Challenges in Converting a Large Scale Proteomics Application to Another Programming Language

by
Nicholas Roos Biddle

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

Master of Science

in Computer Science

Chair of Committee: Dr. Edgar Gabriel

Committee Member: Dr. Amin Alipour

Committee Member: Dr. Margaret Cheung

University of Houston
August 2020

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my thesis advisor, Dr. Edgar Gabriel, for allowing me to be part of this research project. His advice and guidance were incredibly important throughout this project and contributed greatly to its successful completion. Additionally, much of this advice was very practical to me as a beginning software developer and has already proved useful outside of this project.

I would like to thank Dr. Alipour for agreeing to be on my defense committee. I would also like to thank Dr. Margaret Cheung and the rest of the UH XL-MS group for helping me understand many of the concepts that are important to the software packages discussed in this project. Additionally, I would like to thank Guoting Qin and Pengzhi Zhang for providing me with experimental data.

Finally, I would like to acknowledge the love and support of my family and fiancé. Pursuing an MS would not have been possible without their continued support, encouragement, and patience.

ABSTRACT

Cross-linked mass spectrometry has been gaining popularity lately as a relatively cheap and versatile method for providing macromolecular structural data. However, the software required for matching the ion fragments produced during the mass spectrometry experiments presents a scaling issue that can lead to very long run times. The problem is that matching the spectra present in the mass spectrometry data requires a database search. A full database search is $O(n^2)$ in the number of entries in the database. Reducing the number of entries in the database can lead to inaccurate results. It is desirable to be able to perform a full database search as quickly as possible so that the database search is not such a bottleneck for these types of experiments.

Many applications exist for performing the spectra matching required for cross-linked mass spectrometry experiments. However, none of these applications is ready for a high-performance computing environment. It is desirable to provide a proteomics search software package that can be executed on a cluster of computers.

This project approaches this problem by converting an open-source proteomics search package from C# to C++, which is a more appropriate language for high-performance computing applications. As the program selected for this project is very large, this project only details the conversion of certain aspects of it. These aspects include file input and output functionality, unit test functionality, and providing functions and classes that exist in C# but are missing in C++. The converted functions and classes were evaluated using unit tests and execution time benchmarks. The unit tests were used to determine the correctness of the converted code, while the benchmarks were used to make a comparison between the original C# execution time and the converted C++ execution time.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

1.1 Motivation

Mass spectrometry is an analytical chemistry technique used to measure the mass-to-charge ratio of ions. These ions are typically fragments of larger molecules that have been broken into pieces. The mass-to-charge ratios produced are helpful in identifying and understanding the structure of large molecules. There are many proteomics search software packages available for identifying peptide fragments produced during mass spectrometry experiments, including: Kojak[16], StavroX and MeroX[17], pLink[18], ProteinProspector[19], xQuest/xProphet[20], and MetaMorpheus[4]. Of these proteomics search software packages only a few are open source and none are ready for High-Performance Computing. The peptide search aspect of the software presents a scaling issue as performing a full database search is O(n^2) with respect to the size of the database being searched. These software packages are generally designed to be run on personal computers or laboratory computers. Because of this, the search complexity can lead to prohibitively long run times and can hinder experimental progress. Different software packages approach this complexity problem in different ways, with varying degrees of success. Limiting the search space can lead to missing potential peptide matches, which can lead to inaccurate results. Full database searches yield more accurate matches but can take a prohibitive amount when performed on a single computer.

Modern high-performance computing methods offer opportunities for performance improvement with these tasks. Leveraging the power and parallel nature of cluster computing can make performing full database searches more desirable and thus improve search results. However, there are a couple of challenges to running this type of software on a cluster. Many

of these proteomics search packages are closed source, making it difficult to modify them to operate in a multi-node parallel environment. The open source search packages are not written with high-performance computing in mind and are often written in a language that does not natively support multi-node parallel processing frameworks. This project focuses on the conversion of a proteomics software package from C# to C++ so that it may ultimately be run on a Linux based cluster. The proteomics software package selected for this project is open-source and performs a full peptide database search.

1.2 Goals

Overall, the goal of this project is to help convert a proteomics search software package called MetaMorpheus from C# to C++. The conversion of the full package is a large project and out of the scope of an MS thesis, so the focus of this work is on performing the initial conversion of many of the classes and tests required by the software package. This includes converting several file IO classes and methods for different file types, writing several missing functions, converting many unit tests, and debugging the converted code to ensure it is operating correctly.

File input and output are important aspects of the proteomics search process. The MetaMorpheus software package uses .toml files to provide the mass spectrometry instrument configuration required for the proteomics search process. The C# version of the search software utilizes a library called Nett for this purpose. The goal with converting the .toml functionality for the C++ version is to find an alternative to the Nett package and to mimic the usage of it in the C# package. Finding a third-party C++ library to provide the .toml read and write functionality is only part of what is required for the conversion. It is also necessary to

write missing input and output functions using the third-party library structures to mimic how the Nett package is used in the original C# code.

The C# package also requires .mzML files to provide the actual mass spectrometry experiment data necessary for the peptide search process. Reading from and writing to .mzML files is necessary for this functionality. The C# code accomplishes this by using data serialization and deserialization functions to read the .mzML data into objects and write the mzML objects out to .mzML files during the execution of the program. This poses a challenge as the serialization and deserialization functionality is native to C# but not to C++. An alternative is needed for providing equivalent C++ .mzML input and output code.

Several options are available for providing .mzML input and output functionality, including: using a third party library, writing the input and output functions from scratch, or generating the appropriate code with a third party tool. In addition to selecting one of these options, additional methods are required to mimic the usage of the C# .mzML input and output. Ultimately, it is desirable that the converted C++ code reads and writes the .mzML files in the same way as in the C# version. Additionally, it is important to ensure the operations performed on the mzML data after it is read follow what is done in the C# version of the code.

The C# package also performs serialization and deserialization operations with classes that are not related to the .mzML data. The C# .mzML serialization and deserialization rely on a schema specified in an .xsd file, however this is not the case for the other classes being serialized and deserialized. A solution is required that allows for serialization and deserialization of generic data in the converted C++ code. It is important to understand the serialization and deserialization requirements in detail and in order to choose an appropriate

serialization and deserialization solution for the converted code. Several C++ third party libraries exist that provide serialization and deserialization functionality. After choosing an appropriate solution, it is necessary to ensure that the serialization process writes the same data to the desired file as the C# version, and that the deserialization process correctly reads the data from the desired file and creates the appropriate instances of the intended class.

Converting unit tests is also a goal of this project. The unit tests are important for testing the functionality of the converted code as well as testing the correctness of newly written functions. Converting unit tests for .mzML and .toml input and output are particularly important as the data read from and written to these file types must be correct in order for the application to produce the correct results. Overall, the goal with converting the unit tests is to ensure that the converted C++ code performs the same operations as the original C# code.

After the above goals were met, the converted code required evaluation. Both the correctness of the converted code and the performance of the converted code needed evaluation. The correctness of the converted code was mainly tested using the converted unit tests. Once the unit tests were operating properly, they would indicate if the functions being tested produced the correct results. Unit tests were used to test the .toml input and output correctness as well as the .mzML input and output correctness. Additionally, several benchmarks were performed to compare the execution time of the converted code with the original C# code. These benchmarks included testing the .toml read and write functionality, the .mzML read and write functionality, and several unit tests.

1.3 Organization

The background section of this paper begins on page 6 and discusses mass spectrometry background, the computational aspects of mass spectrometry, the differences between the C# and C++ programming languages, and an overview of high-performance computing. The contribution section begins on page 16 and discusses the contributions made to this project. The results section begins on page 29 and quantifies the impact of the contributions made. The conclusions section begins on page 40 and discusses the conclusions drawn from the conversion and testing of the code. Finally, the future work section begins on page 41 and discusses future work applicable to this project.

2. BACKGROUND

2.1 Mass Spectrometry

Proteomics refers to the large-scale study of proteins and proteomes, or the set of proteins produced by an organism. Understanding protein structure and protein-protein interactions is extremely important to understanding how these proteins function within an organism. Traditionally, protein structures are studied using X-ray crystallography, Nuclear Magnetic Resonance (NMR), and more recently using Cryo-Electron Microscopy[1]. These methods are quite powerful for obtaining high resolution three-dimensional protein structural information; however, they are limited by the required sample preparation procedures. X-ray crystallography requires that the proteins are crystallized, which can pose a problem for many proteins[4]. Both NMR and X-Ray crystallography require high protein concentrations, which can be difficult to attain. Because of these limitations, a need has arisen for complementary methods for structural elucidation. Mass Spectrometry based methods are well suited for providing information about protein structures and are gaining popularity. Collectively these methods are called structural mass spectrometry or structural MS.

Structural MS encompasses several mass spectrometry methods, including ion-mobility mass spectrometry, hydrogen-deuterium exchange, covalent labelling, native mass spectrometry, and cross-link mass spectrometry[1]. While not capable of providing complete protein structures, each of these structural MS methods is capable of providing valuable information regarding protein structure that can be used to complement structural data obtained from conventional methods (NMR, X-ray crystallography, and cryo-electron microscopy). The main advantages of these structural MS methods are their “versatility,

sensitivity, and throughput"[1]. In general, these MS techniques require less stringent sample preparation methods and are capable of working with lower protein concentrations than conventional methods like NMR and X-ray crystallography. Their sensitivity is particularly advantageous as MS methods can provide femtomole resolution. Recent advances in mass spectrometry technology and techniques have caused Cross-link mass spectrometry in particular to gain popularity as a complementary structural MS method because it is capable of capturing "protein-protein interactions from the native environment and uncovering their physical interaction contacts"[1].

Cross-link mass spectrometry, also known as XL-MS, is useful for structural validation, integrative modeling, and de novo structure prediction[1]. It can help illuminate the tertiary or 3-dimensional structure of proteins and protein complexes. It has been used successfully along with cryo-electron microscopy and computational modeling by Wang et al. to fully resolve dynamic structures of the human 26S proteasome[3]. In addition to resolving the structures, they detected dynamic states of the proteasome subunits Rpn1, Rpn6, and Rpt6 and identified several new proteasome-interacting proteins. Chen et al. used XL−MS to interpret the architecture of yeast RNA polymerase−TFIIF complex (TFIIF is a transcription initiation factor)[3]. XL-MS has been employed to characterize the distinct modules composing the Nuclear Pore Complex which is a massive transport assembly consisting of at least 456 distinct proteins[1]. XL-MS has also recently been used to extend the knowledge of the RNA polymerase structure[1] and help illuminate the α-helical structures within Human Serum Albumin protein[2].

As the name implies, cross-link mass spectrometry combines a chemical cross-linking step with mass spectrometry analysis. The first step of XL-MS is reacting the protein or

protein complex to be analyzed with a bifunctional chemical cross-linking agent. A cross-linking agent is essentially a chain-like molecule with two reactive ends and a "spacer of defined length"[2]. The selection of an appropriate cross-linking agent is important as its length gives valuable information required for structural analysis of the protein. When the cross-linking agent is introduced to a solution containing the protein, the reactive ends of the cross-linking molecule form covalent bonds to two functional groups of amino acid side chains within the protein or protein complex[2]. This effectively tethers the two amino acid residues together. Because the length of the cross-linking agent is known, this represents a unique way to measure the distance between the two amino acid side chains that bond to the cross-linking agent. During the cross-linking step, many cross-linking molecules are introduced and bond to many different areas within the protein or protein complex.

Once the cross-linking process has finished, the next step is to digest away the remaining protein structure, leaving only the cross-linking molecules and the functional groups to which they have formed bonds. The remaining peptide mixture can then be separated and analyzed using liquid chromatography and tandem mass spectrometry (LC-MS/MS)[1]. The distances spanned by the cross-linked molecules can be used as "distance constraints"[1] when validating, modelling, and even predicting the protein structure. In other words, the remaining cross-linked residues provide "molecular rulers" [2] that provide valuable information about the distances within the protein or protein complex when combined with modelling or structural analysis data. Once the mass spectrometry analysis is finished, a database search is required to identify the crosslinked peptides.

In order to understand the challenges and requirements of XL-MS, some discussion of the liquid chromatography and mass spectrometry step is required. Liquid chromatography

mass spectrometry or LC-MS is an analytical method that uses liquid chromatography to physically separate the mixture of cross-linked residues left over after the protein structure has been digested away. This is done so that the different crosslinked residues may be analyzed by mass spectrometry separately according to their masses. After this, mass spectrometry analysis is performed to ionize the molecules and separate them by their mass-to-charge (m/z) ratios.

It is possible that different molecules sent to the mass spectrometer may have very similar masses. It is necessary to be able to distinguish between these molecules of similar mass. Because of this, XL-MS typically employs tandem mass spectrometry for the mass spectrometry analysis. Tandem mass spectrometry, also known as MS/MS or MS^2 consists of two mass spectrometers that are coupled together. The first spectrometer ionizes the molecules and separates them by their mass-to-charge (m/z) ratios. Ions of particular mass-to-charge ratios can be selected and fragmented further to produce smaller fragment ions. These ionized fragments are then fed to the second mass spectrometer which again separates the fragments by their mass-to-charge (m/z) ratios. The secondary fragmentation allows tandem mass spectrometry to separate and identify ions with similar mass to charge ratios, thus making LC-MS^2 a powerful analytical technique and an excellent choice for the requirements of XL-MS.

The complex MS^2 fragmentation of cross-linked peptides has been one of the main challenges facing XL-MS analysis[1]. This makes identifying cross-linked peptides and assigning cross-link sites difficult. Much effort has been made to overcome this, and in recent years has led to significant advancements in XL-MS analysis[1]. Because MS^2 fragmentation of cross-linked peptides is unpredictable, specialized algorithms are required to identify both

of the cross-linked peptides. Additionally, the MS^2 data obtained represents an ensemble of protein conformities, meaning computational methods are required to deduce the different conformation states. This makes the analysis of the MS^2 data a computationally intensive process involving specialized algorithms and database searches.

2.2 Computational Aspects of Mass Spectrometry

Most peptide database searching platforms operate by comparing the experimental mass spectrometry data to a database of theoretical spectra[1]. This is a limitation of many XL-MS search programs. It is a problem because all possible peptide pairs must be considered, resulting in a large $O(n^2)$ search space. In other words, if the database contains *n* peptides, then the search space contains *n(n+1)/2* combinations, which is $O(n^2)$. Thus, the time complexity of searching the entire database is quadratic in the size of the database. As the number of peptides in sample increases, the time required for the database search increases polynomially. This can lead to very time-consuming searches as the databases can contain millions of entries[5]. Because of this, different methods have been developed to identify the cross-linked peptides with lower time complexity. Broadly speaking, these methods can be divided into two groups: (1) converting the cross-linked peptide pair search into sequentially searching for two peptides assisted by the specific cross-linkers, or (2) using heuristics to prefilter the possible cross-linked peptide pair matches and thus limit the search space[5]. There are benefits and drawbacks to each of these approaches.

Converting searching for a pair of cross-linked peptides to sequentially searching for two peptides with the help of specific cross-linkers is typically done by using MS-cleavable cross-linkers. MS-cleavable cross-linkers are cross-linker molecules that can be broken

during MS analysis[5]. Using MS-cleavable cross-linkers reduce the search space from O(n^2) to O(n). The main drawback to this method is that cleavable cross-linkers are not as readily available as non-cleavable cross-linkers[5].

Limiting the peptide-peptide search space using heuristics is a common approach used by several software packages including Kojak[18], StravoX[19], pLink[20], ProteinProspector[21], and xQuest/xProphet[22]. StavroX uses a precomputed list of possible cross-links to which precursor ion masses are compared[1]. xQuest/xProphet performs isotope-based candidate peptide prefiltering to minimize the number of permutations checked during the database search [1]. ProteinProspector and pLink treat the crosslinked peptides as single peptides with large modifications rather than as peptide pairs[1]. These heuristic approaches lead to faster search times, but, in general, a significant number of peptide-peptide pairs are excluded from the search and thus a significant number of results are missed.

A novel approach was taken by Yu et al.[5]. Their group developed a search tool named ECL2 that performs an exhaustive search for peptide pairs yet only requires linear time and space complexity and does not require MS-cleavable crosslinkers. ECL2 was developed from a predecessor application called ECL. While ECL required quadratic time complexity to perform the peptide pair search, ECL2 utilizes a unique additive scoring function to reduce the complexity from quadratic to linear[5].

The MetaMorpheus software package is a proteomics search platform capable of identifying both MS-cleavable and non-cleavable crosslinked peptides. MetaMorpheus uses an ion-indexing algorithm for efficient database searching. This means all fragment ions are indexed based on their mass-to-charge (m/z) ratio prior to searching[3]. An “open-mass” search is performed for the target spectra. This means the spectra are searched “against a

target and decoy database" [3] with no limit to the precursor mass tolerance. Potential matches found during the search are considered "candidate peptides"[3]. All candidate peptides found while searching are stored in memory and are paired with one another in order to find a mass that matches the original precursor ion fragment mass. If this process is successful, the resulting pair of peptides is considered a "crosslinked peptide spectrum match (CSM) candidate" [3]. The CSMs are then scored based on the sum of the count of the two peptides observed fragment ions plus any signature fragment ions if the cross-linker was MS-cleavable[3]. The CSMs are ranked according to their scores and assessed against a false discovery rate to determine which of the CSMs may be considered positively identified peptide pair.

The datasets obtained from the XL-MS process are typically large. There are "many known Post-Translational Modifications, and databases containing detailed information about such modifications are readily available"[4]. However, searching using these databases to identify Post-Translational Modifications or PTMs in a dataset can pose a problem. For example, one procedure called Global Post-Translation Modification Discovery, G-PTM-D, requires 3 steps: (1) a wide precursor mass tolerance database search is performed, (2) a "database augmentation step adds plausible localized PTMs to the corresponding protein entries in the search database" [4] for peptides whose mass difference corresponds to the mass of a known PTM, and (3) a tight precursor tolerance search is performed with the augmented database. None of these steps is trivial, and the first wide search can take "hours for modest sized datasets"[4].

MetaMorpheus was chosen for this project because it is open source, offers full peptide database search, and is compatible with both MS-cleavable and non-cleavable cross-

linkers. The MetaMorpheus library is written using the .NET framework. This poses a problem when trying to run the MetaMorpheus program on a cluster running a Linux operating system. .NET and C# were initially developed for a Windows environment. There is a framework called Mono available for running C# programs on Linux machines, but this still falls short when it comes to running C# programs on Linux based multi-node clusters. For parallel programming on a Linux-based multi-node environment, C, C++, and Fortran tend to be the most popular language choices. This is because these languages are low-level, fast, widely supported with many compiler choices, and offer support for parallel programming APIs like MPI. For these reasons and because of its many modern features, C++ was chosen as an alternative to C#, with the goal being to convert the MetaMorpheus library to C++ and run it on one of the clusters at UH.

2.3 The C++ Language

C++ was initially developed in the mid 1980's by Bjarn Stroustrup as an extension to the C language[9]. Since 1998 it has been a standardized language. The standard is maintained by the International Organization for Standardization. The initial goal of extending the C language was to add object-oriented capability, i.e., inheritance, encapsulation, and polymorphism. C++ is upwards compatible with C, meaning C++ builds upon C and can run most C code[8]. Additionally, C++ uses a "namespace" to prevent name collisions[6]. It is a strongly-typed, unsafe language [8] that allows for low-level memory manipulation. It supports manifest and inferred typing as well as static and dynamic type checking[8]. A drawback of C++ is that manual memory management is required.

C++ is a compiled language. The build pipeline for C++ includes preprocessing, compiling, and linking. In general C++ source files are compiled into object files. The object files are then linked together to produce an executable[10]. The compiler generally checks for language rules at compile time, while the linker ensures that all names are used consistently and have been correctly defined. Few, if any, checks are performed at run-time[10]. If run-time checking is desired, tests must be written into the source code. The compile-link-execute pipeline can lead to very fast performance if everything is optimized properly, making C++ one of the preferred languages for High-Performance Computing. It is also a portable language, meaning it is supported by many compilers across many platforms. This another important consideration when it comes to High-Performance Computing.

2.4 The C# Language

C# was also built as an extension of the C language. It is based on the Microsoft .NET framework. C# is a compiled language but with an important difference. Rather than being compiled directly to machine code, it is compiled to what is known as managed code. Managed code is code that is "managed by a runtime"[7]. For C#, the managed code is in Common Intermediate Language (CIL), and the runtime is the .NET Common Language Runtime (CLR). The CLR interprets the managed code and converts it to machine code using Just-In Time compilation[7]. Because of this, C# can be thought of as an interpreted language that is just-in time compiled as needed. Some advantages of this are that the CLR handles memory management, thread management, type safety, exception handling, and garbage collection [7]. These features offer many benefits; however, the additional overhead can cause performance to suffer. Though C# was originally designed for use on Windows systems, the

Mono project allows cross-platform compilation and execution. While this allows for executing C# programs on Linux based machines, it does not provide support for multi-node parallelism.

2.5 High-Performance Computing

High-Performance Computing generally refers to using clusters of computers to obtain a higher level of performance than is possible with a desktop computer through parallel execution. This requires that code is written in a way that allows for multi-node parallel processing. One of the more popular standards for multi-node parallel processing in called the Message Passing Interface (MPI). MPI is a standardization API for parallel and distributed computing. There a few notable MPI implementations including MPICH, LAM/MPI, and OpenMPI. OpenMPI provides a free, fully open source, production quality MPI implementation[11]. The core MPI library features native support for C, C++, and Fortran. These languages are the most commonly used for parallel and distributed programming for several reasons including: many HPC applications involve linear algebra and these languages have extensive mathematics libraries, and for linear algebra heavy applications features like garbage collection or run-time polymorphism cause unnecessary overhead.

3. CONTRIBUTION

3.1 C# to C++ Conversion

With an ever-expanding number of programming languages and frameworks, the conversion of large source code bases from one language to another is a problem that many have faced. With a large source code base, conversion entirely by hand can be prohibitive. This has led to a large volume of research and many patents[26][27][28][29] for automated source code conversion methods. The conversion process between languages is non-trivial and there are many challenges. For example, the languages may have vastly different libraries of natively supported methods, the languages may handle memory management and garbage collection differently, and the languages may require different structures in order for the code to be compiled and executed. Each of these problems presents a different set of challenges and can make producing working translated code difficult. However, this does not mean that automated code translation software does not exist. Tangible Software Solutions, for example, has produced many code language converters, including: a C# to C++ converter, a C++ to C# converter, a Java to C++ converter, and a Java to C# converter, to name a few[21]. The fact that there exists software designed to convert large code bases from one language to another indicates that this is a problem faced by many and that there is a market for solutions. Although software exists for this purpose, it is rarely able to produce working converted code beyond very simple examples that does not require some adjustment and debugging.

For this project, the code conversion process begins with the use of the Tangible Software Solutions "C# to C++ Convert" tool[21]. C# to C++ Convert is closed source and designed to automatically convert C# code to C++ code. It is used to initially convert the

MetaMorpheus package as well as the MzLib package to C++. The conversion of the MzLib package is necessary because MetaMorpheus relies on it for a great deal of its functionality. MzLib is a "library for mass spectrometry projects"[30]. It contains functionality for reading mass spectrometry data from and writing it to .mzML files, reading protein database files, reading modification files, and loading online protein databases[30]. The conversion software does a reasonable job when converting these packages but still is not capable of producing working C++ code. The converted C++ code provides a good starting point, but many changes are still necessary. These manual changes can broadly be divided into three categories: adjusting syntax, mimicking functionality that was present natively in C# but not in C++, and the use of third-party libraries where necessary. Overall, this means that many changes must be made to the converted C++ code and a good deal of debugging must be performed on the converted code.

The converted code provided by the C# to C++ Convert tool often provides nearly correct statements that require some adjustment before they can compile and run correctly. For example, Lambda functions are often converted incorrectly. Lambda functions are anonymous functions that can be defined quickly within other functions. The converted lambda functions often required some correction to their syntax. Occasionally, the converted code would supply incorrect object types and a new object would need to be created. For example, some methods require the use of the C++ `std::string`, but the object provided in the converted code was a C string. In this case, a `std::string` object must be created with the same contents as the C string. Additionally, because C# provides garbage collection, manual memory management in the C++ code is necessary. The converted code includes many comments where the converter did not provide the necessary delete statements. This

occurs most often with classes that have member variables consisting of other classes. The dynamic creation of these classes requires that the member variables be dynamically created as well, and thus the deletion of these classes requires the deletion of the member variable objects. The converter is not good at providing this sort of memory management and only leaves a comment stating that manual memory management is required.

When the converter encounters functions native to C# but not C++, it simply converts the syntax to C++ code, but it does not provide the missing functionality. This is encountered many times as C# natively supports many functions that do not exist in C++ or the versions that do exist in C++ provide slightly different functionality. Functions like the C# `ToArray()`, `Quantile()` calculator, `BitConverter()`, `EndsWith()`, and a `HeapSort()` for key-value pairs are not present in the C++ standard library and had to be created manually. A detailed discussion of the challenges faced follows to serve as guidance for subsequent students and work.

The C# `ToArray()` method is used in the original code bases to perform a mapping of several objects typically in a map structure to an array. This is generally encountered when the contents of an array need to be filtered according to some constraint and the remaining elements are the only elements that are required further on in the code. This functionality had to be mimicked in C++. Rather than use the basic array data structure provided in C++, the data structure `std::vector`, provided by the C++ standard library, is preferred because it is a sequence container that can hold arrays while also providing many functions for manipulating the array. These functions include dynamically resizing the array as necessary. Mimicking the mapping to an array or filtering an array functionality of the C# `ToArray()` method is approached by traversing the `std::vector` containing the original

objects and using an if statement to determine if each element meets the necessary criteria. If an element does meet the necessary criteria, it is pushed into another `std::vector`. Once this iteration is finished, the original `std::vector` is set to the filtered `std::vector`, thereby effectively mapping the required elements to the original array.

C# has a `Quantile()` method in the MathNet.Numerics.Statistics package that divides a distribution range into continuous intervals determined by a parameter generally called *tau*. There are nine variants for computing quantiles that are supported natively in C#. Of the nine variants, the default known as R8 is the only one used in the C# MzLib and MetaMorpheus code. Because of this, only the R8 quantile method is necessary in the converted C++ code. The equations for finding the R8 quantile, denoted $Q_i(p)$, are given below[22]:

$$Q_i(p) = (1 - \gamma)x_j + \gamma\, x_{j+1}$$

$$m = \frac{(p + 1)}{3}$$

$$j = np + m$$

$$\gamma = np + m - j$$

Here n is the sample size. An implementation of this is written in C++ and used to replace the calls to `MathNet.Numerics.Statistics.Quantile()` left over from the C# conversion.

The C# `BitConverter` class is used in the MzLib package. Specifically, the `BitConverter.ToSingle()` and `BitConverter.ToDouble()` methods are used to convert arrays of bytes to floating point values with either single or double precision respectively. As C++ lacks native support for this type of conversion, a `BitConverter`

class containing the `ToSingle()` and `ToDouble()` methods is written to mimic the C# version. For The C++ versions, `std::vectors` of unsigned chars are used in place of the C# byte arrays. The conversion is performed by creating another unsigned char array of size 4 for single precision or 8 for double precision. In the `ToSingle()` method, the first 4 elements of the starting unsigned char array are copied to the new unsigned char array while in the `ToDouble()` method the first 8 elements of the original unsigned char array are copied to the new unsigned char array. The `ToSingle()` method then casts the 4-member unsigned char array to a float and returns the value. The `ToDouble()` method casts the 8-member unsigned char array to a double and returns the value.

The C# `EndsWith()` method checks if a string ends with a given substring. As with the other functions already discussed, no C++ equivalent to this exists in the standard library. It is necessary to create a function that performs an equivalent operation. This is done by using the `string.compare()` method in C++. The arguments required by the `string.compare()` method are the starting index and ending index of the string as well as the desired substring. The function then compares the string from the start index to the end index to check if the characters are equivalent to those in the substring. If so, and there are no remaining characters, then the string ends with the substring and the `EndsWith()` function will return true. If not, the function returns false.

The C++ standard library does provide a method called `std::sort()` that supports sorting arrays and `std::vectors`. However, in the C# version of MzLib, there are occasions where key-value pairs are stored in separate arrays rather than some kind of map structure. A C++ sorting function is needed for this as no natively supported function was capable of sorting two arrays based on the values in one of the arrays. Because of this, a basic

heap sort function is implemented in C++. The heap sort algorithm was chosen as it has *O(nlogn)* complexity in the worst case. The heap sort function is designed to take two `std::vectors` and their size as arguments. One of the `std::vectors` contains the keys and the other contains the values, meaning they both have the same size. The heap sort function then sorts both `std::vectors` based on the elements in the keys `std::vector`.

3.2 File IO

The MetaMorpheus and MzLib packages require reading from and writing to several different file types. These include: .toml files, .xml files, .mzML files, and .xsd files, among others. Details are provided on the work necessary to handle some of these file types in the C++ version of the code.

3.2.1 Toml Files

Toml files are generally used to provide configuration data. Toml stands for "Tom's Obvious Minimal Language"[12] and was originally created by Tom Preston Werner[12]. It is designed to "map unambiguously to a hash table"[12]. Because of this, the syntax of toml files consists of key-value pairs. It is designed to use semantics that make it easily readable to humans. A desirable aspect of the toml specification is that it is open-source and accepts community contributions.

The MetaMorpheus and MzLib packages rely heavily on toml files to provide the mass spectrometry experiment configuration. The read and write functionality for toml files was originally provided in the C# code by a library called Nett[14]. The Nett library is also written using C#, so an alternative is required when converting MzLib and MetaMorpheus to

C++. This is accomplished by using a header-only library called TinyToml[23]. TinyToml is a C++ library for reading and writing toml files that supports the C++ 11 standard. It includes features that allow for arrays and tables to be read into and written form the C++ structures `std::vector` and `std::map` respectively. Using the TinyToml library requires refactoring the converted MzLib and MetaMorpheus code to use the new structures and methods in the TinyToml library as well as creating reading and writing methods for the .toml files.

3.2.2 MzML Files

The .mzML files provide the actual mass spectrometry data to both MzLib and MetaMorpheus. Reading and writing to .mzML files requires the XSD serialization classes. Using the generated .mzML classes and methods, the .mzML files can be read and deserialized directly to objects while the program is running. This is done based on the schema provided in the .xsd file used to generate the classes and methods for the mzML objects.

The file extension .xsd stands for XML Schema Definition. The .xsd files are used by MzLib to provide XML schemas for the mass spectrometry data contained in the mzML files. This means that the .xsd files are used to specify the structure of the objects to be used when reading in or writing out data. Several xsd schemas are specified and maintained by the Proteomics Standard Initiative[13] including those required for the mzML functionality in MzLib. The .xsd files can be used to generate the appropriate classes and methods to parse and serialize the mzML files. As mentioned previously, C# has support for binary and XML serialization and deserialization with the `System.Runtime.Serialization` package

while C++ has no equivalent. The C# XML serialization method serializes the "public fields of an object … into an XML stream that conforms to a specific XML Schema Definition language (XSD) document"[15].

3.3 Serialization and Deserialization

Serialization and deserialization of objects to files and vice versa are also important aspects of the code conversion. Serialization refers to the process of taking objects and their data and writing them to a specified file type, usually a binary, .xml, or ,json file. Deserialization is essentially the reverse of the serialization process. It involves reading a specified file type and placing the data read from the file into appropriate structures as the program is being executed. C# has serialization and deserialization support in the `System.Runtime.Serialization` package, while C++ has no native support for this functionality. Because of this, an alternative is necessary.

One of the main challenges during the code conversion process was deciding how to approach the conversion of the C# serialization and deserialization code to C++. The C# MzLib package utilizes code generated from xml schemas provided in several .xsd files. C# has native support for xml serialization and deserialization based on a given xsd schema. C++ has no native support for this functionality. A decision has to be made about how to approach creating the C++ equivalent to the generated C# code. Three possibilities were considered: using and debugging the code generated by an automatic C# to C++ conversion software, using the Code Synthesis XSD tool[25] to generate C++ code directly from the .xsd files, or using 3rd party libraries and trying to write C++ code equivalent to the generated C# code. Ultimately it was decided that both the Code Synthesis XSD tool[25] and a third party library

called the Cereal serialization library[24] are necessary to provide the serialization and deserialization functionality that is present in every scenario in the C# code.

Each of the considered solutions presents advantages and drawbacks. The Code Synthesis XSD tool appears to be the best option when an xml schema is given in an .xsd file. However, there are portions of code in both MetaMorpheus and MzLib where no schema is given, and objects need to be serialized or deserialized directly. Ultimately it was decided that using the Code Synthesis XSD tool[25] is the best approach for the serialization and deserialization based on existing xsd files and that a 3rd party library called Cereal[24] is the best approach for all other serialization and deserialization requirements.

There are several reasons for choosing Code Synthesis XSD as one of the serialization and deserialization solutions. Because Code Synthesis XSD generates C++ classes based on an xml schema provided in a given .xsd file, it can generate classes that are equivalent or very close to equivalent to the generated C# classes. Because of this, it should provide code that is functionally equivalent or close to functionally equivalent to the C# code. It is used to provide xml parsing and serialization functionality. The C# code relies on parsing and deserializing xml data files to objects and being able to serialize objects to xml files. These two functional requirements made Code Synthesis XSD a desirable solution to the choice of conversion methods. Finally, Code Synthesis XSD is an open-source tool, which is also desirable.

During the code conversion, it was noticed that not all of the serialization or deserialization occurred using objects generated by an .xsd schema file. This made the Code Synthesis XSD tool ill-suited for these tasks because the serialization and deserialization methods generated by the tool require the correct generated classes and methods in order to

work. The generated serialization and deserialization methods only work when dealing with classes that have been defined in the .xsd file used to generate the code. An alternative is necessary for cases where there were no classes generated from an .xsd file. The C++ Cereal[24] library is selected for this purpose. Cereal is a header only library with a permissive license that offers serialization and deserialization support for many file types including: xml, binary, and json.

Because C# offers native serialization and deserialization support, the converted code had to be changed substantially to use the Cereal library instead. The correct syntax is required to call the Cereal archive functions that perform the serialization or deserialization. More importantly, every class the requires serialization or deserialization needs to have two Cereal methods added to its public attributes. These functions specify how the data being serialized should be taken from the classes member variables and written to a file, and how to read data into variables and then call the class's constructor using those variables.

3.4 Testing, Debugging, and Determining Correctness

Once the converted C++ code has been adjusted manually and the necessary methods and third-party libraries are added, it is possible to compile the code. Once compiled, it is necessary to ensure that the converted C++ code is functionally equivalent to the original C# code. Fortunately, the C# versions of MetaMorpheus and MzLib contain many unit tests. These tests were initially converted using the C# to C++ Convert tool, however, as with the rest of the converted C++ code, the converted tests still contain many errors. Additionally, C# supports a unit testing framework that C++ does not, so some effort is required to put the C++ unit tests into form that would compile and execute. Overall, 66 unit tests have been

converted to C++ as a part of this project. These tests span a wide range of functions from testing that chemical formulas are correctly parsed to testing the read and write functionality of different file types.

The original C# tests are written using the unit testing framework called the NUnit Framework. The C# tests essentially consist of test functions inside of a namespace. Converting these functions to C++ unit tests requires that a "main" function be created and used to call all of the test functions one by one. Additionally, the tests require manual adjustments to use the new C++ methods and libraries used to accommodate missing functionality. Once this is done the test files could be compiled and executed.

After the compilation of the test files, it is necessary to execute them and ensure that the tests produce the expected values. This is often not the case. In the event that tests do not work correctly, the Gnu Debugger (gdb) is used to track down the errors in the converted code. This process led to many corrections.

3.5 Optimization

With much of the C# MetaMorpheus and Mzlib packages converted, it was noticed that there were opportunities to optimize the converted code. The converted code often contains many repetitive function calls, especially in the files dealing with .mzML functionality. These repetitive function calls are the result of the complex nature of the objects being used as well as the straight-forward nature of the C# to C++ Convert tool. If the conversion tool encounters a long series of function calls to return a value it would simply string those function calls together. If the next line in the code contained many of the same

initial function calls, the conversion tool makes no effort to optimize the number of function calls. For example, there were multiple lines of code that look as follows:

```
if (_mzMLConnection->fileDescription().sourceFileList() != nullptr &&
! _mzMLConnection->fileDescription().sourceFileList().get().sourceFile()
.empty() &&
! _mzMLConnection->fileDescription().sourceFileList()>sourceFile()[0]
.cvParam().empty())
```

Here the `fileDescription().sourceFileList()` methods are called three times in the if statement. This is unnecessary and repetitive. Instead, an object can be created and used to hold the `sourceFileList` as follows in order to eliminate the repetitive calls:

```
auto connection = _mzMLConnection->fileDescription().sourceFileList();
if (connection != nullptr &&
                !connection.get().sourceFile().empty() &&
                !connection->sourceFile()[0].cvParam().empty())
```

Much of the .mzML code has similar repetitive function calls like this. Reducing them was approached in the same fashion. Overall, this approach led to at least 314 fewer repetitive function calls in the mzML code. Of these 314 removed function calls, 22 were removed from the mzML read code and the remaining 292 were removed from the mzML write code. However, the total number of eliminated calls is difficult to determine as some of the occurrences are inside of for loops. These function calls would occur in each iteration through the loop, making the total number of eliminated calls dependent on the number of

iterations of the for loop. These for loops are often dependent on the data itself, so the total number of eliminated repetitive function calls is dependent on the data being operated on.

Another optimization involved the swapping of if statements and for loops. There is a section of mzML code that involved an if statement nested in a for loop, as follows:

```
for (int i = 0; i < length; i++) {
    if (is32bit) { …  }
    else { …  }
}
```

The `is32bit` variable here is a parameter that is passed to the function containing this nested for loop and if statement. The value of `is32bit` does not change, meaning it only needs to be checked once. Checking the value of `is32bit` in every iteration through the for loop is inefficient. Because of this, the if statement and for loop shown above can be inverted, giving the more efficient structure:

```
if (is32bit) {
    for (int i = 0; i < length; i++) { … }
}
else {
    for (int i = 0; i < length; i++)  { … }
}
```

This way, the boolean `is32bit` is only checked once before the operations performed in the for loop.

4. RESULTS

The execution time of converted C++ code was tested and compared with the execution time of the original C# code using the Parallel Software Laboratories Salmon server. This server has an Intel Xeon W3565 3.20GHz processor with 4 cores and 8 logical threads and contains 24GB of main memory. As the code conversion process for the MetaMorpheus library was not yet complete, executing the full version of the program was not possible. Because of this, the parts of the MetaMorpheus and MzLib packages involving file IO and serialization-deserialization were isolated and tested separately. In order to compare the results obtained using the converted C++ code to the C# code, the same isolation of file IO and serialization-deserialization functionality had to be performed using the C# libraries. For both versions, benchmark files were created to call the appropriate functions being tested. These benchmark files were used to test the toml read and write functionality and the mzML read and write functionality which involved the serialization and deserialization of mzML data. In addition to the file IO and serialization-deserialization benchmarks, the execution times of several MetaMorpheus test functions were also measured. In the converted C++ code, the execution times of these tests were measured directly in the test file. In the C# code, the tests had to be called from a main function. Because of this, these tests were added to another benchmarks file that contained a main function and were executed and measured by themselves.

The C++ code was compiled using version 8.2.0 of the g++ compiler from the Gnu Compiler Collection. For all but two of the tests, the g++ optimization flag -O3 was used when compiling in order to provide the highest level of compiler optimization. However, the

two tests shown in Figures 4-7 and 4-8 required that the lowest level of compiler optimization was used, so the g++ -O0 flag was used.

Because C# was designed to be compiled and executed using Windows systems, an extra step was required to compile and execute the C# on the Linux server. The DotNet package was installed on the server, enabling the code to be compiled with the .NET command line interface. Once this was done, the `dotnet` command was used to compile and link the C# code and provide an executable.

For both the C# code and the converted C++ code, each measurement of execution time was performed 5 times with the average over all 5 executions computed for each test. The results are presented below.

4.1 Toml read and write tests

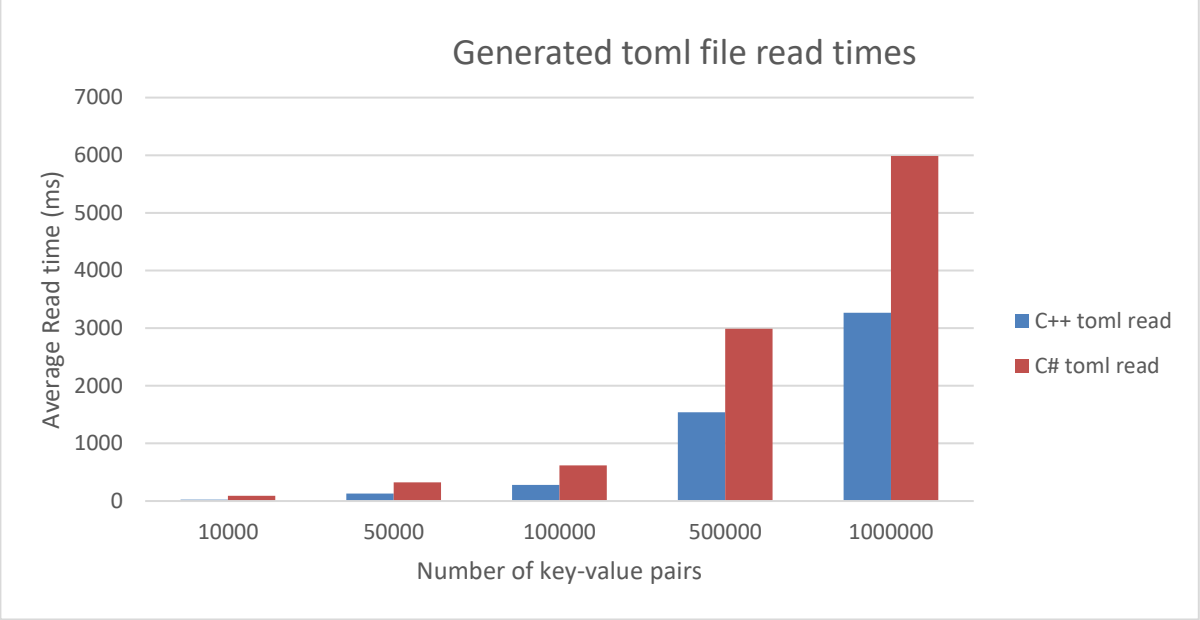

Figure 4-1: The average read time for toml files containing an increasing number of key-value pairs with both the C++ code and the C# code.

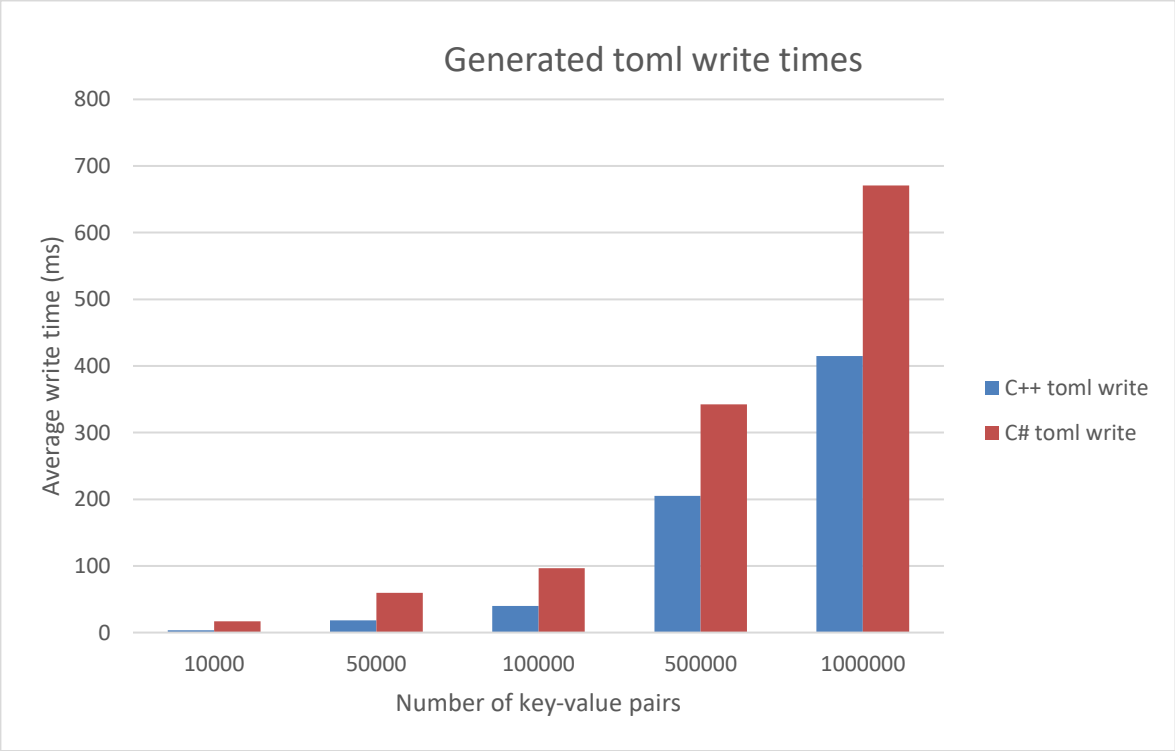

Figure 4-2: The average write time for toml files containing an increasing number of key-value pairs with both the C++ code and the C# code.

Figures 4-1 and 4-2 show the average read and write times respectively of increasingly large toml files. The toml data used to test the execution times shown in Figures 4-1 and 4-2 was generated before the read and write tests. The generated data was used to test read and write functionality for increasingly large toml data sizes. In total, five toml data sizes were tested containing 10000, 50000, 100000, 500000, and 1000000 key-value pairs.

It is clear that both the read and write times are faster in the C++ versions of the code, often nearly twice as fast. The C++ version of the code was able to read and write toml files in about half the time required by the C# version even as the files increased to a large size. The data taken in this test shows how the toml read and write functionality scales with the number of key-value pairs in the toml data. However, it is important to note that the toml

files required for mass spectrometry peptide fragment searches are often within 1-2kb. Because of this, four existing toml files were used to collect read and write execution times. This data is presented below in Figures 4-3 and 4-4.

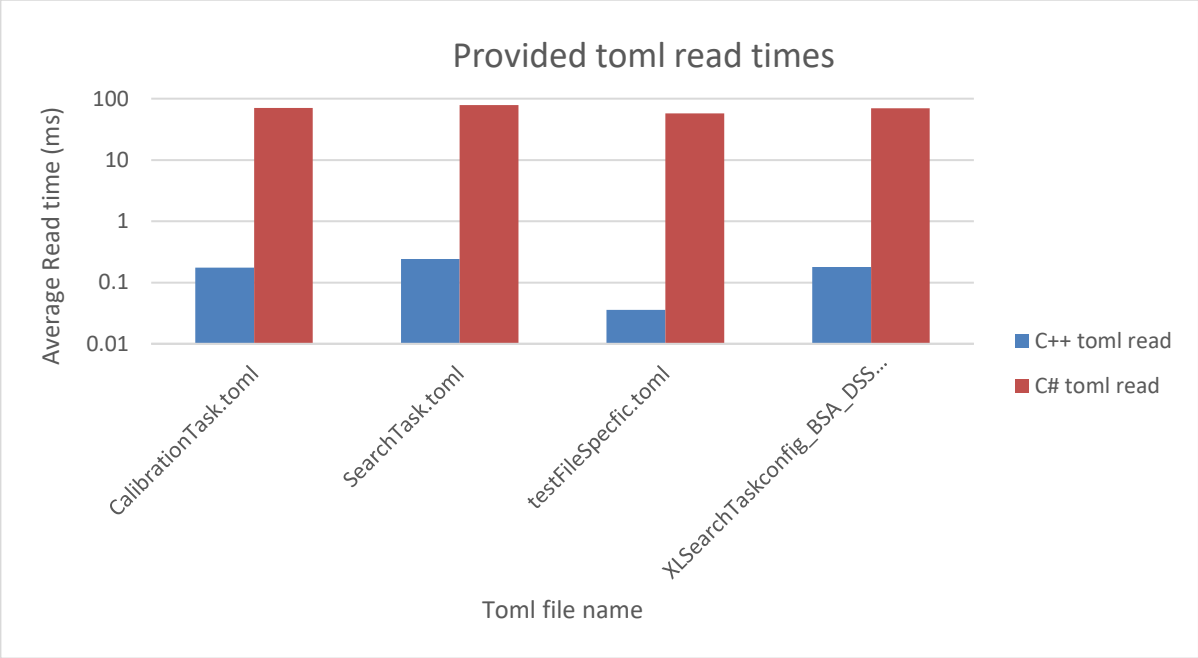

Figure 4-3: The average read times for four toml files contained within the MzLib and MetaMorpheus packages with both the C++ code and the C# code.

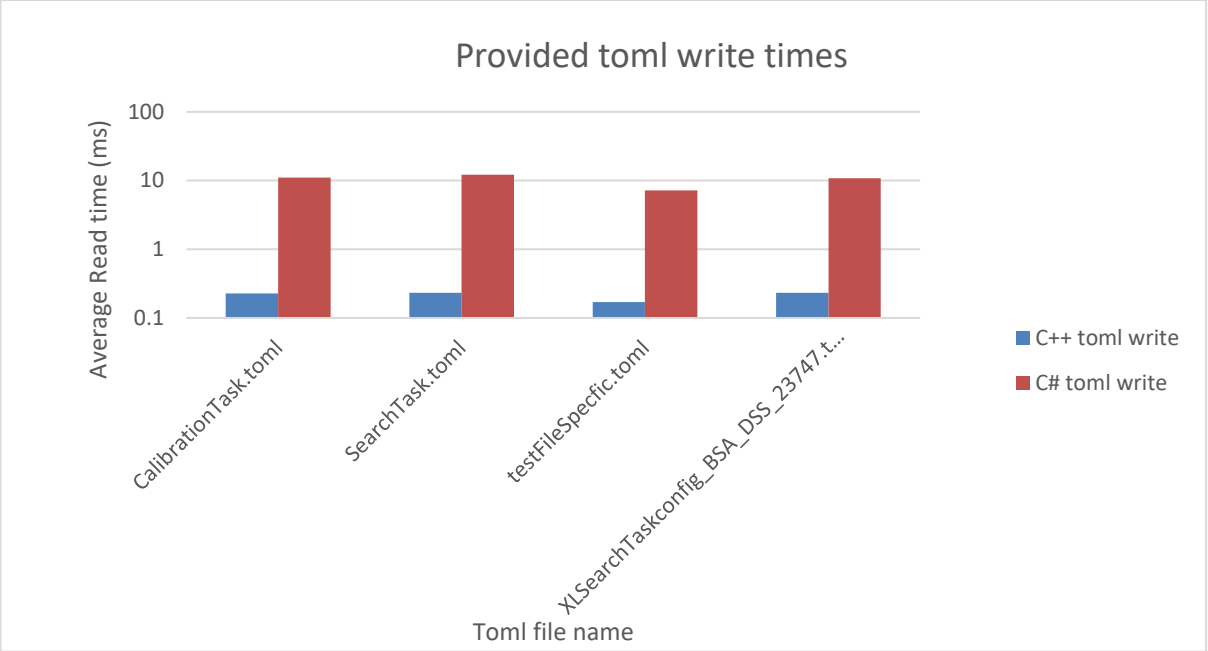

Figure 4-4: The average write times for four toml files contained within the MzLib and MetaMorpheus packages with both the C++ code and the C# code.

Figures 4-3 and 4-4 show the read and write times of real world toml files. These four files were chosen because they existed as part of the test suite or were part of actual mass spectrometry peptide fragment search experiments. The four files chosen are: CalibrationTask.toml, SearchTask.toml, testFileSpecific.toml, and XLSearchTaskconfig_BSA_DSS_23747.toml. These files illustrate the different uses of the toml files when it comes to providing configuration data. The CalibrationTask.toml file provides configuration data for a calibration task. Similarly, the SearchTask.toml provides confiuguration data for a peptide search task. The testFileSpecific.toml file is used to specify a specific protease type. The XLSearchTaskconfig_BSA_DSS_23747.toml is an actual experimental toml file generated for a crosslinked search task using Bovine Serum Albumin, meaning it is an example of a typical toml file required for an actual experiment.

The C++ toml read and write functionality performed much faster than the C# toml read write functionality in every case for the existing toml files. In fact, the C++ code

operated so much faster than the data had to be plotted using a logarithmic scale for the Y axis. The converted code provides a clear performance improvement over the C# version when it comes to reading and writing toml files.

4.2 MzML Read and Write Tests

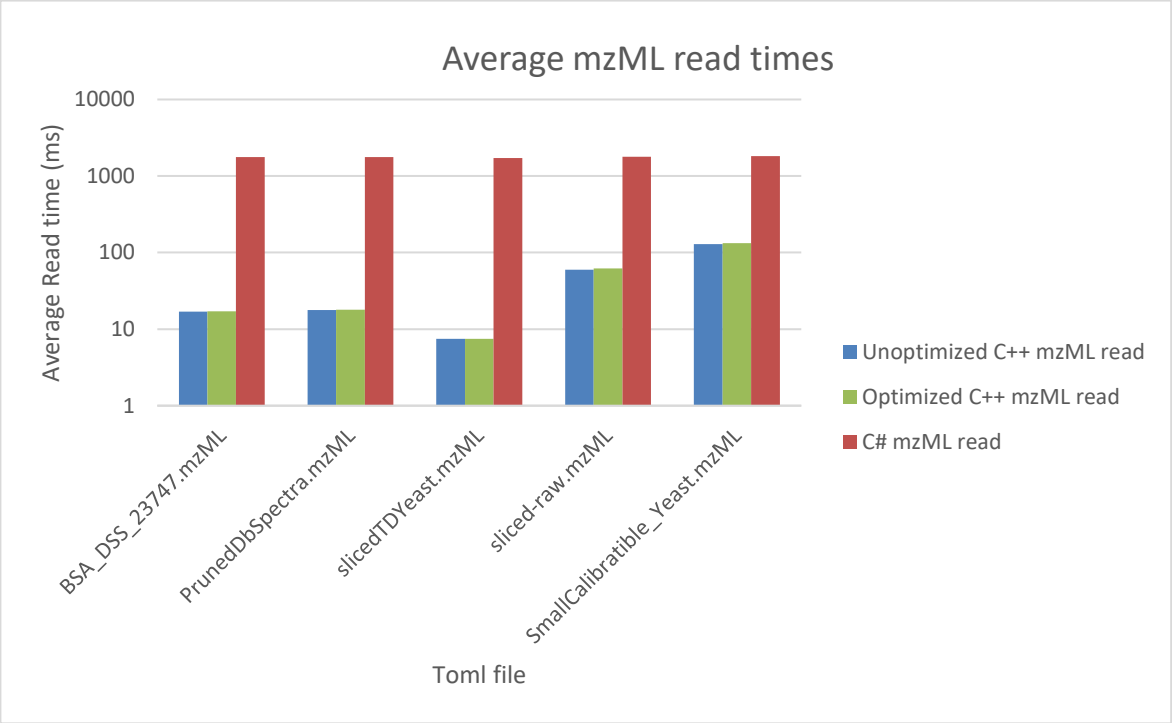

Figure 4-5: The average read time for mzML files contained within the MzLib and MetaMorpheus packages for the unoptimized C++ code, the optimized C++ code, and the C# code.

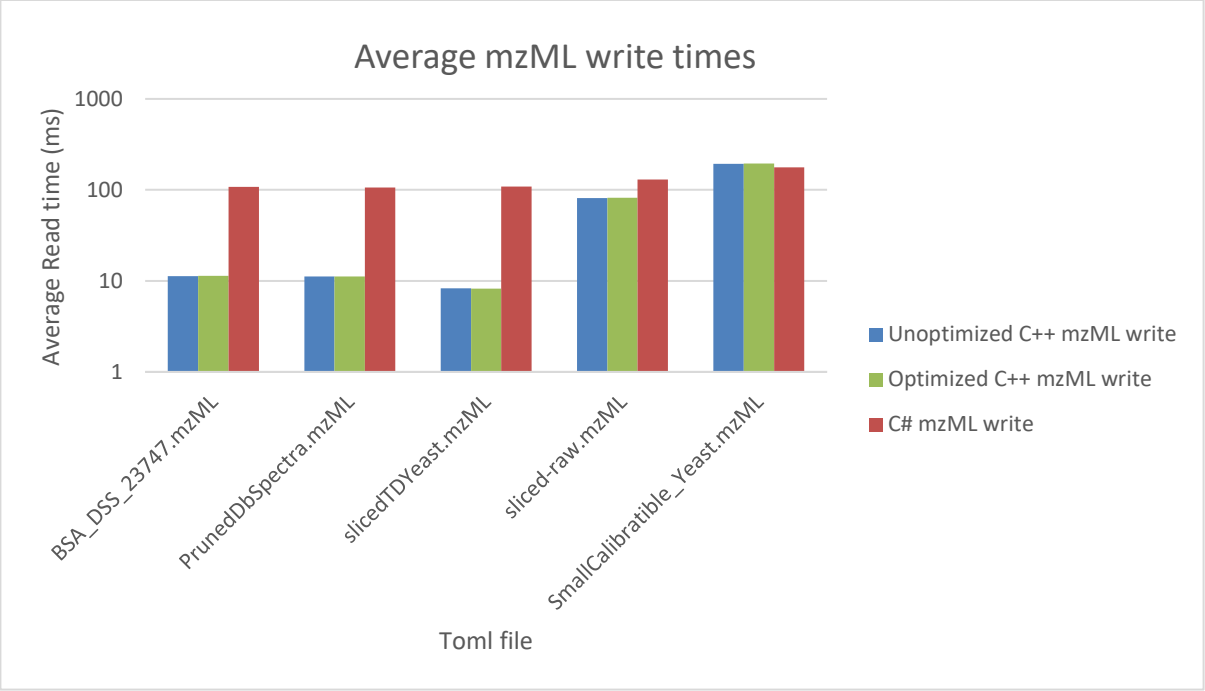

Figure 4-6: The average write time for mzML files contained within the MzLib and MetaMorpheus packages for the unoptimized C++ code, the optimized C++ code, and the C# code.

Figures 4-5 and 4-6 show the average read and write times of mzML files for both the converted C++ code and the C# code. The converted C++ code was tested both with and without the optimizations discussed in section 3.5. The results show that the converted C++ code is faster than the C# code for all but one of the mzML write tests. At this point, it is not clear why the C# code performed slightly faster with the SmallCalibratible_Yeast.mzML file. Further investigation into this is necessary. It is interesting to note that the optimized C++ code did not show any noticeable advantage over the unoptimized C++ code. This may have been due to how the C++ code was compiled and executed when performing these tests. For these tests, the C++ code was compiled with the Gnu compiler, g++, using the -O3 compiler optimization flag. Because of this, it is possible that the compiler was able to optimize the compiled code in such a way that the reduced redundant function calls and swapped if statement and for loop discussed in section 3.5 did not matter. In order to test this, the

optimized and unoptimized converted C++ code was tested again using the -O0 compiler optimization flag. The results of these tests are shown in Figures 4-7 and 4-8.

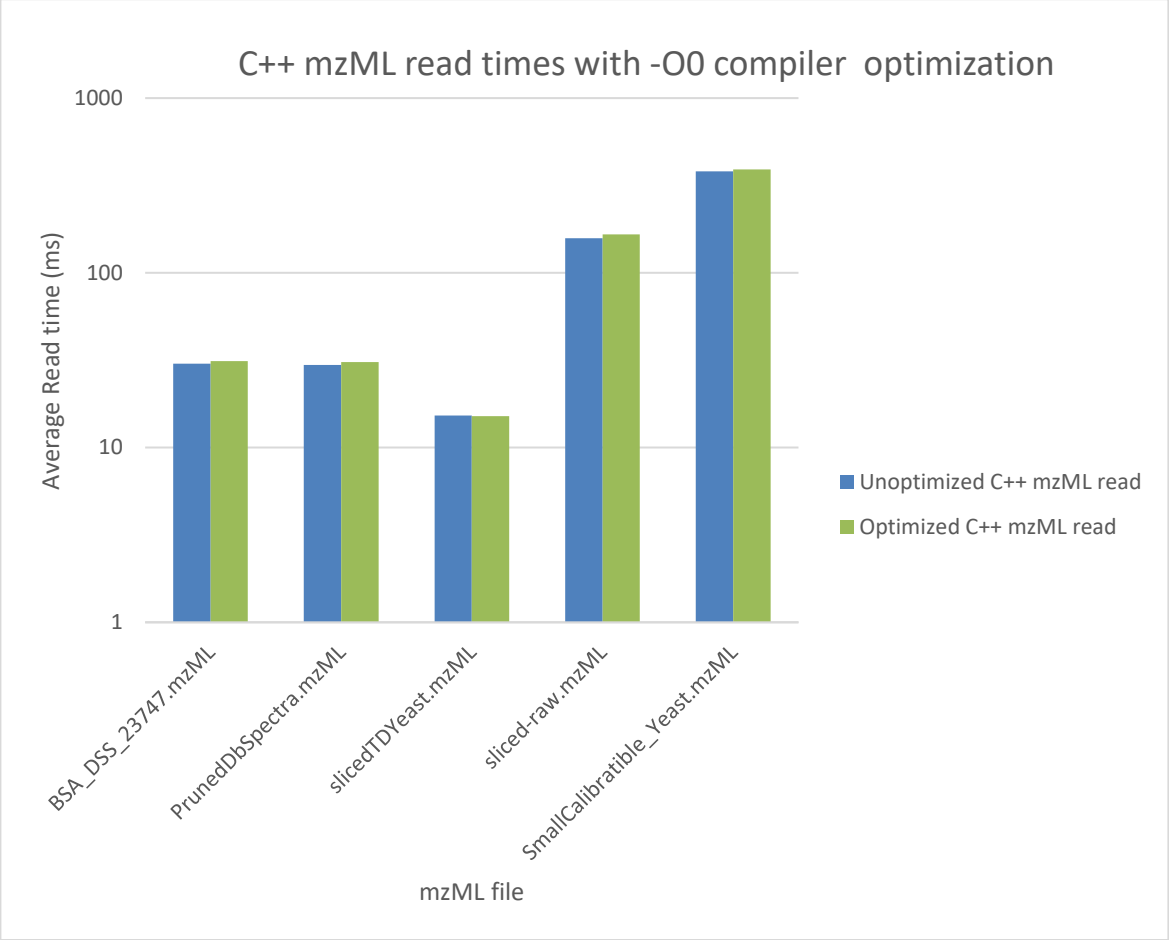

Figure 4-7: A comparison of the read times of five files contained within the MzLib and MetaMorpheus packages with the unoptimized C++ mzML code and the optimized C++ code when using the -O0 gcc compiler optimization flag.

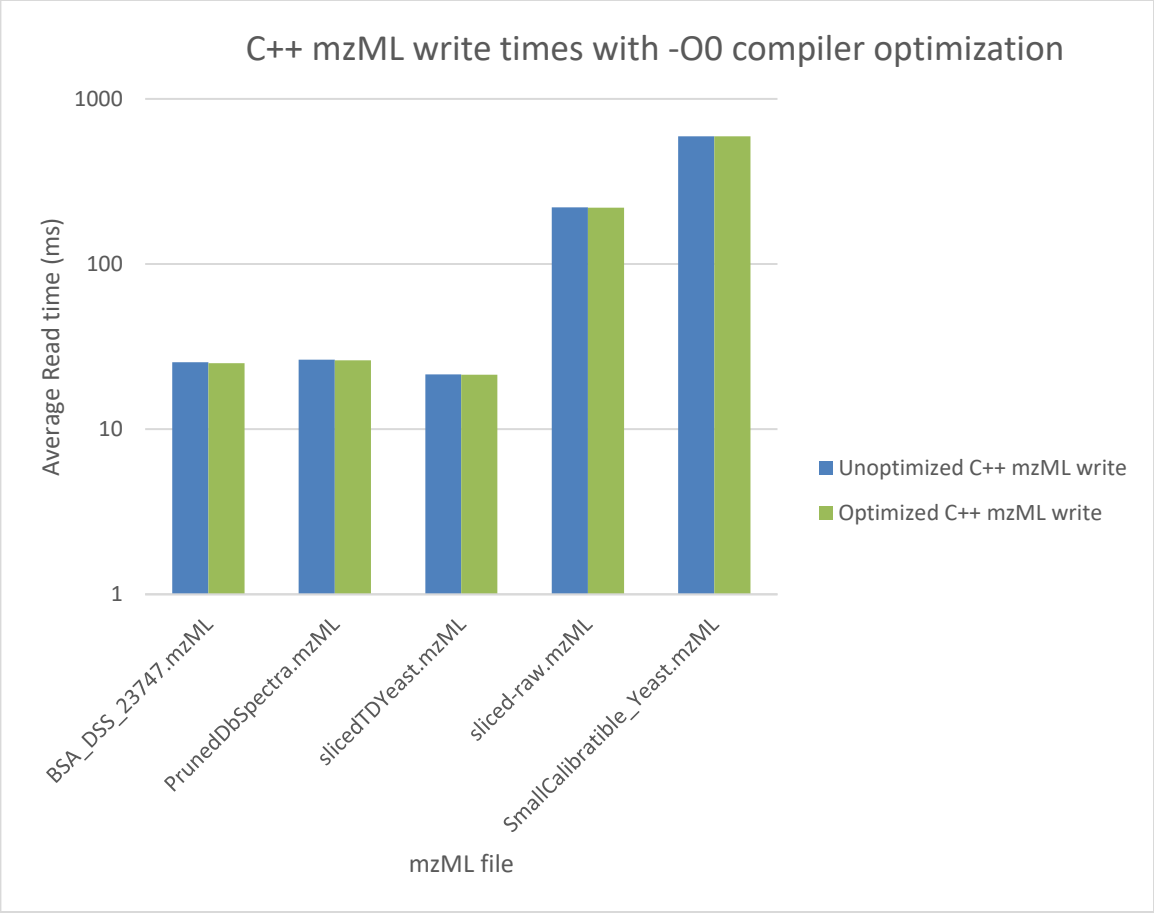

Figure 4-8: A comparison of the write times of five files contained within the MzLib and MetaMorpheus packages with the unoptimized C++ mzML code and the optimized C++ code when using the -O0 gcc compiler optimization flag.

Figures 4-7 and 4-8 show the execution times of the optimized and unoptimized converted C++ mzML functionality when using the gcc -O0 compiler optimization flag. The mzML read functionality relies on serialization of mzML data to C++ objects while the program is running. Overall, there was not much difference between the unoptimized and the optimized C++ mzML read functionality code. The unoptimized mzML read code performed slightly better than the optimized mzML read code for 4 of the 5 files, but the difference was negligible, generally less than 0.5ms. It is somewhat unexpected that the optimized mzML read functionality performed slightly slower than the unoptimized mzML read functionality. However, as discussed in section 3.5, a majority of the optimization changes occurred in the

mzML write functionality. Of 314 changes made to the mzML code, only 22 were made to the mzML read methods. As expected, the optimized mzML write tests performed slightly faster than the unoptimized ones. However, as with the mzML read tests, the differences between the optimized and unoptimized mzML write tests are negligible. This is shown in Figure 4-8, the optimized C++ mzML write code performs slightly faster for each of the 5 mzML files tested.

4.4 MetaMorpheus Tests

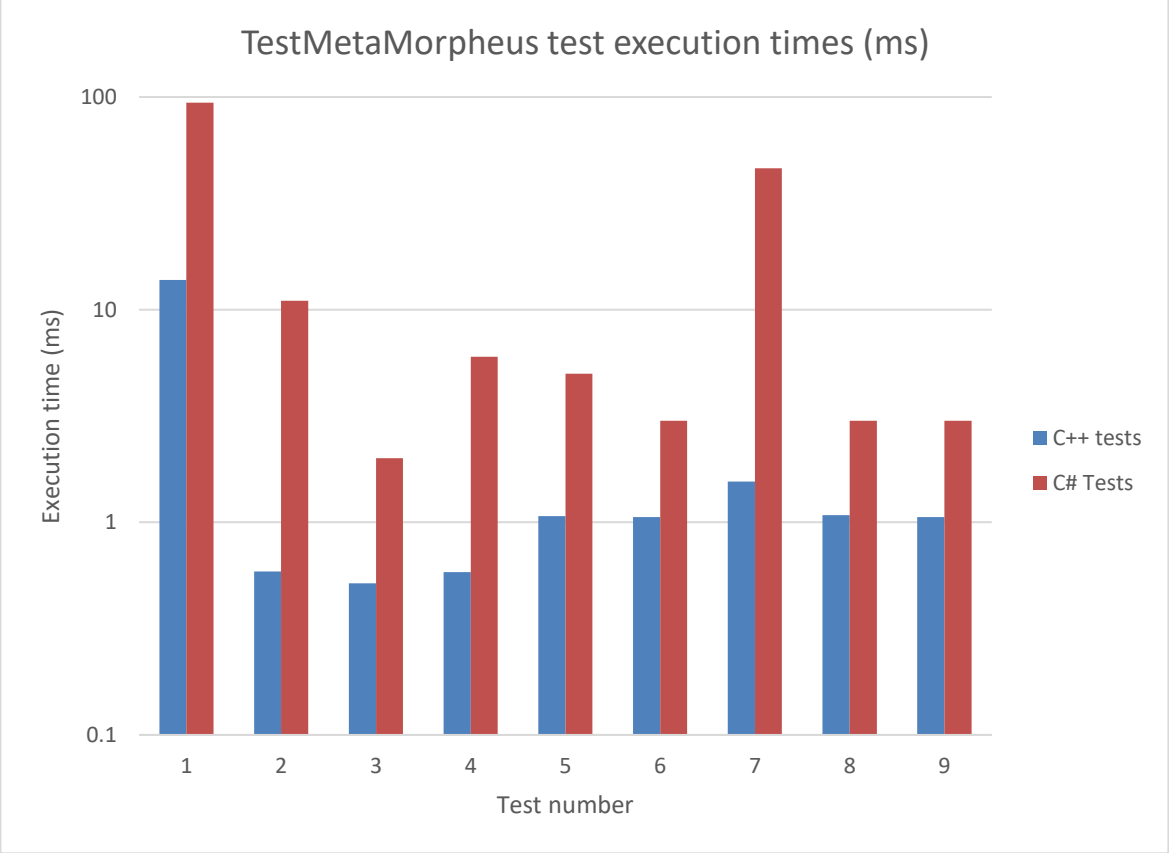

Figure 4-9: A comparison of the execution times of the TestMetaMorpheus tests for the C++ and C# code.

The tests shown in Figure 4-9 did not deal with File IO. Their performance was tested because the actual analysis performed by the application is expected to be in line with the operations being performed in the tests. These tests were chosen to represent a subset of the operations being performed by the C++ application in an actual use case. These tests focused on creating and modifying peptides and then checking to make sure that the formulas were parsed correctly, and the modifications led to the desired results. As with the toml read and write functionality and the mzML functionality, the converted C++ code showed a clear improvement over the C# code when executing the MetaMorpheus tests. Again, the C++ code was often so much faster that a logarithmic scale was needed to display the results as the C# code was often a factor of 10 or more slower than the converted C++ code. These results provide some insight into the expected performance behavior of the C++ version of the application. Although simple, they provide a starting benchmark for what might be expected from the full application.

5. CONCLUSIONS

This project focused on the conversion of two software packages required for peptide fragment matching for mass spectrometry applications. The original software packages, MzLib and MetaMorpheus, were written using the C# language. In order to execute the programs in a Linux environment, they were converted from C# to C++. This was done with the intention of improving performance as the peptide search application can have a very long execution time. The conversion to C++ was also the first step to making it possible to execute the search program on a multi-node cluster.

The File IO, serialization-deserialization functionality, and several general unit tests of the MzLib and MetaMorpheus libraries were isolated and tested to provide execution time benchmarks for the original C# code and the converted C+ code. These tests showed that the converted code is faster than the original code in most cases. This was expected as C++ is a lower level language and is often used in high-performance computing applications, and C# has much more overhead. Because of this, the conversion of these libraries shows promise in speeding up the overall peptide search application especially if multi-node parallelization can be leveraged.

6. Future Work

This project started the process of converting and testing the MzLib and MetaMorpheus packages. There is still much work to be done before the conversion process is complete. Investigating the performance differences between the original C# code and the converted C++ code for different .mzML files is necessary to find out why the converted code outperformed the original code in all but one of the tests. This might highlight areas where the converted code is performing slowly.

Most importantly, the libraries have not been completely converted. The remaining functionality will need to be converted and tested. In general, converting and executing the unit tests gives a good outline of what functionality is missing or working incorrectly. Converting the remaining unit tests in both packages is necessary because if any of the tests do not work correctly, then some part of the peptide search is likely to break or provide incorrect data. It is imperative that these tests be converted, tested, and debugged until they provide the expected results.

Once the tests are converted and tested, if incorrect results are obtained or the tests crash during execution, then the functions being tested need to be debugged. There is a chance that during this process, missing functionality in the converted code may be found. If there is any missing functionality, it will need to be added either by using a third-party library or by adding it manually.

Finally, as explained in the Background section, the peptide database search has $O(n^2)$ complexity with respect to the database size. This means executing a full database search can take a prohibitively long time. However, full database searches are required for the most accurate results. Adding multi-node parallelism to these packages in order to leverage the

performance improvements offered by modern high-performance computing is desirable. If these packages can be executed on a cluster, the overall execution time may be dramatically reduced.

7. BIBLIOGRAPHY

[1] Cross-Linking Mass Spectrometry: An Emerging Techology for Interatomics and Structural Biology. Clinton Yu, Lan Huang. ACS Publications 2017.

[2] Cross-Linking/Mass Spectrometry for Studying Protein Structures and Protein-Protein Interactions: Where Are We Noe and Where Should We Go from Here? Andrew Sinz. Agnew Chem. Int. Ed. 2018, 57, 6390 – 6396

[3] Identification of MS-Cleavable and Noncleavable Chemically Cross-Linked Peptides with MetaMorpheus. Lei Lu, Robert J. Millikin, Sten K. Solntsev, Zach Rolfs, Mark Scalf, Michael R. Shortreed, Lloyd M. Smith. Journal of Proteome Research 2018, 17, p. 2370-2376.

[4] Enhanced Global Post-Translational Modification Discovery with MetaMorpheus. Stefan K. Solntsev, Michael R. Shortreed, Brian L. Frey, Lloyd M. Smith. Journal of Proteome Research 2018, 17, p. 1844 - 1851

[5] Exhaustively Identifying Cross-Linked Peptides with a Linear Computational Complexity. Fenchao Yu, Ning Li, Weichuan Yu. Journal of Proteome Research 2017, p. 3942 – 3952

[6] Understanding the Differences Between C#, C++, and C. Janice Freedman, 17 May 2018. Accessed: 11 December, 2019. https://csharp-station.com/understanding-the-differences-between-c-c-and-c/

[7] Common Language Runtime Overview. Accessed: 11 December, 2019. https://docs.microsoft.com/en-us/dotnet/standard/clr

[8] A Brief Description of C++. Accessed: 11 December 2019. http://www.cplusplus.com/info/description/

[9] An Overview of C++. Bjarne Stroustrup, AT&T Bell Laboratories Murray Hill, New Jersey 07974. 10/1986.

[10] Handbook of Object Technology. Saba Zamir. CRC Press, 1999.

[11] Open MPI Documentation. Accessed: 15 December, 2019. https://www.open-mpi.org/

[12] Toml Language Developers. Accessed: 09 April, 2020. https://github.com/toml-lang/toml

[13] Hupo Proteomics Standards Initiative, mzML 1.1.0 Specification. Accessed: 09 April, 2020. http://www.psidev.info/mzML

[14] Nett Package Developers. Accessed: 15 February, 2020. https://github.com/paiden/Nett

[15] Serialization (C#). Accessed: 15 February, 2020. https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/serialization/

[16] Kojak Documentation. Accessed: 2 March, 2020. http://www.kojak-ms.org/

[17] Stavro and MeroX Documentation. Accessed: 2 March, 2020. https://www.stavrox.com/

[18] A High-Speed Search Engine pLink 2 with Systematic Evaluation for Proteome-Scale Identification of Cross-Linked Peptides. Zhen-Lin Chen, Jia-Ming Meng, Yong Cao, Ji-Li Yin, Run-Qian Fang, Sheng-Bo Fan, Chao Liu, Wen-Feng Zeng, Yue-He Ding, Dan Tan, Long Wu, Wen-Jing Zhou, Hao Chi, Rui-Xiang Sun, Meng-Qiu Dong & Si-Min He. Nature Communications. July 30, 2019.

[19] ProteinProspector V6.2.1. Accessed: 2 March, 2020. http://prospector.ucsf.edu/prospector/mshome.htm

[20] Identification of Cross-Linked Peptides from Large Sequence Databases. Nature methods.

Rinner O, Seebacher J, Walzthoeni T, Mueller L, Beck M, Schmidt A, Mueller M, Aebersold R (2008)

[21]Tangible Software Solutions C# to C++ Converter Documentation. Accessed: 5 May, 2020.

https://www.tangiblesoftwaresolutions.com/product_details/csharp_to_cplusplus_converter_details.html

[22] R Documentation Quantile Specification. Accessed 6 May, 2020.

https://www.rdocumentation.org/packages/stats/versions/3.6.2/topics/quantile

[23] TinyToml Documentation. Accessed: 7 May, 2020. https://github.com/mayah/tinytoml

[24] Grant, W. Shane and Voorhies, Randolph (2017). cereal - A C++11 Library for Serialization. Accessed 7 May, 2020. http://uscilab.github.io/cereal/

[25] Code Synthesis XSD Documentation. Accessed 9 May, 2020.

https://www.codesynthesis.com/projects/xsd/

[26] Michael Richard Cooper, Rabindranath Dutta, Kelvin Roderick Lawrence. US6986101B2, United States Patent and Trademark Office, May 6 1999. Accessed 29 May, 2020. Google Patents, https://patents.google.com/patent/US6986101B2/en

[27] Donald Edward Baisley, Sridhar Srinivasa Iyengar, Ashit Sawhney. US6292932B1, United States Patent and Trademark Office, May 28 1999. Accessed 29 May, 2020. Google Patents, https://patents.google.com/patent/US6292932B1/en

[28] Craig Salter, Christina P. Lau. US7496838B2, United States Patent and Trademark Office, July 7 2002. Accessed 29 May, 2020.

Google Patents, https://patents.google.com/patent/US7496838B2/en

[29] Takahiro Kumura. US7299460B2, United States Patent and Trademark Office, May 6 1999. Accessed 29 May, 2020. Accessed 29 May, 2020. Google Patents, https://patents.google.com/patent/US7299460B2/en

[30] MzLib Documentation. Accessed 5 June, 2020. https://github.com/smith-chem-wisc/mzLib