sequentially Executed DBToaster Programs for the 24 Month Dataset

Runtimes for sequential execution

Update Mode	Runtime	
Single-tuple	38.244	
Batch	39.404	

Appendix E

Runtimes for Fine-Grain Parallelized Programs with Single-Tuple Updates for the 6 Month Dataset

Runtimes for N threads

Optimisations	**N=2**	**N=4**	**N=6**	**N=8**	**N=10**
No optimisations	25.149	19.713	20.651	21.316	22.435
Fuse-trivials	24.224	21.103	21.257	22.275	23.486
Trim-sequential	25.159	20.325	20.461	21.306	22.433
Fuse-sequential	25.098	20.303	20.982	21.979	23.309
Trim-trivials	24.684	19.225	19.874	21.057	21.662
All optimisations	23.985	19.818	20.84	21.368	22.546

Appendix F

Runtimes for Fine-Grain Parallelized Programs with Batch Updates for the 6 Month Dataset

Runtimes for N threads

Optimisations	**N=2**	**N=4**	**N=6**	**N=8**	**N=10**
No optimisations	28.104	22.683	23.951	23.208	27.501
Fuse-trivials	26.976	21.973	22.967	24.573	26.027
Trim-sequential	27.582	22.615	23.772	25.175	27.384
Fuse-sequential	27.415	22.435	23.662	24.642	26.495
Trim-trivials	26.941	22.508	23.345	24.373	26.417
All optimisations	27.314	21.93	23.086	24.205	25.936

Appendix G

Runtimes for Coarse-Grain Parallelized Programs with Single-Tuple Updates for the 6 Month Dataset

Runtimes for N threads

Optimisations	**N=2**	**N=4**	**N=6**	**N=8**	**N=10**
No optimisations	24.16	19.08	19.865	20.42	21.21
Fuse-trivials	24.11	19.381	20.266	20.811	21.724
Trim-sequential	24.303	19.886	20.041	20.833	21.604
Fuse-sequential	24.188	19.362	20.002	20.68	21.633
Trim-trivials	24.704	19.271	19.855	20.627	21.626
All optimisations	24.069	20.695	20.844	21.809	22.942

Appendix H

Runtimes for Coarse-Grain Parallelized Programs with Batch Updates for the 6 Month Dataset

Runtimes for N threads

Optimisations	**N=2**	**N=4**	**N=6**	**N=8**	**N=10**
No optimisations	36.396	35.021	36.933	38.26	39.645
Fuse-trivials	36.711	35.369	36.992	38.391	40.021
Trim-sequential	36.161	36.895	37.528	38.877	40.567
Fuse-sequential	36.441	36.562	37.723	39.062	40.702
Trim-trivials	36.027	37.253	37.898	38.853	39.52
All optimisations	35.416	35.208	37.004	38.402	39.822

Appendix I

Runtimes for Fine-Grain Parallelized Programs with Single-Tuple Updates for the 24 Month Dataset

Runtimes for N threads

Optimisations	**N=2**	**N=4**	**N=6**	**N=8**	**N=10**
No optimisations	34.442	29.769	32.4	35.911	42.268
Fuse-trivials	29.802	25.846	27.687	30.635	34.706
Trim-sequential	34.785	29.845	32.094	35.853	41.926
Fuse-sequential	32.615	28.572	30.728	34.839	39.788
Trim-trivials	28.816	23.222	24.686	27.141	30.418
All optimisations	28.198	24.226	25.306	28.952	30.99

Appendix J

Runtimes for Fine-Grain Parallelized Programs with Batch Updates for the 24 Month Dataset

Runtimes for N threads

Optimisations	N=2	N=4	N=6	N=8	N=10
No optimisations	43.498	39.129	42.744	48.925	57.103
Fuse-trivials	38.367	33.063	36.65	40.656	46.726
Trim-sequential	43.622	38.5	43.207	48.188	56.659
Fuse-sequential	41.021	36.173	39.928	45.012	52.57
Trim-trivials	37.546	33.002	36.334	39.778	46.056
All optimisations	37.55	32.546	36.024	39.815	42.262

Appendix K

Runtimes for Coarse-Grain Parallelized Programs with Single-Tuple Updates for the 24 Month Dataset

Runtimes for N threads

Optimisations	**N=2**	**N=4**	**N=6**	**N=8**	**N=10**
No optimisations	27.522	22.208	23.582	25.157	27.178
Fuse-trivials	27.386	22.644	24.241	25.656	28.331
Trim-sequential	27.471	22.191	23.552	25.194	27.236
Fuse-sequential	27.32	22.408	24.183	25.75	27.754
Trim-trivials	29.386	23.289	24.687	27.255	30.372
All optimisations	28.182	23.937	25.937	28.297	31.376

Appendix L

Runtimes for Coarse-Grain Parallelized Programs with Batch Updates for the 24 Month Dataset

Runtimes for N threads

Optimisations	N=2	N=4	N=6	N=8	N=10
No optimisations	38.344	36.998	39.284	40.963	43.078
Fuse-trivials	38.566	37.171	39.477	41.449	43.849
Trim-sequential	38.371	37.283	39.491	41.501	43.078
Fuse-sequential	38.598	37.004	40.232	41.646	43.765
Trim-trivials	38.071	36.383	38.936	41.208	42.841
All optimisations	38.578	37.035	39.424	41.407	44.01