Abstract— Scientific computing is dominated by team-authored legacy code that has evolved over decades with the purpose of capturing the evolving understanding of a scientific discipline. Accumulated deprecated code, various optimization techniques, and evolving algorithms lead to convoluted source code that is impractical to reverse engineer using mainstream methods. This prevents codes from being truly repeatable or understandable, which are two of the most essential needs in scientific computing. We refactored a long-standing implementation of a common biosequence alignment algorithm in an effort to reproduce its salient behaviors in usable form. Because of the shear size and complexity of this code base, we developed custom tools to visualize and manipulate the source code behavior under a variety of conditions. We present a case study of extracting and refactoring the algorithmic core and a novel process of discovery/prototyping/testing using a combination of openly available and custom-built tools. The result is a reduction in code size of over 2 orders of magnitude while reconstructing the key protein alignment function in BLAST.

Keywords- code reuse; bioinformatics; scientific computing; visualization; program understanding

I. INTRODUCTION

Scientific computing has unique needs in terms of software development, including the frequent absence of up-front requirements, constantly changing algorithms that reflect progress in understanding, and authorship that can include large geographically dispersed collaborative teams with fluid membership and nonstandard coding styles. Driving such a software development environment is an underlying scientific discipline that evolves rapidly in terms of fundamental understanding that must be captured by the code. This leads to an organic style of software development in which code must be modified more quickly than it can be standardized. A second concern for many scientific applications is the need for optimization, which often leads to hard-coded (and often undocumented) code regions that are initially only for testing, but that are eventually absorbed into the functional core.

Along with the need for an organic code development process, scientific computing also has a driving need for repeatability since the professional credibility of its users relies on the ability of others to reproduce important results. However, this is often very hard to realize in an organic development environment. We present here a case study for refactoring one such code: Basic Local Alignment Search Tool (BLAST [1])—one of the most commonly used biological sequence analysis algorithms, having tens of thousands of citations for the original publication and a variety of applications and services built using the BLAST computational core.

BLAST is a large-scale legacy code that is of central importance to the biology community. BLAST was originally developed in the late 1980’s to address the need for comparing genes and proteins based on the text that describes the sequence of chemical subunits in them. The BLAST algorithm was originally published in 1990 and with its related papers has been cited over 100,000 times for use in applied research such as drug discovery and biomarkers research, and decades of fundamental research into molecular processes that give species and communities the capacity to survive. The BLAST algorithm has become so fundamentally important to biological sciences that increasingly large datasets are being analyzed using BLAST. In fact, typical sequencing platforms that are mostly responsible for the influx of new sequences to analyze are increasing their throughput more quickly than Moore’s Law—leading to a situation in which the need for computing is outpacing the underlying hardware improvements. This
motivates a need for parallel implementations of BLAST such as ScalaBLAST [2].

However, BLAST was not implemented as a library, so using it as the algorithmic core of ScalaBLAST and other parallel implementations is challenging because of its lack of external API and problematic because of the possibility of unintentional side effects when modifying the BLAST core. As with most scientific software, the low-level details of how BLAST is implemented have been left out of publications. Even with the large corpus of publications on the details of BLAST, there are many implementation-level details that must be discovered to create a repeatable BLAST compute core. Our goal was to re-implement the BLAST functionality necessary to drive protein comparison calculations (the blastp operating mode) so that we would have complete transparency and understanding of the implementation details, and so that we could be certain that our generalized parallel implementation did not introduce unwanted side effects into the serial BLAST core when driving it with our parallel ScalaBLAST control layer. A second motivation for refactoring the BLAST core is to create a domain-agnostic (i.e. non-biological) string analysis platform. The utility of such a platform has been previously demonstrated in domains such as cyber security [3].

However, in order to use BLAST on data from non-biology domains, the user must map their data into text sequences. This mapping requires converting the data space of a generic domain into the specific amino acid frequencies that occur naturally in biology. If this mapping is not done accurately, it can significantly impact performance and accuracy. This constraint on character frequency that is imposed by using the biological code without modifications makes use of BLAST on non-biological datasets over-constrained in most cases, hence our desire to achieve a domain-agnostic version of the code.

II. THE BLAST ALGORITHM

BLAST was devised to address a fundamental question in biosequence analysis—calculating the statistical confidence behind the assertion that two biosequences are derived from a common ancestor. Biosequences are linear sequences of chemical subunits.

At the heart of the BLAST algorithm is a process of text alignment between two sequences—pairwise alignment. The goal of pairwise alignment is to discover regions of two sequences that have a high degree of similarity (see Figure 1).

![Figure 1. Example of text strings and a local alignment.](image)

Local alignment is calculated efficiently using a staged process where each level is designed to reduce the overall search space that must be examined by the code to identify alignments.

The functions that perform these tasks are captured in a large code base having a high degree of complexity. Table 1 illustrates some of the attributes of the source code for the version of the NCBI BLAST toolkit that was frozen as the basis for ScalaBLAST (BLAST 2.2.13).

<table>
<thead>
<tr>
<th>Number of lines of code</th>
<th>1.5 Million (with comments)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of statements</td>
<td>Nearly 800,000</td>
</tr>
<tr>
<td>Number of files</td>
<td>1953</td>
</tr>
<tr>
<td>Number of functions</td>
<td>Over 25,000</td>
</tr>
<tr>
<td>% lines that are comments</td>
<td>19 (mostly terse source code revision history)</td>
</tr>
</tbody>
</table>

*figures obtained using SourceMonitor from Campwood Software

Not all of the functionality in the original code was needed for our applications but extracting the necessary functionality from the code base required analysis of the entire toolkit to discover which segments of the code were needed for our refactoring.

III. SOFTWARE EXTRACTION

A. Generalizing the BLAST algorithm for non-biological use

ScalaBLAST was originally built on top of BLAST, and was later modified to be more tightly integrated with the BLAST libraries via the use of BLAST data structures and “API”. The tight integration caused ScalaBLAST to become unstable as new versions of BLAST were released. In the biology community, BLAST is considered a software library, but has a very volatile API. Therefore efforts to adapt ScalaBLAST to new versions of BLAST resulted in large programming overhead. To gain stability in ScalaBLAST, the version of BLAST was frozen. This had the advantage of making ScalaBLAST maintainable, but the disadvantage of being unable to compare results with newer releases of BLAST. This presents a problem in the biology community, as BLAST is widely considered the gold standard for sequence alignment.

BLAST is designed to process DNA and protein sequences only. The heuristics and algorithms have been crafted using assumptions from the biology domain. In particular, the statistical models used by the algorithms are based on existing DNA/protein populations. To give the reader an appreciation of the scope of the influence of the statistical models [4] on the software as a whole, there is a 55 page paper summarizing the statistics that heavily influence nearly every algorithm or heuristic[8].

Because of the complexity of the BLAST source code distribution, we searched for existing alternatives to the BLAST library as the basis for refactoring our code. We considered several programs, including Biopython, Bioperl, and Seqan. These tools did not help as they are based on many of the same biological assumptions as BLAST and in some cases are just wrappers that call the BLAST routines underneath.
We attempted to create our own version of BLAST based solely on research papers describing the BLAST algorithms and heuristics. But our results from ground-up BLAST refactoring differed significantly from the open source BLAST due to a large number of undocumented algorithmic details. The sophistication and importance of the underlying statistical model were beyond our ability to replicate effectively.

The complexity of the codebase and various optimization techniques precluded a brute force method of reading and understanding directly from source code. In addition, understanding of the code is complicated by optimization techniques that confound code analysis. These include heavy use of C pointers, use of the ‘register’ keyword, C structs comprised of void* pointers, heavy use of ifdef, and a “super-global” data structure that is constructed, extended, and significantly modified throughout the code. Of course standard issues such as a complicated build system, lack of test code, and undocumented API also existed.

B. Software Archeology

Attempting to understand the BLAST code base resulted in what is commonly called “software archeology” [5]. Digging through the layers of the code allowed us to identify some of the key issues that we would need to address to re-engineer this code. Since BLAST was developed over a period of decades, there has been a “layering” affect in the API. Functions that were once part of the API were wrapped with new functions as requirements changed. These were in turn wrapped with even newer function calls, some of which simply reorder arguments from other parts of the API. These API layers make it extremely difficult to locate key functionality, as it may be hidden under 10 or 20 layers of the call-stack. The BLAST code also shows signs of complete functionality replacement over time and both the original and improved functions are left in the code. This results in having to actually run and debug the software to determine which piece of code is operational. This “abandoned code” and code bloat added to the complexity of detailed understanding.

IV. NON-INTRUSIVE METHODS

A. Commercial Software Attempts

In an attempt to gain an understanding of the overall structure of BLAST (and eventually the underlying details), we used several pieces of commercial software. These included SourceMonitor, Starlight [6], KCachegrind, Visustin, and DDD.

We used SourceMonitor to perform static analysis on BLAST and gathered metrics on the entire code base. In particular, the cyclomatic complexity provided by SourceMonitor proved to be extremely high on average in BLAST. There are over 50 files in the BLAST code base that have a cyclomatic complexity greater than 100 [7].

Starlight is a tool for visualization and exploration of data networks. We used it to visualize a static representation of the potential BLAST call stack. We developed a structure containing every function call in the BLAST code base, and used Starlight to view the resulting associations. We found almost all functions tightly-coupled with the system as a whole. This technique was useful for identifying clusters of functions that make up specific functionality or heuristics.

We used Valgrind in combination with KCachegrind to analyze the function call tree at run-time and gained a high-level understanding of the portions of code that were exercised during a given run of BLAST. KCachegrind is an interactive tool that allowed us to explore multiple aspects of the code, including the call tree, function names, call frequency, looping structures, code coverage, and functionality discovery. A portion of the tree traversed during a run is shown in Figure 2. Each box in Figure 2 represents a single function and the line between the boxes is the number of times the path was traversed during the Valgrind (callgrind) snapshot.

Other tools such as Visustin for static analysis of control flow structures and Visual Studio’s debuggers only confirmed the complexity of the problem but did not provide any useful additional analysis.

We were able to use the information gathered from KCachegrind in conjunction with GDB to walk through the code at run-time. We set breakpoints at the beginning and end of every function that KCachegrind identified as being executed. While this gave us a better understanding of the run-time behavior of BLAST, the complicated control flow structures and “super-global” data structure proved to be too cumbersome for basic debugging. This led us to the use of DDD as a way to visualize the data structures at run-time. DDD is a wrapper around GDB, with the added benefit of visualizing C data structures. Unfortunately, the size and complexity of the data structures again proved too cumbersome for the tool. In addition, since the “super-global” data structure in BLAST is constructed of multiple levels of structs of (void*) pointers, DDD was unable to dynamically display the structure in its entirety because it did not know how to cast the structure to the correct type.

These commercial applications gave us various hints as to the depth and complexity of the code although they did not individually or collectively provide an easy way to understand the functionality and data flow in BLAST.

B. Custom-built debugging tool: GdbShell

In order to track changes to the “super-global” data structure throughout the program run we used GDB to step through the code. We combined GDB with Graphviz to create a visualization of the data structure and the changes that had been made to it. To more easily control the GDB process we wrote a Perl wrapper as a scripting engine for automated GDB control. This allowed us to set breakpoints at an arbitrary number of specific points of interest in the code, and walk through them automatically. We call this tool GdbShell.
GdbShell provides the ability to display the changes to a data structure between two points in a program. This involves setting a breakpoint, asking GDB for a text representation of a given structure, parsing that text for nested structures and finally recursively parsing the GDB responses for additional nested structures. After the data structure had been completely traversed, GdbShell saves it as a snapshot of the data structure. GdbShell continues debugging until another breakpoint is reached. Then another snapshot of the same data structure is captured and compared to the previous one. This comparison involves searching for parts of the structure that had been added or deleted between breakpoints, as well as modifications to the internal values of any part of the data structure. A color-coded image is created based on which portions of the data structure were added, deleted, or modified, shown in Figure 3. The colors for Figure 3 are coded to signify:

- White - no change between breakpoints
- Green - new structures created
- Blue - modified structures (with OLD and NEW)
- Yellow - custom code had to be written to view data structures (e.g. pointers to arrays of pointers to arrays of integers representing 5-bit packed ASCII characters) between breakpoints.
- Purple/Orange – legend showing the breakpoints used to create the image.

GdbShell did not originally have the ability to display complicated dynamically allocated structures such as a pointer to an array of pointers to arrays that represent a two-dimensional matrix. We enhanced the functionality by developing a framework that supports a simple plug-in architecture using a visitor pattern for each unique data structure. When a new data structure is discovered, custom Perl code can be written to convert GDB representation of the data into a human-readable ASCII representation.

GdbShell provided the necessary tools and processes for understanding the BLAST code in a practical timeframe. Without adding the features of automation and the ability to quickly add custom analysis of new data structures the process of detailing the complicated data flows and data structures would have been technically possible but not practical.

V. PROTOTYPING WITH A “DISCOVERY CYCLE”

After using non-intrusive methods to determine where the algorithms of interest were located, and the sequence of the related function calls, we used GdbShell to discover what the algorithms did and how they affected the data structure and then prototyped what we learned in Perl. Perl allows for rapid development, includes object orientation, and works well with text-based problems like BLAST.

This cycle involves using KCachegrind to isolate portions of the code and GdbShell to gain an understanding of data structures (how they changed, and which boundary conditions caused these changes). Once we gain an understanding of a particular feature, we implement it in our Perl prototype. We attempted to “checkpoint” the code in between heuristics, to ensure that each individual piece of our prototype was producing comparable results to the corresponding BLAST heuristic. If the results differed at these checkpoints, we investigated by hand using old-fashioned “intrusive” methods such as printf and exit statements. This allows for comparison of data structures at diverging points which then is fed into the prototype discovery cycle.

VI. TESTING AND VERIFICATION

Our goal was to abstract, extend, and simplify the overall algorithm, while producing comparable results without simply copying the BLAST source code into our prototype. It was essential for us to understand all the details of how the code works.

BLAST is a heuristic chain, meaning that the overall algorithm consists of multiple heuristics, each of which is designed to perform some amount of data reduction on the overall data space. The input data given to BLAST can
influence the underlying statistical models, which in turn influence each of the data reduction heuristics. Such a combination of heuristics creates a data-dependent decision making process wherein a subtle statistical change at one layer of the heuristic can propagate through other layers resulting in very different output. This cascading dependency is illustrated in Figure 4. In addition, each heuristic is so finely tuned for performance that there are too many boundary conditions to exercise all combinations with a reasonable set of test data.

Because of the large amount of boundary conditions and dependencies in the heuristics, we needed a large, diverse data set to test the overall process with. And we were much more concerned with accuracy than speed during the prototyping phase so individual tests could take minutes to hours and there were hundreds or thousands of tests in order to exercise all boundary conditions. To get around this accuracy/time conundrum, we developed a simple test framework on a large cluster (Olympus - 179th on the top 500 list). This allowed us to quickly create tests that used a large data set, exercise relevant code, and combine results in real time in order to compare the aggregate results of our prototype and BLAST. This allowed us to terminate large test runs if we discovered a difference/problem early in the run, or if it was obvious that a recent change did not improve results. At the end of a test run (and in real time), we gathered statistics regarding the accuracy of our prototype, where any problems were located in the code, as well as the details of the individual problems.

For a more detailed test, we used a whole yeast species genome which has 5753 proteins. This whole genome was compared against itself, which should result in a strong alignment between each protein and itself, followed by a collection of lower scoring alignments to more distantly related proteins, followed by a large collection of statistically insignificant alignments, which are useful for our purposes of exposing all the details of how alignments are calculated. Using this detailed dataset to test as much as possible of the operational code regions, we evaluated the performance using the following metrics: (1) the fraction of proteins for which non-self top-hits had the same identity in both our code and the NCBI implementation; and (2) the average number of hits in agreement between the two methods before a difference is found. The first metric assesses one of the most important functions of the BLAST code, namely the ability to recognize highly similar but non-identical proteins to a query. Many
users really only want the top (non-self) hit or the top few, so getting the first non-self hit is the first metric we used. The second metric assesses the quality of the complete hit list. For a given protein query, all of the target proteins that have a significant alignment are returned by the BLAST method. This list is sorted by score and a statistical measure that is calculated by the code. When multiple hits have identical score and statistical measure, they appear in the list in random order. This creates difficulties when comparing two results because things can be in a different order, but still correct because there is no correct order for a collection of alignments that have identical scores. We solved this problem by sorting alphabetically on unique protein names within identical score blocks for both our runs and NCBI BLAST runs before calculating our performance statistics. This ensured that differences that are not resolvable by either code did not count against our results.

Using this procedure, our code achieved the same non-self top hit as NCBI BLAST for 5707 of 5753 proteins (99.2% of proteins tested). This is an encouraging result that suggests we are in agreement the vast majority of the time when alignments are strong alignments (and therefore less likely to be influenced by decision making at the statistical fringe). Table 2 illustrates the performance of our method using a variety of metrics that explore all hits for each protein instead of just the top non-self hits.

<table>
<thead>
<tr>
<th>Cutoff value ‘x’</th>
<th>AHBM</th>
<th>AHBM %</th>
<th>Fraction identical</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>386.7</td>
<td>0.77</td>
<td>.45</td>
</tr>
<tr>
<td>400</td>
<td>311.9</td>
<td>0.78</td>
<td>.45</td>
</tr>
<tr>
<td>300</td>
<td>237.2</td>
<td>0.79</td>
<td>.45</td>
</tr>
<tr>
<td>200</td>
<td>162.3</td>
<td>0.81</td>
<td>.46</td>
</tr>
<tr>
<td>100</td>
<td>86.00</td>
<td>0.86</td>
<td>.51</td>
</tr>
<tr>
<td>50</td>
<td>45.44</td>
<td>0.91</td>
<td>.58</td>
</tr>
<tr>
<td>25</td>
<td>23.71</td>
<td>0.95</td>
<td>.68</td>
</tr>
</tbody>
</table>

For each cutoff value ‘x’, only the top ‘x’ alignments for each protein were considered. The ‘Average Hits Before Mismatch’ (AHBM) value was calculated by locating the first discrepancy between NCBI and our BLAST implementation for each protein. If there was no discrepancy and a protein had fewer than ‘x’ alignments, the value of the first mismatch was counted as ‘x’. Ideal performance for this metric is to have an AHBM value equal to ‘x’, meaning that the end of all lists was reached without a discrepancy. A value of 0 would be the worst case, meaning that on average, lists varied at the top hit location. AHBM% is a second representation of this metric.
that expresses the same score as a fraction of the ‘x’ value. In this case, 1.0 is an ideal score and 0.0 is the worst possible score. The third measure is the fraction of protein queries for which the entire alignment list (after alphabetically sorting within score-invariant blocks) had all the same alignments with the same scores in the same order to that produced by NCBI BLAST for the given cutoff value.

This validation shows two significant results. First, our implementation reproduced the top non-self alignment for the vast majority of test cases. This is an essential feature to capture to make sure our results are relevant to users. Second, on average, mistakes do not occur in the top part of the list (i.e., the part of the list with highest statistical significance), and when only the top 25 alignments are considered, the average error does not occur until the 23rd or 24th alignment. In addition, nearly 70% of the lists were completely identical through the top 25 hits when comparing our BLAST implementation with that of NCBI.

VII. CONCLUSIONS AND FUTURE DIRECTIONS

In this study, we were able to reproduce many of the essential details of BLAST, one of the most pervasive and significant algorithms used by the biological research community. Because of the complexity of the code, we used a combination of commercial products and custom-designed tools to understand the original implementation enough to refactor it. Without automating the “discovery cycle” and prototyping and testing smaller sections of code we would not have been able replicate the outcome to any degree of certainty. Standard tools and processes did not apply to this particular set of constraints so new tools were developed and applied. These tools were built specifically for this effort but are in the process of being abstracted for general use. They could provide other developers attempting to re-construct functionality of code where traditional methods don’t work.

Our testing and validation has shown that we have captured many of the essential core heuristics of the NCBI BLAST implementation, but the differences between them have led us to discover further undocumented details in the BLAST source code that would need additional development for us to replicate.

We must point out that the NCBI BLAST code we used was extremely fast and robust which is a testament to the dedicated developers and their attention to detail. Because our emphasis was on correctness and not performance, we have not introduced optimization into our implementation. Much of the complexity in the NCBI BLAST core is due to hand-optimization of code segments. It is not yet clear how much of this must be captured to reproduce both the performance and the results of the BLAST core. However, our intention is to have a complete implementation of correctly refactored code, then proceed with our own optimization on a much smaller, more formally designed codebase that can be easily maintained and extended to non-biology data domains.

We believe that our experience refactoring the BLAST source code is representative of the complexities of maintaining and refactoring legacy scientific codes and for other multi-author codes that have a similar development cycle. For some applications, emphasis on performance and the evolving nature of the underlying algorithms can lead to highly complex software dependencies. When this is combined with a long-term development cycle for which there is a large number of contributors, gaining transparency into the implementation-level details of an algorithm can become prohibitive. In this paper we present an example of how combining off-the-shelf products with custom analysis can yield some of the transparency needed for more fully understanding these implementation details but acknowledge that further work is needed for complete understanding.

VIII. REFERENCES