A Pattern-based Transformation Approach to Parallelise Software Systems using a System Dependency Graph

Master's Thesis

Johanna Elisabeth Krause

December 23, 2015

Kiel University
Department of Computer Science
Software Engineering Group

Advised by: Prof. Dr. Wilhelm Hasselbring
M.Sc. Christian Wulf

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

Abstract

Decades ago, the first multi-core processor machines were invented. Meanwhile, most computational devices contain two, four or more processors. Nevertheless, lots of software developers avoid concurrent programming because it requires cautious implementation to prevent race conditions and dead locks. However, without concurrency, the potential of multi-core processors cannot be exhausted.

Therefore, a graph-based parallelisation approach for Java systems is demonstrated in this thesis. We start with a System Dependency Graph (SDG) which represents Java source code and is stored in a Neo4J graph database. As prototypes, we choose three patterns which can become parallelised. With pattern matching, we search for these patterns in the SDG. Then, the identified sub graphs are transformed so that they finally represent the parallel version of the original source code. In the evaluation, we test the pattern matching and transformation by means of several examples which all execute successfully. Also, we analyse the occurrences of the chosen prototype patterns in existing Java applications. We determine that two of the three prototypes exist multiple times in the examined source code.

It is planned to generate Java source code from the transformed SDG in future work.

Zusammenfassung

Mehrkern-Prozessoren wurden schon vor Jahrzehnten erfunden. Heutzutage sind in den meisten technischen Geräten zwei, vier oder mehr Prozessoren verbaut. Dennoch meiden immer noch viele Software-Entwickler und -Entwicklerinnen nebenläufig zu programmieren, da dies viel Aufmerksamkeit erfordert, um Race Conditions und Deadlocks zu vermeiden. Jedoch kann das Potential von Mehrkern-Prozessoren ohne Nebenläufigkeit nicht ausgeschöpft werden.

In dieser Masterarbeit wird ein graph-basierter Ansatz zur Parallelisierung von Java Programmen vorgestellt. Wir gehen von einem Systemabhängigkeitsgraphen aus, der Java-Quellcode darstellt und in einer Neo4J-Graphdatenbank vorliegt. Es werden drei parallelisierbare Entwurfsmuster als Prototypen ausgewählt. Diese Entwurfsmuster werden mithilfe von 'Pattern Matching' – Muster-Abgleich – im Systemabhängigkeitsgraphen erkannt. Die identifizierten Teilgraphen werden nun transformiert, sodass sie schließlich die parallelisierte Version des originalen Quellcodes darstellen. Für die Evaluation testen wir das Pattern Matching und die Transformation anhand verschiedener Beispiele, die alle erfolgreich abschließen. Desweiteren analysieren wir das Vorkommen der ausgewählten Entwurfsmuster in bekannten Java Anwendungen. Wir ermitteln, dass zwei der drei Prototypen mehrmals im untersuchten Quellcode vorliegen.

Es ist geplant in naher Zukunft aus dem transformierten Systemabhängigkeitsgraphen Java Quellcode zu generieren.

Contents

List of Figures

List of Tables

Listings

List of Abbreviations

AST	Abstract Syntax Tree
fqn	fully qualified name
SDG	System Dependency Graph
CMQ	Cypher Match Query
CUQ	Cypher Update Query
PEs	Processing Elements

Chapter 1

Introduction

Parallel and distributed programming is an important subject in today's IT as current hardware is able to execute multiple tasks simultaneously. Hence, parallel programs tend to be faster than their sequential equivalent. However, in case the parallel tasks are dependent on each other or on the same resources, the development becomes more complex: Errors due to race conditions and dead locks as a result of mutual exclusions have to be avoided. Since parallel programming might be more complicated, not all software developers exploit potential parallelism. So, a lot of existing programs only use a single core, thus do not exhaust the underlying hardware resources. These programs contain a high potential of performance improvements through parallelisation.

For example, legacy systems in enterprises are often highly customized and therefore difficult to replace. They are often outdated, but nevertheless, they are critical for the business. A transformation of such a legacy system into a parallelised application would potentially improve the performance of the existing system so that it better fits today's requirements.

1.1 Context

This thesis contributes to the approach of Christian Wulf who develops a semi-automatic framework for parallelising sequential Java source code [Wulf 2014]. In Figure 1.1, his approach is visualised: With the help of static and dynamic code analysis, an SDG of the sequential program is automatically extracted and stored in a graph database (S1-S3). Afterwards, the sub graphs of the source code are ranked in a parallelism plan ordered by the parallelisation potential (S4). For the following transformation, patterns that can be parallelised are specified as well as their parallelised version. We call the sequential pattern which can become parallelised *candidate pattern*; the correspondent target version is entitled *parallelisation pattern*. In the semi-automatic transformation step, the specified candidate patterns are seeked by pattern matching and then transformed into their parallelised version (S5,S6). Finally, Java source code is generated from the transformed graph (S7).

In the following, we make contributions to S5 and S6 of the approach mentioned above.

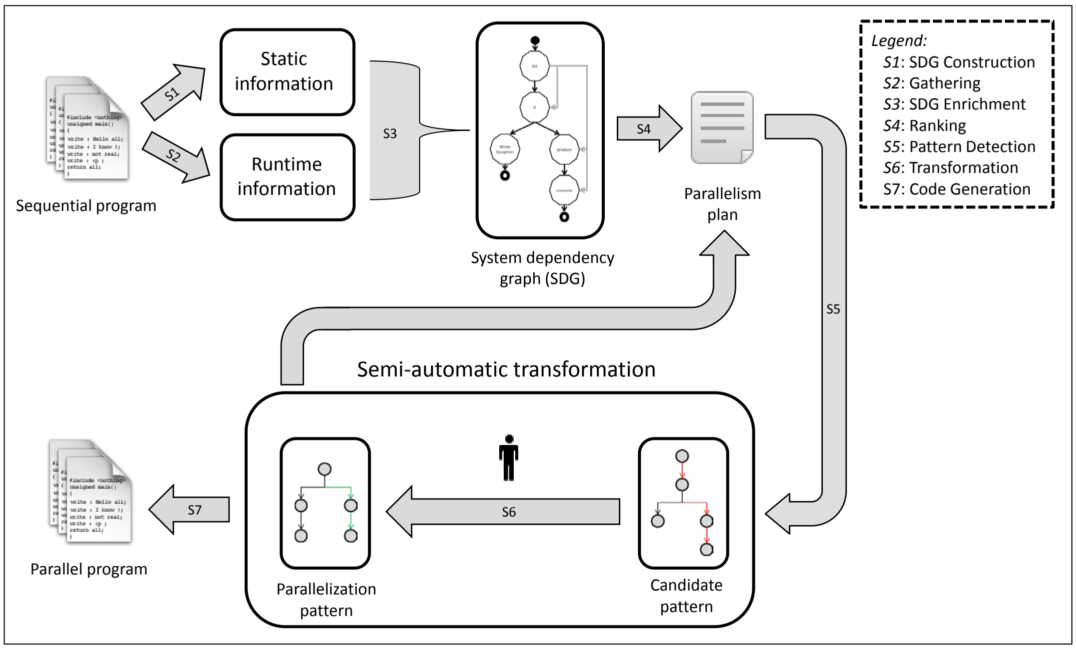


Figure 1.1. Approach: semi-automatic framework for parallelising Java source code [Wulf 2014].

1.2 Goals

The aim of this thesis is the semi-automatic transformation of a graph representing Java source code into its parallelised version. The construction and availability of the source code as a graph in a graph database is a precondition and not part of this work. At first, different patterns have to be mined which can be parallelised. We call these patterns *candidate patterns*. For this thesis, we choose three prototypes for demonstrating the operational capability of our approach. For each of these prototypes, a pattern matching query is formalised with Cypher, which is the query language of the used graph database Neo4J. With these matching queries, those sub graphs are identified which represent parallelisable source code. Then, these sub graphs are transformed so that they represent a parallel version of the originally sequential source code. In the evaluation, we examine the correctness of the pattern matching and the transformation on the basis of several examples. Also, we determine the occurrences of the prototype candidate patterns in two Java applications.

1.3 Document Structure

Subsequent to the introduction, Chapter 2 provides an overview of the related work comprising parallelisation approaches and graphical representation of source code. Then, Chapter 3 shortly describes our approach. In Chapter 4, we present fundamental knowledge and technologies necessary for understanding the thesis' topics. In Chapter 5, the process of mining suitable pairs of candidate and parallelisation patterns is described. Chapter 6 examplifies the formulation of a pattern matching query in Neo4J's query language Cypher for identifying the occurences of the candidate pattern in an SDG. The transformation from a candidate pattern to the corresponding parallelisation pattern is explained in Chapter 7.

Afterwards, in Chapter 8, the feasibility of our approach is evaluated and the number of occurrences of the candidate patterns is examined. Finally, the conclusions follow in Chapter 9.

Chapter 2

Related Work

This chapter summarises the related work to this thesis. It mainly divides into research about design pattern detection in graphs and about automatic parallelisation approaches. The combination of these areas recently emerged and deserve further studies.

2.1 Pattern Detection in Graphs

Heuzeroth et al. [2003] published their research about *Automatic Design Pattern Detection* in Java programs [Heuzeroth et al. 2003]. Based upon the Observer Pattern, they present their approach which aims for the better understanding of the software design and architecture. Firstly, they construct an attributed Abstract Syntax Tree (AST) with the help of a static code analysis tool called 'Recorder'. Then, they collect 'candidates' from the AST. Each candidate is represented as a 'tuple of AST nodes with the appropriate static structure'. For instance, the tuple for a candidate of the Observer Pattern consists of method declarations of the form: `(S.addListener, S.removeListener, S.notify, L.update)`. The identified parts of the candidates are instrumented for the following dynamic analysis. During the dynamic analysis, the candidates are observed and monitored whether they behave conform to the 'protocol' of this design pattern. Candidates that behave contrary to the protocol are excluded from the candidate list. As a result, it remains a list of candidates which probably implement the design pattern and a list of candidates which were not executed during the dynamic analysis so that no decision is possible on those. Similar to our approach, the specification of the patterns is implemented for each pattern separately. With the help of a specification language, Heuzeroth et al. [2003] define the static and dynamic analysis patterns. As future work, they suppose the integration of data flow analysis and naming conventions into the static analysis [Heuzeroth et al. 2003]. In contrast to Heuzeroth et al. [2003], we focus on patterns that can be parallelised, also we work with an SDG instead of an AST and we include data flow analysis. Our pattern matching completely works on the SDG which contains all necessary data including runtime information.

Similarily, Stencel and Wegrzynowicz [2008] also use the 'Recorder' tool to construct an AST from Java source code for their work *Detection of Diverse Design Pattern Variants* [Stencel and Wegrzynowicz 2008]. From the AST, they evolve a metamodel which consists of core elements and relations. Elements are types, methods, or instances, whereby relations represent inheritance, calls, and assignments to variables. The metamodel is stored in a relational database. So, pattern matching is done with SQL. Again, this work focuses on design patterns in general and not on parallelisation. However, in contrast to Heuzeroth et al. [2003], Stencel and Wegrzynowicz [2008] use no dynamic analysis but data flow information to identify design patterns. Also, their approach additionally detects variants of the patterns. As distinguished from our approach, the metamodel of Stencel and Wegrzynowicz [2008] also contains instances. However, our approach considers runtime information and the storage inside a graph database allows a convenient and recursive pattern matching. Nevertheless, our both approaches aim at not only applying Java programs but all object-oriented programming languages, as the representation of the source code is not language specific.

Another motive for pattern detection in graphs is identifying and resolving bugs. Eichinger et al. [2008] have done a lot of research on finding patterns which indicate faults in reduced call graphs [Eichinger et al. 2008; 2010, a; b; 2014]. They construct call graphs for correct executions of the program as well as for failing executions. Then, they discover frequent sub graphs inside the call graphs with the CloseGraph algorithm of Yan and Han [2003]. By comparing the differences between the frequently accessed sub graphs from the correct and failing executions, suspicious methods are identified. They further developed their approach. They added data flow analysis and also analysed parallel programs for race conditions and dead locks.

The research of Sun et al. [2010] is more related to our approach. They construct an SDG from programs in C/C++ with the tool CodeSurfer[1]. So, all static information including data dependencies are regarded. In the SDG, they search for self-defined 'bug patterns'. This is done by graph matching using the graph matching algorithm GADDI [Zhang et al. 2009]. The detected bug patterns are then transformed to the corresponding 'fix patterns'.

2.2 Automatic Parallelisation

Several automatic parallelisation approaches exist. This section presents related work for automatic parallelisation which is not based on graphs. Already in 1991, Merlin has published his work about *Techniques for the Automatic Parallelisation of 'Distributed Fortran 90'* [Merlin 1991]. He has developed the automatic parallelisation based on array partitioning for the programming language Fortran. Likewise, Hall et al. [2005] deal with the parallelisation of Fortran programs. They present a parallelisation approach for the compiler especially focussing on parallelising loops. They use interprocedural array data-flow analysis, array privatisation and reduction recognition. As the parallelisation happens at compile time, purely static analysis is applied. In comparison, our approach also uses runtime information to detect parallelisable pattern. Also, we are not restricted to loops and to Fortran, but focus on object orientation.

Servetto and Potanin [2012] present their research about *Automatic Parallelisation in OO Languages with Balloons and Immutable Objects* [Servetto and Potanin 2012]. It is based on

[1]CodeSurfer: http://www.grammatech.com/research/technologies/codesurfer

the language 'Ballon Immutable Java' which contains specific types of objects: ballons and immutable objects. The balloons are used as a mechanism for restrictive encapsulation whereas immutable objects are exclusively referenced read-only. The original program is reduced to a version containing balloons and immutable objects where possible. Due to their characteristics, balloons and immutable objects allow parallel execution without interference. So, these locations are suitable for parallelisation. During compilation, the program is parallelised at the identified locations by creating fork-join blocks. Servetto et al. [2013] amend their research by adding ownership which allows more parallelisation. For our parallelisation approach, a Java program is represented as an SDG. In the SDG, we look for parallelisable patterns considering data dependencies. We do not differentiate between separate instances. However, we employ semi-automatism, hence, we involve the developer. In contrast to a fully automated parallelisation during compile time, this proceeding provides more parallelisation potential.

Similar to our approach is the research of Molitorisz et al. [2012]. They use the static code analysis tool Soot for detecting parallelisable patterns like asynchronous method calls or loops without data dependencies. For the parallel implementation, they choose Futures as an easy and readable solution. Futures represent a future value and hide if it is already available or not. The parallelisation is executed fully automatically. This limits the parallelisation potential but therefore experiences a high acceptance with concerned developers. Molitorisz et al. [2012] plan to add runtime information in future work. For our approach, we also use Soot for the static analysis, however, we construct the SDG from the provided call graph. So, we can work with the graph to identify parallelisable patterns. We assume, that our SDG is already enriched with runtime information to detect high parallelisation potential.

2.3 Graph-based Parallelisation Approaches

Moseley et al. [2007] present their approach for identifying parallelisable loops. They build a loop call graph which shows the nesting of loops because outer loops have a big parallelisation potential. These profile information can help developers with manual parallelisation. However, Moseley et al. do not examine data dependencies and race conditions, neither do they automatically parallelise the identified loops.

The work of Asenjo et al. [2008] is more comprehensive. They present an approach for the automatic parallelisation of source code based on heap-stored data structures. Therefore, they abstract the data structures by *shape graphs* to model the heap. This shape graphs represent possible connections of heap elements. So, interprocedural data dependencies are detected and loop-level parallelism is exploited. This parallelisation framework is integrated into Cetus, a compiler for science and engineering applications in C. Bae et al. [2013] evalute Cetus. As an internal representation, Cetus features a graph representation similar to Java's class hierarchy. It is traversable and comprises also statements. From the internal representation, array access-related and loop-related information are retrieved beside other data dependencies. Also, induction (e.g. $k = k + i$) is detected and substituted where possible. Recognising these operations is necessary for identifying many loops. The parallelisation is executed fully automatically. For the profitability, only loops with a high workload during runtime are parallelised. For this purpose, a phase of profiling follows to examine the profitability of the parallelisation which compares several executions with different combinations of sequential and parallel code snippets. In contrast to Bae et al. [2013], we aim at a semi-automatic parallelisation approach for exhausting more parallelisation potential. Also, we work on an SDG as a source code representation which is stored in a graph database so that we can parallelise all sorts of patterns and not only loops. We do not strive for profiling several parallelisation combinations. However, we assume that our SDG is enriched with runtime information so that we can focus on source code parts with high performance improvement potential.

Chapter 3

Approach

The overall approach of this thesis is visualised in Figure 3.1. First of all, pairs of candidate and parallelisation patterns have to be identified (1). Then, the candidate patterns are formalised as Cypher Match Queries (CMQs) to find matching source code sub graphs (2). The resulting sub graphs are transformed into their parallelised form with the help of Cypher Update Queries (CUQs) (3). We evaluate the feasibility of our approach on the basis of three candidate patterns. Also, we examine the occurrences of the chosen candidate patterns in e applications (4). For the future, it is planned that after processing all favoured patterns, Java code is generated from the graph. Then, the result could be evaluated by comparing the performance of the transformed application and the original one.

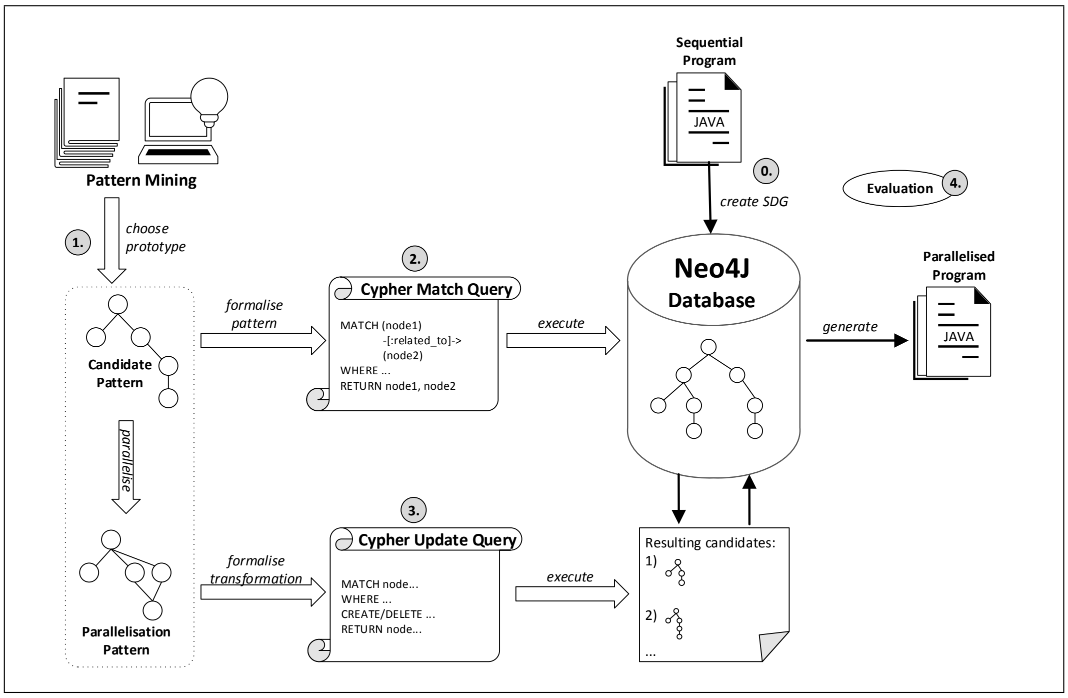


Figure 3.1. Approach of this thesis

Chapter 4

Foundations and Technologies

4.1 Graph-based Representation of Source Code

Mathematically, a *graph* G is defined as a tuple of two sets V– the set of nodes (vertices) – and E – the set of edges between the nodes. Thus, $G = (V, E)$ and $E \subseteq (V \times V)$. Figure 4.1 shows a directed example graph [Edlich et al. 2011, p.209].

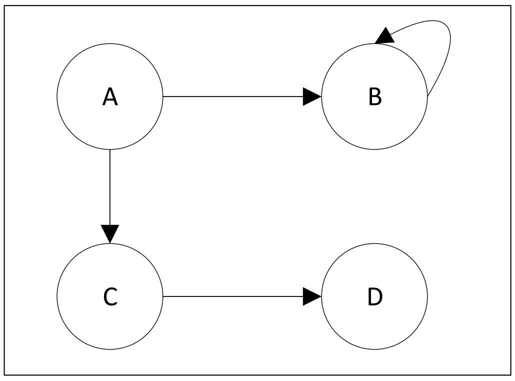


Figure 4.1. Example of a directed graph: $G = (V, E)$ with $V = \{A, B, C, D\}$ and $E = \{(A, B), (A, C), (B, B), (C, D)\}$

A *property graph* is a graph extended by properties – key-value pairs – which are added to the nodes and optionally also to the edges. For source code representation, a suitable property for a node is, for instance, 'code' as key with the correspondent piece of source code as value. Nodes and edges can have multiple properties, so that all relevant information can be attached for later use. Concerning graph databases, in case a node shall be deleted, all the edges from and to the node have to be deleted, too [Robinson et al. 2015, p.4].

In the following, three types of graphs visualising source code are described:

Call Graph Call graphs consist of nodes, which represent static calls, and directed edges, which represent all static calls from one method to another [Grove and Chambers 2001] [Ryder 1979]. Hence, it shows inter-procedural calling relationships.

Control Flow Graph A control flow graph contains nodes which represent linear sequences of program instructions and directed edges representing control flow paths [Allen 1970]. The edges describe the chronological order of the program execution. As the control flow graph shows single statements in a method, it is intra-procedural.

System Dependency Graph (SDG) An SDG combines a call graph and a control flow graph for each method and additionally represents data dependencies. Figure 4.2 shows a property graph which is a small example of an SDG.

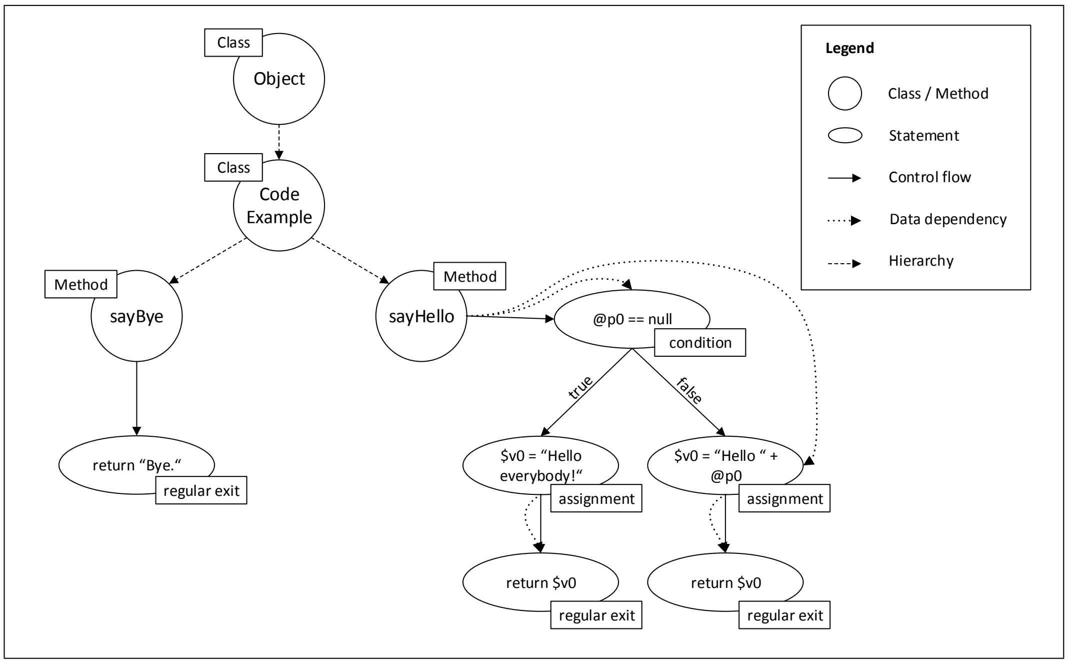


Figure 4.2. Example of a labeled property graph representing an SDG

In this thesis, we operate on SDGs. In the following, the nodes and relationships of an SDG are presented.

4.1.1 Nodes of an SDG

The nodes of an SDG can be categorised depending on what they represent: package declarations, class declarations, method declarations, field declarations, and statements.

These categories form a hierarchy which is also visualised in Figure 4.3. The package declarations are at the top. They contain other package declarations and class declarations. The class declarations contain method declarations and field declarations. Constructors

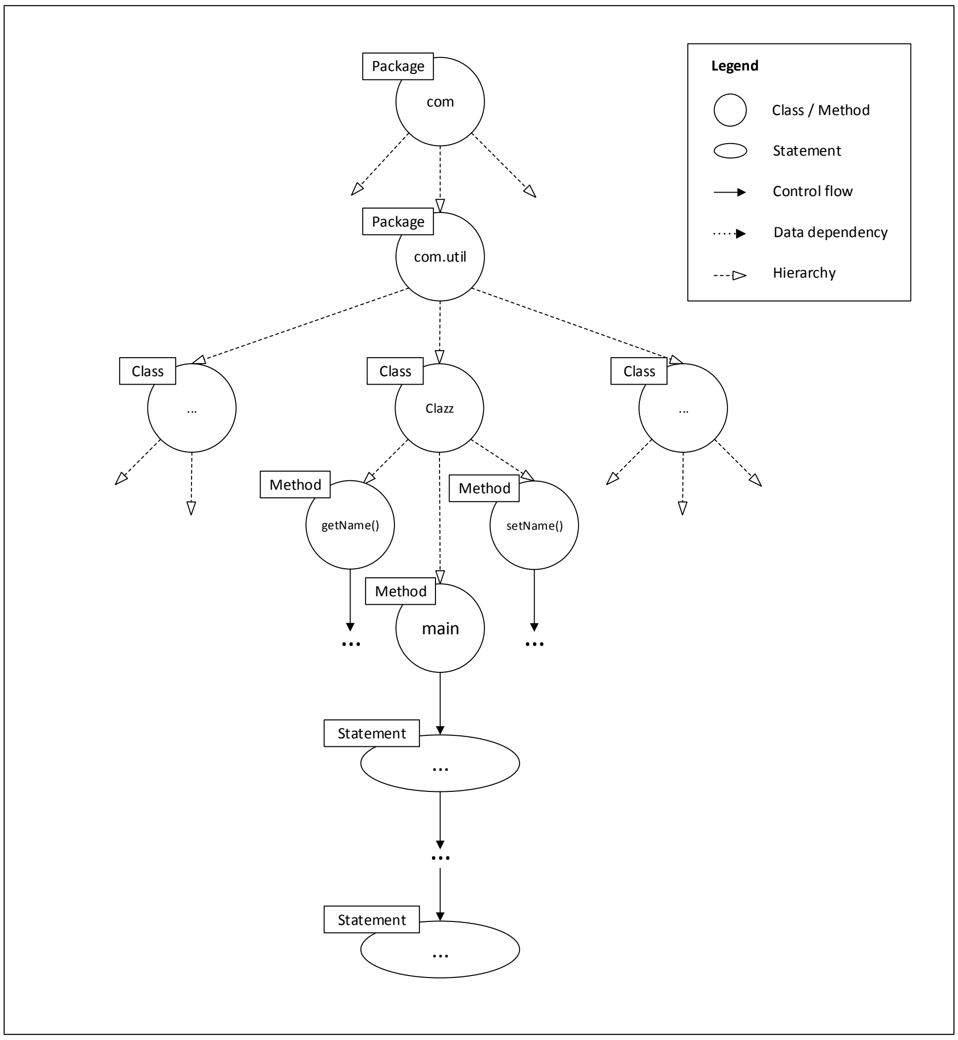


Figure 4.3. Hierarchy in an SDG

can be either represented as methods or concrete as an additional category. The method declarations contain statements. Statements represent the actual source code inside methods and constructors. They can be further distinguished according to their functionality: 'assignments' if a variable is assigned, 'method calls' if a method is called, 'condition' for if-conditions and switch cases, and eventually 'loop' for the head of a loop. The subtypes of the statements are not exclusive, but can appear combined.

4.1.2 Relationships of an SDG

The nodes are connected by edges representing different types of relationships. At least one relationship type is needed to describe the hierarchy inside the SDG, hence which methods belong to which classes and which classes belong to which package. This type can be named 'contains'.

The statements inside a method are connected by 'control flows'. In most cases, a statement is followed by exactly one other statement. However, in case of if-conditions, switch-case-expressions and loops, multiple possible control flow edges lead to different statements. Therefore, a property is useful to distinguish the different routes, e.g. 'case'. When a statement embodies a method call, the statement should link to the called method as it represents the forwarding of the control flow. This relationship type can be called 'calls'. For representing data dependencies, another relationship type is required which represents the 'data flow'. It should be distinguished between reading and writing data flow. Also, it has to be recognisable to which object the data flow belongs.

With the hierarchy relationship type and the control flows alone, the source code can already be depicted unambiguously. The graph would look like several trees with packages as nodes, spreading into classes, then methods and fields and finally statements. The representation of the method calls and data dependencies creates connections between the sub trees. They can link directly the statement and the method, and accordingly the statement and field, or statement and statement. Additionally, the connections can be shown on a higher level for example from method to method or from class to class.

4.2 The Graph Database Neo4J

Among the wide-spread relational databases where the data is structured in tables, other kind of database models have emerged – like graph databases, document-stores or key-value-stores. For representing data with numerous and frequently changing relationships, relational data models struggle with complex and inefficient join-operations and restricted schema-changes. In contrast, graph databases like Neo4J focus on the relationships between data and therefore store data as nodes and edges with various properties – hence as property graph model. *Nodes* represent the entities in the database, whereas the *edges* show the relationships between the nodes. For this reason, in the context of Neo4J, edges are called *relationships*. The *properties* hold information about the nodes, compared to relational database models, they can be seen as records in a table [Neo Technology 2015a].
Neo4J is ACID compliant. Hence, the four important properties of database management systems are satisfied: atomicity, consistency, isolation and durability. This is achieved by transactions. A transaction is either fully executed or rolled back, so that the state of the database is always consistent [van Bruggen 2014, p.45f].

Neo4J does not require a database schema, nevertheless it is reasonable to constitute semi-structured data. For this purpose, nodes can be semantically categorised by assigning different *labels*. A node can hold zero, one or more labels [van Bruggen 2014, p.34ff]. For the SDG, suitable labels are e.g. 'Class' for class declarations, 'Method' for method declarations and 'Statement' for single statements. A statement which represents an assignments as well as a method call, two labels can be assigned. The concrete representation of the SDG processed in this thesis is described in Section 4.3.2.

Nodes as well as relationships can contain multiple properties. Properties are key-value pairs. The key is represented as string, whereas the values can be numeric, a string, a boolean, or an array of the mentioned values [Robinson et al. 2015, p.156]. For class and method definitions suitable properties could be 'fqn' containing the fully qualified name. Whereas statements could embody 'code' or 'operation' and 'varName'. Similar to relational databases, operating on normalised data is recommended as it simplifies comparing of properties. In an SDG, a node representing the source code `int i = calculateFirstIndex()` might be represented with the labels 'Assignment' as well as 'MethodCall'. The properties could be `{code: 'int i = calculateFirstIndex()'}`, `{assignedVar: 'i'}`, `{vartype: 'int'}`, and `{fqnCalledMethod: 'package.Class.calculateFirstIndex()'}`.

Similar to the nodes' labels, relationships are assigned to a *type*. However, relationships may have neither multiple types nor no type at all. They require exactly one type in Neo4J. Yet, it is possible to have multiple relationships between the same nodes. The relationship types needed for an SDG are primarily 'CONTAINS', 'CONTROL_FLOW', 'DATA_FLOW', and 'CALLS'. An important property is 'case' for indicating different possible control flows, e.g. for if-else-conditions.
The detailed structure of the SDG which is used for this thesis is presented in Section 4.3.2.

For indexing, Neo4J uses the text search engine library Apache Lucene. It is an open source Java project providing efficient search algorithms [Edlich et al. 2011, p.290].[1] In Neo4J, indexes can be defined for nodes by naming the concerning label and the property to index [Neo Technology 2015b, p.236].

Neo4J offers several interfaces for providing the graph database. For the purpose of this thesis, we work on a locally installed Neo4J graph database and focus on its Java API [Neo Technology 2015b, p.10]. For development, we additionally use the web browser front end where queries can be executed and the result is presented as a graph. However, the graphical embodiment for large graphs is not clearly arranged, so most graphs in this thesis are designed manually.

In the following, Neo4J's query language Cypher as well as the Java API is explained.

4.2.1 Cypher

Cypher is the query language of Neo4J. It is inspired by SQL *inter alia*, but optimised for describing graph patterns.

In software engineering, design patterns describe a specific context, a problem and a solution for recurring standard problems [Ortega-Arjona 2010, p.382]. It is an abstraction of the source code so that it can be compared to common design structures. A graph pattern is a specified composition of nodes and relationships which represents different characteristics.

Nodes are depicted in round brackets, whereas the relationships are phrased similar to arrows.[2] The following Cypher queries exemplify the representations of a node and a relationship in Cypher:

```
(nodeName:Label {propertyname1: value1, propertyname2: value2})

-[relName:REL_TYPE {propertyname1: value1}]->
```

Hence, labels of a node, and the type of a relationship start with a colon; properties are listed in curly brackets and the nodes and relationships can be named with a variable for referencing inside this query. Depending on the pattern, not all information has to be provided. For example, a pair of round brackets `()` is a valid expression for an unnamed node [Neo Technology 2015b, p.125ff].

An example of a complete Cypher query is shown in Figure 4.4 which creates a small graph with three nodes and two relationships [Neo Technology 2015b, p.125ff].

The three nodes are referenced by *n1*, *n2*, and *n3* for creating the relationships later in the query. They all have exactly one property which is visualised in the graph and the different labels are expressed by the different colours of the nodes. The RETURN statement declares which data is needed as result – similar to the SELECT key word in SQL. The

[1]Apache Lucene Core: https://lucene.apache.org/core/

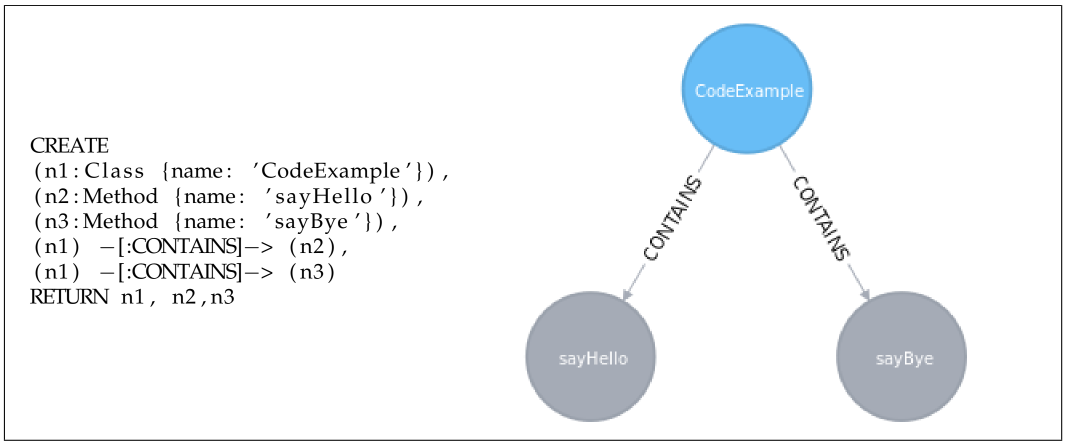
[2]Neo4J Technologies: *http : //neo4j.com/docs/stable/cypher − introduction.html*

keyword CREATE indicates that the following pattern has to be created. If it is MATCH instead, the query phrases a search query looking for the pattern in the graph database.

In this thesis, we distinguish two types of patterns: We refer to patterns representing sequential source code which can be parallelised as *candidate patterns*. The corresponding pattern representing the parallelised source code is called *parallelisation pattern*. According to this distinction, we also differentiate between a CMQ and a Cypher Update Query (CUQ). The *CMQ* describes the graph of a candidate pattern to identify matching sub graphs in the SDG, whereas the *CUQ* formulates the transformation from the candidate pattern to the parallelisation pattern.

Cypher Match Query

The CMQ is a reading query looking for sub graphs matching the candidate pattern. A CMQ is constructed with the keywords *MATCH* followed by the form of the candidate pattern and *RETURN* which indicates the results to display, e.g. nodes or properties. An optional *WHERE*-clause can be used for restricting, for example on some properties. With the keyword *NOT*, an expression can be negated [Neo Technology 2015b, p.104ff]. In Listing 4.1, a CMQ is shown which looks for 'MethodCalls' that take at least 500ms and returns the ids of the matching nodes. The WHERE clause checks the property 'durationInMs' of the node *n*. If *n* does not have the property 'durationInMs' or it is smaller than 500, then *n* does not match the pattern. The function `id(n)` returns the id of the node *n*.



```
CREATE
(n1:Class {name: 'CodeExample'}),
(n2:Method {name: 'sayHello'}),
(n3:Method {name: 'sayBye'}),
(n1) -[:CONTAINS]-> (n2),
(n1) -[:CONTAINS]-> (n3)
RETURN n1, n2,n3
```

Figure 4.4. Example of a Cypher query creating a simple graph

```
MATCH
 (n:MethodCall)
WHERE
 n.durationInMs >= 500
RETURN id(n)
```

Listing 4.1. Example CMQ

```
START n=node(10)
MATCH
(beforeN) -[r1:CONTROLFLOW]-> (n) -[r2:CONTROLFLOW]-> (afterN)
WITH beforeN, afterN, r1, r2, n
CREATE (new:MethodCall {code: 'Magic.doSomeMagic()'})
DELETE r1, r2, n
CREATE (beforeN) -[:CONTROLFLOW]-> (new)
CREATE (new) -[:CONTROLFLOW]-> (afterN)
```

Listing 4.2. Example CUQ

Cypher Update Query

The CUQ is a reading and writing query. For transforming the graph, the keywords *CREATE* and *DELETE* are used for creating and deleting nodes and relationships. In contrast, the keywords *SET* and *REMOVE* are used for setting and removing labels and properties. Typically, a CUQ starts with a matching query to identify the right location for the changes. The keyword *WITH* i.a. separates the reading part of the query from the writing part. WITH is followed by the variables that are needed in the next part of the query [Neo Technology 2015b, p.105].

The optional starting expression *START n = node(givenId)* sets the start node for the pattern matching to the node with the given id. For this query, that node is bound to the variable *n* so that it can be reused. If a starting node is known, it saves time for exhaustive pattern matching. Additionally, it ensures the location is right, in case several sub graphs match the pattern.

Listing 4.2 shows a CUQ which is used to delete the node with the id 10 and inserts a newly created node instead.

When deleting a node, all incoming and outgoing relationships have to be deleted in advance. Hence, in this example, it is assumed that the node *n* does not have more relationships than one incoming and one outgoing 'CONTROLFLOW'. For example, if *n* has an additional 'DATAFLOW', the query will terminate with an error. If *n* has more than one 'CONTROLFLOW's per direction, e.g. because of an if-then-else branch, the query would match several times and create for each match a new node.

4.2.2 Neo4J's Java API

The Neo4J community offers a Java library for managing the graph database via Java which is well documented in the Neo4J Manual [Neo Technology 2015b, p.596-625] and as Javadoc.[3] This section is based on these documentations.

The access to the database is realised with the interface `GraphDatabaseService`. The following statement shows the default creation of this fundamental access point:

```
GraphDatabaseService graphDb = new GraphDatabaseFactory()
                                  .newEmbeddedDatabase( "path/to/graphDB" );
```

The assigned path specifies the location of the database on the file system. In case there already exists one, the database server is started; if there is none, a new, empty database is created. The instance of the `GraphDatabaseService` locks the database, so that only one instance at a time can exists. However, since starting the database server is an expensive operation, it should not be started and shutdown again for each interaction with the database.

After the instantiation of the `GraphDatabaseService`, it is recommended to register a shutdown hook for ensuring the server is shutdown properly when the JVM exits:

```
Runtime.getRuntime().addShutdownHook( new Thread()
{
    @Override
    public void run()
    {
        graphDb.shutdown();
    }
} );
```

All interactions with the Neo4J graph database need to be executed within a *transaction*. Each `Transaction` is bound to its creating thread, thus the current instance of the `GraphDatabaseService` can be shared by multiple threads. Listing 4.3 shows the structure of a `Transaction`. If an exception is thrown in the `try`-block, the method `success()` will never be invoked. When leaving the `try`-block, automatically the `Transaction`'s `close()` method is invoked which will commit the transaction if it was successful and otherwise mark it for rollback.

```
try ( Transaction tx = graphDb.beginTx() )
{
    // Database operations go here
    tx.success();
}
```

Listing 4.3. Structure of a `Transaction`

[3]Neo4J Community: *http : //neo4j.com/api$_d$ocs*/2.0.3/

Exclusively within a `Transaction`, database operations can be executed. Database operations can either be formulated as a Cypher query or with the help of Neo4J's Java API which provides i.a. Node and Relationship objects.

Executing a Cypher query and retrieving the result is implemented as shown in Listing 4.4. The `Result` is an iterator and contains the data specified in the query's *RETURN* clause. Each element in the `Result` represents a row which is structured as a `Map` and contains the entries of each column. Each row has the same columns; the keys are the column titles from the query's *RETURN* clause. The `Result` must be either consumed completely or the method `close()` should be invoked to free the resource.

```
String query = " ... ";
try ( Transaction notNeededTx = db.beginTx();
     Result result = db.execute( query ) )
{
    while ( result.hasNext() )
    {
        Map<String,Object> row = result.next();
        for ( Entry<String,Object> column : row.entrySet() )
       {
           // retrieve information from column
       }
   }
}
```

Listing 4.4. Executing a Cypher query from Java

Neo4J's Java API also provides an object-oriented representation for nodes and relationships in the graph database. The interfaces `Node` and `Relationship` can be comfortably used inside transactions for managing nodes, properties, and relationships. This is especially useful, as the variables in a Cypher query only exist in that query. Referencing nodes and relationships outside of that query requires a lookup through pattern matching. Embedded in Java, the relevant nodes and relationships can be stored for later use. The following example shows the creation of a relationship between two `Node` objects.

```
node1.createRelationshipTo(node2, RelTypes.CONTROL_FLOW);
```

When the database server is not needed any more, it has to be shutdowned properly:

```
graphDb.shutdown();
```

4.3 From Java to Neo4J

Our parallelisation approach operates on an SDG which represents Java source code and is stored in a Neo4J graph database. The translation from Java into a Neo4J graph has been developed by Christian Wulf [Wulf 2014]. As the SDG representation is the basis for this thesis, the proceeding for the translation is briefly described in the following. Firstly, the Java optimisation framework Soot is used which generates a call graph and a program dependency graph for each method. Secondly, Soot's call graph is traversed and a Neo4J representation of the call graph is created.

4.3.1 Soot – The Java Optimisation Framework

Soot[4] was originally developed as a Java optimisation framework. Meanwhile, it is used for several purposes, e.g. analysing, instrumenting, optimisation, and visualisation of Java and Android applications. Soot supports Java up to version 7. It currently does not support Java 8 because its source code frontend extends JastAdd whose custom bytecode parser cannot handle Java 8 code completely correct.[5]

As an intermediate representation, Soot provides i.a. *Jimple*. Jimple is a typed *3-address code* which was developed for optimisations.[4] Statements in 3-address code do not contain more than three components: one assignment and one operation which might consists of two components. Hence, nested method calls are dissolved. For instance, the expression $x = y \circ z$ is in 3-address form where $\circ$ represents the operation. After the original source code is transformed into Jimple, the corresponding call graph is constructed. The call graph is amended by the information of a points-to analysis, and intra- and inter-procedural data-flow analysis so it contains all information of an SDG [Vallée-Rai et al. 1999].

4.3.2 From Soot to Neo4J

On the basis of the SDG constructed by Soot, the information is copied into a Neo4J graph database. Therefore, the call graph is traversed and the nodes and edges are created as needed for the following analysis. When the SDG is available in the graph database, by means of Cypher and the Neo4J's Java API, patterns can be sought and the SDG can be manipulated.

For the representation of the source code as SDG in Neo4J, several different node and relationship types are used which are presented in the following.

In Cypher, the types of nodes are represented by labels, where each node can have multiple labels. The node types of our SDG with the corresponding labels are shown in Table 4.1. The nodes with the labels `Class`, `Interface`, and `Method` are constructed from the class, interface and method declarations. However, in combination with the relationships, they represent the overall class, interface and method and not only the

[4]Soot: http://sable.github.io/soot/
[5]Soot, Issue 394: https://github.com/Sable/soot/issues/394

declaration. The labels `Constructor` and `ConstructorCall` are assigned additionally to `Method` and `MethodCall`. The type statement is used for representing the source code inside methods and constructors. According to the different expressions, different sub types of statements are distinguished: especially 'assignments', 'method calls', and 'condition'. The sub types of the statements are not exclusive, but are combined appropriate to the represented source code statement.

Table 4.1. Cypher label for different node types

Node Type	**Cypher Label**
package	Package
class	Class
interface	Interface
field	Field
method	Method
constructor	Constructor
statement, assignment	Assignment
statement, method call	MethodCall
statement, constructor call	ConstructorCall
statement, condition e.g. if	Condition
return statement	ReturnStmt
throw statement	ThrowStmt

The nodes are connected by edges representing their relationships. Table 4.2 shows the different types of relationships and their use.

The aggregated relationships on method declaration level optimise and simplify the pattern matching. The `AGGREGATED_CALL`s represent the method calls inside the method whereas the `AGGREGATED_FIELD_WRITE`s and `AGGREGATED_FIELD_READ`s represent data flow.

For performance and optimisation reasons, only the classes and methods of the regarded application are analysed. The method definitions of the jdk and other external libraries are included in the SDG, but without statements. As a consequence, the methods do not indicate any aggregated calls and aggregated data flows. For avoiding errors due to missing information, all classes and method declarations are marked with the property `'origin'`. For the analysed nodes of the application, the property `'origin='APP''` is added, whereas the jdk and other not analysed external libraries are marked e.g. `'origin='jdk''`.

The basic idea of our approach is to find specified patterns in the SDG representation of Neo4J and then transform the graph to its parallelised version. It is planned that after the SDG is transformed, Java source code is generated again which includes the adjustments made to the SDG. Therefore, the translation process is inverse: Firstly, the SDG has to be transformed back to the Soot graph. Secondly, Soot is used for generating Java source code from the call graph. The generation of Java source code from an SDG in Neo4J is not part of this thesis.

Table 4.2. Relationship types and their utilisation

Relationship Type	**Utilisation**
CONTAINS_TYPE CONTAINS_METHOD CONTAINS_CONSTRUCTOR CONTAINS_FIELD	packages contain classes classes contain methods classes contain constructors classes contain fields
EXTENDS IMPLEMENTS	connects super class and sub class connects interface and implementing class
CONTROL_ FLOW	represents the control flow connects statements connects the method declarations with its first statement
DATA_ FLOW	represents the data flow shows data dependencies
CALLS	represents the static call to a method or constructor
CALLER_ OF	represents that subsequently from the assigned object, a method is called always in conjunction with data flow
AGGREGATED_ FIELD_ WRITE AGGREGATED_ FIELD_ READ AGGREGATED_ CALLS	represents writing field access on method declaration level represents reading field access on method declaration level represents static calls on method declaration level
THROWS	represents throw exception expression on method declaration level

4.4 Characteristics of Parallel Programs

The following section deals with typical challenges in parallel programming. Section 4.4.1 deals with data sharing. Then, Section 4.4.2 continues with race conditions which result from incorrect synchronisation of shared data.

4.4.1 Data Sharing

In parallel programs, often data needs to be shared between concurrent processes. Shared data can be categorised in three categories: read-only, effective-local and read-write.

When data is *read-only*, its status is not changed, so we do not need any access protection. Also, when the data is *effectively-local*, no access protection is needed because on each data chunk is just operated by one task. For instance, the access to fields of an array can be split, so that the access to the overall array is shared, but the separate fields are accessed by only one process. The main challenge in shared data is *read-write* data – data which is read and modified by different tasks. For guaranteeing that shared read-write data is valid and available, a huge synchronisation overhead might be necessary [Mattson et al. 2004, p.46f][Ortega-Arjona 2010, p.79].

Mattson et al. [2004] present two common read-write data problems which are referred to as 'reduction' and 'multiple-read/single-write'.

The reduction scenario, also mentioned as accumulation, describes patterns where the entries of a data structure are accumulated to a single value. In functional programming, this procedure is called 'map'. Common reductive functions are *sum*, *minimum* or *maximum*, however every associative function is possible. For solving a reduction concurrently, the calculation is decomposed to separate parts of the data structure. For each part of the calculation, an intermediate result is computed. The combination of the intermediate results yields the final result [Mattson et al. 2004, p.46] [McCool et al. 2012, Sec.5.1]. Multiple-read/single-write accesses mean that shared data is written by only one task, whereas other tasks read the initial value. In this case, the shared data should be copied. One copy is accessed by the read-only tasks providing the initial value, the other copy is exclusively for the read-write task. The copy containing the initial value can be discarded after use [Mattson et al. 2004, p.46].

In this thesis, no pattern is formalised which fits the multiple-read/single-write criteria, however in future work, their potential should be exhausted. Since we start from a sequential program, the behaviour must not change, therefore the order of the reads and writes must not change either. So, a parallelisation with the above explained multiplication of the shared data is possible when in the sequential program part, only the last task is modifying the shared data.

In most cases and in this thesis, we can only parallelise patterns without read-write dependencies. Filtering read-write dependencies is demonstrated in the pattern matching phase in Section 6.2.

4.4.2 Race Conditions

In concurrent programs, time-dependent failures can occur if the accesses to shared data is not properly synchronized. These failures are called 'race conditions'. Andrews [2000, p.653] describes race conditions as a 'situation in a shared-variable concurrent program in which one process writes a variable that a second process reads, but the first process continues execution – namely races ahead – and changes the variable again before the second process sees the result of the first change. This usually leads to an incorrectly synchronized program.' Netzer and Miller distinguish two different types of race conditions: General races and data races. '*General races* cause no-ndeterministic execution and are failures in programs intended to be deterministic. *Data races* cause non-atomic execution of critical sections and are failures in (non-deterministic) programs that access and update shared data in critical sections.' [Netzer and Miller 1992] *Critical sections* are blocks of source code which need to execute as if they were atomic. Thus, only one process at a time shall execute the critical sections and modify concerned shared data. So, the final state of variables in the critical sections should only depend upon their initial state and the operations by that section's source code.

As data races concern the execution of critical sections, they are *local* properties of the execution. In contrast, general races are *global* properties of the program. Detecting general races requires analysing all possible execution orderings of the program because they occur if the order of the concurrent tasks has influence to the result. In parallel programs normally the scheduling happens automatically. Since the scheduling varies, different behaviour of non-deterministic programs can be observed – even on the same hardware and same data. Hence, general races cannot easily be reproduced, so detecting them by debugging is difficult. According to Netzer and Miller, the detection of general races is more difficult than the detection of data races [Netzer and Miller 1992].

Race conditions are failures and have to be avoided. As race conditions appear when shared data access is not correctly synchronised, candidate patterns with data dependencies require special attendance [Mattson et al. 2004, 17f]. The different types of shared data were presented in Section 4.4.1. In this thesis, we focus on candidate patterns without read/write dependencies.

Chapter 5

Candidate Pattern Mining

Our parallelisation approach is based on pattern matching on an SDG. Therefore, potential patterns have to be identified, i.e. mined. In this chapter, the mining of candidate patterns is described. We start with the theoretical background of finding concurrency in source code which also includes dependency analysis. Then, we present the three patterns which we choose as prototypes.

5.1 Finding Concurrency

For designing a parallel program, its problems have to be decomposed into tasks. Mattson et al. [2004] define: 'A task is a sequence of instructions that operate together as a group. This group corresponds to some logical part of an algorithm or program.' [Mattson et al. 2004, p.16] Mattson et al. [2004] distinguish between task and data decomposition. The differentiation represents two different points of view which they call *dimensions*. 'The *task-decomposition dimension* views the problem as a stream of instructions that can be broken into sequences called *tasks* that can execute simultaneously.' In constrast, 'The *data-decomposition dimension* focuses on the data required by the tasks and how it can be decomposed into distinct chunks.' [Mattson et al. 2004, p.26] For both decompositions, only a large independence between the separate tasks and data chunks will result in efficient computations. Communication and locking overhead would result in a slowdown otherwise [Mattson et al. 2004, p.26]. For best benefiting of concurrency, the different tasks need to be evenly distributed between the Processing Elements (PEs) like processors [Mattson et al. 2004, p.30].

Mattson et al. [2004, p.30f] present three different locations for identifying parallelisable tasks:

- Tasks might correspond to a method call which is sometimes called *functional decomposition*.
- The single iterations of a loop might be suitable tasks if the iterations are independent and numerous.
- Tasks can also be identified in data-driven decompositions. Updates on large data structures can be split so that distinct tasks update different chunks of the data structure.

Common examples are computations on different segments of an array and the partition of recursive data structures like large trees [Mattson et al. 2004, p.35].

For this thesis, we choose one pattern for each of the mentioned locations as prototypes. They are presented in in Section 5.2.

A dependency analysis examines how data is shared among different tasks which is important for managing the access to shared data correctly [Mattson et al. 2004, p.45]. If data sharing is implemented incorrectly, a task may return wrong results because it accesses invalid data due to race conditions. Tasks without data dependencies can be executed concurrently. As described in Section 4.4.1, shared data between tasks is also unproblematic when it is read-only or effective-local. Hence, one possibility for avoiding race conditions is to divide shared data into separate chunks that can be updated independently. Each chunk is then delegated to a task which executes the update. Thus, these data chunks, associated to tasks, are effective-local. They base upon the data decomposition pattern.

5.2 Pairs of Candidate and Parallelisation Patterns

For detecting patterns which can be parallelised, they have to be identified. In this work, sequential patterns which can become parallelised are entitled *candidate patterns*. Their parallelised version is called *parallelisation pattern*.

The more patterns are identified and their transformation formalised, the more potential concurrency can be found and established in an application. However, we concentrate on three different prototypes for demonstrating the functionality of our approach. We choose patterns which are commonly used in sequential programs so that we could expect a measurable performance improvement. The evaluation which determines the occurrences of the chosen candidate patterns in existing Java applications is documented in Chapter 8.

According to Mattson et al. [2004]'s advice, we find parallelisable tasks in three locations (see Section 5.1). We choose one pattern for each location which are presented in the following subsections.

5.2.1 Independent Successive Method Calls

Mattson et al. [2004, p.30f] mention that parallelisable tasks might correspond to a method call. Clean source code is organised into various methods, whereby some methods may have a long execution time. Therefore, our first prototype aims at parallelising successive, time consuming method calls. This pattern is also referred to as asynchronous method calls [Molitorisz et al. 2012]. As an example, Listing 5.1 shows in lines 4 and 5 two successive methods. The methods establish a connection to different servers. These methods are meant to have a long runtime and to be independent of each other so that the order of the execution does not matter. Instead of waiting for the database connection before establishing the event server connection, both can be executed concurrently.

For this candidate pattern, we desire to handle all sorts of method calls: static and non-static, with access level modifiers from private to public, as well as return-values and various parameters. The feasibility of the parallelisation of the different method types will be exploited in this thesis.

```
1   IDataServerConnection dataSC = new DataServerConnectionImpl();
2   IEventServerConnection eventSC = new EventServerConnectionImpl();
3
4   String hostname = dataSC.connect(100000);
5   eventSC.connect();
6
7   System.out.println("Hello " + hostname);
```

Listing 5.1. Example source code representing a match for the candidate pattern 'Independent Successive Method Calls'

5.2.2 Independent For-Each Loop

As a second prototype, we choose a simple loop pattern without read-write dependencies. It is a for loop which does neither manipulates previous objects nor stores values for later utilisation. Listing 5.2 shows a possible implementation. We assume that the write access to the database does not concern the objects in the Java API. Also we assume, that the connected database enables multi-tasking. For simplifying the pattern, we do not allow interruptions of the loop, thus continue, break or return.

```
    for (ImportantObject o : list) {
        double result = calculateSomethingForQuiteAWhile(o);
        writeResultInDatabase(result);
    }
```

Listing 5.2. Example source code for the candidate pattern 'Independent For-Each Loop'

5.2.3 Reduction of an Array

Our last prototype also contains a loop, however this one represents the reduction of an array [McCool et al. 2012, Sec.5.1]. This loop is not as independent as the example in Listing 5.2 since a value is accumulated in each iteration. As an example, the summation of an array is shown in Listing 5.3. The 'reduction variable' `sum` is initialised in line 2. Line 3 to 5 form the loop which iterates over the array. In the loop body in line 4, the current position of the array is added to `sum`.

```
1  int[] array = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
2  int sum = 0;
3  for (int i = 0; i < array.length; i++) {
4      sum = sum + array[i];
5  }
6  System.out.println("Sum = " + sum);
```

Listing 5.3. Example source code for candidate pattern 'Accumulation of an Array'

Chapter 6

Candidate Pattern Matching

In the phase of pattern matching, candidate patterns are identified in the SDG. For detecting matching patterns in a Neo4J database, the patterns have to be formulated as CMQ (see Section 4.2.1). As an intermediate step, it is recommended to visualise the pattern as an SDG like it is stored in the Neo4J database. It is important that the results of this candidate pattern matching step are correct, otherwise the following automatic transformation will produce an SDG representing a wrong version of the original program. The following section deals with the term correctness in context of pattern matching. In Section 6.2, the representation of dependencies in the SDG is presented. Also, preparative queries are explained which are executed prior to the candidate pattern matching. The concrete formalisation for the three chosen prototype patterns is done in Section 6.3.

6.1 Correct Matching

In computer science, a program is correct if and only if the specifications are complied [Ghezzi et al. 2003]. For our approach, correctness means that the parallel version has the same semantics as the originally sequential program. The candidate pattern matching is the key to our transformation approach. In case, patterns are refused even if they fit the candidate pattern, possible performance improvements are missed. The other way round is even worse: If a pattern is matched even if it is not a suitable candidate pattern, then the transformation will introduce erroneous code. In binary classifications, the terms 'false positives' and 'false negatives' describe these kinds of errors. *False positives* are errors which wrongly report the presence of a condition – in medical testing for example a disease. On the contrary, *false negatives* are errors which indicate the absence of a condition although it is present.

To avoid false positives and negatives, the candidate pattern matching is separated into two parts. The first part matches the structure which has to be present. It has to contain the mandatory combination of statements and control flows. The second part adds restrictions especially concerning data flows for filtering the correct candidate patterns. The more restrictive the first part is formulated, the second part requires less attention. This is shown on the basis of the prototype pattern 'Accumulation of an Array' in Section 6.3.3. In contrast, the prototype pattern 'Independent For-Each Loop' includes a variety of possible source code, so more restrictions or categorisations are necessary (see Section 6.3.2). It is important to identify and implement all possible restrictions, otherwise false positives continue in the transformation process.

6.2 Representation of Dependencies in the SDG

In a graph database, typically dependencies are represented as edges between related nodes. We have already described the structure of the SDG in Section 4.3. The statements inside a method are connected by control flow edges which show the order of the individual statements. For branches, like if-constructs or loops, the control flow edges can contain the property 'case' with possible values 'true' and 'false' i.a.. Additionally, static calls edges connect statements representing method calls to the called method. The relationships of type *call* indicate that the control flow of the program leads to the related method. In case, the method is called from the instance of an object, a 'caller of' relation leads from the instance variable to the method call statement. The caller of relations occur always with a data flow because data flow edges represent data dependencies. Data flows can appear between statements and fields or between two statements. When a data flow edge leads from a statement to a field, it is a *writing access* to the field. When a data flow edge leads in the other direction, from a field to a statement, it is a *reading access* to the field (see Figure 6.1). When a data flow edge connects two statements, it follows the order of the control flow. It is a writing access, if the statement is of type assignment for the concerning variable. If the statement is of the type *method call*, it depends on the called method, whether the concerned variable is accessed only for reading or also for writing.

For simplifying the traversal trough the graph to identify data dependencies, additional relationships are included on method definition level. This means, that all static calls and data flow edges that concern fields or the input parameters of the method are represented by aggregated calls and aggregated field write and read relationships.

6.2.1 Representation of Overridden Methods

It has to be considered that methods which are overridden by other methods do not show any or all dependencies. For instance, the methods of interfaces do not contain any statements, thus the missing of outgoing dependencies could be misunderstood as a read-only method. This would introduce mistakes into the parallelisation process. Also methods of classes which are extended by other classes can be overridden.

Therefore, we introduce a new property for method declarations which we call `'overridden'`. All Cypher functions and expressions are documented in Neo4J's manual.[1] For this thesis, we use the Neo4J version 2.3.1. Listing 6.1 shows the Cypher query which matches all method declarations in the SDG which are overridden by another method.

In line 1 and 2, the pattern is described which we search: a method whose class is extended by a subclass and which also contains a method, or a method which belongs to

[1]The Neo4J Manual: http://neo4j.com/docs/stable/

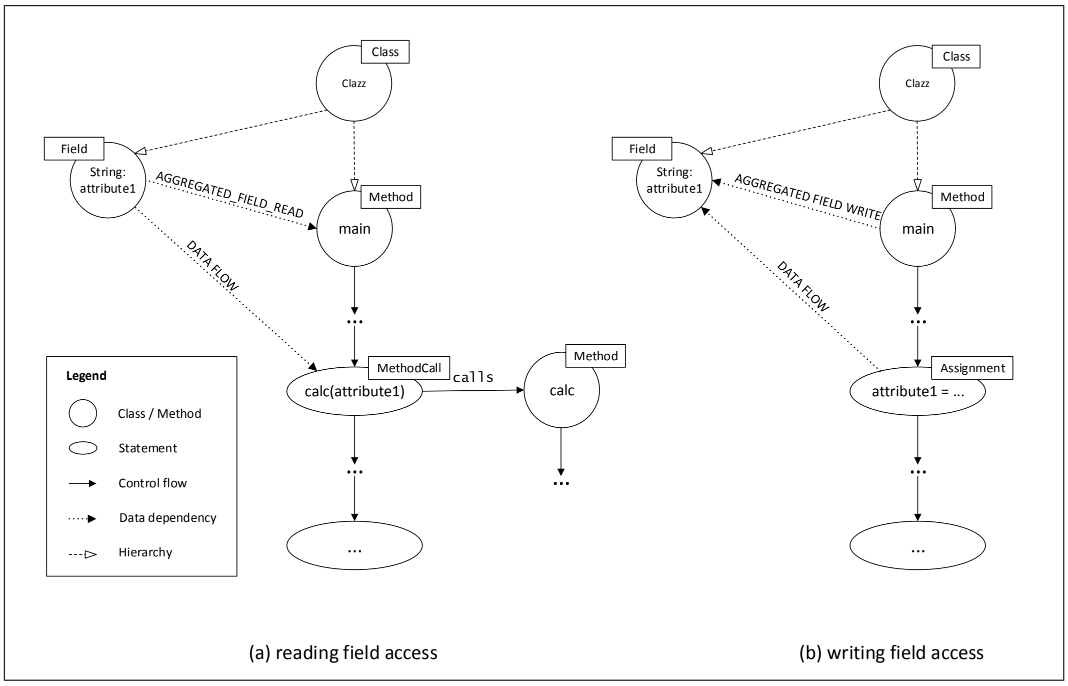


Figure 6.1. Data flows representing write or read access

```
1  MATCH (m:Method) <-[:CONTAINS_METHOD]- (classOrInterface)
2          <-[:EXTENDS|IMPLEMENTS*1..]- (subclass) -[:CONTAINS_METHOD]-> (method:Method)
3  WHERE m.displayname = method.displayname
4  SET m.overridden=true
```

Listing 6.1. Cypher Query which marks overridden method declarations

an interface which is implemented by a class. As we do not name labels for the classes or interfaces, we match both possibilites. With the relationship `<-[:EXTENDS|IMPLEMENTS*1..]-`, we cover the whole hierarchy because the pipe between the both relationships types indicates that both are matched. The asterisk means that the length of this relationship path is indeterminate whereas the `1..` requires at least one edge. Otherwise, we would include that `m = method`. In line 3, we regard the methods and ensure that they have the same name and signature. In our representation of the SDG this can be done by comparing the property `'displayname'`. Finally, the property `'overridden=true'` is set for all matched methods that are overridden. In the following queries, we have to ensure that the called method is either not overridden or that all methods that override the method pass the criteria.

6.2.2 Representation of Read-Only Methods

For optimising and simplifying the CMQ, we also add the property `'isReadOnly=true'` to methods without external writing dependencies. So neither fields, nor the input parameters are modified inside the method. Methods that only need reading access to variables are perfect candidates for parallelisation.
For complete SDGs where also the jdk and libraries are analysed, the Cypher query would be as simple as shown in Listing 6.2:

```
1 MATCH (m:Method)
2 WHERE
3   NOT (exists(m.overridden) OR m.overridden <> true)
4   AND NOT (m) -[:AGGREGATED_FIELD_WRITE|AGGREGATED_CALLS*]-> (:Field)
5 WITH m
6 SET m.isReadOnly=true
```

Listing 6.2. Cypher Query which marks read-only method declarations for completely analysed SDGs

In line 1, we exclusively regard nodes of the type `Method`. Then, we restrict that we only match methods that are not overridden. As we only set `'overridden=true'`, but not the contrary, we have to check the existence of the property `'overridden'` first because non-existing properties are NULL in Cypher and are not equal to anything. Line 4 contains the essential part: `NOT (m) -[:AGGREGATED_FIELD_WRITE|AGGREGATED_CALLS*]-> (:Field)`. With the pipe, several possible relationship types are regarded. The asterisk indicates that the number of following relationships is arbitrary. Hence, this expression excludes all method declarations which are connected with a field by a sequence of aggregated field writes and aggregated calls. This query shows how simple recursive patterns can be described in Cypher.

Nevertheless, as presented in Section 4.3.2, we work on an SDG where the classes and methods of external libraries are not analysed. As a consequence, only the method declarations are represented but without statements or aggregated calls and field accesses. So, we have to exclude the methods from external libraries because they misleadingly appear to be read-only. This is done by regarding the property `'origin'`. If `'origin='APP''` is set, the class or method belongs to the application and is therefore analysed by Soot.

Listing 6.3 and Listing 6.4 show the Cypher queries for identifying read-only methods.

```
MATCH (m:Method)
WHERE
  m.origin = 'APP'
  AND (NOT exists(m.overridden) OR m.overridden <> true)
  AND NOT (m) -[:AGGREGATED_FIELD_WRITE]-> (:Field)
  AND (NOT (m) -[:AGGREGATED_CALLS]-> (:Method))
WITH m
SET m.isReadOnly=true
```

Listing 6.3. Cypher Query which marks direct read-only method declarations

The first query is similar to the previous query, but we only regard the method itself and not recursively all called methods as well. As a consequence, we can only certainly declare methods as read-only if they do not have field write dependencies (line 4) and no outgoing method call (line 5).

In the second query, we populate the property to methods that do not write to fields and that call only methods which are marked as read-only. It has to be executed several times until no new properties are set.

```
1  MATCH (mRO:Method) <-[:AGGREGATED_CALLS]- (m:Method)
2  WHERE
3    mRO.isReadOnly=true
4    AND NOT EXISTS(m.isReadOnly)
5    AND (NOT EXISTS(m.overridden) OR m.overridden <> true)
6    AND NOT (m) -[:AGGREGATED_FIELD_WRITE]-> (:Field)
7    AND (all(path IN ((m) -[:AGGREGATED_CALLS]-> (:Method))
8         WHERE all(method IN nodes(path)
9                WHERE m = method
10                     OR method.isReadOnly=true)))
11   WITH m
12 SET m.isReadOnly=true
```

Listing 6.4. Cypher Query which marks indirect read-only method declarations

In line 1, we match a method `mRO` which is called by another method m. In line 3, we specify that `mRO` contains the property `'isReadOnly=true'`.

In line 4, we check that the regarded method is not yet marked as `'isReadOnly=true'`. As we only inserted the positive case and not `'isReadOnly=false'`, we have to use the function `'EXISTS'` because a missing property returns null which is not comparable. In line 5, we ensure that `m` is not overridden. Line 6 excludes methods which write to a field. Within the lines 7 to 10, it is ensured that all outgoing method calls from method `m` lead to a method which is marked as read-only. The collection function `'all'` returns true if all elements in the collection satisfy the defined constraints. It is formalised as follows: `all(n IN collection WHERE ...)`. In line 7, we define the collection of paths which lead from method `m` to the called methods. Thus, each path contains method `m`, the called method and the `'CALLS'`-relationship between both methods. As constraint, another `'all'` function is defined. With the function `'nodes'`, the nodes of a path are extracted as a collection. In this case, it is always method `m` and the called method. We accept `m` as it is, but the called method has to be read-only.

6.2.3 Representation of Parallelisable Methods

For this thesis, we introduced the property 'isParallelisable' for indicating that a not analysed method of the jdk or other external library is parallelisable. We also can manually mark, for example, the method `Logger.info(..)` as parallelisable because mostly the order does not matter to the result.

For spreading the property `'isParallelisable=true'`, we slightly adjust the previous query, so that we match all method declarations which do not write a field and exclusively call read-only or parallelisable methods. In this case, they are marked as parallelisable as well.

```
MATCH (mRO:Method) <-[:AGGREGATED_CALLS]- (m:Method)
WHERE
  (mRO.isReadOnly=true OR mRO.isParallelisable=true)
  AND NOT EXISTS(m.isReadOnly)
  AND NOT EXISTS(m.isParallelisable)
  AND (NOT EXISTS(m.overridden) OR m.overridden <> true)
  AND NOT (m) -[:AGGREGATED_FIELD_WRITE]-> (:Field)
  AND (all(path IN ((m) -[:AGGREGATED_CALLS]-> (:Method))
      WHERE all(method IN nodes(path)
            WHERE m = method
                  OR method.isReadOnly=true
                  OR method.isParallelisable=true)))
  WITH m
SET m.isParallelisable=true
```

When the parallelisable methods are declared for all non-overridden methods, we can also analyse overridden ones. It has to be ensured that each implementation of the method is read-only or parallelisable, so also the overridden method is certainly parallelisable. Listing 6.5 shows the according Cypher query. As the previous queries, it has to be executed several times because in each iteration, new methods are declared parallelisable. It is also possible to describe the query differently, so that it recursively checks the methods. However, on large graphs our queries perform better.

```
MATCH (m:Method) <-[:CONTAINS_METHOD]-
        () <-[:IMPLEMENTS|EXTENDS*]-
        (:Class) -[:CONTAINS_METHOD]->
        (impl:Method)
WHERE
  m.overridden = true
  AND m.displayname = impl.displayname
  AND NOT EXISTS(m.isReadOnly)
  AND NOT EXISTS(m.isParallelisable)
  AND NOT (m) -[:AGGREGATED_FIELD_WRITE]-> (:Field)
  AND (all(path IN ((m) -[:AGGREGATED_CALLS]-> (:Method))
      WHERE all(method IN nodes(path)
            WHERE m = method
                  OR method.isReadOnly=true
                  OR method.isParallelisable=true)))
WITH m, collect(impl) AS impls
WHERE
 all(i IN impls WHERE i.isParallelisable=true OR
                      i.isReadOnly=true )
SET m.isParallelisable=true
```

Listing 6.5. Cypher Query which marks overridden parallelisable method declarations

In the scope of this thesis, we do not need the differentiation between `'isReadOnly'` and `'isParallelisable'`. Therefore, we set the attribute `'isParallelisable=true'` for all methods which are read-only:

```
MATCH (m:Method) WHERE m.isReadOnly= true WITH m SET m.isParallelisable=true
```

So, in the following we do not need to look at both attributes.

6.3 Formalisation of Candidate Patterns

The formalisation of candidate patterns happens in two stages which is shown in Figure 6.2: Firstly, a matching example of source code is represented as an SDG. Secondly, the graph needs to be translated into a CMQ so that this candidate patterns can be searched and found in the Neo4J database with the SDG of a Java program.

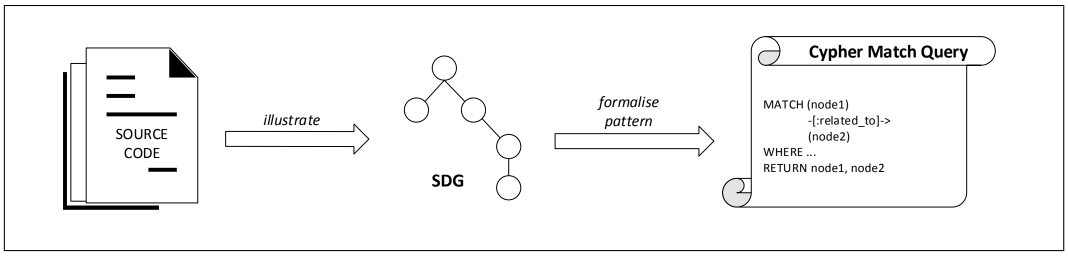


Figure 6.2. Formalisation of candidate patterns

For creating the SDG of the pattern analogously to the representation in Neo4J, the source code has to be in three-address-form. Then, the single statements are represented as single nodes, connected by control flows. Finally, the data flows and method calls are added.

With the help of the SDG of the pattern, the CMQ can be build. The first part of the CMQ typically formalises the pattern on the basis of the control flow. It needs to be abstract enough to match more than the example SDG, but only relevant ones. The second part of the CMQ typically formalises the restrictions, for example absent data flows or that the value of the property `'isParallelisable'` may not be `'false'`.

In the following, the formalisation of the three prototype candidate patterns is demonstrated.

6.3.1 Independent Successive Method Calls

According to the example source code in Listing 5.1, an abstract SDG representing the candidate pattern is created. It is presented in Figure 6.3.

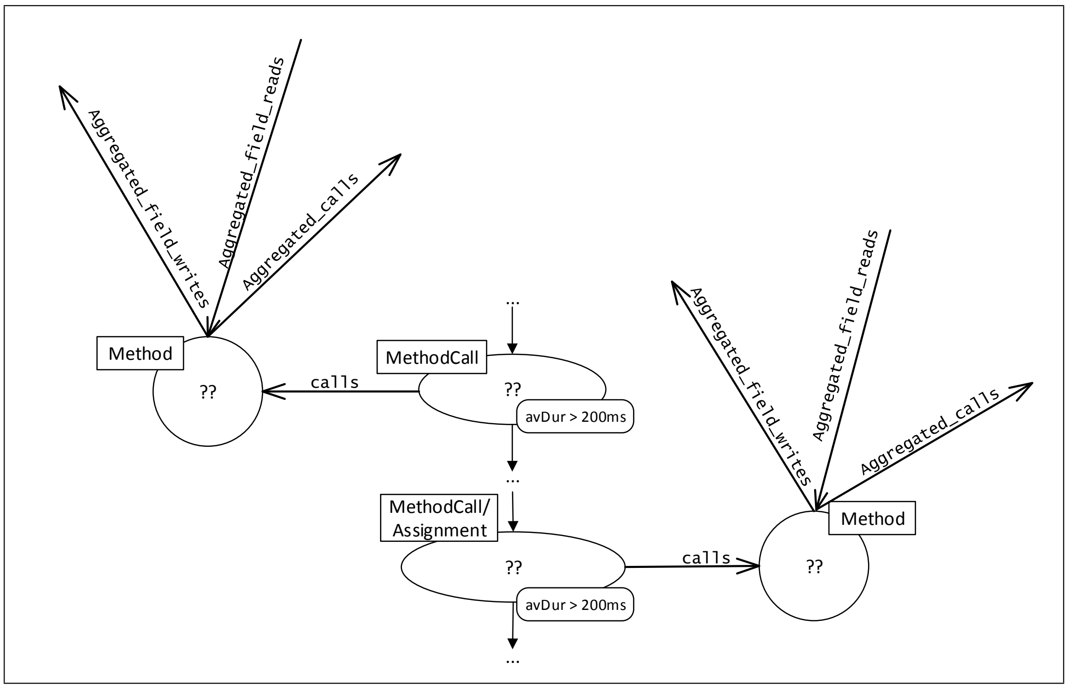


Figure 6.3. SDG of the source code from Listing 5.1

The essential part of this candidate pattern is that two or more method calls succeed each other. In Cypher, this is formulated by the following:

```
MATCH (n1:MethodCall) -[:CONTROL_FLOW*1..5]-> (n2:MethodCall)
RETURN collect(DISTINCT id(n1))
```

An asterisk ($*$) inside a relationship represents a path of that type of relationship. For performance reasons, it is recommended to define a maximum bound for the path length. For various reasons we choose 5 as maximum bound. Since the SDG represents source code in three-address-form, some intermediate statements might be inserted. Also, often methods are called from the instance of an object, so the object has to be instantiated. Therefore, only allowing directly succeeding method calls is very restrictive and would miss parallelisation potentials. However, all statements between the method calls have to be checked such that they do not require the results from the preceding method call. Otherwise, the parallelisation has no effect. The more statements are between the methods, the more analysing effort is necessary and the less chances are that the statements are independent of the method calls. As a consequence, we chose 5 as maximum bound and 1 as a minimum bound because 0 would mean that the nodes `n1` and `n2` are the same. In Cypher, the variable length of a path is described directly after the asterisk.

In line 2, the return statement is shown which we use for each CMQ. The function *id* returns the technical id of the node in the Neo4J database. The function *collect* is used to concat all results in a list. So, the term 'RETURN collect(id(n1))' returns a list of node ids which are always the first node of the candidate patterns. For this pattern, we expect duplicates because `n1` can be followed by another method call after 1, 2, 3, 4 or 5 control flow edges. Therefore, we add the keyword 'DISTINCT' which filters duplicates.

For this prototype, if more than two method calls succeed each other, the pattern matching would match for each combination of two succeeding method calls. However, in this case it might be possible to parallelise them in a way that they all run concurrently, instead of forming pairs of two. We decided to keep the matching of this candidate pattern and check in the transformation step how many succeeding methods can be parallelised (see Section 7.4).

Restriction: Minimum Average Duration

Since source code consists of a lot of method calls, but most of them execute very fast, not all of them are suitable for parallelisation. Creating a thread is time-consuming, so the method call must take a remarkable duration. For this purpose, the SDG must be enriched by runtime information containing the average duration of the method calls. In future work, the enrichment should be achieved automatically with the help of dynamic analysis. At this stage, the runtime information needs to be added manually for the demonstration of our prototypes. These additional information are added with the help of a new property which we call `quotavgDurationInMs`.

So, we extend the CMQ for restricting the minimum average duration of the method call in milliseconds:

```
MATCH (m1:MethodCall) -[:CONTROL_FLOW*1..5]-> (m2:MethodCall)
AND m1.avgDurationInMs > 200 AND m2.avgDurationInMs > 200
RETURN collect(DISTINCT id(m1))
```

The value 200 as a fifth of a second is only a place holder. In our implementation, we construct the query with the help of a constant which can be configured individually. Nevertheless, 200 ms might be a reasonable choice, as the performance can be remarkably improved in many applications. We add the runtime information constraint at first, because it is very restrictive and hence optimises the query. Therefore, we order the restrictions according from high and inexpensive filter potential to more complex patterns.

Restriction: No Branches

For this prototype, we exclude branches like if-then-else or switch case. Therefore, we forbid the property `'case'` of the control flows between the two successive method calls:

```
1  MATCH (m1:MethodCall) -[cfs:CONTROL_FLOW*1..5]-> (m2:MethodCall)
2  WHERE m1.avgDurationInMs > 200 AND m2.avgDurationInMs > 200
3  AND none(cf IN cfs WHERE has(cf.case))
4  RETURN collect(DISTINCT id(m1))
```

We name the control flow relationships so that we can reference them. They are represented as a collection which we can further examine. In line 3, we ensure that none of the control flows of the collection contains the property `'case'`.

Restriction: No Direct Dependency between the Method Calls

Another essential restriction is the independence of the succeeding method calls. The order of their execution must not matter. So, the succeeding method must not use the preceding method call's result. Furthermore, none of the method calls may modify a field which is also used by the other one. For restricting that the succeeding method call is directly dependent on the preceding one, we exclude a data flow between them. The expression in line 4 of Listing 6.6 also excludes transitive dependencies because of the asterisk.

```
1  MATCH (m1:MethodCall) -[cfs:CONTROL_FLOW*1..5]-> (m2:MethodCall)
2  WHERE m1.avgDurationInMs > 200 AND m2.avgDurationInMs > 200
3  AND none(cf IN cfs WHERE exists(cf.case))
4  AND NOT (m1) -[:DATA_FLOW*1..5]-> (m2)
5  RETURN collect(DISTINCT id(m1))
```

Listing 6.6. Cypher query excluding direct dependencies between method calls

Restriction: Called Methods are Parallelisable

As a next step, we only allow method calls to methods which are marked as `'isParallelisable=true'` and we exclude that the method calls assign to the same field if they are assignments. It is presented in Listing 6.7.

```
1  MATCH (d1:Method) <-[:CALLS]-
2  (m1:MethodCall) -[cfs:CONTROL_FLOW*1..5]-> (m2:MethodCall)
3  -[:CALLS]-> (d2:Method)
4  WHERE m1.avgDurationInMs > 200 AND m2.avgDurationInMs > 200
5  AND none(cf IN cfs WHERE exists(cf.case))
6  AND NOT (m1) -[:DATA_FLOW*1..5]-> (m2)
7  AND NOT (m1) -[:DATA_FLOW]-> (:Field) <-[:DATA_FLOW]- (m2)
8  AND d1.isParallelisable=true
9  AND d2.isParallelisable=true
10 RETURN collect(DISTINCT id(m1))
```

Listing 6.7. CMQ for candidate pattern 'Independent Successive Method Calls' without writing access

The `MATCH` part of the query is extended for retrieving information about the called method declarations which we name `d1` and `d2`. In the restricting `WHERE` part, it is checked that the method call statements are no assignments by regarding the labels. In line 8 and 9 is ensured, that both method declarations are marked as `'isParallelisable=true'`.

Restriction: No Dependency Between First Statement and Intermediate Ones

It is not only important that the two method calls are independent of each other. Additionally, the statements between the two method calls may not be dependent on the first method call. Therefore, we check that no data flow leads from the first method call to one of the intermediate statements. This can only occur if the first method call is an assignment. Furthermore, we have to ensure that the first method call and the intermediate methods do not modify concurrently accessed fields. We simplify this constraint by restricting method calls. For the intermediate statements, we only allow method calls to constructors and parallelisable methods.

The following CMQ represents the adjustments:

```
1  MATCH (d1:Method) <-[:CALLS]-
2      (m1:MethodCall) -[cfs:CONTROL_FLOW*1..5]-> (m2:MethodCall)
3      -[:CALLS]-> (d2:Method)
4  WHERE m1.avgDurationInMs > 200 AND m2.avgDurationInMs > 200
5  WITH m1, m2, d1, d2, cfs
6  MATCH path = (m1) -[:CONTROL_FLOW*1..5]-> (m2)
7  WITH m1, m2, d1, d2, cfs, filter (intermediateNode IN nodes(path)
8                                        WHERE intermediateNode <> m1
9                                         AND intermediateNode <> m2)
10                               AS intermediateNodes
11 WHERE
12   NOT (m1) -[:DATA_FLOW*1..5]-> (m2)
13   AND none(cf IN cfs WHERE exists(cf.case))
14   AND NOT (m1) -[:DATA_FLOW]-> (:Field) <-[:DATA_FLOW]- (m2)
15   AND d1.isParallelisable=true
16   AND d2.isParallelisable=true
17   AND all(node IN intermediateNodes
18            WHERE
19                NOT(m1) -[:DATA_FLOW]-> (node)
20                AND (NOT node:MethodCall
21                    OR all(pathcall IN ((node) -[:CALLS]-> ())
22                        WHERE all(call IN rels(pathcall)
23                            WHERE endNode(call).isParallelisable=true
24                                OR endNode(call):Constructor))))
25 RETURN collect(DISTINCT id(m1))
```

In line 6, we name the `path` from one method call to the other `path` for later referencing. In lines 7 to 10, we extract the nodes from the path by means of the function `nodes(path)`. Then, we filter the intermediate nodes between the two method calls by excluding the method calls and name this collection of nodes `intermediateNodes`. In lines 17 to 24, we traverse the `intermediateNodes` and check the mentioned constraints with the help of the collection function `all`: We ensure that the intermediate nodes do not have an incoming data flow from the first method call (line 19). Then, we exclude method calls except they comply to the following nested `all` functions. We define a collection of paths containing the current node, a call relation and the end node of that relation in line 21. As each node can only call one method due to the three-address-form presented in Section 4.3, the collection will have 0 or 1 elements depending on the current intermediate node. In line 22, we extract the relationships from the path which is the single call relation. In line 23 and 24, we check that the called method – hence the end node of the calls relation – is either a constructor or parallelisable.

As the new query is more resource intensive, we decide to put the runtime restriction at the beginning for optimisation.

This CMQ is correct as it only returns parallelisable method calls. However, it is very restrictive because it only allows methods which are marked as parallelisable.

Extension: No Modification of Concurrently Accessed Fields

We can formulate the CMQ less restrictive when we allow more than the method calls to methods which are marked as parallelisable. It is sufficient when the fields which are accessed by both method calls are not modified. All other fields can be changed. For information about the dependencies of a method, the relations `AGGREGATED_CALLS`, `AGGREGATED_FIELD_WRITE` and `AGGREGATED_FIELD_READ` from the method declaration nodes are regarded. However, when we have methods which are not necessarily marked as parallelisable, we have again the problem with the not analysed methods from the jdk and external libraries and with overridden methods. As presented in Section 6.2, not analysed methods might appear as read-only because they do not have outgoing relationships, whereas for overridden methods all overwriting methods have to satisfy the constrains.

In Listing 6.8 we present the changes of the CMQ. Instead of `'AND d1.isParallelisable=true AND d2.isParallelisable=true'`, we check that d1 and d2 are either not parallelisable or that the indefinite chain of relationships `AGGREGATED_CALLS` and `AGGREGATED_FIELD_WRITE` do not modify a field which has a direct connection to the other method call. The direct connection would be a data flow which does not matter if it is reading or writing because the chain already represents a writing relationship. We exclude called methods which are overridden because they need special attendance: for the overriding methods the concurrent access must be detected.

In the following, the handling of overridden methods is explained. The complete CMQ is attached in the appendix. We handle each combination of overridden and not overridden methods separately. Hence, we formulate one query for the case that none of the method calls target overridden methods, one query which represents that the first called method is overridden and the second one is not, one query which represents that the first called method is not overridden but the second one, and one for the case that both method calls are overridden.

For overridden methods, the `MATCH` part of the query has to be extended because we need

```
...
AND NOT d1.overridden=true AND NOT d2.overridden=true
AND (d1.isParallelisable=true
     OR NOT (d1) -[:AGGREGATED_FIELD_WRITE|AGGREGATED_CALLS* ]-> (:Field) -- (d2))
AND (d2.isParallelisable=true
    OR NOT (d2) -[:AGGREGATED_FIELD_WRITE|AGGREGATED_CALLS* ]-> (:Field) -- (d1))
RETURN collect (DISTINCT m1)
```

Listing 6.8. CMQ for candidate pattern 'Independent Successive Method Calls' without overridden methods

to refer to the overwriting methods. The following listing extends the pattern matching. As methods which are not overridden do not match the pattern, we need to seperate the query:

```
  (m:MethodCall) -[:CALLS]-> (d:Method) <-[:CONTAINS_METHOD]- ()
   <-[:IMPLEMENTS|EXTENDS*]- (:Class) -[:CONTAINS_METHOD]-> (impl:Method)
   ...
   WHERE d.displayname = impl.displayname
```

The next listing demonstrates how all implementations of the overridden methods are checked:

```
...WITH m1, d1, collect(impl) AS impls
  all(i IN impls WHERE
        i.isParallelisable=true  OR
        NOT (i) -[:AGGREGATED_FIELD_WRITE|AGGREGATED_CALLS* ]-> (:Field) -- (d1) )
```

Lastly, we present the check when both called methods are overridden:

```
all(i1 IN impls1
   WHERE i1.isParallelisable=true
      OR ( all(i2 IN impls2
           WHERE i2.isParallelisable=true
              OR NOT (i2) -[:AGGREGATED_FIELD_WRITE|AGGREGATED_CALLS* ]
                          -> (:Field) -- (i1) )))
```

So, we were able to solve the challenge with overridden methods. However, the uncertainty because of the not analysed methods from the jdk and other external libraries remain. It is unlikely, that inside the called methods objects are changed in the jdk which are used by the other method as well. We suppose a higher risks due to external libraries. As a consequence, we could further investigate in the transformation step and exhaust the semi-automatism by asking a human developer.

6.3.2 Independent For-Each Loop

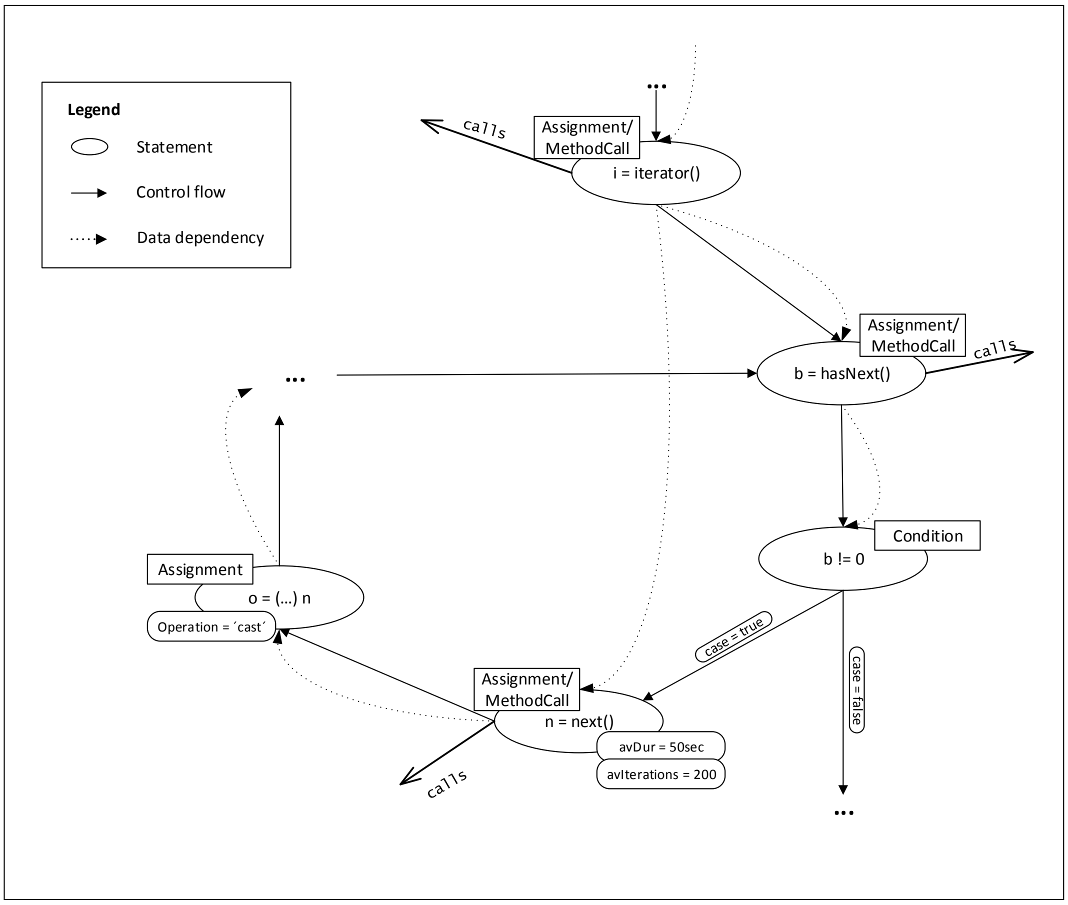


Figure 6.4. SDG of the candidate pattern 'Independent For-Each Loop'

The candidate pattern 'Independent For-Each Loop' aims at for-each loops. As a first filtering, the candidate pattern's control flow is regarded which can be read from Figure 6.4. Listing 6.9 shows the CMQ which matches foreach loops.

In the `MATCH` part of the query in line 1 to 7, we specify a chain of 6 nodes. We specify the nodes as good as possible. The method calls to `java.util.Iterator.hasNext()` and `java.util.Iterator.next()` are identified by the fully qualified name (fqn). For optimising the performance, we create an index in the Neo4J graph database by executing `CREATE INDEX ON :MethodCall(fqn)`. As the fqn identifies the called method declarations, we do not need to include these in the matching query. The method call to the method `java.util.Iterator.iterator()` is described by the return type because the fqn depends

```
1  MATCH
2    (iterator:Assignment:MethodCall {returntype:'java.util.Iterator', displayname:'iterator()'})
3    -[:CONTROL_FLOW]-> (hasNext:Assignment:MethodCall {fqn:'java.util.Iterator.hasNext()'})
4    -[:CONTROL_FLOW]-> (if:Condition)
5    -[:CONTROL_FLOW]-> (next:Assignment:MethodCall {fqn:'java.util.Iterator.next()'})
6    -[:CONTROL_FLOW]-> (firstStatementInLoop:Assignment {operation:'cast'})
7    -[:CONTROL_FLOW*]-> (hasNext),
8    (next) -[df:DATA_FLOW]-> (firstStatementInLoop)
9  WHERE
10   iterator.var = hasNext.caller
11   AND iterator.var = next.caller
12   AND if.name= hasNext.var + ' == 0' OR if.name= hasNext.var + ' != 0'
13 RETURN collect (id(next))
```

Listing 6.9. CMQ for for-each-loop

on the implementation of `Collection`, e.g. `List` or `Map`. The loop body is represented by the control flow relationship with variable length in line 7. Line 8 ensures, that there is a dataflow from the `next` node to the `firstStatementNode`. The `firstStatementNode` always represents the cast of the `Object` returned by the `next` method call. Hence, we do not match loops on pure objects. This has to be done separately in another candidate pattern. In line 10 to 12, further constraints are formalised to ensure that it is a for-each loop. Therefore, we compare the variables of the assignments and the caller of the method calls whether they are the same. For the condition following the `hasNext` node, we allow `var != 0` as well as `var == 0` because the representation in the SDG is dependend on the compiler. We do not have to match the data flows as well because we already compared the variables and callers of dependend nodes. As a result, we return the list of ids of the matching `next` nodes for the following transformation process.

Restriction: Regard Runtime Information

By concentrating on the control flow relationships, the pattern matching includes all possible candidate subgraphs, but also those with unwanted data flow or those which are not worthwile as their execution completes very fast. For excluding the sub graphs which do not need parallelisation, we add restrictions: we expect in average at least 2 loop passes and each loop pass has to last longer than a specified duration – in our case 100 milliseconds, but the runtime information thresholds is configurable. These information are stored in the `next` node of loops. As the CMQ is resource intensive because of the undefined length of the loopbody, we change the order of the matching which is shown in Listing 6.10. First, we filter `next` nodes which indicate the high parallelisation potential. Only the belonging loops are further analysed.

```
MATCH
  (next:Assignment:MethodCall {fqn:'java.util.Iterator.next()'})
WHERE
  next.avgIterations > 2 AND next.avgDurationOfIterationInMs > 400
WITH next
MATCH
  (iterator:Assignment:MethodCall {returntype:'java.util.Iterator', displayname:'iterator()'})
  -[:CONTROL_FLOW]-> (hasNext:Assignment:MethodCall {fqn:'java.util.Iterator.hasNext()'})
  -[:CONTROL_FLOW]-> (if:Condition)
  -[:CONTROL_FLOW]-> (next:Assignment:MethodCall {fqn:'java.util.Iterator.next()'})
  -[:CONTROL_FLOW]-> (firstStatementInLoop:Assignment {operation:'cast'})
  -[:CONTROL_FLOW*]-> (hasNext),
  (next) -[df:DATA_FLOW]-> (firstStatementInLoop)
WHERE
  iterator.var = hasNext.caller
  AND iterator.var = next.caller
  AND if.name= hasNext.var + ' == 0' OR if.name= hasNext.var + ' != 0'
RETURN collect (id(next))
```

Listing 6.10. CMQ for foreach-loop with runtime constraints

Restriction: No Interruption of Loops

In the scope of this thesis, we do not allow interruptions of a loop, hence we exclude `continue`, `break` or `return` statements. We will handle `throw exception` in the transformation step (see Section 7.5). The interruption of loops is mostly an optimisation for avoiding executing unneeded program parts. In a parallelised version, this optimisation would be less useful because the termination of all threads has to be awaited. Also, the transformation is more complex with interrupted loops.

For excluding the `continue` statement, we further examine the data flows which lead to the `hasNext` node. It should have exactly two incoming control flows: one starting the loop and one iterating through the loop. More than two control flows indicate the presence of the `continue` statement and has to be excluded. For excluding the `break` statement, only one control flow should leave the loop body which is the outgoing control flow from the if condition leaving the loop body. As the if condition might be `'b == 0'` oder `'b != 0'`, we cannot follow the true or false case of the control flow. Instead, we follow both control flows and examine the node which does not belong to the loop body. For excluding the `return` statement, we check all nodes of the loop body that do not have the label `ReturnStmt`.

Listing 6.11 shows the adjusted CMQ.

```
MATCH
  (next:Assignment:MethodCall {fqn:'java.util.Iterator.next()'})
WHERE
  next.avgIterations > 2 AND next.avgDurationOfIterationInMs > 400
WITH next
MATCH
  (iterator:Assignment:MethodCall {returntype:'java.util.Iterator', displayname:'iterator()'})
  -[:CONTROL_FLOW]-> (hasNext:Assignment:MethodCall {fqn:'java.util.Iterator.hasNext()'})
  -[:CONTROL_FLOW]-> (if:Condition)
  -[cf1:CONTROL_FLOW]-> (next:Assignment:MethodCall {fqn:'java.util.Iterator.next()'})
  -[:CONTROL_FLOW]-> (firstStatementInLoop:Assignment {operation:'cast'}),
  (next) -[df:DATA_FLOW]-> (firstStatementInLoop),
13   (if) -[cf2:CONTROL_FLOW]-> (firstStatementAfterLoop),
14   loopbody = (firstStatementInLoop) -[:CONTROL_FLOW*]-> (hasNext)
15 WHERE
16   cf1.case <> cf2.case
17   AND iterator.var = hasNext.caller
18   AND iterator.var = next.caller
19   AND if.name= hasNext.var + ' == 0' OR if.name= hasNext.var + ' != 0'
20   AND size((hasNext) <-[:CONTROL_FLOW]- ()) = 2
21   AND size((firstStatementAfterLoop) <-[:CONTROL_FLOW]- () ) = 1
22 WITH next, nodes(loopbody) AS loopbodynodes
23 WHERE
24   all(statement IN loopbodynodes WHERE NOT statement:ReturnStmt)
25 RETURN collect (id(next))
```

Listing 6.11. CMQ for foreach-loop without interruptions

In line 13, we add a `MATCH` statement which grabs the first statement after the loop. In line 16, we ensure that it is the first statement after the loop and not again the `next` node. This is achieved by determining that the case of the control flows from the if statement are different – in this case `true` and `false`. In line 20 and 21, we use the function `size` which returns the size of a collection. In line 20, we estimate the number of incoming control flows to the `hasNext` node. When it is 2, it is accepted, otherwise the loop contains a `continue` statement. Accordingly, we estimate the number of incoming control flows to the first statement after the loop in line 21. We only allow one, as we suspect a `break` statement inside of the loop.

For excluding the `return` statement, we have to examine each node of the loop body. Therefore, we match the loop body separately with the undefinite length of control flows in line 14. We name the path `loopbody` so that we can refer to it in the following. In line 22, we extract the nodes from the path `loopbody` and call that collection of nodes `loopbodynodes`. In the following `WHERE clause`, we traverse each node of the `loopbodynodes` and check that it does not have the label `ReturnStmt`.

Restriction: No External Write Dependencies

For matching only parallelisable loops, the statements in the loop body may not effect the following iterations and no changed variables may be read after the loop. Therefore, assignments may not be made to fields and the data flow of assigned variables may not leave the loop. Otherwise, we would have to ensure, that in each iteration the same value is assigned. Also, we do not allow assignments to variables which come from outside the loop as we would have to ensure that they are not read in advance in the loop.

For method calls, we only allow calls to read-only or parallelisable methods.

Listing 6.12 shows the complete CMQ without dependencies.

In lines 26 to 39, the statements of the loop body are further examined. At first, we allow the `hasNext` node and the `firstStatement` node. They would not pass the constraints because they have data flow connections to nodes outside of the loop body. However, that is totally right because the `firstStatementNode` is dependent on the `next` node, and `hasNext` interacts with the collection for continuing or terminating the loop. For all other

```
 1 MATCH
 2   (next:Assignment:MethodCall {fqn:'java.util.Iterator.next()'})
 3 WHERE
 4   next.avgIterations > 2 AND next.avgDurationOfIterationInMs > 400
 5 WITH next
 6 MATCH
 7    (iterator:Assignment:MethodCall {returntype:'java.util.Iterator', displayname:'iterator()'})
 8   -[:CONTROL_FLOW]-> (hasNext:Assignment:MethodCall {fqn:'java.util.Iterator.hasNext()'})
 9   -[:CONTROL_FLOW]-> (if:Condition)
10   -[cf1:CONTROL_FLOW]-> (next:Assignment:MethodCall {fqn:'java.util.Iterator.next()'})
11   -[:CONTROL_FLOW]-> (firstStatementInLoop:Assignment {operation:'cast'}),
12   (next) -[df:DATA_FLOW]-> (firstStatementInLoop),
13   (if) -[cf2:CONTROL_FLOW]-> (firstStatementAfterLoop),
14   loopbody = (firstStatementInLoop) -[:CONTROL_FLOW*]-> (hasNext)
15 WHERE
16   cf1.case <> cf2.case
17   AND iterator.var = hasNext.caller
18   AND iterator.var = next.caller
19   AND if.name= hasNext.var + ' == 0' OR if.name= hasNext.var + ' != 0'
20   AND size((hasNext) <-[:CONTROL_FLOW]- ()) = 2
21   AND size((firstStatementAfterLoop) <-[:CONTROL_FLOW]- () ) = 1
22   WITH next, nodes(loopbody) AS loopbodynodes
23 WHERE
24  all(statement IN loopbodynodes
25          WHERE NOT statement:ReturnStmt
26          AND (statement = hasNext
27          OR  statement = firstStatementInLoop
28          OR ( NOT statement:Assignment
29                OR all(pathdf IN ((statement) -[:DATA_FLOW]- ())
30                    WHERE all(df IN rels(pathdf)
31                          WHERE (df.var <> statement.var)
32                            OR (startnode(df) IN loopbodynodes
33                            AND endnode(df) in loopbodynodes)))
34             AND
35             ((NOT statement:MethodCall
36               OR all(pathcall IN ((statement) -[:CALLS]-> ())
37                    WHERE all(call IN rels(pathcall)
38                          WHERE endNode(call).isParallelisable=true
39                             OR endNode(call):Constructor))))))
40 RETURN collect (id(next))
```

Listing 6.12. CMQ for foreach-loop without dependencies

nodes of the loop body, we ensure that they are either no assignment or no method call or that they do not have any external impact. In line 29 to 33, assignments are handled. We use a nested `all` function. In line 29, we collect all data flow paths from the currently regarded `statement` node to the neighbouring node. In line 30, we extract the relationships of the path with the function `rels(path)`. We ensure that the data flows do not concern the assignment by comparing the data flow's property `'var'` with the statement's `'var'` in line 31. In case it concerns the assignment, we check in line 32 and 33 that the start node and the end node of the data flow is inside the loop body. So, we only allow assignments without external impact.

For analysing the method calls, we need to regard the called method's declaration node. Therefore, we also use a nested `all` function which is in line 35 to 39. In this case, the collection of paths only contains one path because each method call only calls one method.

```
int sum = 0;
for (int i = 0; i < array.length; i++) {
   sum = sum + array[i];
}
```

Inside the path is exactly one relationship and we regard its end node for ensuring that it is parallelisable or a constructor.

Extension: Supporting Write Access to Iterating Object

The CMQ in Listing 6.12 matches loops without external writing access. However, it is also possible to parallelise loops where only the currently iterated object of the collection is changed. This is mostly the case when the method is called from that object. A possible extension of the previous CMQ is shown in the following. It is added in the expression about method calls, that the `statement`'s caller has to be the variable of the `firstStatementInLoop` where the currently iterated object is assigned.

```
          ((NOT statement:MethodCall
            OR all(pathcall IN ((statement) -[:CALLS]-> ())
                 WHERE all(call IN rels(pathcall)
                         WHERE endNode(call).isParallelisable=true
                           OR statement.caller = firstStatementInLoop.var ))))))
```

Nevertheless, even if it is not typical to manipulate other objects than only the caller, it is possible. Therefore, the called method has to be analysed whether it really only changes the called object. Also, it is more precise to distinguish between different instantiations of an object. However, the SDG represents no instances. As a consequence, also the parameters of the method have to be regarded as well as their fields, and the fields of the fields and so on. At this stage, we recommend to either check the called method in the transformation phase or to exploit the semi-automatic approach and ask the user.

6.3.3 Reduction of an Array

The following listing presents the example source code for the candidate pattern 'Reduction of an Array':

This candidate pattern is the most precise and less flexible one. Therefore, all information can be seen in the SDG in Figure 6.5.

Firstly, we present a CMQ of the candidate pattern on basis of the control flows between the nodes which is shown in Listing 6.13.

The combination of the six nodes is fixed. In contrast to the other chosen candidate patterns, no additional nodes are allowed. We match on the relevant and special properties of the nodes so that we do not match wrong sub graphs. Also, we avoid matching on fixed variable names because they are irrelevant for the pattern and we would exclude fitting patterns. The nodes `length` and `assign` describe operations on an array which lead

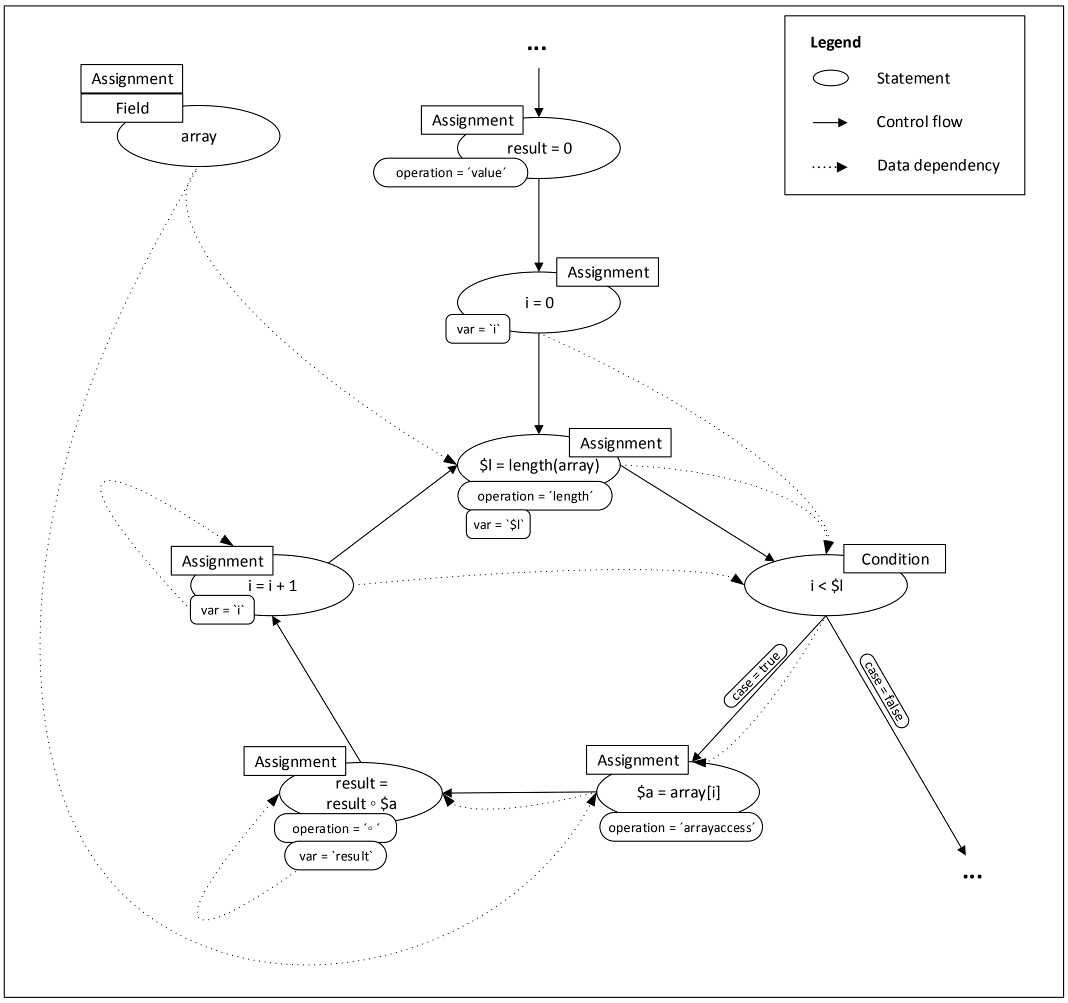


Figure 6.5. SDG of the candidate pattern 'Reduction of an Array'

```
1  MATCH
2    (itstart:Assignment {operation:'value'}) -[:CONTROL_FLOW]->
3    (length:Assignment {operation:'length'}) -[:CONTROL_FLOW]->
4    (if:Condition) -[:CONTROL_FLOW]->
5    (assign:Assignment {operation:'arrayaccess'}) -[:CONTROL_FLOW]->
6    (rd:Assignment) -[:CONTROL_FLOW]->
7    (itplus:Assignment) -[:CONTROL_FLOW]->
8    (length)
9  WHERE
10   itstart.var = itplus.var
11   AND if.name = itstart.var + ' < ' + length.var
12   AND (rd.operation=' + ' OR rd.operation=' - ' OR rd.operation=' * ')
13 RETURN length
```

Listing 6.13. CMQ of the candidate pattern 'Reduction of an Array'

the matching in the right direction. We ensure, that the variables are correct by either comparing them or by regarding the data flows. As operations for the reduction, we allow +, - and *. We could also allow division, but it is an unlikely scenario to create the quotient from the entries of an array. This candidate pattern only matches on numeric patterns. Other reductions like the concatenation of strings are not covered. The constraint for the condition in line 11 is restrictive. It does not contain the actual variable names, but it suggests that the termination expression is formulated as `'iterator < array.length'`. Most of iterations over arrays are restricted in this way. However, it is also possible to adjust the constraint, so that the condition is formulated less restrictive. Another solution is e.g. the use of regular expression like `'if.name= ('.*' + itstart.var + '.*') AND if.name= ('.*' + length.var + '.*')'`. Though, Soot often names new variables with $. As this is a special character in regular expressions, it needs two preceding backslashes. Alternatively, the SDG could be adjusted that the statement of conditions is split into several properties.

In Listing 6.14, the CMQ is complete. We added all data flows and further constraints so that only for loops with the reduction of an array are matched. We also add a node which represents the array. It can either be a field or it can be assigned in a previous assignment. And we add the the node which represents the initial value for the reduction, e.g. `'sum=0'`. Theoretical, also the initial value could be a field or an assignment. However, we assume that it is always directly before the for loop.

```
MATCH
  (initial:Assignment {operation:'value'}) -[:CONTROL_FLOW]->
  (itstart:Assignment {operation:'value'}) -[:CONTROL_FLOW]->
  (length:Assignment {operation:'length'}) -[:CONTROL_FLOW]->
  (if:Condition) -[:CONTROL_FLOW]->
  (assign:Assignment {operation:'arrayaccess'}) -[:CONTROL_FLOW]->
  (rd:Assignment) -[:CONTROL_FLOW]->
  (itplus:Assignment) -[:CONTROL_FLOW]->
  (length),
  (assign) <-[d1:DATA_FLOW]- (array) -[d2:DATA_FLOW]-> (length),
  (itstart) -[:DATA_FLOW]-> (itplus) -[:DATA_FLOW]-> (if),
  (itstart) -[:DATA_FLOW]-> (assign),
  (assign) -[d5:DATA_FLOW]-> (rd) -[:DATA_FLOW]-> (rd),
  (itplus) -[:DATA_FLOW]-> (itplus)
```

```
15 WHERE
16    itstart.var = itplus.var
17   AND if.name = itstart.var + ' < ' + length.var
18   AND (rd.operation=' + ' OR rd.operation=' - ' OR rd.operation=' * ')
19   AND assign.var = d5.var
20   AND array.vartype='int[]' OR array.vartype='long[]' OR array.vartype='double[]'
21   AND (array.var = d1.var = d2.var OR array.name = d1.var = d2.var)
22   AND initial.var = rd.var
23 RETURN length
```

Listing 6.14. CMQ of the candidate pattern 'Reduction of an array pattern'

Data flow relationships contain the name of the variable which is passed. For checking that variables of different nodes are the same, we can either compare the variables or regard the variable of the related data flow. In line 20, we determine that the array is of type `int[]`, `double[]` or `long[]`. In line 21, we compare the variable name of the array with the data flows `d1` and `d2`. The property is `var` if the array is assigned in an assignment node and the property is called name, if it is stored in a field.

Chapter 7

Transformation to Parallelisation Pattern

The target of the graph transformation is a parallelised version of the originally sequential source code representation. Therefore, we start with the development of the target source code. By comparing the sequential and the parallelised implementations of a pattern, we determine the required graph transformations for eliminating the differences. As a supporting step, we plot the SDG of the parallelised pattern. Finally, the transformation is implemented in Java using Cypher queries and the Neo4J API.

In this chapter, we firstly present the Master/Worker Pattern which is a design pattern for parallel programming which we choose for our parallelisation approach. Then, we explain in detail the design decisions for the target source code in Section 7.2. It follows the description the actual graph transformation which is for all three patterns similar. Finally, the specifics of the individual patterns are emphasised.

7.1 Master/Worker Pattern

The Master/Worker Pattern is a concurrent design pattern which balances the work on a set of tasks. The structure of the Master/Worker Pattern is visualised in Figure 7.1. The master process creates several worker processes which handle one or more tasks. These worker processes are collected in a 'bag'. The workers individually execute their tasks while the master awaits their termination. Then, the master process collects the results from the workers in the 'bag' and continues [Mattson et al. 2004, p.143f] [Magee and Kramer 2006, Section 11.2][Ortega-Arjona 2010, p.67ff].

The development of the Master/Worker Pattern from a sequential program is easy, as often blocks from the sequential code for the tasks can be directly reused [Magee and Kramer 2006, Section 11.2].

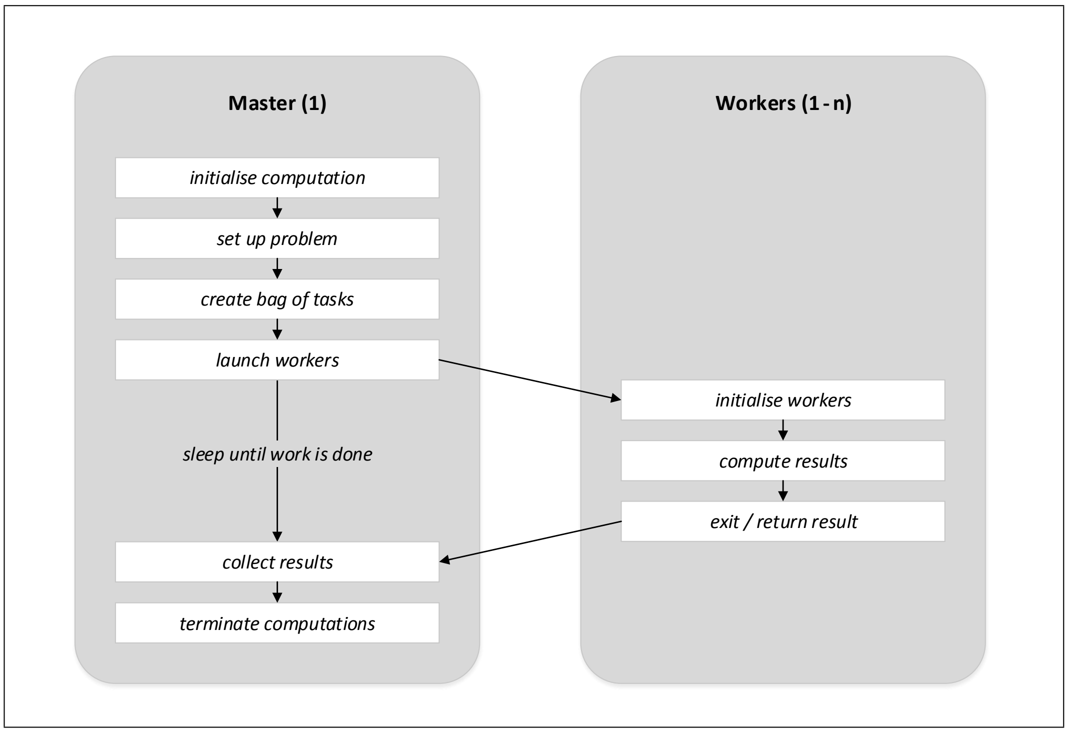


Figure 7.1. Master/Worker Pattern [Mattson et al. 2004, p.145]

Magee and Kramer assert that 'usually, it is best to have one worker process per physical processor' [Magee and Kramer 2006, Section 11.2]. However, they also remind that a significant speed-up is only achieved 'if the tasks require significantly more computation time than the time required for communication with the workers' [Magee and Kramer 2006, Section 11.2]. In case, the computation time of the sequential program is very small, the parallelisation might decrease the performance because creating the workers and retrieving the results takes longer than the sequential execution.

In total, the parallel efficiency of the implementation is dependent on evenly splitted tasks or load balancing and the just mentioned improvement potential. Load balancing means the allocation of threads to the processors in a way, that the work is distributed reasonably equal [Mattson et al. 2004, p.17]. It is especially important when the complexity of the tasks differ extensively. Hence, it is dependent on the candidate patterns but we do not deal in detail with load balancing in this thesis.

7.2 Target Source Code – Design Decisions

For the implementation of the parallelised source code of all candidate patterns, we choose the *Master/Worker Pattern*. As explained in the previous section, the Master/Worker Pattern is a parallel design pattern. The main thread, referred to as master, creates and manages several worker threads. In Java, the Master/Worker Pattern is realised by the interface `ExecutorService` which handles the parallel execution of threads (see Section 7.2.1 for more details) [Mattson et al. 2004, p.148].

Listing 7.1 shows an example source code for the candidate pattern 'Independent For-Each Loop', whereas Listing 7.2 presents the corresponding target source code.

```
public static void doImportantCalculations(List<ImportantObject> list) {
   for (ImportantObject o : list) {
      double result = calculateSomethingForQuiteAWhile(o);
      o.setResult(result);
      writeResultInDatabase(result);
   }
}
```

Listing 7.1. Example source code for candidate pattern 'Independent For-Each Loop'

The structure of the parallelised source code is always the same for the chosen patterns: Firstly, a thread pool – in this case an `ExecutorService` – has to be created. Then, classes which implement the interface `Callable` and undertake the sequential subtasks are added to the pool. The `ExecutorService` starts for each `Callable` a thread. The results of `Callables` are represented as `Futures`. For retrieving the results of the `Callables`, the method `get()` is executed on the `Futures` which returns the result when available. As the `Callables` are executed in threads, `InterruptedExceptions` may occur, also all thrown exceptions inside the thread are converted into `ExecutionExceptions` which have to be handled. Finally, the thread pool is shutdown.

In the following subsections, the design decisions for the implementation are explained in detail. As an example, we use the 'Independent For-Each Loop' because it is compact. However, the design is for all three parallelisation patterns the same.

7.2.1 Java's ExecutorService

The principal part of the parallelised implementation is the `ExecutorService` which manages the creation and scheduling of threads. It belongs to Java's `java.util.concurrent`-package which contains several utility classes for concurrent programming. We use the class `Executors` which provides factory methods for creating the `ExecutorService`. For our purpose, it can be chosen between the static methods `newCachedThreadPool()` and `newFixedThreadPool(int nThreads)`. The method `newCachedThreadPool()` creates an `ExecutorService` that starts for each submitted `Runnable` or `Callable` a new thread. Therefore, finished threads are reused if they are available and removed if they have not been used for 60 seconds. In contrast, the `newFixedThreadPool`-method creates an `ExecutorService` that

```java
import java.util.*

public class ParallelisedClassWithLoop {

   public static void doImportantCalculations(List<ImportantObject> list) {

      int noProcessors = ParallelisationUtil.NUMBER_OF_PROCESSORS;
      ExecutorService pool = Executors.newFixedThreadPool(noProcessors);

      List<Future<Void>> listOfFutures = new ArrayList<Future<Void>>();
      for (ImportantObject o : list) {
         LoopCallable call = new LoopCallable(o);
         java.util.concurrent.Future<Void> future = pool.submit(call);
         listOfFutures.add(future);
      }
      try {
         for (Future<Void> f : listOfFutures) {
            f.get();
         }
      } catch (InterruptedException e) {
         throw new IllegalStateException("Unexpected Interruption");
      } catch (java.util.concurrent.ExecutionException e) {
         Throwable cause = e.getCause();
         if (cause instanceof Error) {
            throw (Error) cause;
         }
         if (cause instanceof RuntimeException) {
            throw (RuntimeException) cause;
         }
      }
      pool.shutdown();
   }

   static class LoopCallable implements Callable<Void> {

      ImportantObject o;

      public LoopCallable(ImportantObject o) {
         this.o = o;
      }

      @Override
      public Void call() throws Exception {
         double result = calculateSomethingForQuiteAWhile(o);
         writeResultInDatabase(result);
         return null;
      }
   }
}
```

Listing 7.2. Parallelised source code example for the candidate pattern 'Independent For-Each Loop'

creates exactly as many threads as given by the parameter `nThreads`. So, the `ExecutorService` will not execute more than `nThreads` threads at the time. Additional tasks are waiting in an unbounded queue until a thread is available.

In our parallelisation approach, we firstly submit all the tasks to the `ExecutorService` and then wait for the completion. Therefore, the cached thread pool would create a thread for each `Callable` and could not reuse them. So we decide to use the fixed thread pool with the parameter `nThreads` as configurable variable which represents the number of processors of the application server [Mattson et al. 2004, p.294f].[1] Hence, as shown in Listing 7.2 in line 7 and 8 , the instantiation of the thread pool is as follows:

```
    int nThreads = parallelisation.config.ParallelisationConfiguration
                .NUMBER_OF_THREADS;
    java.util.concurrent.ExecutorService pool = java.util.concurrent.Executors
                .newFixedThreadPool(nThreads);
```

In the SDG, all objects are defined with fully qualified names. Nevertheless, we would have implemented all new objects fully qualified as it is possible that for instance different `Executors` or `ExecutorServices` are already imported which would cause compilation conflicts. The class `ParallelisationConfiguration` will be added to the source code, it holds *inter alia* the number of processors and configuration settings for the candidate matching of several patterns. It has to be ensured, that there is no other variable called *pool*, otherwise the `ExecutorService` has to be called differently. This issue is part of Section 7.3.1. For each task which shall be managed by the `ExecutorService`, a newly implemented `Callable` object is added using the method `submit(Callable callable)` which returns a `Future` object containing the result of the `Callable`:

```
    Future<?> futureResult = pool.submit(new Callable(...))
```

The `Callable` implementation is handled in the next section.

7.2.2 Callable Implementation

This section presents our decisions regarding the implementation of the threads. The goal is clean code which covers as many parallelisation patterns as possible.

For executing the parallelisable tasks, we add new classes to the `ExecutorService` which implement the interface `Callable`. The benefits from using `Callables` instead of `Runnables` are that `Callables` can return results and that they can throw exceptions. Even when we do not need a return value for each task, nevertheless we always use the `Callables` for exception handling which is described in Section 7.2.3.

The new classes, implementing the interface `Callable` and representing tasks, override the method `call()`. This method is the equivalent of `Runnables' run()` method and therefore is executed, when the belonging thread is started. Thus, all sequential statements which belong to the parallelisable tasks are moved in this method – depending on the pattern

[1]Oracle Java Documentation: https://docs.oracle.com/javase/tutorial/essential/concurrency/pools.html

slightly adjusted. It has to be ensured that no statement in the `call()` method throws `Throwables`, as the signature only allows `Exceptions`. However, we hope this is only a theoretical restriction, because throwing `Throwables` instead of the precise `Exception` is bad practice [Ullenboom 2014, p.567]. Nevertheless, the pattern matching should be adjusted in future work.

As the method `call()` does not have any parameters, the new class needs fields for each parameter which has to be passed to the method. These fields have to be set in an individually created constructor.

With the fields and the constructor, additionally to the `call()` method, in our opinion the class is too large for an anonymous class, since it would decrease the clearness and understandability of the source code. Furthermore, we dismiss creating a new public class for each `Callable`, as they will only be used once in the method which is parallelised. As a result, we examine the use of nested classes. We determine that we will have less complications with import statements when we do not move the statements for executing the tasks in a different file. However, currently the SDG is presented with fully qualified names so it does not matter. Nevertheless, the representation of the SDG could be further developed so we realise this as advantage of nested classes. Additionally, we discover that we can access private methods of nested classes. So, we choose *nested classes* which are distinguished in *static nested classes* and non-static `inner classes`. [2] In static contexts – thus, if the method we parallelise is static – we implement a private, static nested class which is instantiated as follows:

```
NewStaticCallableClass callable = new Outerclass.NewStaticCallableClass(...);
```

Whereas, if the context is non-static, we implement a private static inner class which needs an instance of the outer class for instantiation. However, in non-static contexts, the keyword ***this*** references to the current object, so the new inner class can be instantiated as follows:

```
NewCallableClass callable = this.new NewCallableClass(...);
```

7.2.3 Exception Handling

The transformed program shall return the same results as the original one. Therefore, before and after the transformation the same exceptions have to be thrown. The method `call()` of the `Callable` class allows to throw `Exceptions`. However, as long as the result of the `Callable` is not retrieved, a possibly thrown exception remains hidden from the master thread. Therefore, the method `get()` from the `Future` object holding the `Callables` result has to be invoked which also happens in Listing 7.2 in line 18. If successfully executed, the `get()` method returns the result of the `Callable`. Otherwise, if the thread is interrupted during the execution, an `InterruptedException` is thrown. Further, an `ExecutionException` is thrown if the `call()` method terminates with an exception. As the method declaration

[2] The Java Tutorials: https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html

of `Future.get()` contains `throws InterruptedException, ExecutionException` we have to throw these exceptions in the modified method as well or we have to catch them. We cannot change the signature of the method which we parallelise because it would cause compilation errors if the method is called by another one. Also, we have to ensure that the overall behaviour of the program does not change. Hence, if before the transformation an exception is thrown, the same should be thrown afterwards. For instance, some type of exceptions might be catched and handled.

In the sequential code, there was no `InterruptedException` and we only expect them in extraordinary situations like a hard shutdown of the process or application. Hence, we catch the `InterruptedException` and throw an `IllegalStateException` instead. That way, an exception would terminate the application, however, as it is a `RuntimeException`, the signature of the concerning method is not changed.

For the `ExecutionException`, we have to recover the original exception from the `ExecutionException`. This can be achieved by catching the exception and calling the `Exception.getCause()` method as done in lines 22 and 23 of Listing 7.2. The method `getCause()` returns a `Throwable` object. We want to throw the cause because this was the original behaviour, but throwing the cause would normally lead to compilation errors, except `Throwables` are already catched in the method or thrown in the method declaration. So we collect all checked exceptions which might be thrown in the method `call()`. Additionally, we handle `Errors` and `RuntimeExceptions` which are regarded as unchecked exceptions for Java's compile-time checking [Ullenboom 2014, p.551ff].[3] For each of this `Throwables` we add an if-branch as in lines 24 to 29 of Listing 7.2:

```
if(cause instanceof Exception){
   throw (Exception) cause;
}
```

The order of the if-branches does not matter, nevertheless, this construction might become quite long.

[3]Oracle Java Documentation: https://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html

7.3 Graph Transformation

We aim at transforming the identified candidate pattern into the corresponding parallelisation pattern. Therefore, additional nodes have to be added and some relationships moved. For modifying the SDG in the Neo4J database, we have two possibilities: directly via Cypher or via Neo4J's Java API.

As we formulated the candidate patterns flexible, the transformation needs to cover all variations. For instance, the number and names for the fields of the `Callable` differ. Also, the 'Independent Successive Method Calls' pattern allows method calls with and without returntype. Creating a flexible Cypher query would mean to build a long string with several loops and conditions. The maintenance of such a construction would be extensive and possibly confusing. Also, when the query contains a syntax or semantic error, it cannot be debugged. The error has to be located by looking on the long query and trying to split it into smaller ones.

Another disadvantage of pure Cypher usage is that the queries cannot easily be reused e.g. for other candidate patterns. On the other hand, already short expressions in Cypher can execute powerful request. So, we use Neo4J's Java API to create new nodes and relationships and Cypher for request against the database, e.g. to find a special node. The initial conception to formulate a single CUQ for each pair of candidate and parallelisation pattern as presented in Chapter 3 emerged as not practicable.

For the parallelised SDG, we concentrate on the nodes and on the relationships which are needed to generate source code from the SDG. We do not add the data flows, the aggregated data flows or the 'caller of' relation. Also, the relationship `calls` which leads from a method call statement to the called method is not necessary, because the method calls contain the property `'fqn'` which definitely identifies the called method. This decision simplifies the transformation process and prevents mistakes due to the wrong representation of data dependencies. As a consequence, we cannot transform more than one pattern per method at a time. However, as soon as the generation of source code from the SDG is developed, several iterations of parallelisation can be processed. We set the property `dirty=true` to the method node of the transformed method and check on this prior to transforming a matched pattern.

In the following, we describe typical source code snippets which are needed for the transformation of the SDG from the candidate pattern to the parallelised version. Then, the specifics in the transformation phase are presented for each of the three pattern.

7.3.1 Inserting New Neo4J Nodes

The parallelisation pattern contains more nodes than the candidate pattern. Therefore, new nodes have to be created and inserted at the right location into the SDG. The creation of a Neo4J node can only be done within a transaction. It is done with the method `GraphDatabaseService.createNode()`. Then, the necessary labels are added to the node which is formulated in line 2 and 3 of the following listing. Line 4 shows how a property is added.

```
1       Node node = graphDB.createNode();
2       node.addLabel(SDGLabel.METHODCALL);
3       node.addLabel(SDGLabel.ASSIGNMENT);
4       node.setProperty(SDGPropertyKey.FQN, "...");
5         ...
```

The method `Node.addLabel(Label)` is not applicable for strings. Instead, we use constants which create a `Label` as follows:

```
  public static final Label METHODCALL = DynamicLabel.label("MethodCall");
```

We also use constants for the property keys. So, the labels and keys can be easily adjusted without a lot of maintenance effort.

```
  public static final String FQN = "fqn";
```

Ensuring Fresh Variables

When a new assignment node is created and the variable is new, it has to be ensured that the variable is not used within the scope, yet. Therefore, we execute two Cypher queries like in Listing 7.3 which return all variable names which are already in use for that method. For retrieving all used variable names, we have to collect the field names and the variables of assignments. As the parameters of a method are assigned to their variables at the start of each method, we do not have to handle them separately. In this example, we search for the variable names in the scope of the method which node id is 42 (see line 1). 42 is a placeholder. In our implementation, we build a string with the given node id. Line 2 formulates the relationship between the method node and the field nodes. Line 4 returns the list of field names. The second query traverses the method's statements and collects all assigned variables. As the variables can be assigned multiple times, we use the keyword `DISTINCT` to filter duplicates. The combined result of both queries contains all variable names which should not be used for new variables. In case, the chosen variable name already exists, we attach a number so that the used variable is fresh.

```
1  START m=node(42)
2  MATCH (m) <-[:CONTAINS_METHOD]- () -[:CONTAINS_FIELD]-> (field:Field),
3  RETURN collect(field.name) AS variablenames
4
5  START m=node(42)
6  MATCH (m) -[:CONTROL_FLOW*]-> (assign:Assignment)
7  RETURN collect(DISTINCT assign.var)
```

Listing 7.3. Cypher queries for retrieving variable names in the scope of the current method

Dealing with Relationships

The new nodes have to be inserted at a specific point of the SDG. When we create a totally new class or method which we do for the new callable classes, we can create new nodes, connect them with each other and finally connect the concerning `Package` node with the `Class` node. So, the new class is inserted in the SDG. The creation of relationships with Neo4J's Java API is implemented as follows:

```
  node1.createRelationshipTo(node2, RelTypes.CONTAINS\_TYPE);
```

In this example, a relationship of type `'CONTAINS_TYPE'` is created from `node1` to `node2`. Again, this operation is only possible within a graph database transaction.

In case, we want to insert new statements between existing statements, we need to move the relationships. As we do not have to bother about data flow, caller of and calls edges, we only have to concentrate on the control flow. We decide at which position the new statements shall be inserted and determine the replaced control flow and neighbouring nodes. As visualised in Figure 7.2, the outgoing control flow of the preceding statement node has to lead to the first new statement node. The incoming relationship of the subsequent statement node has to start at the last new statement node. If only one statement is inserted, it is handled as the first and last statement. In Neo4J, it is not possible to move or duplicate relationships. Instead, the old one has to be deleted and a new one created. Listing 7.4 presents the implementation using Neo4J's Java API. In line 2, a new control flow is created from the original start node of the replaced control flow to the first statement node of the newly inserted sub graph. In line 3, the properties of the original control flow are copied to the new one. This is important after conditions. In line 4, the relationship is created from the last statement node of the newly inserted sub graph to the end node of the original control flow. Finally, in line 5, the original relationship is deleted.

```
1        public static void insertSubGraphAtEdge(Node firstnode,
2                                               Node lastnode,
3                                               Relationship replacedCF) {
4        Relationship incoming = replacedCF.
5                                getStartNode().
6                                  createRelationshipTo(firstnode, RelTypes.CONTROL_FLOW);
7        copyProperties(replacedCF, incoming);
8        lastnode.createRelationshipTo(replacedCF.getEndNode(), RelTypes.CONTROL_FLOW);
9        replacedCF.delete();
10   }
```

Listing 7.4. Source code for inserting new statements

7.3.2 The ExecutorService's Instantiation

Analogue to lines 7 and 8 of the Listing 7.2, we create nodes that represent the instantiation of the `ExecutorService` as in Figure 7.3. We proceed as described in the previous section for creating the nodes and ensuring that the variables `nThreads` and `pool` are not used, yet.

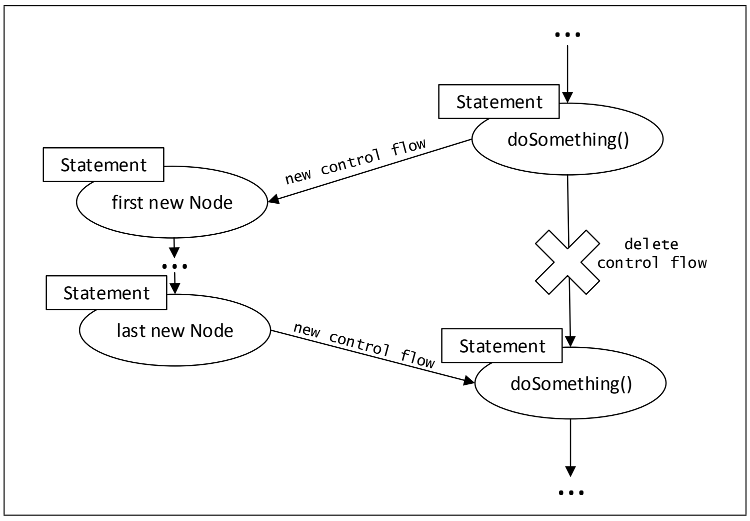


Figure 7.2. Insertion of new statement nodes in the SDG

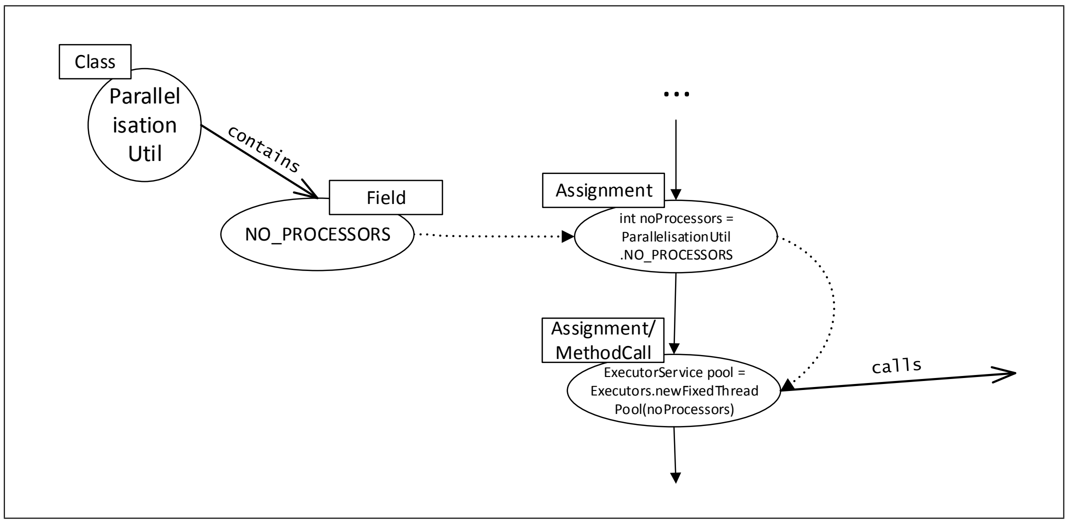

Class
Parallel isation Util
contains
Field
NO_PROCESSORS
...
Assignment
int noProcessors = ParallelisationUtil .NO_PROCESSORS
Assignment/ MethodCall
ExecutorService pool = Executors.newFixedThread Pool(noProcessors)
calls

Figure 7.3. Nodes for the instantiation of the `ExecutorService`

Depending on the candidate pattern, we can either insert the instantiation directly before the pattern, e.g. the loop or we can put it at the beginning of the method directly after the assignments of the parameter. The advantage of the insertion before the pattern is clean code. The insertion at the beginning of a method is no good practice, however we prevent that the variables are out of scope for later referencing, e.g. if instantiated inside an if clause.

For our candidate patterns, we can insert the instantiation of the `ExecutorService` directly before our pattern.

7.3.3 The New Callable Class

Corresponding to lines 34 to 48 in Listing 7.2, one or more `Callable` classes have to be developed which execute the separate tasks. Therefore, a sub graph is created which is visualised in Figure 7.4.

For preventing naming conflicts, we have to ensure that the new classname is not used yet in the package. In the SDG, nested classes are represented with an `''`fqn composed of the outer classname, the dollar sign '$' and the nested class name: `'Outerclassname$InnerClassname'`. We can retrieve the classnames of the concerning package with the following Cypher query:

```
1 START m=node(42)
2 MATCH (m) <-[:CONTAINS_CLASS]- () <-[:CONTAINS_TYPE]- (package) -[:CONTAINS_TYPE]-> (c)
3 RETURN collect(c.name)
```

Again, 42 is the place holder for the id of the modified method. Line 2 describes the relations between the concerning method and the classes in the package. In line 3, we return the list of classnames. The property `'name'` contains the classname. We could also work with the `'fqn'` but as all classes are in the same package, we do not. In the list of classnames, the name 'Outerclassname$InnerClassname' should not exist yet, otherwise we add a number. The created callable class node has to be connected to the package by the `CONTAINS_TYPE` relation. Also, the `IMPLEMENTS` relation to the node representing the `Callable` interface is necessary. Additionally, we create the containing method `call()` and the constructor.

The method `call` does not contain any parameters. Therefore, all needed parameters have to be set as fields. So, for each parameter, we add a `Field` to the `Callable` as well as two `Assignments` to the Constructor. In each method, and hence also constructor, at first each parameter variable is assigned to the parameter. Then, we can assign the field. Additionally, we set the properties of the constructor node corresponding to the parameters, hence `'fqn'`, `'displayname'`, `'parameterscount'` and for each parameter `p0`, `p1` etc.

The statement nodes which shall be executed in parallel are moved to the `call()` method of the `Callable`. The return type of the method is either `Void` if no return value is needed or the '`vartype`' of the assignment node. In all cases, the method has to be completed with a `ReturnStmt` which either returns `null` or the returning variable.

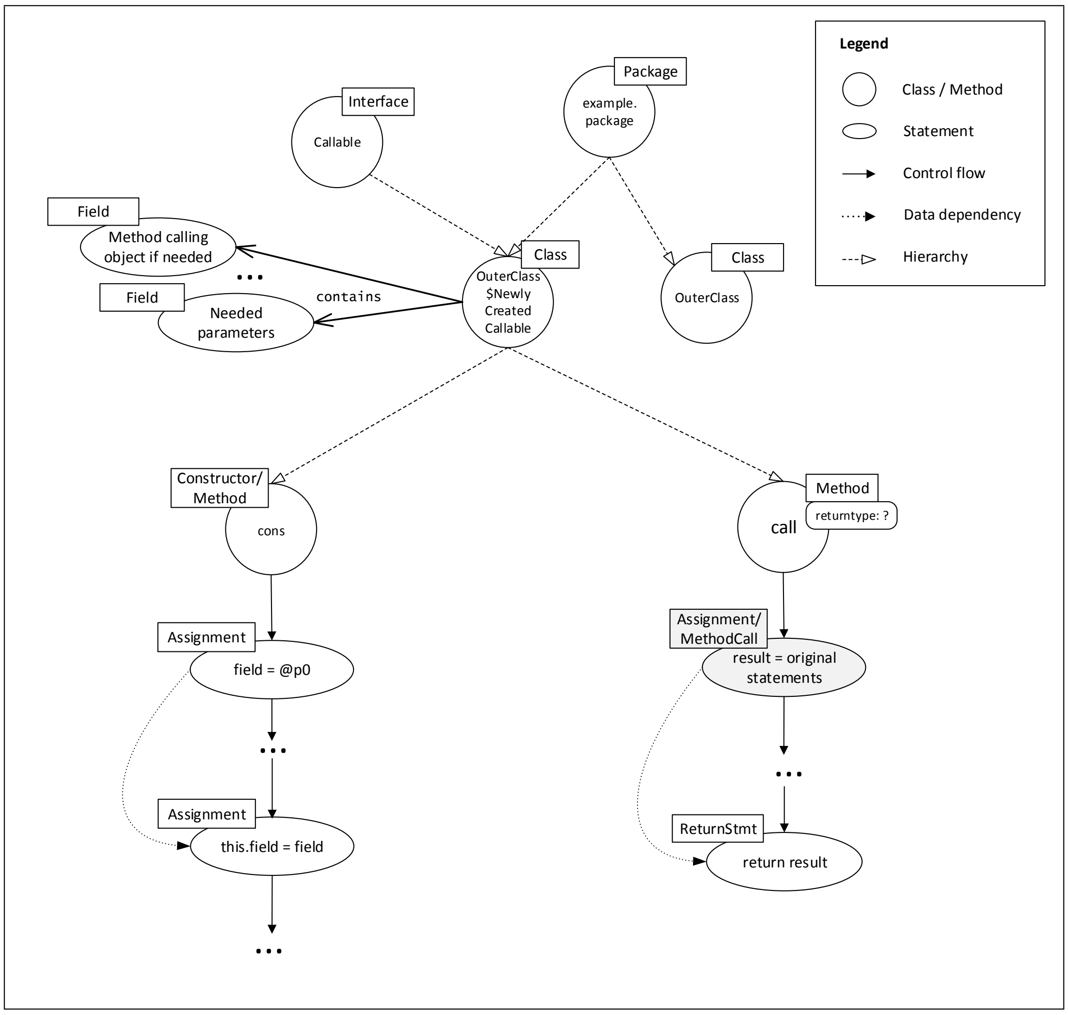


Figure 7.4. SDG of the new Callable class

7.3.4 Joining the Futures and ExecutorService's Termination

In lines 17 to 19 in Listing 7.2, the future results of the `Callables` are retrieved. As the used method `Future.get()` throws exceptions, they have to be catched and handled which is formulated in lines 16 to 30. Finally, the `ExecutorService` is terminated with the method `ExecutorService.shutdown()` in line 31.

An example of the SDG is visualised in Figure 7.5. Within the `try`-block, the method call `Future.get()` has to be executed for each `Callable`. This might be formulated as a loop. In case the return value is needed, the variable has to be assigned prior to the `try`-block. The node for the `ExecutorService.shutdown()` method call directly follows the method calls of `Future.get()`.

In Soot, the expression `catch` is represented as an assignment with a control flow from the method declaration node to the `catch` node. Additionally, we relate the `try` to the `catch` statements. We have to collect all exceptions which might be thrown inside the call method and add nodes for the condition `cause instanceof ...`, the cast to that exception type and the `ThrowStmt`. We collect the exceptions by regarding the property `'exceptions'` of all method calls inside the `call()` method.

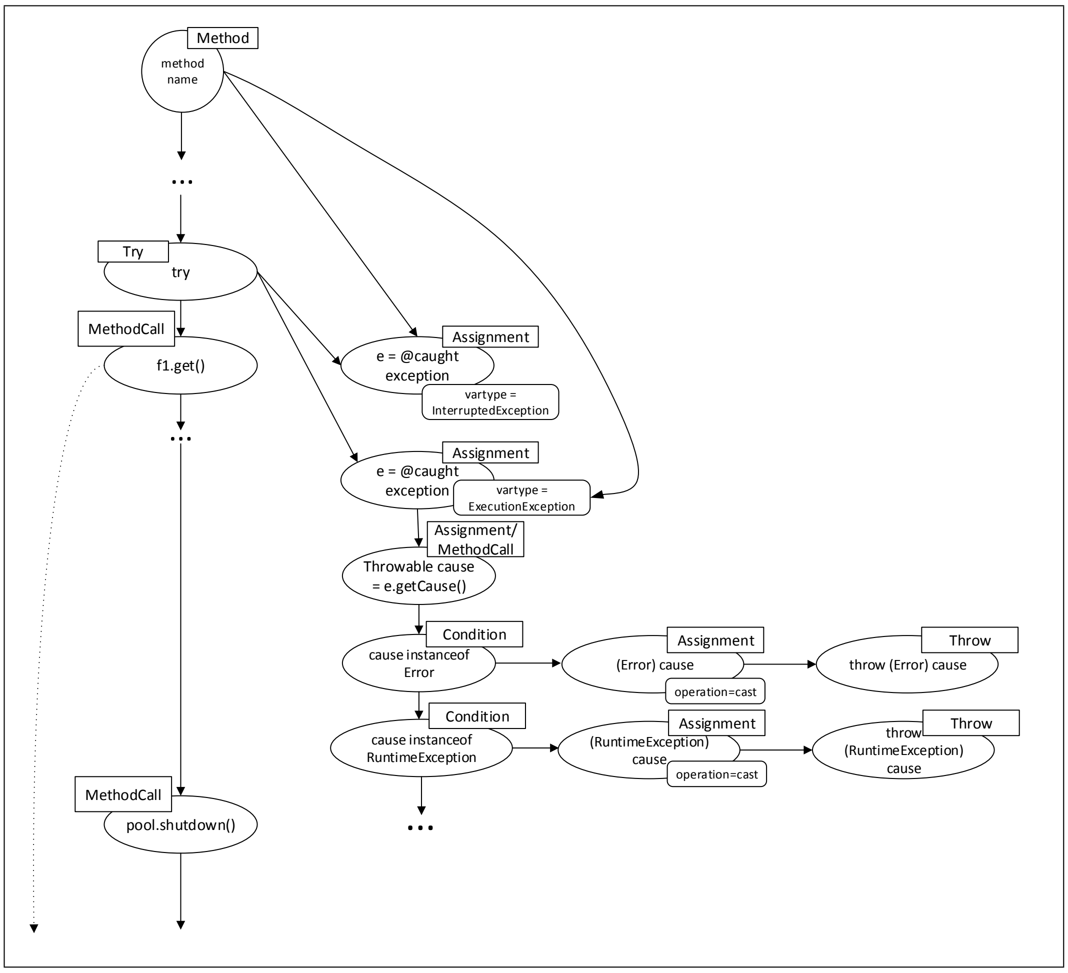


Figure 7.5. SDG of joining the futures and ExecutorService's termination

7.4 Transformation of 'Independent Successive Method Calls'

For transforming the matched candidate patterns for the 'Independent Successive Method Calls', we proceed as described in the previous sections. We create a `Callable` class for each method call which shall be parallelised inside the regarded method. As parameters, we need the parameters for the method call as well as the caller of the method. Inside of the `call` method, only the method call is executed. An example is visualised in the appendix 9.2.

For handling more than two independent successive method calls, we examine the method which shall be transformed in advance and match all pairs of suitable method calls. Then, we check all combinations of method calls by executing the slightly adjusted CMQ presented in **??**. We only add `START m1=node(66), m2=node(77)` previous to the CMQ and replace the return statement by `RETURN true`. Also, we remove the restriction that the method calls may not have more than 5 statements between each other. If the nodes 66 and 77 are method calls which comply the adjusted CMQ, then true is returned, otherwise the result is empty. We start with the first two parallelisable method calls in the method and then extend the check for the following method call so that we identify a list of method calls which we can parallelise in one step.

7.5 Transformation of 'Independent For-Each Loop'

Since Java 8, Java contains so called 'streams' which can be executed in parallel.[4] The following listing presents a possible parallel implementation for the pattern 'Independent For-Each Loop'. It is formulated with the help of a lambda expression (line 1) which is also new in Java 8.

```
1  list.parallelStream().forEach((o) -> {
2          double result = calculateSomethingForQuiteAWhile(o);
3          o.setResult(result);
4      });
```

However, as mentioned in Section 4.3, we do not support Java 8 because the tool Soot which we use for the generation of the SDG does not neither. Therefore, we implement the parallelisation pattern according to the Master/Worker Pattern using the `ExecuterService` and `Callables` as presented previously.

As shown in Listing 7.2, the `Callables` are instantiated and added to the thread pool inside the original loop. The resulting `Futures` are stored in a newly created list for retrieving the future results in the `try-catch` block. Each `Callable` executes one iteration of the loop body. So, the nodes representing the loop body in the sequential program have to be moved to the `call()` method of the `Callable`. As a consequence, all variables accessed in

[4]Oracle Java Documentation: https://docs.oracle.com/javase/tutorial/collections/streams/parallelism.html #executing_streams_in_parallel

```
1   START next = node(42)
2   MATCH
3     (hasNext:Assignment:MethodCall {fqn:'java.util.Iterator.hasNext()'})
4     -[:CONTROL_FLOW*2]-> (next),
5     loopbody = (next) -[:CONTROL_FLOW*]-> (hasNext)
6   WITH next, hasNext, nodes(loopbody) AS loopbodynodes
7   WITH (filter (statement In loopbodynodes WHERE
8            statement <> hasNext
9            AND statement <> next
10           AND any(path IN ((statement) <-[:DATA_FLOW]- ())
11                      WHERE all(df IN rels(path)
12                            WHERE NOT startnode(df) in loopbodynodes))))
13        AS nodesWithIncomingDataflow
14  UNWIND nodesWithIncomingDataflow AS nodeWithDF
15  MATCH
16    (nodeWithDF) <-[:DATA_FLOW]- (assignment)
17  RETURN DISTINCT assignment.var AS var, assignment.vartype AS vartype
```

the loop which are assigned outside of the loop have to be added as fields to the `Callable` class and assigned in its constructor. We collect the required parameters with the following Cypher query:

In line 1, we start the pattern matching with the `next` node. In lines 3 to 5, the loop is represented and the loopbody is named for later references. In line 7 to 12, we filter all statements that have an incoming relationship from outside the loop. We exclude the nodes `next` and `hasNext` because they do not belong to the loop body. In line 14, we transform the collection of statement nodes to individual nodes. For each node we match the assignment node which is the start node of the data flow. As the result, we return a distinct list of the variable names and vartypes of the assignment.

For the exception handling sub graph, we need to collect all exceptions which might be thrown in the loop body. Therefore we traverse the loop body nodes and retrieve the exceptions from the called methods and from `ThrowStmt` nodes. The Cypher query for this request is formulated as follows:

```
START next=node(42)
MATCH
  (hasNext:Assignment:MethodCall {fqn:'java.util.Iterator.hasNext()'})
  -[:CONTROL_FLOW*2]-> (next),
  loopbody = (next) -[:CONTROL_FLOW*]-> (hasNext)
WITH next, hasNext, nodes(loopbody) AS loopbodynodes
WITH (filter (statement In loopbodynodes
        WHERE statement <> hasNext
          AND statement <> next
          AND statement:MethodCall))
      AS methodCalls
UNWIND methodCalls AS m
MATCH
  (m) -[:CALLS]-> (calledMethod)
  UNWIND calledMethod.throws AS exception
RETURN DISTINCT exception
UNION
MATCH
  (hasNext:Assignment:MethodCall {fqn:'java.util.Iterator.hasNext()'})
  -[:CONTROL_FLOW*2]-> (next),
  loopbody = (next) -[:CONTROL_FLOW*]-> (hasNext)
WITH nodes(loopbody) AS loopbodynodes
WITH (filter (statement In loopbodynodes WHERE statement:ThrowStmt)) AS throwStatements
  UNWIND throwStatements AS t
RETURN t.vartype AS exception
```

We separate both requests and combine them with the keyword `UNION`. Technically, they are similar to the previous one.

7.6 Transformation of 'Reduction of an Array'

With the help of the streams of Java 8, also the sum of a collection can be easily calculated in parallel. The corresponding source code would be:

```
      int sum = Arrays.stream(array).parallel().sum();
```

The reduction method `sum` is already provided by Java 8. However, it is possible to describe individual reductions with a lambda expression as presented in the following listing: [5]

```
int sum = Arrays.stream(array).parallel().reduce(0, (a, b) -> a * b);
```

Java 8 provides `IntStreams`, `DoubleStreams` and `LongStreams`[6] which would make the different variations of the candidate pattern possible.

However, as long as Soot does not support Java 8, we parallelise again with the help of Java's `ExecutorService`. The parallelised source code is attached in the appendix 9.2.

After the instantiation of the `ExecutorService`, the `Callables` are instantiated. Each `Callable` shall execute the reduction for a specific range of the array. Therefore, each `Callable` retrieves the fields `array`, `startindex` and `endindex`. The `startindex` and `endindex` are calculated from the number of threads – hence instances of `Callables` and the size of the array, so that the ranges for the `Callables` are evenly distributed. The resulting `Futures` are added to a list for retrieving the individual reduction results `try-catch` block and accumulate them.

We do not need any exception handling as the reduction does not throw exceptions. Therefore, we catch the `InterruptedException` and `ExecutionException` and throw an `IllegalStateException` instead.

[5]Oracle Java Documentation: https://docs.oracle.com/javase/tutorial/collections/streams/reduction.html
[6]Java.util.stream: http://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html

Chapter 8

Evaluation

In this chapter, we evaluate our pattern-based parallelisation approach. In Section 8.1, we examine the feasibility of our approach by regarding the pattern matching and transformation for specified examples. In Section 8.2, the number of occurrences of the candidate patterns in existing Java applications is examined for demonstrating the utility of our approach. This part of the evaluation is structured according to the GQM (Goal-Question-Metrics) principles [Basili and Rombach 1988].

Finally, Section 8.3 evaluates the extendibility of the approach.

8.1 Goal: Demonstrating the Feasibility of the Approach

As presented in the introduction, the main goal of this thesis is the successful transformation of an SDG in its parallelised version. Figure 8.1 shows the overall approach of this thesis, as already presented in Chapter 3.

At first (1), suitable candidate patterns are mined. For each of the chosen candidate patterns, a CMQ is formulated for matching the candidate patterns in an SDG (2). Then, the matched sub graphs are transformed into the corresponding parallelisation pattern (3). The following section evaluates the correctness of the pattern matching and the transformation (4). Therefore, we regard appropriate examples for each candidate pattern and assert that the patterns are matched when expected. For the correctly matched examples, we check that the transformed sub graph correctly represents the parallelised source code.

In future work, the pattern matching could be evaluated in more detail when the generation of Java source code is possible. The parallelised application has to compile and to return the same results than the original one.

8.1.1 Question: Is the result of the approach correct?

For evaluating the correctness of the pattern matching and transformation, we regard for each candidate pattern different examples presented as source code. We choose examples which should match as well as those which should not. For evaluating the result of the transformation, we compared the expected transformed graph with the actual one.

As the visualisation of Neo4J is not clearly arranged, we draw the graph of the second example of the candidate pattern `'Independent Successive Method Call'`. For the other

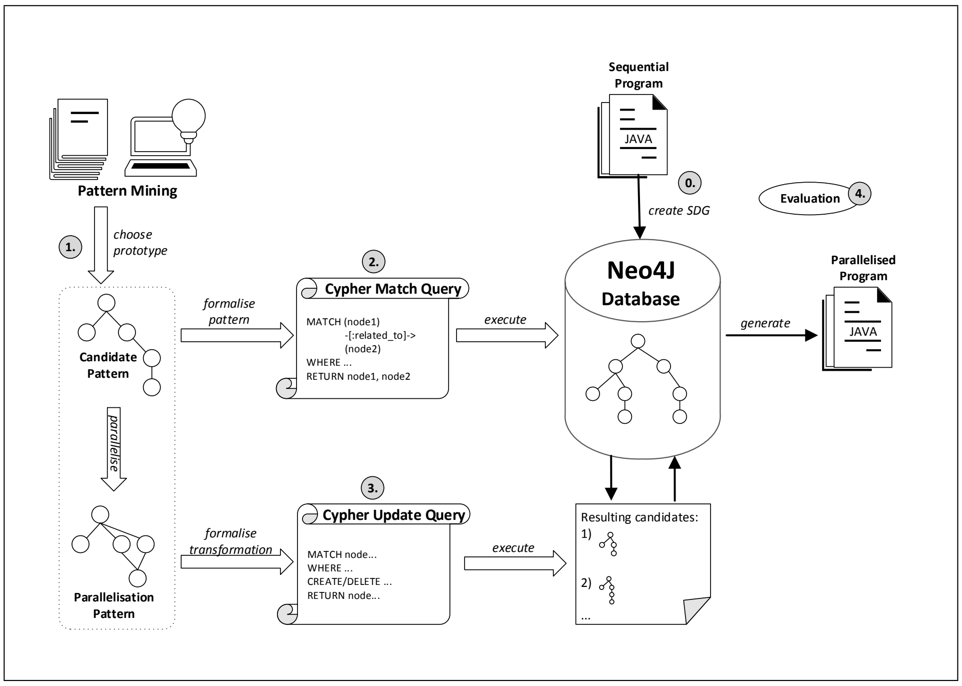


Figure 8.1. Approach of this thesis

examples, we confine ourselves to record our findings in a table. Additionally, the results are stored as Neo4J database and attached to this thesis.

Metrics: Checking Examples for 'Independent Successive Method Calls'

For the following examples, the objects `'dataSC'` and `'eventSC'` are introduced which represent the connection to a data server or event server:

```
        IDataServerConnection dataSC = new DataServerConnectionImpl();
        IEventServerConnection eventSC = new EventServerConnectionImpl();
```

We assume, that these servers are capable of multi-tasking.

Our first example are two independent method calls which directly succeed each other. One needs a parameter, but the other does not. It is presented in Listing 8.1. As the called methods are read-only, the CMQ should match.

```
        dataSC.connect(100000);
        eventSC.connect();
```

Listing 8.1. 1. Example source code for 'Independent Successive Method Calls'

In Listing 8.2, we present the example slightly adjusted because the first called method returns a value. This has to be regarded in the transformation step.

```
    String hostname = dataSC.connect(100000);
    eventSC.connect();
```

Listing 8.2. 2. Example Source Code for 'Independent Successive Method Calls'

Listing 8.3 shows a slightly different example because the called methods are static. Nevertheless, they should be matched by the CMQ.

```
    DataServerConnection.connect(100000);
    EventServerConnection.connect();
```

Listing 8.3. 3. Example source code for 'Independent Successive Method Calls'

In Listing 8.4, we test a sequence of method calls. The method `'Logger.info'` is marked as parallelisable. The other called methods are read-only and marked with long execution times. The method call in line 7 returns a value. The CMQ only checks two successive method calls with long duration at a time and returns the method id of the first one. So, we expect that the pattern matching will match the nodes of line 1, 3, and 5. The transformation should parallelise all four methods together. Hence, only one thread pool is created managing all four callables.

```
1       dataSC.updateProducts(eventSC);
2       LOGGER.info("Products are successfully updated.");
3       dataSC.updateProductPrices(eventSC);
4       LOGGER.info("Product Prices are successfully updated.");
5       dataSC.updateAvailability(eventSC);
6       LOGGER.info("Stock is successfully updated.");
7       int numNewCustomers = dataSC.addNewCustomers(eventSC);
8       LOGGER.info(numNewCustomers
9               + "New customers are successfully inserted in the database.");
```

Listing 8.4. 4. Example source code for 'Independent Successive Method Calls'

In contrast, we expect that the sub graph representing the source code in Listing 8.5 is not matched. The example contains three possible successive method call pairs: statements 1 and 2, statements 1 and 3 and statements 2 and 3. The statements 1 and 2 should not be parallelised because the result of line 1 is needed for line 2. The statements 1 and 3 should not be parallelised because of the same reason. As the statement in line 2 has to wait for the statement in line 1, the parallelisation does not make sense because the statement in line 3 will be started after statement 2. Whereas the statements 2 and 3 should not become parallelised because the method `'Customer.incrementNoPurchases'` changes the field `'noPurchases'` of the object `Customer` which is read in the following method `'Reporting.addRelationsPostCodeNoPurchases'`.

```
1       Customer customer = dataSC.retrieveCustomer(customerid);
2       customer.incrementNoPurchases(dataSC);
3       Reporting.addRelationsPostCodeNoPurchases(dataSC, customer);
```

Listing 8.5. 5. Example source code for not matching 'Independent Successive Method Calls'

Table 8.1. Evaluation 'Independent Successive Method Calls'

# Example for 'Independent Successive Method Calls'	**matching expected**	**correct matching**	**correct transformation**
1. Example: 2 successive method calls	yes	√	√
2. Example: As Example 1, but with assignment	yes	√	√
3. Example: As Example 1, but static method calls	yes	√	√
4. Example: 4 not directly succeeding method calls	yes	√	√
5. Example: Dependent method calls	no	√	–

Table 8.1 presents the results of the pattern matching and transformation. For evaluating the result of the transformation, we compared the expected transformed graph with the actual one. The original and the parallelised sub graph of the second example are shown in the appendix. The other transformed examples can be accessed in the Neo4J database attached to this thesis.

The table shows that the matching of the candidate pattern 'Independent Successive Method Calls' and its transformation to the parallelisation pattern executed as expected for each example. However, as `Callables` cannot handle `Throwables`, we cannot parallelise methods which throw `Throwable`. In Section 8.2, we will therefore determine the number of methods throwing `Throwables`.

Metrics: Checking Examples for 'Independent For-Each Loops'

In the following examples, the methods `'calculateSomethingForQuiteAWhile'` and `'writeResultInDatabase'` represent read-only methods whose execution takes long.

Hence, the example in Listing 8.6 has no external dependencies and should be matched by the CMQ.

```
    for (ImportantObject o : list) {
        double result = calculateSomethingForQuiteAWhile(o);
        writeResultInDatabase(result);
    }
```

Listing 8.6. 1. Example source code for 'Independent For-Each Loop'

Listing 8.7 represents three examples. They are similar to the previous one, but contain either a `continue` or `break` or `return` statement. As we exclude interruptions of the array from our candidate pattern, we expect that it does not match.

The example in Listing 8.8 contains a `set` method. It only modifies the object of the current iteration of the loop, hence the example can become parallelised. However, not always can is known that the method only accesses fields of the current object. Therefore, in the transformation step, the method has to be analysed in more detail or the user has to decide. Nevertheless, we expect the extended CMQ to match, whereas the CMQ which only allows read-only methods should not match.

```
for (ImportantObject o : map) {
   double result = calculateSomethingForQuiteAWhile(o);
   if (result > 100.0) {
      continue/break/return;
   }
   writeResultInDatabase(result);
}
```

Listing 8.7. 2. Example source code for Not Matching 'Independent For-Each Loop'

Table 8.2. Evaluation 'Independent For-Each Loop'

# Example for 'Independent For-Each Loop'	**matching expected**	**correct matching**	**correct transformation**
1. Example: Read-only methods	yes	√	√
2. Example: As Example 1, but with interruptions	no	√	–
3. Example: Modification of current object	yes	√	√
4. Example: Modifying an external variable	no	√	–

```
for (ImportantObject o : list) {
   double result = calculateSomethingForQuiteAWhile(o);
   o.setResult(result);
}
```

Listing 8.8. 3. Example source code for 'Independent For-Each Loop'

In Listing 8.9, a variable is modified in each iteration. It has to be a field or was assigned before the loop, because variables assigned inside the loop are only in the scope of that loop. As a variable is assigned which can be read from outside the loop, it cannot be parallelised because the result is dependent on the order of the iterations.

```
for (ImportantObject o : list) {
   result = result + calculateSomethingForQuiteAWhile(o);
}
```

Listing 8.9. 4. Example source code for not matching 'Independent For-Each Loop'

In Table 8.2, the results of the pattern matching are summarised as well as the results of the transformation to the parallelisation pattern. It shows that the pattern matching and transformation successfully executed as expected.

Metrics: Checking Examples for 'Reduction of an Array'

For the evaluation of the CMQ of the candidate pattern 'Reduction of an Array', we start with the summation of an array. The tested source code is shown in Listing 8.10. As

Table 8.3. Evaluation 'Reduction of an Array'

# Example for 'Reduction of an Array'	matching expected	correct matching	correct transformation
1. Example: Summation of an array	yes	√	√
2. Example: Multiplication of an array	yes	√	√
3. Example: Concatenation of Strings	no	√	–

expected, the CMQ matched the pattern. We obtained the same result when the array was stored in a field or when it was assigned in the method.

```
int sum = 0;
for (int i = 0; i < array.length; i++) {
   sum = sum + array[i];
}
```

Listing 8.10. 1. Example source code for 'Reduction of an Array'

Listing 8.11 shows a variation of the previous source code example. The variable names are different, the elements of the array are multiplied and the reduction variable is of type `long` instead of `int`. Nevertheless, the pattern is matched as expected.

```
long product = 1;
for (int j = 0; j < arrayNumbers.length; j++) {
   product = product * arrayNumbers[j];
}
```

Listing 8.11. 2. Example source code for 'Reduction of an Array'

In Listing 8.11, we present the concatenation of strings. As we do not allow it in the CMQ, we expect no match. As expected, the corresponding sub graph was not matched.

```
String result = "";                    // better StringBuilder..
for (int i = 0; i < array.length; i++) {
   result = result + array[i];
}
```

Listing 8.12. 3. Example source code for not matching 'Reduction of an Array'

Table 8.3 presents the results of the pattern matching and transformation which are as expected.

8.2 Goal: Demonstrating the Utility of the Approach

In this section, we examine whether our approach is useful. Therefore, we explore the existence of the chosen candidate patterns in existing Java applications. Additionally, we examine the extendibility of the approach in Section 8.3.

8.2.1 Question: Do the Candidate Patterns Exist in Real-World Applications?

For evaluating the existence of the chosen candidate patterns in real-world applications, we create the SDGs of two Java applications and execute our evolved CMQs. The applications are Checkstyle[1] and Findbugs[2]. Both applications are static analysis tools for Java applications. Checkstyle inspects the compliance with code standards whereas Findbugs focusses on detecting errors in the source code. The analysing character of these applications makes them good candidates for parallelisation.
In the following section, we determine the number of occurrences of the three prototype candidate patterns in Checkstyle and Findbugs.

Metrics: Occurrences of the Candidate Pattern in two Java Applications

At first, we have to prepare the SDGs representing the tested applications. We add the properties `'overridden'`, `'isReadOnly'`, and `'isParallelisable'` as described in Section 6.2. We do not manually mark any methods as parallelisable. Therefore, we call the read-only methods in the following read-only which is more precise than parallelisable. For performance improvements, we set indexes for the property `'fqn'` on methods and classes as follows:

```
CREATE INDEX ON :Class(fqn)
CREATE INDEX ON :Method(fqn)
```

We remove the constraints concerning the runtime information from the CMQs as we do not have runtime information in the SDG yet.

For a first overview, we determined general information about the SDG which are presented in Table 8.4. It shows, that the SDG representing Checkstyle consists of more than 80 000 nodes, whereas Findbugs is almost twice as big with 140 000 nodes. Only regarding nodes with the property `'origin='APP''`, we count 762 classes with over 6000 methods in Checkstyle and 1160 classes with more than 9000 methods in Findbugs. We also determine the number of overridden methods which is roughly 10% of Checkstyle's methods in contrast to 20% of Findbug's methods.

[1]Checkstyle in version 6.13: http://checkstyle.sourceforge.net/
[2]Findbugs in version 3.0.1: http://findbugs.sourceforge.net/

Table 8.4. Nodes in the SDG of Checkstyle and Findbugs

♯	Checkstyle	Findbugs
Overall nodes in SDG	83619	140875
All classes in SDG	942	1357
All Methods in SDG (without constructors)	8759	11983
Classes in app	762	1160
Methods in app (without constructors)	6772	9392
Overriden methods in app	754	505

Table 8.5. Occurrences of 'Independent Successive Method Calls' in Checkstyle and Findbugs

♯	Checkstyle	Findbugs
Classes in app	762	1160
Methods in app (without constructors)	6772	9392
Read-only methods	963	1953
Method calls	20664	39699
Read-only-method calls	1208	4078
Successive method calls	12870	29595
Successive read-only method calls	26	352
Independent successive method calls	551	–

Metrics: Occurrences of the Candidate Pattern 'Independent Successive Method Calls'

Table 8.5 presents information concerning the candidate pattern 'Independent Successive Method Calls'. We determined the overall number of method calls, as well as the number of read-only methods and method calls to read-only methods. This gives an comparison to the number of occurrences of the candidate pattern. In Checkstyle, we match 26 successive method calls which are read-only. This does not appear a lot in comparison to more than 1000 read-only method calls. With the extension of the pattern which allows method calls from the current iteration's object, we match 551 patterns. However, as the SDG is not complete for the jdk or external libraries, these patterns have to be further examined or semi-automatism has to be exhausted.

In Findbugs, we detect 352 method calls with the read-only constraint which is a promissing result. It is likely, that some of the successive method have a long execution duration which makes a parallelisation profitable. The candidate pattern which is constrained to read-only methods executed within a few seconds. In contrast, we terminated the execution of the extended pattern after about two hours. As we did not had any problems executing the query against Checkstyle, we assume that this behaviour results from the size difference of the graph databases. However, in our opinion, this performance issue will be solved when the runtime information is available because much less successive method calls will have to be analysed.

Table 8.6. Occurrences of 'Independent For-Each Loop' in Checkstyle and Findbugs

♯	Checkstyle	Findbugs
Classes in app	762	1160
Methods in app (without constructors)	6772	9392
For-Each Loops	97	234*
Read-only For-Each Loops	16	75*
Independent For-Each Loops	39	103*

We additionally determined the number of methods which throw `Throwables` as we cannot handle them in the transformation phase. In Findbugs, no method throws `Throwables`. In contrast, Checkstyle contains 15 methods which throw `Throwables`. So, it is recommended to add this restriction to the CMQ.

Metrics: Occurrences of the Candidate Pattern 'Independent Loop Pattern'

In Table 8.6, we list the occurrences of for-each loops. In Checkstyle, we count overall 97 for-each loops. 16 of them are read-only. When we allow method calls from the currently iterated object, we match 39 locations. Hence, more than one third of the loops might be suitable candidates for parallelisation. However, we recommend only to parallelise parts that have a long runtime as the creation and managing of the threads might take longer than the original loop.

We experienced again performance issues when executing the Cypher queries on the SDG of Findbugs. The CMQs as designed in Section 6.3.2 did not terminate in an appropriate amount of time. As a consequence, we limit the size of the loop body to a maximum of 30. With this restriction, the query executed within a few seconds. The results are presented in the table, marked with $*$ because of the restriction of the loop body size. However, restricting the loop size seems to be an appropriate optimisation. Especially, as the chances slim, the bigger a loop body is, that it is independent.

We matched more than twice as many patterns in Findbugs compared to Checkstyle. We are optimistic that the transformation of this candidate pattern will enable high performance improvements.

Metrics: Occurrences of the Candidate Pattern 'Array Reduction Pattern'

As presented in Table 8.7, we do not have a single match for the 'Reduction of an Array' pattern. We additionally determined the number of array assignments, array length operations and array access operations. Each of those exists at least 500 times in the

Table 8.7. Occurrences of 'Reduction of an Array' in Checkstyle and Findbugs

♯	Checkstyle	Findbugs
Classes in app	762	1160
Methods in app (without constructors)	6772	9392
Array length operation	717	636
Array access operation	614	982
Array assignments	2424	536
Array reduction	0	0

applications. Nevertheless, the reduction of an array does not seem to be a typical task for static code analysis tools.

8.3 Extendibility of the Approach

The structure of our transformation implementation is designed to easily add new candidate patterns. Each candidate pattern is implemented in a separate class which extends a super class. For a new candidate pattern, its CMQ has to be formulated. Also, the transformation has to be implemented. For the transformation, lots of utility methods are available. Especially, when the Master/Worker Pattern is chosen for the parallel version using the `ExecutorService`, the creation of the needed nodes can be reused.

Our approach supports Java applications up to Java 7. As presented in Section 4.3, the analysing tool Soot is used for creating the SDG from source code. As long as Soot does not support Java 8, we cannot neither. The pattern matching and transformation is dependent on the specifications of the SDG. When the representation of the SDG changes, the CMQ and the transformation implementation might have to be adjusted. However, the implementation is well structured with configurable constants for the properties, labels and relationship types for simplifying the maintenance.

Chapter 9

Conclusions and Future Work

9.1 Summary

This thesis demonstrates the feasibility of pattern-based parallelisation by transforming three prototypes of candidate patterns to the parallelisation pattern. On the basis of an SDG which is enriched by runtime information, candidate patterns can be matched and automatically or semi-automatically transformed to their parallelised version. In our case, the SDG contains additional information about dependencies in form of aggregated relationships. These information simplify the pattern matching and allow more optimised queries. The framework for the pattern matching is also relevant for the success of the approach. We operate on SDGs which are stored in a Neo4J graph database and provides the query language Cypher.

At the beginning of our work, we expected that we will formulate one CMQ and one CUQ for each candidate pattern. However, we learned that we can formulate the candidate patterns more restrictive or more flexible. The more possibilities exist for a matching pattern, the more attention has to be paid to necessary restrictions. It is important to identify all constraints which have to limit the candidate pattern, otherwise potential faults like race conditions are implemented in the transformation process. Hence, we present CMQs with different degrees of limitations. Also, it is not feasible to formulate a single CUQ as the transformation is very complex. Especially, the flexible candidate patterns also require a flexible transformation. Therefore, we decided to combine Neo4J's Java API and Cypher. So, we can profit from the reusability of Java source code and from the powerful query language Cypher.

We work on SDGs which represent the complete source code of an application, however, the methods of the jdk and referenced libraries are represented as nodes without information about the data dependencies. As a consequence, we deal with a well-sized graph which allows a fast generation of the graph and fast query executions. On the other hand, the CMQs become more complex because methods from the application have to be handled differently than methods from the jdk or other external libraries. Especially, the use of indefinite relations for comparing field accesses is not possible. Hence, a CMQ which does not restrict the candidate pattern matching to read-only methods is dependend on the decision of a human. With a semi-automatic approach, we can exhaust more of the parallelisation potential of an application.

9.2 Future Work

In this thesis, we introduce the property `'isParallelisable=true'` for marking not analysed methods from the jdk or external libraries as parallelisable. Also, the methods for logging are marked as parallelisable although they write to their fields and are therefore not read-only. It could be worth additionally supporting `'isParallelisable=false'` and `'isParallelisable=unknown'` for extending semi-automatism.

For this thesis, we manually add runtime information to the SDG. More research is required to automatically insert runtime information retrieved from a dynamic code analysis.

Another advancement which should be evolved is the generation of Java source code from the SDG. This would also allow a different kind of evaluation. For instance, it is a good start when the generated source code compiles. Also, the original and parallelised applications can be compared with each other. The results of the execution should not change, whereas a performance improvement is desirable and expected.

For simplifying the transformation step, we decided to insert only the relationships which are necessary to generate source code form the transformed SDG. Hence, we add all new nodes, but only control flow relationships as well as the hierarchy for the newly created classes and methods. The call relationship is not necessary because each method call contains the property `'fqn'` which definitely identifies the called methods. Also, the data flows and aggregated field accesses are not necessary for the source code generation. This procedure prevents errors in the representation of the dependencies. As a consequence, we cannot transform more than one pattern per method. However, as soon as the generation of source code works, the transformation can be done in several iterations. As an optimisation, the generation of the SDG could be executed only for the modified methods.

Currently, the approach can handle Java applications up to Java 7. The creation of the SDG is based on Soot, a framework for analysing Java and Android applications. As Soot does not support Java 8, yet, our approach does neither. However, it could be adjusted for supporting Android applications. The graph-based approach in general should be transferable to all object-oriented programming languages which could be further researched. However, the pattern matching and transformation implementation might have to be adjusted when the representation of the SDG changes. Therefore, maintainability effort is expected, e.g. when the generation of Java source code from the SDG is developed.

Furthermore, additional pairs of candidate and parallelisation patterns should be processed, so that as much parallelisation potential is exhausted as useful. For instance, specialised candidate patterns which are fitted to the regarded application could be mined by data decomposition. The parallel implementation could then be based upon the divide-and-conquer concept using Java's `'ForkJoinPool'`[1]. Also, candidate patterns which change

[1] Oracle: http://www.oracle.com/technetwork/articles/java/fork-join-422606.html

the order of statements for achieving a better parallelisation improvement should be investigated.

The results of this thesis indicate, that the pattern-based parallelisation approach is powerful, but complex.

Appendix

Complete CMQ for 'Independent Successive Method Calls'

```
MATCH (d1:Method) <-[:CALLS]-
      (m1:MethodCall) -[cfs:CONTROL_FLOW*1..5]-> (m2:MethodCall)
      -[:CALLS]-> (d2:Method)
WHERE m1.avgDurationInMs > 200 AND m2.avgDurationInMs > 200
WITH m1, m2, d1, d2, cfs
MATCH path = (m1) -[:CONTROL_FLOW*1..5]-> (m2)
WITH m1, m2, d1, d2, cfs,
        filter (intermediateNode IN nodes(path)
                WHERE intermediateNode <> m1
                 AND intermediateNode <> m2)
      AS intermediateNodes
WHERE d1.origin='APP' AND d2.origin='APP'
   AND none(cf IN cfs WHERE has(cf.case))
   AND NOT (m1) -[:DATA_FLOW*1..5]-> (m2)
AND (NOT has(d1.overridden) OR d1.overridden <> true)
AND (NOT has(d2.overridden) OR d2.overridden <> true)
AND all(node IN intermediateNodes
            WHERE
               NOT(m1) -[:DATA_FLOW]-> (node)
               AND (NOT node:MethodCall
                  OR all(pathcall IN ((node) -[:CALLS]-> ())
                      WHERE all(call IN rels(pathcall)
                         WHERE endNode(call).isParallelisable=true
                            OR endNode(call):Constructor))))
AND (d1.isParallelisable=true
     OR NOT (d1) -[:AGGREGATED_FIELD_WRITE|AGGREGATED_CALLS*]-> (:Field) -- (d2))
AND (d2.isParallelisable=true
    OR NOT (d2) -[:AGGREGATED_FIELD_WRITE|AGGREGATED_CALLS*]-> (:Field) -- (d1))
RETURN collect (DISTINCT id(m1))

UNION

MATCH (d1:Method) <-[:CALLS]-
       (m1:MethodCall) -[cfs:CONTROL_FLOW*1..5]-> (m2:MethodCall)
      -[:CALLS]-> (d2:Method)
        <-[:CONTAINS_METHOD]- () <-[:IMPLEMENTS|EXTENDS*]-
          (:Class) -[:CONTAINS_METHOD]-> (impl2:Method)
WHERE m1.avgDurationInMs > 200 AND m2.avgDurationInMs > 200
WITH m1, m2, d1, d2, cfs, impl2
MATCH path = (m1) -[:CONTROL_FLOW*1..5]-> (m2)
WITH m1, m2, d1, d2, cfs, impl2,
        filter (intermediateNode IN nodes(path)
                WHERE intermediateNode <> m1
                 AND intermediateNode <> m2)
      AS intermediateNodes
WHERE d1.origin='APP' AND d2.origin='APP'
AND (NOT has(d1.overridden) OR d1.overridden <> true)
   AND none(cf IN cfs WHERE has(cf.case))
   AND NOT (m1) -[:DATA_FLOW*1..5]-> (m2)
    AND d2.displayname = impl2.displayname
    AND all(node IN intermediateNodes
            WHERE
                NOT(m1) -[:DATA_FLOW]-> (node)
                AND (NOT node:MethodCall
                    OR all(pathcall IN ((node) -[:CALLS]-> ())
                        WHERE all(call IN rels(pathcall)
                            WHERE endNode(call).isParallelisable=true
                                OR endNode(call):Constructor))))
WITH m1, d1, collect(impl2) AS impl
WHERE d1.isParallelisable=true
      OR none(i IN impl
          WHERE (d1) -[:AGGREGATED_FIELD_WRITE|AGGREGATED_CALLS*]-> (:Field) -- (i))
      AND all(i IN impl WHERE
              i.isParallelisable=true
              OR NOT (i) -[:AGGREGATED_FIELD_WRITE|AGGREGATED_CALLS*]-> (:Field) -- (d1))
RETURN collect (DISTINCT id(m1))

UNION

MATCH (impl1:Method) <-[:CONTAINS_METHOD]-
       (:Class) -[:IMPLEMENTS|EXTENDS*]-> () -[:CONTAINS_METHOD]->
       (d1:Method) <-[:CALLS]-
      (m1:MethodCall) -[cfs:CONTROL_FLOW*1..5]-> (m2:MethodCall)
      -[:CALLS]-> (d2:Method)
WHERE m1.avgDurationInMs > 200 AND m2.avgDurationInMs > 200
WITH m1, m2, d1, d2, cfs, impl1
MATCH path = (m1) -[:CONTROL_FLOW*1..5]-> (m2)
WITH m1, m2, d1, d2, cfs, impl1, filter (intermediateNode IN nodes(path)
                                      WHERE intermediateNode <> m1
                                       AND intermediateNode <> m2)
                                  AS intermediateNodes
WHERE d1.origin='APP' AND d2.origin='APP'
AND (NOT has(d2.overridden) OR d2.overridden <> true)
   AND none(cf IN cfs WHERE has(cf.case))
   AND NOT (m1) -[:DATA_FLOW*1..5]-> (m2)
    AND d1.displayname = impl1.displayname
    AND all(node IN intermediateNodes
            WHERE
                NOT(m1) -[:DATA_FLOW]-> (node)
                AND (NOT node:MethodCall
                    OR all(pathcall IN ((node) -[:CALLS]-> ())
                        WHERE all(call IN rels(pathcall)
                            WHERE endNode(call).isParallelisable=true
                                OR endNode(call):Constructor))))
WITH m1, d2, collect(impl1) AS impl
WHERE
       all(i IN impl WHERE
             i.isParallelisable=true
             OR NOT (i) -[:AGGREGATED_FIELD_WRITE|AGGREGATED_CALLS*]-> (:Field) -- (d2) )
      AND d2.isParallelisable=true
      OR none(i IN impl
          WHERE (d2) -[:AGGREGATED_FIELD_WRITE|AGGREGATED_CALLS*]-> (:Field) -- (i) )
RETURN collect (DISTINCT id(m1))

UNION

MATCH      (impl1:Method) <-[:CONTAINS_METHOD]-
           (:Class) -[:IMPLEMENTS|EXTENDS*]-> () -[:CONTAINS_METHOD]->
          (d1:Method) <-[:CALLS]-
          (m1:MethodCall) -[cfs:CONTROL_FLOW*1..5]-> (m2:MethodCall)
         -[:CALLS]-> (d2:Method)  <-[:CONTAINS_METHOD]-
           () <-[:IMPLEMENTS|EXTENDS*]- (:Class)
          -[:CONTAINS_METHOD]-> (impl2:Method)
WHERE m1.avgDurationInMs > 200 AND m2.avgDurationInMs > 200
WITH m1, m2, d1, d2, cfs, impl1, impl2
MATCH path = (m1) -[:CONTROL_FLOW*1..5]-> (m2)
WITH m1, m2, d1, d2, cfs, impl1, impl2, filter (intermediateNode IN nodes(path)
                                    WHERE intermediateNode <> m1
                                     AND intermediateNode <> m2)
                         AS intermediateNodes
WHERE d1.origin='APP' AND d2.origin='APP'
     AND none(cf IN cfs WHERE has(cf.case))
     AND NOT (m1) -[:DATA_FLOW*1..5]-> (m2)
      AND d1.displayname = impl1.displayname
      AND d2.displayname = impl2.displayname
      AND all(node IN intermediateNodes
           WHERE
              NOT(m1) -[:DATA_FLOW]-> (node)
              AND (NOT node:MethodCall
                 OR all(pathcall IN ((node) -[:CALLS]-> ())
                     WHERE all(call IN rels(pathcall)
                        WHERE endNode(call).isParallelisable=true
                           OR endNode(call):Constructor))))
WITH m1, collect(impl2) AS impl2, collect(impl1) AS impl1
WHERE
 all(i1 IN impl1 WHERE i1.isParallelisable=true OR (
     all(i2 IN impl2 WHERE i2.isParallelisable=true
         OR NOT (i2) -[:AGGREGATED_FIELD_WRITE|AGGREGATED_CALLS* ]-> (:Field) -- (i1) )))
RETURN collect (DISTINCT id(m1))
```

Listing 9.1. Complete CMQ for 'Independent Successive Method Calls'

Example SDG of 'Independent Successive Method Calls' before and after the transformation

In the following, we present the transformation of the SDG for the source code example in Listing 9.2. Figure 9.1 shows the SDG of the candidate pattern. The SDG of the corresponding parallelisation pattern is split into three parts. Figure 9.2 shows the modified method which instantiates the new `Callables`. The SDGs of the `Callables` are presented in Figure 9.3 and Figure 9.4. They especially differ because the first method call needs a parameter and returns a value whereas the second one does not. The nodes representing the method calls are represented with a grey background so that their displacement is visualised.

```
      String hostname = dataSC.connect(100000);
      eventSC.connect();
```

Listing 9.2. 2. Example Source Code for 'Independent Successive Method Calls'

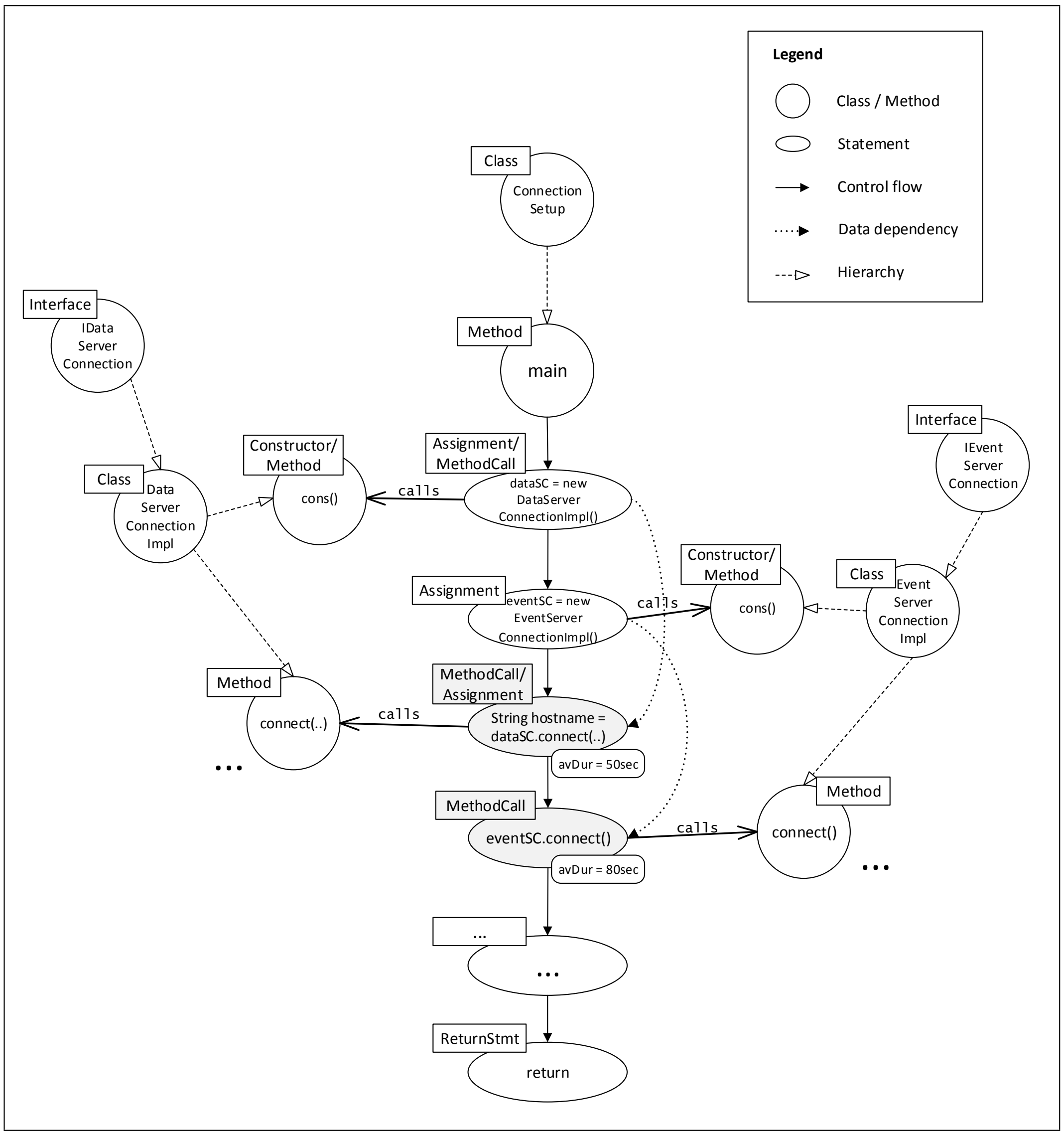


Figure 9.1. Example SDG of the candidate pattern 'Independent Successive Method Calls'

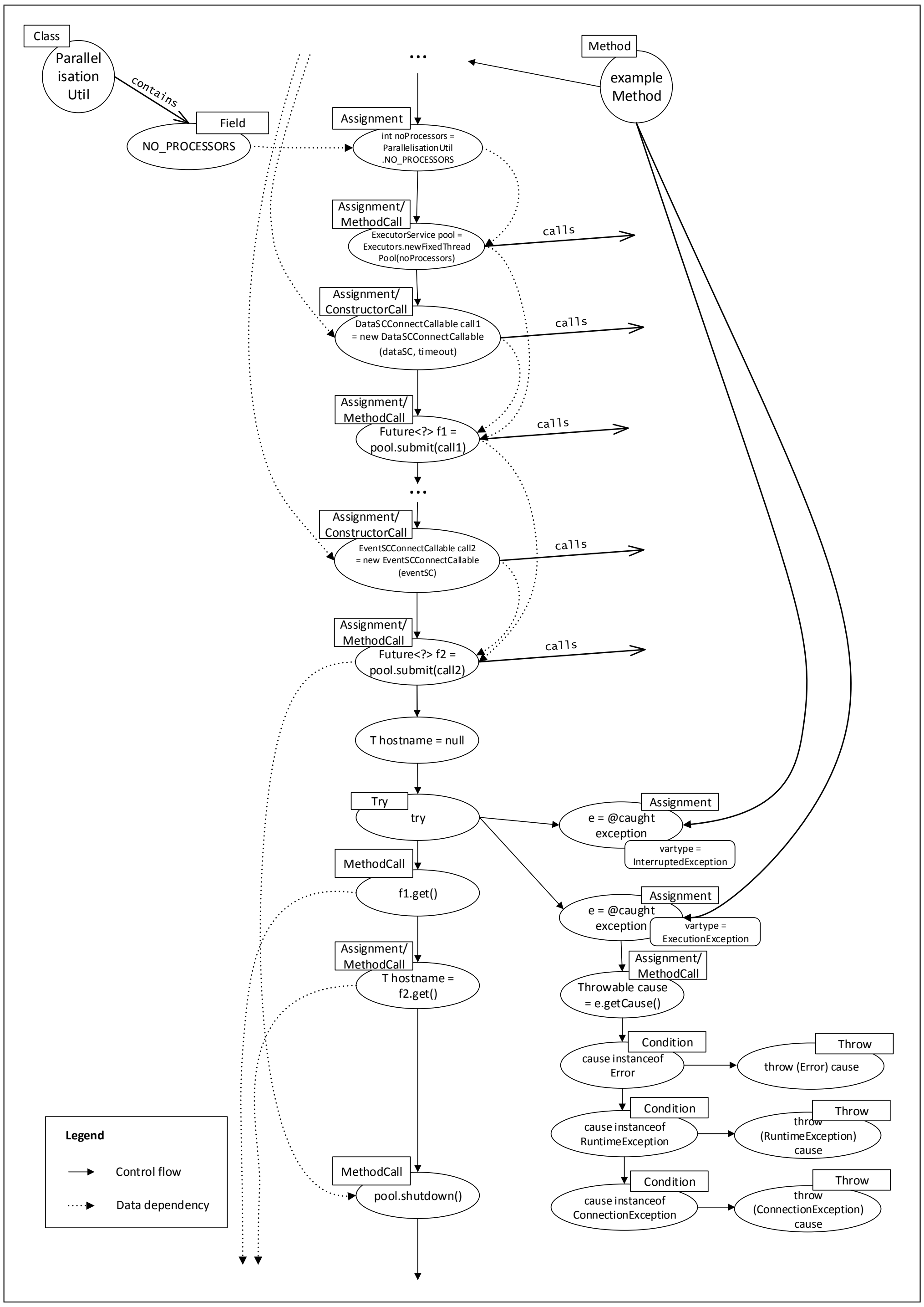

Class
Parallel isation Util
contains
Field
NO_PROCESSORS
...
Method
example Method
Assignment
int noProcessors = ParallelisationUtil .NO_PROCESSORS
Assignment/ MethodCall
ExecutorService pool = Executors.newFixedThread Pool(noProcessors)
calls
Assignment/ ConstructorCall
DataSCConnectCallable call1 = new DataSCConnectCallable (dataSC, timeout)
calls
Assignment/ MethodCall
Future<?> f1 = pool.submit(call1)
calls
...
Assignment/ ConstructorCall
EventSCConnectCallable call2 = new EventSCConnectCallable (eventSC)
calls
Assignment/ MethodCall
Future<?> f2 = pool.submit(call2)
calls
T hostname = null
Try
try
Assignment
e = @caught exception
vartype = InterruptedException
MethodCall
f1.get()
Assignment
e = @caught exception
vartype = ExecutionException
Assignment/ MethodCall
Throwable cause = e.getCause()
Assignment/ MethodCall
T hostname = f2.get()
Condition
cause instanceof Error
Throw
throw (Error) cause
Condition
cause instanceof RuntimeException
Throw
throw (RuntimeException) cause
Condition
cause instanceof ConnectionException
Throw
throw (ConnectionException) cause
MethodCall
pool.shutdown()
Legend
Control flow
Data dependency

Figure 9.2. Example SDG of the parallelisation pattern 'Independent Successive Method Calls'

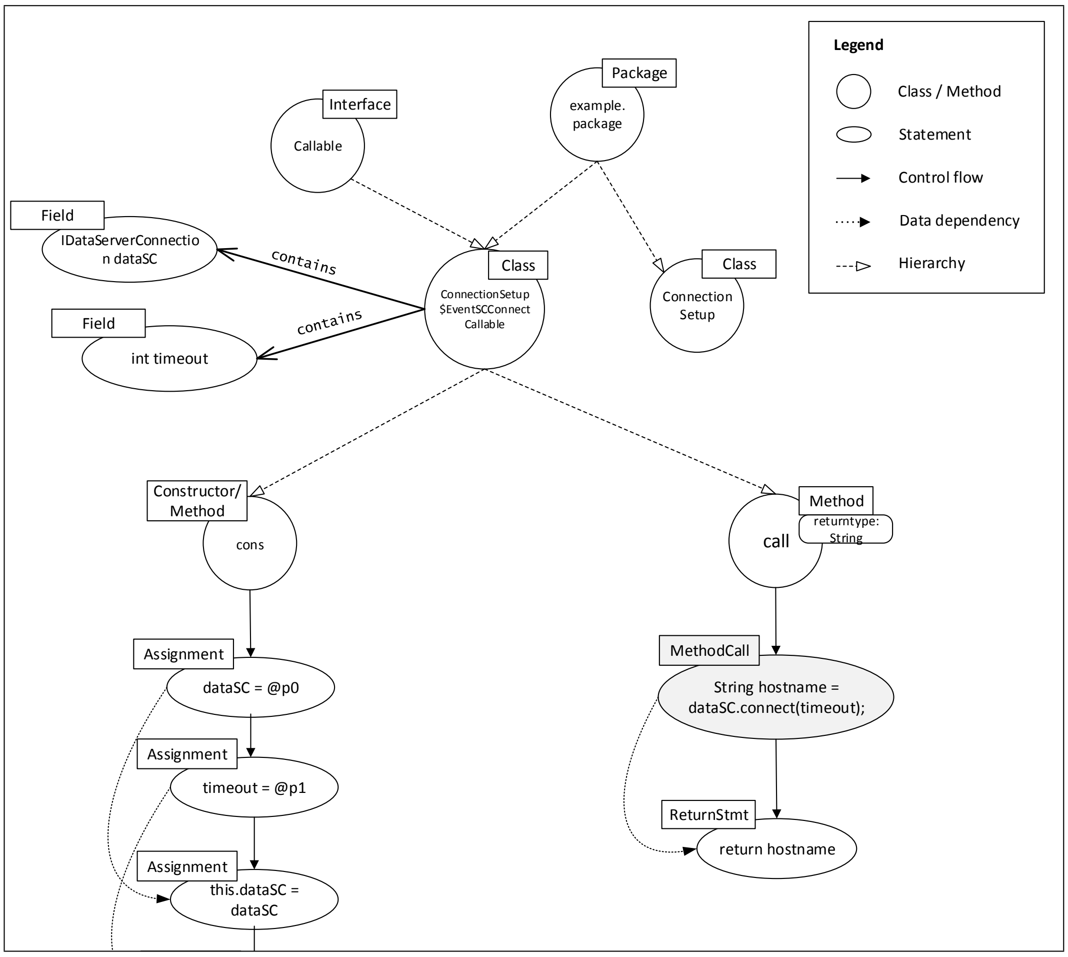


Figure 9.3. Continuation of the example parallelisation pattern: SDG of the Callable with Assignment

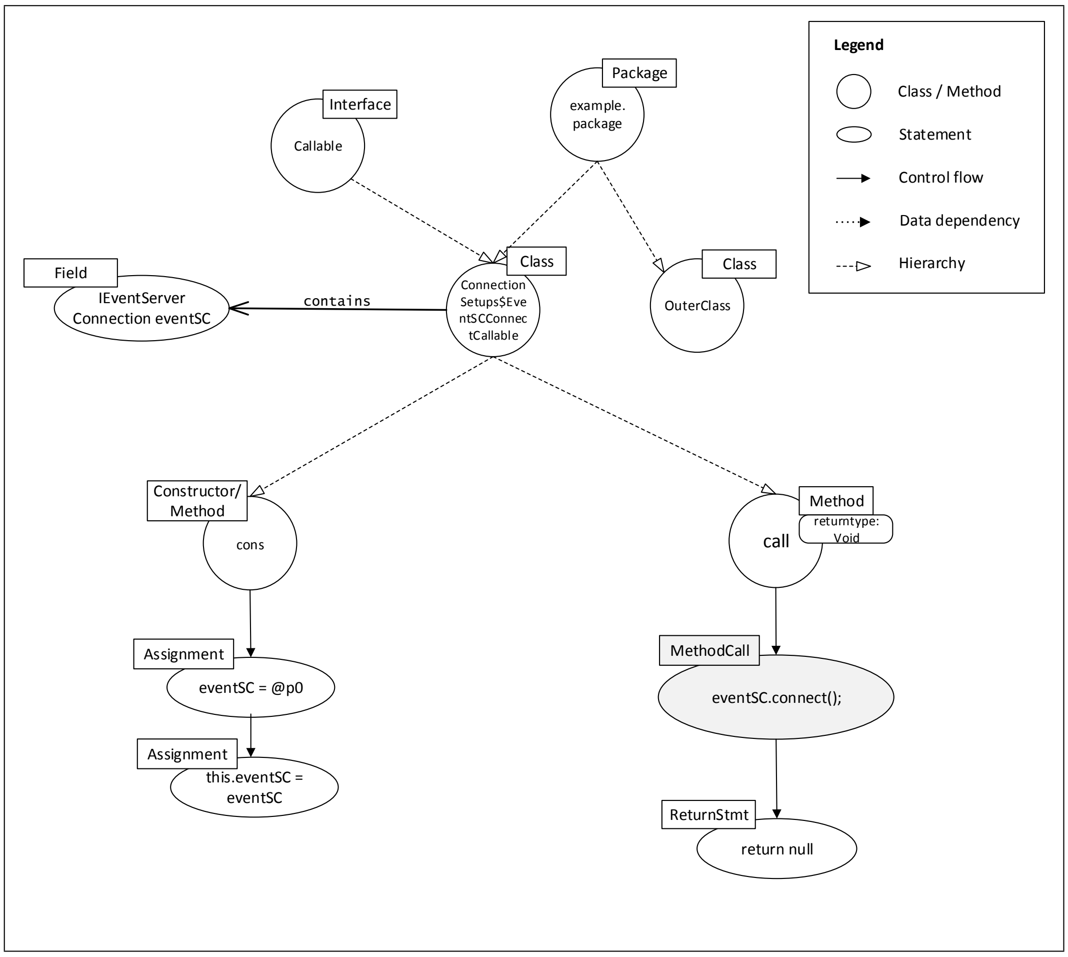


Figure 9.4. Continuation of the example parallelisation pattern: SDG of the Callable without Assignment

Parallelised source code for 'Reduction of an Array'

```
import java.util.*;

public class ParallelisedClassWithReductionOfAnArray {

   public static void main(String[] args) {

      int[] array = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 };

      int nThreads = ParallelisationConfiguration.NUMBER_OF_THREADS;
      ExecutorService pool = Executors.newFixedThreadPool(nThreads);

      int entriesPerThread = array.length / nThreads;
      int remainedEntries = array.length % nThreads;
      int startindex = 0;
      int endindex = 0;
      List<Future<Integer>> futureResults = new ArrayList<Future<Integer>>();
      for (int i = 0; i < nThreads; i++) {
         endindex = startindex + entriesPerThread - 1;
         if (remainedEntries > 0) {
            endindex++;
            remainedEntries--;
         }
         SumCallable callable = new SumCallable(array, startindex, endindex);
         Future<Integer> futureSum = pool.submit(callable);
         futureResults.add(futureSum);
         startindex = endindex + 1;
      }

      int sum = 0;

      try {
         for (Future<Integer> f : futureResults) {
            sum = sum + f.get();
         }
      } catch (InterruptedException e) {
         throw new IllegalStateException("Unexpected Interruption")
      } catch (ExecutionException e) {
         throw new IllegalStateException("Unexpected Exception")
      }
      System.out.println("Sum = " + sum);
   }

   private static class SumCallable implements Callable<Integer> {

      int startindex;
      int endindex;
      int[] array;

      public SumCallable(int[] array, int startindex, int endindex) {
         this.array = array;
```

```
            this.startindex = startindex;
            this.endindex = endindex;
        }

        @Override
        public Integer call() throws Exception {
            int sum = 0;
            for (int i = startindex; i <= endindex; i++) {
                sum = sum + array[i];
            }
            return sum;
        }

    }
}
```

Listing 9.3. Parallelised source code example for the candidate pattern 'Independent For-Each Loop'

Bibliography

[Allen 1970] F. E. Allen. Control flow analysis. In: *Sigplan Notices*. ACM, 1970. (Cited on page 12)

[Andrews 2000] G. Andrews. Foundations of Multithreaded, Parallel, and Distributed Programming. Addison Wesley Longman, Inc, 2000. (Cited on page 25)

[Asenjo et al. 2008] R. Asenjo, R. Castillo, F. Corbera, A. Navarro, A. Tineo, and E. Zapata. Parallelizing irregular c codes assisted by interprocedural shape analysis. In: *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. 2008, pages 1–12. (Cited on page 7)

[Bae et al. 2013] H. Bae, D. Mustafa, J.-W. Lee, Aurangzeb, H. Lin, C. Dave, R. Eigenmann, and S. P. Midkiff. The cetus source-to-source compiler infrastructure: overview and evaluation. *Int. J. Parallel Program.* 41.6 (Dec. 2013), pages 753–767. URL: http://dx.doi.org/10.1007/s10766-012-0211-z. (Cited on pages 7, 8)

[Basili and Rombach 1988] V. Basili and H. Rombach. The tame project: towards improvement-oriented software environments. *IEEE Transactions on Software Engineering* 14.6 (1988). (Cited on page 71)

[Edlich et al. 2011] S. Edlich, A. Friedland, J. Hampe, and B. Brauer. NoSQL: Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken. Second. Carl Hanser Verlag, 2011. (Cited on pages 11 and 16)

[Eichinger et al. 2008] F. Eichinger, K. Böhm, and M. Huber. Improved Software Fault Detection with Graph Mining. In: *International Workshop on Mining and Learning with Graphs (MLG)*. Edited by S. Kaski, S. Vishwanathan, and S. Wrobel. Helsinki, Finnland, 2008. URL: http://dbis.ipd.kit.edu/download/eichi/eichinger08improved.pdf. (Cited on page 6)

[Eichinger et al. 2010a] F. Eichinger, V. Pankratius, P. W. L. Große, and K. Böhm. Localizing Defects in Multithreaded Programs by Mining Dynamic Call Graphs. In: *Testing: Academic and Industrial Conference – Practice and Research Techniques (TAIC PART)*. Edited by L. Bottaci and G. Fraser. Volume 6303. Lecture Notes in Computer Science. Springer, 2010, pages 56–71. URL: http://dbis.ipd.kit.edu/download/eichi/eichinger10localizing.pdf. (Cited on page 6)

[Eichinger et al. 2010b] F. Eichinger, K. Krogmann, R. Klug, and K. Böhm. Software-Defect Localisation by Mining Dataflow-Enabled Call Graphs. English. In: *Machine Learning and Knowledge Discovery in Databases*. Edited by J. Balcázar, F. Bonchi, A. Gionis, and M. Sebag. Volume 6321. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pages 425–441. URL: http://dx.doi.org/10.1007/978-3-642-15880-3_33. (Cited on page 6)

[Eichinger et al. 2014] F. Eichinger, V. Pankratius, and K. Böhm. Data Mining for Defects in Multicore Applications: An Entropy-Based Call-Graph Technique. *Concurrency and Computation: Practice and Experience* 26.1 (2014), pages 1–20. URL: http://dbis.ipd.kit.edu/download/eichi/eichinger14concurrency.pdf. (Cited on page 6)

[Ghezzi et al. 2003] C. Ghezzi, M. Jazayeri, and D. Mandrioli. Fundamentals of Software Engineering. 2nd edition. Person Education, 2003. (Cited on page 31)

[Grove and Chambers 2001] D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst.* 23.6 (Nov. 2001), pages 685–746. URL: http://doi.acm.org/10.1145/506315.506316. (Cited on page 11)

[Hall et al. 2005] M. W. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lam. Interprocedural parallelization analysis in suif. *ACM Trans. Program. Lang. Syst.* 27.4 (July 2005), pages 662–731. URL: http://doi.acm.org/10.1145/1075382.1075385. (Cited on page 6)

[Heuzeroth et al. 2003] D Heuzeroth, T. Holl, G. Högström, and W. Löwe. Automatic design pattern detection. In: 2003. (Cited on pages 5, 6)

[Magee and Kramer 2006] J. Magee and J. Kramer. Concurrency: State Models and Java Programs. John Wiley & Sons, 2006. (Cited on pages 53, 54)

[Mattson et al. 2004] T. G. Mattson, B. A. Sanders, and B. L. Massingill. Patterns for Parallel Programming. Second. Addison Wesley, 2004. (Cited on pages 24, 25, 27–29, 53–55, and 57)

[McCool et al. 2012] M. McCool, J. Reinders, and A. Robison. Structured Parallel Programming. Morgan Kaufmann, 2012. (Cited on pages 24 and 30)

[Merlin 1991] J. Merlin. Techniques for the Automatic Parallelisation of 'Distributed Fortran 90'. Technical report. Department of Electronics and Computer Science, Southampton, 1991. (Cited on page 6)

[Molitorisz et al. 2012] K. Molitorisz, J. Schimmel, and F. Otto. Automatic Parallelization Using AutoFutures. English. In: *Multicore Software Engineering, Performance, and Tools*. Edited by V. Pankratius and M. Philippsen. Volume 7303. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pages 78–81. URL: http://dx.doi.org/10.1007/978-3-642-31202-1_8. (Cited on pages 7 and 29)

[Moseley et al. 2007] T. Moseley, D. A. Connors, D. Grunwald, and R. Peri. Identifying potential parallelism via loop-centric profiling. In: *Proceedings of the 4th International Conference on Computing Frontiers*. CF '07. Ischia, Italy: ACM, 2007, pages 143–152. URL: http://doi.acm.org/10.1145/1242531.1242554. (Cited on page 7)

[Neo Technology 2015a] Neo Technology. Neo4J. Overcoming SQL Strain and SQL Pain. 2015. (Cited on page 15)

[Neo Technology 2015b] Neo Technology. The Neo4j Manual. http://neo4j.com/docs/pdf/neo4j-manual-2.2.5.pdf. 2015. (Cited on pages 16–19)

[Netzer and Miller 1992] R. Netzer and B. Miller. What are race conditions? some issues and formalizations. *ACM Letters on Programming Languages and Systems* 1.1 (1992). (Cited on page 25)

[Ortega-Arjona 2010] J. L. Ortega-Arjona. Patterns for Parallel Software Design. John Wiley & Sons, Ltd., 2010. (Cited on pages 16, 24, and 53)

[Robinson et al. 2015] I. Robinson, J. Webber, and E. Eifrem. Graph Databases. New Opportunities for Connected Data. Edited by M. Beaugureau. Second. O'Reilly Media, 2015. (Cited on pages 11 and 15)

[Ryder 1979] B. Ryder. Constructing the call graph of a program. *Software Engineering, IEEE Transactions on* SE-5.3 (1979), pages 216–226. (Cited on page 11)

[Servetto and Potanin 2012] M. Servetto and A. Potanin. Automatic Parallelisation in OO Languages with Balloons and Immutable Objects. Technical report. Victoria University of Wellington, New Zealand, 2012. (Cited on page 6)

[Servetto et al. 2013] M. Servetto, D. Pearce, L. Groves, and A. Potanin. Balloon Types for Safe Parallelisation over Arbitrary Object Graphs. Technical report. Victoria University of Wellington, New Zealand, 2013. (Cited on page 7)

[Stencel and Wegrzynowicz 2008] K. Stencel and P. Wegrzynowicz. Detection of diverse design pattern variants. In: *2008 15th Asia-Pacific Software Engineering Conference*. 2008. (Cited on pages 5, 6)

[Sun et al. 2010] B. Sun, G. Shu, A. Podgurski, S. Li, S. Zhang, and J. Yang. Propagating bug fixes with fast subgraph matching. In: 2010. (Cited on page 6)

[Ullenboom 2014] C. Ullenboom. Java ist auch eine Insel. Einführung. Galileo Press Praxis, 2014. (Cited on pages 58, 59)

[Vallée-Rai et al. 1999] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In: *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*. CASCON '99. Mississauga, Ontario, Canada: IBM Press, 1999, pages 13–. URL: http://dl.acm.org/citation.cfm?id=781995.782008. (Cited on page 21)

[Van Bruggen 2014] R. van Bruggen. Learning Neo4J. Packt Publishing, 2014. (Cited on page 15)

[Wulf 2014] C. Wulf. Pattern-based detection and utilization of potential parallelism in software systems. In: *Software Engineering 2014, Fachtagung des GI-Fachbereichs Softwaretechnik, 25. Februar - 28. Februar 2014, Kiel, Deutschland*. 2014, pages 229–232. (Cited on pages 1, 2, and 21)

[Yan and Han 2003] X. Yan and J. Han. Closegraph: mining closed frequent graph patterns. In: *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '03. Washington, D.C.: ACM, 2003, pages 286–295. URL: http://doi.acm.org/10.1145/956750.956784. (Cited on page 6)

[Zhang et al. 2009] S. Zhang, S. Li, and J. Yang. Gaddi: distance index based subgraph matching in biological networks. In: *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*. EDBT '09. Saint Petersburg, Russia: ACM, 2009, pages 192–203. URL: http://doi.acm.org/10.1145/1516360.1516384. (Cited on page 6)