A Maintainable Software Architecture for Fast and Modular Bioinformatics Sequence Search

Jeremy Archuleta, Eli Tilevich, Wu-chun Feng
Dept. of Computer Science
Virginia Tech
Blacksburg, VA 24061
{jsarch, tilevich, 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, thus adversely affecting maintainability and extensibility. Alleviating this shortcoming requires an architectural refactoring to improve maintenance and extensibility while preserving high performance.

Toward that end, this paper evaluates five different software architectures and details how each satisfies our design objectives. In addition, we introduce a novel approach to using mixin layers to enable mixing-and-matching of modules in constructing sequence-search applications for a variety of high-performance computing systems. Our design, which we call "mixin layers with refined roles", utilizes mixin layers to separate functionality into complementary modules and the refined roles in each layer improve the inherently modular design by precipitating flexible and structured parallel development, a necessity for an open-source application. We believe that this new software architecture for mpiBLAST-2.0 will benefit both the users and developers of the package and that our evaluation of different software architectures will be of value to other software engineers faced with the challenges of creating maintainable and extensible, high-performance, bioinformatics software.

1. Introduction

Bioinformaticists have been using mpiBLAST, a popular, parallel, bioinformatics package that runs on a computer cluster, for their research activities ever since we first released the package over four years ago [9, 10]. During these four years, 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 many, high-performance, cluster distributions such as [5, 7, 14, 15, 17, 19, 20, 21] and is an officially supported application at high-performance computing facilities such as [26, 27].

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 package. Alas, the enthusiastic support of this 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.

To ensure that mpiBLAST continues to benefit bioinformaticists, we have performed an architectural refactoring of the package with the goals of improving its maintainability and extensibility while preserving high performance to better support all of the mpiBLAST stakeholders. Specifically, end-users require easy-to-use interfaces and expedient support; system administrators require quick installation and upgrade procedures; and developers require a modular codebase to enable seamless maintainability and extensibility.

Despite the clear objective, our architectural refactoring undertaking presented several software engineering challenges. In the technical realm, our effort entailed refactoring 15K lines of code (LOC) written with little software engineering discipline into a high-quality software package adhering to the important software engineering principles of modularity, reusability, and encapsulation. It is unusual for a software system used in several major commercial cluster software distributions to consist of only 15K LOC, considering that many modern desktop applications often contain millions of LOC. However, the core codebase of mpi-

BLAST is a highly-optimized kernel that defines the overall structure of the system. Keeping the core small in size simplifies maintenance, fosters development, and facilitates extensibility. With these characteristics it is quite possible that the package size will grow substantially in size as it incorporates new features and enhancements.

Before embarking on the search for a new and better architecture for mpiBLAST, we analyzed why the package was valuable to its users. The answer was obvious: the major draw of mpiBLAST has always been its high-performance functionality (i.e., super-linear speedup). Therefore, the new design had to improve (or at the very least maintain) performance while providing the additional benefits of disciplined software engineering to its stakeholders. Achieving this balance between good performance and solid software engineering is a growing trend in modern software development.

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 a common software maintenance task. To help software engineers learn from our experiences, this paper details our evaluation of five different designs: four intermediate and one final, in terms of how they satisfy the requirements of a high-performance parallel bioinformatics open-source application. Our evaluation has concluded that the final design, which we call *mixin layers with refined roles*, is optimal for the task at hand. We believe that our experiences with and evaluation of each design will be of value to software engineers faced with the challenges of creating maintainable and extensible software for 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 refactoring experiences with several designs during this effort are reported in Section 4, with an analysis of the final design that we chose for mpiBLAST-2.0, mixin layers with refined roles, detailed in Section 5. Some future directions for mpiBLAST, as well as how the new design will support them, are outlined in Section 6. We conclude in Section 7 with our experiences and lessons learned from this architectural refactoring of mpiBLAST.

2. mpiBLAST Overview

The advent of genome sequencing has brought bioinformaticists a wealth of genetic sequence information accessible by way of large sequence databases. To search these sequence databases for regions of homology between an unknown query sequence and a known sequence residing in the database, researchers consume the vast majority of their compute cycles using the Basic Local Alignment Search Tool, commonly known as BLAST. Because the BLAST algorithm detects local 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, an important feature with wide-ranging impacts such as pathogen detection [12].

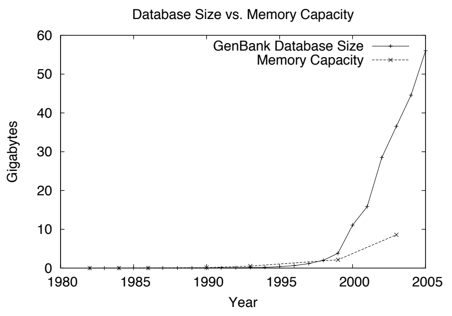

Figure 1. Comparison of the Growth of GenBank against the Growth of Memory Capacity

BLAST finds these regions of similarity quickly by using heuristics to prune the search space. However, the rapid search of these databases with BLAST can only be performed in a shared-memory, and usually sequential, environment. Compounding matters, until recently these databases could fit in main memory, but as shown in Figure 1 this is no longer the case. 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 [4, 13].

In 2002 and 2003, we developed an open-source software package, mpiBLAST, which augments the de facto 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). Since its release it has been downloaded over 40,000 times showing that mpiBLAST is of high utility to the bioinformatics community. It is also important to note that mpiBLAST is primarily designed for computer clusters which can range from only a few computers to hundreds and even thousands of computers. Therefore, it is safe to assume that mpiBLAST is running on over 40,000 computers, and quite likely on more than 100,000!

To take advantage of the processing power of computer clusters, the mpiBLAST algorithm follows a Master-Worker parallelization model that consists of three basic

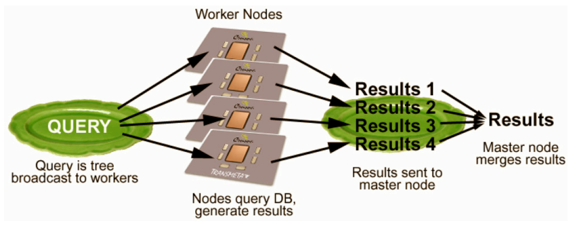
Worker Nodes
QUERY
Query is tree broadcast to workers
Nodes query DB, generate results
Results 1
Results 2
Results 3
Results 4
Results sent to master node
Results
Master node merges results

Figure 2. High-level View of mpiBLAST Algorithm

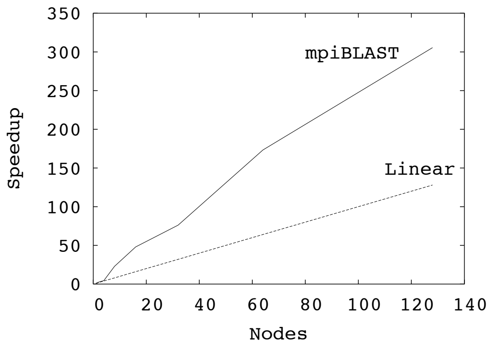

Figure 3. Performance of mpiBLAST 1.4.0

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 the database and query are segmented into pieces such that each compute node searches a portion of the database and a portion of the query. This gives the notable advantage that a super-linear speedup, shown in Figure 3, is obtained when the database being searched is too large to store in an individual compute node's memory. Furthermore, even if the original database fits in memory, mpiBLAST still improves throughput by finishing each search faster due to multiple queries being searched in parallel.

As sequence databases experience exponential growth and sequence searching becomes more computationally intensive each year, so grows the importance of mpiBLAST functionality to bioinformaticists [16].

3. Refactoring Objectives

The challenge of our refactoring effort was in preserving the high-performance properties of mpiBLAST-1.4.0 while simultaneously improving maintainability and extensibility. As shown in Figure 3, this version exhibits superlinear speedup across more than 100 processors through the use of query and database segmentation, sophisticated scheduling, pipelined results gathering, and asynchronous communication [8]. In addition, mpiBLAST-1.4.0 provides these performance advantages on multiple platforms (e.g., GNU/Linux, Windows, Mac OS X, BSD, and other Unix variants).

Past development efforts have primarily focused on improving the algorithmic quality of the package rather than the quality of its codebase. This pattern of development resulted in a "feature-rich but ad hoc" codebase, making every discovered defect tougher to correct and each new feature more difficult to implement. Thus despite its rich functionality and impressive performance, mpiBLAST became difficult to maintain and extend; the very success of the continued development of mpiBLAST depended on our ability to improve the quality of the codebase. At the same time, our improvements should not jeopardize the existing features of mpiBLAST – a maintainable and extensible package with inadequate performance and poor portability would quickly lose its utility. Therefore, the product of our refactoring effort had to satisfy *all* of the mpiBLAST stakeholders.

Our experiences with mpiBLAST-1.4.0 pointed toward improving the package's modularity as the main focus of our refactoring effort as modularity has long been recognized as a desirable software engineering objective [3, 18]. By focusing on creating a modular software package, the ability to encapsulate different pieces of mpiBLAST functionality in separate code modules should lead to improved maintainability and extensibility.

3.1. Maintainability

The users of mpiBLAST 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 [27] and System X [26]). The new architecture, therefore, should facilitate keeping the package up-to-date by finding and fixing bugs quickly or installing new versions.

The maintainability requirements of mpiBLAST are defined primarily by the developers of the package with the system administrators and end-users also having roles. 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. It is the mpiBLAST developers, however, who address and resolve the issues and make available a patch or new release of mpiBLAST to all interested parties through the mpiBLAST website.

The key to simplifying the maintenance process is to maximize the modularity of the new design, thereby enabling changes (e.g., bug fixes) on a per module basis and reducing ripple-effects. For example, mpiBLAST-1.4.0

is dependent on specific versions of the NCBI C Toolkit, thereby requiring a new and custom patch for every new release of the Toolkit (which occurs every six months on average). If all of the interactions with the NCBI Toolkit could be placed in a single mpiBLAST module, it would streamline this necessary (but tedious and error-prone) task, improving the package's maintainability and portability.

Modularity has the additional benefit of not requiring familiarity with the entire software package when making most modifications. For example, an issue with command-line processing should 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 should be located in an intuitively-named source file containing only command-line functionality. The placement of each module in its own source file further maximizes encapsulation and modularity.

Finally, improved modularity makes mpiBLAST more conducive toward an open-source development model in which multiple developers can focus on maintaining different parts of the software concurrently. This enables developers to make and integrate changes independently of 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% every year. Such challenges call for new and advanced algorithms and data structures to be integrated into mpiBLAST simply to allow the software to keep pace with the growth of the databases. For example, mpiBLAST-1.0 was run on the NT database (5.1 gigabytes in size) in 2003 and achieved an 170-fold speedup, finishing in 8 minutes. In 2005, mpiBLAST-1.4.0 was again run on the NT database (now more than 14 gigabytes in size), and even with a 305-fold speedup improvement, the execution time was *slower*, finishing in 10 minutes!

We foresee that future performance improvements to mpiBLAST will span the entire gamut of the application: better search algorithms, improved communication mechanisms, more intuitive user interfaces (UI), and efficient parallel Input/Output (I/O) strategies to name a few. Obviously, other developers will also make novel contributions, also possibly spanning the entire application. Therefore, if mpiBLAST fails to facilitate such contributions and thereby failing to keep performance on pace with the growth of database, it will quickly lose its utility.

To make the software package extensible, the new design should provide an intuitive and flexible design in which all developers can incorporate their novel algorithms and data structures with minimal effort. While we expect a certain level of knowledge from the 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 require knowledge relevant to the module(s) they are enhancing. Once again, improved encapsulation and modularity of the package will flatten the learning curve for developers who want to contribute various novel features and enhancements to the package.

4. Design Evaluation

Parallel bioinformatics is a young research area. From a technical perspective, a lack of proven architectural solutions available for this important but still emerging domain required us to evaluate multiple designs. We evaluated each design in terms of its fitness with respect to satisfying the following key objective: improving maintainability and extensibility while preserving high performance. Achieving this main objective requires satisfying the following design goals: (1) retain high-performance guarantees across multiple platforms, (2) structure the system as a collection of reusable and interchangeable software modules, (3) express dependencies and correspondences between different modules, (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, assuming that the developer is already proficient in ANSI C++. ANSI C++ was a natural implementation language choice for us because (1) the NCBI Toolkit API is migrating to C++, (2) the most recent version of mpiBLAST contains a combination of ANSI C/C++, and (3) we needed to maintain cross-platform compatibility. Furthermore, C++ is a *systems programming language* whereby developers have direct access to the low-level system components necessary to acheive fine-tuning of the parallel hardware architecture upon which mpiBLAST executes.

Our objective to refactor the existing codebase into a more modular design as well as mpiBLAST's development patterns led us to structuring the system as a collection of reusable and interchangeable software modules. Deciding on what the module decomposition should be was not much of an issue – it was apparent to us that the stages of the mpiBLAST algorithm (e.g., *Scatter*, *Search*, etc.) should be the primary modules, rather than the process roles (e.g., *Master* and *Worker*). This particular decomposition logically partitions the system into the units that are most likely to be modified. For instance, as the system evolves, it is more

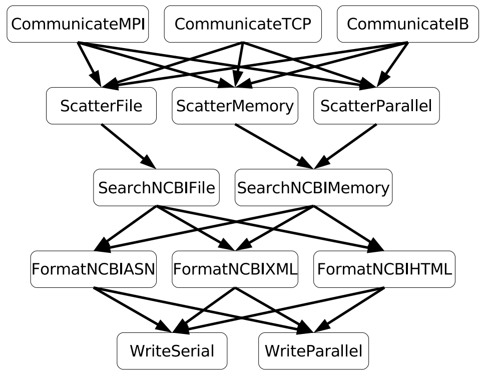

Figure 4. Module Correspondence Graph (compatibility is dictated with an arrow)

likely that a new search algorithm or write strategy will be developed rather than the parallelization model changing from master-worker to peer-to-peer.

However, the fact that each stage of the mpiBLAST algorithm can be represented as a separate module does not mean that the resulting modules are entirely independent of each other. In fact, like in most modular designs, the inter-module compatibility of mpiBLAST is defined by the input and output types: if *Scatter* outputs fragments as files, then *Search* must accept files instead of memory buffers as input, as seen in Figure 4. With this in mind, our final design had to express and resolve these dependencies and correspondences between modules, preferably early on in the development process (i.e., during compilation instead of execution).

In the following subsections we discuss the different designs that we evaluated and detail how each satisfies our design goals.

4.1. GoF Design Patterns

Seeking to find a purely object-oriented design for mpiBLAST, we tried to create a suitable solution utilizing 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 modules in a system are created [11]. The details of creating multiple modules in a system are encapsulated inside factory objects that are themselves expressed 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 modules is handled entirely by a factory object. Furthermore, introducing new combinations of modules 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 module of mpiBLAST has input and output dependencies, sets of modules and even some individual modules could be replaced independently. As shown in Figure 4, *Communicate* (which provides generic communication primitives) is independent of every other module. 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 needed to provide a unique factory object for every possible combination of modules. This is a formidable challenge, as every extension of mpiBLAST that produces a new module would result in an explosion of new factory objects.

Further exacerbating this design, it would be the responsibility of the developer of a new module to create new factory objects that enforce the correct usage of the new module. This means that the developer requires knowledge of all of the modules within the new factory objects, thereby significantly raising the barrier to entry for new development. While a different and/or novel design pattern may have provided an elegant solution, we ended up not pursuing design patterns further during this refactoring because we discovered an alternative approach that satisfies all of our design objectives.

Finally, the use of most design patterns invariably involves using indirection and dynamic dispatch through virtual methods. Most 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 other designs in which the cost of abstractions would be minimized by being resolved at compile time. This immediately directed our efforts toward solutions that utilize C++ templates as their abstraction mechanism.

4.2. 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 [25]. 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 with respect to performance, template abstractions do not incur runtime overhead.

Naturally, we attempted to utilize such template abstractions to enforce the compatibility requirements between mpiBLAST modules. In this scheme, each mpiBLAST module was modeled as a template class whose template parameters defined the types of modules used within the class. This forced the creation of modules 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*
5. *Write* had to use the same *Communicate* and *Search* as *Scheduler*.

The corresponding C++ template definition for a parametric, polymorphic *Scheduler* can be seen below:

```
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 **/  };
```

While this complex arrangement may succeed in enforcing inter-module compatibility for a particular combination of mpiBLAST modules, 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. Such rigidity makes it essentially impossible to accommodate future extensions where modules require different numbers of parameters.

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, and 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 **/  }
```

This complexity led us to consider an alternative, template-based, module-oriented design called mixins.

4.3. Mixins

A mixin is an abstract subclass through which one can extend the behavior of a variety of superclasses [6]. In C++, a mixin can be implemented as a generic class with a template parameter specifying its superclass:

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

Mixin-based inheritance can provide a powerful mechanism for composing modules. In this setup, different modules participate in an inheritance relationship, in which the exact version of all of the modules 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 modules while still making it possible to easily interchange modules. However, unlike a factory-based design, a mixin-based design specifies modules only once in a single declaration. In other words, this scheme makes it impossible to use incompatible modules because an mpiBLAST object combines modules only through a single inheritance relationship. Furthermore, the template definition of the main mpiBLAST object takes template arguments specifying the types of each module used; it then instantiates exactly one instance of each module.

Despite the benefits of a mixin-based design, we discovered that it had several deficiencies. Specifically, while the phases of the mpiBLAST algorithm are represented as separate modules, 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 in fact 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 implicit restrictions. This realization led us to pursue a refinement of this design by using mixin layers to explicitly separate the *Master* and *Worker* roles into distinct submodules.

4.4. Mixin Layers

Mixin layers is a flexible mixins-like design for implementing collaboration-based designs by assembling software modules in layers where each successive layer is represented as a collection of inner classes [22, 23, 24]. Both the enclosing class and its inner classes participate in an inheritance relationship with an abstract superclass. Specifically, the enclosing class inherits from the enclosing superclass, and each inner class inherits from its corresponding inner class in the super enclosing class. 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 typical mixin layer implementation looks as follows:

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

We evaluated two variations 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 evaluate mixin layers with refined roles in Section 5.

4.4.1. Mixin Layers with General Roles

Building on our classical mixins design described in Section 4.3, we added two inner classes to each module representing the roles played by each mpiBLAST process: *Master* and *Worker*. In this way, *Master* and *Worker* functionality was explicitly separated between these two classes and 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, mixin layers with general roles 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 replicating a substantial amount of code with all of the inherent negative consequences one would expect with code duplication such as having to modify multiple, but identical, pieces of code.

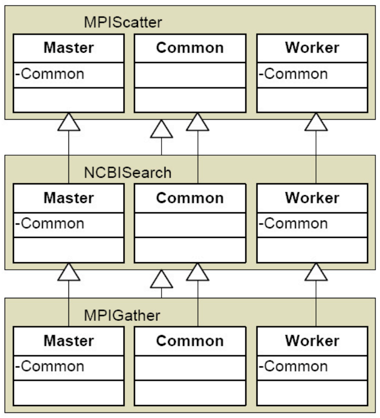

Figure 5. Example Software Architecture of Mixin Layers with Refined Roles

4.4.2. Mixin Layers with Refined Roles

To retain the benefits of mixin layers without the issue of having to duplicate code we added another inner class, *Common*, that contains common functionality between *Master* and *Worker*, exemplified in Figure 5. This variation on mixin layers has the advantages of allowing code reuse and avoiding code duplication.

As mentioned above, mixin layers with general roles did not satisfy all of our design objectives because common functionality between *Master* and *Worker* needed to be *duplicated* in each inner class. Clearly, being able to encapsulate common functionality in a separate class would allow the code to be reused rather than duplicated, positively affecting both maintainability and extensibility. With respect to maintenance, code reuse greatly reduces the possibility of *Master* and *Worker* from having different (and incompatible) implementations of the same functionality; in terms of extensibility, having a *Common* class makes it possible to enhance *Master* and *Worker* classes simultaneously.

Taking advantage of the refined roles required us to resolve the following technical challenge: finding an elegant implementation that would allow both *Master* and *Worker* classes to call methods in *Common*. This entailed experimenting with several approaches that codify the "has-a" relationship between *Common* and *Master* or *Worker* (i.e., inheritance and containment). We quickly determined that using multiple inheritance (*Master* and *Worker* inheriting both their mixin super class and *Common*) unnecessarily complicates the implementation. On the other hand, having *Master* and *Worker* contain *Common* as a member field implements the required functionality in a simple and intuitive way. Furthermore, to ensure that a single *Common* instance is available and shared by each layer, the *Common* field in each layer is a C++ reference, which according to the lan-

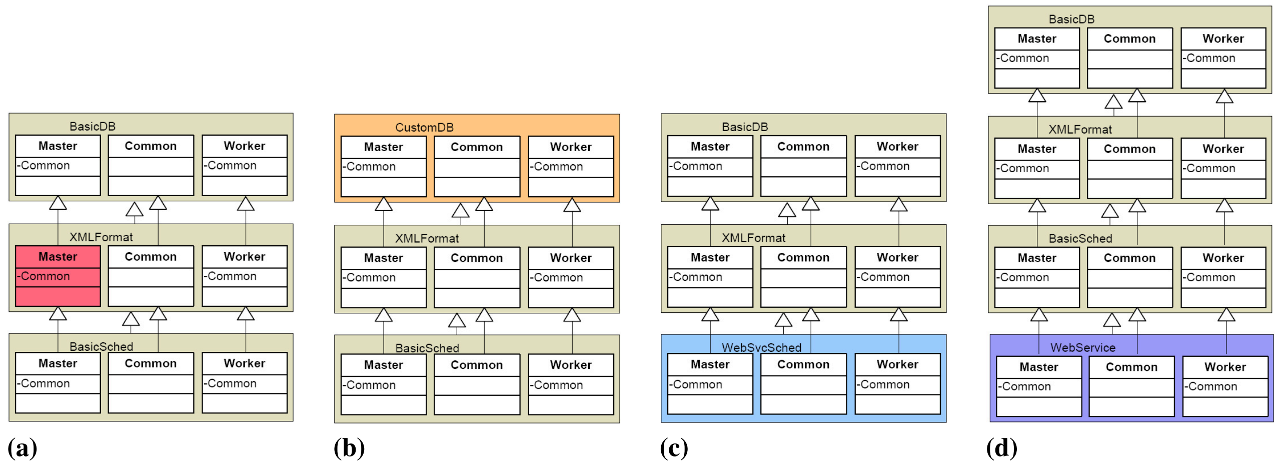

Figure 6. Composing Applications by Interchanging Layers (color highlights modification/addition)

guage rules must be initialized at construction time. This arrangement has an additional benefit in that the discipline imposed by the layered architecture extends to *Common* objects as well: even though a single (most derived) *Common* object is shared by all layers, each individual layer views the object as an instance of the *Common* class in its own layer.

5. Design for mpiBLAST-2.0

The software architecture that we chose for mpiBLAST-2.0 was *mixin layers with refined roles*, as it satisfied all of the design goals we had set for this refactoring effort. Specifically, this design provides portable high performance due to its use of templates, which are evaluated at compile time, as an abstraction mechanism. Mixin layers provide plug-in replaceable and reusable software modules that can be easily mixed and matched into different configurations. Furthermore, the dependencies between different layers and the roles expressed as inner classes are verified at compile time. That is, different layers become Lego™ blocks for constructing an application, and the shape of a block determines with which other blocks it can be combined. The high modularity of this design should flatten the learning curve for new developers, i.e., instead of learning the entire system, developers can focus on a particular layer (e.g., *Search*, *Gather*, etc.) that encapsulates self-sufficient units of functionality. Finally, code duplication is mitigated by creating modules that are capable of re-using the functionality of other modules in a highly flexible fashion.

Satisfying these design objectives indicated to us that mpiBLAST-2.0, implemented with mixin layers with refined roles, will be maintainable and extensible while also having high performance. Of course, only a realistic field study would confirm whether our refactoring efforts helps real developers, and we do plan to conduct such a study as future work.

To provide additional assurance that we in fact fulfilled our design objectives, we mapped three real-world mpiBLAST case studies to the new design. Influenced by our prior experiences with the package, these case studies are differentiated by both the type of users and the type of changes that are likely to be made to the package once it is released.

The first case study outlines a scenario of an end-user (e.g., a bioinformaticist) discovering an inconsistency in the formatting of search results and reporting this inconsistency to the mpiBLAST development community. A formatting problem most likely has to do with the formatting functionality, which is encapsulated within *Format*. This requires the developer to only understand this particular module in order to be able to correct the problem.

To perform this task successfully, the developer does not need to understand neither how the layers are composed together or the functionality of any other layer other than *Format*. Furthermore, knowing that it is the *Master* that is responsible for formatting the results, the developer need only focus her attention on the *Master* inner class within the *Format* layer, highlighted in Figure 6a. This simple but common case indicates that the new architecture of mpiBLAST makes it easy not only to find and fix bugs, but also facilitates the task of disseminating the changed source code back to the users of the package.

In the second case study, a pharmaceutical company uses mpiBLAST for its internal research and development activities and has a database stored in a format different from the database format used by the standard distribution of the package. (In fact, over the years mpiBLAST has been used

	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 (satisfied = '+', unsatisfied = '-')

by several commercial entities in that capacity.) Thus, an internal developer might be charged with the task of extending the functionality of mpiBLAST to be able to use their internal database format as input.

In developing a new Database (DB) layer, the original Database layer can be used as a working example on which to base the new functionality, seen in Figure 6b. Unlike the previous case study, in which the developer does not need to understand how layers are put together, here such an understanding is required to be able to interchange the original Database layer with a new custom layer. To determine the Database layer's correct placement in the mixin composition, the developer need only consult the existing composition of layers. If the functionality is incompatible with the composition, the issues will be revealed at compile time.

As another example in the same case study, the company might want to expose the functionality of mpiBLAST as a web service. Once again, mixin layers with refined roles makes it fairly straightforward to add such functionality: the specific requirements of the new service would determine which modules need to be modified. Two potential ways to provide this functionality are by creating a new replacement layer or by adding an extra layer, shown in Figure 6c and Figure 6d respectively. At any rate, this case study indicates that the new architecture makes mpiBLAST amenable to new extensions and enhancements with minimal effort on the part of the developer.

6. Future Directions

We look forward to seeing what mpiBLAST developers will add to the package now that mpiBLAST-2.0 facilitates experimentation with new features and functionality. One promising direction for new development that we foresee is improving various parallel algorithmic properties such as architecture-specific searching algorithms and parallel I/O strategies. These improvements are vital because, as aforementioned, the exponential growth of sequence databases is outpacing current sequential algorithmic enhancements. But most importantly, by separating the different phases of the application into modules, the new design makes it possible to develop and incorporate new features orthogonally to the routine maintenance of the codebase.

From the software engineering perspective, we aim to further improve the usability of the package for both developers and end-users. For developers, this requires quantifying the extent to which newly added features affect the maintainability of the package. If a custom Database layer or a Web service layer is added to the package, how many files, classes, and methods would need to be created or modified? How many files, classes, and methods would need to be modified in order to apply the *Format* layer bug fix from our case study? While we foresee that the high modularity of the new design will enable independent development of new features orthogonal to routine maintenance of the core functionality, it would be prudent to support these claims empirically.

For end-users, improved usability comes in the form of an improved user-interface. Most mpiBLAST end-users are not experts in computer science, nor do they aspire to become ones. Therefore, while the current command-line interface is a functional user-interface, it is unnecessarily complex requiring the user to remember the multitude of options available. Not coincidentally, the modular design of mpiBLAST-2.0 is well suited to providing new user-interface facilities (i.e., add a user-interface layer), thereby opening up an entirely new area of 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 tools.

First, Table 1 shows a summary of how our five design goals are satisfied by the designs we considered, As the table shows, each design satisfied some combination of our stated design objectives, however, only mixin layers with refined roles satisfies *all* of our stated design objectives. The deciding design objective for a "mixins-like" design was the easily expressed dependencies of this software architecture. While it may be possible to express dependencies using design patterns or parametric polymorphism, in our case, using a mixins-like implementation reduced the complexity of the resulting implementation. Interestingly, the deciding factor not to use a pure mixins approach was the lack of explicit separation of process roles, while the deciding factor not to use mixin layers with general roles was that the separation of process roles was too strict. For a parallel, open-source, bioinformatics application, mixin layers with refined roles enables maintenance and extensibility to be orthogonal development processes with functionality confined to well-encapsulated logical units. Furthermore, the modules are easily interchangeable allowing for a wide variety of specialized applications with compatibility enforced at compile time.

Second, high-performance bioinformatics software can be structured in a modular fashion without sacrificing performance using mixin layers. As expected, preliminary evaluation of (our prototype of) mpiBLAST-2.0 indicates that it performs well in comparison to mpiBLAST-1.4.0 – mpiBLAST-2.0 maintains or even decreases overall execution time on average. A more rigorous and extensive performance evaluation of mpiBLAST-2.0 (e.g., broad range in the number of processors, different BLAST parameters, different architectures, phase profiling and analysis, etc.) will be conducted over the next few months but is outside the scope of this paper.

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 of mixin layers with refined roles satisfies all of our design objectives demonstrating 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.

Acknowledgments We would like to thank Shawn Bohner, Spencer Rugaber, Yannis Smaragdakis, and the anonymous reviewers whose comments have helped to improve this paper. This research was supported by the Department of Computer Science at Virginia Tech.

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] C. Baldwin and K. Clark. *Design Rules: The Power of Modularity*. MIT Press, 1989.

[4] 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.

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

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

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

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

[9] 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.

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

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

[12] J. Gans, W. Feng, and M. Wolinsky. Whole genome, physics-based sequence alignment for pathogen signature design. In *12th SIAM Conference on Parallel Processing for Scientific Computing*, 2006.

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

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

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

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

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

[18] D. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the Association of Computing Machinery*, 15(12):1053–1058, 1972.

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

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

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

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

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

[24] 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.

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

[26] Advanced Research Computing. `http://www.arc.vt.edu/`.

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