Performance of Serialization Libraries in a High Performance Computing Environment

by
Allen Michael Casey

A thesis submitted to the Department of Computer Science,
College of Natural Sciences and Mathematics
in partial fulfillment of the requirements for the degree of

Master of Science
in Computer Science

Chair of Committee: Omprakash Gnawali

Committee Member: Edgar Gabriel

Committee Member: Weidong Shi

University of Houston
May 2022

Copyright 2022, Allen Michael Casey

ACKNOWLEDGMENTS

The work described here would not have been possible without the support and guidance of many people. First, a special thank you to my mentor and advisor throughout this process, Dr. Edgar Gabriel. From our first conversations, he not only provided advice and assistance that proved instrumental in the success of this effort; he also considered my academic and professional aspirations every step of the way, and tailored this experience accordingly. His mentorship has been invaluable, and I am very grateful for it.

Additionally, I would like to thank Dr. Omprakash Gnawali for chairing my thesis committee, as well as Dr. Weidong Shi for agreeing to take part in my committee. Their feedback and advice in the development of this work greatly contributed to its success, and I truly appreciate their time and efforts. I would also like to express my gratitude to the rest of the Computer Science faculty at the University of Houston. My experiences with them throughout the graduate program made me a better engineer, as well as a more curious and meticulous person. I must also thank my UHCS colleague and LaTeX extraordinaire Khalid Hourani, who provided further revisions and document formatting assistance.

Finally, I would like to express my deep gratitude to my friends and family for their love and support throughout this process. A very special thanks to my partner, Molly, for her constant encouragement and patience. My sincere appreciation to all of you for helping me bring this work to fruition.

ABSTRACT

High performance computing is a subset of distributed computing, and is a paradigm that involves building a cluster of interconnected machines capable of performing operations in parallel. This parallelization enables the cluster to reduce the time needed to perform operations by distributing the work across multiple cluster nodes. The process is heavily dependent on internode communication, and requires nodes to coordinate and communicate by passing messages among themselves.

High performance computing requires that this messaging be very efficient. The messaging process involves serializing the message contents prior to transmission, and deserializing it upon receipt by the receiver. Several libraries have emerged to facilitate serialization and deserialization including Protocol Buffers, FlatBuffers, and MessagePack.

The goal of this thesis is to evaluate the performance of these libraries within the context of a high performance computing software package. As an evaluation infrastructure, a parallelized mass spectrometry tool currently under development at the University of Houston is used, and a new mechanism for serialization is contributed to this tool using each of these three serialization libraries. The libraries are evaluated holistically within the context of the above software package; with many metrics being observed including their performance in terms of execution time and hardware utilization, as well as their general ease of development.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 Introduction

The goal of this thesis is to evaluate the performance of multiple serialization libraries within the context of a high performance computing environment. The libraries evaluated in this analysis include Protocol Buffers, FlatBuffers, and MessagePack. The evaluation was executed by developing a complete serialization mechanism within a large-scale high performance computing package using each of these serialization libraries. The performance of each of these mechanisms was then measured across multiple metrics. After comparing the resulting measurements, it is clear that MessagePack outperforms the other libraries in this use case.

This introductory chapter begins by laying out the motivation for the analysis performed in this thesis, then moves to a description of the goals of the effort. This is followed by a brief summary of the results of the evaluation, and the chapter concludes with an explanation of how the remainder of the thesis is organized.

1.1 Motivation

In several academic and scientific disciplines including weather forecasting, fluid modeling, and quantum mechanics, computing workloads often surpass the computational power provided by traditional desktop machines. The computations necessary in these fields typically include the analyses of extremely large amounts of data, and the execution of many operations. Computing tasks of this scale would require an inordinate amount of time and power in order to be executed by a conventional workstation. For this reason, the distributed computing paradigm High Performance Computing (HPC) has become increasingly prevalent when working with tasks of this nature. HPC is able to parallelize the many operations necessary in these workloads by distributing the work across multiple compute nodes through the use of inter-node messaging, as well as by utilizing the speed of the well-provisioned

hardware found in compute clusters of this type [2].

One scientific area that has yet to take advantage of the benefits of HPC is the field of proteomics research [18]. Proteomics is the science of understanding the impact and interaction of proteins on organisms. Many methods exist within the research area to extract data about these proteins. One method involves the analysis of various particles using a Mass Spectrometer instrument, and often requires an extreme amount of computations on very large datasets. After a review of the literature, no software package could be found that has parallelized the computations of these particle analyses. For this reason, the Parallel Software Technologies Laboratory (PSTL) [21] at the University of Houston has begun developing an HPC-oriented proteomics software package known as HPCMetaMorpheus [20]. The package is based on an open-source proteomics software package written in C# called MetaMorpheus [9]. At this point, the original package source code has been translated to C++ in order to be compatible with the traditional HPC software stack, and the process of refactoring the code in order to achieve parallelization is underway.

One of the primary steps of this refactoring process is the facilitation of the passing of messages between nodes in the cluster. PSTL has chosen to use the Message Passing Interface (MPI) standard [17] to enable the communication necessary. MPI provides an interface for establishing connections among nodes, keeping track of the nodes participating in communication, and sending and receiving messages. In addition to the enablement of the communication protocol, facilitating message passing also requires providing the package with the ability to prepare the messages themselves. In the aforementioned scientific workloads - including proteomics - nodes must be capable of sending complex data structures to and from each other as part of their data sharing and workload distribution. In order for this to be possible, the structure and contents of this data must be somehow converted to a format compatible with being sent via the node interconnect or written to a file.

This compatibility is achieved through a process called serialization, which involves the translation of data from its native format and structure into a more compact and transmissible representation. Many standards and libraries have been introduced to simplify this transformation, including JSON [7], XML [13], Protocol Buffers [11, 31, 16], FlatBuffers [6, 29, 28, 14], and MessagePack [8, 30, 36, 15]. In addition to the use of libraries, programmers can also create ad hoc serialization implementations using binary conversion or string representation. In the current iteration of the HPCMetaMorpheus library, the serialization mechanisms in use are examples of these ad hoc binary and string serialization methods. However, in an HPC environment, all aspects of execution must be optimally efficient - which means that the choice of serialization mechanism must be made in an informed manner. This need has prompted the PSTL members to explore other options for a serialization solution that may potentially be more efficient than the current serialization implementations in the software package. In addition to discerning the most appropriate serialization mechanism for HPCMetamorpheus, an analysis of this nature provides a multifaceted set of measurements and observations comparing multiple serialization libraries in an HPC environment.

1.2 Goals

The overall goal of this work is to contribute an efficient serialization mechanism to the HPCMetaMorpheus proteomics software package currently being developed by PSTL, and in doing so, contribute a performance comparison of multiple serialization libraries in the context of an HPC software package. In order to meet this goal, this work will conduct an empirical and holistic analysis of various serialization libraries in the context of a high performance computing environment. Specifically, this work will explore the serialization mechanisms of Protocol Buffers, FlatBuffers, and MessagePack. The three libraries chosen

are open-source, and each have unique characteristics and differing features. Serialization implementations will be fully developed using all three of them in the HPCMetaMorpheus package, and their performance will be observed and evaluated using several metrics. In reviewing the literature, we find multiple performance analyses of serialization mechanisms [23, 25, 32, 33], but none contribute a comprehensive evaluation and comparison of multiple libraries within the environment of a large high performance computing software package. This work aims to address that void.

First, scaled-down proof-of-concept data structures will be designed and serialized using the three candidate libraries. Once this is done, preliminary performance data will be collected on each of them. This step will also provide development experience with each library. This insight helps ensure that during the implementation of each serialization mechanism in the software package, the design choices being made are leading to the best representation of the given library's performance capabilities.

Once the preliminary testing is complete, a full end-to-end serialization mechanism will be developed for the HPCMetaMorpheus software package using each of the three libraries. The HPCMetaMorpheus package contains multiple classes and data structures that must be serialized and deserialized as part of the necessary message passing processes. The reasoning behind this methodology is that it will give a more complete picture of the performance of each library, rather than simply implementing a subset of the package using each library. This serves to capture as fair of an assessment as possible.

After the serialization mechanism is developed using each library, an empirical analysis and comparison will be performed. The comparison will not only compare the performance of the three libraries brought into this work, but also the currently existing serialization mechanisms implemented within HPCMetaMorpheus. Measurements will be taken of each library's performance in terms of execution time, memory footprint, hardware utilization,

and the size of the code generated by the libraries. In addition to empirical measurements, several non-numeric observations about each library (e.g. quality of API documentation) will also be made in order to contribute to a holistic evaluation of the serialization libraries involved. This evaluation serves to compare the strengths and weaknesses of each of the mechanisms, and if possible, determine the overall most desirable library of the group in this context.

1.3 Summary of Results

The evaluation performed in this thesis compared the performance of Protocol Buffers, FlatBuffers, and MessagePack within a high performance computing environment. These serialization libraries were evaluated across several metrics including serialization execution time, standard deviation of the execution time, size of the serialized representation of data, size of the generated source code, and hardware resource utilization. After comparing all measurements for all libraries, MessagePack was been identified as the serialization library best suited for this environment and use case. MessagePack performed better than its counterparts in all metrics, with two exceptions - the execution time of the encoding operation and the standard deviation of the encoding execution time.

1.4 Organization

The remaining chapters of this work describe the background, development, and evaluation of the serialization libraries within the context of the HPCMetaMorpheus software package. Chapter 2 provides background information on elements of high performance computing, the HPCMetaMorpheus software package, and serialization - including a discussion of the serialization libraries evaluated later in the work. Chapter 3 details the implementation of the three versions of the serialization mechanism within HPCMetaMorpheus using each of

the three serialization libraries. Chapter 4 presents a formal evaluation of the libraries along several performance and development criteria. Finally, Chapter 5 discusses the conclusions of the analysis, as well as opportunities for future work.

2 Background

This chapter provides an overview of the elements of high performance computing including MPI and serialization, as well as the mass spectrometry software package associated with this work. The discussion of the software package includes a description of its existing serialization mechanisms in order to provide motivation and context for the mechanisms contributed by this work. Since the project described later in this manuscript is written in C++, all code examples in this chapter will be displayed in that language.

2.1 High Performance Computing

High performance computing (HPC) is a distributed computing paradigm used to make computationally intensive tasks more efficient. The model consists of building a cluster of compute nodes joined together by a high-throughput network interconnect, with each node being capable of communicating with the others. Compute tasks are distributed across several of these nodes, and the cluster works together as each node computes its portion of the work in parallel. This parallelization of the work enables the cluster to complete the task much more quickly than is possible on a single node. In addition to the parallel nature of the model, each node of the cluster is typically provisioned with powerful hardware which further enhances its ability to accelerate computations.

2.2 Serialization

Within the context of computing, serialization refers to the process of converting a given data object or data structure into a stream of bytes so that it may be stored in disk or memory, or be transmitted to a different system. This serialized version must preserve the structure and state of the data object so that it can be correctly reconstructed later - potentially

by a different system than its origin - in a process known as deserialization. Serializing primitive data types is relatively trivial, but the process becomes much more complex as the data objects become more complex. Structures that contain references - like linked lists and acyclic graphs - are more difficult, with arguably the most difficult objects to serialize being cyclic graphs due to the potential cycles of references.

Data serialization has several use cases. A very common use of serialization involves writing the serialized version of the data to disk, which can include writing the data to a database. This is performed when there is some need to persist an object or structure for later use. Similarly, it is sometimes useful to store the serialized form of an object or structure in memory, which can act as a cache if it is likely that it will be used again soon. Another use for serialization that is frequently employed is sending the serialized data "over the wire" in a message passing context between two machines. This practice is utilized in several environments, including the communication between clients and servers in web-based applications, as well as between nodes in distributed cluster computing scenarios.

The relationship between serialization and message passing has become increasingly important as the use of distributed computing has proliferated. Over the last several years, big data analytics has become a very popular area of research and commercial investment. One of the major paradigms in big data analytics is MapReduce, which involves splitting up work among multiple nodes in a compute cluster, and collecting their results on a single node after their processing is complete. This fundamental concept of parallelization is also present in HPC applications. In addition to messaging used by applications to disperse data, applications in a distributed context must frequently fetch data from other nodes in the cluster due to the data being distributed and partitioned across multiple machines. This process also requires serialization so that the fetched data can be properly reconstructed on the receiving machine.

When selecting a serialization technique or library to employ in their application, developers have several considerations to make. One decision that must be made is whether or not the resulting serialized data will at any point be visually inspected by humans, in which case the serialized representation must be in some human-readable format. In some cases, an even more critical choice centers around the performance of the chosen serialization library. In high performance computing environments for example, the efficiency of data being passed in messages from one node to another is of particularly high importance. This means that the speed at which the serialization library is able to serialize and deserialize the data can impact the performance of the inter-node messaging to a tremendous extent, and developers in high performance use cases will likely prioritize a library that can perform these transformations in as little time as possible. In addition to the readability and execution time of serialization, an important consideration that must be made by developers is the size of the serialized data resulting from serializing a given data object. Different serialization libraries use different schemes and representations for their serialized data, and hence the sizes of these various resulting data objects can vary tremendously. This becomes a concern in memory-scarce environments like mobile devices, in situations where the amount of serialized objects is extremely large, or when the size of the serialized object is such that it begins to adversely affect the performance of sending it in a message or writing it to disk.

There have been multiple standards developed over the years for the serialization of data, with one of the first being Sun Microsystems' External Data Representation (XDR) [5] proposed in 1987, which was capable of encoding data in a binary format. Several human-readable standards followed XDR. Extensible Markup Language (XML) [13] is an open format, text-based serialization standard defined in 1998. JavaScript Object Notation (JSON) [7], another text-based open format serialization standard, was introduced in 2001 and standardized in 2013. Since both XML and JSON are text-based serialization methods,

their serialized data is human-readable, but they are much less space-efficient than their binary-based alternatives. These two standards are also language-independent, so they can easily be deserialized by a different programming language than the one they were serialized by. These standardized schemes, defined for the purpose of sending data from one programming language to another in a format that both can understand, can be described as Interface Definition Languages (IDLs).

Many modern programming languages provide built-in support for object and data structure serialization. JavaScript is capable of natively serializing and deserializing JSON data using the built-in JSON object and its methods parse() and stringify(). In Java, the Java Virtual Machine can automatically internally serialize instances of classes that implement the java.io.Serializable interface [22]. Python's standard library includes the pickle serialization module [27], but also includes several other modules for supporting serialization and deserialization of several encoding standards such as XDR, XML, and JSON. C and C++ do not provide a high-level, abstracted serialization mechanism. However, these languages do support encoding their basic data types as well as data structs into binary format. This ability enables the programmer to develop their own serialization methods.

It is important to note that Java's Serializable class and Python's pickle module differ from the other language-supported modules mentioned here in that they are not simply serializing a data structure containing fields and values; they are serializing a full object. This means that the object's serialized representation includes reference counters, internal methods, and other data not seen by the user - which is a much richer set of information than is typically held in a serialized data structure.

In addition to natively-supported serialization functionalities provided directly by various programming languages, several third-party libraries have been developed that have implementations for several different languages. Protocol Buffers is one such mechanism developed

by Google. It is described as similar to XML, but smaller and more efficient. FlatBuffers is another serialization mechanism released by Google, and is a solution designed with the goal of being as memory-efficient as possible. MessagePack is yet another library that is designed to result in relatively small serialization representations. These are the three libraries explored and evaluated in this effort, and are described in more detail below.

2.2.1 Protocol Buffers

Protocol Buffers [11] is a language-neutral, platform-neutral, open-source serialization mechanism initially developed and used internally at Google in the early 2000's. It was subsequently extended and updated, and the second version of Protocol Buffers - called "Proto2" - was released to the public in July of 2008. "Proto3" was released in July 2016 and includes new features along with support for more programming languages than its predecessor [31]. As of now, there are several programming languages supported by the library including C++, Java, Python, Go, and Dart. Google compares the Protocol Buffers format to XML, but claims that Protocol Buffers are "smaller, faster, and simpler" than the XML format. Similar to the use case of XML, Protocol Buffers are designed to be used with structured data [11].

When integrating Protocol Buffers serialization into a project, development begins by defining the structure of the data to be serialized with a schema file. This definition file is independent of the programming languages being used for serialization, and has the file extension .proto. Within this schema file, the programmer defines "messages", which are the objects that specify the structure of the data to be serialized. These message definitions list all fields present in the data as well as their respective data types. Message fields can be any of the typical scalar data types present in modern programming languages including integers and floating-point values, as well as strings. Fields can also contain composite data

```
message Employee {
    required string name = 1;
    optional int32 employee_id = 2;
    repeated double sales = 3;
}
```

Figure 1: Example of Protocol Buffers message definition

types including enumerations and other message types defined by the programmer in other schema files. In addition to a data type, a "rule" must be specified for each field, out of the options required, optional, and repeated. Fields that are required must contain exactly one value, optional fields may contain either zero or one of the given field's value, and repeated fields can contain zero or more values. Each field defined within a message must also be assigned a unique integer field number. An example of a message definition in a Protocol Buffers schema file is in Figure 1.

Once the structure of the data has been defined within the .proto file, this file must be compiled using the Protocol Buffers compiler, called protoc. When we compile the schema file, the Protocol Buffers compiler generates source code files based on the structure and fields defined within the .proto file. The code in these generated files provides an API that will be used by the programmer to manipulate the data that was described in the schema. This includes setting and accessing field values, along with serializing and deserializing the target data.

Since a given message can include data types defined by messages in other .proto files, the programmer has the ability to import the necessary other .proto files into the current file for use as needed. If these files are in other directories, however, the programmer must specify the so-called –proto-path flag during the Protocol Buffers compiler invocation. This flag allows the programmer to tell protoc where to search for the imported schema files, and multiple paths can be passed in a single compiler invocation. In addition to the import path,

```
$ protoc --proto_path=../.. --cpp_out=. Employee.proto
```

Figure 2: Sample Protocol Buffers compiler invocation for C++ generated code

the programmer can include one or more output directives that specify which programming language(s) the resulting code should be generated in, and which directories the source files should be placed inside of. This feature enables the user to generate source code in multiple programming languages from one protoc invocation on a single .proto file. The Protocol Buffers compiler also allows the passing of multiple schema files at once. A protoc invocation demonstrating some of the options discussed here is shown in Figure 2 above. This invocation generates C++ files in the current directory based on the structure of Employee.proto.

After the source code files have been generated by the Protocol Buffers compiler, the programmer can now leverage them to perform serialization tasks on the data described in the schema file. The details of this process differ depending on the programming language that is being used, but the overall flow of the process remains the same across languages. Before serializing the data, the first task is to build a Protocol Buffers object by setting the field values that were defined in the .proto file. This is achieved with setter methods provided in the generated code for each field value defined. Once this object has been created and assigned values to the necessary fields, other methods in the generated code can be used to serialize the data to the desired data type. The range of data types the programmer can serialize to is language-dependent, but the most common choice is to serialize the data to string format. This string can then be sent over the wire or written to a file as needed. The process of creating and serializing a Protocol Buffers object is shown in Figure 3 below.

```
Employee employee;
// set name and id
employee.set_name(nameString);
employee.set_employee_id(12345);
// create and set monthly sales vector
std::vector<double> sales{1254.67, 6429.83, 878.16};
*employee.mutable_sales() = {sales.begin(), sales.end()};
// serialize object
std::string dataString;
employee.SerializeToString(&dataString);
```

Figure 3: Creating and serializing a Protocol Buffers object

When the serialized data in this string is ready to be accessed, it must first be deserialized back into a Protocol Buffers object. This is similar to the process of serialization, and involves methods provided by the generated code. After deserialization, the field values within the Protocol Buffers object can be accessed using the generated getter methods. If the programmer needs to create a class instance out of the Protocol Buffers object, they must copy each field value from the Protocol Buffers object to the destination class instance. The general deserialization process is shown in Figure 4.

```
// parse object from string
Employee employee;
employee.ParseFromString(inputDataString);
// get values from object
std::string name = employee.name();
int employeeID = employee.employee_id();
auto sales = employee.sales();
std::vector<double> monthlySales = {sales.begin(), sales.end()};
```

Figure 4: Deserializing a Protocol Buffers object

2.2.2 FlatBuffers

FlatBuffers [6] is a cross platform, open-source, language-independent serialization library developed by Google. Wouter van Oortmerssen, a member of Google's game development team Fun Propulsion Labs, wrote much of the initial implementation of the library. The development was motivated by the performance-critical needs of the mobile gaming applications that the team was developing. Since the initial target use of FlatBuffers was with games on mobile devices, the design of the library placed a heavy emphasis on memory efficiency and low serialization operation execution times [28]. Version 1.0 of FlatBuffers was made publicly available in June 2014, with Version 2.0 being released in May 2021 [29]. As of now, there are many programming languages that are at least somewhat supported by the library, but different languages have varying amounts of capabilities and features. The most fully-developed languages include C++, Java, C#, Go, and Swift - but since the project is open source, there are frequent community contributions that increase support for various languages.

Similar to Protocol Buffers, before data can be serialized using FlatBuffers it must first be described within a language-independent schema file. These files have file extension .fbs, and are structured in much the same way as Protocol Buffers' .proto files. The primary method of defining structures within FlatBuffers schema file is the table, which is analogous to the "message" structure of Protocol Buffers. Within the table, the programmer defines fields that have a name and data type, as well as an optional default value. In contrast with Protocol Buffers, fields do not require a unique integer assigned to them. All of the fields specified in a table are optional, and can be omitted when constructing an object. In addition to the table, the programmer can also define other structures within the .fbs file including structs, enums, and arrays. The fields inside of a table can be other tables, structs, enums, strings, or any of the usual scalar types such as integers, floating-point values, and

boolean variables. On the other hand, a struct may only contain scalar values and arrays. An example of a FlatBuffers schema file is shown in Figure 5 below, defining the same string, integer, and array of floating-point values used in Figure 1.

```
table Employee {
    name:string;
    employee_id:int;
    sales:[double];
}
```

Figure 5: Example of a FlatBuffers table definition

FlatBuffers has its own compiler called flatc, much like Protocol Buffers does. After the programmer has finished defining their data in the .fbs file, they must compile this file and generate the code associated with the data they have defined. To compile a given schema file, the command flatc is invoked and is passed the filename to be compiled, along with a generator option that tells the compiler which programming language to generate code for. In contrast to Protocol Buffers, FlatBuffers only creates one generated source file, whereas Protocol Buffers generates two for most programming languages. This singular source file acts as a header file for the programmer to work with the data to be serialized, and contains a similar API with getter and setter methods as the Protocol Buffers generated code, along with serialization and deserialization functionalities. A FlatBuffers compiler invocation is shown in Figure 6. The given generator option denotes that the compiler should create code for use with C++, and since there is no output path declared, the generated files would be placed in the current directory.

```
$ flatc --cpp Employee.fbs
```

Figure 6: Example of FlatBuffers compiler invocation

The FlatBuffers compiler allows several optional arguments when it is invoked. The programmer must include at least one generator option, but multiple languages can also be requested and generated with a single invocation. If the schema file includes objects defined in other schema files, arguments can be added that specify the path along which flatc should search for these other files. Additionally, an argument can be added that tells the compiler which directory to place the generated code in. There are several other optional arguments, but these are those used the most often.

To begin the serialization process using the generated code, the programmer must first create an instance of FlatBufferBuilder. This object helps to construct the structures(s) to be serialized based on the schema. With the exception of strings and vectors, all values are ready to be assigned to schema fields as they are. Strings must be prepared to be serialized in the structure by using the FlatBufferBuilder.CreateString() method, and vectors must be prepared with the CreateVector() method from the same builder class. These vectors must be converted into a FlatBuffers construct known as an Offset in order to be serialized. In order to assign values to schema fields, we can set them individually using the setter methods provided by the generated code, or we can set them all at once using a constructor method. Once all desired fields have been set in the object, the serialization process is executed by calling the builder's Finish()method and passing in the completed object to be serialized. After this is complete, the pointer to the buffer containing the serialized data can be accessed using the GetBufferPointer() method of the builder. An example of the serialization of the Employee object defined in Figure 5 is shown in Figure 7 below.

```
// declare builder
flatbuffers::FlatBufferBuilder bldr;
// initialize field values
auto name = bldr.CreateString(nameString);
int employeeID = 1234;
std::vector<double> tmpVec{1254.67, 6429.83};
flatbuffers::Offset<flatbuffers::Vector<double>> sales = bldr.CreateVector(tmpVec);
// create and serialize Employee object
auto employee = CreateEmployee(bldr, nam, employeeID, sales);
bldr.Finish(employee);
// access buffer pointer for serialized data
char* buf = (char*)bldr.GetBufferPointer();
```

Figure 7: Creating and serializing a FlatBuffers object

Perhaps the most powerful feature of FlatBuffers is that it allows the programmer to access field values of the serialized data without deserializing it first. This "zero-copy" deserialization means that copying the data to a secondary location in memory prior to access is unnecessary, which enables FlatBuffers to be efficient in terms of memory and execution time. If the programmer wishes to rebuild a class object using the serialized data, they can simply extract each field value directly using the buffer pointer, and construct the new object. The unpacking of the Employee object serialized in Figure 7 is depicted in Figure 8 below.

```
// get pointer to Employee data buffer
auto emp = GetEmployee((uint8_t*) buf);
// access field values
std::string name = emp->name()->str();
int employeeID = emp->employee_id();
std::vector<double> sales = {emp->sales()->begin(), emp->sales()->end()};
```

Figure 8: Deserializing a FlatBuffers object

2.2.3 MessagePack

MessagePack [8] is an open-source, cross-platform binary serialization library. It was designed to be similar to JSON in that it can easily send data between multiple programming languages, but the developers of the library claim that MessagePack is "faster and smaller" than JSON. Through its encoding schemes, MessagePack is able to convert data to a binary representation that is more space-efficient than JSON. The serialization mechanism is used by several companies in industry including Redis, Fluentd, Treasure Data, and Pinterest. Currently it has releases that support its use in many programming languages including C++, Java, Lua, Go, Haskell, JavaScript, and Swift, and others [30]. The use of MessagePack requires the Boost C++ libraries [3] in order to facilitate serialization and deserialization.

The development process of MessagePack differs in multiple ways from other serialization libraries like Protocol Buffers and FlatBuffers. First, MessagePack can be used to serialize and deserialize data without first defining a schema. When using this feature, the data that are serialized are appended in their binary representation to the serialized data buffer, and can be deserialized in the same order - all without specifying any explicit structure. When compared to the other serialization libraries, this ability provides a higher level of flexibility in terms of implementation. For more complex data structures, MessagePack enables the

programmer to define the structure of a serialized object by using the MSGPACK_DEFINE() method within the data structure definition. This can be used inside of a class, raw struct, or any other data structure. A small example of defining a C++ raw struct for MessagePack serialization is shown in Figure 9 below.

```
struct Employee {
    std::string name;
    int employee_id;
    std::vector<double> sale;
    MSGPACK_DEFINE(name, employee_id, sales);
};
```

Figure 9: Example of MessagePack data structure definition

Another way in which MessagePack diverges from Protocol Buffers and FlatBuffers is that it does not have a library-specific compiler - mostly due to the fact that there are no schema files involved in the development process that must be compiled in a particular way. This means that depending on the programming language being used, the MessagePack-related additions to the code are either compiled or interpreted as normal with the rest of the code in the application. This feature points to a major difference between this library and the other two libraries discussed here - MessagePack does not generate any accompanying source code files. In the case of Protocol Buffers and FlatBuffers, extensive code is generated upon the compilation of the schema files by their respective compiler. Since MessagePack has neither schema files nor its own compiler, code generation does not take place.

Whether the programmer has defined data structures to be consumed by MessagePack, or they have decided to utilize the schema-free features of the library, the process of serialization using this library is accomplished with the pack() method. In C++, this method takes as arguments an instance of std::stringstream, as well as the object to be serialized. When the

pack() method is called, the data structure is converted to a binary representation based on MessagePack's encoding scheme, and the binary is stored in the stream. This stringstream can then be converted to a buffer pointer or written to a file as necessary. An example of the serialization process in MessagePack is shown in Figure 10.

```
// create sales vector, instantiate Employee
std::vector<double> sales{1234.56, 5432.12};
Employee employee = {nameString, 123, sales};
// serialize Employee into stringstream
std::stringstream ss;
msgpack::pack(ss, employee);
```

Figure 10: Example of serialization in MessagePack

To deserialize the data, the unpack() method must be invoked. In C++, this method's arguments include the stringstream containing the binary data, as well as the size of that binary data in bytes. The process of fully extracting the original object from this binary data is not direct. The unpack() method returns a MessagePack object_handle, which can be converted to a MessagePack object, which can then be converted to the original data structure type that was serialized. An example of deserialization in MessagePack is shown in Figure 11; it uses the stringstream of binary data from Figure 10.

```
msgpack::object_handle handle = msgpack::unpack(ss.str().data(), ss.str().size());
msgpack::object const& obj = handle.get();
Employee deserialized_employee = obj.as<Employee>();
```

Figure 11: Example of MessagePack deserialization

2.3 The HPCMetaMorpheus Software Package

Mass Spectrometry (MS) is an analytical technique capable of classifying protein data generated by a mass spectrometer instrument [24]. The mass spectrometer breaks a given protein into smaller parts known as ion fragments, and captures data characterizing these fragments. After the data is collected, software can be utilized to compare the data gathered by the mass spectrometer with publicly-available databases of known proteins, resulting in a series of possible matches ranked by favorability criteria. One such software package is MetaMorpheus [9], an open-source protein database search tool.

When the mass spectrometer breaks up a protein, the number of resulting fragments can be enormous, which can yield many Gigabytes of data containing fragment characteristics. In addition, the protein databases against which the fragment data must be compared can contain millions of candidate proteins. These two properties combined make the process of protein database search both computationally and data intensive, as each fragment datum must potentially be compared with each entry in the database.

The nature of this compute task makes it an ideal candidate for parallel computations, as the databases can be split and searched independently. In an attempt to leverage this fact, MetaMorpheus is designed to support multithreading. While this enables a degree of parallelization of the computation tasks, there is no support within MetaMorpheus to distribute the work across nodes in a cluster, which would provide a much higher level of parallelism and performance. With this in mind, the Parallel Software Technologies Laboratory [21] at the University of Houston is developing a high performance computing version of MetaMorpheus called HPCMetaMorpheus [20, 18], with the goal of leveraging multi-node parallelism. Throughout the parallelized processing of data in a compute cluster, HPCMetaMorpheus must send many messages between compute nodes, which often contain the serialized version of protein structures. With the goal of developing a new optimized serialization mechanism

for the software package, multiple serialization libraries were evaluated as candidates for the implementation. The analyses performed in this work are a means to that end.

2.4 Existing HPCMetaMorpheus Serialization Mechanisms

The HPCMetaMorpheus software package is fairly large; containing over a hundred thousand lines of code, and consisting of a complex directory structure. There are various directories dedicated to specific aspects of the mass spectrometry analysis process including the reading of inputs, the processing of inputs, and the comparison of the inputs with protein databases in order to produce the results of the particle analysis process. The size of this codebase means that there are a great deal of object-oriented classes defined and utilized throughout the package. This means that large refactorizations of the project code have the potential to become quite labor-intensive very quickly.

From the perspective of serialization however, we can focus the majority of our efforts on only four of the classes contained in the package, which define the specific objects and data that will be serialized and sent from node to node during the parallelized analysis process. These classes are defined as PeptideWithSetModifications, MatchedFragmentIon, FdrInfo, and CrosslinkSpectralMatch, and are listed in Table 1. Table 1 also provides the quantity of fields within each of the four classes that require serialization.

From these numbers we can see that the CrosslinkSpectralMatch class contains many more fields needing to be serialized compared to the other three, which suggests that the

Table 1: Relevant classes and their class member counts

Class Name	Fields to be Serialized
PeptideWithSetModifications	9
MatchedFragmentIon	9
FdrInfo	11
CrosslinkSpectralMatch	29

serialization process of this class will be the most complex. This complexity is due to the fact that the CrosslinkSpectralMatch class is the primary object being operated on during the serialization mechanism, and the members of that class actually include instances of the other three classes listed above. This means that an instance of CrosslinkSpectralMatch contains not only its own members, but those of the nested objects of the other three classes. If we combine these numbers, this implicitly brings the total number of fields in an instance of CrosslinkSpectralMatch to 58.

The majority of the members requiring serialization in the classes above are primitive data types including integer values, double-precision floating point values, string values, and boolean values. In addition to the primitive types listed, CrosslinkSpectralMatch also includes a few containers of type std::vector, as well as the aforementioned instances of the other classes involved in the serialization process.

When serializing data, different data types require more information to be stored so that they may be properly deserialized when necessary. This means that certain types like double-precision floating points and string values inherently result in larger serialized representations of their data. Since different member data types can influence the size of the eventual serialized data object, the data types represented in each class to be serialized are enumerated in Table 2.

Prior to this work, the serialization mechanisms utilized within the HPCMetaMorpheus package have all been ad hoc implementations developed by the maintainers of the package.

Table 2: Data type representation in each serialized class

Class Name	bool	int	double	string	vector	Object
PeptideWithSetModifications	0	4	0	5	0	0
MatchedFragmentIon	0	3	4	2	0	0
FdrInfo	2	0	9	0	0	0
CrosslinkSpectralMatch	3	11	9	3	3	3

The first iteration of a serialization mechanism was string-based, and converted all data involved in the serialization process into some adequate string representation. Each of these string values was appended to a single lengthy string that became the cumulative serialized object. This implementation proved to be insufficient for multiple reasons.

First, the conversion and manipulation of string values is a very computationally inefficient task in terms of execution time, which led to significant performance penalties throughout the serialization process. Considering the fact that the target environment for HPCMetaMorpheus is the high performance computing space, these inefficiencies were not tolerable. In addition to the temporal considerations, the string-based mechanism resulted in prohibitively large serialized representations of objects. In particular, double-precision floating point values required that many fractional digits be included in the serialized representation, each of which requiring one byte of space due to the string format. However, if the developers chose to reduce the amount of fractional digits required by reducing the floating-point precision, this resulted in wildly inaccurate results of the mass spectrometry analysis. The inflated serialized objects of the string-based mechanism not only increased memory pressure, but would also require more time to be transmitted over the wire during message passing procedures within a compute cluster. Again, this is not a desirable trait for a high performance computing software package.

In order to address the issues with the string serialization mechanism discussed above, a new ad hoc mechanism was developed based on the binary representation of the data being serialized. The strategy of this serialization version involved developing a set of functions for packing and unpacking each of the primitive data types of C++, with the binary data of the serialized values being stored in a buffer.

Each of the packing functions takes two arguments as input: the buffer pointer that the programmer using the API is maintaining for the serialized data, and the data value

to be serialized by the function. The function then uses the memcpy() function of C++ to write the binary data of the serialized value to the buffer, and returns the size of the data being serialized. This enables the programmer to advance the pointer in memory as needed to prepare for writing the next value. The unpacking functions are similar, and take two arguments: a buffer pointer to the serialized data, and a pointer to a variable of the data type to be deserialized. The function again uses memcpy() to write the binary data from the buffer into the variable pointer passed into the function, and returns the size of the associated data type. Since these functions are only used internally, this design leverages the fact that the programmer will know exactly the nature of the data to be serialized and deserialized, meaning that the offset of each data value need not be stored and maintained. When the time comes to unpack the data from the buffer, the programmer will unpack the data in the same order it was packed, and the buffer pointer will be advanced as needed according to the size of each data value's binary representation. An example of the binary mechanism's packing and unpacking functions can be found below in Figure 12.

```
static size_t PackInt(char *buf, const int &val) {
    memcpy (buf, &val, sizeof(int));
    return sizeof(int);
}
static size_t UnpackInt(char *buf, int &val) {
    memcpy (&val, buf, sizeof(int));
    return sizeof(int);
}
```

Figure 12: Binary serialization mechanism functions for integers

The "value-level" packing and unpacking functions described above were utilized to create the larger "instance-level" functions responsible for serializing and deserializing instances of

the classes listed in Table 1 and Table 2. Separate versions of these instance-level functions were created within each of the four necessary classes, but the general format across all of them was fairly similar. The function interfaces and structures will be discussed in some detail here, as they are relevant for discussions later in this work.

Generally, the Pack() function for each class was designed to take three arguments as input: the buffer in which to store the serialized data, a pointer value where we will store the buffer length, and the class instance to be serialized. The flow of the function consists of packing the binary representation of each member of the class instance one by one into the target buffer using the value-level functions described above for each data type. Throughout this process, the buffer length is incremented appropriately each time. When the serialization process is complete, the buffer contains the serialized data of all necessary members of the class instance, and the buffer length pointer value contains the length of the buffer. In some cases, it is possible that the programmer will need to serialize several instances of the class at one time. Due to this need, accompanying Pack() functions were developed that accept a std::vector as input instead of a single class instance. In these functions, each item of the vector is iterated over, with the single-instance Pack() function described above being invoked using it. This process results in all members of the vector being sequentially packed into the same buffer.

The Unpack() method of each class has a similar function signature as the Pack() functions, accepting three input arguments: a pointer to the buffer containing the serialized data, a variable indicating the length of the buffer, and a pointer to an "empty" instance of the given class. Data is unpacked from the buffer sequentially using the appropriate value-level functions, and the extracted values are set as members of the empty instance passed as an argument. When the function concludes, the instance the pointer refers to will contain the data from the buffer. Similarly to the Pack() function, there is a version of each Unpack()

function that can take a std::vector as input, in order to satisfy the need to deserialize a group of instances sequentially. Examples of Pack() and Unpack() function signatures can be seen in Figure 13 below.

```
static int Pack(char* buf, size_t &buf_size, FdrInfo *fdr);
static void Unpack(char* buf, size_t &buf_size, FdrInfo **newfdr);
```

Figure 13: Binary serialization function signatures for the FdrInfo class

The serialization and deserialization mechanism of the CrosslinkSpectralMatch class requires some additional discussion. The interfaces of the packing and unpacking functions for the class are the same as those described above. However, due to the increased complexity of the class instances an additional packing or unpacking function is called internally by the outward-facing functions, and is responsible for handling the instance data values appropriately. These functions, named Pack_internal() and Unpack_internal(), serialize or deserialize each of the many data values sequentially. As mentioned previously, the CrosslinkSpectralMatch class contains instances of the other three classes as member variables. This means that the Pack() and Unpack() functions from the other classes are called from within the Pack_internal() and Unpack_internal() functions. This implies that - regardless of what serialization mechanism is being employed - the mechanism must be capable of nested serialization in which serialized objects contain instances of other objects.

3 Implementation

The overall goal of this work is to analyze and evaluate the performance of multiple serialization libraries within the context of a high performance computing software package. As a byproduct of this analysis, an efficient serialization mechanism will be developed for HPCMetaMorpheus to replace the binary mechanism described in Section 2.4.

Three prominent serialization libraries are selected for the analysis: Protocol Buffers, FlatBuffers, and MessagePack. These libraries were chosen based on their prominence in the domain, and their significant backing and contribution from industry leaders like Google and Facebook [11, 6, 28, 8]. In the interest of conducting as fair an analysis as possible, the methodology chosen to perform this evaluation and comparison involved developing the entire serialization mechanism within the software package using each of the libraries being evaluated. This methodology enabled the measurement and observation of full end-to-end runs of the serialization mechanisms within the package, leading to a comprehensive evaluation of the performance of each serialization library. The purpose of this chapter is to describe the process of developing the complete serialization mechanism within the HPCMetaMorpheus software package using each of the three libraries involved in this analysis.

The primary development of the new mechanisms was done on the *salmon* server on the University of Houston's campus, which runs an installation of the openSUSE Linux operating system, Version 15.2 [34]. Before development could begin, it was necessary to install the serialization libraries utilized in the analysis on the *salmon* development server. Since the server runs an openSUSE Linux distribution, the zypper [1] command line package manager was used to install each serialization library package At the time of installation, the most up-to-date version of each serialization library available within zypper was installed on the server, and their respective software versions are listed in Table 3 below.

Table 3: Software versions of the serialization libraries installed

Serialization Library	Version Installed
Protocol Buffers	3.9.2
FlatBuffers	1.12.0
MessagePack	3.2.1

As the design process of the new serialization mechanisms began, multiple high-level design choices were made that affected all three of the new mechanisms. Firstly, despite the internal serialization processes requiring a dramatic refactoring, it was decided that the original interface of the Pack() and Unpack() functions from the binary mechanism would remain unchanged when developing the new mechanisms. This not only streamlined the development process itself, but also helped to ensure that the changes being made to develop the new mechanisms would not cause disruptions in other parts of the codebase that interacted with the serialization functions. The idea was to maintain the "black box" element of the HPCMetaMorpheus serialization functionality, while internally improving its performance.

Another general design decision that was made early in the development process dealt with how individual objects were handled as they were prepared for serialization. The existing binary implementation contained two versions of each Pack() function: one accepting single objects as input, and another accepting a C++ std::vector of the object as input. This enabled the overloaded function to be called with either a single input or multiple inputs and still perform as expected. This ability needed to be preserved in the newly-developed mechanisms, but the nature of the schema files in the Protocol Buffers and FlatBuffers libraries made this a deceptively complex task.

In these libraries, the serialization process is most smooth when the same type of object

will be serialized predictably each time. This means that to alternate between a single object and a vector of objects could make serialization much more difficult. In order to circumvent this issue, each version of the Pack() function that accepted a single object would simply place the object into a std::vector, then call the std::vector version of Pack() using the new single-element container. This indeed caused a performance penalty, but it meant that all objects being serialized would be presented in the same format regardless of quantity, thereby making the serialization schema development process significantly more straightforward. An example of this internal vectorization is shown in Figure 14 below.

```
int MatchedFragmentIon::Pack(char *buf, size_t &buf_len, MatchedFragmentIon *MaF) {
    std::vector<MatchedFragmentIon*> MaFVec;
    MaFVec.push_back(MaF);
    int pos = MatchedFragmentIon::Pack(buf, buf_len, MaFVec);
    return pos;
}
```

Figure 14: Single-object Pack() function the vectorizes input

In addition, the implementation of each of the serialization mechanisms required revisions to the Makefiles of the HPCMetaMorpheus package, with varying degrees of complexity. Details concerning Makefile changes would typically be too fine-grained for a thesis of this scope, but they are discussed here to contrast the effort required to integrate each serialization library into an existing code base. This consideration becomes increasingly relevant as the target code base grows in size and complexity, and the discussion of these efforts serves to contribute to the overall comparison of the libraries involved.

Although there were commonalities in the high-level design, the development of each of the three new serialization mechanisms introduced a unique set of challenges. The differences among each of the three libraries' APIs - as well as the variance in the schema requirements

between them - contributed to these obstacles. In order to communicate the design and development of each of the mechanisms clearly, the creation of each of the three mechanisms is described in detail in the following sections.

3.1 Protocol Buffers Serialization Mechanism

As noted in Section 2.3.1, in order to utilize Protocol Buffers for serialization, the programmer must first define a Protocol Buffers schema file that describes the nature of the data to be serialized. This meant that a .proto definition file had to be created for each of the four classes being serialized within HPCMetaMorpheus, and the creation of these files was the first step in the development of the Protocol Buffers serialization mechanism. Each value seerialized within the four involved classes was appropriately defined within the four respective .proto files, which would inform the Protocol Buffers compiler of the quantity and types of variables to expect. These definition files were then added to the same directory as the source file of the class that they described. For example, FdrInfo.proto was placed in the same directory as FdrInfo.cpp. This was done so that when the definition file was later compiled using the Protocol Buffers compiler, protoc, the resulting generated code files would be accessible within the same directory as the source files of each class. The FdrInfo.proto file is included in Figure 15 below in order to provide an example definition file. This is the most simple of the four schema files, but is an adequate reference.

Extra considerations had to be made for the .proto file of the CrosslinkSpectralMatch class. As discussed previously, CrosslinkSpectralMatch objects contain nested instances of the other three classes involved in the serialization process. This meant that the CrosslinkSpectralMatch definition file had to be capable of referencing the serialized objects defined in the other three schema files. In order to facilitate this within CrosslinkSpectralMatch.proto, the import capability described in Section 2.4.1 was employed. This enabled the objects defined

```
// set protobuf version
syntax = "proto2";
// define the serialized object
message SerializedFdrInfo {
    optional double cumulativeTarget = 1;
    optional double cumulativeDecoy = 2;
    optional double qValue = 3;
    optional double cumulativeTargetNotch = 4;
    optional double cumulativeDecoyNotch = 5;
    optional double qValueNotch = 6;
    optional double maximumLikelihood = 7;
    optional double eValue = 8;
    optional double eScore = 9;
    optional bool calculateEValue = 10;
    optional bool hasFdr = 11;
}
```

Figure 15: FdrInfo.proto schema file

in the other three schema files to be referenced within the CrosslinkSpectralMatch schema file as if they had been defined directly within it. The additional work necessary to be able to locate these schema files from CrosslinkSpectralMatch schema file will be discussed later in this section.

With the .proto files developed, the next step was to compile these schema files using the protoc Protocol Buffers compiler in order to create the associated generated code files. One potential method to accomplish this would be to manually enter each directory containing a schema file, then execute the protoc command on the file. This would clearly be extremely tedious, and would be required after each change to the schema files. In order to avoid this tedium, a new Protocol Buffers-oriented rule called proto was created in the Makefile of the HPCMetaMorpheus home directory. This rule switches to each directory containing a schema file, and executes another proto rule in that directory's Makefile that invokes the Protocol Buffers compiler on the schema file. This enables the user to recompile all of the .proto files by navigating to the home directory and running the command 'make proto'. This compilation would in turn create the generated .pb.cc and .pb.h files that provide the

serialization functionality for each of the four classes involved. These files are generated in the same directory as the schema file, so that they are accessible from the source files of each class.

To begin the development of the Protocol Buffer serialization mechanism, the API provided by the generated code was carefully studied. This API enables the programmer to create serialized Protocol Buffers objects that have been described in schema files, as well as set values for the members of these objects. After declaring an object of the type described in a given .proto file, the programmer can set each value contained in the object using a set of functions provided by the generated code that follow a consistent format:

<*object name*>.set_<*schema-defined variable name*>(<*value to set*>)

These setter methods were used to set the values within the Protocol Buffers objects. The goal was to extract the values from within the input objects passed as arguments to the Pack() function, and transfer each one to the Protocol Buffers objects, one by one. Once all values had been set, the objects were ready to be serialized to their binary representation using the Protocol Buffers API. Since the Pack() function interface dictates that the serialized data be stored in a buffer of type char*, the Protocol Buffers API function SerializeToArray() was invoked on the object, which converted it to its Protocol Buffers binary representation and wrote this data into the char buffer passed to the Pack() function. The API function ByteSizeLong() was also invoked on the object, which returns the integer value representing the number of bytes required to represent the object in binary. This value was then sent to the buf_len pointer passed to the Pack() function, completing the contract set by the function interface. For clarity, an example Protocol Buffers Pack() method is included in Figure 16 which depicts the use of the setter methods as well as the binary serialization method. A similar serialization process was implemented for each of the other three classes.

```
int FdrInfo::Pack(char* buf, size_t &buf_len, FdrInfo *fdr) {
    SerializedFdrInfo sFdr;
    sFdr.set_hasfdr(fdr != nullptr);
    // build serialized fdr if it exists
    if (fdr != nullptr) {
        sFdr.set_cumulativetarget(fdr->getCumulativeTarget());
        /* ~snip~ */ // set remaining values from input object
    }
    size_t size = sFdr.ByteSizeLong();
    sFdr.SerializeToArray(buf, size);
    buf_len = (int)size;
    return (int)size;
}
```

Figure 16: Protocol Buffers FdrInfo Pack() function

The Unpack() function implementation for each class essentially performed the Pack() process in reverse. The first task was to deserialize the Protocol Buffers binary data from the buffer passed to Unpack(). This was done using the Protocol Buffers API function ParseFromArray(). This function enables the programmer to parse a buffer and construct a schema-defined Protocol Buffers object using its contents.

With the object deserialized, its member data values could be accessed with getter methods provided by the generated code file. These functions were simply generated with the same name as the member they access, with the function name being in all lowercase. For example, the getter method for a member named eValue would be called evalue(). Using these getter methods, the member data was sequentially extracted and set inside of the object pointer passed to the Unpack() function, thus completing the contract of the deserialization. As with the serialization function, the Unpack() function is shown in Figure 17,

and is representative of the analogous functions in the other classes.

```
void FdrInfo::Unpack(char* buf, size_t &buf_len, FdrInfo **newfdr) {
    // parse fdr object from string
    SerializedFdrInfo sFdrInfo;
    sFdrInfo.ParseFromArray(buf, buf_len);
    // check if the fdr is null
    bool has_fdr = sFdrInfo.hasfdr();
    // rebuild fdr
    if (has_fdr) {
        FdrInfo* tempVar= new FdrInfo();
        tempVar->setCumulativeTarget(sFdrInfo.cumulativetarget());
        /* ~snip~ */ // extract values from input object
        *newfdr = tempVar;
    }
    else {
        *newfdr = nullptr;
    }
}
```

Figure 17: Protocol Buffers FdrInfo Unpack() function

3.2 FlatBuffers Serialization Mechanism

The first step of the development process for the FlatBuffers serialization mechanism was much like that of the Protocol Buffers mechanism - schema files needed to be defined for all classes involved in the serialization in order to use the FlatBuffers API. As described in Section 2.3.2, the schema files have the file extension .fbs, and must describe each data value that will be serialized within each class instance. A schema file was completed in this way for each of the four classes to be serialized within HPCMetaMorpheus, and these .fbs files were

placed in the directory with the associated source file they describe. The reasoning behind this location for these files is of similar logic as for Protocol Buffers - when the schema files would later be used to generate code files, the generated files would be accessible to the .cpp source files requiring access to the APIs they provide. Figure 18 provides an example of an .fbs file definition.

```
table SerializedFdrInfo {
    cumulativeTarget:double;
    cumulativeecoy:double;
    qValue:double;
    cumulativeTargetNotch:double;
    cumulativeDecoyNotch:double;
    qValueNotch:double;
    maximumLikelihood:double;
    eValue:double;
    eScore:double;
    calculateEValue:bool;
    hasFdr:bool;
}
root_type SerializedFdrInfo;
```

Figure 18: FdrInfo.fbs schema file

With the FlatBuffers schema files for each class developed and placed in the proper directory, these files needed to be compiled using the flatc FlatBuffers compiler so that the resulting generated code could be utilized by the source files of each class. In order to accomplish this, the decision was made to follow the same process as what was done in the Protocol Buffers mechanism. Rather than manually navigating to each directory and issuing the command to compile with flatc, a rule was made in the Makefile of HPCMeta-Morpheus home directory called flatbuffers. This rule sequentially navigated to each of the four directories containing .fbs files, and triggered a secondary rule in the Makefiles of those directories, also called flatbuffers. The secondary rule invoked flatc, and in turn generated the FlatBuffers code based on the objects defined in the schema files. In the case of the CrosslinkSpectralMatch flatc compilation, the command had to provide the paths to each of

the exterior schema files, so that flatc could locate these definitions. In addition this rule, the only other necessary addition was to the CrosslinkSpectralMatch directory's Makefile, and involved adding the include paths to the other three serialization-involved classes within the Makefile's compilation command using the -I option of gcc.

By invoking the FlatBuffers compiler on each of the schema files, this resulted in a header file being generated for each of the files compiled. These header files followed a consistent naming convention as predicated by FlatBuffers: <*class name*>_generated.h. Before the API provided by these generated header files could be utilized by the source files responsible for serialization, the header files needed to be included in the source files. Similar to Protocol Buffers, this was accomplished using the #include preprocessor directive.

The API produced by the generated FlatBuffers header files provides two separate mechanisms for serializing data to its binary representation, and each one is used at various points in the Pack() process. The first mechanism is similar to an object-oriented constructor, and allows the programmer to pass all values to be set into the serialized instance at once. The second mechanism enables the programmer to declare a serialized instance, and subsequently set each member value sequentially. The first method was utilized for all classes but CrosslinkSpectralMatch, with the second method handling CrosslinkSpectralMatch serialization.

As described in Section 2.3.2, the programmer must initialize an instance of FlatBufferBuilder before they can begin to serialize objects to binary. This object is defined by FlatBuffers, and is used to package data values into a buffer during the serialization process - regardless of which of the two previously mentioned serialization methods is employed. In addition to the general FlatBufferBuilder, FlatBuffers also provides a specialized Builder for each class described in the generated header files. These specialized Builder objects are

specific to their class, and subclass the general FlatBufferBuilder. To initialize a specialized Builder, the programmer passes an instance of the general FlatBufferBuilder to the specialized constructor.

These specialized instances were only utilized in the CrosslinkSpectralMatch class' serialization procedure, and were only necessary due to the nested nature of that class. This is because FlatBuffers does not allow the programmer to add a data object to a given Builder if there is an incomplete object in the process of being added to that Builder. For example, an error would result from attempting to add an FdrInfo FlatBuffers object to the FlatBufferBuilder while still adding the other members of CrosslinkSpectralMatch to it. This meant that in order to avoid FlatBuffers-related runtime errors, a specialized Builder was necessary to add each FlatBuffers object of the other three classes to the CrosslinkSpectralMatch FlatBufferBuilder.

Regardless of which FlatBuffers-provided serialization mechanism was used, all classes concluded the serialization process in the same way. Once all data values had been set using the given Builder for a class, the function Builder.Finish() was called, with the FlatBuffers object to be serialized passed as an input argument. This function serializes the data to its FlatBuffers representation and stores it in a buffer. Subsequently, a pointer to this buffer is retrieved by calling the Builder.GetBufferPointer() function. Once the pointer to the serialized data is acquired, this data could be copied into the memory address indicated by the pointer passed to the Pack() function, thereby completing the packing functionality. A truncated example of an HPCMetaMorpheus FlatBuffers Pack() function is in Figure 19 below.

The unpacking process within the FlatBuffers mechanism is fairly straightforward. First, the binary data in the buffer must be deserialized and used to reconstruct a FlatBuffers object. This is done using the GetSerializedFdrInfo() function, which takes the buffer pointer

```
int FdrInfo::Pack(char* buf, size_t &buf_len, FdrInfo *fdr) {
    flatbuffers::FlatBufferBuilder builder;
    /* snip */ // extract values from input object
    // create FlatBuffers object, and serialize it
    auto fdrInfo = CreateSerializedFdrInfo( /* pass values here */);
    builder.Finish(fdrInfo);
    // get data pointer and size of data
    char* tempBuf = (char*)builder.GetBufferPointer();
    int pos = builder.GetSize();
    // copy data to buffer
    memcpy(buf, tempBuf, pos);
    buf_len = pos;
    return pos;
}
```

Figure 19: FlatBuffers Pack() function

as input that contains the serialized data, and returns a FlatBuffers object. The function expects a pointer of type uint8_t*, and since the buffer pointer passed in is a character buffer of type char*, we must cast the pointer in order for it to be accepted by the function. After deserializing the data, each member value can be sequentially extracted from the resulting FlatBuffers object and set into an instance of the given class. This was done using a temporary pointer variable, and once all values were set, the object pointer passed to the function was made to point to this newly-constructed instance. This completed the deserialization process. A truncated version of the Unpack() function can be studied in Figure 20.

```
void FdrInfo::Unpack(char* buf, size_t &buf_len, FdrInfo **newfdr) {
    // deserialize object from data
    auto sFdrInfo = GetSerializedFdrInfo((uint8_t*)buf);
    // reconstruct object if it exists
    if (sFdrInfo->hasFdr()) {
        FdrInfo* tempVar= new FdrInfo();
        tempVar->setCumulativeTarget(sFdrInfo->cumulativeTarget());
        /* snip */ //extract and set remaining values
        *newfdr = tempVar;
    } else {
        *newfdr = nullptr;
    }
}
```

Figure 20: FlatBuffers Unpack() function

3.3 MessagePack Serialization Mechanism

In contrast to the previous two serialization libraries discussed, the use of the MessagePack library does not require that a schema file be defined that describes the object(s) to be serialized. In fact, it is possible to successfully implement a full serialization mechanism without defining any sort of object structure at all, simply by manually serializing each data value one by one using the MessagePack API. Despite not having the notion of a schema file, MessagePack does enable the programmer to describe an object's structure by defining a struct containing all of the member values to be serialized within that object. Due to the complexity of some of the objects serialized within HPCMetaMorpheus, the design decision was made to utilize this latter approach to develop the MessagePack serialization mechanism for the software package.

In order to adhere to the C++ convention of class and struct definitions being located within C++ header files, the struct describing the serialized representation of each class was placed within the header file of that class. As described in Section 2.2.3, all data members that were to be serialized within each class were declared within the struct. Then, the MSGPACK_DEFINE() function was called, and all of the defined data members were passed as arguments. An example of one of these MessagePack definitions is shown below in Figure 21.

```
struct SerializedFdrInfo {
    double cumulativeTarget, cumulativeDecoy, qValue, cumulativeTargetNotch;
    double cumulativeDecoyNotch, qValueNotch, maximumLikelihood;
    double eValue, eScore;
    bool calculateEValue, has_fdr;
    MSGPACK_DEFINE(cumulativeTarget, cumulativeDecoy, qValue,
        cumulativeTargetNotch, cumulativeDecoyNotch, qValueNotch,
        maximumLikelihood, eValue, eScore, calculateEValue, has_fdr);
};
```

Figure 21: MessagePack FdrInfo struct definition

Since there is no generated code associated with the MessagePack process, and the header file of each class will already have been included within the associated source file, no extra #include preprocessor statements have to be added to the .cpp file for each class. However, in order for the header file to utilize the MSGPACK_DEFINE() function, <msgpack.hpp> must be included into the header file using the #include preprocessor directive.

The method used within each Pack() function of the MessagePack mechanism is similar to the one described in Section 2.2.3. First, all data values are extracted from the object to be serialized. An instance of the MessagePack struct is then initialized using these values as the initialization arguments. Next, the MessagePack function msgpack::pack() is invoked, and the struct object is passed to it, along with a newly-declared std::stringstream object. This invocation results in the values within the struct being serialized into their MessagePack

binary representation, with the serialized data being stored in the std::stringstream. Since we want this data to end up in the char buffer pointer that was passed to the Pack() function, we must convert this std::stringstream into char* to complete the function, which can only be done in the fairly indirect process depicted in Figure 22.

```
int FdrInfo::Pack(FdrInfo *fdr) {
    std::stringstream sstream;
    bool has_fdr = (fdr != nullptr);
    if (has_fdr) {
        // build msgpack struct
        SerializedFdrInfo sFdr = {
            fdr->getCumulativeTarget(),
            /* snip */ //extract and set remaining values
        };
        // get serialized data string
        msgpack::pack(sstream, sFdr);
        std::string tmp = sstream.str();
        // get data size
        int bufSize = tmp.size();
        buf_len = bufSize;
        //copy data to buffer
        memcpy(buf, tmp.c_str(), bufSize);
        return bufSize;
    }
    return 0;
}
```

Figure 22: MessagePack FdrInfo Pack() function

Again, the Unpack() process largely follows the MessagePack deserialization process explained in Section 2.2.3. The msgpack::unpack() function only accepts an std::stringstream as the data input, so the first step is to convert our char buffer, which is accomplished using the std::stringstream.write() function. Following this conversion, the msgpack::unpack() function is ready to be called, along with its accompanying functions described in Section 2.2.3. Once the data has been deserialized into a defined MessagePack struct, each data member it contains can be extracted and set into the object that was passed into the function, which completes the deserialization process. An example of the Unpack() function is shown in Figure 23.

```
void FdrInfo::Unpack(char* buf, FdrInfo **newfdr) {
    // convert data to sstream
    std::stringstream sstream;
    sstream.write(buf, buf_len);
    // deserialize data
    msgpack::object_handle objHandle = msgpack::unpack(buf, buf_len);
    msgpack::object const& obj = objHandle.get();
    auto sFdrInfo = obj.as<SerializedFdrInfo>();
    // rebuild object
    if (sFdrInfo.has_fdr) {
        FdrInfo* tempVar= new FdrInfo();
        tempVar->setCumulativeTarget(sFdrInfo.cumulativeTarget);
        /* snip */ // extract all data members
        *newfdr = tempVar;
    }
    else {
        *newfdr = nullptr;
    }
}
```

Figure 23: MessagePack FdrInfo Unpack() function

4 Evaluation

This chapter will present an evaluation of the three serialization mechanisms discussed in Chapter 3. The performance of the three new mechanisms will also be compared to that of the previously-developed binary serialization mechanism described in Section 2.4. The evaluation performed will produce data and observations along several metrics for each mechanism including execution times, standard deviation of the execution times, size of the serialized representation of data, size of the generated source code, and hardware resource utilization.

All measurements presented in Sections 4.1, 4.3, 4.4, 4.5, and 4.6 were performed on the salmon server at the University of Houston. The server contains a single Intel W3565 3.20GHz processor with 4 cores (8 threads), and has 24GB of main memory. The operating system running on the server is openSUSE 15.2 [34], and the gcc version is 10.2 [12]. Refer to Table 3 to review the version of each serialization library used in the development of the mechanisms described.

All measurements presented in Section 4.2 were performed on the Parallel Software Technology Laboratory crill cluster [19] at the University of Houston. Each node utilized in this analysis contains four 2.2 GHz 12-core AMD Opteron processors, for a total of 48 cores per node. Each node is provisioned with 64 GB of main memory, and is connected to other nodes using QDR Infiniband and Gigabit Ethernet interconnects. The cluster runs the openSUSE operating system version 42.3 [35], and utilizes gcc version 5.2.0. The message passing performed for the measurements was implemented using Open MPI [10] version 3.0.1.

Within the context of this discussion, the term "input size" refers to the number of peptide objects to be serialized by an execution, with "peptide" referring to an instance of the class CrosslinkSpectralMatch referred to throughout this document. This object was chosen as the evaluation input because it is the object most often serialized by the HPCMetaMorpheus software package [20], and its data field values can be inspected in Table 1. The expected

workload of HPCMetaMorpheus is highly variable. Thus, in order to capture a fair and thorough evaluation of each mechanism's performance, the analysis is performed using a wide range of input sizes. This includes two sets of performance trials - one with smaller input sets, and one with larger input sets.

The smaller input trial measures the performance of each mechanism using input sizes increasing incrementally from 4 to 100 peptides, and the larger input trial measures performance using input sizes increasing incrementally from 100 to 2000 peptides. These measurements using incrementally larger inputs not only evaluate the scalability of each HPCMetaMorpheus serialization mechanism, but also aim to provide a general insight into how the various libraries perform and scale. The overall goal of this effort is to contribute benchmark data with relevance that extends past the specific implementations of the HPCMetaMorpheus software package.

For all measurements of execution time, the Pack() and Unpack() operations were run ten times for each input size, and the arithmetic mean of the results is presented here. In addition, the performance data from the same ten runs was used to extract the standard deviation data related to execution times. These measurements of execution time were captured using the high_resolution_clock class provided by the chrono library [4]. In order to record the size of the serialized representation of the data produced by each mechanism, the buf_len variable produced by each Pack() function was documented. This variable conveniently already holds the size of the generated data buffer, so its value was simply captured for each input size.

4.1 Size of Serialized Data Buffer

As each serialization mechanism packs its input peptides into their serialized representation, it accumulates this data in a buffer of type char*. For a given input size, the size of the

buffer varies from mechanism to mechanism, and is dependent on the serialization format utilized by each underlying serialization library or implementation. It is advantageous for this buffer to be as small as possible for two primary reasons. First, a smaller buffer will bring about less memory pressure within the system, which is most important in memory-scarce environments or systems with large serialization workloads. Second, smaller buffers will require less time to write to disk or send over the wire to a remote node, increasing overall performance.

A comparison of the buffer sizes resulting from serializing various inputs is shown in Figure 24 below, using the input sizes described earlier in this chapter. Across all input sizes, the binary mechanism produces the largest data buffer. FlatBuffers result in significantly smaller buffers than the binary mechanism, and Protocol Buffers create buffers notably smaller than those of FlatBuffers. MessagePack produces the smallest buffers, generating buffers that are consistently approximately 9% smaller than those of Protocol Buffers.

The primary reason that MessagePack and Protocol Buffers achieve the smallest serialized representation is the way these two libraries handle integer values. Using a process that Protocol Buffers refers to as Varints [16, 15], these libraries automatically encode a given integer value in as few bytes as possible. For example, a 32-bit integer containing the value 5 could be encoded in two bytes in both of these libraries - one byte to denote the type of value encoded (Varint), and the other byte holding the value 5 itself. This two-byte encoding is half of the four-bytes of space necessary to perform the same operation in the binary mechanism. FlatBuffers allows the programmer to specify values of uint_8 or uint_16 in the schema files to explicitly utilize fewer bytes for encoding, but this practice can make the resulting mechanism less modular, so 32-bit integers were allocated within the HPCMetaMorpheus FlatBuffers schema files.

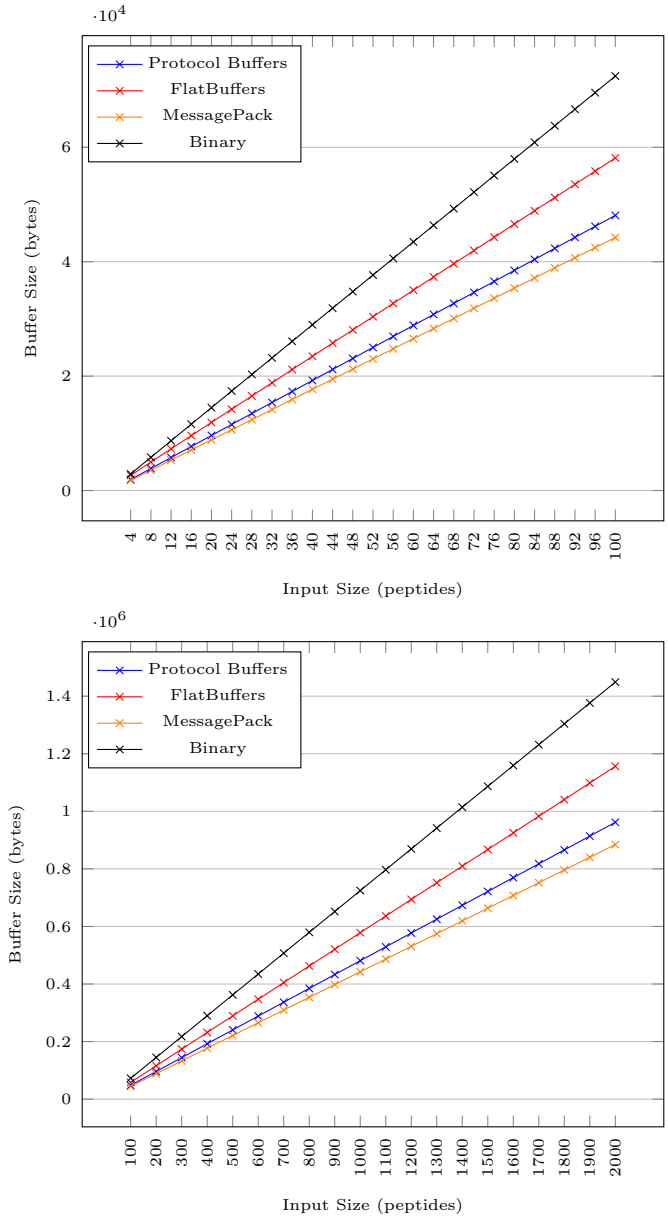

Figure 24: Serialized data buffer sizes, with smaller inputs (top) and larger inputs (bottom)

The binary mechanism's buffers are quite large because it packs the full-sized representation of each data value in the buffer; there is no dynamic shrinking of the encoded representation to the minimal number of necessary bytes. FlatBuffers also packs the full size of each value into the buffer, but the buffers it generates are generally smaller than those of the binary mechanism due to the fact that FlatBuffers can share metadata between separate

objects when multiple similar objects are serialized at once [14]. This means that as we serialize peptide after peptide, there is some data reuse, meaning the associated memory usage is amortized across multiple object buffers.

4.2 Transmission Time of Serialized Data

HPCMetaMorpheus is designed as a high performance computing software package, and is meant to be deployed on compute clusters in order to analyze mass spectrometry data in an efficient, parallelized manner. As described in Section 2.3, this parallelized execution requires that data be serialized and sent "over the wire" from node to node in the cluster. This implies that the time necessary to transmit a given data value from one node to another is a crucial element in determining the overall efficiency of the analysis. Several factors that contribute to the efficiency of this transmission are more or less constant, including the node processing efficiency, the network latency, and the network throughput. This means that the time required to transmit data between nodes is essentially a function of the size of that data buffer.

Within the context of this analysis, the size of the transmitted data is governed by the serialized data buffer sizes discussed in Section 4.1. In order to capture the impact these buffer sizes have on the performance of internode communication, measurements were taken of the transmission time required to transmit buffers of the given sizes. This is a crucial metric within our effort to compare the involved serialization libraries in the context of a high performance computing environment.

In order to measure the transmission time using varying sizes of data buffers, a parallelized C++ program was developed using Open MPI, and was deployed on the crill cluster at the University of Houston. For each buffer size recorded and presented in the previous section, this program generated a corresponding buffer of alphanumeric char values of the same

length. These buffers were created sequentially, with each one being sent from one node to another node, then transmitted back. This operation was repeated for a total of twenty-five times per buffer size generated, and the arithmetic mean of these twenty-five runs is presented below. The transmission time was captured on the original sending node by measuring the time between sending the initial message to the remote node and receiving the remote node's response. Therefore, the measurements presented in this section represent the round trip time (RTT) required to transmit a data buffer of each size shown in the previous section.

In Figure 25 below, the RTT measurements using buffer sizes associated with each library's serialized input size are presented. As the input size increases, the resulting buffer sizes increase, which in turn causes a gradual increase in the RTT values for all mechanisms. Across all input sizes, the buffer sizes produced by MessagePack are consistently transmitted the fastest, while those produced by the binary mechanism require the most time. For smaller input sizes, there is a clear distinction between the RTT values of the Protocol Buffers mechanism and those of the FlatBuffers mechanism, with Protocol Buffers achieving faster transmission times. However, for the range of input sizes between 500 peptides and 1500 peptides, their performance values merge and frequently overlap each other. After 1500 peptides, Protocol Buffers regains a sizable lead over FlatBuffers.

The measurements presented above show the expected result of the transmission time profiling. As discussed, the variation of transmission time between two nodes is primarily a function of the size of the data being sent. This phenomenon was observed throughout the measurements presented in this section; the mechanisms producing the smallest buffers resulted in the fastest transmissions, and vice versa.

As can be seen in the figures below, the time required to send and receive these data buffers is approximately two orders of magnitude smaller than the time required for serialization and deserialization. This means that within a single end-to-end process of serialization,

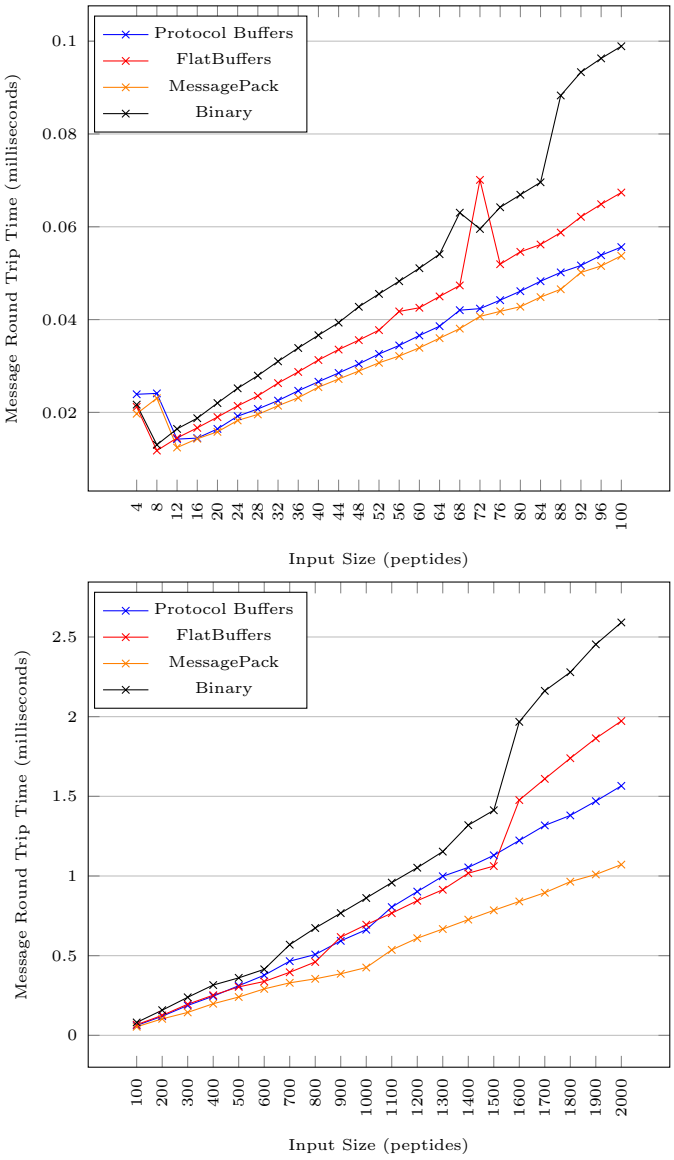

Figure 25: Round trip times resulting from transmitting data sizes resulting from each serialization mechanism using small inputs (top) and larger inputs (bottom)

transmission, and deserialization, the time required for transmission will make up a small proportion of the total time. However, this transmission time data would become much more of a consideration in cases where data is serialized once and then transmitted multiple times. In any case, a one percent disparity in performance is non-negligible in the area of high performance computing, so the data above are meaningful in the context of the overall

analysis.

4.3 Serialization Execution Time

In order to evaluate the execution time performance of the three new serialization mechanisms and the existing binary mechanism, we will separate the two stages of the overall serialization process and examine each stage's performance, as well as the overall execution time. In other words, we will separately discuss how each mechanism performs during the Pack() stage, during the Unpack() stage, and the overall sum of these two stages. This is because some mechanisms perform better in one stage and worse in the other, so dividing the evaluation in this way serves to communicate as fair and thorough an evaluation as possible of each mechanism's capabilities, and how they compare to the other mechanisms.

The measurements of the execution time required for each mechanism to execute the Pack() function is shown below in Figure 26. Across all input sizes, the serialization execution time performance scales linearly for all mechanisms. It can be seen that Protocol Buffers and FlatBuffers perform quite similarly across all inputs, and consistently require the most execution time to complete the Pack() operation out of the four mechanisms. MessagePack performs slightly better, consistently requiring approximately 85% of the time that Protocol Buffers require.

Interestingly, we see that the ad hoc binary serialization mechanism performs significantly better than the three newly-developed mechanisms in this packing case, from the smallest input sizes to the largest shown in Figure 26. The disparity between the binary execution time and that of the other libraries grows substantially as the input size increases, but remains roughly proportionally equivalent. The binary mechanism consistently requires around 35% of the time MessagePack requires for packing, and about 27% of the time the Protocol Buffers and FlatBuffers require for the same operation.

The efficient packing performance of the binary mechanism is most likely due to the minimal bookkeeping or dynamic analysis performed during the serialization process. To utilize the mechanism, the programmer invokes the appropriate packing function, and this function simply appends the binary version of the given data to the end of the buffer. There is very little metadata set into the buffer regarding the data being written, which means the

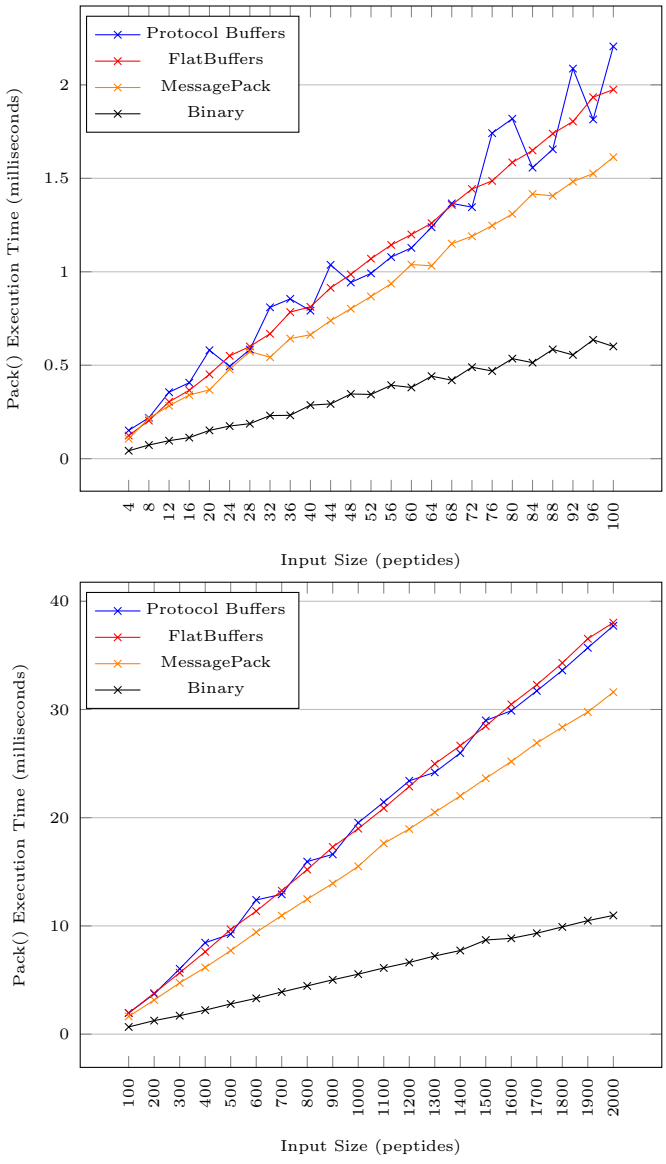

Figure 26: Pack() execution time for all mechanisms, with smaller inputs (top) and larger inputs (bottom)

performance overhead is quite small. The other mechanisms' serialization operation requires them to set headers or other metadata for each value being written [16, 14, 15], and in the case of Protocol Buffers and MessagePack, the size of integer values must be evaluated to determine the minimal bytes necessary to encode them. The inherent tradeoff in this case is that a more complex encoding scheme may cause longer packing execution times, but it will likely decrease the buffer size. This will, in turn, reduce the transmission time as was discussed in Section 4.2.

The recorded execution times required for each mechanism to perform the Unpack() function are shown in Figure 27 below, using the same input size scheme used in Figure 26. Similar to the Pack() execution times observed previously, the deserialization performance scales linearly for all mechanisms as input size increases. We see that the mechanisms are essentially split into two groups of two - with Protocol Buffers and the binary mechanism performing similarly, as well as FlatBuffers and MessagePack performing similarly. The former pair consistently requires the most execution time to complete the deserialization, with the latter pair performing slightly better. This pattern holds true across all input sizes, with the divide between the two pairs becoming increasingly wide as the input size grows.

The fast unpacking performance of the FlatBuffers mechanism is mostly due to the zero-copy memory access capabilities of FlatBuffers. This means that each element within the serialized buffer can be extracted directly from the buffer, and must not first be copied to another place in memory for processing, which greatly reduces deserialization time. FlatBuffers accomplishes this by storing a table of offsets with each serialized object, containing the relative address of each serialized value within that object's buffer. This is an example of another tradeoff, in that the additional metadata that FlatBuffers introduces into the buffer increases the packing time and the size of the buffer, but can significantly reduce the time needed to access fields within the buffer upon deserialization.

Protocol Buffers does not feature this in-buffer metadata, so it must parse and copy each value in memory before they can be accessed by the application, leading to its performance being the slowest out of the four mechanisms by a small margin. The binary mechanism uses the C++ function memcpy() to copy all values in the buffer bit by bit into their destination variables, and suffers from the associated performance penalty, which results in it being

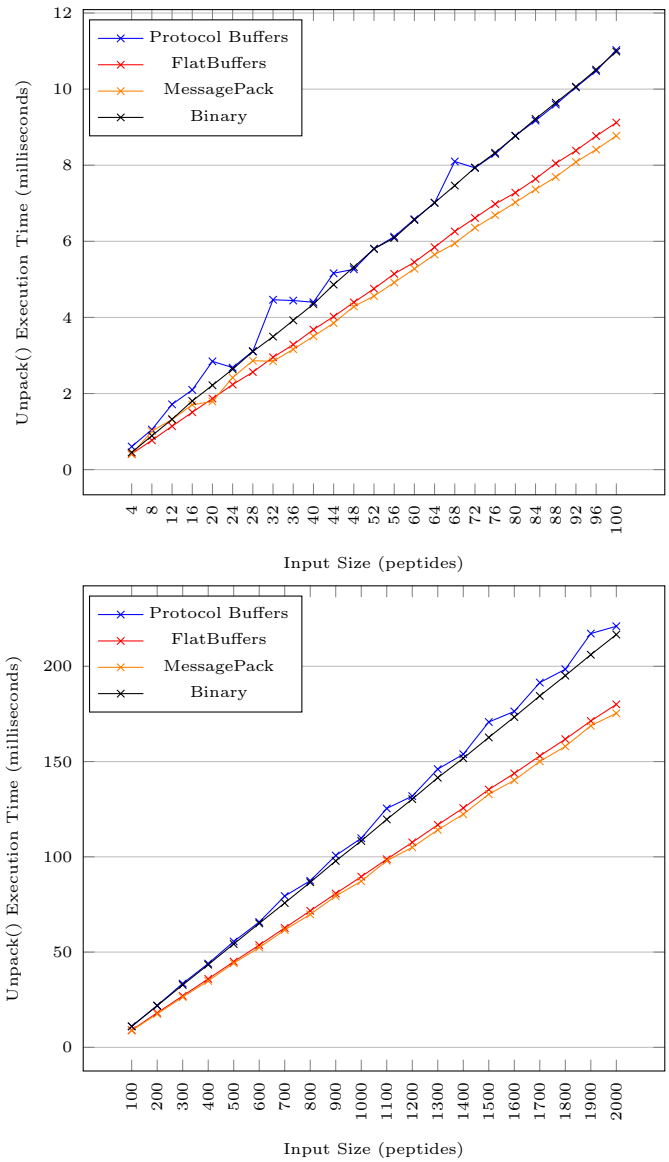

Figure 27: Unpack() execution time for all mechanisms, with smaller inputs (top) and larger inputs (bottom)

nearly the slowest unpacker. MessagePack is the fastest-performing mechanism despite also needing to parse and copy all data into another location in memory like Protocol Buffers. It is unclear based on the documentation why it is able to achieve the impressive performance it exhibits in these measurements.

The execution times required for each serialization mechanism to complete both the

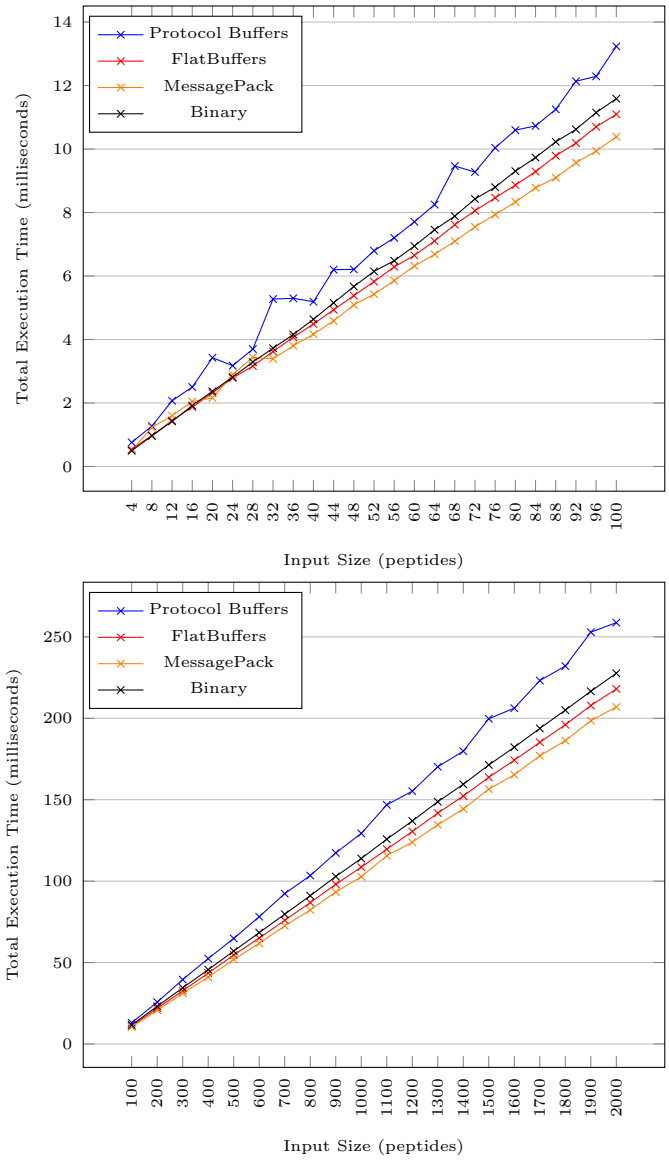

Figure 28: Total execution time for all mechanisms, with smaller inputs (top) and larger inputs (bottom)

Pack() and Unpack() operations is shown in Figure 28 above. The value of total time for a given input size and mechanism is found by summing the execution times of the associated Pack() and Unpack() measurements. This sum is a meaningful data point, because it reveals how each mechanism performs in terms of serialization in the overall end-to-end process. An interesting observation we can make in Figure 28 is that, despite exhibiting the fastest Pack() execution time by a significant margin, the binary serialization mechanism is consistently outperformed by MessagePack and FlatBuffers when it comes to total serialization time. This is because the execution times observed in the Unpack() measurements are approximately an order of magnitude larger than those observed during the Pack() measurements, meaning the binary mechanism's strong performance while packing was not enough to overcome its comparatively poor performance while unpacking. The Protocol Buffers mechanism requires the most execution time in total, which is due to it being among the slowest mechanisms in the Pack() operation as well as the Unpack() operation. MessagePack is consistently the fastest in terms of total time after being the fastest at unpacking and the second fastest at packing, with FlatBuffers close behind.

4.4 Standard Deviation of Serialization Execution Times

As was mentioned earlier in this chapter, each measurement of execution time presented in Section 4.4 is the arithmetic mean of ten measurements made using the given library and input size. This series of multiple measurements was performed in order to reduce the impact of outliers in the observations, and to capture a fair and thorough representation of how the libraries perform given various input sizes. After capturing these series of measurements, however, it was noted that some libraries performed more consistently than others. These consistent libraries would achieve similar execution times from run to run, while the time measurements for other libraries would vary more often, and would include more outliers.

This is notable because an application utilizing serialization may require that operations occur within a consistent time frame - due to a Service Level Agreement or some other reason - in which case the variation in time required for serialization becomes a crucial metric.

This section presents a quantitative comparison of this variation in the execution time required for serialization among the four mechanisms involved in this analysis. In order to capture and communicate this variation, the standard deviation of each series of ten serialization runs is used. The standard deviation can be defined as a measure of how dispersed a given dataset is in relation to the mean of that dataset [26], which indicates that it is an appropriate method to communicate the amount of variation across observed serialization execution times. The measurements for the serialization and deserialization operations will be discussed separately. A comparison of the standard deviations for the Pack() function are shown for various input sizes in Figure 29 below, and the analogous metric is shown for the Unpack() function in Figure 30.

We can see in the figures below that the binary mechanism generally exhibits the lowest standard deviation across all input sizes during the packing stage of the serialization process, meaning it is most consistent in its execution time. For the smaller inputs, the Protocol Buffers mechanism consistently has the highest standard deviation, although the standard deviation of MessagePack and FlatBuffers rises to meet that of Protocol Buffers as the input grows. We can also observe in the figures that the standard deviation has a tendency to increase as the input size increases. This is not true for the binary mechanism, however. As the input grows, the standard deviation of the binary mechanism appears to remain consistent. This success in scalability contributes to the increasing disparity between the standard deviation of the binary mechanism and that of the other mechanisms as input size increases.

The figures below show that, in general, the standard deviation of the execution times of the unpacking stage serialization is much higher than that of the packing stage. The relationship between the standard deviations of the various libraries is also quite different from the packing measurements. For very small input sizes, all libraries exhibit fairly similar standard deviations, but at an input size of approximately 55 peptides, the binary mechanism

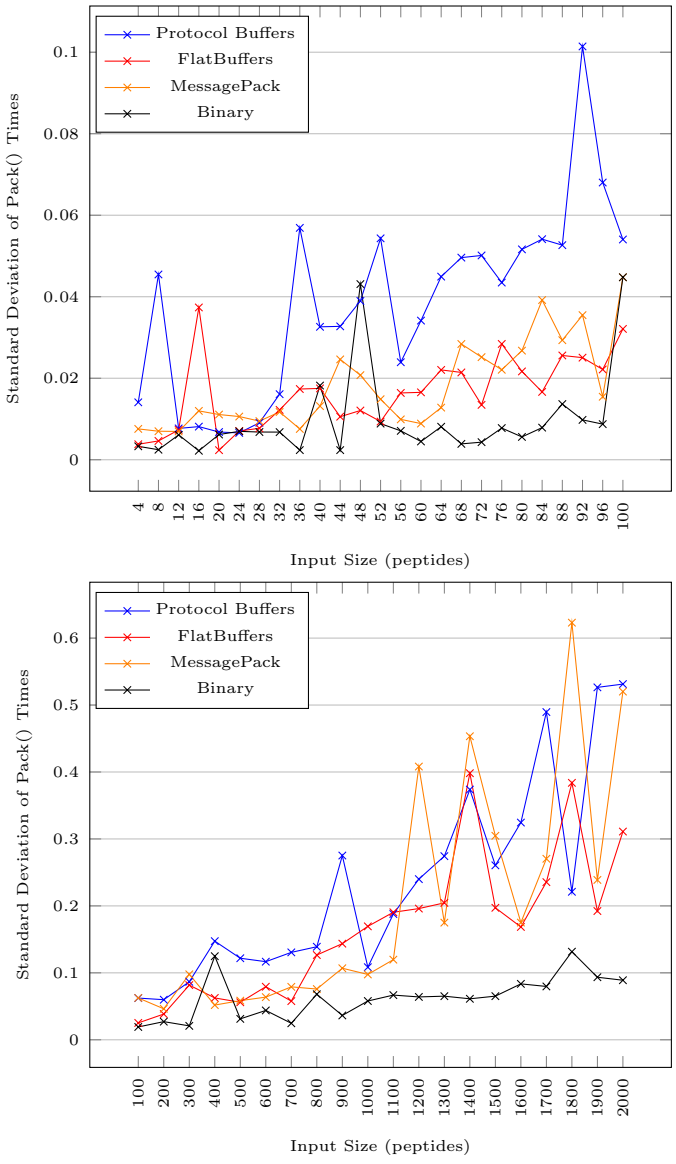

Figure 29: The standard deviation of the Pack() execution times for small inputs (top) and larger inputs (bottom)

begins to consistently show more deviation than the other libraries. In the bottom figure above, this disparity becomes much more pronounced as input size increases. Interestingly, the FlatBuffers standard deviation increases very similarly to the binary mechanism as the input grows large. The Protocol Buffers and Message Pack standard deviations are observed to increase until an input size of approximately 500 peptides, then level off as inputs continue

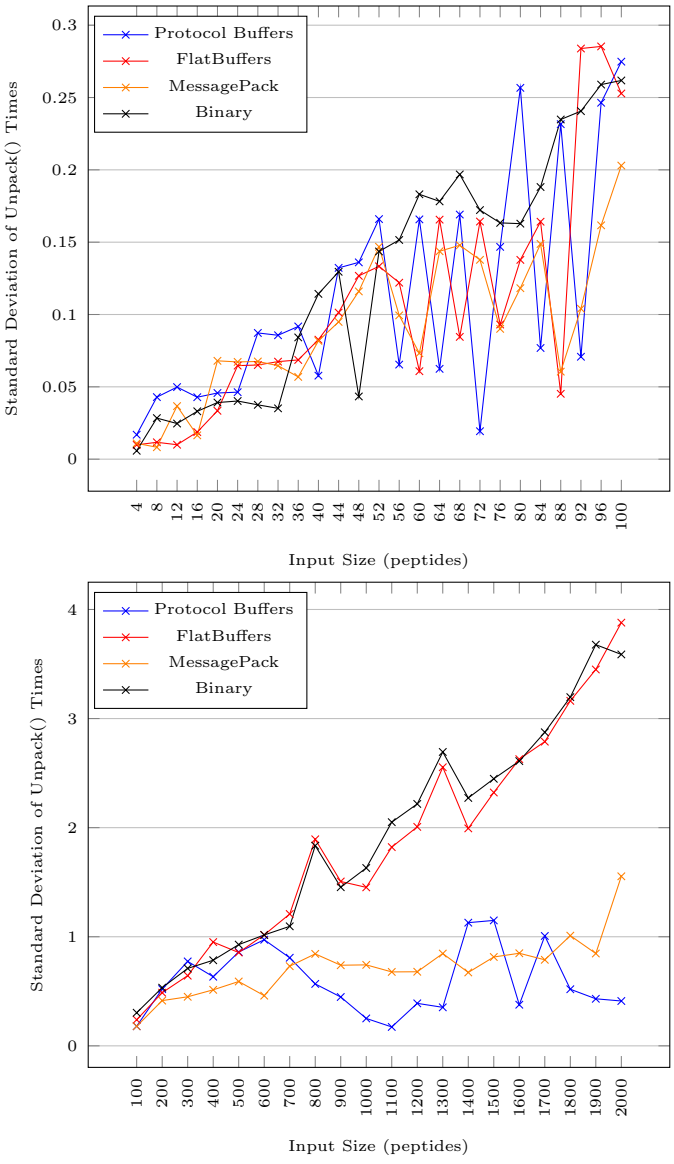

Figure 30: The standard deviation of the Unpack() execution times for small inputs (top) and larger inputs (bottom)

to grow.

Based on the observations discussed above, if an application using serialization requires that the serialization operations be performed in a consistent amount of time, the choice of which library to utilize becomes more critical as the input size increases. Although the binary mechanism performs consistently across all input sizes for the packing operation, its variation is among the highest for the large input unpacking operation. Inversely, although Protocol Buffers exhibit the highest standard deviation for the small input packing operation, it is arguably the most consistent performer during the unpacking operations, especially when given large inputs.

4.5 Size of Generated Source Files

As discussed in Section 2.2, two of the serialization libraries involved in this analysis require that schema files be written describing the data to be serialized - Protocol Buffers and FlatBuffers. These schema files must then be compiled using the compiler specific to each library - protoc for Protocol Buffers and flatc for FlatBuffers. The byproduct of this compilation process is a set of generated source files that provide the API that the programmer can then use to interact with the serialization library's utilities and manipulate the data to be serialized. The size of these generated files for each library is dependent on the quantity of the data values that are defined in the schema files.

Table 4 compares the size of the generated files for each of the four mechanisms analyzed, in terms of kilobytes as well as source lines of code. In the case of both metrics, the sizes of all generated files were added together, and the total is displayed here. Despite not generating any additional files, the MessagePack and binary mechanisms are included in the table to visually emphasize this absence, and to keep all comparisons consistent between all four mechanisms.

Table 4: Size of generated source code files in terms of kilobytes and source lines of code

Serialization Library	Size (KB)	Size (SLOC)
Protocol Buffers	390.38	8039
FlatBuffers	50.25	1114
MessagePack	0	0
Binary	0	0

It is clear from the figure above that Protocol Buffers generated much larger code files than FlatBuffers in this case, in terms of kilobytes as well as lines of code. More specifically, the total FlatBuffers file size is 12.9% that of Protocol Buffers in terms of kilobytes, and 13.9% the size of Protocol Buffers in terms of lines of code. This disparity is partially caused by the fact that Protocol Buffers generates two source files per schema file compiled, whereas FlatBuffers only generate one source file per compiled schema. The Protocol Buffers file sizes are more than two times larger than those of FlatBuffers, however, so this clearly does not account for the entire gap.

The remainder of the disparity can most likely be explained by the level of richness provided by the API supplied by each mechanism's generated source code. The FlatBuffers files are indeed smaller, but contain very little functionality other than getter and setter methods used in the creation and access of serialized FlatBuffers objects. On the other hand, each Protocol Buffers file is larger than its FlatBuffers counterpart, but delivers more robust functionality to the programmer. In addition to the basic getter and setter files, the Protocol Buffers generated files provide utility functions that can return the length of the data buffer, serialize using different formats, and parse from those different formats.

4.6 Hardware Resource Utilization

The process of capturing accurate hardware resource utilization data for each serialization mechanism required a refactoring of the program used to capture execution times. In the original program, each set of peptides for the given input size is created, serialized, and destroyed for each iteration. This is acceptable because only the period of serialization is measured in terms of time, and its duration is unaffected by the peptide construction stage. However, if this same process were to be used to capture hardware utilization, extraneous and irrelevant data about memory and CPU usage would be collected during the peptide construction stage - potentially invalidating our overall results. In order to prevent this, only one input size was used - a set of 2000 peptides. In the new program, these were created only once, after which they were serialized and deserialized. This created an initial period of irrelevant utilization data during the peptide construction stage, but the remainder of the data constituted a long stretch of valid utilization statistics.

In order to capture the hardware resource utilization of each library in terms of CPU and main memory consumption, a Python script was developed using the psutil Python module [41]. This module is capable of tracking the activity of a specific process based on its PID, and extracting its associated hardware resource statistics. Once developed, this script was executed during the process described above using each of the four mechanisms, and the results were written to various CSV files.

Upon inspecting and analyzing the data resulting from these hardware utilization measurements, it was observed that the CPU utilization reached nearly 100% for all four mechanisms, and remained there consistently for the entirety of the measurement period. On occasion, the CPU usage would spike to just above 100%. Since the serialization process is sequential and takes place on a single core, the values greater than 100% are most likely

due to extraneous noise captured by the monitoring script. In terms of memory, the performance results among the four mechanisms were also very similar, but did not portray the serialization process as being as resource-greedy as with the CPU utilization results. Across all mechanisms, the average memory utilization for the serialization process hovered around 2.5 GB, and the maximum memory utilization was approximately 3.2 GB. These values are presented for each library in Table 5 below, with the average and maximum values both presented for main memory and CPU utilization. Despite the extremely similar values among the CPU utilization results, they are presented in the table in the interest of thoroughness.

Table 5: Main memory and CPU utilization for each serialization mechanism

Mechanism	Avg Memory (GB)	Max Memory (GB)	Avg CPU (%)	Max CPU (%)
Protocol Buffers	2.527	3.287	99.953	100.900
FlatBuffers	2.481	3.265	100.003	100.900
MessagePack	2.481	3.216	99.944	100.900
Binary	2.452	3.279	99.972	101.000

4.7 Non-Quantitative Observations

The previous sections of this chapter have discussed and evaluated the serialization mechanisms involved in this analysis along various quantitative metrics. There are, however, multiple relevant criteria related to the mechanisms that are not precisely quantifiable. By their nature, discussions of these non-quantitative criteria are more subjective than those of the previous sections. In order to minimize this subjectivity, an effort will be made to only cite observable items in these discussions. The metrics analyzed here include the quality of the library documentation and the robustness of the serialization API provided by the library. Since these topics are primarily development-specific, and the binary mechanism

was developed prior to this work, it will be omitted from this discussion.

4.7.1 Documentation Quality

One of the more crucial metrics for a programmer investigating a new tool is the quality of that tool's documentation. It is important that the documentation not only be clear and detailed, but also be easily found by those needing to reference it. Protocol Buffers and FlatBuffers are both very successful in the latter regard. Upon a web search of the name of each of those libraries, combined with the word "documentation", the first search result in every search engine tested was the landing page for that library's primary documentation. This meant that the number of steps taken to access the information required was minimized.

On the other hand, accessing the MessagePack documentation requires many more layers of navigation, and is much less direct. A similar web search for MessagePack documentation returns the MessagePack webpage [8] as the first search result. This webpage contains no documentation, only information about which prominent companies utilize the library, and links to the GitHub pages of each programming-language-specific MessagePack implementation. Upon clicking on the C++ implementation link, the user is brought to a secondary webpage, which explains that documentation can be found through a link to the msgpack-c "wiki" page [36]. Once on the wiki page, the user could then navigate a series of menus to peruse the documentation. Overall, locating the relevant MessagePack documentation was much less straightforward compared to the other two libraries utilized in this analysis.

In terms of documentation quality, Protocol Buffers offers the most comprehensive and thoughtfully laid out information. The landing page gives a lengthy and detailed walkthrough on the construction of the .proto schema files, and there are also thorough tutorials available in several programming languages that walk the user through the entire serialization process, giving hints about potential pitfalls along the way. FlatBuffers also provides very

good documentation, beginning with information about defining the schema file, and including easy-to-find links to more detailed language-specific tutorials. The primary shortcoming identified in the FlatBuffers documentation was that its tutorial only presented trivial examples of serialization, not more complex examples like the nested serialization necessary for the objects within HPCMetaMorpheus. In order to decipher how to properly implement these more complex serialization tasks, the developer of the HPCMetaMorpheus serialization mechanisms was forced to search the example code files in the FlatBuffers GitHub repository for relevant examples, which was a time-consuming process.

The tutorials offered in the MessagePack documentation were less thorough than either of the above libraries. At the date and time of retrieval, the tutorial on the C++ wiki page only demonstrated the packing functionality, with the unpacking only described in a different webpage. This lack of cohesion made it more difficult to conceptualize the full end-to-end serialization process. Overall the MessagePack documentation was less direct to navigate to, and less thorough, than the other two libraries analyzed here.

4.7.2 Robustness of Serialization API

Each of the three libraries involved in this analysis provides some form of API that the programmer can utilize throughout the serialization process. In the case of Protocol Buffers and FlatBuffers, this API is offered through the generated code files resulting from compiling their associated schema files. When using MessagePack, there are no generated files or schema definitions, and the programmer accesses the API by including the msgpack.hpp header file as described in Chapters 2 and 3. In either case, through invoking the API, the programmer can employ functions responsible for packing, unpacking, and mutating data. The depth and variety of these functions can influence the ease of development when using the library; the more options a programmer has at their disposal, the more likely it is that

one of the options will smoothly fit the programmer's use case.

The Protocol Buffers API is quite robust, providing multiple functions capable of serializing to and from various data formats. For example, the programmer has the choice to serialize the data to a char array, a std::string value, or a C++ ostream. There are also accompanying deserialization functions capable of parsing serialized data from each of these formats. This diverse set of options gives the programmer the freedom to serialize to whichever format makes their execution the most straightforward. In addition, Protocol Buffers provides several helper functions that aid the programmer when handling data. For example, the ByteSizeLong() function is called on an object prior to serialization, and returns the size in bytes required to serialize that object. This is especially helpful when serializing to and from char arrays, since the programmer must keep track of the length of the data buffer.

The options provided by the FlatBuffers API are slightly more limited than those provided by Protocol Buffers. The most prominent limitation is the fact that FlatBuffers only allows the programmer to serialize to one type of data format - a buffer pointer. This format can then be manipulated into other formats, but this is often a less direct process than with Protocol Buffers. However, an example of flexibility that FlatBuffers provides is its options for the method in which objects are serialized to the buffer; they can be passed all at once to a constructor method, or they can be set one by one into an object by setter methods. The former method does not make the serialization process more performant, but it does simplify the programming process. Both of these options were used at various points in the HPCMetaMorpheus serialization mechanism development.

The flexibility of the MessagePack API is difficult to categorize, because in some ways it is very flexible, and in others it is quite limited. A characteristic of the API that provides a high amount of freedom to the programmer is the fact that no schema definition is necessary

in the serialization process. By calling a single function, the programmer can serialize and deserialize any type of object necessary, including C++ structs as used in HPCMetaMorpheus and described in Chapters 2 and 3. However, this freedom is restricted by the fact that the API is essentially limited to two functions: pack() and unpack(), each only responsible for serialization and deserialization. In addition, MessagePack only allows the programmer to serialize to and from the std::stringstream type. Not only is the singular nature of this characteristic limiting, it is also a much less convenient type to have to contend with - manipulating an instance of std::stringstream is a slow operation. In the development of HPCMetaMorpheus, this characteristic forced the developer to perform inefficient additional transformations.

5 Conclusions

The primary goal of the work presented in this thesis was to evaluate the performance of multiple serialization libraries within the context of a high performance computing (HPC) environment. The libraries selected for the analysis were Protocol Buffers [11, 31, 16], FlatBuffers [6, 29, 28, 14], and MessagePack [8, 30, 36, 15]. The methodology of this evaluation used a large HPC software package called HPCMetaMorpheus [20] as a testing environment, and involved developing a full implementation of the software package's serialization mechanism using each of the libraries being evaluated. This enabled performance metrics to be gathered using complete end-to-end runs of the serialization process, leading to a fair and thorough representation of each library's performance capabilities. As presented in this document, these three serialization mechanisms were successfully implemented and evaluated. The resulting measurements were then compared among the three libraries, as well as with the previously-developed binary serialization mechanism in use within HPCMetaMorpheus.

In terms of the size of the serialized data buffer, the previously-developed binary mechanism yielded the largest buffers, while Protocol Buffers and MessagePack consistently generated the smallest buffers primarily through their use of Varints [16]. These buffer size measurements directly correlated to the time required to transmit the data of each buffer to a remote node in a compute cluster and receive its response, meaning that MessagePack messages were sent the fastest and binary mechanism messages required the most time.

When discussing the execution time required for serialization and deserialization operations using each mechanism, the measurements were split into the packing stage, the unpacking stage, and the sum of these two stages. In terms of packing efficiency, the binary implementation was consistently the fastest by a large margin, due to the bookkeeping performed by the underlying libraries in the other three mechanisms. When unpacking, Protocol Buffers and the binary mechanism performed the slowest, with FlatBuffers and MessagePack

being most efficient. The combination of these two categories saw MessagePack as the overall fastest library, with the Protocol Buffers mechanism being the slowest. The standard deviation analyses performed on these execution times showed that the binary mechanisms showed the most consistent performance for small input sizes, while MessagePack and Protocol Buffers were most consistent for larger input sizes.

The files generated by compiling the schemas of Protocol Buffers and FlatBuffers were incongruous, with the Protocol Buffers files ending up to be much larger than those of FlatBuffers. Due to their lack of schema files to compile, neither MessagePack nor the binary mechanism produced any ancillary files. The results of the hardware utilization profiling show that all four mechanisms consume a comparable amount of main memory and CPU resources.

From the results of the evaluation detailed in this document, there are two trade-offs that we can identify in regards to the design of the internal workings of serialization libraries. First, if the designer of a serialization library chooses to include metadata within the serialization buffer, this will increase the size of that buffer and increase the time required to serialize due to extra bookkeeping. However, these initial penalties can mean efficient access to the serialized data in the buffer at deserialization time, which can yield tremendous performance benefits as we see in the case of FlatBuffers. The other trade-off witnessed in these evaluations involves the inspection of input data prior to serialization, with the goal of identifying ways in which the size of the serialized representation of the data can be minimized. This technique is used for integer values in Protocol Buffers and MessagePack, and while it successfully reduces the size of their buffers, it introduces extra processing time during the serialization phase.

The realizations about trade-offs above, along with the performance results presented in Chapter 4 of this thesis, constitute the primary contribution and impact of this work

to the field of High Performance Computing. Specifically, the observations made in this effort have the potential to help inform the selection of a serialization library for the use within an HPC environment. It should be noted that the measurements and performance comparisons observed here may differ slightly from environment to environment, and that the performance benefits of using one serialization library over another are heavily dependent on the workload of the system. For example, if a given workload has a high rate of message passing - and therefore a high rate of serialization - then the performance of the serialization mechanism will be more impactful than when using a workload with infrequent messaging.

5.1 Results

Upon holistically examining all observations presented throughout this effort, and considering the trade-offs discussed above, MessagePack has been identified as the serialization library best suited for this environment and use case. MessagePack performed better than all of its counterparts in all quantitative metrics, with two exceptions - the execution time of the Pack() operation and the standard deviation of the Pack() execution time. In alternative use cases, if the Pack() time metric is exceedingly crucial in terms of speed and consistency, the original binary serialization mechanism may be the preferred option. In terms of the non-quantitative observations discussed, the shortcomings of MessagePack in these categories are not egregious enough to disqualify the library from selection.

5.2 Future Work

The development of each of the serialization mechanisms in this work provided an opportunity to evaluate a set of popular serialization libraries within an environment in which they were previously unevaluated. However, this effort was also undertaken due to its byproduct of a fully-developed, efficient serialization mechanism within HPCMetaMorpheus. The

development of this serialization mechanism was a key milestone in the development of HPCMetaMorpheus, which is still in the process of being built.

Due to the fact that the results of this effort fit into the development of HPCMetaMorpheus, a path of future work could include completing the parallelized implementation of the software package, which involves the use of Open MPI [10]. Once that is completed, another opportunity for future work is to run additional performance evaluations using the serialization mechanisms developed in this effort, but within the entire parallelized proteomics analysis process of HPCMetaMorpheus. This avenue of proposed work would assess each mechanism within a highly-parallelized, distributed application. Finally, an exploration could be undertaken in search of ways to optimize the serialization mechanisms developed in this effort. The documentation of each library was studied with diligence, but it is possible that each library's API was left underutilized in these implementations, causing the maximum performance capabilities to be left unattained.

Bibliography

[1] GCC version 10.2 Documentation. URL: https://gcc.gnu.org/onlinedocs/10.2.0/. Retrieved January 18, 2022.

[2] Intel Corporation. *What is HPC?* URL: https://www.intel.com/content/www/us/en/high-performance-computing/what-is-hpc.html. Retrieved November 21, 2021.

[3] Boost Libraries Documentation. URL: https://www.boost.org/. Retrieved January 25, 2022.

[4] C++ chrono::steady_clock Documentation. URL: https://en.cppreference.com/w/cpp/chrono/steady%5C_clock. Retrieved December 18, 2021.

[5] External Data Representation (XDR) Documentation. *RFC 1014*. URL: https://datatracker.ietf.org/doc/html/rfc1014. Retrieved November 26, 2021.

[6] FlatBuffers Documentation. URL: https://google.github.io/flatbuffers. Retrieved November 21, 2021.

[7] JSON Documentation. URL: https://www.json.org/json-en.html. Retrieved November 26, 2021.

[8] MessagePack Documentation. URL: https://msgpack.org/index.html. Retrieved November 21, 2021.

[9] MetaMorpheus Package Documentation. URL: https://github.com/smith-chem-wisc/MetaMorpheus. Retrieved November 26, 2021.

[10] Open MPI Documentation. URL: https://www.open-mpi.org. Retrieved November 21, 2021.

[11] Protocol Buffers Documentation. URL: https://developers.google.com/protocol-buffers. Retrieved November 21, 2021.

[12] SUSE Zypper Package Manager Documentation. URL: https://documentation.suse.com/smart/linux/html/concept-zypper/index.html. Retrieved January 18, 2022.

[13] XML Standard Documentation. URL: https://www.w3.org/TR/xml/. Retrieved November 26, 2021.

[14] FlatBuffers Serialization Encoding. URL: https://google.github.io/flatbuffers/flatbuffers%5C_internals.html. Retrieved February 07, 2022.

[15] MessagePack Serialization Encoding. URL: https://github.com/msgpack/msgpack/blob/master/spec.md. Retrieved February 07, 2022.

[16] Protocol Buffers Encoding. URL: https://developers.google.com/protocol-buffers/docs/encoding#cheat-sheet. Retrieved February 3, 2022.

[17] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. URL: https://www.mpi-forum.org/docs/mpi-1.0/mpi-10.ps. Retrieved November 21, 2021.

[18] E. Gabriel et al. "Translating a Large Software from C# to C++: An Experience Report". In: *ISCA 30th International Conference on Software Engineering and Data Engineering* 77 (2021), pp. 1–12.

[19] Edgar Gabriel. *Crill Technical Information*. URL: http://pstl.cs.uh.edu/resources.shtml. Retrieved November 21, 2021.

[20] Edgar Gabriel. *HPCMetaMorpheus Software Package*. URL: https://github.com/PSTL-UH/HPCMetaMorpheus. Retrieved November 21, 2021.

[21] Edgar Gabriel. *Parallel Software Technologies Laboratory.* URL: http://pstl.cs.uh.edu/index.shtml. Retrieved November 21, 2021.

[22] Oracle Java Documentation: Serializable Interface. URL: https://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html. Retrieved November 26, 2021.

[23] J. Li. "HPS: A C++11 High Performance Serialization Library". In: *Cornell University Department of Physics* (2018).

[24] C. Lu and L. Huang. "Cross-Linking Mass Spectrometry: An Emerging Technology for Interactomics and Structural Biology". In: *Anal. Chem. 2018* (2018), pp. 144–165.

[25] K. Maeda. "Performance Evaluation of Object Serialization Libraries in XML, JSON and Binary Formats". In: *Second International Conference on Digital Information and Communication Technology and its Applications* (2012), pp. 177–182.

[26] National Library of Medicine. *Standard Deviation.* URL: https://www.nlm.nih.gov/nichsr/stats%5C_tutorial/section2/mod8%5C_sd.html. Retrieved January 29, 2022.

[27] Python pickle Module Documentation. URL: https://docs.python.org/3/library/pickle.html. Retrieved November 26, 2021.

[28] Wouter van Oortmerssen. *FlatBuffers: A Memory-Efficient Serialization Library.* URL: https://www.googblogs.com/flatbuffers-a-memory-efficient-serialization-library/. Retrieved January 25, 2022.

[29] FlatBuffers Version Releases. URL: https://github.com/google/flatbuffers/releases. Retrieved November 26, 2021.

[30] MessagePack C++ Version Releases. URL: https://github.com/msgpack/msgpack-c/releases. Retrieved November 21, 2021.

[31] Protocol Buffers Version Releases. URL: https://github.com/protocolbuffers/protobuf/releases. Retrieved November 26, 2021.

[32] A. Sumary and S. M. Makki. "A Comparison of Data Serialization Formats for Optimal Efficiency on a Mobile Platform". In: *6th International Conference on Ubiquitous Information Management and Communication* (2012), pp. 1–6.

[33] J. Vanura and P. Kriz. "Performance Evaluation of Java, JavaScript and PHP Serialization Libraries for XML, JSON and Binary Formats". In: *2018 International Conference on Services Computing* (2018), pp. 166–175.

[34] openSUSE Version 15.2 Release Notes. URL: https://doc.opensuse.org/release-notes/x86%5C_64/openSUSE/Leap/15.2/. Retrieved January 18, 2022.

[35] openSUSE Version 42.3 Documentation. URL: https://en.opensuse.org/Archive:42.3. Retrieved January 30, 2022.

[36] msgpack-c Wiki Page. URL: https://github.com/msgpack/msgpack-c/wiki/v1%5C_1%5C_cpp%5C_unpacker. Retrieved January 28, 2022.