Architectural Refactoring for Fast and Modular Bioinformatics Sequence Search

Jeremy Archuleta
School of Computing
University of Utah
Salt Lake City, UT 84112
jsarch@cs.utah.edu

Eli Tilevich
Dept. of Computer Science
Virginia Tech
Blacksburg, VA 24061
tilevich@cs.vt.edu

Wu-chun Feng
Dept. of Computer Science
Virginia Tech
Blacksburg, VA 24061
feng@cs.vt.edu

Abstract

Bioinformaticists use the Basic Local Alignment Search Tool (BLAST) to characterize an unknown sequence by comparing it against a database of known sequences, thus detecting evolutionary relationships and biological properties. mpiBLAST is a widely-used, high-performance, open-source parallelization of BLAST that runs on a computer cluster delivering super-linear speedups. However, the Achilles heel of mpiBLAST is its lack of modularity, adversely affecting maintainability and extensibility; an effective architectural refactoring will benefit both users and developers.

This paper describes our experiences in the architectural refactoring of mpiBLAST into a modular, *high-performance software package. Our evaluation of five component-oriented designs culminated in a design that enables modularity while retaining high-performance. Furthermore, we achieved this refactoring effectively and efficiently using eXtreme Programming techniques. These experiences will be of value to software engineers faced with the challenge of creating maintainable and extensible, high-performance, bioinformatics software.*

1. Introduction

At the start of the summer of 2006, we set upon the formidable task of refactoring mpiBLAST, a popular, parallel, bioinformatics package that runs on a parallel computing cluster [12, 14]. Bioinformaticists have been using mpiBLAST for their research activities ever since we first released the package over three years ago. mpiBLAST has proven to be a very useful scientific discovery tool with more than 40,000 downloads across five major releases. Due to its proven utility, mpiBLAST has become an integral component of several, major, high-performance, cluster distributions [6, 10, 19, 20, 25, 26, 28, 30].

One of the reasons for the widespread popularity of mpiBLAST is its open-source development model, which fueled a grassroots movement to provide support for the code. Alas, the enthusiastic support of this ad hoc grassroots movement exposed shortcomings in the overall design of mpiBLAST (e.g., lack of modularity and consistency) that needed to be addressed fully and expediently. Hence, we undertook the architectural refactoring of mpiBLAST, and in the short course of three months, we successfully accomplished this challenging task.

Despite the clear objective of creating a new design, we had to overcome software engineering challenges from both a purely technical and human resources standpoint. In the technical realm, our effort entailed refactoring 10K lines of functional code with little software engineering discipline into a high-quality software package that adheres to state-of-the-art software engineering principles such as modularity, reusability, and encapsulation. However, the major draw of mpiBLAST has been its high-performance functionality (i.e., super-linear speedup). Therefore, any changes in the design had to maintain (or improve) performance while providing the additional benefits of disciplined software engineering to its stakeholders. Specifically, end-users require easy-to-use interfaces and expedient support; system administrators require simple installation and upgrade procedures; and developers require a modular codebase to enable seamless maintainability and extensibility.

Furthermore, our refactoring effort had to be completed by a team of three in three calendar-months but utilizing only five person-months [8]. With such a short timeframe, this project called for a software development model that allowed for a rapid development cycle. To this end, we employed several techniques of eXtreme Programming to explore the design space to the fullest degree possible within our time constraints, i.e., rapid prototyping, pair programming, and unit testing [3].

Our experience in refactoring mpiBLAST exemplifies how a small team size complements the rapid development cycle by fostering an environment in which designs can be

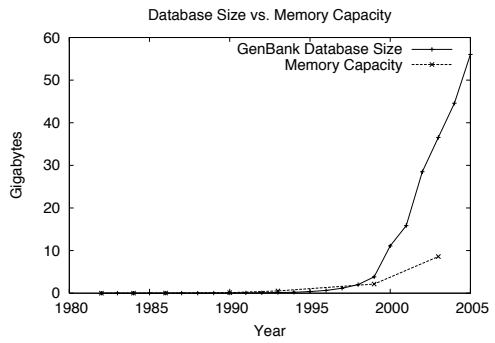

Figure 1. Comparison of the Growth of GenBank against the Growth of Memory Capacity

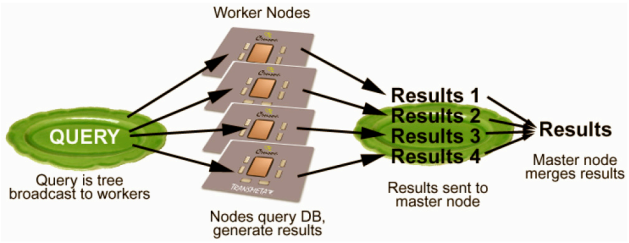

Figure 2. High-level View of mpiBLAST Algorithm

quickly generated, evaluated, and critiqued. It is important to note that this effort did not start from scratch, but rather from a functioning software package that had been in active use and development for several years. Instead of starting from a clean slate, we embraced our prior experiences with mpiBLAST and focused on the architectural refactoring [9, 15, 36].

Additionally, our endeavor is an example of a common trend. As parallel computation becomes a requirement for a large and growing number of computing applications, their increased complexity will likely lead to decreased maintainability, making the architectural refactoring of such parallel applications more and more common. This paper reports on our experiences of refactoring a high-performance, parallel, bioinformatics, open-source application, and we believe that our experiences will be of value to software engineers faced with the challenge of creating maintainable and extensible software in this important domain.

The rest of this paper is structured as follows. Section 2 explains the significance of mpiBLAST from the user's perspective as well as the main aspects of its algorithmic design. Section 3 details both the motivation behind our refactoring effort and the design objectives we set for ourselves. Our methodology and refactoring experiences during this effort are reported in Section 4 and Section 5, respectively, culminating in the final design that we chose for mpiBLAST-2.0. We then outline some future directions for mpiBLAST as well as how the new design will enable them in Section 6. Lastly, Section 7 summarizes our experiences and lessons learned of this architectural refactoring of mpiBLAST.

2. mpiBLAST Overview

The advent of genome sequencing has brought biomedical researchers a wealth of DNA sequence information. Researchers commonly use the Basic Local Alignment Search Tool (BLAST) to search these sequence databases for regions of homology between a query sequence and one of the database sequences. Because the BLAST algorithm detects both local and global alignments, regions of similarity that are embedded in otherwise unrelated proteins can be detected [1, 2]. Both types of similarity can reveal key insights into the function of uncharacterized proteins.

BLAST enables the rapid search of nucleotide and protein databases in a sequential environment, and until recently, these databases could fit in main memory. However, this is no longer the case as shown in Figure 1. More importantly, because the size of sequence databases is doubling every 12 months and far outpacing memory growth, which is quadrupling every 36 months (or doubling every 18 months), sequence databases are unlikely to ever fit in a sequential environment's main memory again [5, 18].

In 2002 and 2003, we developed an open-source software package, mpiBLAST, which augments the standard BLAST software, developed by the National Center for Biotechnology Information (NCBI), by executing it in parallel on a network of computers (i.e., compute cluster). At a high level, the mpiBLAST algorithm follows a Master-Worker parallelization model that consists of three basic steps: (1) distributing the query to be searched by each Worker, as shown at the left of Figure 2, (2) searching the query on each Worker, and (3) merging the results from each Worker into a single output file, as shown at the right of Figure 2.

The significance of mpiBLAST's parallelization scheme is that it segments the database into pieces such that each compute node searches a portion of the database, giving the notable advantage that it offers super-linear speedup when the database being searched is too large to store in an individual compute node's memory. Furthermore, even when the original database fits in memory, mpiBLAST still improves throughput by finishing the search faster.

As sequence databases experience exponential growth and sequence searching becomes more computationally intensive each year, so grows the importance of mpiBLAST functionality to bioinformatics researchers [24].

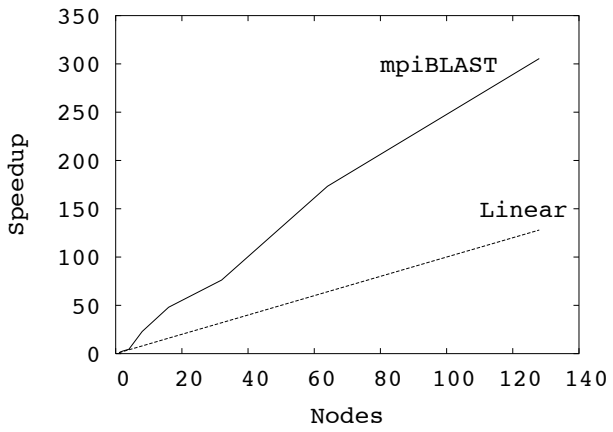

Figure 3. Performance of mpiBLAST 1.4.0

3. Refactoring Objectives

Our starting point for the architectural refactoring was mpiBLAST-1.4.0. As shown in Figure 3, this version exhibits super-linear speedup across more than 100 processors through the use of query and database segmentation, sophisticated scheduling, pipelined results gathering, and asynchronous communication [11].

Despite its functionality, the codebase substantially hinders maintainability and extensibility. For example, mpiBLAST-1.4.0 is dependent on specific versions of the NCBI C Toolkit, thereby requiring a new and custom patch to be created for every new release of the Toolkit. Therefore, one can significantly improve the package's maintainability and portability by introducing an abstraction that streamlines this necessary but tedious and error-prone task.

Additionally, many changes to mpiBLAST-1.4.0 require developers to have intimate knowledge of the entire package. For example, changing the way results are written requires substantial modifications to several different functions scattered across multiple source files. Not only does this level of knowledge hinder extensibility, but this also makes the package more difficult to maintain.

However, that is not to say that efforts have not been made to improve the quality of the mpiBLAST codebase. In fact, with each successive release the codebase has moved more towards a modular design. Nevertheless, the desire to improve the algorithmic quality of the package has overshadowed the objective to improve the modularity of the codebase. From this pattern of development, a "feature-rich", but ad hoc, codebase has evolved.

The very success of the continued development of the package depends on a concerted effort to add new features *while* applying solid software engineering principles to mpiBLAST. This challenging task was further exacerbated by several design constraints.

The challenge of our refactoring effort was in preserving the high-performance properties of mpiBLAST-1.4.0 while simultaneously improving maintainability and extensibility. In addition, we had to preserve the above properties across all of the multiple platforms on which mpiBLAST-1.4.0 runs (i.e., GNU/Linux, Windows, Mac OS X, and other BSD and Unix variants). A new design that is highly maintainable and extensible but exhibiting poor performance characteristics and only supported on one platform would be desirable to its developers, but of zero utility to its users. Therefore, we must ensure that the product of our refactoring effort satisfies all of the mpiBLAST stakeholders: end-users, system-administrators, and developers.

3.1. Maintainability

mpiBLAST has many users that are both geographically and institutionally disparate. In many cases, a single mpiBLAST installation serves hundreds of users as part of a shared supercomputing environment (e.g., Teragrid [35]). Therefore, keeping the package up-to-date by finding and fixing bugs quickly or installing new versions is of great importance.

However, two different groups of mpiBLAST users have unique sets of responsibilities in this process. System administrators, and often the end-users as well, are responsible for installing new versions of the software as well as for reporting any emerging issues to the developer community. mpiBLAST developers address and resolve the issues, and a patch or new release of mpiBLAST is then made available to all interested parties through the mpiBLAST website.

To simplify the maintenance process, we aim to maximize the modularity of mpiBLAST, thereby enabling changes (e.g., bug fixes) on a per module basis and reducing ripple-effects. This has the additional benefit of requiring minimal familiarity with the entire software package when making modifications. For example, an issue with command-line processing will only require familiarity with the command-line module and not other modules such as scheduling or formatting. Furthermore, the code for the command-line module will be located in an intuitively-named source file containing only command-line functionality. Placing each module in its own source file maximizes encapsulation and modularity.

Modularity also enables mpiBLAST to embrace an open-source development model in which many developers can work on the software concurrently. Loosely coupled code allows developers to make and integrate changes orthogonally to each other.

3.2. Extensibility

The growth of bioinformatics data continuously presents new computational challenges. Specifically, databases are growing faster than a single node's physical memory by 33%-50%. Such challenges call for new and advanced algorithms and data structures to be integrated into mpiBLAST.

It is likely that future contributions will span the entire gamut of the mpiBLAST application: better search algorithms, improved communication mechanisms, more intuitive user interfaces (UI), and efficient parallel Input/Output (I/O) strategies to name a few. If mpiBLAST-2.0 fails to facilitate such contributions, it will lose its utility.

Therefore, mpiBLAST should provide an intuitive and flexible design in which all developers can incorporate their novel algorithms and data structures into the package. While we expect a certain level of knowledge from such developers, we should not require developers to possess complete knowledge of the inner workings of the entire package in order to make a contribution. Rather, developers should only need knowledge relevant to the module(s) they are enhancing.

4. Methodology Experiences

As is the case with most successful software engineering endeavors, our general approach not only took into consideration the purely technical characteristics of the task at hand but also the human factors as well.

4.1. Technical Perspective

Parallel bioinformatics is a young research area. Therefore, from a technical perspective, having no proven architectural solutions available for this important, but still emerging, domain required us to be able to evaluate many designs thoroughly and quickly.

To evaluate multiple designs, it was imperative that we keep a record of our efforts and thoughts on each. This included keeping track of our discussions, evaluations, and source code for each design. In addition, these records had to be readily available to all the members of the team at any time for examination and modification. To this end, we utilized a WIKI implementation called pmWIKI to keep track of all non-source-code documents [21, 27]. We chose this particular WIKI implementation because of its simple installation, ease of use, and utility. Of particular utility was the revision control feature of pmWIKI which saved each revision of every document that it managed. This enabled us to replay our thought process whenever we needed to recall why we made a particular decision at some point in the past.

Revision control was also instrumental in source-code development, as it allowed us to keep track of multiple concurrent designs under consideration. The revision control system (RCS) that we chose to use was darcs [29]. The fast and straightforward branching mechanism in darcs significantly facilitated concurrent development of multiple designs. That is, at any point in time, we could branch (i.e., deviate) from our current design and pursue a radically different approach on a whim, while still being able to revert back to the pre-branch design at any time. Additionally, unlike a traditional RCS, darcs did not rely on a centralized server, enabling individual developers to pursue their own branches (i.e., designs) independently of each other. While we did not require the decentralized feature during the refactoring, this feature will greatly facilitate geographically disparate, open-source developers to contribute to mpiBLAST.

Lastly, the advantage of having a fully functional version of mpiBLAST to start with enabled us to develop a thorough suite of unit tests that we used consistently throughout the development process [4]. If the current design could not elegantly pass all of the unit tests, we either branched to a new design or reverted back to a previous design. This practice helped to document our progress and gave us confidence that our efforts were leading to the desired outcome.

4.2. Human Resource Perspective

Our team was composed of three members with each member having defined primary roles in this endeavor, which were determined by their areas of expertise. Our project manager/visionary has been the principal investigator of the mpiBLAST project since its very inception. Our technical lead has substantial experience with architecting large-scale software projects both in industry and academia. Last, but not least, our developer has been involved with the mpiBLAST project for the last three years, including a large-scale effort in sequence-searching the largest nucleotide database against itself [17].

With each team member allocating different amounts of time to the project, these roles were not rigidly assigned. This enabled each team member to adjust to the demands of the highly-dynamic and fast-paced refactoring effort by participating in multiple roles whenever necessary. Specifically, meeting the short time constraint required team members to assume different roles at different times due to members having to fulfill their other duties as required by the realities of an academic work environment.

As a development methodology, we used eXtreme Programming techniques, i.e., rapid prototyping, pair programming, and (as aforementioned) unit testing [3]. These techniques significantly aided our project for several reasons. First of all, with multiple designs to pursue and evaluate, we needed to be able to make conclusions about their suit-

ability quickly [37, 39]. To achieve this objective, it was not always necessary to produce a fully functional version of mpiBLAST. In short iteration cycles, a small, architectural prototype was often sufficient to reveal the deficiencies of a design and thus determine its suitability.

Additionally, as with most software teams, our team members possessed complementary expertise, and no single member had sufficient knowledge required to accomplish the task independently. Thus, we needed to fuse two different areas of expertise possessed by the members of our team: parallel algorithmic design and software architecture implementation. Through pair programming we were able to utilize the combined expertise of individual members to create a synergy of ideas and talents [38]. This technique helped us to meet our objectives by enabling a concerted effort to minimize the number of programming errors and maximize productivity during initial prototyping. To clarify, whereas we employed pair programming to create initial skeletal prototypes, maturing the final version into a functioning system was accomplished individually.

Furthermore, impromptu discussions during our joint programming sessions often engendered novel ideas that warranted further evaluation. We were pleasantly surprised to find that some of the more off-the-wall ideas generated during these discussions were incorporated into our final design.

5. Refactoring Experiences

Before describing our refactoring experiences, we reiterate our main design objectives: maintainability and extensibility while still retaining high-performance. Achieving these objectives requires achieving the following goals: (1) retain high-performance guarantees across multiple platforms, (2) structure the system as a collection of reusable and interchangeable software components, (3) express dependencies and correspondences between different components, (4) flatten the learning curve for development and maintenance, and (5) avoid code duplication.

The motivation behind the pursuit of several of these goals is self-evident, such as guaranteeing high-performance, flattening the learning curve, and avoiding code duplication. With respect to flattening the learning curve, we were primarily concerned with making it easier to develop and maintain the application, leaving learning to program in ANSI C++ to the developer. We chose ANSI C++ as the implementation language because, (1) the NCBI Toolkit API is migrating to C++, (2) the newest version of mpiBLAST is a combination of ANSI C/C++, and (3) we needed to maintain cross-platform compatibility.

Our stated objective to refactor the codebase into a more modular design and the patterns of mpiBLAST development led us to structuring the system as a collection of

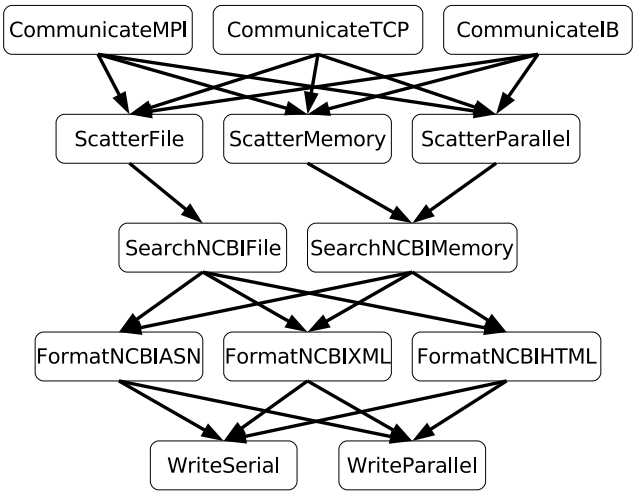

Figure 4. Module Correspondence Graph

reusable and interchangeable software components. As to what the component decomposition would be, it became apparent to us early on that the stages of the mpiBLAST algorithm (e.g., *Scatter*, *Search*, etc.), rather than the process roles (e.g., *Master* and *Worker*), should become the main components. The reason for this form of component decomposition is because it logically partitions the system into the units that are most likely to be modified. For example, it is highly foreseeable that a new search or write strategy would be implemented as the system evolves, whereas changing the master-worker paradigm to a peer-to-peer model would be less likely.

However, decomposing stages into components did not produce entirely independent components. In fact, like all component-based designs, inter-component compatibility is defined by their input and output types: if *Scatter* distributes fragments as files, then *Search* must accept files rather than memory buffers as input. Our final design had to express these dependencies and correspondences between components, preferably early on in the development process (i.e., during compilation vs. execution).

Next we discuss the different designs we pursued and how each satisfied our design objectives.

5.1. GoF Design Patterns

Seeking to find a pure object-oriented design for mpiBLAST, we tried to find a suitable solution that utilized design patterns. The design pattern that most closely captures our requirements for mpiBLAST was *Abstract Factory*. An *Abstract Factory* is a flexible means in controlling how different components in a system are created [16]. The details of creating multiple components in a system are encapsulated inside factory objects that are themselves expressed

```
template <class Communicate,
  template <class Communicate> class Scatter,
  template <typename Scatter> class Search,
  template <typename Search, class Communicate> class Gather,
  template <typename Gather, typename Search, class Communicate> class Write>
class Scheduler {  /** body **/  };
```

Figure 5. Template Definition for Scheduler in Parametric Polymorphic Design

only as abstract interfaces. A specific implementation of a factory creates a specific type of an entire system. Thus, the task of enforcing the compatibility between different components is handled entirely by a factory object. Furthermore, introducing new combinations of components is straightforward: it only requires implementing a new factory object.

However, in our case, this design suffered from a combinatorial explosion of factory objects. Although every component of mpiBLAST has input and output dependencies, sets of components and even some individual components could be replaced independently. As shown in Figure 4, *Communicate* (which provides generic communication primitives) is independent of every other component. On the other hand, *Search* dictates the version of *Scatter* that must be used: if *Search* uses files as input, *Scatter* must produce files as output. Thus, with an *Abstract Factory* design, we need to provide a unique factory object for every possible combination of components. This is a formidable challenge, as every extension of mpiBLAST that produces a new component would result in an explosion of new factory objects.

Further exacerbating this design, it is the developer of a new component who is responsible for creating new factory objects that enforce the correct usage of the new component. This means that the developer requires knowledge of all of the components within the new factory objects, thereby significantly raising the barrier to entry for new development.

While a different and/or novel design pattern, or a combination thereof, could have provided an elegant design for mpiBLAST-2.0, the use of most design patterns invariably involves using indirection and dynamic dispatch through virtual methods. Some sophisticated C++ compilers are capable of reducing the cost of such abstractions significantly, however, with our objective of cross-platform portability, we could not assume that all of the supported platforms would have such a compiler available. Therefore, we chose to look into designs in which the cost of abstractions would be minimized by being resolved at compile time. This immediately directed our efforts toward solutions that make use of C++ templates as their abstraction mechanism. (As it turned out, we did not pursue another design based on design patterns during this refactoring because we found an alternative design that satisfies all of our requirements.)

5.2. Parametric Polymorphism

C++ templates provide a powerful mechanism for generic programming. A class or a method can be parameterized with a template parameter that specifies the constraints on the type used. For example, the C++ Standard Template Library (STL) makes extensive use of templates not only to provide powerful functionality and diverse data structures, but also to enforce compatibility between STL classes and methods [34]. Specifically, the *sort* algorithm method of STL accepts template parameters of type *RandomAccessIterator*, thereby disallowing STL *list* to be sorted, because the iterator for STL *list* (i.e., *BidirectionalIterator*) is incompatible. However, STL *sort* works seamlessly with an STL *vector*'s *RandomAccessIterator*. Furthermore, such incompatibilities are signaled at compilation time as errors, and in regards to performance, template abstractions do not incur a runtime overhead.

Naturally, we attempted to utilize such template abstractions to enforce the compatibility requirements between mpiBLAST components. In this scheme, each mpiBLAST component was modeled as a template class whose template parameters defined the types of components used within the class. This forced the creation of components to have structural conformance: types used as template arguments had to have matching methods with exact names and signatures.

Unfortunately, adequately enforcing structural conformance caused the template definitions to become increasingly complex and unwieldy. For example, to create a *Scheduler* the following requirements had to be satisfied: (1) references to *Communicate*, *Scatter*, *Search*, *Gather*, and *Write* were needed, (2) *Scatter* had to use the same *Communicate* as *Scheduler*, (3) *Search* had to use the same *Scatter* as *Scheduler*, (4) *Gather* had to use the same *Search* and *Communicate* as *Scheduler*, and (5) *Write* had to use the same *Communicate* and *Search* as *Scheduler*. The C++ template definition for a parametric, polymorphic *Scheduler* can be seen in Figure 5.

While this complex arrangement may succeed in enforcing inter-component compatibility for a particular combination of mpiBLAST components, this solution is far from general. Specifically, we have now forced every *Write* to take exactly three parameters and for *Gather* to take exactly two parameters and so forth. Obviously such rigidity makes

it impossible to accommodate future extensions where components require different numbers of parameters.

To be fair, several techniques could alleviate this issue. We could provide adapter template functions that bridge between classes taking different number of input template parameters. However, such adapter functions are non-trivial to write, as it is not always possible to provide default template parameters. It is also possible to use larger blocks of template definitions as a single typename but the dependencies between template types inside such blocks cannot be enforced. For example, in the following definition, there is no guarantee that *Scatter* and *Search* use the same version *or* instance of *Communicate*.

```
template <typename Communicate,
  typename Scatter>
class Search {  /** body **/  }
```

From this complexity, it became clear to us that we needed to take advantage of an alternative, template-based, state-of-the-art, component-oriented design.

5.3. Mixins

A mixin is an abstract subclass through which one can extend the behavior of a variety of super classes [7]. In C++, a mixin can be implemented as a generic class with a template parameter specifying its super class:

```
template <class Super>
class Mixin : public Super { /** body **/ };
```

Mixin-based inheritance can provide a powerful mechanism for composing components. In this setup, different components participate in an inheritance relationship, in which the exact version of all of the components for a particular object is not specified until instantiation time. Furthermore, the inheritance tree is built using a bottom-up approach: subclasses are specified before superclasses. For example, *Cat<Animal> mixinAnimalCat*, specifies that *Cat* is an *Animal*. On the other hand, *Cat<Picture> mixinPictureCat*, specifies that *Cat* is a *Picture*. Notice that both definitions use the same mixin subclass *Cat*, and it is the superclass, *Picture* or *Animal*, that defines the functionality.

In our experience, a mixin-based design provides the required structural conformance between different mpiBLAST components while still making it possible to easily replace components. However, unlike a factory-based design, a mixin-based design specifies components only once in a single declaration. In other words, this scheme makes it impossible to use incompatible components because an mpiBLAST object combines components only through a single inheritance relationship. Furthermore, the template definition of the main mpiBLAST object takes template arguments specifying the types of each component used; it then instantiates exactly one instance of each of these components.

Despite the benefits of a mixin-based design, we discovered that it has several deficiencies. Specifically, while the phases of the mpiBLAST algorithm are represented as separate components, the *Master* and *Worker* roles are only defined implicitly. This makes it possible for *Master* to directly call a *Worker*-specific method and vice versa. Making such direct calls will introduce insidious consistency errors, as *Master* and *Worker* are disparate and distinct processes that do not share any memory address space. The only valid sharing of data between *Master* and *Worker* is through *Communicate*.

Separating the *Master* and *Worker* functionality through a coding convention proved to be insufficient, as the developer could easily bypass such restrictions. This realization led us to pursue a refinement of this design by explicitly separating the *Master* and *Worker* roles into distinct subcomponents.

5.4. Mixin Layers

Mixin layers is a flexible mixins-like design for implementing collaboration-based designs by assembling software components in layers in which each successive layer is represented as a collection of inner classes [31, 32, 33]. Both the enclosing class and its inner classes participate in an inheritance relationship with an abstract super class. Specifically, the enclosing class inherits from the enclosing super class, and each inner class inherits from its corresponding inner class in the super enclosing class. Additionally, this design allows functionality to be added with each layer in a flexible manner: a layer defines inner classes only for those objects for which it needs to add functionality. In C++, a mixin-layer looks like the following:

```
template <class Super>
class MixinLayer : public Super {
  class Inner1 : public Super::Inner1 {
    /** body **/
  };
  class Inner2 : public Super::Inner2 {
    /** body **/
  };
};
```

We tried two versions of this design: "mixin layers with general roles" (*Master* and *Worker*), and "mixin layers with refined roles" (*Master*, *Worker*, and *Common*). We detail our experiences with each below and argue that mixin layers with refined roles is best suited for mpiBLAST-2.0.

5.4.1. Mixin Layers with General Roles

Building on our classical mixins design described in 5.3, we added two inner classes to each component representing

```
class Base{
  class Common {  /** body **/  };
  class Master : virtual private Common {  /** body **/  };
  class Worker : virtual private Common {  /** body **/  };
};

template <class Super>
class MixinLayer : protected Super {
  class Common : virtual protected Super:: Common {  /** body **/  };
  class Master : virtual private Common, protected Super::Master {  /** body **/  };
  class Worker : virtual private Common, protected Super::Worker {  /** body **/  };
};
```

Figure 6. C++ Implementation of Mixin Layers with Refined Roles

the roles played by each mpiBLAST process: *Master* and *Worker*. This way, *Master* and *Worker* functionality was explicitly separated between these two classes. In other words, it was now impossible for a *Master* process to explicitly or inadvertently call a method in the *Worker* process and vice versa.

At first glance, a mixin layers with general roles design retains all of the advantages of classical mixins with the added improvement of strictly separating mpiBLAST process roles. However, the strict separation, despite its desirable properties, also makes it impossible for *Master* and *Worker* to share any common functionality. For example, both *Master* and *Worker* processes need to send messages. With no common functionality between them, both the *Master* and *Worker* inner classes have no choice but to duplicate all of the message sending primitives. This results in duplicating a substantial amount of code with all of the inherent negative consequences, such as having to modify multiple, but identical, pieces of code.

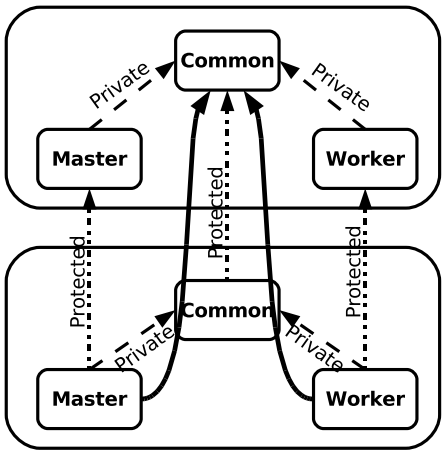

Figure 7. Graphical View of Mixin Layers with Refined Roles

5.4.2. Mixin Layers with Refined Roles

To retain the benefits of mixin layers without the issues of having to duplicate code we added another inner class, *Common*, that contains common functionality between *Master* and *Worker*. This common inner class serves as a base for the two other inner classes and to codify the *has-a* relationship between them, we use private inheritance. Figure 6 shows this mixin-layer with refined roles design as implemented in C++.

It is important to note that even with private inheritance, *Master* and *Worker* inner classes can still access the *Common* inner class in the upper layers through the *Common* inner class in its own layer as shown in Figure 7. It is the use of multiple inheritance for *Master* and *Worker* classes that enables this behavior. That is, *Master* and *Worker* inner classes inherit from their corresponding super inner class, as is normal for mixin layers, but also inherit from the *Common* class in their own layer. To avoid duplicating each *Common* class instance in the multiple inheritance tree, we use virtual inheritance when inheriting a *Common* class.

In a different language environment, we could have implemented the common functionality in the enclosing class. However, unlike Java, C++ does not automatically enable inner classes to access their enclosing class and doing so programmatically without pointers or references is non-trivial. Because a non-trivial implementation of mpiBLAST does not meet one of our primary design goals, namely keeping the codebase simple and elegant and thereby flattening the learning curve for development and maintenance, we chose not to implement the *Common* functionality in the enclosing class.

Table 1 shows a summary of how our five design goals are satisfied by the designs we considered and as one can see, a mixin layers with refined roles design satisfies all of our stated design objectives, simplifying the design for modularity and extensibility while retaining the requisite

	GoF Design Patterns	Parametric Polymorphism	Classic Mixins	Mixin Layers w/ General Roles	Mixin Layers w/ Refined Roles
Portable High Performance	-	+	+	+	+
(Re)Usable Components	+	+	+/-	+	+
Expressed Dependencies	-	+/-	+	+	+
Shallow Learning Curve	+	-	+/-	+	+
Avoids Code Duplication	-	+	+	-	+

Table 1. Summary Comparison of Design Fitness

performance characteristics. Specifically, the system is clearly decomposed into separate components based both on the stage of the mpiBLAST algorithm and also on the process role. Through the use of mixin layers, components are easily interchangeable and compatibility is still enforceable. This design enables mpiBLAST developers to maintain and extend the package in a well-modularized fashion with different functionality confined to well-encapsulated logical units (i.e., classes within mixin layers).

Lastly and most importantly, the refactored mpiBLAST remains a high-performance application. Unfortunately, comparing the performance of different versions of mpiBLAST is challenging due to the differences in the underlying search engine, NCBI BLAST. Since the release of mpiBLAST-1.4.0, NCBI has released BLAST four times with each release improving various qualities including performance. Therefore, although mpiBLAST-2.0 improves overall execution time in relation to mpiBLAST-1.4.0, e.g., by 43% for 32 workers, it is difficult to quantify precisely what percentage of this improvement is due to the new design. However, unlike mpiBLAST-1.4.0, the new design allows developers to quickly and straightforwardly incorporate each new NCBI BLAST engine, the prime avenue for improving performance.

6. Future Directions

The successful architectural refactoring makes mpiBLAST immediately amenable to new and exciting developments. We are looking forward to seeing what mpiBLAST developers will add to the project, now that the new design facilitates experimenting with new features and functionality. Anecdotal evidence is encouraging: we recently introduced our new implementation to several incoming graduate students at Virginia Tech and they found the new design intuitive and easy to follow.

One future direction we foresee for mpiBLAST is improving various algorithmic properties such as searching and Input/Output strategies. These improvements are vital because, as aforementioned, sequence databases are experiencing exponential growth and sequence searching is becoming more computationally intensive each year. But most importantly, the new design makes it possible to develop and incorporate these new features orthogonally to routine maintenance of the codebase.

From the software engineering perspective, we aim to further maximize the usability of the package for both developers and end-users. For developers, one area to explore is alternative ways to implement the *has-a* relationship between *Common* and *Master*/*Worker* roles. For end-users, we aim to improve usability. Most mpiBLAST end-users are not experts in computer science, nor do they aspire to become ones. Thus, it is the developers who are ultimately responsible for improving all aspects of the end-user experience. Not coincidentally, the modular design of mpiBLAST-2.0 is well suited to providing new UI facilities, thereby opening up an entirely new area of mpiBLAST development focused solely on the user experience.

7. Conclusions

In conclusion, the architectural refactoring of mpiBLAST has been an all-around positive experience that has provided us with multiple insights that we believe are applicable in other parallel bioinformatics search algorithms. Some of our insights are as follows:

1. High-performance software can be structured in a modular fashion without sacrificing performance using mixin layers.
2. Idiosyncrasies of the implementation language may significantly influence the final design.
3. Complex architecture refactoring can be accomplished by a small team in a short time frame using eXtreme Programming techniques.
4. Pair programming can effectively combine disparate expertise of team members into a synergy of ideas and approaches.
5. Rapid prototyping of experimental ideas can be an effective approach to evaluating multiple designs quickly and efficiently.

Consequently, insights 3-5 confirm several of the findings described in the research literature on eXtreme Programming [13, 22, 23].

Lastly, parallel bioinformatics software has earned the reputation of being difficult to develop and to use that we think is undeserved. As our experience shows, sound software engineering principles can and should be applied to the development and maintenance of this type of software. Our final design indeed satisfies all of our design objectives and thus mpiBLAST-2.0 proves that one does not have to give up performance to achieve desirable software engineering objectives such as modularity. We are optimistic that mpiBLAST-2.0 will enable scientists to concentrate on their own science rather than on computer science.

References

[1] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic Local Alignment Search Tool. *Journal of Molecular Biology*, 215:403–410, 1990.

[2] S. F. Altschul, T. L. Madden, A. A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped BLAST and PSIBLAST: A New Generation of Protein Database Search Programs. *Nucleic Acids Research*, 25:3389–3402, 1997.

[3] K. Beck. *eXtreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[4] K. Beck. *Test-Driven Development: By Example*. Addison-Wesley, 2003.

[5] D. A. Benson, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, B. A. Rapp, and D. L. Wheeler. Genbank. *Nucleic Acids Res.*, 30:17–20, 2002.

[6] BioBrew / NPACI Rocks. `http://bioinformatics.org/biobrew/`.

[7] G. Bracha and W. Cook. Mixin-based inheritance. In *ECOOP/OOPSLA*, pages 303–311, 1990.

[8] F. P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley Professional, 1982.

[9] M. Christensen, C. H. Damm, K. M. Hansen, E. Sandvad, and M. Thomsen. Design and evolution of software architecture in practice. In *TOOLS-Pacific*, 1999.

[10] Cray. `http://www.cray.com/solutions/life/applications.html`.

[11] A. Darling. mpiblast evolves: success, collaborations, and challenges. In *Bioinformatics Open-Source Conference (BOSC'2005)*, 2005.

[12] A. Darling, L. Carey, and W. Feng. The design, implementation, and evaluation of mpiblast. In *International Conference on Linux Clusters: The HPC Revolution 2003*, 2003.

[13] H. Erdogmus and L. Williams. A value-based analysis of pair programming. *The Engineering Economist*, 48(4):283–319, 2003.

[14] W. Feng. Green destiny + mpiblast = bioinfomagic. In *International Conference on Parallel Computing (ParCo)*, 2003.

[15] M. Fowler, K. Beck, and E. Gamma. *Refactoring: Improving the design of existing code*. Addison-Wesley, 2005.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[17] M. Gardner, W. Feng, J. Archuleta, H. Lin, and X. Ma. Parallel genomic sequence-searching on an ad-hoc grid: Experiences, lessons learned, and implications. In *ACM/IEEE SC2006: The International Conference on High-Performance Computing, Networking, and Storage*, 2006.

[18] Gold - Genomes Online Database. `http://www.genomesonline.org/`.

[19] IBM BlueGene. `http://researchcomp.stanford.edu/hpc/archives/BlueGene.pdf`.

[20] iNquiry. `http://www.bioteam.net/`.

[21] B. Leuf and W. Cunningham. *The Wiki Way, Quick Collaboration on the Web*. Addison-Wesley, 2001.

[22] M. Lindvall, V. Basili, B. Boehm, P. Costa, K. Dangle, F. Shull, R. Tesoriero, L. Williams, and M. Zelkowitz. Empirical findings in agile methods. In *XP/Agile Universe*, 2002.

[23] M. Marchesi, G. Succi, D. Wells, L. Williams, and J. D. Wells. *Extreme Programming Perspectives*. Pearson Education, 2002.

[24] F. Meyer. Genome sequencing vs. moore's law: Cyber challenges for the next decade. *CTWatch Quarterly*, 2, 2006.

[25] Orion Multisystems. `http://www.orionmulti.com/support/faq_mpiblast`.

[26] Penguin Computing / Scyld. `http://bioinformatics.org/biobrew/`.

[27] pmWIKI. `http://www.pmwiki.org/`.

[28] Rocketcalc. `http://www.rocketcalc.com/package.php?Key=15`.

[29] D. Roundy. Darcs: distributed version management in haskell. In *2005 ACM SIGPLAN workshop on Haskell*, 2005.

[30] Scalable Informatics. `http://www.scalableinformatics.com`.

[31] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. In *European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag LNCS 1445, 1998.

[32] Y. Smaragdakis and D. Batory. Mixin-based programming in c++. In *Generative and Component-Based Software Engineering Symposium (GCSE)*. Springer-Verlag, LNCS 2177, 2000.

[33] Y. Smaragdakis and D. Batory. Mixin layers: An object-oriented implementation technique for refinements and collaboration-based designs. *ACM Transactions on Software Engineering and Methodologies (TOSEM)*, 11(2):215–255, 2002.

[34] Stepanov, A and Lee, M. The Standard Template Library. Incorporated in ANSI/ISO Committee C++ Standard, 1995.

[35] Teragrid. `http://www.teragrid.org/`.

[36] J. Tran, M. Godfrey, E. Lee, and R. Holt. Architecture repair of open source software. In *International Workshop on Program Comprehension (IWPC)*, 2000.

[37] L. Williams and H. Erdogmus. On the economic feasibility of pair programming. In *International Workshop on Economics-Driven Software Engineering*, 2002.

[38] L. Williams and R. Kessler. *Pair Programming Illuminated*. Addison-Wesley Professional, 2002.

[39] L. Williams, A. Shukla, and A. I. Antón. An initial exploration of the relationship between pair programming and brooks' law. In *Agile Development Conference*, 2004.