Bioinformatics Advance Access published September 23, 2004
Bioinformatics © Oxford University Press 2004; all rights reserved.

GLAD: a system for developing and deploying large-scale bioinformatics grid

Yong-Meng TEO[1,2], Xianbing Wang[1,2] and Yew-Kwong NG[1]

[1]Department of Computer Science, National University of Singapore, Singapore 117543
[2]Singapore-MIT Alliance, Singapore 117576

Abstract

Motivation: *Grid computing is used to solve large-scale bioinformatics problem with gigabytes database by distributing the computation across multiple platforms. But up to now in developing bioinformatics grid applications, it is extremely tedious to design and implement the component algorithms and parallelization techniques for different classes of problems, and to access remotely located sequence database files of varying formats across the grid. Herein, we propose a grid programming toolkit, GLAD (Grid Life sciences Applications Developer), to facilitate the development and deployment of bioinformatics applications on a grid.*

Results: *GLAD has been developed using ALiCE (Adaptive scaLable Internet-based Computing Engine), a Java-based grid middleware, which exploits task-based parallelism. Two bioinformatics benchmark applications, distributed sequence comparison and distributed progressive multiple sequence alignment, have been developed using GLAD.*

Availability: *GLAD and ALiCE are available at http://www.comp.nus.edu.sg/~teoym/alice.htm*

Contact: *teoym@comp.nus.edu.sg*

1 Introduction

Bioinformatics encompasses the methodologies of operating on biological information in order to facilitate research in molecular biology. Common operations on biological data include analysis of protein structures, comparison of genome sequences, visualization of sequence alignment results, and placement of sequence databases.

The volumes of biological information stored in bioinformatics databases hosted by genome research centers such as the National Center for Biotechnology Information (NCBI homepage) and Genome Institute of Singapore (GIS homepage) are enormous. It must be noted that, in the context of bioinformatics, the term database refers to a large set of catalogued sequences, and does not encompass the capabilities of standard data management systems such as data sharing and hashing. Each sequence database file is in the range of gigabytes of data. In a distributed environment such as a wide-area grid, it may be necessary for bioinformatics programs that are executing in the grid to access sequence data in bulk from several geographically disparate databases.

The essential problem in bioinformatics today is the lack of adequate support tools to bridge the gulf between information technology and the life sciences comprehensively. There have been many prolific examples of bioinformatics applications that are able to provide solutions extensively to specific problems in life sciences research (Altschul *et al.*, 1990, Fratini *et al.*, 1982, and Laskowski *et al.*, 1993). There have also been extravagant efforts to integrate existing bioinformatics resources to promote knowledge sharing amongst molecular biologists on a wide level. Some of these software projects have bore fruit after years of research and development, and their suite of products utilized by the world's leading bioscience consortiums (Bosson and Riml, 2003). The goals of the individual projects differ, however, depending on the mission of their developers. Table 1 illustrates a list of some successful bioinformatics projects in history.

Project	Objective	Developers	Base Language
BLAST (Altschul *et al.*, 1990)	Sequence comparison and search	NCBI	C
CLUSTAL (Higgins and Sharp, 1988)	Progressive Multiple Sequence Alignments	Higgins, et al., SGI	C, C++
CURVES (Lavery and Sklenar, 1988)	Nucleic acid helical analysis	Lavery, et al.	C
GENSCAN (Burge and Karlin, 1997)	Gene structure predictions for genome sequences	MIT	Perl
ISYS (Siepel *et al.*, 2001)	Integrated environment for bioinformatics resources	NCGR	Java
PROCHECK (Laskowski *et al.*, 1993)	Knowledge-based analysis of protein structures	Thornton, et al.	VAX Fortran
SRS (Etzold and Argos, 1993)	Database integration and biological information search	Lion Bioscience	C++, Java, Perl, Python

Table 1. Bioinformatics Software Examples

However, bioinformatics applications typically require a high performance orientation, and insufficient work has been done to provide an environment for the development and deployment of flexible, modular bioinformatics software solutions that can be parallelized for execution on a large scale, such as a wide-area grid environment. Grid computing has the overwhelming potential to apply supercomputing power to address a vast range of bioinformatics problems. Problems composed of non-trivial algorithms and operating over large datasets, such as fragment assembly of DNA molecules (Pevzner *et al.*, 2001), three-dimensional structure protein analysis and prediction (Rost, 1998), and genetic linkage analysis

(Terwilliger and Ott, 1991) can all be deployed for parallel execution on grids by adopting a divide-and-conquer approach, decomposing the problem into smaller task granules that can be distributed to the compute resources. The difficulty posed in grid deployment stems from implementing the parallelization steps of each algorithm component, which is typically tedious, error-prone and, consequently, consumes extensive developmental time. This is especially crucial for large-scale bioinformatics problems, which usually comprises of several algorithmic stages that involve vigorous computations. Furthermore, efficient grid computing middleware is necessary in order to facilitate the partitioning and dissemination of tasks to the available compute resources for execution.

The close associations attributed to grid computing and the life sciences in recent years will probably see a greater demand for more flexible integrated systems that can support the creation of medium- to large-scale bioinformatics and biomedical grids. This is a crucial step towards globalization of the life sciences industry, as it enhances the means to share biological knowledge on a worldwide basis, thereby providing opportunities to foster productive collaborations between bioscience enterprises.

In this paper, we present GLAD (Grid Life sciences Applications Developer), which is a Java-based programming environment for bioinformatics problems on grids. GLAD is built on the ALiCE (Adaptive scaLable Internet-based Computing Engine) (Alice, 2000) grid core middleware, and sets out to provide bioscience researchers with an efficient workbench to implement distributed bioinformatics applications for deployment on a grid. GLAD provides the underlying mechanisms to handle the extraction and shipment of biological data across the grid. The toolkit comprises of an extension layer, which encapsulates a set of commonly used bioinformatics algorithm components that can be adopted in the development of large applications.

This paper discusses the design and implementation of the GLAD application toolkit, demonstrating how an extensible library of bioinformatics algorithm components and a set of Java-based constructs can facilitate the development and deployment of medium-to large-scale bioinformatics problems on grids. The toolkit architecture hides the underlying ALiCE grid infrastructure, remote sequence data fetching and parsing mechanisms, and task communications from the user, enabling the user to concentrate on mapping the problem into the GLAD environment. It also allows previously developed GLAD applications to be added to the extension library and reused as components in future applications development.

It shows how bioinformatics problems that involve regular and semi-regular parallelization patterns (Trelles, 2001) and operate over huge biological sequence databases can be efficiently deployed for scalable execution on homogeneous and heterogeneous grid environments, with the adoption of task level parallelism. Case studies of applications developed using GLAD include distributed sequence comparison (regular parallelization) presented in this paper and distributed PMSA (semi-regular parallelization).

2 ALiCE middleware

2.1 ALiCE architecture

ALiCE is a grid computing core middleware developed in the School of Computing, National University of Singapore. Unlike Globus (Foster and Kesselman, 1997), which is a collection of fundamental grid construction tools and focuses on low-level services, ALiCE is a grid computing system designed to aid the implementation of general-purpose applications and focuses on application programming models for grid environments. The current Globus Toolkit version 3.2 is a reference implementation of the evolving OGSA grid standard. We plan to align ALiCE to conform to OGSA once it is finalized. The ALiCE grid architecture consists of three main layers, supported by a set of existing Java technologies, as illustrated in Figure 1.

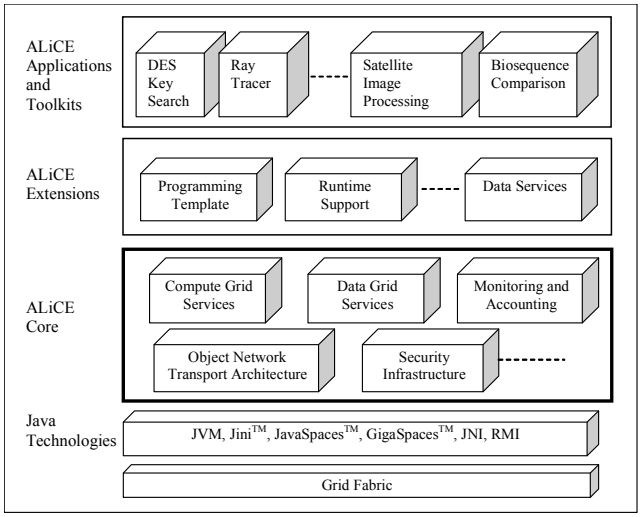

Figure 1: *ALiCE Grid Architecture*

Resource discovery and allocation, as well as object communications within ALiCE, are realized using Jini (Jini, 2001) and JavaSpaces™ (Hupfer, 2000). GigaSpaces™ (GigaSpaces, 2002), a commercial cousin of JavaSpaces, is adopted in a more recent implementation of ALiCE. GigaSpaces is different from JavaSpaces in the sense that the former implements distributed-shared memory physically by coupling together several spaces hosted on different machines, while the latter merely provides a logical distributed-shared memory model.

ALiCE Core is the central engine of the grid system, comprising of a suite of basic services that provide the functionalities of a grid. Compute Grid Services schedules tasks for computation, besides handling resource management, discovery and allocation. Data Grid Services takes care of the access, location, management and integrity of data within the grid. Other ALiCE Core services include security management, ONTA (*Object Network Transport Architecture*) for communications within the grid, and a monitoring and

accounting framework to keep track of vital statistical information of the grid environment, such as the utilization of each of the participant resources.

ALiCE Extensions encompasses the ALiCE runtime support infrastructure for application execution, and provides the grid application developer with an API. Finally, the *ALiCE Applications and Toolkits* layer is a non-exhaustive collection of grid applications and programming models developed using the ALiCE user API. This is the only layer of the ALiCE grid architecture that is visible to application users.

2.2 ALiCE runtime system

The ALiCE runtime system is an integration of the Compute Grid Services and ONTA components from the ALiCE Core layer in the grid architecture. It consists of the following four processes as shown in figure 2:

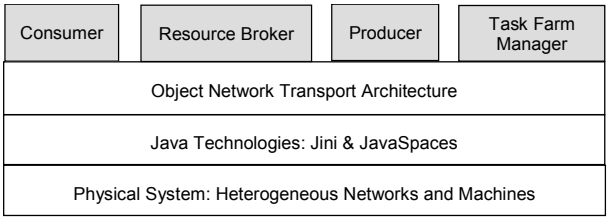

Figure 2: ALiCE Runtime System Framework

- **Consumer** is the users' entry point to an ALiCE grid for submitting applications to execute and collecting the corresponding results.
- **Producer** runs on any machines that volunteer their compute cycles to an ALiCE grid. It retrieves tasks generated from ALiCE applications, executes them and returns the results to the consumer.
- **Resource Broker** coordinates and controls the scheduling of applications and tasks, ensuring that all concurrently executing applications can complete within satisfactory turnaround times, and that all the producers within the grid are well utilized. It also registers, manages and allocates producers and other grid resources.
- **Task Farm Manager** generates tasks for ALiCE applications. ALiCE supports task farm managers for developmental languages other than Java, and this relieves the resource broker of the problem of handling non-Java application codes.

A consumer and a producer can be simultaneously run on a common machine. However, resource brokers and task farm managers must be run on separate nodes. The ALiCE runtime system is robust and has the capability to extend to wide-area grid environments, which would involve multiple heterogeneous resources.

2.3 ALiCE API

To support the development of grid applications and domain-specific application programming toolkits, ALiCE programming Template provides an API that adopts a *TaskGenerator-ResultsCollector* programming model, encompassing the following four extensible classes:

- **TaskGenerator** generates the application's tasks on a task farm manager machine. It implements a method process that generates a new task and sends a reference to the task back to the resource broker machine for scheduling.
- **Task** models a task object generated by an application, and comprises of a series of computations to be performed by a producer machine.
- **Result** models a result object returned from a producer machine to the resource broker.
- **ResultCollector** is the entry point class of the ALiCE grid system, handling application user administrative issues such as data input and results visualization at a consumer machine. It implements by retrieving a result object from the relevant resource broker.

An application, with its codes encapsulated in a .jar file, is submitted at a consumer machine to the resource broker, which siphons the codes to a task farm machine, where the *TaskGenerator* is run. The generated *Task* objects are then disseminated to the producer machines for execution, returning the computation results in a *Result* object to the resource broker. In the interactive mode, the *ResultsCollector* running on the consumer machine retrieves the *Result* objects. In the batch mode, the resource broker stores the results.

The ALiCE API enables grid application developers to exploit the distributed nature of the ALiCE grid without needing to know about the technologies for communications and dynamic code linking, by providing the technical functionalities at abstract level.

3 GLAD Toolkit Architecture

Though ALiCE provides the runtime environment and programming APIs for grid applications, there are still lots of works which have to be done for developing various grid applications by using ALiCE middleware. For example, how to design user portal, how to parallize jobs, and how to generate biological tasks, etc., these issues are related to underlying implementation of ALiCE. For different applications in the same fields, these issues can be reused. Thus, we develop GLAD to provide programming assistance for bioinformitics grid applications.

GLAD is a bioinformatics application workbench on the ALiCE grid architecture for successfully deploying large-scale bioinformatics applications on wide-area grid environments. It provides programming assistance in the development of Java-based distributed bioinformatics applications by reducing the burden posed in implementing the parallelization, and allowing the user to invoke reusable basic components. The GLAD workbench is portable to any hardware platforms that run a Java Virtual Machine and support the ALiCE runtime system.

3.1 GLAD architecture

The GLAD architecture comprises of four constituent layers, as illustrated in Figure 3.

3.1.1 Execution control layer

An effective parallel execution control system should minimize the overhead incurred from data communications and ensure that parallel execution results in reasonable and significant speedup. These issues are especially important when considering the grid deployment of bioinformatics applications, which may process operations on up to tens of gigabytes of biological sequence data from several foreign sources. Then good performance and scalability can be achieved.

Execution control in GLAD is realized by five component mechanisms, as discussed in the following:

- **Parallelization engine** is responsible for the effective and efficient partitioning of the various algorithmic stages of a given problem into tasks for parallel execution. This enables the programmer to concentrate on specifying the biological computational routines, instead of being bogged down by the tedious routines of task mapping.

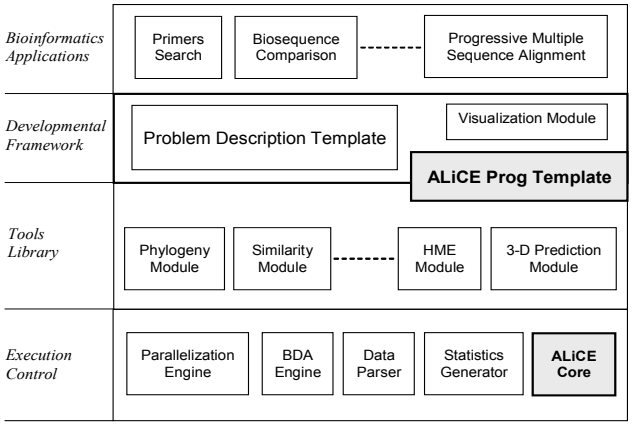

Figure 3: GLAD Architecture

- **Biological Data Access engine** facilitates fast and reliable access to data stored in biological databases during the application's execution. When the BDA engine is invoked during the application initialization phase or a task's computation, a connection is established with the corresponding database host machine, fetching only the specified block(s) of sequences from the database. A caching scheme is provided at the ALiCE consumer and producer machines running the application, so that future runs that include this dataset can access the data locally.
- **Data Parser** serves as a sequence data pre-processing mechanism. It inspects a given sequence file, determines the sequence description format adopted, and retrieves sequences from the file for processing by the application. The BDA engine uses it to extract the appropriate sequences at remotely hosted databases.
- **Statistics Generator** meta-details about the experimental dataset, such as its cardinality, the average length and corresponding standard deviation for all the sequences, the presence of updates, and its download time, are valuable information that are derived here. Besides dataset information, this component also monitors the performance of applications, keeping track of the task generation time, mean task processing time and the overall turnaround time of each GLAD application run.

3.1.2 Tools library

A wide variety of long-existing bioinformatics algorithms are commonly used in the development of larger applications. For instance, the maximum parsimony method (Fitch, 1977) is a fundamental character-state method in the area of evolutionary science, while the (Fitch and Margoliash, 1967) approach is a well-established weighted distance-matrix method for the construction of deep phylogenetic trees. The Smith-Waterman dynamic programming algorithm (Monge and Elkan, 1996), on the other hand, is commonly used for the computation of similarity scores of sequence pairs.

To simplify the work of large-scale, multi-stage bioinformatics application developers is to provide a library of such algorithms, so that the programmers can invoke the appropriate tools as they need. This potentially reduces developmental time and allows for greater attention to be placed on the more complicated portions of the problem. This approach also promotes modularity and code reusability, which are essential features of grid systems. The tools library in GLAD is extensible and non-exhaustive, thereby enabling newly implemented bioinformatics algorithms, including even GLAD applications, to be added to the set.

3.1.3 Applications development

The development of applications using the GLAD toolkit is supported with the help of a Problem Description Template (PDT), which is really a Java-based programming template for users to model their bioinformatics applications. This template itself was implemented using the ALiCE programming template for grid deployment facility. It allows for development at a certain level of abstraction by keeping the details of grid programming transparent to the programmer, leaving the programmer with the simple job of filling up a set of predefined methods.

The developmental framework also permits the programmer to customize the application's user interface by offering a number of standard visualization components for the graphical illustration of DNA and protein sequences, textual display of results, development of user interfaces for general parametric inputs associated with bioinformatics computations, and presentation of statistical information regarding the application's dataset. However, the programmer is free to implement his/her own visualization without the aid of any of these provisions.

3.2 GLAD Application Structure

Figure 4 shows the conceptual model of the structure of a GLAD bioinformatics application and its mapping into ALiCE. In the GLAD execution paradigm, a bioinformatics application is viewed as a composition of one or more successive biological stages, and comprises of two parts: the *user portal* and the *problem description*.

The user portal enables interactions between the application user and the application itself, obtaining parametric inputs for the problem during the initialization phase, and reporting the results upon completion of the application's execution in a manner subject to the user's customization. The user portal runs on the consumer machine in the ALiCE runtime system, and its underlying implementation is based on the ALiCE *ResultsCollector* process.

The problem description is a sequence of *biological stages* that constitute the bioinformatics problem modeled in the application. The stages are executed chronologically, and each stage can either be executed sequentially or deployed for parallel execution in a distributed environment using the ALiCE runtime system. In the latter case, the stage is comprehensively decomposed into *biological tasks* that are responsible for solving subparts of the problem associated with it. The task partitioning is performed by the *parallelization routine* for that stage. The parallelization routines required for all the biological stages involved in the application are collated by the ALiCE *TaskGenerator* process running at the resource broker. Biological tasks are scheduled for execution at the producer machines, where the task routines are processed, which may involve the invocation of algorithmic tools in the GLAD bioinformatics tools library.

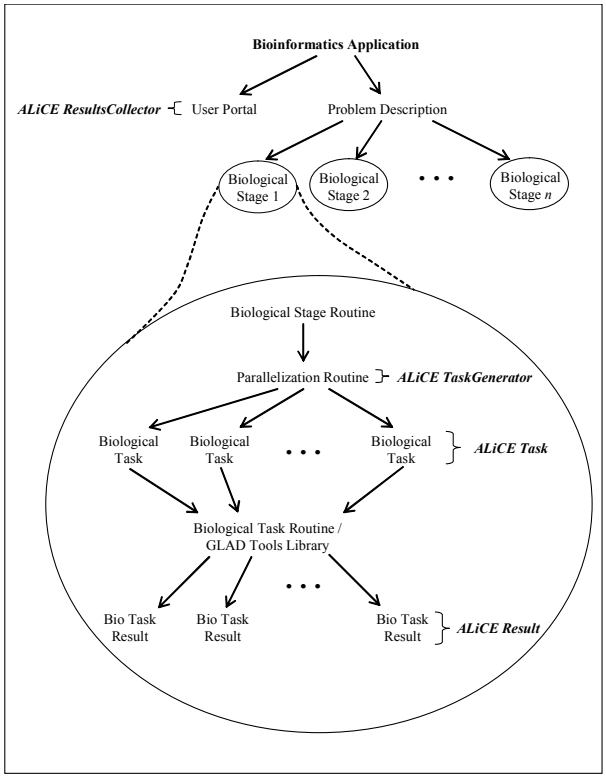

Figure 4: Structure of a GLAD Application

4 GLAD Library Implementation

The GLAD library is implemented based on Java, and the underlying distributed execution engine is supported by the ALiCE runtime system. In the GLAD paradigm, a bioinformatics problem can be modeled by several biological stages, and each stage generates numerous biological tasks during parallel execution. This can be readily described by the object-oriented programming approach that is being advocated by the Java language. Furthermore, the parallelization and task execution routines can be simply translated into ALiCE, which provides very generic constructs for all kinds of grid applications. It is this element of simplicity that enables ALiCE to be an outstanding infrastructure tool for developing domain-specific programming models and toolkits, such as GLAD.

The GLAD library is a structured composition of two different types of classes, namely *developmental classes* and *control classes*. Developmental classes are the ones that model the problem, while control classes are responsible for the underlying mechanisms that support the application's execution. GLAD's Problem Description Template (PDT) provides the environment in which the application programmer can model the bioinformatics problem.

4.1 Developmental Classes

GLAD supports distributed bioinformatics applications development with five main classes, as illustrated in Figure 5. Each class provides a set of primitives that can be manipulated in the course of implementation, and a list of abstract methods that are to be filled in by the programmer to dictate the different execution routines involved.

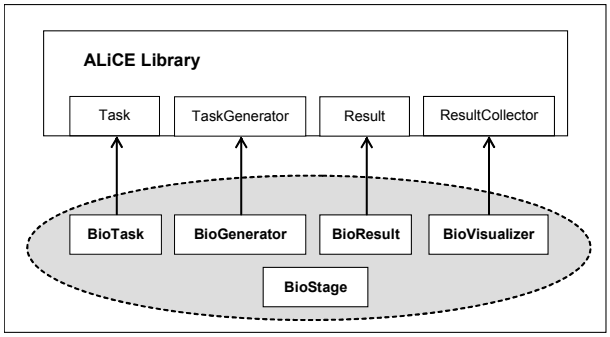

Figure 5: GLAD Developmental Classes

The *BioStage* class models a biological stage in a given application, describing the algorithms involved in that stage and providing parallelization capabilities that are kept transparent from the developer. However, the developer can override the parallelization routines when sophisticated parallelization strategies are required to address the problem.

The *BioTask*, *BioGenerator* and *BioVisualizer* classes are inherited from the three major classes in the ALiCE programming template. *BioTask* models a biological task that is to be scheduled for processing at the ALiCE producer machines. *BioGenerator* runs the biological task generation codes at the task farm manager. *BioVisualizer* is the visualization framework for the GLAD application. The *BioResult* class, on the other hand, is an extension of ALiCE's *Result* class, and is used in the propagation of results produced from the computation of a *BioTask*.

4.2 Problem Description Template

A distributed bioinformatics application can be implemented using the Problem Description Template (PDT) provided by the GLAD library. This programming template, illustrated in Figure 6, comprises of the four developmental classes above.

PDT essentially highlights the methods that the GLAD application developer has to implement in the corresponding subclasses in order to successfully deploy a distributed bioinformatics application on a grid system. The developer will typically declare problem-specific attributes, data structures as well as subroutines in the subclasses. In general, the developer is required to provide the following five basic items:

- linkage of the user interface and visualization components
- application execution logic in terms of stages
- code for generating the biological tasks for each problem stage
- logic for computation of each type of biological task, and
- code for processing of results from the execution of each stage

The application's execution commences from the *BioVisualizer* running on the ALiCE consumer machine, with the entry and exit points of the distributed computation occurring at the StartApplication() method. One or more *BioStage* processes will be created and activated sequentially. Each *BioStage* spins off a parallel computation in the ALiCE runtime environment, in which *BioTask* objects are generated by the *BioGenerator* and disseminated to the producer machines for processing. The results of each task computation are written to a *BioResult* object that is returned to the *BioVisualizer* at the consumer for post-processing by the HandleResult() routine.

BioStage Template

```
public class BioStage
{
 /* Class attributes */

 public BioStage(BioVisualizer v)

 public void StartStage()
 {
  // Place the algorithm associated
  // with this stage here. You may use
  // the Parallelize<x>() suite of
  // task partitioning methods.
 }

 protected void Parallelize()
 {
  // Place your customized
  // parallelization algorithm here.
 }
}
```

BioTask Template

```
import alice.consumer.*;
public class BioTask extends Task
{
 /* Class attributes */

 public BioTask()

 protected void ProcessTask()
 {
  // Place your task execution codes
  // here.
 }
}
```

BioGenerator Template

```
import java.io.*;
import java.util.*;
public class BioGenerator
extends TaskGenerator
{
 /* Class attributes */

 public void main(String[] args)
 {
  // Place your biological task
  // generation routine here.
 }
}
```

BioVisualizer Template

```
import alice.result.*;
import java.awt.*;
import java.net.*;
import java.swing.UIManager;
public class BioVisualizer
extends ResultsCollector
{
 /* Class attributes. */

 public BioVisualizer(int numStages)

 protected void StartApplication()
 {
  // Place your application control
  // codes here.
 }

 protected void HandleResult(int stage)
 {
  // Place your stage-wise result
  // processing codes here.
 }
}
```

Figure 6: GLAD Problem Description Template

4.2.1 BioStage Template

GLAD defines a bioinformatics application as an exhaustive sequence of biological algorithmic stages, where each stage is responsible for addressing a portion of the entire problem. The *BioStage* class provides the features and procedures associated with each stage.

The main procedure of the class is the StartStage() method, which is implemented by the developer in the sub-classes to dictate the algorithm involved with the respective biological stages.

The developer is allowed to implement the task partitioning procedure for more complex parallelization techniques in extended classes of *BioStage* via the Parallelize() method. Essentially, this process involves writing codes that generate a task partitioning schedule, which will be transmitted to the *BioGenerator* process via the resource broker.

The single argument, vsr, of the constructor refers to the *BioVisualizer* object that accounts for the user inputs and visualization for the application.

4.2.2 BioTask Template

In the GLAD application execution structure, each biological stage may be decomposed into multiple disparate biological tasks that can be processed in parallel according to the parallelization pattern of the algorithm associated with the stage.

The developer has to fill up the ProcessTask() method with the codes that account for the computation of the biological task, which may involve, for instance, the comparison between blocks of remotely located protein sequences or the reassembling of two given fragments of DNA molecules. The results that are derived from the

task computation are to be stored in a data structure, result, which is a *BioResult* object.

4.2.3 BioGenerator Template

The *BioTask* objects must be created centrally before they can be siphoned to the pool of producers for computation. This is achieved by the *BioGenerator* class, which generates them appropriately according to the task partitioning schedule received from the application's visualizer running on the consumer machine in the course of each biological stage. The *BioGenerator* class is a subclass of the *TaskGenerator* class in the ALiCE library, and can be used to spawn different types of *BioTask* objects for the various stages in the problem.

4.2.4 BioVisualizer Template

GLAD provides a visualization module in the form of the *BioVisualizer* class, to enhance the analysis of application results. It also allows application users to specify problem-specific and execution parameter values prior to execution through a customizable graphical user interface. The *BioVisualizer* class is inherited from the *ResultsCollector* class in the ALiCE programming template. Besides handling the collection of *BioResult* objects in the course of the application's execution, the *BioVisualizer* also coordinates the formatting of the visualization frames and windows, and controls the manner in which the results will be presented to the application user.

4.3 Control Classes

The execution of GLAD applications is driven by a number of classes that provide the underlying mechanisms supporting remote access to biological databases, interpretation of biological data, as well as monitoring of application performance. Unlike the GLAD developmental classes, these *control classes* are not inherited from the ALiCE core or extensions layers, and provide the modularity that separates the various functional components in the GLAD execution kernel from the PDT. The detailed implementation and capabilities of the control classes are transparent to application developers.

4.3.1 Handling Biological Data

Biological data essentially comprises of structured sequences that may be hosted on the local machine or sited on remote databases that are geographically distant from the ALiCE machine executing the GLAD application. The administration of biological data is performed by the Data Parser and Biological Data Access (BDA) engine components in the GLAD architecture, which involve the following classes:

- **DataParser** provides an interface between the application and a sequence file, comprising of methods to extract the sequence descriptors, actual sequence bodies and information regarding the biological classification of the sequences, such as whether a particular set of sequences are chromosomes or amino acids.
- **SequencesClient** represents the client end of the BDA engine, running on a compute producer. Whenever a set of sequences hosted on a remote database are required for computation, it establishes a TCP/IP connection with its counterpart process, the *SequencesServer*, running on the remote machine. The relevant sequences are then downloaded across the network and passed to the *DataParser* object on the local producer for translation.
- **SequencesServer** runs on each remote machine hosting a portion of the sequence dataset required by the GLAD application. It accepts a socket connection request from a *SequencesClient* on an ALiCE producer, processes the request by loading the relevant sequences from the disk of the machine on which it is running, and returns the fetched sequences to the *SequencesClient* across the network.

Typical protein databases are in the range of gigabytes. The BDA engine classes provide efficient transportation of biological data across the network by downloading only the required sequences. Figure 7 illustrates the collaboration between the abovementioned classes in handling biological data.

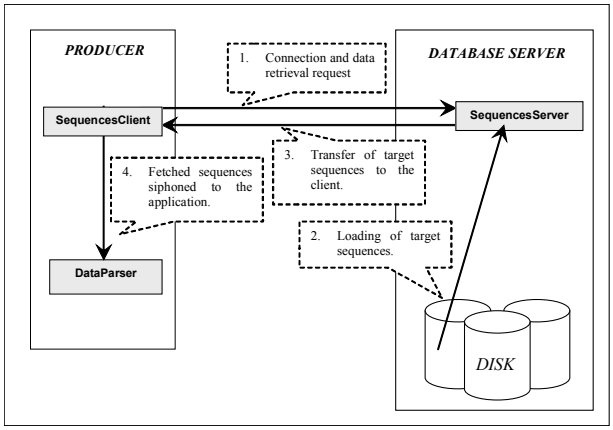

Figure 7: Biological Data Access Model

4.3.2 Statistical and performing monitoring

It may be important to determine certain meta-features of the biological dataset used by the application prior to execution proper. The *Statistics* class enables the developer to retrieve information such as the cardinality of the dataset, the mean length of all the sequences in the dataset, the corresponding standard deviation of sequence length, and the number of updates to a particular database since it was last accessed by the same GLAD runtime environment. This information not only helps to determine the total number of tasks that would be generated for a given execution, but also provides the user with an overview of the features of the dataset to be used.

The *Statistics* class also monitors the performance of the GLAD application execution by keeping track of the turnaround time at the *BioVisualizer*, the average task processing time for each *BioStage* in the grid environment, and the actual amount of time taken for each execution of the application.

4.4 Implementation Structure

The structure of the Java classes that constitute the GLAD application toolkit is presented in Figure 8. The implementation of the GLAD library modularizes the various functional components of a bioinformatics application. The developmental classes, which provide the workbench for the application developer work upon, are cleanly separated from the mechanisms for task decomposition, handling of biological data, and distributed object communications.

The library also includes a special class, *LibraryLoader*, which provides an interface from which the developer can invoke the various algorithmic components stored in the GLAD tools library for the application's use. For instance, a developer faced with a problem that involves the Huffman Encoding (HME) algorithm in one of its stages may invoke methods in the HME tool directly, without worrying about implementing the complete algorithm as one of its stages. This simplifies coding and reduces programming time, as the developer is able to manipulate commonly used small- and medium-scaled bioinformatics algorithms at an abstract level. Newly implemented GLAD applications can also be added to this set of bioinformatics tools by placing the class codes in the relevant sub-directories under the tools directory in the GLAD library, and replicating the codes across all systems in the GLAD runtime environment.

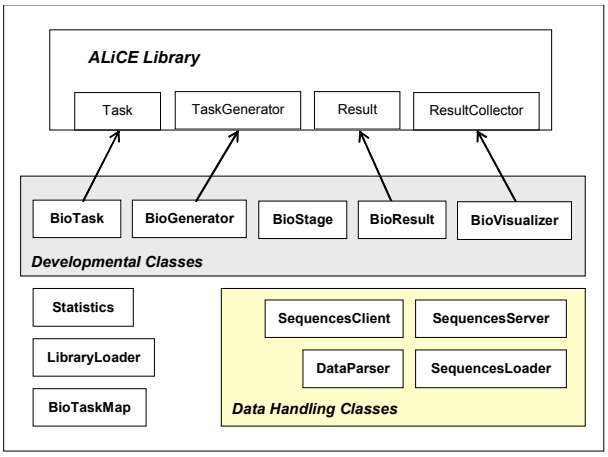

Figure 8: Class Structure of GLAD

5 Benchmark Applications

The GLAD library has been used to develop and deploy two different applications on grid systems. The first is the distributed sequence comparison, which has a relatively straightforward algorithm, a regular computational pattern, and is both compute- and data-intensive. The second is the distributed progressive multiple sequence alignment (PMSA), which comprises of multiple algorithmic stages, involves a semi-regular computational pattern, involves lots of heavy computations but does not typically involve large datasets. The different level of task dependencies between the two problems allows us to study the different approaches taken to parallelize and deploy each of them for grid execution.

For illustration, we introduce how to map the distributed sequence comparison problem onto the GLAD developmental framework. The map of the distributed progressive multiple sequence alignment onto the GLAD is available in (ALiCE web page, 2000).

Sequence comparison is one of the most important primitive operations in computational biology, serving as a basis for many other more sophisticated manipulations. In laymen terms, it involves discovering the similarity of parts of two sequences. The end result would be the provision of an optimal alignment for the pair of protein or DNA sequences. Two main concepts are involved in this problem: the similarity and alignment of the two sequences. The similarity of two sequences is a metric that dictates how syntactically matching they are. The alignment of two sequences is a way of placing one sequence above the other in order to clarify the correspondence between residues and portions of the sequences. Gaps can be inserted in arbitrary locations along the given sequences so that they end up with the same length, thus, enabling them to be comparable with each other.

Sequence comparison has a regular computation pattern, and the entire problem can be implemented in one biological stage. Figure 9 illustrates the implementation of the application using the GLAD library.

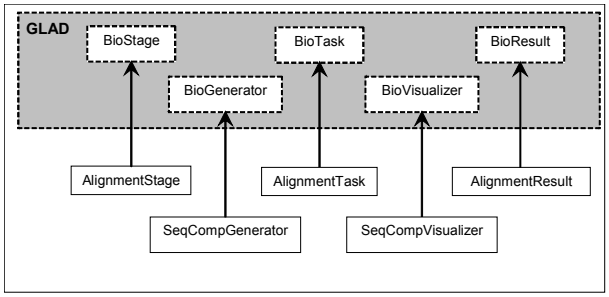

Figure 9: Implementation of Distributed Sequence Comparison

The SeqCompVisualizer provides the entry and exit points of the application, handling the parametric inputs for similarity computation, spawning of the AlignmentStage process, and the visualization of alignment results. AlignmentStage coordinates the flow of the similarity computation and sequence alignments across the queries and dataset, by generating the task partitioning schedule for the SeqCompGenerator. The SeqCompGenerator, in turn, creates the AlignmentTask objects for dissemination to the producer machines. Each AlignmentTask is responsible for the alignment of a query with a specific number of database sequences.

For example, if we assume a task size of 2,000 sequences, then each AlignmentTask is responsible for determining the top scoring sequences amongst an exclusive partition of 2,000 sequences in the database, and returning the score and relevant sequences in the form of an AlignmentResult object. The SeqCompVisualizer updates the highest score from the entire dataset as the AlignmentResult objects are retrieved, and eventually displays, for each query, the

highest scoring sequences in the dataset and the optimal alignments thus derived (Figure 10). The flow diagram in Figure 11 illustrates this procedure for a task size of 2,000 comparisons.

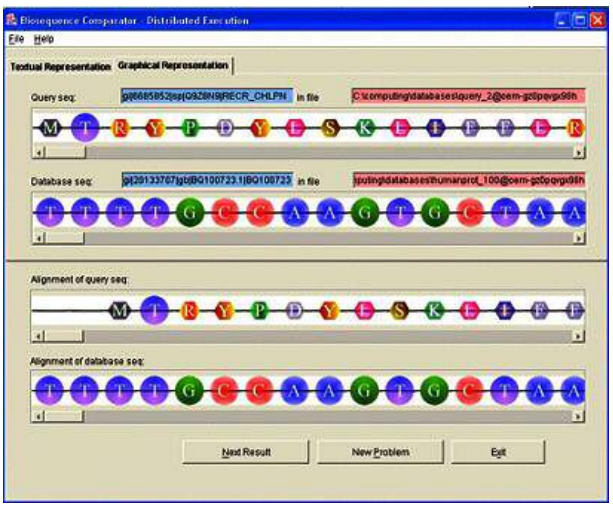

Figure 10: Optimal Alignment of Two Sequences

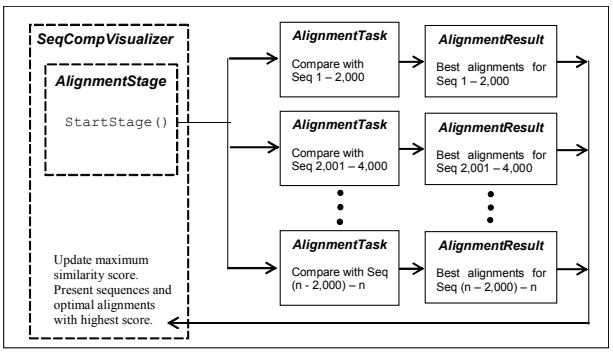

Figure 11: Distributed Sequence Comparison Flow

6 Concluding Remarks

We have developed a bioinformatics application toolkit, GLAD, which is implemented using the ALiCE paradigm. GLAD enables the researcher to work with a set of primitives and constructs, without specific technical knowledge of the means in which parallelization and biological data processing are being handled by the system. GLAD is scalable and supports the development and deployment of applications involving regular and semi-regular parallelization patterns and operating over huge biological datasets distributed over several databases on the network.

GLAD provides a grid-based bioinformatics applications development workbench that is designed for the deployment of medium- to large-scale applications to facilitate extensive research in the life sciences. The main objective of GLAD is to support the implementation of parallel bioinformatics applications on a variety of grid environments without being handicapped by the lack of human expertise in the construction of computational grids and the hassle of manually coding the treatment of biological datasets involved in the individual executions.

We have also demonstrated the use of GLAD in the development of two grid-based bioinformatics applications: distributed sequence comparison and distributed PMSA, and deployed them on homogeneous and heterogeneous cluster grids separately for our performance scalability experiments.

If you can point to anything that could be improved in GLAD, have suggestions, need benchmark applications and the GLAD toolkit, please contact us at the address above.

Acknowledgement

The authors would like to thank Dr Li Kuo-Bin from the Bioinformatics Institute of Singapore for the many useful discussions and helpful suggestions.

References

ALiCE (2000) Grid Computing Project, http://www.comp.nus.edu.sg/~teoym/alice.htm.

Altschul, S., Gish, W., Miller, W., Myers, E. and Lipman, D. (1990) Basic Local Alignment Search Tool, *Journal of Molecular Biology,* 215, 403-410.

Bosson, O. and Riml, N. (2003) Overview in Bioinformatics 2003, Bioinformatics in Sweden, http://artedi.ebc.uu.se/course/overview/swedish_industry.html.

Burge, C. and Karlin, S. (1997) Prediction of Complete Gene Structures in Human Genomic DNA, *Journal of Molecular Biology,* 268, 78-94.

Etzold, T. and Argos, P. (1993) SRS: An Indexing and Retrieval Tool for Flat File Data Libraries, *Computer Applications in the Biosciences*, 9, 49-57.

Fitch, W. (1977) On The Problem of Discovering The Most Parsimonious Tree, *Proceedings of the National Academy of Sciences*, 111, 223-257.

Fitch, W. and Margoliash, E. (1967) Construction of Phylogenetic Trees, *Science*, 155, 279-284.

Foster, I. and Kesselman, C. (1997) Globus: A Metacomputing Infrastructure Toolkit, *International Journal of Supercomputer Applications*, 11(2), 115-128.

Fratini, A., Kopka, M., Drew, H. and Dickerson, R. (1982) Reversible Bending and Helix Geometry in a B-DNA Dodecamer: CGCTAATTCGCG, *Journal of Biological Chemistry,* 24, 14686-14707.

Genome Institute of Singapore Home Page, http://www.gis.nus.edu.sg/.

GigaSpaces (2002) Platform White Paper GigaSpaces Technologies, Ltd..

Higgins, D. and Sharp, P. (1988) CLUSTAL: A Package for Performing Multiple Sequence Alignment on a Microcomputer, *Gene*, 73, 237-244.

Hupfer, S. (2000) The Nuts and Bolts of Compiling and Running JavaSpaces Programs, *Java Developer Connection*, Sun Microsystems, Inc.

Jini (2001) Network Technology - An Executive Overview, White Paper, Sun Microsystems Inc., California, USA.

Laskowski, R., MacArthur, M., Moss, D. and Thornton, J. (1993) PROCHECK: A Program to Check the Stereochemical Quality of Protein Structures, *Journal of Applied Crystallography*, 26, 283-291.

Lavery, R. and Sklenar, H. (1988) The Definition of Generalized Helicoidal Parameters and of Axis Curvature for Irregular Nucleic Acids, *Journal of Biomolecular Structure and Dynamics,* 6, 63-91.

Monge, A. and Elkan, C. (1996) The Field-matching Problem: Algorithm and Applications, *Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining*, 267-270.

National Center for Biotechnology Information (NCBI) Homepage, National Library of Medicine, Maryland, United States of America, http://ncbi.nlm.nih.gov.

Pevzner, P., Tang, H. and Waterman, M. (2001) A New Approach to Fragment Assembly in DNA Sequencing, *Proceedings of the 5th Annual International Conference on Computational Biology*, New York, USA, 256-267.

Rost, B. (1998) Protein Structure Prediction in 1D, 2D, and 3D, *The Encyclopaedia of Computational Chemistry*, 3, 2242-2255.

Siepel, A., Farmer, A., Tolopko, A., Zhuang, M., Mendes, P., Beavis, W. and Sobral, B. (2001) ISYS: A Decentralized, Component-based Approach to the Integration of Heterogeneous Bioinformatics Resources, *Bioinformatics*, 17(1), 83-94.

Terwilliger, J. and Ott, J. (1991) Analysis of Human Genetic Linkage, 2nd Edition, *Johns Hopkins University Press*, Baltimore, USA.

Trelles, O. (2001) On the Parallelization of Bioinformatics Applications, *Briefing in Bioinformatics*, Henry Stewart Publications, 2(2), 181-194.