Enabling Graph Appliance for Genome Assembly

Rina Singh*, Jeffrey A. Graves*, Sangkeun Lee† Sreenivas R. Sukumar†and Mallikarjun Shankar†
*Knowledge Discovery Lab, Tennessee Technological University, TN, USA
†Computational Sciences and Engineering Division, Oak Ridge National Laboratory, TN,USA
Email: {rsingh43, jagraves21}@students.tntech.edu and {lees4, sukumarsr, shankarm}@ornl.gov

***Abstract*—In recent years, there has been a huge growth in the amount of genomic data available as reads generated from various genome sequencers. The number of reads generated can be huge, ranging from hundreds to billions of nucleotide, each varying in size. Assembling such large amounts of data is one of the challenging computational problems for both biomedical and data scientists. Most of the genome assemblers that have developed use de Bruijn graph techniques. A de Bruijn graph represents a collection of read sequences by billions of vertices and edges, which require large amounts of memory and computational power to store and process. This is the major drawback to de Bruijn graph assembly. Massively parallel, multi-threaded, shared memory systems can be leveraged to overcome some of these issues. The objective of our research is to investigate the feasibility and scalability issues of de Bruijn graph assembly on Cray's Urika-GD system; Urika-GD is a high performance graph appliance with a large shared memory and massively multithreaded custom processor designed for executing SPARQL queries over large-scale RDF data sets. However, to the best of our knowledge, there is no research on representing a de Bruijn graph as an RDF graph or finding Eulerian paths in RDF graphs using SPARQL for potential genome discovery. In this paper, we address the issues involved in representing de Bruin graphs as RDF graphs and propose an iterative querying approach for searching cycles to find Eulerian paths in large RDF graphs. We evaluate the performance of our implementation on real world ebola genome datasets and illustrate how genome assembly can be accomplished with Urika-GD using iterative SPARQL queries.**

I. Introduction

Knowledge about genome sequences has become critical in many areas, such as molecular biology, evolutionary biology, medical diagnosis, biotechnology, forensic biology, and biological research. Genomes sequences are obtained using many different methods and technologies, all of which are commonly referred to as genome or DNA sequencing. The process of determining the sequences of nucleotides bases (Adenine (A), Guanine (G), Cytosine (C), and Thymine (T)) of an organism's genome [1] is called genome sequencing. It plays an important role in identifying the causes of disease by decoding individual genes, helping with personalized medicine, improving agriculture, energy, the environment, and public health and welfare [2]. The genomic sequencing technologies available today do not produced the complete genomic sequence in a single pass, but instead generate short extracts called the reads. Genomic reads generated by different genome sequencers are constantly growing, and may result in hundreds to billions of reads, each varying in size. For example, the Illumina HiSeq [3] system can generate up to 3 billion reads. These reads are not usable or meaningful unless they are assembled into the original genome. Genome assembly is the process of reconstructing the original genome sequences of an organism from the read sequences [4].

However, assembling those massive amounts of reads into a potential genome has been one of the most challenging problems in bioinformatics, since the reads are not complete extracts of the original sequence and contain several types of errors. For example, common errors include repeats, gap, overlaps, and sequencing errors. Dealing with such errors in unavoidable with most assemblers, because the de Bruijn graph is heavily affected by even the smallest of sequencing errors. Hence, it is necessary to uncover and reconsider those sequencing errors in the reads before assembly. Our main goal of this research is to discover the existence of an Eulerian path in a given graph with no gaps, overlaps, errors, or repeats using SPARQL queries. In the presence of errors, we apply existing methods [5], [6] to deal with sequencing errors in reads and then apply our approach for assembly.

When similar genomes are available, they can be used to guide the reconstruction, and reads can be aligned with the available genome. This assembly approach is called reference assembly. In contrast, genome assembly performed without the aid of a reference genome is called de novo genomic assembly. De novo genomic assembly becomes necessary when working with a new species or new genome sequence such as the sequences produced by Illumina HiSeq [3] and Genome 10K [7]. Various research has been going on in the area of de novo assembly of genome reads even though the problem has proven to be NP-hard [8]. De Bruijn graph assembly is one type of de novo assembly, based on mathematical de Bruijn graph concept. Many assemblers (Velvet [5], MEGAHIT [9], SOAPdenov2 [10]) have used de Bruijn graphs for genome assembly.

The de Bruijn graph used in de novo assembly is similar to, but not exactly the same as, the mathematical concept and plays a very important role in assembly. The de Bruijn graph used in genome assembly is defined as a directed graph where each node represents a unique k-mer (substring of length k) present in the input reads, and an edge exists between two vertices when corresponding k-mers share an exact $(k-1)$ overlap [11]. These k-mers are obtained directly from the reads, and this representation of the reads as a graph can result in billions of vertices and edges. For example, nearly three billion vertices and edges exist in the human genome graph [11]. Hence, the challenge faced when using de Bruijn graphs for de novo assembly is the size of the graph, which requires

massive amounts of memory and computational power for storage and processing. Cray's graph appliance, Urika-GD, has been built to address the challenges of processing and transforming huge amounts of data while at the same time helping to find unknown and hidden relationships and patterns in big data with its scalable-shared memory architecture [12]. Urika-GD is optimized for the Semantic Web technologies: Resource Description Framework (RDF), SPARQL Protocol and RDF Query Language (SPARQL). RDF has been used as a scalable graph representation for integrating, querying, and analyzing large data volumes. SPARQL is the query language used to access information stored in RDF graph data sets. The objective of our research is to investigate the scalability of de Bruijn graph assembly using RDF graphs and SPARQL on Cray's Urika-GD appliance.

A. Motivation

Genomic data has been growing exponentially and representations are often ambiguous. Various ontologies for representing data both structurally and semantically have been published to explicitly describe experimental details in order to provide better understanding and quality checking, as well as promote reuse, preproduction, and integration of the data [13]. For example, Gene Expression Omnibus (GEO) [14], Gemma repository [15], Oncomine [16], and ArrayExpress [17] all meet the requirements outlined by the Minimum Information About a Microarray Experiment (MIAME) [18] standard. The MIAME standard allows these biomedical ontologies to provide consistent annotations of experiments. However, the annotation represented by these ontologies are not machine-readable by other software. Hence, the integration of knowledge across different data sources remain a challenge. The availability of powerful graph appliance (Urika-GD) for processing huge datasets and Semantic Web techniques (RDF/SPARQL) to represent DNA sequences in a machine-readable format is great motivation for assembling entire genomes. The main goal of the Semantic Web is to assist knowledge integration by creating a cross domain and distributed knowledge base while representing data with explicit meaning.

The increasing popularity of Semantic Web technologies has huge potential in different research areas. We have tried to apply this powerful technique to genome assembly research, which we hope will be a useful contribution to both the Semantic Web community and the healthcare domain in different ways. For example, genome annotation is a process of providing descriptions about the genetic elements of a sequenced genome in such a way as to aid all types of biologists in extracting the knowledge needed for analysis and interpretation [19]. In other words, annotations represent the meaning of the sequenced genome. The main goal of annotation is to identify the key features of the genes in order to describe the life cycle of the species (i.e., the structure and reproduction of the genes along with the proteins they encode) [20]. After annotation, biologists are interested to discover how the annotated regions interact with each other.

```
Escherichia coli RecA protein (recA) gene,
    complete cds.

  gene    783..1961
          /gene="recA"

  CDS  783..1961
          /gene="recA"
          /function="DNA repair protein"
          /product="RecA protein"
```

(a) Traditional Methods

```
gene:Escherichia_coli
  a:hasStart integer:783;
  a:hasEnd integer:1961;
  a:hasGeneName string:recA.

CDS:exampleCDS
  a:hasGene gene: Escherichia_coli;
  a:hasStart integer:783;
  a:hasEnd integer:11;
  a:hasProdDescription string:RecA protein;
  a:hasFunDescription string:DNA repair protein;
```

(b) Semantic Web Technology

Fig. 1. Example of Prokaryotic Gene Annotation

Here, Semantic Web technologies play a vital role in discovering relationship patterns among genes, providing huge potential for diagnostic purposes. For example, neuroscience information and the mapping of gene expressions across the entire brain can be combined to comprehend illnesses such as Parkinson's Disease [21], [22]. RDF stores annotation information in a standard way that is understandable to other organizations. Figure 1 illustrates how Prokaryotic gene is expressed using traditional methods as well as semantic web technology. Another contribution could be to the Linked Open Data (LOD) Cloud which is a representation of the RDF triples that can be searched through SPARQL endpoints using the SPARQL query language. RDF and SPARQL are two major and powerful technologies of the Semantic Web. RDF is the data model proposed by the World Wide Web Consortium (W3C) that allows to linked entities across various data sources on the web. RDF allows integration of annotations from different data sources linked via the LOD cloud. RDF is the standardized unified framework proposed by W3C for accessing and integrating this data with other datasources.

B. Challenges

Motivated by the above features of Semantic Web technologies, we have developed a semantic aware de Bruijn graph assembly solution on one of Cray's Urika-GD appliances. Our aim is to assemble large genome sequences with the Urika-GD platform. However, implementing a genome assembly algorithm with a de Bruijn graph approach based on SPARQL queries is not a trivial task and presents several challenges that arise when processing large graphs:

- De Bruijn graphs are not straightforward to represent as RDF graphs, and Eulerian path finding algorithms are not easy to implement with SPARQL queries. An iterative

algorithm needed to be designed to find Eulerian paths in RDF graphs using mainly SPARQL queries.
- The RDF representation of de Bruijn graphs results in huge sets of triples, and hence the scalability issue remain the same as without the RDF representation.
- Many algorithms for finding Eulerian paths are recursive, and iterative counterparts needed to be found.
- SPARQL does not directly support iterative queries, but rather a sequence of queries to perform the task must be generated.

C. Our Solution

To meet the above challenges, we design an iterative Eulerian path finding algorithm based on Hierholzer's algorithm using SPARQL. To achieve this, we apply the framework outlined by Techentin et al. [23], and borrow some ideas from GM-SPARQL [24]. The main contributions of this paper are as follows:

- We present an iterative version of Hierholzer's algorithm based on SPARQL for iteratively obtaining Eulerian paths in de Bruijn RDF graphs.
- We give a SPARQL implementation to solve the all pairs shortest path problem.
- We evaluate the performance of our implementation with Apache Jena Fuseki, a framework for processing RDF data and the Urika-GD appliance. The experimental results show that the Urika-GD appliance has potential for use in de Bruijn de novo assembly.

The following sections are organized as follows. Section II introduces basic graph theory definitions as well as background information about Semantic Web technologies. In Section III we describe our proposed method to find Eulerian paths in de Bruijn graph using RDF/SPARQL. Section IV presents our experimental results, and finally we conclude our paper with some potential future work in Section V.

II. Background

In this section we introduce basic definitions, concepts, and techniques from graph theory and Semantic Web that have been used for genome assembly on Cray's Urika-GD appliance.

A. Directed Graph

While none of the graphs we will be working with are simple directed graphs, we take this opportunity to provide the definition:

Definition 1: A (simple) directed graph, $G = (V, E)$ is an ordered pair consisting of a set of vertices, V, and a set of edges, $E \subseteq (V \times V) \setminus \{(v, v) \mid v \in V\}$.

A simple directed graph does not contain loop edges (edges from a vertex to itself) or multiple edges between a given pair of vertices. The mathematical concept of a de Bruijn graph, outlined in Section II-C, is a directed graph with loop edges.

Definition 2: A directed graph with loop edges is an ordered pair, $G = (V, E)$, consisting of a set of vertices, V, and a set of edges, $E \subseteq (V \times V)$. An edge of the form (v, v) where $v \in V$, is called a loop edge.

The de Bruijn graph used in genome assembly is not a directed graph with loop edges but rather a directed multigraph with loop edges.

Definition 3: A directed multigraph with loop edges is an ordered pair, $G = (V, E)$, consisting of a set of vertices, V, and a multiset of edges, $E \subseteq (V \times V)$.

B. Eulerian Path

The idea of an Eulerian path is very importnat to de novo genome assembly that uses de Bruijn graphs. It is the possible Eulerian paths in the graph that define the potential genomes.

In a graph, a **walk** is a sequence of alternating vertices and edges such that the end points of each edge are the same as the preceding and following vertices. If there exists a walk starting at vertex v and ending at vertex u, we say that vertex u is **reachable** from vertex v. If all vertices in the sequence are unique, we call the walk a **path**. In the case that all of the vertices in the walk are distinct except the first and the last, the walk is called a **cycle**. If all edges in the walk are distinct, we call the walk a **trail**. If the first and last vertices in a trail are the same, we call it **circuit**. An **Eulerian trail** (often called an **Eulerian path**) is a trail in which every edge in the graph appears. If the first and last vertices in an Eulerian trial are the same, the trial is called an **Eulerian circuit** (often called an **Eulerian cycle**).

The conditions required for a directed multigraph with possible loop edges to contain an Eulerian path/cycle are well known, and are given in the following theorem:

Theorem 1: Let G be a directed (multi)graph. G has an Eulerian cycle if and only if every vertex has equal in-degree and out-degree, and all vertices of nonzero degree belong to a single weakly connected component (a maximal collection of vertices in which every vertex is reachable by another vertex in the collection when edge direction is ignored). G has an Eulerian path if at most one vertex has $(\text{out-degree}) - (\text{in-degree}) = 1$, at most one vertex has $(\text{in-degree}) - (\text{out-degree}) = 1$, every other vertex has equal in-degree and out-degree, and all vertices of nonzero degree belong to to a single weakly connected component.

C. The De Bruijn Graph

There is a difference in the mathematical concept of a de Bruijn Graph, named after Nicolaas Govert de Bruijn, and the de Bruijn graph used in genome assembly, both of which are defined over a set of fixed length sequences of symbols coming from a known alphabet. The mathematical definition is as follows:

Definition 4: An nth dimensional de Bruijn graph defined on an alphabet A is a directed graph representing the overlaps between all possible sequences of m symbols from A. More specifically, if $A = \{a_1, \ldots, a_m\}$ is a collection of symbols, then the nth dimensional de Bruijn graph on A is given by $DB_{n,A} = (V, E)$

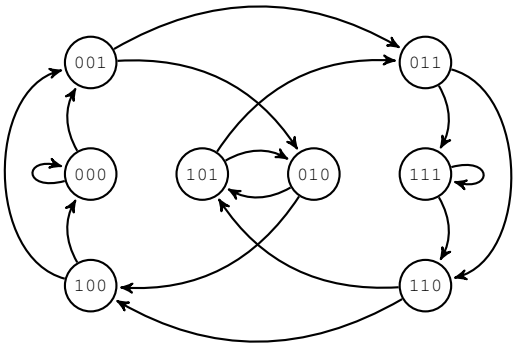


Fig. 2. Example De Bruijn Graph

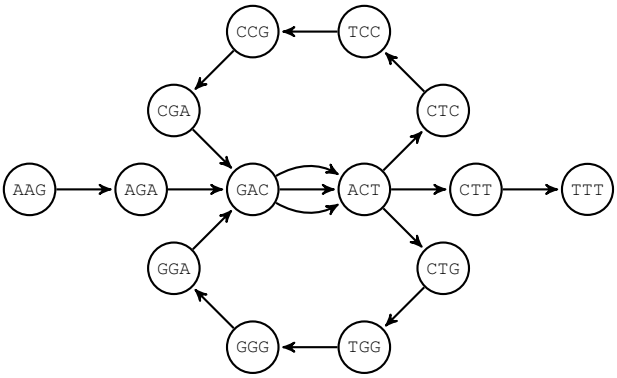


Fig. 3. De Bruijn Graph Representation of AAGACTCCGACTGGGACTTT for $k = 4$

where $V = \{s_1, \ldots, s_n \mid s_i \in A, 1 \leq i \leq n\}$ and $E = \{((s_1, \ldots, s_n), (s_2, \ldots, s_{n_1})) \mid s_i \in A, 1 \leq i \leq n+1\}$.

Informally, a de Bruijn graph is thought of as a directed graph with loop edges. The vertices of the graph represent sequences of characters from the alphabet, and two vertices have an edge between them if their character strings share an $n-1$ character overlap. Figure 2 depicts the de Bruijn graph $DB_{n,A}$ when $n = 3$ and $A = \{0, 1\}$. It is known that every de Bruijn graph contains an Eulerian cycle and a Hamiltonian cycle (a cycle in which every vertex appears).

A de Bruijn graph representation of a collection of strings can be constructed in the following manner. Let $k > 1$ be a fixed but arbitrary integer. For each k-mer (substring of length k) in the strings, define the left and right $k-1$-mer to be the first $k-1$ and last $k-1$ characters of the string, respectively. Each distinct $k-1$-mer becomes a vertex in the de Bruijn graph representation. For every k-mer, add an edge from the vertex representing the left $k-1$-mer to the vertex representing the right $k-1$-mer. This construction produces a directed multigraph with loop edges. It differs from a true de Bruijn graph in the following ways: (1) There can be multiple edges between a pair of vertices. (2) Two vertices that represent a $k-1$ character overlap only have an edge between them if the represented substrings appeared together as left and right $k-1$-mers of some k-mer in the original collection of strings. The de Bruijn graph representation of the text AAGACTCCGACTGGGACTTT when $k = 4$ can be seen in Figure 3.

D. Eulerian Path Based Approach for Genome Assembly

In order to construct potential genomes from a sequence of reads, the reads are first converted into a de Bruijn graph representation. Due to reads overlapping, possible errors in the data, and insufficient coverage, the graph representation of the reads will almost certainly not contain an Eulerian path. Various techniques can be applied to transform the graph to ensure it will have an Eulerian path, but repeats in the genome can give rise to the existence of multiple distinct Eulerian paths. Our primary focuses has been to determine if it is possible to find Eulerian paths using SPARQL, rather than trying to modify the graph to ensure the existence of an Eulerian path. We apply existing methods [5], [6] to deal with sequencing errors in the reads to obtain an Eulerian graph in which to apply our assembly approach. Once an Eulerian path has been found, the reconstructed genome is found by taking the characters from the first vertex in the path, followed by the last character of each subsequent vertex. The graph seen in Figure 3 has two possible Eulerian paths resulting in the following two potential genomes:

- AAGACT**CC**GACT**GG**GACTTT
- AAGACT**GG**GACT**CC**GACTTT

E. RDF/SPARQL

The Semantic Web provides a huge collection of linked datasets where RDF provides the standard representation format and facilitates easy interlinking with each other. RDF is defined as an infrastructure that enables the encoding, exchange, and reuse of structured metadata [25]. It supports relationships between entities of web resources that are identified by URIs (Uniform Resource Identifiers). There are several common serialization file formats for expressing the web resources in RDF data (i.e., Turtle, N-Triples, N-Quads, RDF/XML). An RDF statement is expressed as a triple in the form `subject,predicate,object`. For example, Tim Berners-Lee (subject) invented (predicate) World Wide Web (object). The subject and predicate represents the web resources (expressed as URIs), whereas objects can be either a web resources or literals (integers or strings), which may define other properties. One way to think of a triple is as a directed edge in a graph where the subject is a source vertex, the object is the destination vertex, and the predicate is a label assigned to the edge. Properties of resources are identified with the properties types. A property type is used to express the relationship between subject resources and object resources in a triple. A set of properties that define the same resource is referred to as description. When two or more resources are described with the same URI or have the same description, they can be merged across different data sources [26]. And so, in our research, an RDF data model is very useful and flexible for defining different kinds of genes, which can be

easily aggregated in cases of similarity. The RDF data across the datasets can be retrieved by a standard SPARAL query.

SPARQL is a W3C standard query language that allows querying over the vast amount of RDF graph data. SPARQL essentially works by matching patterns in the graph to retrieve the RDF data. The basic concept of SPARQL queries are triple patterns, where variables are bound to the subjects, predicates or objects of the triples in the query to the triples of the RDF data [27]. SPARQL queries are executed on RDF triplestores via their SPARQL endpoints. There are various triplestores available such as Sesame [28], rdfDB [29], Jena [30], Redland [31], Kowari [32], Apache Jena Fuseki [33], and so on. We have used Apache Jena Fuseki and Cray's Urika-GD for our triplestores. Various SPARQL libraries are available to interact with RDF triplestores such as SPARQLWrapper for Python [34], Sgvizler [35], EasyRdf [36]. We have used SPARQL-Wraper to query the corresponding SPARQL endpoints with Python scripts.

III. Finding Eulerian Paths in De Bruijn Graphs Using SPARQL

Assembling a genome from a sequence of reads using de Bruijn graph involves finding all unique Eulerian paths in the graph. In order to performed de novo assembly on a de Bruijn graph using SPARQL, we first had to design an RDF representation of a de Bruijn graph. Every edge in the de Bruijn graph must be represented as a triple in the RDF graph. The source (subject) and destination (object) of the edge must be URIs. We use URNs of the form `urn:[text]` as the source and destination. Here, `[text]` is the substring (the left or right $k-1$-mer) represented by the vertex. To achieve multiple edges between a given pair of vertices, predicates are created using the form `urn:edge/[`n`]`, where n is an arbitrary unique label for the edge. Before attempting to find an Eulerian path, we apply existing techniques for modifying the de Bruijn graph representation in order to obtain a graph with an Eulerian path. The obvious first attempt to obtain an Eulerian path involves writing a direct query in SPARQL to obtain the path. Upon constructing the query it becomes obvious that this approach will not work. We have chosen to implement an iterative version of Hierholzer's algorithm written using a sequence of SPARQL queries.

A. Direct Queries

While it is possible to construct a direct query to obtain an Eulerian path from a graph, it is unfeasible for large graph. We choose to present it here since a very similar approach is used to extract cycles from a graph needed for the Hierholzer's algorithm. Figure 4 depicts the direct query used to obtain the Eulerian path of the graph seen in Figure 6b. Notice that for each edge in the graph, a triple must be specified. This part of the query grows linearly with respect to the number of edges in the graph. Also notice the use of filters to ensure that no edge is repeated. The number of conditions in the filter grows at a quadratic rate with respect to the number of edges. And so, even for relatively small graphs, the query generated

```
SELECT ?v0 ?e0 ?v1
  ?e1 ?v2 ?e2
  ?v3 ?e3 ?v4
  ?e4 ?v5 ?e5
  ?v6 ?e6 ?v7
  ?v7 ?e7 ?v8
WHERE {
  VALUES (?v0 ?v8) {(<urn:a> <urn:i>)}
  <urn:a> ?e0 ?v1 .
  ?v1 ?e1 ?v2 .
  ?v2 ?e2 ?v3 .
  ?v3 ?e3 ?v4 .
  ?v4 ?e4 ?v5 .
  ?v5 ?e5 ?v6 .
  ?v6 ?e6 ?v7
  ?v7 ?e7 <urn:i> .
  FILTER (
    ?e0!=?e1 && ?e0!=?e2 && ?e0!=?e3 &&
    ?e0!=?e4 && ?e0!=?e5 && ?e0!=?e6 &&
    ?e0!=?e7 && ?e1!=?e2 && ?e1!=?e3 &&
    ?e1!=?e4 && ?e1!=?e5 && ?e1!=?e6 &&
    ?e1!=?e7 && ?e2!=?e3 && ?e2!=?e4 &&
    ?e2!=?e5 && ?e2!=?e6 && ?e2!=?e7 &&
    ?e3!=?e4 && ?e3!=?e5 && ?e3!=?e6 &&
    ?e3!=?e7 && ?e4!=?e5 && ?e4!=?e6 &&
    ?e4!=?e7 && ?e5!=?e6 && ?e5!=?e7 &&
    ?e6!=?e7
  )
}
```

Fig. 4. Example Direct Query

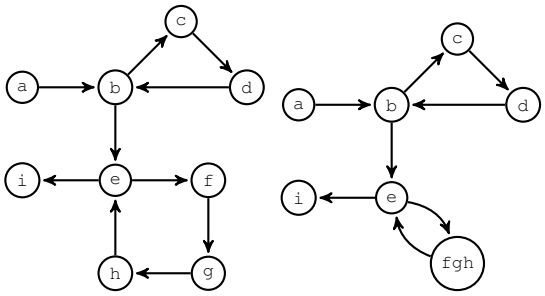
becomes very large. In order to reduce the search space the SPARQL query engine must explore, the starting and ending vertices of the path are specified within the query. Determining the starting and ending vertex in SPARQL is fairly straight forward, and a query to obtain these vertices can be found in Figure 5. Note that the starting vertex is the vertex with one more outgoing edge than incoming, and the ending vertex is the vertex with one more incoming edge than outgoing.

```
SELECT ?v ?in_deg ?out_degree
WHERE {
  {
    SELECT ?v
    (COUNT(?dst) AS ?out_degree)
    (COUNT(?src) AS ?in_deg)
    WHERE {
      {?v ?edg ?dst}
      UNION
      {?src ?edg ?v}
    }
  }
  FILTER (?out_degree != ?in_deg)
}
```

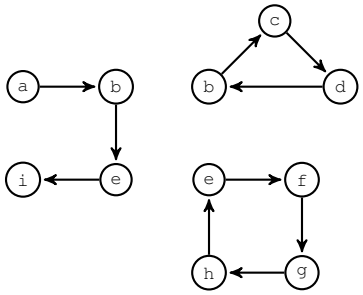
Fig. 5. Sample Path Finding Query

B. Hierholzer's Algorithm

Hierholzer's algorithm [37] is one of most efficient algorithms known for finding Eulerian cycles in graphs. Hierholzer's algorithm begins by choosing an arbitrary vertex and finding an arbitrary circuit involving the chosen vertex. A circuit is guaranteed to be found because the graph is strongly connected and the in degree is equal to the out degree for every



(a) Sample Directed Graph (b) Sample Compressed Graph



(c) Cycles Removed

Fig. 6. Iterative Queries Approach

vertex. Upon finding a circuit, the algorithm checks to see if any vertices along the circuit contain edges that are not part of the circuit. While such edges remain, the algorithm repeats the process to ensure only unused edges are traversed. By expanding the newly found circuit in the previously found circuit, the Eulerian cycle is build up. Traditionally, this processes is implemented using a stack and a depth first search. The correctness of the Hierholzer's Algorithm hinges on a lesser known characterization of Eulerian graphs: a graph contains an Eulerian circuit if and only if all vertices of nonzero degree belong to a single weakly connected component and the graph can be decomposed into edge-disjoint cycles.

C. Hierholzer's Algorithm in SPARQL

The first step for implementing Hierholzer's Algorithm using SPARQL is to ensure the graph has an Eulerian cycle instead of an Eulerian path. Since an Eulerian path exists in the graph, one can simply add an edge from the ending vertex in the path to the starting vertex in the path to create a graph with an Eulerian cycle. It is easy to identify the staring and ending vertices of the Eulerian path, and the SPARQL query to accomplish this can be found in Figure 5. After this, we proceed by identify and removing edge-disjoint circuits from the graph using SPARQL queries. Once all edges have been removed from the graph, the resulting circuits can be expanded into an Eulerian cycle. To obtain all possible Eulerian cycles, one must explore all possible expansions of the circuits found. To find circuits of a fixed length, it is possible to write a direct query similar to the one found in Figure 4 for obtaining paths by requiring the starting vertex to be the same as the ending vertex (thus obtaining a cycle instead of a path). Naively guessing at the length of the cycles can result in queries

1: **ALGORITHM** HIERHOLZER'S ALGORITHM(G)
2: Identify beginning vertex src in Eulerian path.
3: Identify ending vertex dst in Eulerian path.
4: Add edge (dst, src) to G
5: **while** G is not empty **do**
6: Run n iterations of $AllPairsShortestPath$.
7: Identify cycle lengths.
8: Retrieve and store disjoint cycles.
9: Remove cycle edges from G.
10: **end while**
11: Expand cycles. (Accomplished using pure Python)
12: **end ALGORITHM**

Fig. 7. Hierholzer's Iterative SPARQL Algorithm

that return no results, hence wasting computation. Rather than guessing a possible cycle lengths, we designed a SPARQL implementation to solve the all pairs shortest path problem. A Python program was written to generate and submit the necessary SPARQL queries, and after obtaining the circuits using SPARQL, the Python program was solely used to expand the circuits into an Eulerian cycle. After obtaining an Eulerian cycle, the desired Eulerian path can be obtained by removing the additional edge that was inserted to ensure the graph had an Eulerian cycle as apposed to an Eulerian path. Figure 7 outlines the basic algorithm for submitting and evaluating SPARQL queries to find an Eulerian cycle.

D. All Pairs Shortest Path in SPARQL

We implemented an iterative SPARQL query algorithm to solve the all pairs shortest path problem to aid us in identifying cycles. While it is possible to build a direct query to find cycles of length n, if no such cycles exist, all of that computation could have been used trying find cycles of length $n+1$. The key idea of the algorithm is to allow each vertex to keep a record of its neighborhood and expand that neighborhood by one additional hop each iteration. The algorithm starts by recording the adjacent neighbors for each vertex using the initialization SPARQL query found in Figure 8. Figure 9 depicts the first two queries used to fine all neighbors two and three hops away.

```
INSERT {
  GRAPH <urn:shorest_path> {?v ?o 1}
}
WHERE {
  GRAPH <urn:graph> {?s ?p ?o}
};
```

Fig. 8. All Pairs Shortest Path Initialization Query

IV. EVALUATION AND ANALYSIS

We evaluated our approach on the Ebola virus 2014 genome [38]. Using $k = 10$, the final de Bruijn RDF graph representation of the read sequence consisted of 18,948 edges (triples). Along with Urika-GD, the data was loaded into an Apache Jena Fuseki triplestore. The hardware overview for the machines used to run Apache Jena Fuseki can be seen in Table I while Table II depicts the hardware overview for the

```
INSERT {
  GRAPH <urn:shorest_path> {?v ?o 2}
}
WHERE {
  GRAPH <urn:shorest_path> {?v ?s 1}
  GRAPH <urn:graph> {?s ?p ?o}
  MINUS
  {
    GRAPH <urn:shorest_path> {?v ?o ?hop}
  }
};

INSERT {
  GRAPH <urn:shorest_path> {?v ?o 3}
}
WHERE {
  GRAPH <urn:shorest_path> {?v ?s 2}
  GRAPH <urn:graph> {?s ?p ?o}
  MINUS
  {
    GRAPH <urn:shorest_path> {?v ?o ?hop}
  }
};
```

Fig. 9. All Pairs Shortest Path Update Query

Hardware Overview	
Processor Name:	Intel Xeon Processor X5690
Processor Speed:	3.47 GHz
Number of Processors:	4
Cores per Processor:	6
Total Number of Cores:	24
Intel Smart Cache:	12 MB
Memory:	96 GB

TABLE I
APACHE JENA FUSEKI HARDWARE OVERVIEW

machine running the Urika-GD appliance. Note that while the machines running Apache Jena Fuseki had 24 logical cores, Apache Jena Fuseki executes SPARQL queries in a single thread and can only make use of the additional cores for garbage collection.

Using the all pairs shortest path algorithm, it was found that the diameter of the graph (the longest of the shortest paths) was 363 hops, while the longest cycle was 224 hops. The execution time of the Hierholzer's Algorithm in SPARQL is greatly affected by the time it takes to determine the cycle lengths. Figure 10 depicts the run time of each iteration of the all pairs shortest path algorithm for both Urika-GD and Apache Jena Fuseki. Urika-GD takes roughly the same amount of time per iteration. Whereas, Apache Jena Fuseki's time increases quickly over the first couple of iterations. This is probably due to the fact that Apache Jena Fuseki is not parallel. Also, the number of vertices in the n hop neighborhood of any given vertex increases quickly at first and then begins to stabilize, resulting in roughly the same execution time for all iterations 700 and above. Figure 11 shows the combine time required to perform the first n iterations.

Hardware Overview	
Processor Name:	Threadstorm4 Graph Accelerators
Number of Processors:	64
Hardware Threads / Processor:	128
x86 Management and I/O	32
Global Shared Memory:	2 TB

TABLE II
CRAY'S URIKA-GD HARDWARE OVERVIEW

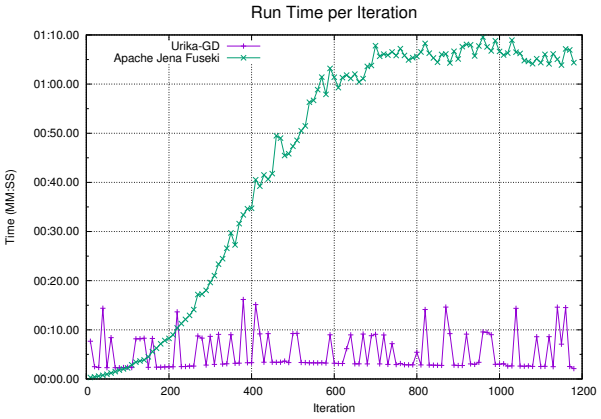


Fig. 10. Execution Time Per Iteration

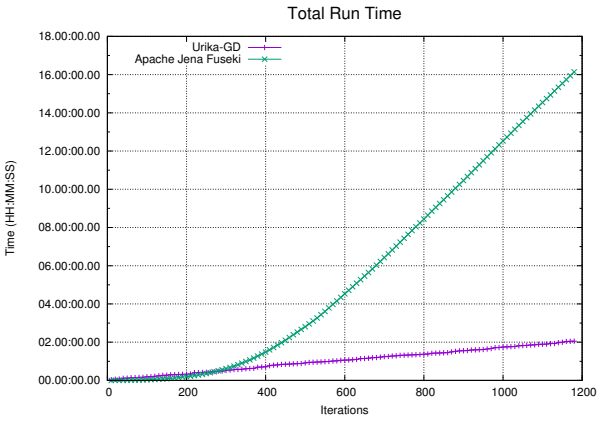


Fig. 11. Total Execution Time

V. CONCLUSION AND FUTURE WORK

In this paper we described the use of an iterative Eulerian path finding algorithm using SPARQL queries wrapped within Python scripts. Based on the timing results we achieved, it appears that the Urika-GD appliance has potential for use in de Bruijn de novo assembly. Our major contributions are an all pairs shortest path algorithm and Eulerian path finding algorithm written primarily in SPARQL. In the future, rather than having to rely on existing tools, we would like to develop an RDF/SPARQL approach for error correction in the read sequences so as to ensure the existence of an Eulerian path in the de Bruijn graph representation.

ACKNOWLEDGMENT

This manuscript has been co-authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

This work was partially supported by the Lab Directors Research and Development Program, and implementation was tested for the CADES environment at the Oak Ridge National Laboratory.

REFERENCES

[1] A. Limasset, "Assembly improvements by read mapping and phasing," Ph.D. dissertation, IRISA, Inria Rennes, 2014.

[2] L. T. Corporation, "Decoding dna," Tech. Rep., 2012.

[3] I. Illumina, "An introduction to next-generation sequencing technology." 2015. [Online]. Available: http://www.illumina.com/

[7] K.-P. Koepfli, B. Paten, and S. J. O'Brien, "The genome 10k project: A way forward," *Annu. Rev. Anim. Biosci.*, vol. 3, no. 1, pp. 57–111, 2015.

[8] A. Bowe, T. Onodera, K. Sadakane, and T. Shibuya, "Succinct de bruijn graphs," in *Algorithms in Bioinformatics*. Springer, 2012, pp. 225–235.

[9] D. Li, C.-M. Liu, R. Luo, K. Sadakane, and T.-W. Lam, "Megahit: an ultra-fast single-node solution for large and complex metagenomics assembly via succinct de bruijn graph," *Bioinformatics*, p. btv033, 2015.

[10] R. Luo, B. Liu, Y. Xie, Z. Li, W. Huang, J. Yuan, G. He, Y. Chen, Q. Pan, Y. Liu *et al.*, "Soapdenovo2: an empirically improved memory-efficient short-read de novo assembler," *Gigascience*, vol. 1, no. 1, p. 18, 2012.

[11] R. Chikhi, A. Limasset, S. Jackman, J. T. Simpson, and P. Medvedev, "On the representation of de bruijn graphs," *Journal of Computational Biology*, vol. 22, no. 5, pp. 336–352, 2015.

[12] (2015) Cray urika-gd graph discovery appliance. [Online]. Available: http://www.cray.com/products/analytics/urika-gd

[13] E. Merrill, S. Corlosquet, P. Ciccarese, T. Clark, and S. Das, "Semantic web repositories for genomics data using the exframe platform," *Journal of biomedical semantics*, vol. 5, no. Suppl 1, p. S3, 2014.

[14] R. Edgar, M. Domrachev, and A. E. Lash, "Gene expression omnibus: Ncbi gene expression and hybridization array data repository," *Nucleic acids research*, vol. 30, no. 1, pp. 207–210, 2002.

[15] A. Zoubarev, K. M. Hamer, K. D. Keshav, E. L. McCarthy, J. R. C. Santos, T. Van Rossum, C. McDonald, A. Hall, X. Wan, R. Lim *et al.*, "Gemma: a resource for the reuse, sharing and meta-analysis of expression profiling data," *Bioinformatics*, vol. 28, no. 17, pp. 2272–2273, 2012.

[16] D. R. Rhodes, J. Yu, K. Shanker, N. Deshpande, R. Varambally, D. Ghosh, T. Barrette, A. Pander, and A. M. Chinnaiyan, "Oncomine: a cancer microarray database and integrated data-mining platform," *Neoplasia*, vol. 6, no. 1, pp. 1–6, 2004.

[17] H. Parkinson, U. Sarkans, N. Kolesnikov, N. Abeygunawardena, T. Burdett, M. Dylag, I. Emam, A. Farne, E. Hastings, E. Holloway *et al.*, "Arrayexpress update—an archive of microarray and high-throughput sequencing-based functional genomics experiments," *Nucleic acids research*, p. gkq1040, 2010.

[4] T. Seemann, "Genome assembly strategies."

[5] D. R. Zerbino and E. Birney, "Velvet: algorithms for de novo short read assembly using de bruijn graphs," *Genome research*, vol. 18, no. 5, pp. 821–829, 2008.

[6] R. Chikhi, G. Rizk *et al.*, "Space-efficient and exact de bruijn graph representation based on a bloom filter." *Algorithms for Molecular Biology*, vol. 8, no. 22, p. 1, 2013.

[18] A. Brazma, P. Hingamp, J. Quackenbush, G. Sherlock, P. Spellman, C. Stoeckert, J. Aach, W. Ansorge, C. A. Ball, H. C. Causton *et al.*, "Minimum information about a microarray experiment (miame)—toward standards for microarray data," *Nature genetics*, vol. 29, no. 4, pp. 365–371, 2001.

[19] L. Stein, "Genome annotation: from sequence to biology," *Nature reviews genetics*, vol. 2, no. 7, pp. 493–503, 2001.

[20] P.McClean, "Genome annotation," 2005. [Online]. Available: https://www.ndsu.edu/

[21] H. Wu, T. Fujiwara, Y. Yamamoto, J. Bolleman, and A. Yamaguchi, "Biobenchmark toyama 2012: an evaluation of the performance of triple stores on biological data," *Journal of biomedical semantics*, vol. 5, no. 1, p. 32, 2014.

[22] M. E. Martone, A. Gupta, and M. H. Ellisman, "E-neuroscience: challenges and triumphs in integrating distributed data from molecules to brains," *Nature neuroscience*, vol. 7, no. 5, pp. 467–472, 2004.

[23] R. W. Techentin, B. K. Gilbert, A. Lugowski, K. Deweese, J. R. Gilbert, E. Dull, M. Hinchey, and S. P. Reinhardt, "Implementing iterative algorithms with sparql." in *EDBT/ICDT Workshops*, 2014, pp. 216–223.

[24] S. Lee, S. R. Sukumar, and S.-H. Lim, "Graph mining meets the semantic web," in *6th International Workshop on Data Engineering meets the Semantic Web*, 2015.

[25] E. Miller, "An introduction to the resource description framework," *Bulletin of the American Society for Information Science and Technology*, vol. 25, no. 1, pp. 15–19, 1998.

[26] R. Singh, "Graphical user interface for silk-a link discovery framework for the web of data," Master's thesis, TU Delft, Delft University of Technology, 2011.

[27] S. Groppe, J. Groppe, and V. Linnemann, "Using an index of precomputed joins in order to speed up sparql processing." in *ICEIS (1)*, 2007, pp. 13–20.

[28] J. Broekstra, A. Kampman, and F. Van Harmelen, "Sesame: A generic architecture for storing and querying rdf and rdf schema," in *The Semantic Web—ISWC 2002*. Springer, 2002, pp. 54–68.

[29] R. V. Guha. (2004) Rdfdb: An rdf database. 2000. [Online]. Available: http://www.guha.com/rdfdb/

[30] J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson, "Jena: implementing the semantic web recommendations," in *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*. ACM, 2004, pp. 74–83.

[31] D. Beckett, "The design and implementation of the redland rdf application framework," *Computer Networks*, vol. 39, no. 5, pp. 577–588, 2002.

[32] D. Wood, P. Gearon, and T. Adams, "Kowari: A platform for semantic web storage and analysis," in *XTech 2005 Conference*. Citeseer, 2005, pp. 05–0402.

[33] (2015) Fuseki: serving rdf data over http. [Online]. Available: jena.apache.org/documentation/serving_data/

[34] D. Shivalingaiah and U. Naik, "Semantic web tools: An overview," 2009.

[35] M. G. Skjæveland, "Sgvizler: A javascript wrapper for easy visualization of sparql result sets," in *The Semantic Web: ESWC 2012 Satellite Events*. Springer, 2012, pp. 361–365.

[36] N. Humfrey, "Easyrdf: A php library designed to make it easy to consume and produce rdf," in *Proceedings of Web Science Conference 2012*, 2010.

[37] L. Lesniak and O. R. Oellermann, "An eulerian exposition," *Journal of Graph Theory*, vol. 10, no. 3, pp. 277–297, 1986.

[38] N. resources. (2014) The sierra leone 2014 assembly of the ebola virus 2014 genome. [Online]. Available: http://hgdownload-test.cse.ucsc.edu/goldenPath/currentGenomes/Ebola_virus/bigZips/