NoSQL HPC Ontology Reasoner using Task Parallel Library

HPC Graduate Project

Written and Submitted by:
Altaf Hussain
Student Number: 201104026
WebFX ID: X2011bep

This document contains the project report of Parallel Computing Graduate Project.
1. **Overview and Abstract**

Ontologies, and ontology based expressions, are becoming increasingly important. They provide a common vocabulary together with computer accessible descriptions of the meaning of relevant terms through relationships with other terms. For instance, in an ontology describing a medical system can describe human, roles, disease, medication and their relationships. Ontologies play a major role in the Semantic Web and in e-Science where they are widely used in, e.g., bio-informatics, medical terminologies and other knowledge management applications. One of the most important aspects of ontologies is that they contain knowledge structured in a special way. The users of ontologies are typically interested in obtaining information about relationships between concepts described in ontologies and querying the ontologies. Both tasks require reasoning tools that can derive new knowledge from the knowledge explicitly stated in ontologies. Ontology classification—computing the subsumption relationships between classes is one of the foundational reasoning tasks provided by many reasoners. However, in cases many reasoners fails or show poor performance when ontology model becomes very large. For some existing medical ontologies, the models are so big that they do not fit into the main memory of a computer. Here, I am proposing an approach of ontology classification by high performance computing approach using many core machines and state of the art document based database system. Task Parallel Library in .NET provides a way to leverage cores in many core machines as separate executing units to run task in parallel. The document based database is a lightweight alternative of RDMS can significantly reduce database operations for its schema less approach. The preliminary performance evaluation showed this approach (NoSqlReasoner) provides better results over most renowned reasoners.

2. **Introduction**

Ontologies are formal languages of terms describing specific subjects like human body parts, genes, or animal species. The terms in ontologies are "defined" by means of relationships with other terms of the ontology using ontology languages. Ontology languages based on Description Logics (DLs) [1], such as OWL 1 are becoming increasingly popular among ontology developers thanks to the availability of ontology reasoners, which provide automated support for visualization, debugging, and querying of ontologies. Classification is a central reasoning service provided by ontology reasoners. The goal of classification is to compute a hierarchical relation between classes. The class hierarchy is used to browse ontologies in ontology editors. Most existing ontology reasoners do not derive logical consequences of ontological axioms explicitly, but instead they check whether it is possible to construct a model of the ontology where the target consequence does not hold, e.g., they try to construct a situation where
[Heart] would be a part of the [Circulatory System] but not a part of the [Muscular System]. If such a situation is not possible, then it is concluded that the target consequence follows from the axioms in the ontology. One problem with this technique is that when ontology expresses long and possibly cyclic dependencies between terms, e.g., [Heart] is a part of [Circulatory System] which has a part [Lung] which is a part of [Respiratory System] which has a part [Trachea], etc., and then the reasoner has to construct very large models. For some existing medical ontologies, the models are so big that they do not fit into the main memory of a computer.

With the explosion of Linked Data\(^1\), communities are making efforts to develop formal ontologies for annotating their databases and are publishing these databases as RDF triples. Examples of this are biopax\(^2\) in the field of Life Science and LinkedGeoData\(^3\) in the field of Geographic Information Systems. This means that formal ontologies with a large number (billions of) instances are now available. In order to manage these ontologies, current platforms need a scalable, high-performance repository offering both light and heavy-weight reasoning capabilities. The majority of current ontologies are expressed in the well-known Web Ontology Language (OWL) that is based on a family of logical formalisms called Description Logic (DL). Managing large amounts of OWL data, including query answering and reasoning, is a challenging technical prospect, but one which is increasingly needed in numerous real-world application domains from Health Care and Life Sciences to Finance and Government.

Another problem is that the ontology may potentially have a large number of different models, each of which must be independently explored by the reasoner. Ontology languages provide for constructors called 'number restrictions', which result in a particularly large number of models. These limitations of model-building reasoners, therefore, pose a serious problem for the development of large medical and bio-chemical ontologies -- without efficient reasoning tools, for example, the users of such ontologies may not be able to obtain the information that they are interested in.

In this paper, I am presenting an implementation of efficient Ontology classification leveraging high performance computing technique by distributing tasks into cores using Task Parallel Library (TPL) in .NET \(^2\) and Parallel Linq \(^3\). In addition, efforts of creating modern web scale databases resulting development of No-Sql (Not-Only SQL) database supporting Horizontal scalability and faster operations over traditional RDMS. Combining both of the approach, NoSql database and TPL am investigating an alternate approach which could provide competitive performance classifying large ontologies.

The reminder of the paper organized as follows. In section 3 related works is discussed. Section
4 and 5 discusses document based database and task parallel library. Project implementation details are given in section 6. Section 7 describes the performance evaluation and we concluded in section 8.

1http://linkeddata.org/
2http://www.biopax.org/
3http://linkedgeodata.org/About

3. RELATED WORKS

Ontology classification — computing the subsumption relationships between classes is one of the foundational reasoning tasks provided by many reasoners. The task of ontology classification is the foundation for other ontology reasoning problems. Ontology classification computes the subsumption relation between all pairs of concepts and then organizes the concepts into a subsumption hierarchy. The goal of ontology classification is to find:

1. Subsumption or Subclass relationships
2. Transitive closure of the classes
3. Transitive closure of the individuals (ABOX) depending on class relationships
4. Transitive closure of the Properties depending on property relationships depending sub property axioms.

In ontology, class (TBOX) and property relations may not be explicit. These relations can be expressed as chain of relationship axioms. The axioms indicate which class or property is a sub/super class or property respectively. This criteria resulting computing intensive exhaustive processing for relationships chains. Also, as ontologies can be large, caching already found relationship chains need large memory resource or materialization in database.

The current description logic (DL) reasoners may either require unacceptably high resources or produce incomplete results when classifying large and complex ontologies with expressive DL languages such as ALCHOI. Tableau-based and consequence-based reasoners are two dominant types of reasoners that provide the ontology classification service. Tableau-based reasoners, such as HermiT [23], Fact ++ [24] and Pellet [25], try to build counter-models $A\cup B$ for candidate subsumption relations, based on sound and complete calculi such as [25] and [23]. These reasoners are able to classify ontologies in expressive DLs like SROIQ (D).

Most of the currently-available ontology reasoners are based on so-called model building procedures such as tableau [25] and hyper-tableau [23] calculi. Such procedures classify an
input ontology, in general, by iterating over all necessary pairs of classes, and trying to build a model of the ontology that violates the subsumption relation between them. Recent investigation of tractable DLs led to the discovery of another type of reasoning procedure \cite{26}. Where, instead of enumerating pairs of classes and building counter-models, the procedure derives subclass relationships explicitly using special inference rules. The advantage of this method is that subsumption relations are computed “all at once” in a goal-directed way without costly enumerations.

In Consequent based reasoning \cite{28} showed a better performance for a tractable fragment of OWL-DL. In \cite{29} Weihong et al. presented a hybrid technique using both tableau and consequence based approach and practically showed it provide good performance then state of the art reasoner like HermiT \cite{23}, Fact ++ \cite{24} and Pellet \cite{25}. For ALOCHOI ontologies most of test cases found similar or faster results expect one.

Using a completely different approach called Materialization \cite{30} of inferring and storing implicit knowledge from ontology. This relatively simple approach of materialization used in many reasoners including \cite{31}, \cite{32} and \cite{33}. These approaches used RDMS and Sequential program execution models. In this project, I am proposing a materialization approach using document based database, MongoDB and parallel execution model through Task Parallel Library. It is already established in the community that, depending data model MongoDB can provide better scalability and faster CRUD operation which is discussed in Section 4. Also, parallel execution model, Task Parallel Library- leveraging many core machines can provide significant performance gain over sequential execution model discussed in Section 5. Through the aforementioned two level performance gains, we can achieve promising results. In performance evaluation section a performance comparison table shows that this approach shows promising result to further investigate the technique.

\[^{4} \text{http://www.w3.org/2004/OWL/} \]

4. **NoSQL Databases**

Document-oriented database evolved for storing, retrieving, and managing document-oriented, or semi-structured data and information. Document-oriented databases are one of the main categories of so-called NoSQL (Not Only SQL) databases and the popularity of the term "document-oriented database" (or "document store") has grown with the use of the term NoSQL itself. In contrast to well-known Relational databases and their notions of
"Relations", these systems are designed around an abstract notion of a "Document".

Relational database management systems (RDBMSs) today are the predominant technology for storing structured data in web and business applications. Since Codd's paper “A relational model of data for large shared data banks” [4] from 1970 these data stores relying on the relational calculus and providing comprehensive ad hoc querying facilities by SQL (cf. [5]) have been widely adopted and are often thought of as the only alternative for data storage accessible by multiple clients in a consistent way. Although there have been different approaches over the years such as object databases or XML stores these technologies have never gained the same adoption and market share as RDBMSs. Rather, these alternatives have either been absorbed by relational database management systems that e.g. allow to store XML and use it for purposes like text indexing or they have become niche products for e.g. OLAP or stream processing.

4.1 Motives and Drives of NoSQL Database

The term NoSQL was first used in 1998 for a relational database that omitted the use of SQL [6]. The term was picked up again in 2009 and used for conferences of advocates of non-relational databases such as Last.fm developer Jon Oskarsson, who organized the NoSQL meetup in San Francisco [7]. A blogger, often referred to as having made the term popular is Rackspace employee Eric Evans who later described the ambition of the NoSQL movement as “the whole point of seeking alternatives is that you need to solve a problem that relational databases are a bad fit for” (cf. [Eva09b]). This section will discuss rationales of practitioners for developing and using nonrelational databases and display theoretical work in this field. Furthermore, it will treat the origins and main drivers of the NoSQL movement.

The Computerworld magazine reports in an article about the NoSQL meet-up in San Francisco that “NoSQLers came to share how they had overthrown the tyranny of slow, expensive relational databases in favor of more efficient and cheaper ways of managing data.” [7]. It states that especially Web 2.0 startups have begun their business without Oracle and even without MySQL which formerly was popular among startups. Instead, they built their own datastores influenced by Amazon’s Dynamo [9] and Google’s Bigtable [10] in order to store and process huge amounts of data like they appear e.g. in social community or cloud computing applications; meanwhile, most of these datastores became open source software. For example, Cassandra originally developed for a new search feature by Facebook is now part of the Apache Software Project. According to engineer Avinash Lakshman, it is able to write 2500 times faster into a 50 gigabytes large database than MySQL [11].

The Computerworld article summarizes reasons commonly given to develop and use NoSQL
Avoidance of Unneeded Complexity Relational databases provides a variety of features and strict data consistency. But this rich feature set and the ACID properties implemented by RDBMSs might be more than necessary for particular applications and use cases. As an example, Adobe’s ConnectNow holds three copies of user session data; these replicas do not neither has to undergo all consistency checks of a relational database management systems nor do they have to be persisted. Hence, it is fully sufficient to hold them in memory [8].

**4.2 MongoDB from 10Gen**

MongoDB is one of the leading document based database. According to 10gen CTO Eliot Horowitz [12] -

"MongoDB wasn’t designed in a lab. We built MongoDB from our own experiences building large scale, high availability, robust systems. We didn’t start from scratch; we really tried to figure out what was broken, and tackle that. So the way I think about MongoDB is that if you take MySQL, and change the data model from relational to document based, you get a lot of great features: embedded docs for speed, manageability, agile development with schema-less databases, easier horizontal scalability because joins aren’t as important. There are lots of things that work great in relational databases: indexes, dynamic queries and updates to name a few, and we haven’t changed much there. For example, the way you design your indexes in MongoDB should be exactly the way you do it in MySQL or Oracle, you just have the option of indexing an embedded field."

**4.3 Why MongoDB?**

MongoDB is well known and a good choice among document oriented database for the following features:

- **Document-oriented**
  - Documents (objects) map nicely to programming language data types
  - Embedded documents and arrays reduce need for joins
  - Dynamically-typed (schemaless) for easy schema evolution
  - No joins and no multi-document transactions for high performance and easy scalability
- **High performance**
• No joins and embedding makes reads and writes fast
• Indexes including indexing of keys from embedded documents and arrays
• Optional streaming writes (no acknowledgements)

• **High availability**
  • Replicated servers with automatic master failover

• **Easy scalability**
  • Automatic sharding (auto-partitioning of data across servers)
    • Reads and writes are distributed over shards
    • No joins or multi-document transactions make distributed queries easy and fast
  • Eventually-consistent reads can be distributed over replicated servers

• **Rich query language**

4.4 **MongoDB Data Model:**
• A Mongo system (see deployment above) holds a set of databases
• A database holds a set of collections
• A collection holds a set of documents
• A document is a set of fields
• A field is a key-value pair
• A key is a name (string)
• A value is a
  • basic type like string, integer, float, timestamp, binary, etc.,
  • a document, or
  • an array of values

4.5 **MongoDB Philosophy**

4.5.1 **Design Philosophy:**
New database technologies are needed to facilitate horizontal scaling of the data layer, easier development, and the ability to store order(s) of magnitude more data than was used in the past.

A non-relational approach is the best path to database solutions which scale horizontally to many machines.

It is unacceptable if these new technologies make writing applications harder. Writing code should be faster, easier, and more agile.

The document data model (JSON/BSON) is easy to code to, easy to manage (schemaless), and yields excellent performance by grouping relevant data together internally.

It is important to keep deep functionality to keep programming fast and simple. While some things must be left out, keep as much as possible – for example secondary indexes, unique key constraints, atomic operations, multi-document updates.

Database technology should run anywhere, being available both for running on your own servers or VMs, and also as a cloud pay-for-what-you-use service.

4.5.2 Focus:
According to [12], MongoDB focuses on four main things: flexibility, power, speed, and ease of use. To that end, it sometimes sacrifices things like fine grained control and tuning, overly powerful functionality like MVCC that require a lot of complicated code and logic in the application layer, and certain ACID features like multi-document transactions.

Flexibility
MongoDB stores data in JSON documents (which we serialize to BSON). JSON provides us a rich data model that seamlessly maps to native programming language types, and since its schema-less, makes it much easier to evolve your data model than with a system with enforced schemas such as a RDBMS.

**Power**

MongoDB provides a lot of the features of a traditional RDBMS such as secondary indexes, dynamic queries, sorting, rich updates, upserts (update if document exists, insert if it doesn't), and easy aggregation. This gives you the breadth of functionality that you are used to from an RDBMS, with the flexibility and scaling capability that the non-relational model allows.

**Speed/Scaling**

By keeping related data together in documents, queries can be much faster than in a relational database where related data is separated into multiple tables and then needs to be joined later. MongoDB also makes it easy to scale out your database. Autosharding allows you to scale your cluster linearly by adding more machines. It is possible to increase capacity without any downtime, which is very important on the web when load can increase suddenly and bringing down the website for extended maintenance can cost your business large amounts of revenue.

**Ease of use**

MongoDB works hard to be very easy to install, configure, maintain, and use. To this end, MongoDB provides few configuration options, and instead tries to automatically do the "right thing" whenever possible. This means that MongoDB works right out of the box, and you can dive right into developing your application, instead of spending a lot of time fine-tuning obscure database configurations.

**4.6 Performance Comparisons- MongoDB Vs RDBMS**

In [13] Michel Kennedy showed a simple performance comparison among RDBMS and MongoDB. The following picture depicts it:
5. **Task Parallel Library and Parallel LINQ**

According to Microsoft Developer Network Documentation (MSDN) [14], many personal computers and workstations have two or four cores (that is, CPUs) that enable multiple threads to be executed simultaneously. Computers in the near future are expected to have significantly more cores. To take advantage of the hardware of today and tomorrow, you can parallelize your code to distribute work across multiple processors. In the past, parallelization required low-level manipulation of threads and locks. Visual Studio 2010 and the .NET Framework 4 enhance support for parallel programming by providing a new runtime, new class library types, and new diagnostic tools. These features simplify parallel development so that developers can write efficient, fine-grained, and scalable parallel code in a natural idiom without having to work directly with threads or the thread pool. The following illustration provides a high-level overview of the parallel programming architecture in the .NET Framework 4.
The Task Parallel Library (TPL) is a set of public types and APIs in the System.Threading and System.Threading.Tasks namespaces in the .NET Framework 4. The purpose of the TPL is to make developers more productive by simplifying the process of adding parallelism and concurrency to applications. The TPL scales the degree of concurrency dynamically to most efficiently use all the processors that are available. In addition, the TPL handles the partitioning of the work, the scheduling of threads on the ThreadPool, cancellation support, state management, and other low-level details. By using TPL, developers can maximize the performance code while focusing on the work that the program is designed to accomplish.

Starting with the .NET Framework 4, the TPL is the preferred way to write multithreaded and parallel code. However, not all code is suitable for parallelization; for example, if a loop performs only a small amount of work on each iteration, or it doesn't run for many iterations, then the overhead of parallelization can cause the code to run more slowly. Furthermore, parallelization like any multithreaded code adds complexity to your program execution.

TPL supports Task Parallelism [15] and Data Parallelism [16] through parallel language integrated query (PLINQ) [17]. Task parallelism support unrelated task running on several threads. Task parallelism is not used largely in this project however, Data Parallelism used extensively in this project.

In an overview of TPL in CodeProject [22], Varun Manipal showed the following performance
comparison for Matrix Multiplication between Parallel Execution of TPL implementation and Sequential execution.

Figure 4: Matrix multiplication running on a Quad Core machine.

5.1. Data Parallelism through PLINQ

In [18], Language-Integrated Query (LINQ) was introduced in the .NET Framework version 3.0. It features a unified model for querying any System.Collections.IEnumerable or System.Collections.Generic.IEnumerable<T> data source in a type-safe manner. LINQ to Objects is the name for LINQ queries that are run against in-memory collections such as List<T> and arrays.

Parallel LINQ (PLINQ) is a parallel implementation of the LINQ pattern. A PLINQ query in many ways resembles a non-parallel LINQ to Objects query. PLINQ queries, just like sequential LINQ queries, operate on any in-memory IEnumerable or IEnumerable<T> data source, and have deferred execution, which means they do not begin executing until the query is enumerated. The primary difference is that PLINQ attempts to make full use of all the processors on the system. It does this by partitioning the data source into segments, and then executing the query on each segment on separate worker threads in parallel on multiple processors. In many cases, parallel execution means that the query runs significantly faster.

Through parallel execution, PLINQ can achieve significant performance improvements over legacy code for certain kinds of queries, often just by adding the AsParallelQuery operation to the data source. However, parallelism can introduce its own complexities, and not all query operations run faster in PLINQ. In fact, parallelization actually slows down certain queries. A certain level of experience can help developer to understand situation when PLINQ and Data Parallelism is useful to use.

5.2. Understanding and Factor to Impact PLINQ Performance

According to [18], the following section lists some of the most important factors that impact parallel query performance. These are general statements that by themselves are not sufficient
to predict query performance in all cases. As always, it is important to measure actual performance of specific queries on computers with a range of representative configurations and loads.

1. **Computational cost of the overall work.**
To achieve speedup, a PLINQ query must have enough delightfully parallel work to offset the overhead. The work can be expressed as the computational cost of each delegate multiplied by the number of elements in the source collection. Assuming that an operation can be parallelized, the more computationally expensive it is, the greater the opportunity for speedup. For example, if a function takes one millisecond to execute, a sequential query over 1000 elements will take one second to perform that operation, whereas a parallel query on a computer with four cores might take only 250 milliseconds. This yields a speedup of 750 milliseconds. If the function required one second to execute for each element, then the speedup would be 750 seconds. If the delegate is very expensive, then PLINQ might offer significant speedup with only a few items in the source collection. Conversely, small source collections with trivial delegates are generally not good candidates for PLINQ.

2. **The number of logical cores on the system (degree of parallelism).**
This point is an obvious corollary to the previous section, queries that are delightfully parallel run faster on machines with more cores because the work can be divided among more concurrent threads. The overall amount of speedup depends on what percentage of the overall work of the query is parallelizable. However, do not assume that all queries will run twice as fast on an eight core computer as a four core computer. When tuning queries for optimal performance, it is important to measure actual results on computers with various numbers of cores. This point is related to point #1: larger datasets are required to take advantage of greater computing resources.

3. **The number and kind of operations.**
PLINQ provides the AsOrdered operator for situations in which it is necessary to maintain the order of elements in the source sequence. There is a cost associated with ordering, but this cost is usually modest. GroupBy and Join operations likewise incur overhead. PLINQ performs best when it is allowed to process elements in the source collection in any order, and pass them to the next operator as soon as they are ready.

4. **The form of query execution.**
If you are storing the results of a query by calling ToArray or ToList, then the results from all parallel threads must be merged into the single data structure. This involves an unavoidable computational cost. Likewise, if you iterate the results by using a foreach (For Each in Visual Basic) loop, the results from the worker threads need to be serialized onto the enumerator thread.

5. **The type of merge options.**
PLINQ can be configured to either buffer its output, and produce it in chunks or all at once after
the entire result set is produced, or else to stream individual results as they are produced. The former result is decreased overall execution time and the latter results in decreased latency between yielded elements. While the merge options do not always have a major impact on overall query performance, they can impact perceived performance because they control how long a user must wait to see results.

6. The kind of partitioning.
In some cases, a PLINQ query over an indexable source collection may result in an unbalanced work load. When this occurs, you might be able to increase the query performance by creating a custom partitioner.

5.2. ParallelEnumerable Operators and Lambda Expression
In case of working with TPL, ParallelEnumerable and Lambda Expression are important. Operator AsParallel works as entry point for PLINQ and specifies that rest of query should be parallelized. Also ForAll is another multithreaded enumeration method that, unlike iterating over the results of the query, enables results to be processed in parallel without first merging back to the consumer thread which is a counterpart of ForEach that used in sequential enumeration. A detailed discussion on ParallelEnumerable operators can be found in [19]

![Diagram of ParallelEnumerable Operators](image)

5.2. 1. Lambda Expression
A lambda expression, introduced in .NET 3.0 is an anonymous function that you can use to create delegates or expression tree types. By using lambda expressions, you can write local
functions that can be passed as arguments or returned as the value of function calls. Lambda expressions are particularly helpful for writing LINQ query expressions. Lambda Expression is discussed in details as Expression Lambdas, Statement Lambdas, and with the Standard Query Operators in [20] and [21].

6. IMPLEMENTATION AND ONTOLOGY CLASSIFICATION

6.1. Implementation
The project developed in .NET 4.5 and designed in Service Oriented Architecture (SOA). The project architecture defined in a way which ensures it can easily be deployed and integrated with other feature and easy for continuous development. The n-tire layered fashion implementations ease the maintenance and development easy. The project architecture is shown in the following figure 6.

Figure 6: NoSqlReasoner Architecture
6.1.1 Project Description

The NoSqlReasoner project’s important parts details are given below:

- **Protégé**: Protégé is an ontology editor used to edit and create ontologies and traverse classes and properties. Protégé created ontology files are used as input.
- **NoSqlReasoner Frontend**: Contains User interfaces and interact with the user.
- **NoSqlReasoner Data Transfer Object**: These files contain object definition implementation defining class, property individual and responsible to refer objects in the domain in C#.
- **Helper and Business Classes**: These files provide domain rule support and conversion of data object to and from C#, DotNetRDF API and NoSql Classes.
- **Data Access Layer**: Contains classes' implement data access and insert, update and delete operation to and from MongoDB.

![Figure 7: Visual Studio Project structure](image)

6.2 Tools and Technology Used

- **C#.NET 4.5**: C#.NET 4.5 is used to develop the project. User Interface and logic classes are written in C#.
• Task Parallel Library, LINQ and Parallel Linq, Lambda Expressions:
  Task Parallel Library, Language Integrated Query, and Parallel Linq are used. TPL and PLinq are available in .NET 5. Documentation of these technologies can be found in Microsoft Developer Network (MSDN) documentation and sited references.

• MongoDB:
  MongoDB is used as a document based database. Installation guide can be found in [34].

• MongoDB C# Driver:
  To work with MongoDB and managing CRUD operation via C#, MongoDB C# driver is being used.

• APIs for Ontology Processing in .NET
  There are several API for working with ontologies in .NET. Among those, DOTNETRDF and LINQ2RDF are most rich. The DOTNETRDF API is used to process ontology with C# which reflects recent Technology and support. More details can be found at:  
  http://www.dotnetrdf.org/

• Visual Studio 2010:
  To work with MongoDB and managing CRUD operation via C#, MongoDB C# driver is being used.

• MongoVUE:
  MongoVUE is being used as MongoDB management console.

7. Performance Evaluation
The implemented NoSqlReasoner using proposed approach is compared with both the tableau based reasoners (HermiT, FaCT++, and Pallet) and also consequence based reasoners TrOwl and WSReasoner. In [29], [28] Weihong et al. provided a comparison table for large and small ontology. The full Galen\textsuperscript{5} and SNOMED\textsuperscript{6} Wine\textsuperscript{6} ontologies used as input ontology to classify. For Galen and Wine ontology results from [29] are used where experiments are conducted using an 8-core 2.00GHz CPU and 40GB RAM running 64-bit Ubuntu 10.04. SNOMED ontology experiments results mentioned in [28] and measured using a PC with a 2GHz IntelCore Duo processor and 1.5GB RAM operated by Linux v.2.6.27 with Java VM v.1.6.0. In this excrement NoSqlReasoner is measured in Windows 7 (64 Bit) laptop running on Intel Core i3 PC 2.27GHz
with 4GB ram.

<table>
<thead>
<tr>
<th>Ontology</th>
<th>Hermit (Sec)</th>
<th>Pallet (Sec)</th>
<th>FaCT++ (Sec)</th>
<th>TrOWL (Sec)</th>
<th>WSReasoner (Sec)</th>
<th>NoSqlReasoner (Sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wine</td>
<td>29.11</td>
<td>377.88</td>
<td>7.33</td>
<td>0.81</td>
<td>7.34</td>
<td>3.31</td>
</tr>
<tr>
<td>SNOMED</td>
<td>--</td>
<td>--</td>
<td>650.37</td>
<td>--</td>
<td>1185.70</td>
<td>758.58</td>
</tr>
<tr>
<td>Galen</td>
<td>115.10</td>
<td>133.25</td>
<td>465.35</td>
<td>--</td>
<td>7.59</td>
<td>5.18</td>
</tr>
</tbody>
</table>

Note: "--" entry means that the reasoner was unable to classify the ontology due to some problems.

Table 1: A comparison of classification time in seconds

The result of the experiment is summarized in Table 1. The time measured including materialization, classification and subclass closure. In the above experiment we can see that the strategy appears to be efficient in cases when number of classes in the ontology is medium and large.

8. Future Work and Conclusion

The implemented NoSql reasoner can be extended and can be used in following cases:

1) A SPARQL query interface can be provided which can accept queries from user about classifications and provide result after reasoning. To be able to do that, a SPARQL to
NoSQL conversion is needed as MongoDB doesn’t work with SPARQL or SQL. For queries requiring joins, Map-Reduce can be used. However, recent versions of MongoDB support a limited number of joined queries.

2) The NoSQL reasoner can be extended with ontology alignment which can be used in matchmaking. The matchmaking is important in Semantic web and Semantic webservice discovery. I have planned to work on Semantic discovery in my master's thesis where this reasoner can be used after match making implementation. Then it can be used to discover best matched webservice depending on user query and based on an ontology described service repository.

In this project, we proposed an optimized approach for ontology classification. The proposed strategy makes use of state of the art document based database technology in addition to Task parallel library which can use recent many cores machines to speed up task execution.
9. Reference


[22] TPL performance comparison overview:
http://www.codeproject.com/Articles/362996/Multi-core-programming-using-Task-Parallel-Library


[29] Weihong Song, Bruce Spencer and Weichang Du. WSReasoner: A Prototype Hybrid
Reasoner for ALCHOI Ontology Classification using a Weakening and Strengthening Approach.


[34] MongoDB installation: http://altafhussainbd.wordpress.com/tag/installing-mongodb/

1 APPENDIX

Solution code given below (the serial code is not given for space. Complete solution can be found in mail attached):