# Parallel execution of first-order operations

Sina Madani, Dimitrios S. Kolovos, Richard F. Paige

Department of Computer Science, University of York, UK
{sm1748,dimitris.kolovos,richard.paige}@york.ac.uk

**Abstract.** The growing size of software models poses significant scalability challenges. Amongst these scalability issues is the execution time of queries and transformations. Although the processing pipeline for models may involve numerous stages such as validation, transformation and code generation, many of these complex processes are (or can be) expressed by a combination of simpler and more fundamental operations. In many cases, these underlying operations are pure functions, making them amenable to parallelisation. We present parallel execution algorithms for a range of iteration-based operations in the context of the OCL-inspired Epsilon Object Language. Our experiments show a significant improvement in the performance of queries on large models.

## 1 Introduction

Modern software systems are often required to process an ever-increasing volume of complex data with more stringent throughput and latency requirements. Although model-driven engineering helps to curtail the complexity of systems, the performance of many popular modelling tools leaves much to be desired. This is particularly problematic since larger projects are arguably likely to benefit the most from a model-driven approach. Scalability is a notable challenge with model-driven engineering [1], and a multi-faceted one too [2].

Model management workflows often involve a variety of tasks such as validation, comparison, model-to-model and model-to-text transformations. Despite their differences, these tasks typically use a common set of queries and transformations on collections of model elements. With very large models (in the order of millions of elements and gigabytes in size), these operations can incur a significant performance cost, making the process slow and less productive. Furthermore, even the most complex queries on collections of data (model elements) can be expressed using a relatively small set of simpler operations. The Object Constraint Language (OCL) is one of the most well-known and frequently used languages for querying and validating models. As a functional and declarative language, OCL offers a useful set of operations on collections, including operations involving predicate logic. In these first-order operations, a function is applied to some or all of the elements.

In this paper, we demonstrate that nearly all first-order operations can be executed in a data-parallel manner. We illustrate this with a broad set of operations, outlining the challenges with implementing parallel algorithms for these

operations and their respective solutions. Given the diminishing generational improvements in single-thread performance due to physical and technical constraints, combined with the increasing number of cores in virtually all computing devices, we posit that sequential algorithms for inherently parallelisable computations are not optimal when processing large data sets on modern computers.

The remainder of the paper is organised as follows. Section 2 reviews pertinent work on optimising iteration operations. Section 3 introduces Epsilon, which we use to implement and evaluate our solutions, as well as discussing some general pre-requisites for concurrent execution. Section 4 reviews twelve iteration-based operations and describes how they can be re-implemented with a parallel execution algorithm. Section 5 provides a brief overview of our testing methodology and performance metrics. Section 6 concludes the paper and suggests extensions for future developments.

## 2 Background and related work

Most first-order operations on collections have some desirable properties which enable a number of optimisations to be made in their execution algorithms. All first-order operations involve a lambda expression as a parameter, which is typically a predicate. This function takes as input an element from the source collection and evaluates an expression on it (typically to a Boolean value in the case of predicates). Regardless of the transformation type, one of the most crucial properties of first-order operations is that their transformation functions can operate on each element independently. That is, the lambdas are usually pure functions and so they have no effect on the global state of the program in languages like OCL. The motivation for parallel execution thus stems from the observation that in most cases, the same lambda expression can be executed on each element of the source collection in any order, with no dependencies between each invocation and, therefore, no inherent requirement for serial execution.

Parallel execution of the same instructions over multiple data (SIMD) is a form of data-parallelism which is common in traditional "pleasingly parallel" tasks such as video rendering. This class of problems can often map closely to the specialised hardware architectures of graphics processing units.

**Parallel streams** Data parallelism can also be harnessed by general-purpose applications on conventional processors. The Java standard library provides streams [3], which are an abstract processing pipeline over a fixed or infinite data stream. Streams are iterator-based, and so they can be used on collections. The Streams API in Java provides first-order operations such as filtering and transformations in a generic and flexible manner, allowing the programmer to chain operations and perform post-processing or collect the results into any desired data structure. Streams can also execute in parallel, which inevitably requires that the output is unordered.

Parallel streams internally use a divide-and-conquer approach, delegating Java's fork-join processing framework. The key to making this possible is the

ability to split the data source[1], and perhaps more fundamentally, assuming that none of the operations have side-effects or rely on mutable global state.

Furthermore, the iterator-based nature of streams means that the entire operation chain can be evaluated on individual elements, enabling lazy evaluation [4]. That is, instead of requiring the intermediate results of e.g. a filter (*select* in OCL) operation be pooled into a collection only to be filtered again, the operations themselves can be fused to provide short-circuiting behaviour[2].

**Lazy evaluation** On the topic of lazy evaluation of expressions on collections, Tisi et al. (2015) also propose an iterator-based approach in [5]. The basic premise of their work is to treat OCL collections in a similar manner to Java Streams, such that operations on collections are evaluated only when required by a subsequent computation. This is achieved by returning an iterator which evaluates the desired expression on each element when it is iterated over; which may be in a chain of other operations. The authors also apply this lazy approach to the *allInstances()* operation, which retrieves all elements of the target type. Since this is often the source collection on which further operations are invoked, it enables a lazy evaluation strategy to be applied in the entire chain of computation rather than only the intermediate operations.

**Formal parallelism** In [6], Vajk et al. (2011) take a more formal approach to parallel execution of OCL expressions. Using the well-established Communicating Sequential Processes (CSP) model of concurrency, the authors provide a mapping between OCL expressions and CSP processes, focusing on binary expressions for task parallelism and iterators for data parallelism. The CSP is then compiled to C# code. However they do not provide a complete library of parallel operations, instead relying on the most general first-order operation – *iterate* – to implement and evaluate their approach.

Our contribution does not attempt to prove the correctness of a parallel approach, rather it attempts to highlight and demonstrate solutions for the practical technical challenges in hand-coding parallel algorithms for iteration operations on collections in OCL-based languages.

## 3 Epsilon Object Language (EOL)

Epsilon [7] is an open-source family of task-specific model management languages which build upon an imperative, dynamically typed and interpreted "model-oriented" language. This base language – the Epsilon Object Language (EOL) [8] – provides many of the familiar programming constructs and syntax of Java, such as loops, if/else statements, mutable variables and user-defined operations, which may be impure functions. EOL is inspired by OCL in its design, and thus provides almost all of the functionality of OCL with similar syntax and names

[1] see `docs.oracle.com/javase/10/docs/api/java/util/Spliterator.html`
[2] see [4] for an example of this

for types and built-in operations. Invariants can be expressed in the more specific Epsilon Validation Language [9].

Where the Epsilon Object Language differs from OCL is in its more expressive features, allowing for a mix of imperative and declarative style of programming. Since Epsilon is interpreted and written in Java, one of the distinguishing features is that of native types, which enable the creation of native Java objects and invocation of methods on these types. Another notable feature is extended properties, which are additional key-value pairs that can be used to decorate individual variables or model element types.

### 3.1 Epsilon thread safety

Before considering parallelisation of first-order operations, it is important to note and address some general concerns with concurrent execution of an interpreted language like EOL. Although first-order operations are usually free from side-effects, they may depend on global state in the program and on operations; which in EOL could potentially have side-effects. For instance, a first-order operation's predicate may depend on some threshold value which is set globally or as a parameter to the program. The programmer may also find it useful to factor out common functionality of their lambda expressions into an operation to avoid repetition and improve readability. Another common use case in complex queries is the nesting of first-order operations. All of these cases should be supported when executing concurrently in a transparent manner to the user.

Building on our previous work in [10], we identified and solved the main challenges with concurrency in EOL. These can be summarised as follows:

**Variable declarations** Concurrent execution requires a clear policy on variable scoping and visibility between threads. The policy for Epsilon is that any variables declared prior to concurrent execution are accessible for both reads and writes, with writes being strongly discouraged. Variables declared during concurrent execution are localised to the executing job or thread, with no visibility or access once the job has completed. This is achieved by using a thread-local data structure and making the main thread's frame stack visible to all threads.

**Exception handling** The EOL engine keeps track of the execution trace for the program so that in the event of an error, such as navigation of a null property or references to a non-existent variable or operation, the location in the program which caused the error can be reported to the user through a (Java-like) stack trace. With multiple threads of execution, the engine needs to keep track of the trace for each thread, terminating execution when an exception occurs and reporting the cause in the main thread. We solve the former issue by having a stack trace manager for each thread. Terminating execution upon encountering an exception requires us to create an additional thread which waits for concurrent execution to complete, and having the main thread wait on a condition. When concurrent execution has completed or an exception is raised, the main

thread is notified and appropriate action is taken. In the case of exceptions, the Epsilon stack trace is computed and cached, all concurrent jobs are cancelled, the executor service is terminated and finally the exception is reported.

**Caches** Various shared mutable data structures exists in the EOL engine, some of which are lazily initialised to save memory. For example, invocations of *allOfKind* and *allOfType* on model element types populate a cache to avoid querying the model. Features such as cached operations and extended properties also pose concurrency complications. All of these caches are shared and thus thread-safe using a high-performance data structure which only synchronizes for writes (i.e. *ConcurrentHashMap*).

## 4 Parallel implementation of first-order operations

In this section, we explore the declarative operations on Collection types in Epsilon and demonstrate our solutions for their parallel variants. As previously noted, since Epsilon is arguably more feature-rich and expressive than OCL, the solutions outlined in this section should be generalisable to other OCL-like languages. The more general challenges with concurrency discussed in the previous section are effectively encapsulated in the engine's execution context, allowing us to implement the first-order operations in parallel without needing to consider such issues in every case.

Most first-order operations in Epsilon are "select-based", as will become apparent in the subsequent subsections. In general, there are two types of operation executions: those which execute the given lambda expression on all elements, and those which execute it until a certain condition is met. In the latter case, this is applicable to operations where the lambda expression is a predicate. Table 1 shows the first-order operations and their properties. The type $T$ refers to the type of the source collection, whilst other letters refer to derived types.

Although not all of the operations in Table 1 exist in OCL, we chose to implement these in order to deeply explore the challenges with converting from sequential to parallel execution algorithms.

### 4.1 *select* and *reject*

The *select* operation is a filter on the collection, returning a subset for which the given predicate is true. Since this operation's processing logic can be reused, there are two additional internal parameters. The first of these is whether the operation should return when a match is found (i.e. the predicate evaluates to true), and the second is whether the operation is a reject or select. With these parameters, the first four operations in Table 1 can be handled by a single code base in the sequential implementation. *reject* can also be expressed using *select*, since c.reject(i|<*p*>) === c.select(i|not <*p*>) for a predicate *p*. In the sequential implementation, it is also relatively trivial to implement *selectOne* and *rejectOne* in the same code as *select* and *reject*. *selectOne* ("*any*" in OCL) returns

**Table 1.** Iteration-based operation classifications

| Operation | Return On Match | Return Type | Lambda Type |
|---|---|---|---|
| select | No | Collection<T> | Boolean |
| selectOne | Yes | T | Boolean |
| reject | No | Collection<T> | Boolean |
| rejectOne | Yes | Collection<T> | Boolean |
| exists | Yes | Boolean | Boolean |
| forAll | Yes | Boolean | Boolean |
| nMatch | Yes | Boolean | Boolean |
| collect | No (N/A) | Collection<R> | R |
| sortBy | No (N/A) | Collection<T> | Comparable<R> |
| aggregate | No (N/A) | Map<K, V> | K, V |
| mapBy | No (N/A) | Map<K, Collection<T>> | K |
| closure | No (N/A) | Collection<R> | R |

the first value which satisfies the predicate, whilst *rejectOne* adds all of the source collection to the results and removes the first value which does not satisfy the predicate. The implementation becomes more complex when executing in parallel, so the same approach and degree of re-use is not possible.

### 4.2 *parallelSelect* and *parallelSelectOrdered*

As with all other first-order operations, we can execute *select* in parallel by submitting a new job to an executor service such as a thread pool (which takes care of managing thread lifecycles and mapping jobs to threads), for each element in the collection, so that the predicate is then evaluated for each element independently. In the sequential implementation, every time a matching value is found it is added to the results collection. However when executing in parallel, this would require synchronization on the result collection for every write, which would greatly diminish the performance gains from parallelism. There is also the matter of ordering the results in a consistent manner with sequential *select*. For this reason, we offer two implementations: *parallelSelect* when order is immaterial, and *parallelSelectOrdered* when ordering is desired.

If encounter order is to be preserved, one solution is to use Futures [11]. The idea is that when each job is submitted to the executor service, an object which encapsulates the result of the job's computation is returned, and can later be used to retrieve the result. We therefore add all of the futures to a collection and subsequently loop through this collection and get the result for each job in the order that they were submitted[3]. Although getting the result from a Future is blocking, since *select* requires us to operate on every element of the source collection, all results must be obtained before proceeding.

Collecting the results is also not as straightforward as in the sequential case. Recall that a Future returns the result of an asynchronous computation. In the

[3] Ordering is guaranteed since we add the futures and request their results sequentially

case of *parallelSelectOrdered*, this result is not always present. Although the type of the result is determined – the same as that of the source Collection's elements – we also need a mechanism to signal the absence of a result for values which the predicate is false. Note that this cannot simply be *null*, since an item for which the predicate is true may also be null. We therefore wrap the result of the computation into an *Optional*[4] (or any other arbitrary container, such as a singleton collection). This way, if the *Optional* itself is null, we know there was no value present. If the value is present, it can be represented by an Optional (a null value is represented by an empty Optional). Once all jobs have completed and their values obtained, we can add the contents of all non-null Optionals to the results collection sequentially, eliminating the need for a thread-safe collection.

Although the solution with Futures guarantees ordering, it does not offer the best performance due to the overhead of creating additional wrapper objects. For improved throughput, an alternative for gathering results would be to use persistent thread-local collections – essentially, a map of threads to their respective collection of values – and merge results at the end. We use this implementation in *parallelSelect*. Another option would be to annotate objects with their encounter order and restore this order in the returned collection, however this would pose additional sorting and object creation overhead.

### 4.3 *parallelReject* and *parallelRejectOrdered*

Since *reject* is effectively the same operation as *select* with negation of the predicate, we can apply the same approach for *parallelSelect* (and *parallelSelectOrdered*) as we did in the sequential variant. In both cases, the predicate needs to be evaluated on the entire source collection and a collection is returned.

### 4.4 *parallelSelectOne*

The *selectOne* operation differs from *select* in that the return type is a single value, rather than a collection of values. This means that the operation may return without evaluating the predicate on the entire source collection, once a matching value is found. In the case of *parallelSelectOne*, we cannot simply modify the *parallelSelect* logic to achieve this behaviour as with the sequential implementation. Since we only need to find a single value, there is no need to use Futures and wait for all computations to complete. Instead, we need a mechanism which allows us to notify the main thread that a result has been found, pass this result and stop all other jobs, since they are no longer required once a matching value has been found. Furthermore, we also need to ensure that if no matching value can be found, the main thread does not wait indefinitely for a result.

To achieve this, we use an "execution status" object which encapsulates multiple lock conditions, allowing us to wait for either of them. There are two possible ways in which parallel execution of submitted jobs can complete: exceptionally or successfully. So far, we have assumed that successful completion is signalled

[4] `docs.oracle.com/javase/10/docs/api/java/util/Optional.html`

by the completion of all jobs. However for the case of *parallelSelectOne*, we add another condition to our execution status so that when a result is available, the job which found it can invoke a method on the execution status object with the value, which will interrupt the main thread in a similar manner to how an exceptional completion would. Upon interruption, all running jobs are terminated and the value set on the execution status object is returned.

This operation introduces an inconsistency with the sequential *selectOne* in that the returned value may potentially be different for each invocation, whereas the sequential variant will always return the first value which matches the predicate as returned by the source collection's iterator. This is because each computation occurs in parallel and it is non-deterministic which will finish first. However as the name of the OCL operation suggests ("*any*"), the chosen value need not necessarily be the first. The parallel variant is thus not suitable in cases where the user desires a "*selectFirst*" operation.

### 4.5 *parallelRejectOne*

The *rejectOne* operation is a hybrid of *selectOne* and *select* in that it does not need to evaluate the predicate on the entire source collection, but it returns a collection instead of a single value. Consequently, *parallelRejectOne* is not a simple modification of *parallelSelect* or *parallelSelectOne*. However, since we only need to exclude a single value from the source collection which matches the predicate, we can delegate the predicate to a *parallelSelectOne* operation, and remove the returned value from the source collection. However we still have the issue of determining whether a result was found or not, so that we do not inadvertently remove a null value from the source. This can be circumvented by using a simple Boolean flag on the *parallelSelectOne* operation instance, which is set when a result is found. Since this value will only ever be set from *false* to *true* and never queried during parallel execution, there are no concurrency issues to resolve. As with *parallelSelectOne*, this operation is inconsistent with its sequential variant in that the returned collection may exclude an element which is not the first value for which the predicate is not satisfied.

### 4.6 *parallelExists*

The *exists* operation returns *true* if there is at least one element in the source collection which satisfies the predicate (essentially a logical OR of the predicate on each element). This is therefore shorthand for whether a *selectOne* operation with that predicate has a result. Since we added this Boolean flag to *parallelSelectOne*, we can simply delegate this operation and return the value of the flag.

### 4.7 *parallelNMatch*

The *nMatch* operation requires that exactly $n$ elements in the source collection satisfy the predicate. Commonly, this value is either zero (*none* operation) or one (*one* operation). In all cases, the predicate does not need to be evaluated on

all elements if either *n+1* elements are found for which the predicate is satisfied, or if not enough elements have been found depending on the relative size of the source collection and *n*. This requires a loop variable which is incremented every time a match is found, and also the number of elements evaluated (the loop index). As the former is a shared mutable counter, we use an *AtomicInteger*, which enables us to atomically increment and retrieve the value directly from memory using a single CPU instruction without any synchronization. Although the latter counter is incremented in each iteration, it is effectively an immutable input parameter to each parallel job (much like the current element) because we loop through the source collection and submit jobs sequentially. The coordination mechanism for short-circuiting is similar to that of *parallelSelectOne*, except that we do not need to set a value as the result in the execution status.

### 4.8 *parallelForAll*

The *forAll* operation is similar to *exists* except that it requires the predicate to hold for all elements in the collection (i.e. logical AND instead of OR). One possible implementation would be to delegate to *nMatch*, with *n* being the size of the source collection. Alternatively, we can re-use *parallelSelectOne* by adding a new flag which inverts the predicate similar to *rejectOne*. If no element satisfies the negated predicate, we effectively achieve the same semantics as the sequential variant. In other words, we are looking for a counter-example to the predicate and in doing so, we can return without evaluating the predicate for all elements.

### 4.9 *parallelCollect* and *parallelCollectOrdered*

The *collect* operation maps every element of the source collection to another value, potentially of a different type. We can therefore perform this mapping for every element in parallel. As with *parallelSelect*, we offer two modes: ordered and unordered. It is worth noting that in the ordered case, we do not need a wrapper such as Optional since every element will be mapped.

### 4.10 *parallelSortBy* and *parallelSortByOrdered*

The *sortBy* operation orders the elements of the source collection by comparing the result of the expression for each element. This expression can be executed independently for each element in parallel, since this part is a simple extension of the *parallelCollect* operation. Once this collection of comparable properties is obtained through the mapping function, we can sort the collection in parallel; - a task which we leave to the Java standard library to perform using a divide-and-conquer approach[5].

[5] `docs.oracle.com/javase/10/docs/api/java/util/Arrays.html`

### 4.11 *parallelMapBy*

The *mapBy* operation derives a property from each element in the collection and uses it as a key to map the elements. Since this operation returns a multi-map (that is, a key which can have multiple values associated with it), there is an inherent need for synchronization because the key may or may not be present in the results collection at any given time, and the collection of associated values also requires synchronization for writes. To avoid this synchronization requires a fundamentally different execution algorithm. Our solution once again achieves data-parallelism in a similar manner to the *parallelCollect* operation, however instead each job returns a tuple; representing a mapping from the derived result (as obtained by executing the expression for the given element) to the element itself. We can then merge the individual entries (tuples) afterwards, where we resolve duplicate mappings by joining the mapped values into collection for each duplicate entry key, which can be performed in a declarative manner using Java Streams and Collectors API.

### 4.12 *aggregate*

The *aggregate* operation is unique in that it takes up to three lambda expressions as a parameter and returns a key-value map. The first derives a key property, similar to *sortBy*, the second derives a value property, and the third is an optional initialisation parameter. Rather than simply performing a key-value mapping, *aggregate* also has an accumulator variable (the "total"), which is the value mapping associated with the key expression. If no value is set, then the initialiser expression is executed, if supplied to the operation. This operation is effectively then a more complex variant of OCL's *iterate* operation; which behaves like a traditional *for* loop. Due to this accumulator variable, this algorithm is difficult to express in a non-sequential manner. One possibility is to execute each key expression in parallel, since their "totals" (value aggregations) are unrelated. Without some pre-requisite knowledge supplied as additional parameters, annotations or advanced static analysis, a parallel solution may not provide a significant performance benefit.

### 4.13 *closure*

The *closure* operation recursively evaluates the lambda expression on each element and its results and returns them as a flattened collection. This is theoretically parallelisable by using a Fork/Join (divide-and-conquer) approach, and we leave this as future work.

## 5 Evaluation

In this section, we evaluate our parallel solutions for both correctness and performance. Resources for our experiments can be found on GitHub[6].

[6] `github.com/epsilonlabs/parallel-erl`

### 5.1 Correctness

We tested both the sequential and parallel variants of each first-order operation using EUnit [12], which is a framework for testing model management programs using familiar unit testing paradigms such as assertions. EUnit allows us to write test operations in EOL in a similar manner to writing JUnit tests. We create a test for each first-order operation over a simple data set (e.g. a sequence of integers from 0 to 10) and ensure that we exercise all outcomes where possible. For example, when testing the *selectOne* operation, we write two queries: one where there are is at least one possible result and one with no results. Similarly our tests for operations which return a Boolean, such as *forAll* and *exists*, we ensure that both *true* and *false* possibilities are tested. For short-circuiting operations such as *exists* and *nMatch*, we manually ensured that no unnecessary evaluations occur using print statements in the code. We compare the results of the operations to some expected outcome for both the sequential and parallel versions. Of course for the *selectOne* and *rejectOne* operations, we ensure that only one possible solution exists for consistency of the parallel variants.

In addition, we also test for equivalence between the parallel and sequential variants using a different set of tests with different data types (which may also be *null*). We also test that an exception is thrown correctly (with the expected stack trace and detail message in accordance with the sequential implementation) by referencing a non-existent property in the lambda expression for each first-order operation. Finally, for completeness we also test the visibility of variables outside the scope of the lambda expression parameters and also invocation of both user-defined and built-in operations on the iteration variable. This also includes nested first-order operations.

Epsilon also has a large test suite which also makes use of first-order operations. After running these tests many times, our experiments revealed no incorrect results, failed assertions, deadlocks or unexpected exceptions.

### 5.2 Performance

Given the large number of first-order operations and the computational similarities between many of them, our benchmarks exercise the *parallelSelect* and *parallelSelectOne*, since most of the other predicate-based operations can be expressed in terms of these two operations[7]. For our test script and model, we took inspiration from the "Find Couples" transformation in [13] into a query. The script attempts to find all pairs of people who played together in at least three movies, using a simple model of the Internet Movie Database.

The *select* benchmark query consists of two *select* operations, one nested inside another. The outer select's source is all Person instances (i.e. actors and actresses), whilst the inner's source is the set of co-actors for a given person. In our *parallelSelect* benchmark, we only parallelise the outer *select*, relying on the

[7] Evaluation of other operations will be considered in future work

complexity of the inner computation and the number of Person instances to provide sufficient benefit from our data-parallel approach. We ran this experiment using the largest model from [13], which contains over 3.5 million elements.

In our *parallelSelectOne* query, we iterate through all Movie instances, find all of the co-actors for all Persons in the Movie, and assert whether there exists a Person whose name and hash code matches the mid-most Person in the collection of *Person.allInstances()*. We should also note that unlike the *select* benchmark, we used a cached operation for finding the co-actors in this experiment. As this is a much more demanding query, we ran this with a smaller model.

We ran each experiment three times in separate JVM invocations, and took the mean average time[8]. We exclude the time taken for parsing the model, which on average took approximately 40 seconds for the largest model with over 3.53 million elements. For our parallel variants, we used as many threads as physical cores in the system. The test environment for our experiments was as follows:

AMD Ryzen Threadripper 1950X (stock clocks, "Creator Mode") 16-core / 32-thread CPU, 32 (4x8) GB DDR4 RAM @ 3003 MHz, Fedora 28 OS (Linux kernel 4.16), Oracle JDK 10.0.1, JVM options: "*-XX:MaxGCPauseMillis=500 -XX:+AggressiveOpts -XX:+UseNUMA -Xmx32g -Xms4g*"

**Table 2.** Benchmark results for Threadripper 1950X system with 16 threads

| Operation | Model Size | Time (ms) | Speedup | Memory (MB) |
|---|---|---|---|---|
| select | 3 531 618 | 9 086 798 | – | 12 623 |
| parallelSelect | 3 531 618 | 1 578 430 | 5.76 | 17 862 |
| parallelSelectOrdered | 3 531 618 | 1 576 731 | 5.76 | 16 430 |
| selectOne | 500 716 | 10 366 020 | – | 7 147 |
| parallelSelectOne | 500 716 | 1 751 976 | 5.92 | 15 983 |

Table 2 shows that although execution time is significantly improved, the speedup is not consistent with the number of cores. This can perhaps be attributed to the complex architecture of Threadripper, which is effectively four quad-core modules (CCXes) split across two dies on a single chip. The lack of shared cache between each module and CCX switching, which moves threads between cores (which may be on a different die) to balance temperatures and the non-uniform memory access (NUMA) nature, could all potentially contribute to this slowdown. To investigate whether the poor speedups were due to a flaw in our parallel algorithm or experiments, we ran the benchmarks on smaller models (due to memory constraints) using a more conventional system: Intel Core i5-3470 quad-core CPU, 12 GB DDR3-1600 MHz RAM, Windows 10 64-bit v1803. We used the following JVM flags: "*-Xms640m -XX:MaxRAMPercentage=95 -XX:MaxGCPauseMillis=500 -XX:+AggressiveOpts*"

The results in Table 3 show a marked improvement in efficiency of the parallel variants compared to the Threadripper system. We see that our *selectOne*

[8] We intend to perform more runs and remove outliers in future experiments

**Table 3.** Benchmark results for i5-3470 system with 4 threads

| Operation | Model Size | Time (ms) | Speedup | Memory (MB) |
|---|---|---|---|---|
| selectOne | 100 024 | 222 800 | – | 970 |
| parallelSelectOne | 100 024 | 70 471 | 3.16 | 2 043 |
| selectOne | 200 319 | 1 141 585 | – | 967 |
| parallelSelectOne | 200 319 | 326 986 | 3.49 | 1 906 |
| select | 500 716 | 770 363 | – | 2 510 |
| parallelSelect | 500 716 | 324 455 | 2.37 | 3 611 |
| parallelSelectOrdered | 500 716 | 331 500 | 2.32 | 3 911 |
| select | 1 013 510 | 1 863 270 | – | 3 523 |
| parallelSelect | 1 013 510 | 763 902 | 2.44 | 4 759 |
| parallelSelectOrdered | 1 013 510 | 786 753 | 2.37 | 4 007 |

benchmark shows particularly favourable results compared to our *select* one, with a more impressive speedup which increases with the number of elements. Given that both *parallelSelect* and *parallelSelectOne* use the same parallelisation strategy (i.e. the number of jobs submitted to the thread pool is equal to the number of elements), one possible explanation for this disparity in speed-up could be that our *parallelSelectOne* benchmark is more compute-intensive.

In both the 4 and 16-core systems, we see virtually no significant difference in performance between *parallelSelect* and *parallelSelectOrdered*. We also see that our solution appears to have a "speed-up limit" of 6x, since the performance difference between *parallelSelect* and *parallelSelectOne* is much smaller with 16 threads than with 4 threads. We intend to investigate this in future work.

## 6 Conclusions and future work

In this paper, we have motivated the case for the parallel execution of iteration-based operations on collections, arguing that the functional nature of these operations makes them particularly suitable for data parallelism. Furthermore, we have seen how the computational similarity between certain operations gives rise to re-usable patterns for implementing parallel variants of these operations. More specifically, we discovered that short-circuiting predicate logic operations require a different implementation strategy than non short-circuiting operations. Finally, we presented the challenges and solutions in implementing parallel execution algorithms for a diverse set of first-order operations in the context of a feature-rich model-oriented scripting language. Our evaluation showed a significant reduction in execution time, with no perceivable difference between ordered and unordered parallelisation strategies.

A notable omission from our parallel operations is *closure*, which could also be parallelised. The *aggregate* or OCL's *iterate* operation are perhaps the most powerful and complex of the operations, so it would be interesting to provide parallel implementations for a restricted subset of their use cases. We also intend to perform a more thorough evaluation in the future and identify potential

performance bottlenecks. Perhaps more fundamentally we would ideally like to achieve a combination of both parallelism and laziness, as provided by Java Streams API. Future work could therefore investigate how lazy iterator-based collections – as proposed in [5] – can be used in conjunction with the parallel algorithms outlined in this paper, and the associated technical challenges.

## References


1. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: Scalability: The Holy Grail of Model Driven Engineering. In: Proceedings of the First International Workshop on Challenges in Model Driven Software Engineering, Toulouse, pp. 10-14 (2008).
2. Kolovos, D.S., Rose, L.M., Matragkas, N., Paige, R.F., Guerra, E., Cuadrado, J.S., De Lara, J., Ràth, I., Varrò, D., Tisi, M., Cabot, J.: A research roadmap towards achieving scalability in model driven engineering. In: Proceedings of the Workshop on Scalability in Model Driven Engineering, Budapest. Article No. 2 (2013)
3. Java Streams API, `https://docs.oracle.com/javase/10/docs/api/java/util/stream/package-summary.html`
4. Subramaniam, V.: Lets Get Lazy: Explore the Real Power of Streams, Devoxx United States (2017). Available at: `https://youtu.be/F73kB4XZQ4I`
5. Tisi, M., Douence, R., Wagelaar, D.: Lazy evaluation for OCL. In: Proceedings of the 15th International Workshop on OCL and Textual Modeling co-located with 18th International Conference on Model Driven Engineering Languages and Systems, Ottawa. pp. 46-61 (2015)
6. Vajk, T., Dávid, Z., Asztalos, M., Mezei, G., Levendovszky, T.: Runtime model validation with parallel object constraint language. In: Proceedings of the 8th International Workshop on Model-Driven Engineering, Verification and Validation, Wellington. Article No. 7 (2011)
7. Paige, R.F., Kolovos, D.S., Rose, L.M., Drivalos, N., Polack, F.A.C.: The Design of a Conceptual Framework and Technical Infrastructure for Model Management Language Engineering. In: Proceedings of the 2009 14th IEEE International Conference on Engineering of Complex Computer Systems, Potsdam. pp. 162–171 (2009)
8. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: The Epsilon Object Language (EOL). In: Proceedings of the Second European Conference on Model Driven Architecture – Foundations and Applications, Bilbao. pp. 128–142 (2006)
9. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: On the evolution of OCL for capturing structural constraints in modelling languages. Rigorous Methods for Software Construction and Analysis, pp. 204–218. Springer Berlin, Heidelberg (2009)
10. Madani, S., Kolovos, D.S., Paige, R.F.: Parallel Model Validation with Epsilon. In: Proceedings of the 14th European Conference on Modelling Foundations and Applications, Toulouse. pp. 115–131. Springer, Cham (2018)
11. Futures in Java, `https://docs.oracle.com/javase/10/docs/api/java/util/concurrent/Future.html`
12. García-Domínguez, A., Kolovos, D.S., Rose, L.M., Paige, R.F., Medina-Bulo, I.: EUnit: a unit testing framework for model management tasks. In: Proceedings of the 14th international conference on Model driven engineering languages and systems, Wellington. pp. 395–409. Springer Berlin, Heidelberg (2011).
13. LinTra IMDb case study, `http://atenea.lcc.uma.es/index.php/Main_Page/Resources/LinTra#IMDb`