OPTIMIZING SPARQL QUERIES ON MULTI-CORE PROCESSORS

by

ANKUR OBERAI

(Under the Direction of Krzysztof J. Kochut)

ABSTRACT

The Semantic Web is an extension of the current World Wide Web where information is represented in a machine processable way. Resource Description Framework (RDF) is the format created for publishing data in the Semantic Web and SPARQL is the standard language for querying RDF data. The size of the RDF data sets being published has been ever increasing. Consequently, the execution times of SPARQL queries have been growing, as well. In this thesis, we present a novel SPARQL query execution algorithm which takes advantage of multi-processor and multi-core architectures and executes individual queries in parallel. We have implemented our algorithm in Java using Java threads and conducted a number of experiments on a multi-processor, multi-core computer system to assess the efficiency of the algorithm. The results of the experiments indicate a significant speed up of our parallel approach in comparison to the sequential SPARQL query processors in use today.

INDEX WORDS: SPARQL query optimization, parallel query execution, Semantic Web, RDF
OPTIMIZING SPARQL QUERIES ON MULTI-CORE PROCESSORS

by

ANKUR OBERAI

BTECH, DehraDun Institute of Technology, India, 2008

A Thesis Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment

of the Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2011
OPTIMIZING SPARQL QUERIES ON MULTI-CORE PROCESSORS

by

ANKUR OBERAI

Major Professor:   Krzysztof J. Kochut
Committee:        John A. Miller
                 Khaled Rasheed

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
August 2011
DEDICATION

This thesis is dedicated to my parents.
ACKNOWLEDGEMENTS

I would like to thank Dr. Krzysztof J. Kochut, for all his support. I would also like to thank my friends Chinmay Vishnu Kale (the java guru) and Henning Meyerhenke (parallel computing expert) for their valued suggestions.
TABLE OF CONTENTS

Page

ACKNOWLEDGEMENTS ........................................................................................................... v

CHAPTER

1 INTRODUCTION ................................................................................................................1

1.1 Introduction to the Semantic Web ................................................................................. 1

1.2 Why the Semantic Web? ............................................................................................... 2

1.3 Motivation ....................................................................................................................... 4

2 BACKGROUND ................................................................................................................. 6

2.1 Resource Description Framework (RDF) .................................................................... 6

2.2 RDF Schema (RDFS) .................................................................................................. 10

2.3 Web Ontology Language (OWL) ................................................................................. 10

2.4 SPARQL Protocol and RDF Query Language (SPARQL) ............................................ 11

3 RELATED WORK ............................................................................................................. 18

4 PARALLEL EXECUTION OF SPARQL QUERIES ......................................................... 21

4.1 A High Level View ...................................................................................................... 21

4.2 Outline of the Method ................................................................................................. 21

4.3 Factors affecting the Parallel Execution ........................................................................ 24

4.4 A Detailed View .......................................................................................................... 26

4.5 Segmenting the taskQueue .......................................................................................... 32
1.1 Introduction to Semantic Web

The World Wide Web has become an indispensable part of our lives. The ability of the humans to create content on the web has given rise to a large amount of data. Today, people can put out their thoughts in the form of blogs, podcasts, videos, etc. Social networking gives a new medium to propagate this content. The web is truly by the people and for the people. Mankind produced 150 exabytes of information in 2005 and is expected to keep the numbers soaring to 1,200 exabytes in 2010 [1]. The huge amount of data on the web makes it difficult to present, access and update that information. The web has grown to the extent that the latest research problems we are addressing are related to managing that data. There is so much data on the web, that it is becoming increasingly hard to find relevant information.

When browsing the current web, the audience learns what is present in a web page, navigates from page to page to explore further. The navigation is made possible as each HTML page can freely point to other pages. Each page contains some information which can be analyzed to make decisions. The World Wide Web (WWW) brings the ability to request for any web page, which when returned is displayed by browsers. This was one of the dreams of the founder of the WWW, Sir Tim Berners Lee. The second goal was to bring the web to a level where it is an automated, integrated and a self adapting web.

The word ‘syntax’ means how something is represented and the word ‘semantics’ refers to what it means. The first reference of adding machine processable semantics to the web was
made in the year 1994 at the first World wide web conference at CERN, Geneva [2]. This would in turn provide a better experience for the users. This new extension to the current web was named the Semantic Web. The high level technological components behind the realization of the Semantic Web are as follows: Resource Description Framework (RDF) [3], RDF schema (RDFS) [4], Web Ontology language (OWL) [5] and SPARQL query language (SPARQL)[6]. It is established by including machine processable information in files without modifying the underlying web structure.

The Semantic Web aims at linking the data in the web pages instead of the web pages themselves. Using the Semantic Web, software agents would be able to mediate between the user needs and the information present on the web. This would in turn lower the burden on the end user, as machines would provide better filtering, categorization and searching of information sources [7]. With the Semantic Web in effect, users will be able to browse distributed knowledge rather than a distributed presentation layer on the web.

1.2 Why the Semantic Web?

The computers can understand the web pages as per the HTML tags included on the page. The HTML tags can only convey information about the rendering of the content. With the current web, when a task has to be performed, the user looks up an appropriate web page. Choosing this web page is in itself a difficult decision. For example, when looking for an air ticket, the customer might go directly to the airlines website (among the many airlines) or might want to book through some agent which gets cheaper tickets. Finally, the end user might not even be sure of whether he got the best deal or if there were some unexplored options. While browsing, one might come across incomprehensible data as it might be written in some foreign language.
Interesting items on a page can be further explored by clicking the link or starting a new search for the item. The developer selects the data source he wants to make accessible from his page and pulls out the data he wants to be displayed. The users of the web use the presentation layer and not the actual data layer. The cause of this problem is that the data on the current web is scattered and in different formats.

The ultimate goal of the Semantic Web is to use the data on the web the same way we use the web pages right now [8]. This can be made possible by envisioning a new format of data that should also be understandable by the machines in order to give them the reasoning power. This new connected data forms the web of data which is a building block of Semantic Web applications.

The need for a different data format for the Semantic Web is filled by RDF. The entities on web pages can be considered as resources. RDF gives the ability to make statements about resources. As the Semantic Web gains popularity among the research community as well as the industry [9], the amount of the RDF data is on the rise. RDF is a great fit for applications which deal with ad-hoc incoming data and is widely used in self publishing software, such as Livejournal [10] and Typepad. Today, RDF databases can have millions (or even billions) of triples (e.g. [11] [12]). Efforts such as linked data community [13] have come up with standards to publish and connect the data on the web. The adoption of these practices has brought together data from many diverse domains. To make any use of data, query mechanisms should be available. This is where SPARQL comes into the picture and gives the power to query the valuable RDF data. With ever more semantically annotated data being added, more complex queries need to be formulated to serve the user with relevant information. Executing these complex queries over ever larger data sets adds to the query execution time. Optimization of
SPARQL queries is therefore an important goal. This thesis focuses on lowering the running time of SPARQL queries by executing them in parallel.

1.3 Motivation

When machines can understand the content of a web page they will be able to provide better filtering, categorization and presentation of content. The search engines will be able to fetch more accurate results as they will no longer be searching based just on keywords. This would lead to more automation and convenience in the daily use of the web. This promise of the Semantic Web vision is driving the growth in the RDF data. The community felt a strong need to be able to query this valuable RDF data. We chose to optimize SPARQL as it is the standard for querying RDF data. The interest in this field is indicated by the fact there were a number of RDF query languages like rdfDB [14], RDQL [15], and SeRQL [16] even before SPARQL was finalized by W3C as a recommendation [17]. W3C is actively working on enhancing features in SPARQL through the SPARQL Working Group [18].

The web of data opens up a number of possibilities for the user. One such possibility is to execute complex queries over distributed data sources. The current way to work with distributed RDF sources is to fetch the data in a centralized triple store and then execute queries against it [19]. This might at times lead to huge RDF files. In [15], the authors present a report on the current alternative to gathering this data in files or in a database. As indicated, the approaches are still in a stage of infancy and none of the implementations exist on a large scale. To ease the problem of huge time consumption, we try to optimize the execution time for the SPARQL queries so that the user can get good response time even while posing complex queries to huge datasets.
There has been some work on optimizing SPARQL queries, but not much work has been done on parallelizing their execution. In this thesis, we aim to speed up the execution of SPARQL queries by implementing an approach which makes use of the multiple processing units.

A considerable amount of research has gone into optimizing Relational Database Systems. This thesis is a step towards speedup of queries executed against RDF data. RDF data model can either be stored in the main memory or the disk. For the disk based model, the RDF data is stored in RDBMS (Relational Database Management System) and the SPARQL queries are re-written in SQL and then executed. The focus of our work is to study the query optimization for the main memory RDF model. Given a query, there can be many plans (orders of execution) that can be followed to produce the answer. All plans are the same with respect to the output they produce, but vary in the amount of time they take to run. In this work, it is assumed that a good query plan is given and then we try to minimize the execution time by implementing a parallel iterator based algorithm. The independent parts are executed in parallel on different processors. The aim of this algorithm is to make a good use of the computing resources by keeping them busy as long as the query is executing. We execute a SPARQL query using pipelining iterators. The details of our algorithm are presented later in the thesis.
CHAPTER 2

BACKGROUND

In this chapter, we explain in detail some of the technical terms used extensively throughout the thesis. We cover core technologies making up the Semantic Web as well as some terms which might help understand the implementation.

The current web is based on the granularity level of web pages, so web pages are identified uniquely through URLs. In case of the Semantic Web the aim is to shift to a finer granularity level of the data contained in the pages. In the Semantic Web, the entities on a web page are considered as resources. A resource might be anything from a person or a book to another web page, describing some resource. Information is composed of the statements made about resources on the web page. All the resources on the web pages need to be uniquely identified in order to make unambiguous statements about them. These unique names of items on the web are called Uniform Resource Identifiers [20] (URI). A Uniform Resource Locater (URL) is a form of a URI which tells us where to find a particular page on the web.

2.1 Resource Description Framework (RDF)

RDF is a language used to represent information in the Semantic Web. It is used for describing resources on the web in a machine readable manner. A resource is anything we can identify with a URI. It helps create a web of data by creating links between the data which is already available on the web.
RDF data can be represented in the form of triples. A triple consists of the following three parts:

Subject: item being talked about.

Predicate: the property mapping a subject to an object.

Object: the value of the above property for the above object.

The object of a statement can either be a resource or a literal.

In the graph representation in Figure 1, the subject is in the ellipse and the object is in the rectangle. The labeled edge represents the property. The parts of an RDF triple are divided into two parts by a ‘:’. The part before the ‘:’ is used to represent the namespace. This concept is borrowed from the XML qualified names to simplify the representation of the properties. The part after the ‘:’ is the local name. The graph above represents a declarative RDF statement stating-“http://myexample..../JohnTaylor has the email address JohnATaylor@xyz.com”.

![Figure 1 - Graphical representation of an RDF Triple](image-url)
When considering the RDF data model as a graph, a triple can be thought of as the subject node being connected to the object node by a directed edge, called the predicate. The predicate serves the dual purpose of representing the value of an attribute for a resource and also for defining a relation between two resources. A triple gives the power to express a declarative statement. Each triple represents a single knowledge fact. Triples can be collected and searched as machine-readable graphs with each of its components tagged with a Uniform Resource Identifier. Namespaces can be used to simplify the URIs. A sample RDF graph for the RDF/XML in Appendix C is shown in Figure 2.
Figure 2 - A sample RDF Graph
RDF helps merge different data sources by giving the ability to explicitly state that a particular entity in a dataset is the same as some other in a different dataset. By using a URI, for each of the terms RDF allows anyone, anywhere, anytime to add information about a resource. RDF thus integrates data sources at different locations on the web, in different formats and having different names for the relations. RDF triples form a directed labeled graph. RDF is often serialized as XML and some other human friendly formats like N3 [21], NTriples [22] and Turtle [23]. RDF data can be queried using tools written in Java, C++, Python, PHP and others [24].

2.2 RDF Schema (RDFS)

RDFS is used to add some extra knowledge to RDF. It defines the terms that can be used, restrictions and any extra relationships that exist. RDFS defines resources, classes and properties that create a taxonomy for arranging the RDF data. A taxonomy can be formed both for the classes and the properties. Everything in RDFS is a resource and these resources can be grouped together into Classes. The members of a class are called instances of that Class. A particular resource can be an instance of more than one Class. The Classes are also resources and can be defined by properties. A property is the relation between the resources. The domain and the range of the properties can be specified. RDFS thus allows limited amount of reasoning.

2.3 Web Ontology Language (OWL)

An ontology is used to describe the kinds of entities in the world and how they are related. As such, an ontology should provide ways to describe the following items:

- Classes or things relevant in a particular domain.
- Relationship between those things.
- Properties or attributes of those things.

OWL [5] makes RDF and RDFS more expressive by adding vocabulary based on description logic. OWL adds to the limited reasoning power provided by RDFS. OWL provides a way to state transitive, inverse and symmetrical properties. OWL also provides cardinality constraints. OWL is a collection of three increasingly expressive sub languages:

OWL Lite: Capable enough for representing simple ontologies by providing hierarchies of classes and properties.

OWL DL: OWL DL is more expressive than OWL Lite and is decidable.

OWL Full: OWL Full is an extension of RDF. An owl full ontology is not decidable. With OWL Full the complete syntax of RDF can be used.

Every legal OWL Lite ontology is a legal OWL DL ontology and every legal OWL DL ontology is a legal OWL full ontology. If a conclusion is valid in OWL Lite, it will be valid in OWL DL and if a conclusion is valid in OWL DL it will be valid in OWL Full. Every valid OWL document is a valid RDF document but all RDF documents might not be valid OWL Lite or OWL DL documents.

2.4 SPARQL Protocol and RDF Query Language (SPARQL)

SPARQL Protocol and RDF Query Language (SPARQL) [6] is the W3C standard for querying RDF data graphs. SPARQL is to RDF databases what SQL is to relational databases. RDF graphs can be considered to be a collection of triples. The difference between a triple and a triple pattern is that in a triple pattern any of the subject, predicate or object can be a variable. A triple pattern has to have at least one of the subject, predicate or object as a variable. If the variable
part can have a value in some triple in the data, then that triple is said to match the triple pattern. This matching of a triple to a triple pattern is called a binding. A SPARQL query can be used to find such triples matching a triple pattern. A number of triple patterns together make a graph pattern. SPARQL queries are written using basic graph patterns.

A query will return results if there is a binding for its graph pattern in the RDF data. This can be thought of as finding a subgraph in the main RDF graph. Subgraphs of the RDF graphs are obtained by binding the symbols. If there is a selection, the query returns bound resources. SPARQL is usually used over the network through SPARQL endpoints. A SPARQL endpoint accepts a SPARQL query and returns the results over HTTP (Hyper Text Transfer Protocol) [25]. A SPARQL endpoint could be specific (bound to a particular dataset) or generic (allows the user to specify the dataset for query execution). The syntax for the SPARQL query language is SQL-like, making it familiar to the database users.

SPARQL provides functionality to:

- Extract information in the form of URIs, blank nodes and literals.
- Extract RDF subgraphs.
- Construct new RDF graphs based on information in the queried graphs.

In the query shown in Listing 1, the `PREFIX` is defined so that the full URL is not required to be typed every time it is referred to. What comes after the `SELECT` clause is what is returned as a result of the query execution. ‘?’ or ‘$’ at the beginning of an identifier can be used to represent SPARQL variables. The `FROM` clause is optional and represents the location of the RDF data. This can either be a URL of the dataset on the web or a local file. A series of triple patterns follow the `WHERE` clause. These triple patterns together compose the graph pattern.
The query in Listing 1, consist of three triple patterns joining on the common variable `?professor`. In this example, the model Univ.rdf will be searched for triples where there is a subject for a predicate of a `ub:teacherOf` and some `?course`. The subject of the resulting statements is bound to the SPARQL variable `?professor`. The second triple pattern searches for all the triples where the subject is the bound value of `?professor`, having a predicate of `ub:undergraduateDegreeFrom` and an object bound to the variable `?uni`. The third triple pattern searches for all the triples where the subject is the bound value of `?professor`, having a binding for the predicate of `ub:emailAddress`. All the triple patterns in the graph represent an and condition. In the end, all the values bound to `?professor` and `?course` are returned. Considering the data in Figure 2 the results derived are shown in Listing 2
Note that \(< \text{http://www.Department0.University0.edu/FullProfessor1} >\) is not listed in the result as it does not have a binding for the triple pattern \(?\text{professor ub:emailAddress ?email}\).

SPARQL resembles SQL to some extent. However, in the relational model [26], the table structure is rigid. As such the column names with no values have to be NULL. In the RDF model, information is represented by statements. The information represented by a triple is either present or not. This makes sure that there are no NULLs in the RDF model. The relationship between the tables has to be inferred and is not explicit, sometimes resulting in complex joins across tables. Also the data and relationships in a database are local to that database and this makes integration of disparate sources an intricate issue. By structure, the RDF model is of graph nature and allows voluntary schema adjustment [27]. RDF data conforms to the open world assumption. Thus the absence of a statement in the current model does not mean that it is not true. It can be added independently later. Linking data across different datasets is as simple as adding a statement with the subject and the object belonging to the different domains to be linked. Adding new data is also really simple with RDF. A new schema does not have to be added for every new database, but this can be accomplished by adding another RDF file to the

<table>
<thead>
<tr>
<th>(?\text{professor})</th>
<th>(?\text{course})</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{<a href="http://www.Department0.University0.edu/FullProfessor0%7D">http://www.Department0.University0.edu/FullProfessor0}</a></td>
<td>\text{<a href="http://www.Department0.University0.edu/Course0%7D">http://www.Department0.University0.edu/Course0}</a></td>
</tr>
<tr>
<td>\text{<a href="http://www.Department0.University0.edu/FullProfessor0%7D">http://www.Department0.University0.edu/FullProfessor0}</a></td>
<td>\text{<a href="http://www.Department0.University0.edu/GraduateCourse0%7D">http://www.Department0.University0.edu/GraduateCourse0}</a></td>
</tr>
<tr>
<td>\text{<a href="http://www.Department0.University0.edu/FullProfessor0%7D">http://www.Department0.University0.edu/FullProfessor0}</a></td>
<td>\text{<a href="http://www.Department0.University0.edu/GraduateCourse1%7D">http://www.Department0.University0.edu/GraduateCourse1}</a></td>
</tr>
</tbody>
</table>

Listing 2- Results of the Query
existing triple store. One of the driving forces behind the Semantic Web is the need to integrate together the data originating from various resources. The web of data is a web where each RDF triple is a part of a global datastore. RDF is a step towards satisfying the need to integrate, mine and analyze data from different parts of business and industries.

The success of the Semantic Web technology rests on the availability of large amounts of interlinked RDF datasets. The W3C SWEO [28] project Linking Open Data [29] is dedicated to publishing large RDF datasets on the web. These datasets are linked by introducing triples where the subject is the URI from one dataset and the object is the URI from a different dataset. The D2RQ platform [11] is one of approaches to convert the data present in the Relational databases to RDF. A mapping defines how properties and resources are generated out of relational databases. A survey of techniques for RDB to RDF conversion is presented in [30]. Systems like Revyu.com [31] have made it easy for the user to generate some RDF metadata while reviewing real world articles. All these initiatives add to the huge amount of RDF data that Semantic Web applications would have to query.

DBpedia [32] is a community effort to extract structured information from the Wikipedia. Various datasets on the web have been linked together to form the linked data cloud [33] depicted in Figure 3. The data in the linked data cloud uses various serialization syntaxes for RDF for encoding the data. The number of participating data sources in this cloud are doubling every 10 months since 2007 [34].
With this high amount of data, improvements in the SPARQL query execution is especially important.

There are two parts to query optimization:

- Generating an optimal query plan.
- Executing the query plan optimally.

In the current work, the second point is addressed as applied in case of SPARQL queries. Hence we assume that we are given a good query plan.
Most of the techniques for parallel processing of joins in relational databases are based on parallel hash join algorithms e.g., [35] [36] [37] [38] [39] and [40]. Most work done by the database community does not specialize on the RDF triple model and triple patterns of the SPARQL queries.

The approach presented in this thesis harnesses the power of multiple CPU’s on a shared memory architecture.
CHAPTER 3

RELATED WORK

There have been some attempts at increasing the speed of processing of SPARQL queries. Looking at a broad picture, there are a number of steps involved from the time query is posed to the time the results are available to the user. There have been attempts to optimize these different steps.

To a large extent, query execution time is hugely impacted by the Query execution plan. If the results in the intermediate stages of a query can be reduced, this leads to a significant pruning of the solution search space. In OptARQ [41] a technique called selectivity estimation is used to minimize the size of the intermediate result sets of triple patterns. The proposed framework reorders the triple patterns after the query parsing and syntax checking, to generate an optimal Query Execution Plan. The system caches some statistical data about the endpoints. This data is used to design elementary cost functions. The triples are ranked according to a cost generated by a cost function which takes into account the cost of the subject, predicate and the object of the triple.

Optimized index structures for RDF databases are studied in [42]. The authors adapt the database techniques for RDF storage and indexing. The indexes make the lookup fast, thus improving the query processing time. This work deals with optimizing the disk-based RDF model.

The Semantic Web or the web of data enables the users to pose complex queries which might be run on distributed data sources. A federated query searches multiple distributed data sources concurrently. In SPARQL query execution, data sources are discovered asynchronously...
at runtime by traversing the links in the RDF data. A recent report on the current approaches to federated query execution for SPARQL is presented in [19]. Federated query execution would ease the requirement of loading huge RDF data on a centralized source. Also executing SPARQL queries in a federated manner could reduce the execution time but a point to note here is that the approaches described are still in their infancy and have no large scale implementations.

In [43], the authors try to minimize the run time of queries by using pre-materialized sub-queries. Sometimes it can be determined that some joins are related closely and thus almost always occur as a part of the query. Such information can be processed in an offline query so that the results are available when the other longer queries need them. The run time execution of the query uses a combination of such materialized views from smaller queries to generate the final result.

Attempts have been made to parallelize the joins within the SPARQL queries [44]. A bounded buffer is a data structure useful in multi-threaded producer-consumer problem. A constant associated with a bounded buffer can be defined which specifies the storage limit of the buffer. The `get` and the `put` operations on the bounded buffer are guarded by a monitor. A `get` operation on an empty buffer, causes the thread to sleep unless some more elements are available in the buffer. Similarly, a `put` operation on a full buffer causes the producer thread to sleep until some element is removed from the buffer. In this approach the author splits the data amongst the threads used for join computation. The triples are split based on the value of a join variable. All the triples having a particular value for a join variable are allotted to the same thread. The data is split using hash partitioning such that the threads have all the data they need in their bounded buffer. The threads perform a sequential join algorithm on its two input bounded buffers. The result of the join is processed for other operators in the query.
At the time of this writing, not much work has been done on parallel execution of SPARQL queries. This thesis presents a method of executing SPARQL queries in parallel.
CHAPTER 4

PARALLEL EXECUTION OF SPARQL QUERIES

This chapter explains our parallel algorithm in detail. Also explained is the rationale behind certain technical decisions behind the implementation. The reader, in this section will see how each component fits into the overall implementation.

4.1 A High Level View

The implementation uses the Supervisor-Matcher thread model. The main thread is the Supervisor thread. The Supervisor thread generates tasks for the Matcher threads and then spawns the Matcher threads. The main thread then waits for all the Matcher threads to finish and then ends.

A Matcher thread is initialized by the Supervisor. It is passed a reference to the common data structure which contains the work for all the threads. This data structure is implemented as a queue. The Matcher threads obtain the tasks from the queue and work on them until the queue is exhausted. Once there is no more work to be done, the threads coordinate their end to the Supervisor and exit.

4.2 Outline of the Method

To obtain high level of parallelism we design a data structure called a task. A task consists of a list of triples and an integer called level. The level of a task represents the query plan iterator which needs to be executed on the task. The list of statements represents the triples bound to
the different levels of the query plan iterators which have already been explored for this task. The tasks procured from the queue can be executed by each thread independently. A thread can explore all the statements forming the solution starting from a fixed value of a particular node. Once a solution is generated it can be stored in separate files which can be merged to get all the solutions to the query. We maintain a buffer called taskQueue which contains the work to be done by the threads.

The operation of the algorithm can be best explained with an example. Let us consider the RDF graph in Figure 2 and the query in Listing1 for our example. For a simple representation the RDF data in the graph in Figure 2 can be represented as:

<table>
<thead>
<tr>
<th>teacherOf</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P₀</td>
<td>C₀</td>
</tr>
<tr>
<td>P₀</td>
<td>GC₀</td>
</tr>
<tr>
<td>P₀</td>
<td>GC₁</td>
</tr>
<tr>
<td>P₁</td>
<td>GC</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>underGraduateDegreeFrom</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P₀</td>
<td>uni84</td>
</tr>
<tr>
<td>P₁</td>
<td>uni84</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>emailAddress</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P₀</td>
<td>P₀@d0.u0.edu</td>
</tr>
</tbody>
</table>

Each of the triple patterns of the query can be associated with an iterator over the statements which are the solution to that triple pattern. Assuming a good query plan, evaluating the query in the order presented will give us the optimal result.
The first step of the algorithm is to completely materialize the first iterator. Materializing implies finding the bindings of the variables which satisfy the given triple pattern.

In this example, the first iterator would produce the following triples represented by the corresponding statements:

<table>
<thead>
<tr>
<th>Subject</th>
<th>Predicate</th>
<th>Object</th>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₀</td>
<td>teacherOf</td>
<td>C₀</td>
<td>S₀</td>
</tr>
<tr>
<td>P₀</td>
<td>teacherOf</td>
<td>GC₁</td>
<td>S₁</td>
</tr>
<tr>
<td>P₀</td>
<td>teacherOf</td>
<td>GC₀</td>
<td>S₂</td>
</tr>
<tr>
<td>P₁</td>
<td>teacherOf</td>
<td>GC₂</td>
<td>S₃</td>
</tr>
</tbody>
</table>

In this case, the following tasks will be added to the taskQueue:

<table>
<thead>
<tr>
<th>List of Statements</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;S₀&gt;</td>
<td>1</td>
</tr>
<tr>
<td>&lt;S₁&gt;</td>
<td>1</td>
</tr>
<tr>
<td>&lt;S₂&gt;</td>
<td>1</td>
</tr>
<tr>
<td>&lt;S₃&gt;</td>
<td>1</td>
</tr>
</tbody>
</table>

Once the taskQueue has been populated with the tasks from the first iterator, the Supervisor spawns the Worker threads which procure a task from the taskQueue. Each worker can independently execute the remaining iterators on the task. The threads procure task from the taskQueue and then explore the solution tree in a depth-first manner. The thread which works on the first statement will try and find a binding for <P₀, undergraduateDegreeFrom ?uni>. In this case this would result in <P₀, undergraduateDegreeFrom, uni84>. The data is then searched for a binding like <P₀, emailAddress, ?email>. A solution exists if there is a binding for all the variables in the query. In this case the triple <P₀, emailAddress, P₀@d0.u0.edu> completes the query solution.

The final solution written by the thread is a collection of statements, each representing the solution at different iterator levels. In this case the solution would be as shown below:
In the sequential version, a single thread procures tasks from the task queue and explores the iterators in a depth-first manner for a solution. The thread procures the next task once it has done the complete depth first exploration for a single task. In the case of parallel implementation, multiple threads procure tasks from the taskQueue and begin to explore the solution tree depth-first, in parallel.

4.3 Factors affecting the Parallel Execution

Suppose only one of the statements generated after the first Iteration had computation intensive work. The thread working on this statement might take a lot of time to finish while the other threads will finish early. The other threads are idle while one thread continues to work. This lowers the overall processor utilization. For effective resource utilization all the threads should be kept busy for possibly all the time the application is executing.

There are two heuristics we use in the implementation. The combination of these heuristics decides how many tasks have been materialized in the taskQueue. We have the following parameters in the implementation:
**taskThreshold**

This is the minimum amount of tasks (work units) which this approach tries to maintain in the common data structure for work. This is done to uniformly distribute the work amongst the Matcher threads. When a thread finishes executing the query plan for a task, it polls another task from the task queue. Before getting the task from the buffer a check is made on the number of tasks in the buffer. If the number of tasks in the buffer is less than `taskThreshold`, the thread checks for `iteratorThreshold` and then materializes the current iterator of the task. The resultant materialized tasks are added back to the buffer and then the thread picks up one more task and releases the lock on the buffer. This is done to make sure that the `taskQueue` is not empty prematurely. Thus, this parameter encourages the threads to materialize.

**iteratorThreshold**

The above step will continue to add work to the `taskQueue`. To put an upper limit on the amount of work in the taskqueue we have a parameter called `iteratorThreshold`. A check will be made on the iterator level of the `task` being taken out of the buffer. Eg. If this is 3 and the `iteratorThreshold` is set to 3 or less, then no materialization will take place. The `taskQueue` can contain tasks at different level ranging from 1 to the `iteratorThreshold`.

The work items which are already evaluated to a higher iterator level will take lesser time to execute on the processor. If we allow tasks to be added at any level of the iterator materialization, the `taskQueue` would be full of tasks which take a very small time to execute. In such a case, more time is spent in thread trying to gain work and less time actually processing it. The swapping is increased which increases the overhead leading to thrashing. Thus, this parameter restricts the threads from materializing beyond a certain iterator level.
It is really important to limit the materialization involved in our approach. Materialization makes sure that work is available for the Matchers, but on the other hand consumes memory. The two parameters discussed above help us achieve this balance.

The `taskQueue` initially contains tasks which are materialized for the first level. As the size of the `taskQueue` reduces and materialization takes place, there will be tasks of higher iterator levels. It is to be understood that a task with a higher level of materialized iterator will take lesser time to execute.

### 4.4 A Detailed View

The evaluation algorithm has been implemented in Java, Jena and Java threads have been used for parallel processing. This section walks the reader through the pseudo code as the implementation is discussed. The implementation of the Supervisor and the Matcher are discussed separately.
**PseudoCode**

Supervisor

1. SET propertiesFile = location of the properties file
2. SET the execution parameters:
   3. maxTaskThreshold
   4. maxIteratorThreshold
   5. maxThreadCount
   6. queryno
   7. data
3. SET model = READ the rdf data file in a jena Model
4. SET queryplan = Get query Plan based on current model and query
5. READ t1 = CALL getCurrentThreadCPUtime()
6. CALL firstIteration(queryplan)
7. FOR i = 0 to noThreads
   8. SET passQueryPlan = CALL getUserInput(model, queryno)
   9. CREATE WorkerThread
   10. START MatcherThread
   11. ENDFOR
8. WAIT for all WorkerThreads to Finish
9. SET t2 = CALL getCurrentThreadCPUtime()
10. SET t = t2 - t1
11. Subroutine firstIteration(QueryPlan queryplan):
   12. SET s = List of statements in queryplan
   13. CASE iteratorType of queryplan
      14. Subject Predicate given:
      15. Set stmIterator = Construct iterator
      16. Predicate object given:
      17. Set StmIterator = Constructor iterator
      18. Predicate given:
      19. Set Stmiterator = Constructor iterator
   20. ENDCASE
   21. Materialize StmIterator and add tasks(Results) to taskQueue
   22. END firstIteration

**Supervisor data Structures:**

**QueryPlan:**
- int statementIndex;
- int iteratorIndex;
- Property property;
- Model model;
- ResourceType resourceType;
- RDFNode rdfNode;
- IteratorType iteratorType;

**Result**
- List<Statement> path;
- int level;

enum ResourseType : {
  SUBJECT, PREDICATE, OBJECT
}

enum iteratorType: {
  S, P, O, SP, SPO, PO, SO
}
1 Get lock on taskQueue
2 IF (size(taskQueue)==0)
3   killThread
4 ELSE
5   IF (size(taskQueue)<taskThreshHold)
6     SET datastructure firstTask=taskQueue[0]
7     IF firstTask[level]<iteratorThreshHold
8       materializeMode(firstTask)
9     SET firstTask=POLL(taskQueue)
10    ENDIF
11  ENDIF
12  SET firstTask=POLL(taskQueue)
13 ENDIF
14 RELEASE lock on taskQueue
15 IF (firstTask!=NULL)
16   dfs(firstTask)
17 ENDIF
18 END
19 SUBROUTINE killthread()
20   Signal end to waiting Controller
21 END SUBROUTINE
The implementation can be divided into the following steps from the Supervisor’s view:

- **Query setup** (for experiments purpose only) (Pseudocode 8):

From a usability standpoint, a data structure is defined which takes the user query input, iterator by iterator and generates a query plan. This simplifies the way the user frames the input query.

- **Materializing the first iterator** (Pseudocode 11):

```java
SUBROUTINE dfs(Result work)
    SET indx = work.getLevel();
    SET boolean firstTime = true;
    SET boolean backtrack = false;
    StatementIterator iterator = null;
    WHILE (true)
        WHILE (indx < noIterators)
            IF (firstTime)
                firstTime = false;
            ELSE
                IF (iteratorStack.empty() || indx == 0)
                    return;
                ENDIF
            ENDIF
            IF (backtrack == false)
                QueryPlan currQueryPlan = null;
                currQueryPlan = this.queryPlans[indx];
                currQueryPlan.getStm().clear();
                currQueryPlan.addStatements(work.getPath());
                iterator = getIterator(currQueryPlan);
            ELSE
                iterator = iteratorStack.pop();
                backtrack = false;
            ENDIF
            IF (iterator.hasNext())
                iteratorStack.push(iterator);
            ELSE
                Break = true;
            ENDIF
            Statement stm = iterator.nextStatement();
            work.getPath().add(stm);
            indx++;
        ENDWHILE
        IF (indx == noIterators)
            List<Statement> res = work.getPath();
        ENDIF
        indx--;
        IF (work.getPath().size() != 0)
            work.getPath().remove(work.getPath().size() - 1)
        ENDIF
        backtrack = true;
    ENDWHILE
END SUBROUTINE
```
Following the pseudo code from the Supervisor thread’s view:

- Materializing the first iterator (Psuedo code 11):

  The Supervisor loads the data and then executes the first iterator. This step is used to generate work for the threads. The power of a multi-processor is exploited when the main thread has finished with materializing the first iterator. Parallel implementation of the execution of the first iterator might be a part of future work.

  One of the main benefits of parallel programming is to improve performance. In our case, the improvement in the performance refers to lowering the execution time of the application by using the given processing units more effectively. Higher performance is achieved by having work readily available for the processing units as and when they become free. To make use of the power of multi-processor we need to make sure that the problem is divisible into parts which can be executed in parallel. The lesser interaction these parallel executions have, the more efficient the application. Each of statements is a perfect example of an independent task. This can be best understood with the help of an example. Suppose a query consisted of four iterators. Remember each of the “.” conditions is a conjunction. For a statement to be a part of the final result, a binding has to exist for this statement for all of the iterators in the query plan as explained in section 4.2. After the first iterator has been materialized, the remaining three still need to be executed for every statement. If a binding does not exist for a particular statement for a particular iterator, this statement is discarded without exploring further iterators. In our assumption of a high-end machine, we assume a system with a large amount of main memory. Complete materialization is done to make full utilization of the memory available so that tasks are not exhausted often. In [44] the author used a bounded buffer
in place of the queue. Although this might be beneficial in cases where memory is limited, we expect it to hamper the speedup. A bounded buffer has a pre-defined capacity. If a thread tries to add an element to a full buffer, the thread is made to sleep until there is space to accommodate more tasks. The threads having to wait to materialize might lower the efficiency.

- Spawning threads to work on the tasks produced by the first iterator (Pseudo code 12-16):

Java threads have been used to implement the Matcher threads. Each of the threads has a reference to the main memory data structure where the RDF triples are stored. In the case of multiple threads inside a single JVM, Jena provides a multiple reader/ single writer concurrency support through locks [45].

- The main thread waiting for the Matchers to finish all the work in the common queue (Pseudo code 17):

The wait mechanism for the Supervisor was implemented with the class CountDownLatch [46]. It allows for one or more threads to wait until a set of operations are completed. Our problem at hand is a typical scenario where this Java class can be used.

Following the pseudo code from the Matcher thread’s view:

- Poll task from the taskQueue (Pseudo code 1-14):

  taskQueue is a structure shared by multiple threads so the threads have to access it in a synschronized code. After gaining the lock on the taskQueue the thread polls a task.

- Decide whether to materialize (Pseudo code 4-13):
The thread decides whether to materialize the next level of triples based on the values of `iteratorThreshold` and `taskThreshold`. First a check is made to see if the size of the `taskqueue` has fallen below the `taskThreshold`. If so, the second check is made on the `iteratorThreshold` being higher than the current level up till which the task is materialized.

- Materialize if required (Pseudo code 5-11):

  If materialization is required, all the tasks are generated by applying the iterator based on the task. Then the generated tasks are added to the `taskQueue`

- Poll a task and explore the solution in a DFS manner (Pseudo code 15-17):

  This step is only performed if there was materialization. In case of materialization, the task picked up earlier is materialized. So the thread polls another task from the `taskQueue` after adding the materialized tasks to the `taskQueue`.

### 4.5 Segmenting the `taskQueue`

Passing one task to the Matcher at a time can lead to rapid thread context switching if the task is quickly executed. Consequently, we have also created a modification of the above approach to minimize the effect of rapid thread context switching. In the modified approach, each Matcher thread has its own local `taskQueue`. Instead of assigning a single task to a Matcher thread, we assign it a group of tasks, which we call a segment. A segment is copied into the local task queue of a Matcher. Each Matcher, once it finishes exploring a task, checks its local `taskQueue` for the next task. Once the local `taskQueue` is exhausted, the Matcher obtains the next segment of tasks from the main `taskQueue`. If the number of tasks in the main `taskQueue` is less
than the segment size, the Matcher thread will try to materialize the first task in the
\texttt{taskQueue}. If this task has reached its \texttt{iteratorThreshold}, no materialization takes
place and the Matcher thread obtains the remaining tasks on the \texttt{taskQueue}.

For experimental purpose, we pass in a parameter (\texttt{segment-multiplier}) which affects the
number of segments that the materialized \texttt{taskQueue} is divided into. The idea is to produce
\texttt{segment-multiplier} times the number of Matchers segments, before the Matchers are
spawned. The formulas in listing 3 explain the use of this parameter.

\begin{center}
\begin{minipage}{0.9\textwidth}
\begin{align*}
\text{Number of segments} &= \text{Number of Matchers } \times \text{segment-multiplier} \\
\text{Size of a segment} &= \text{total number of tasks produced after first iterator is} \\
& \quad \text{materialized} / (\text{Number of segments})
\end{align*}
\end{minipage}
\end{center}

Listing 3: Segmentation Formulas

If after the first iteration, the number of tasks produced is less than the number of
segments, more of the tasks from the \texttt{taskQueue} are materialized till the size of the\texttt{taskqueue} is equal to the number of segments. In case the value for the
\texttt{iteratorThreshold} is reached with insufficient number of tasks in the \texttt{taskQueue}, the
segment size is considered to be 1. We call this variant approach as the segmentation approach.
The timing comparison of the segmentation approach to the original approach is presented later
in the thesis.
CHAPTER 5

IMPLEMENTATION

In this chapter, the technology used for the implementation of our algorithm is covered briefly.

5.1 Jena

Once the RDF data is generated for the experiments, there is a need to be able to access that data programmatically. A triple store is used to load the data so that it can be queried. The Jena Semantic Web toolkit [47] for the Java platform is used as the triple store. Jena is open source software released under a BSD license. Jena can use both in memory and persistent storage to store the RDF data. Our choice for Jena as a triple store is guided by the fact that it is one of the best available triple stores for in memory models [48] considering the time to load, configure, display and browse the RDF data. Jena also provides a Java API to create and manipulate RDF graphs. Jena has object classes to represent graphs, resources, properties and literals. The interfaces representing resources, properties and literals are called Resource, Property and Literal respectively. SPARQL support in Jena is currently available via a module called ARQ. SPARQL queries can be executed either from the command line or by using the Jena Java API. We use the Jena API to evaluate our parallel algorithm. In the Jena API, the RDF graph is represented by the Model interface. We can use the methods from the ModelFactory to create memory models. The Model interface is used to read in the RDF XML files generated by the UBA generator and create an in memory Jena Model.
Each triple in Jena is represented as a Statement. The complete RDF model is represented as a series of these statements. Each statement, like a triple has a subject, property and an object. The Statement interface provides methods to access each of these. The Model interface defines methods to get iterators over statements. These iterators are subclasses of the Java Iterator. The iterator has methods to list the statements it iterates over.

A basic query pattern can be expressed with a Selector. The listStatements method returns an iterator which satisfies the pattern expressed by the Selector. Jena API operations are not thread safe by default. The application we design takes care of all the problems which might arise due to multithreading.

5.2 Java Threads

At the core of our implementation is an algorithm which is responsible for the parallel execution of the SPARQL SELECT queries. We implement the approach using the Java programming language. Java supports concurrent programming through the API’s in the Java.util.concurrent package.

The main units of concurrent programming are threads and processes. Each process has its own memory space. A thread is like a light weight process. A process can have one or more threads. All threads belonging to a particular process, share its resources. A computer system at any given time has multiple active threads and processes. Time slicing refers to an OS feature where execution time is divided time between multiple threads and processes. On a single core processor practically all the time, there are multiple active threads taking turns to share the running time on a single processor. Concurrency is not to be confused with the actual parallel execution. Concurrency just gives a notion that multiple tasks are executing simultaneously.
Concurrency is even possible on single core processors in a case where some threads may be doing input/output operations while one uses the CPU.

On the latest multi-core machines parallel execution is possible however, multiple threads accessing the same resources can lead to some problems [49]. The Java solution to these problems is thread synchronization. Synchronization guarantees that only one thread can at a time, execute the synchronized code on a single object. In this way the threads can safely access the shared resources.

Our implementation is run on a Linux server with Java 1.6. After Java 2 Release 1.3, Linux system threads are used to implement Java threads [50]. With this addition, the multi threaded Java applications can actually take advantage of adding more processing units to the system. The JVM decides the way the Java threads are mapped to the OS threads on a multi-processor system. Some earlier JVM’s used green threads model in which many Java threads are mapped to only one OS level thread. Later versions of the JVM’s use native threads which may run as separate OS threads on a multi-processor system.

Thus, if the problem can be divided into independent parts, the Java threads can work on the independent parts in parallel, each thread scheduled as a separate Linux system thread. This leads to true parallelism in the application.

5.3 Exploiting Parallelism

Running a SPARQL query can be considered as the following steps in order: Query parsing, Query rewriting, Query execution. The query rewriting is done to generate a good Query Execution Plan. In a SPARQL query each variable in multiple triple patterns of the query can be considered as a join. In a good Query Plan the triple patterns are ordered in a way to aim for
optimized results. In the current work, we assume that we are given a good query plan i.e. join order of the triple patterns is fixed. The aim of the algorithm is to take this plan and improve its execution time by taking advantage of the parallelism.

Jena is used to frame different queries (APPENDIX A). The entire query is framed as a sequence of iterators. The order of these iterators is assumed to be optimal (i.e. resulting in minimum intermediate results). Each iterator takes in a statement applies some operations and produces a series of output statements. These statements might be an input to some other iterator in the query plan. The entire tree is explored in a Depth-first fashion. Exploring the tree depth first is an efficient with regards to the memory requirements. A solution is generated if its value can be bound for all the iterators in the query plan.

The query execution is divided into the following parts:

- **Query setup:**

  In this step, the query is taken as a user input and a query plan is generated.

- **Materializing the first iterator:**

  In this step, a Jena selector is generated which matches the first triple pattern of the Query plan. This Jena selector is used to iterate over all the statements which match the selector. These statements are stored as tasks in a queue. This process of generating and storing the statements is called materializing.

- **Spawning threads to work on the statements produced by the first iterator.**

- **Main thread waiting for the Matchers to finish all the work in the common queue.**

  In Jena, a triple is called a statement. Materializing the first iterator implies finding all the statements which are accessible using the first iterator. These statements are all possible part of
solutions to the query. Thus, this statement is like a work unit for which all the remaining
iterators need to be explored. This is where the parallelism comes into the picture. Each of these
statements can be processed independently by different threads. All these threads run in parallel
exploiting the multiple execution threads that the modern high performance machines have to
offer.
CHAPTER 6

EXPERIMENTS

6.1 Hardware

We have conducted our experiments on a Intel(R) Xeon(R) CPU E5530 @ 2.40GHz server with 32 GB of main memory, 2 processors with 4 cores each. Cores are physical blocks of logic in the processor which are capable of running applications. Each of the cores is hyper-threaded, thus giving a total of 16 live execution threads. The system runs on a 64 bit version of Centos 5 operating system (a version of RedHat Enterprise Linux OS).

6.2 Experimental Details

One of the problems faced is that of the increased objects on the heap and the program running out of memory. We had to increase the Java Heap space because of the large number of objects being produced in the materialization stage. J2SE 5 release introduced the concept of ergonomics, where the default values for heap size, garbage collector and the JVM HotSpot (client or server) are chosen based on the machine configuration [51]. Any machine with 2 or more physical processors and 2 GB of memory or greater is defined as a server machine. A server class machine has a default initial memory size of $\frac{1}{64}$ Size of physical memory (minimum 32 MB) and a maximum of $\frac{1}{4}$ Size of physical memory (maximum 1 GB). By default the garbage collector in the server HotSpot is the parallel collector.
All the experiments were run with the help of a build file and Linux shell scripts. Each experiment was run 10 times and the average results were reported. This is done to nullify the impact of the outliers.

6.3 Queries

The SPARQL SELECT query is the basic and the most frequently used SPARQL query. It uses the graph pattern, also used in other query types, which is the main focus of the work in this thesis. We can expect similar speedups when the algorithm is applied to other forms of queries.

6.4 Lehigh University Benchmark (LUBM)

The term benchmarking means running a set of programs to access their relative performance. Lehigh University Benchmark [52] (LUBM), is a system for benchmarking Semantic Web knowledge based systems. A list of available RDF benchmarks is available at the ESW Wiki Page about RDF Benchmarks [53]. The widely used LUBM is chosen over other systems like Berlin SPARQL Benchmark [54] and SP2 Bench [55] because of the high number of classes and properties involved in the ontology. LUBM uses Univ-Bench ontology [56], which is an ontology from the university domain. The ontology contains 43 classes and 32 properties depicting the functioning of a university. Queries that take substantial execution time are needed in order to note the improvement in execution time due to the parallel execution. The high number of classes and properties enable us to pose complex queries in SPARQL. LUBM provides us with the advantage to query over a large dataset that commits to a single ontology. It comes with a synthetic data generator (UBA), which generates random and repeatable data of varied sizes. Both the Univ-Bench ontology and the data are in the OWL Lite [57] sublanguage.
Using the data generator, the user is given the option to:

Specify seed for random number generation.

The number of universities.

The starting index of the universities.

By exercising the above options, the user can specify which and how many universities to generate. We need growing data sizes to measure the scalability of our algorithm. The size of the data grows as we increase the number of universities. A graph depicting the sizes corresponding to the university numbers is shown in Figure 4.

![University data size](image)

**Figure 4 - LUBM dataset sizes**

The data generated is in the OWL format. Each university contains 15 to 25 departments, each represented as a separate OWL file. For this implementation all OWL files are converted into
RDF files and then merged into a single RDF file. This complete file is easier to specify in the SPARQL query rather than listing all the files individually.

### 6.5 Timing Mechanism

The time after the data load is considered while measuring the speedup. This represents the time actually spent in the query execution.

Measuring time accurately is of prime importance in a study of efficiency. Wall clock was the only way to measure time prior to Java 5. Wall clock can be quite inaccurate as it also includes the time spent by the processor on the processes the application does not care about. Java 5 has introduced the `java.lang.management` package that provides methods which can measure the CPU and user time per thread. This package is ideal for benchmarking as the time provided is not affected by other system activity. In spite of the advantages of the above package, some inherent problems in the nature of our problem and missing functions from the API prevent us from making use of the package. Timing the complete application can be divided into the following parts:

- Loading the data.
- Time for first iteration (Constant part)
- Time spend by the Supervisor waiting for the Matchers to finish

The CPU time for each of the matchers and the supervisor can be computed individually with the above API. In an ideal situation all the matcher threads would be scheduled on different processors. In such a case, the application could have been timed by adding the CPU time for the Supervisor and maximum of the time taken by the matcher threads. However, the real scenario
might be different and not all matcher threads would have processing units all the time since the time they are spawned. Thus for timing the implementation we use the conventional

System.currentTimeMillis() after the data is loaded and just after the Matcher threads die. The difference of the two timings gives the time taken by the implementation to run. To minimize the external activity we disable all other user level processes.
CHAPTER 7

EVALUATION OF RESULTS

7.1 Evaluations

For the evaluation of our parallel algorithm we use Jena’s in-memory storage as a triple store. There might be some concerns about the memory limiting the size of the RDF data that might be processed. The problem with the limited memory can be solved if run on a clustered environment with distributed in-memory model. Another reason for choosing the in-memory model is that optimized queries run much faster on in-memory models as compared to on-disk models [58]. Query execution time is highly impacted by the number of joins in the query. We run experiments with three different queries with increasing number of joins to test the approach against varying loads of query complexity. For each query, experiments have been performed for data varying in size from 332MB to 670MB.

Like most programming problems, our query execution algorithm can be divided into a parallel and a serial execution part. It is the sequential part which is the bottleneck to the overall performance of the approach. There are following sequential parts to our approach:

- The Supervisor materializing the first iterator:
  This is a fairly quick step as checked with the wall clock time in Java.

- The Matcher threads accessing the work buffer to get work or to materialize:
  The time for which a particular thread was blocked or the time spent in waiting for a structure can be calculated with the Java class ThreadInfo in the `java.lang.management`
package. The time for which a thread waited can be calculated in up to milli second accuracy.

Our results show that the threads are rarely blocked. If they are they are blocked for a time of 0 millisecond. This proves that accessing of the common buffer by the threads is not a bottleneck as common sense might assume.

Also another part consuming the time might be the time taken by the JVM thread management. More research in the performance tuning of this approach might reveal some other avenues to explore for exploiting parallelism in SPARQL queries.

The graphs indicate the performance of our approach against three queries of increasing complexity (APPENDIX A). For each query we draw a set of graphs for a particular data-load. For each data load, for a particular query, we plot the execution time as the number of processing threads is increased. The same algorithm with one thread is considered as the sequential version to generate accurate comparisons.

**Query 1: 2 Joins**

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?stu ?course ?uni
WHERE
{
  ?stu ub:takesCourse ?coursetaken.
  ?stu ub:undergraduateDegreeFrom ?uni.
}
```
332 MB (3968864 triples)

338 MB (4643866 triples)

444 MB (5307458 triples)
501 MB (5988689 triples)

557 MB (6654562 triples)

612 MB (7319990 triples)
670 MB (8003821 triples)
Query 2: 4 Joins

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>

SELECT ?stu ?course ?uni
WHERE
{
?stu ub:undergraduateDegreeFrom ?uni.
?stu ub:advisor ?prof1
}
332MB (3968864 triples)

338 MB (4643866 triples)

444 MB (5307458 triples)
501 MB (5988689 triples)

557 MB (6654562 triples)

612 MB (7319990 triples)
670 MB (8003821 triples)
Query 3: 12 Joins

```sparql
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?stu ?course ?uni
WHERE
{
?stu ub:undergraduateDegreeFrom ?uni.
?prof1 ub:worksFor ?uni.
?prof1 ub:researchInterest ?research.
?prof1 ub:headOf ?dep.
?prof1 ub:undergraduateDegreeFrom ?uni3.
}
```
332 MB (3968864 triples)

338 MB (4643866 triples)

444 MB (5307458 triples)
501 MB (5988689 triples)

557 MB (6654562 triples)

612 MB (7319990 triples)
Comparison of the Segmentation approach

For comparing the Segmentation variation with the above approach we time the experiments varying the `segmentation-multiplier` from 4 to 12 in increment of 2. Each of the tables below show a comparison of the two approaches when ran under similar parameters: `iteratorThreshold, taskThreshold, number of threads` and varying the `segmentation-multiplier`. The experiments are run on different queries and data sets to try a variety of combinations.
Query 1 on dataset size 332 MB

Timing for the original approach: 931.9 ms

<table>
<thead>
<tr>
<th>Segmentation-multiplier</th>
<th>Segmentation approach (Time in milisecond)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>873.1</td>
</tr>
<tr>
<td>6</td>
<td>914</td>
</tr>
<tr>
<td>8</td>
<td>919.7</td>
</tr>
<tr>
<td>10</td>
<td>857.5</td>
</tr>
<tr>
<td>12</td>
<td>1295.6</td>
</tr>
</tbody>
</table>

Query 2 on dataset size 444 MB

Timing for the original approach: 1042.8 ms

<table>
<thead>
<tr>
<th>Segmentation-multiplier</th>
<th>Segmentation approach (Time in milisecond)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1119.1</td>
</tr>
<tr>
<td>6</td>
<td>1161</td>
</tr>
<tr>
<td>8</td>
<td>1073</td>
</tr>
<tr>
<td>10</td>
<td>1103.9</td>
</tr>
<tr>
<td>12</td>
<td>1046.5</td>
</tr>
</tbody>
</table>
Query 3 on dataset size 557 MB

Timing for the original approach: 1520.2 ms

<table>
<thead>
<tr>
<th>Segmentation-multiplier</th>
<th>Segmentation approach (Time in milisecond)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1523.3</td>
</tr>
<tr>
<td>6</td>
<td>1501.5</td>
</tr>
<tr>
<td>8</td>
<td>1502.1</td>
</tr>
<tr>
<td>10</td>
<td>1433.9</td>
</tr>
<tr>
<td>12</td>
<td>1559.7</td>
</tr>
</tbody>
</table>

In an ideal situation, there should be a linear speed up in the processing time as we increase the number of processing units. Very few approaches can actually achieve that kind of speed up due to some non parallelizable part of the computation. We can achieve almost linear speed up until a constant value for the number of processing units. This number increases with the increase in the data size. This is because each thread has more data to go through in order to search for the complete solution, thus implying less time waiting for access to the synchronized data structure.

Also more complex queries see better speedups in most cases. This is a proof that the approach performs better if the data to be searched is bigger and also when the query is more complex. This conclusion addresses the main motivation for this thesis: response time of SPARQL queries against growing data sets and query complexity.

Another point to note is that no significant speedups are observed after the number of Matcher threads reaches a constant value. This constant value is close to the number of physical cores.
present on the machine. The reason for this observation may be the fact that having hyper-threaded cores might not be as effective as having same number of physical cores [59]. As the threads on the hyper-threaded core share the cache this might limit the performance as time is spent in context switching and fetching the data needed in the cache. In many cases the threads might need different data which leads to flushing of the cache with each context switch. This leads to thrashing of the cache and limits performance gains due to hyper-threading.

We can also observe that there is no major improvement with the Segmentation approach. In some cases there is a minor improvement while in some cases the time taken is more than the original approach. One of the reasons of this non-consistent performance could be the extra materialization step that is done by the Supervisor in case that the number of tasks produced after the first iteration is lesser than the number of segments. The low access times for the taskQueue might also be one of the reasons that no major improvements are observed with the segmentation approach. Additional materialization is done by the Matcher when it obtains a segment from a shrinking taskQueue.
CHAPTER 8

SUMMARY

8.1 Conclusions

With each passing day, the Semantic Web vision moves closer towards becoming a reality. The interest within the research community as well as the industry continues to add to the data in the RDF format. The main current approaches to querying Semantic Web data still use a single RDF ontology composed of data from distributed sites. SPARQL is the standard query language for RDF data. The response time of SPARQL queries continues to increase as the data size and the complexity of queries increases. The usefulness of any data format depends on the effectiveness of its query mechanisms.

This thesis contributes to one of the least explored methods to SPARQL Query optimization, namely parallel execution. We have developed and implemented an iterator based algorithm to execute SPARQL queries in parallel, harnessing the power of modern multi-core machines. The approach implemented primarily speeds up the graphs pattern matching which is used in all SPARQL query constructs. Thus a similar implementation could be useful in speeding up queries involving other SPARQL constructs.

The evaluation section highlights the speedup generated by using our parallel evaluation approach. Graphs generated indicate the effect of increasing the processing units on the overall execution time of the queries. As observed, good speedups are generated until a constant value of the number of processing units. This constant value is close to the actual number of cores present
on the machine, signaling that hyper-threading might not be as effective as real cores. Highly independent tasks are generated as the complexity of the queries and the data size increases. This leads to a better performance of the approach in such cases.

While this work is not a comprehensive study of the parallelism in SPARQL queries, it definitely is one of the first in-depth discussions of such a possibility.

8.2 Future work

For materializing the first iterator we can make use of the multi-processors if we can divide the input data into chunks. Each of the threads then acts upon a chunk independently. After materializing the first iterator the results from different threads might be merged. As more benchmarks are released we may run the algorithm against some more complex queries.

Altering the heuristics to control the common work data structure might lead to a study of its own. The study of these heuristics might throw some light into setting of these parameters for optimal performance. Relationship between the type of the query and the optimal parameter settings can be explored.

The synchronization on the common data structure causes some sequential execution and might be done away with. This is possible if some efficient splitting of the data structure can be done. The approach can be envisioned as division of the data structure into buckets, each bucket belonging to a thread. Optimal performance can be achieved if each of the threads finishes processing their buckets in almost same time.

The evaluation of this algorithm on an on-disk memory store remains to be studied. SPARQL queries are converted to SQL and then run against the RDF data stored in the on-disk
stores. SQL query optimization is an extremely mature field as compared to SPARQL queries. It might be interesting to study how this algorithm performs on a on-disk memory model.

In the current work we use the basic SELECT construct for the SPARQL language for framing the test queries. The query parser can be enhanced so that experiments can be run against other forms of SPARQL queries. The impact of the FILTER clause can bring about some change in the experiment timings. The FILTER clause can be applied as soon as the bindings for that variable are produced. This will limit the search space for iterators which still need to be calculated based on intermediate triples.
REFERENCES


[57] "OWL Lite." Available: [http://www.w3.org/TR/owl-ref/#OWLLite](http://www.w3.org/TR/owl-ref/#OWLLite).


APPENDICES

APPENDIX A: Evaluation Queries

Query 1:

Query 2:
Query 3:
APPENDIX B: Sample RDF/XML

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<rdf:RDF
 xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
 xmlns:owl="http://www.w3.org/2002/07/owl#"
 xmlns:ub="http://www.lehigh.edu/%7Ezhp2/2004/0401/univ-bench.owl#">
  <ub:FullProfessor
    rdf:about="http://www.Department0.University0.edu/FullProfessor0">
    <ub:name>FullProfessor0</ub:name>
    <ub:teacherOf rdf:resource="http://www.Department0.University0.edu/Course0" />
    <ub:teacherOf rdf:resource="http://www.Department0.University0.edu/GraduateCourse0" />
    <ub:teacherOf rdf:resource="http://www.Department0.University0.edu/GraduateCourse1" />
    <ub:undergraduateDegreeFrom>
      <ub:University rdf:about="http://www.University84.edu" />
    </ub:undergraduateDegreeFrom>
    <ub:mastersDegreeFrom>
      <ub:University rdf:about="http://www.University875.edu" />
    </ub:mastersDegreeFrom>
    <ub:doctoralDegreeFrom>
    </ub:doctoralDegreeFrom>
    <ub:worksFor rdf:resource="http://www.Department0.University0.edu" />
    <ub:emailAddress>FullProfessor0@Department0.University0.edu</ub:emailAddress>
    <ub:telephone>xxx-xxx-xxxx</ub:telephone>
    <ub:researchInterest>Research20</ub:researchInterest>
  </ub:FullProfessor>

  <ub:FullProfessor
    rdf:about="http://www.Department0.University0.edu/FullProfessor1">
    <ub:teacherOf rdf:resource="http://www.Department0.University0.edu/GraduateCourse2" />
  </ub:FullProfessor>
</rdf:RDF>
```