

**(Genome Pipeline Assembler)**

Software Design Document

Names: Taylor Flatt, Marty Grawe, Ryan Sample

Lab Section: CS 498/499

Date Updated: (05/04/2016)

Table of Contents

[**1. INTRODUCTION** 4](#_Toc450218051)

[**1.1 Purpose** 4](#_Toc450218052)

[**1.2 Scope** 4](#_Toc450218053)

[**1.3 Overview** 4](#_Toc450218054)

[**1.4 Reference Material** 5](#_Toc450218055)

[**1.5 Definitions and Acronyms** 5](#_Toc450218056)

[**2. SYSTEM OVERVIEW** 5](#_Toc450218057)

[**3. SYSTEM ARCHITECTURE** 6](#_Toc450218058)

[**3.1 Architectural Design** 6](#_Toc450218059)

[**3.2 Decomposition Description** 7](#_Toc450218060)

[**3.2.1 Job Entity** 7](#_Toc450218061)

[**3.2.2 Account Entity** 8](#_Toc450218062)

[**3.2.3 SGA Entity** 8](#_Toc450218063)

[**3.2.4 MaSuRCA Entity** 8](#_Toc450218064)

[**3.2.5 WGS Entity** 8](#_Toc450218065)

[**3.3 Assemblers** 8](#_Toc450218066)

[**3.3.1 SGA** 8](#_Toc450218067)

[**3.3.2 Masurca** 10](#_Toc450218068)

[**3.3.3 WGS** 11](#_Toc450218069)

[**3.4 Design Rationale** 12](#_Toc450218070)

[**4. COMPONENT DESIGN** 13](#_Toc450218071)

[**4.1 Job Detail** 13](#_Toc450218072)

[**4.2 Account Detail** 15](#_Toc450218073)

[**4.3 Web Server Detail** 15](#_Toc450218074)

[**4.4 BigDog Detail** 16](#_Toc450218075)

[**5. HUMAN INTERFACE DESIGN** 16](#_Toc450218076)

[**5.1 Login Screen** 16](#_Toc450218077)

[**5.2 Register Screen** 16](#_Toc450218078)

[**5.3 Home Screen** 17](#_Toc450218079)

[**5.3 About Screen** 18](#_Toc450218080)

[**5.4 FAQ Screen** 18](#_Toc450218081)

[**5.5 Create a Job Wizard** 19](#_Toc450218082)

[**5.5.1 Step 1** 19](#_Toc450218083)

[**5.5.2 Step 2** 20](#_Toc450218084)

[**5.5.3 Step 3** 21](#_Toc450218085)

[**5.5.4 Step 4** 21](#_Toc450218086)

[**5.5.5 Step 5** 22](#_Toc450218087)

[**5.6 View Job Details Screen** 23](#_Toc450218088)

[**5.7 View Jobs Screen** 24](#_Toc450218089)

[**5.8 Manage Profile Screen** 25](#_Toc450218090)

[**5.9 Verify Account Screen** 25](#_Toc450218091)

[**5.10 Manage Users Screen** 26](#_Toc450218092)

[**6. DETAILED DESIGN** 26](#_Toc450218093)

[**6.1 Registration and Login** 26](#_Toc450218094)

[**6.2 Verify Account** 27](#_Toc450218095)

[**6.3 Create Job** 28](#_Toc450218096)

[**6.4 Update Job** 30](#_Toc450218097)

[**6.5 Manage Users** 33](#_Toc450218098)

[**6.6 Analysis** 34](#_Toc450218099)

[**7. RESULTS** 35](#_Toc450218100)

# **1. INTRODUCTION**

## **1.1 Purpose**

This software design document describes the architecture and system design of multiple genome assemblers running identical data through these multiple assemblers to increase likelihood of accurate results.

## **1.2 Scope**

The ultimate goal of this project is to create a parallelized pipeline of tasks that data can be subjected to in order to assemble genome data on a distributed system. The main objective is to produce the most accurate results possible with this infrastructure.

There are several major objectives to accomplish. The first objective being the overall infrastructure itself with the sequencers and pipeline for the genome data. This would include the conversion steps, if necessary, to account for different input data types and output data types.

The second objective is to provide an efficiency (accuracy) analysis of the data using CAP3. The focus here wouldn’t necessarily be the implementation of the algorithm but making sure the comparison algorithm is as cheap and effective as possible.

The third objective is to create a web interface for easy interaction with the system. This would include an authentication system coupled with a page for all the values to setup the assembly process. Finally, the web interface would have to include a way to view the data in a manageable way. That includes being able to halt a current running job, view the current status of a job, view previous job results, as well as a location to store the data for user download.

A final objective if time and resources are permitting is to do any refactoring of the assemblers themselves. This goal is unlikely to be realized but if there is time and a way to increase the efficiency of the product, and then we will address the issue.

## **1.3 Overview**

This document will focus primarily on the design and implementation of the structures for the genome assembly itself. A focus will be placed on the process over the specific pseudocode.

It is also important to note that this design document may change in the process of implementing this project. However, the goals will remain the same but the implementation of those goals is subject to change.

## **1.4 Reference Material**

Here we will list the locations of all the assemblers that we use as well as any FAQ/documentation that is necessary in the use of those products. In addition, we will also include our own documentation on how to effectively use each product.

Currently, the only implemented software on our cluster is MaSuRCA. This is likely to change moving forward.

## **1.5 Definitions and Acronyms**

* OSS: Open Source Software.
* VPN: Virtual Private Network.
* RAID: Redundant array of inexpensive disks.
* MVC: Model/View/Controller
* FTP: File transfer protocol. A location to store file that can be publically accessed.
* Genome Assembly: The process of taking a large number of small DNA sequences and putting them back together to create a representation of the original chromosomes from which the DNA originated.
* Here user and researcher are used interchangeably.

# **2. SYSTEM OVERVIEW**

The primary problem we are aiming to address is the lack of sufficient documentation of sequence assemblers and a way to effectively run data through several assemblers in a distributed environment. In particular, the sequencers will live on the BigDog cluster at Southern Illinois University Carbondale.

The back-end will contain at minimum three sequence assemblers. A primary focus of our accuracy goal is to increase the sample size of the data from or as large as we can get it. However, currently there is only one implemented. There will be several scripts that can be run to convert data from a certain data type to an acceptable data type. These are usually implemented in python.

The front-end will contain a website which uses its own authentication. From there, the user has the ability to create a new job, halt a current job, view their job status, and see job statistics which may include previous data outputs.

The idea is to provide an easy to use interface that includes several assemblers that a particular data set can be run through. It is also imperative that this system provide a means in converting input data into acceptable formats for the assemblers to use.

# **3. SYSTEM ARCHITECTURE**

## **3.1 Architectural Design**

First, the researcher (otherwise known throughout as “user”) will access this program through a web interface. This web interface is programmed in ASP.NET MVC 5 with C# and a MySQL backend. In this case, each assembly “job” is our model. So each job has a set of parameters which range from who created the job, to data parameters such as the location, to specific assembler details.



**Figure 3.1**

Job creation is handled by a simple wizard. The data entered is stored in the session and validated on the DOM and errors are returned to the user dynamically. As such, all validation is done at this step and although it is validated at the model level too, it will always pass because of our JavaScript validation.

So once the wizard is filled out on the web interface, scripts are dynamically created on the web server and stored on the local FTP. The first of these scripts will contain the location of their data as well as the specific scripts that represent the assemblers that they chose. So if they chose two assemblers, three scripts would be created two of which are assembler scripts that initiate the assemblers themselves and an init script that initializes the data download and assemblers to run.

After the scripts have been created and stored, an SSH connection is created to BigDog with the user’s credentials. At this point, multiple directories are created under the user’s account with the following structure relative to BigDog:

C:\School\CS 498 (Senior Project I)\Documentation\FileStructure.png

**Figure 3.2**

It is now obvious where everything will be stored prior to and during the runtime of a job. The scripts downloaded from the web server will be stored in the config folder. The data will be downloaded to the data folder at runtime. The logs of each assembler’s run will be stored in the log folder for future consult. The output of each assembler is stored under its own folder in the output directory.

Once the run is completed, the entire directory structure is compressed into a zip and send back to the web server FTP where it will be made available for download by the researcher.

Once the web server is aware the data has been completed, it will also notify the researcher of the job’s completion whether it is an error or success.

## **3.2 Decomposition Description**

### **3.2.1 Job Entity**

This is a complete list of parameters that are inherent to each job to successfully run. A detailed description will be given in Section 4.1.

### **3.2.2 Account Entity**

This entity is just a user for our platform. A detailed description will be given in Section 4.2

### **3.2.3 SGA Entity**

This entity includes all of the information for the SGA assembler and how it will runs. A detailed description will be given in Section 3.3.1.

### **3.2.4 MaSuRCA Entity**

This entity includes all of the information for the MaSuRCA assembler and how it will runs. A detailed description will be given in Section 3.3.2.

### **3.2.5 WGS Entity**

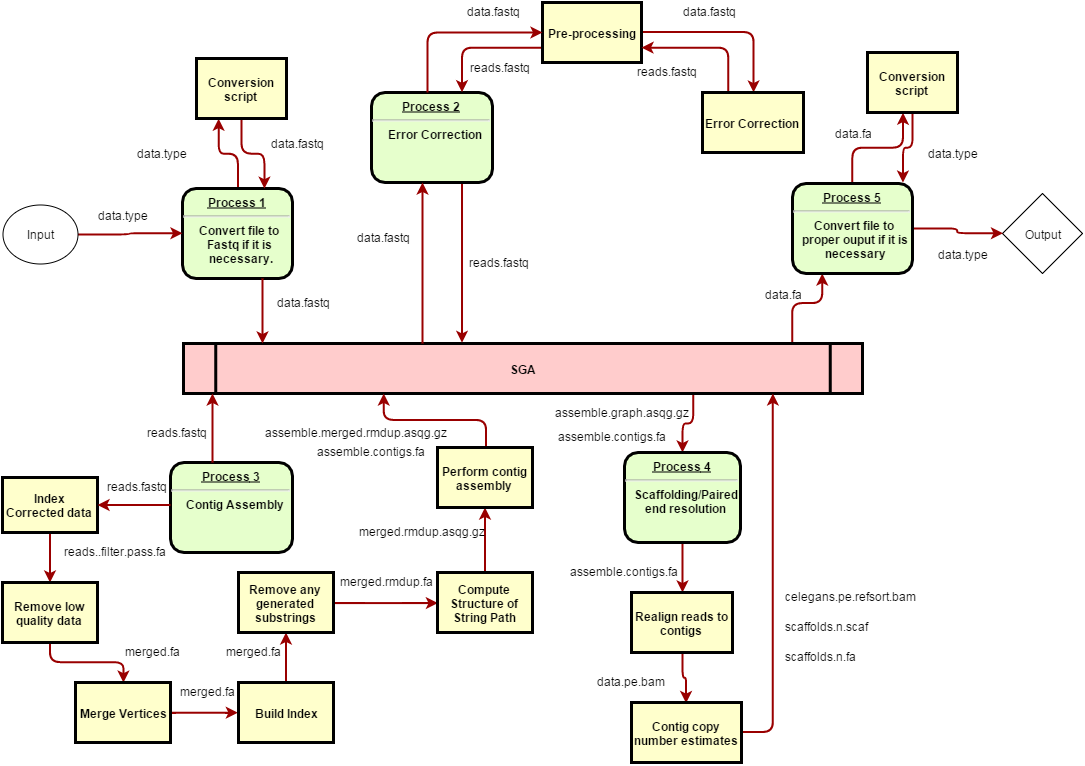
This entity includes all of the information for the WGS assembler and how it will runs. A detailed description will be given in Section 3.3.3.

## **3.3 Assemblers**

### **3.3.1 SGA**

SGA is a de novo assembler for DNA sequence reads. It is based on Gene Myer’s string graph formulation of assembly and uses the FM-index/Burrows-Wheeler transform to efficiently find overlaps between sequence reads. SGA will accept a fastq as input and then requires the following parameters to perform error correction, contig assembly, and scaffolding end resolution.

* Parameters:
  + Input files.
  + SGA program version specific.
  + Overlap parameter.
  + Number of CPU threads. (static, based on our number)
  + Merge Reads.
  + K-mer value.
  + Minimum K-mer coverage.
  + Overlap parameter for FM-Merge.
  + Small repeat resolution algorithm.
  + Minimum number of pairs to link two contigs.
  + Minimum length of contigs.
  + Scaffold tolerance.
  + Bubble collapse toggle.
* Error Correction:
  + Build index for error correction using ropebwt.
  + Perform error correction with specific k-mer value on a number of CPUs.
* Contig Assembly:
  + Index the corrected data with ropebwt.
  + Remove exact match duplicates and reads with low-frequency k-mers.
  + Merge simple, unbranched chains of vertices.
  + Build an index of the merged sequences.
  + Remove any substrings that were generated from the merge process.
  + Compute the structure of the string graph.
  + Perform the contig assembly without bubble popping.
* Scaffolding end resolution:
  + Realign reads to the contigs.
  + Make contig-contig distance estimates.
  + Make contig copy number estimates.



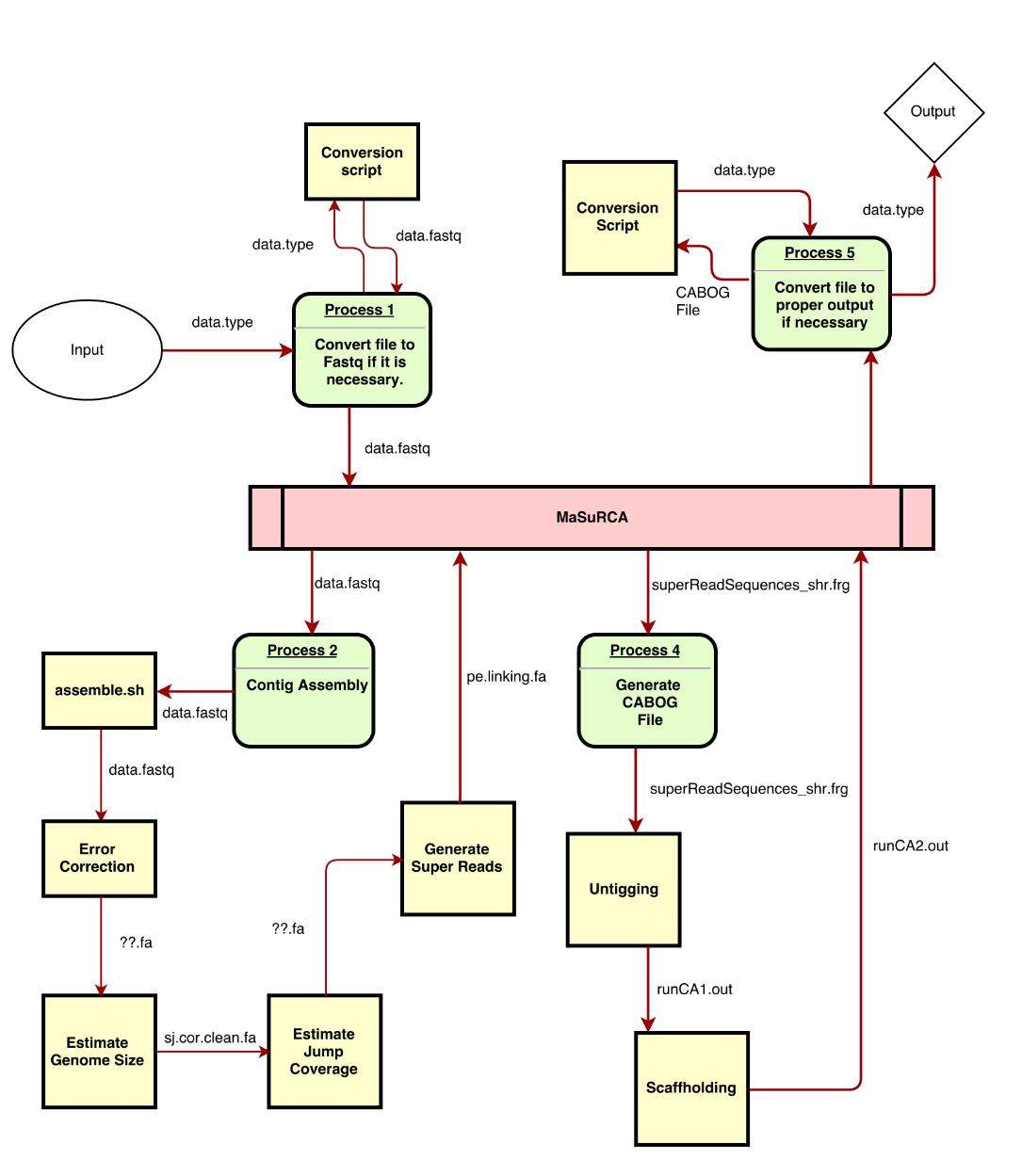
**Figure 3.3**

### **3.3.2 Masurca**

MaSuRCA is a whole genome assembly package. It uses the de Brujin graph and Overlap-Layout-Consensus (OLC) to increase efficiency. As a requirement, this assembler will only accept short reads from Illumina sequencing or a mixture of short reads and long reads.

* Parameters:
  + Number of threads to run the assembler on.
  + Jellyfish hash size (usually 10x the genome size).
  + Use linking mates (binary)
  + Graph k-mer size.
  + Limit jump coverage.

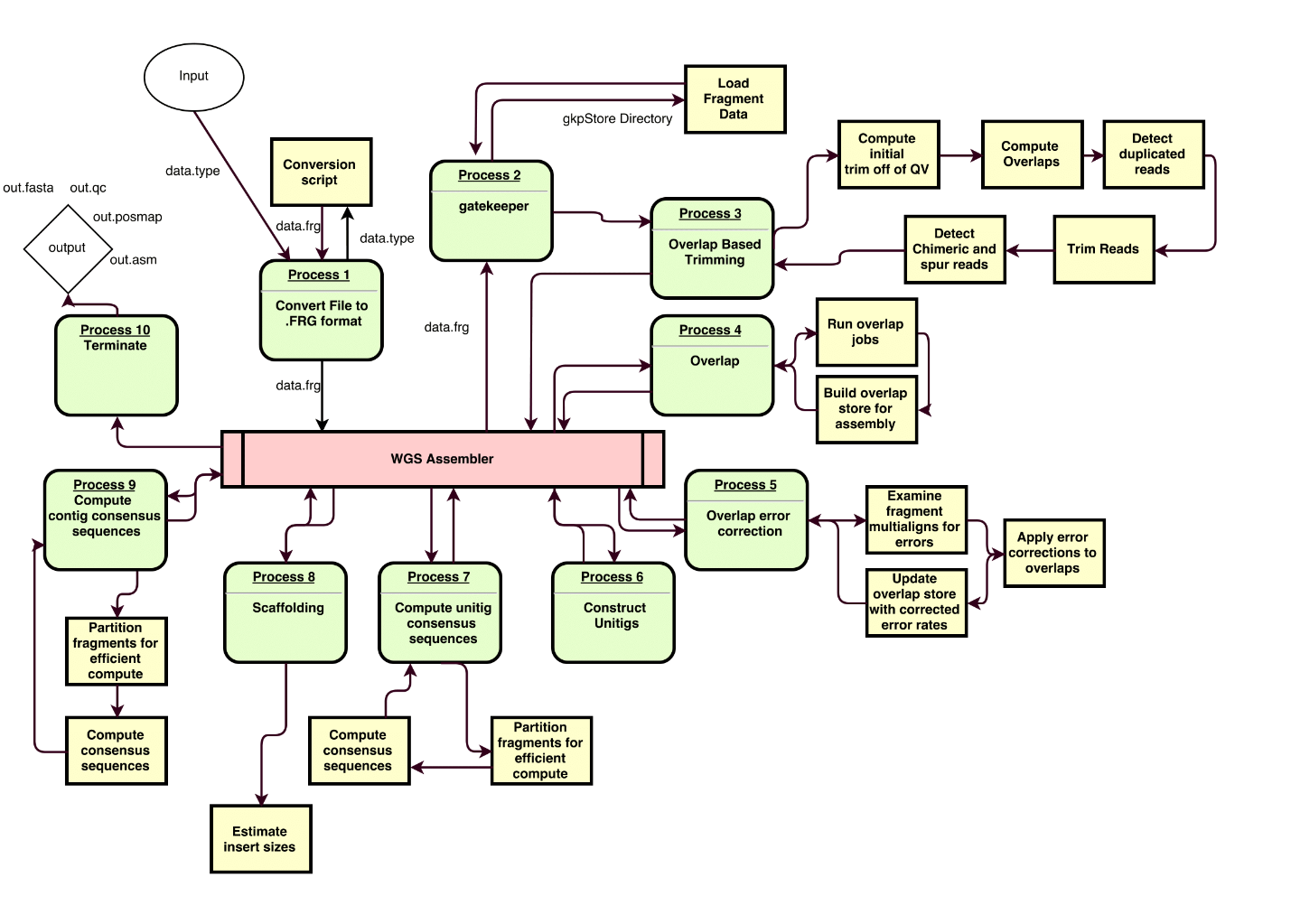
Soap assembly (set to true if contigging and scaffolding done by SOAPdenovo2 instead of CABOG). This will reduce the runtime but possibly less accuracy.



**Figure 3.4**

### **3.3.3 WGS**

Otherwise known as the Celera Assembler. It gets its name from being a de novo whole-genome shotgun (WGS) DNA sequence assembler. It reconstructs large sequences of genomic DNA from fragment data produced by whole-genome shotgun sequencing.



**Figure 3.5**

## **3.4 Design Rationale**

The web interface is an easy way to make an accessible platform for researchers who are not familiar with the Unix environment to easily run tasks through the assemblers. The reason we decided to restrict use to a university IP is that at the current time there is no reason for anyone to be requesting a job outside of the network. Not to mention, to access BigDog, the user needs to be on campus or on a VPN.

We decided on using the user account as a means to start the jobs since the BigDog administrative team would like accurate logs as to who has run what and when. Otherwise, we just run updates and upload their data via a special account.

The reason for generating almost everything on the web server was a way to create the most controlled environment possible. Instead of sending the parameters via command line, it is much easier to create files locally and then upload those created files. It improves code readability, the ability to extend the program to more assemblers, as well as ease of maintainability.

Instead of storing the researcher’s data back where we downloaded their input data, we decided on storing the information locally. This was done for a variety of reasons:

1. We may not have write permissions on their FTP.
2. We did not want to involve another account in the process further complicating things.
3. We wanted a controlled environment where the pathing is predictable.
4. We wanted local data redundancy in the form of a mirrored RAID.
5. We wanted the possibility of local removal of the research data.
6. It is easier.

Most of the reasons are fairly obvious. However, #3 warrants more detail. If all other reasons were accounted for, we still needed to predict the structure of the remote FTP which is nearly impossible to do consistently. We could just dump the data on the root directory where we login at but this isn’t in line with best practices. So, instead, we can setup our own FTP where the user has access to their data in a nice format where jobs are separated neatly. Furthermore, it allows us to dynamically create a download link to send them via email and display on the job detail page.

In addition, doing this allows for a more involved transfer process with the user being aware of the transfer. If something were to go wrong, there is a higher likelihood that the errors can be caught and dealt with.

Since we are using our own FTP, there is a finite amount of space and a potentially unlimited number of jobs. That means that something must give. So we will setup a pruning that will remove the oldest jobs from the system so that the drives don’t get too full. Wanting to use the majority of the FTP, prune would ideally only be initiated after the FTP hits a certain disk usage of say 90%. Then we would remove old jobs until the disk usage is down to say 85% or 80%.

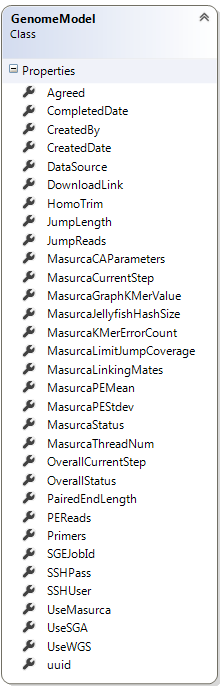
Alternatively, we can just setup a time-based prune so the removal of the job is predictable and there is enough warning in the form of emails indicating the removal date and reminders prior to the removal. This form of pruning will most likely be implemented.

# **4. COMPONENT DESIGN**

In this section, the particulars of each component will be discussed in greater detail.

### **4.1 Job Detail**

**Figure 4.1**

Each particular job that is created by the user will have the following properties:

* Agreed: Whether they agreed to the instructions displayed on step 1.
* Completed Date: When the job was completed successfully or unsuccessfully.
* Created By: Who the job was created by.
* Created Date: When the job was created successfully.
* Data Source: A concatenated string of URLs representing the paths to the researcher data.
* Download Link: A link pointing to the location where their compressed data is stored on the web server FTP.
* Homo Trim: Whether the homopolymers have been trimmed.
* Jump Lengths: The length of the jump reads.
* Jump Reads: Whether the researcher data has Jump Reads.
* Masurca CA Parameters: Whether the data source is a bacteria.
* Masurca Current Step: The current step of the analysis relative to the masurca assembler.
* Masurca Graph KMer Value: The graph graph K-Mer value.
* Masurca Jellyfish Hash Size: Typically 10x the size of the genome.
* Masurca KMer Error Count: The K-Mer count threshold.
* Masurca Limit Jump Coverage: Whether masurca will limit the jump coverage.
* Masurca Linking Mates: Whether the data is illumina data.
* Masurca PE Mean: The paired-end mean of the data.
* Masurca PE Stdev: The paired-end standard deviation.
* Masurca Status: The description of the current step.
* Overall Current Step: The current step (dynamic/will change based on number of assemblers chosen).
* Overall Status: The description of the current step.
* Paired-End Length: The length of the paired-end reads.
* Primers: Whether the primers have been removed or not.
* SGE Job Id: The job id the scheduler assigns to a queued or running job.
* SSH User/SSH Pass: Both are not stored values. They are never assigned.
* Use Masurca: Whether the job is using the masurca assembler.
* Use SGA: Whether the job is using the SGA assembler.
* Use WGS: Whether the job is using the WGS assembler.
* UUID: The key in the database that is unique to each job.

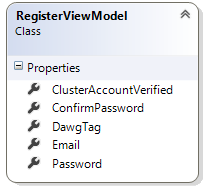
It should be noted that the job model above only includes the parameters associated with the MaSuRCA assembler. So as the platform expands to accept more assemblers, the job model too will expand to include the parameters specific to the new assemblers as well.

These values in the model are all used in some way to either display information back to the user or construct the information surrounding their job.

The specifics of how the job is created and modified will be shown in greater detail in Section 5.1.

### **4.2 Account Detail**

The account is typical of an ASP.NET MVC 5 account model. Although, we have added two additional fields. The generalized account parameters are:



**Figure 4.2**

* Cluster Account Verified: This will be true if they have successfully checked their quota/permissions on BigDog. This is not a recurring check and is only assigned once.
* DawgTag: Represents the user’s SIUC DawgTag.
* Email: Synonymous with the username of the researcher.
* Password: The hashed password of the user.

Every user will have these properties.

The first step of the process starts with the web interface and requiring the user to be on a campus IP. Provided the user is on the campus network, they will create an account on the website using a valid email and supplying their dawgtag as well. It is important to note that unless someone is registered and verified, they will be unable to access any website features.

Once they have created an account, their account will be marked as “unverified”. There are three user roles in the system. The default is unverified and they don’t have any real access to the website. They cannot create a job or perform any tasks. Then there is verified and those users have full use of the system. Finally there is the admin. The only additional task they have access to is the ability to delete users and change user roles.

By default, each user is marked as “unverified” which means an admin will need to change their status to “verified” to receive full access. Ultimately, the goal is to somehow tie into the SIUC AD to run verification of user accounts past the BioInfo group that would check if their email + dawgtag is in that group and then email their account with a verification email. In the meantime however, an admin must approve accounts prior to their use.

### **4.3 Web Server Detail**

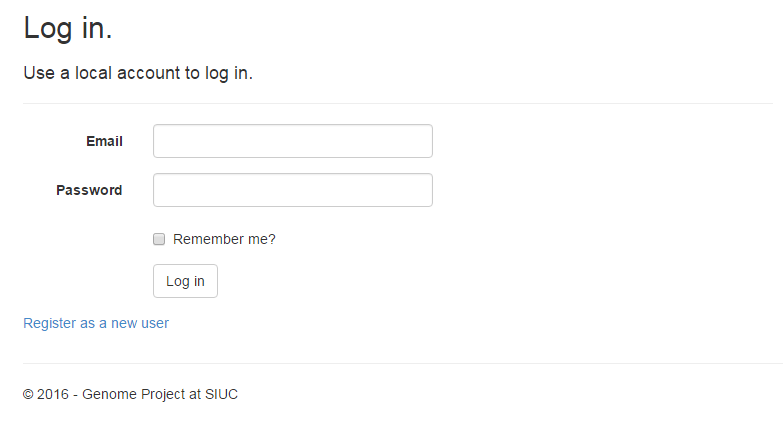
The web server is a Windows 8.1 machine with IIS installed running the web server with a MySQL backend. There is an FTP where the scripts will be stored as well as the final data.

### **4.4 BigDog Detail**

The cluster on campus at SIUC has huge computational power but for this project, the nodes used are compute node 24 and compute node 25 which individually have 750GB of RAM. In total, BigDog has 4TB of RAM, 800 CPU cores, 46TB of space, and 40 nodes.

# **5. HUMAN INTERFACE DESIGN**

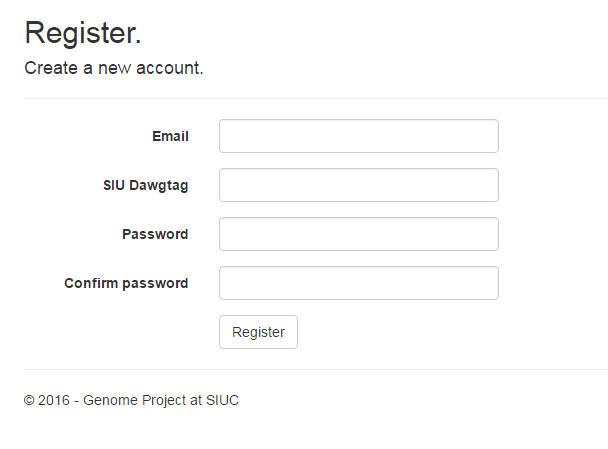
## **5.1 Login Screen**



**Figure 5.1**

The login page requires an email and password to login as well as a redirection to the register page.

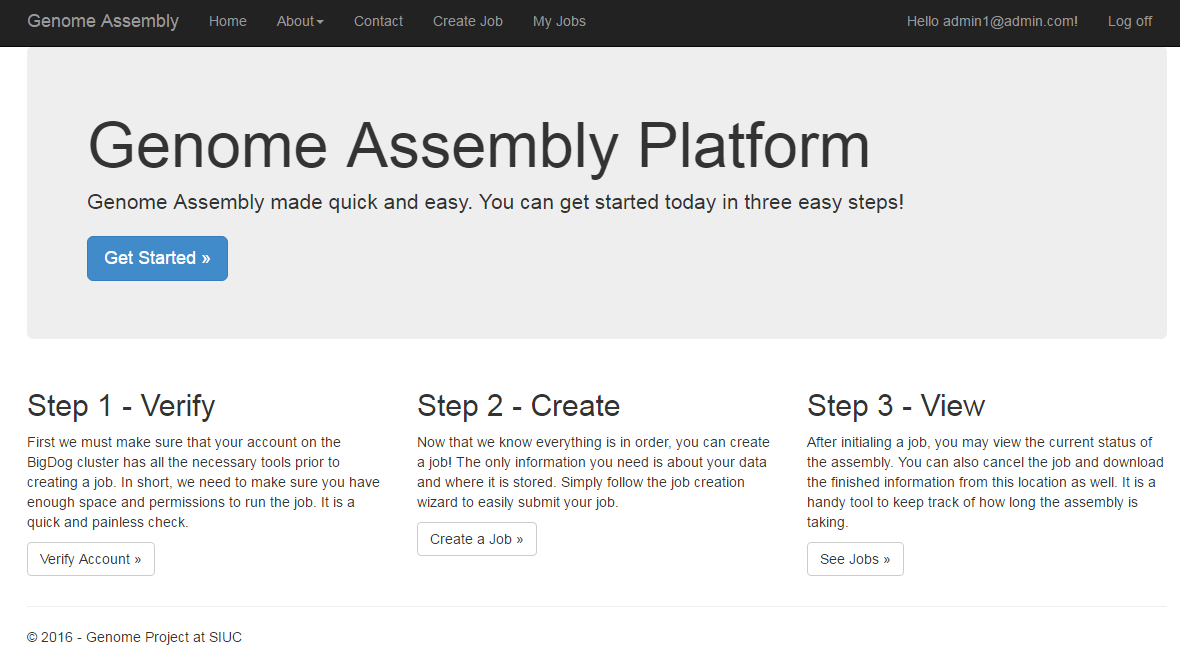
## **5.2 Register Screen**



**Figure 5.2**

This screen allows a user to create an “unverified” (more on that in Section 6) account consisting of an email, their dawgtag (only numbers), and a password. Once the register button is hit, the account is verified for any errors and if successful redirects to the homepage.

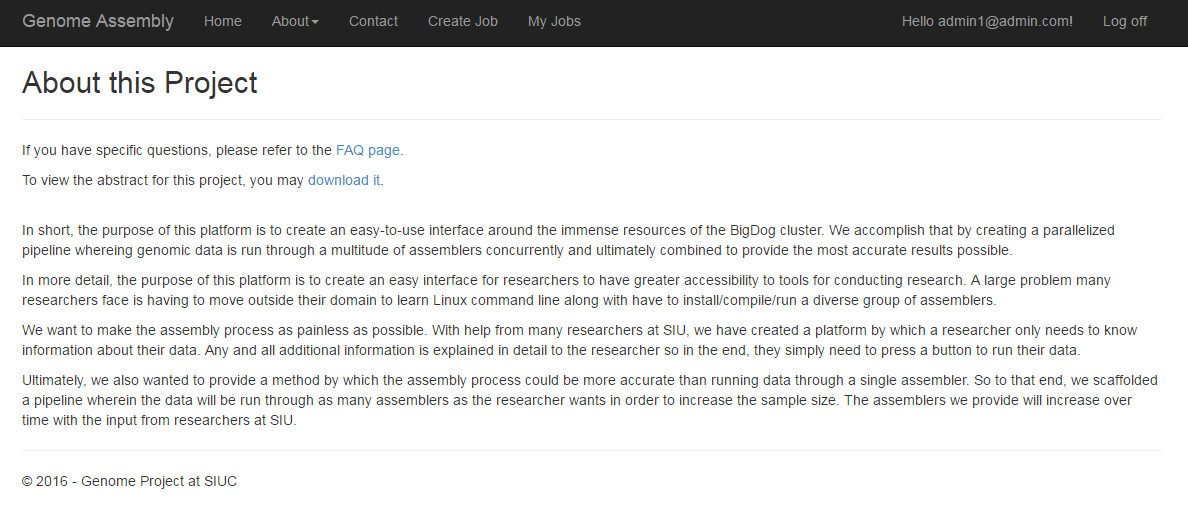
## **5.3 Home Screen**



**Figure 5.3**

This screen is the hub for all action on the platform. It is the starting point for new users as well as provides helpful links on how to get started on creating your first job.

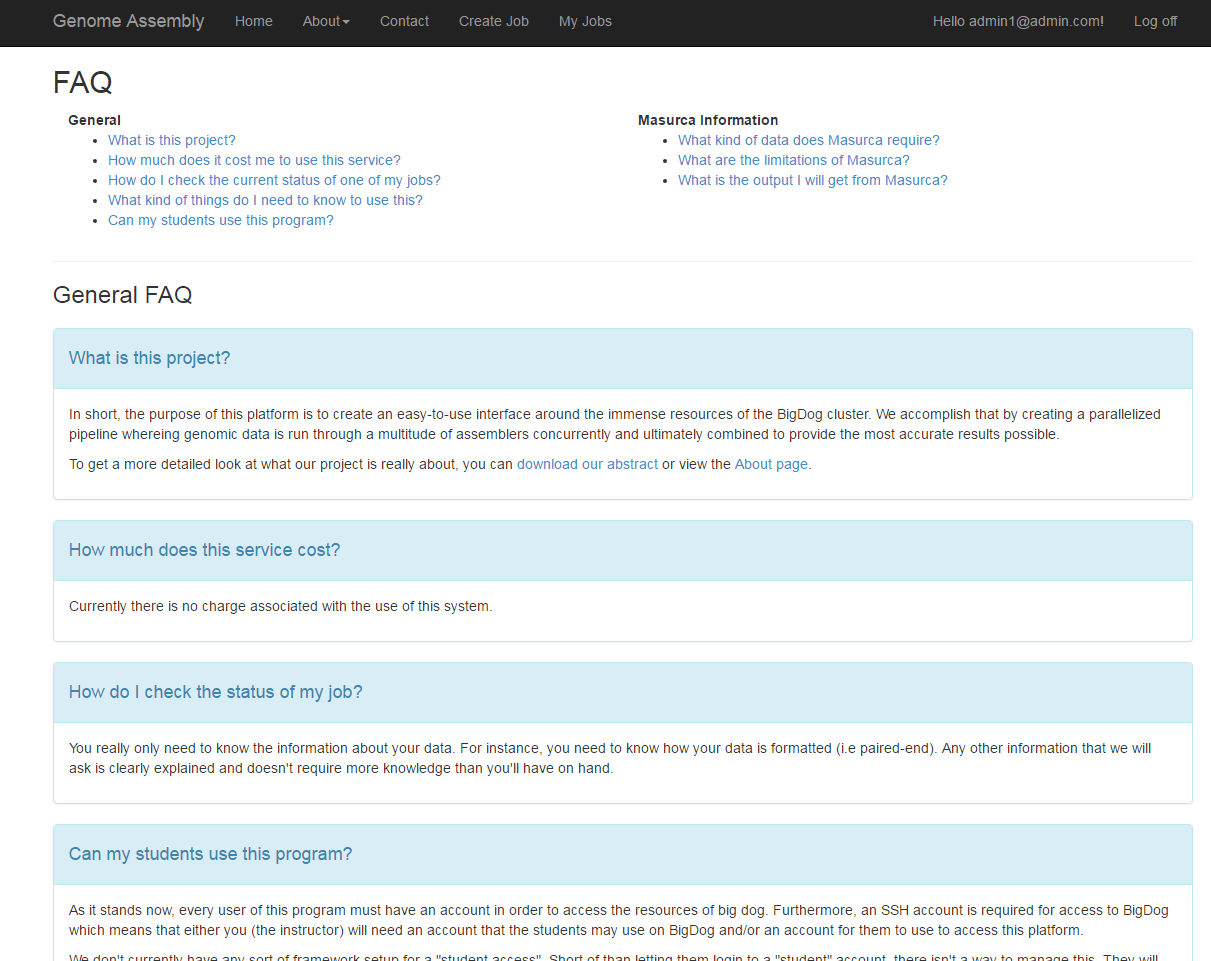
## **5.3 About Screen**



**Figure 5.4**

This page simply outlines the objectives of what this platform can and cannot do. Mainly it provides an overview of its purpose.

## **5.4 FAQ Screen**



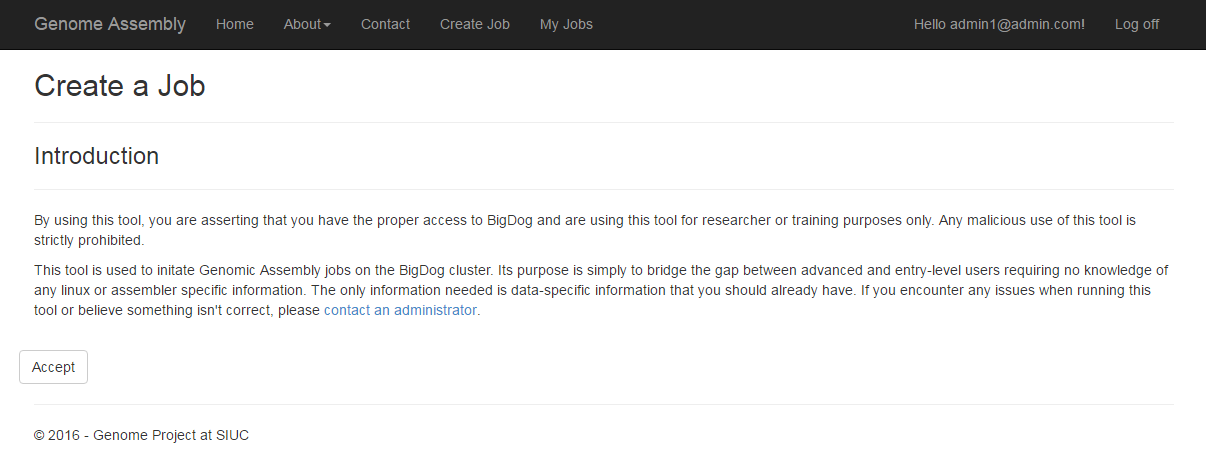
**Figure 5.5**

The rationale of an FAQ page is to provide a sort of targeted first line of defense for users if they have any issues using the system. While it isn’t currently populated, it would include things like assembler specific information and instruction as well as what to expect while a job is running vs when it has finished.

## **5.5 Create a Job Wizard**

The following sections outline what is to be expected when creating a new job.

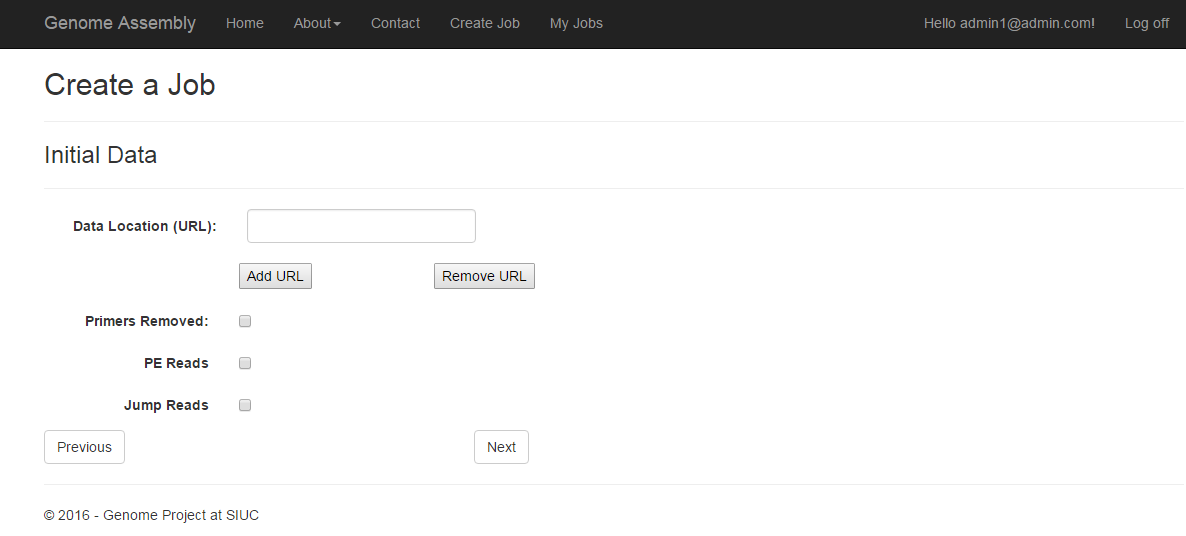
### **5.5.1 Step 1**



**Figure 5.6**

This first step is a simple outline of what this wizard is and maybe some legalese on the use of the system. It is pretty basic. Accepting these terms are required to move onto the next step of the wizard process.

### **5.5.2 Step 2**



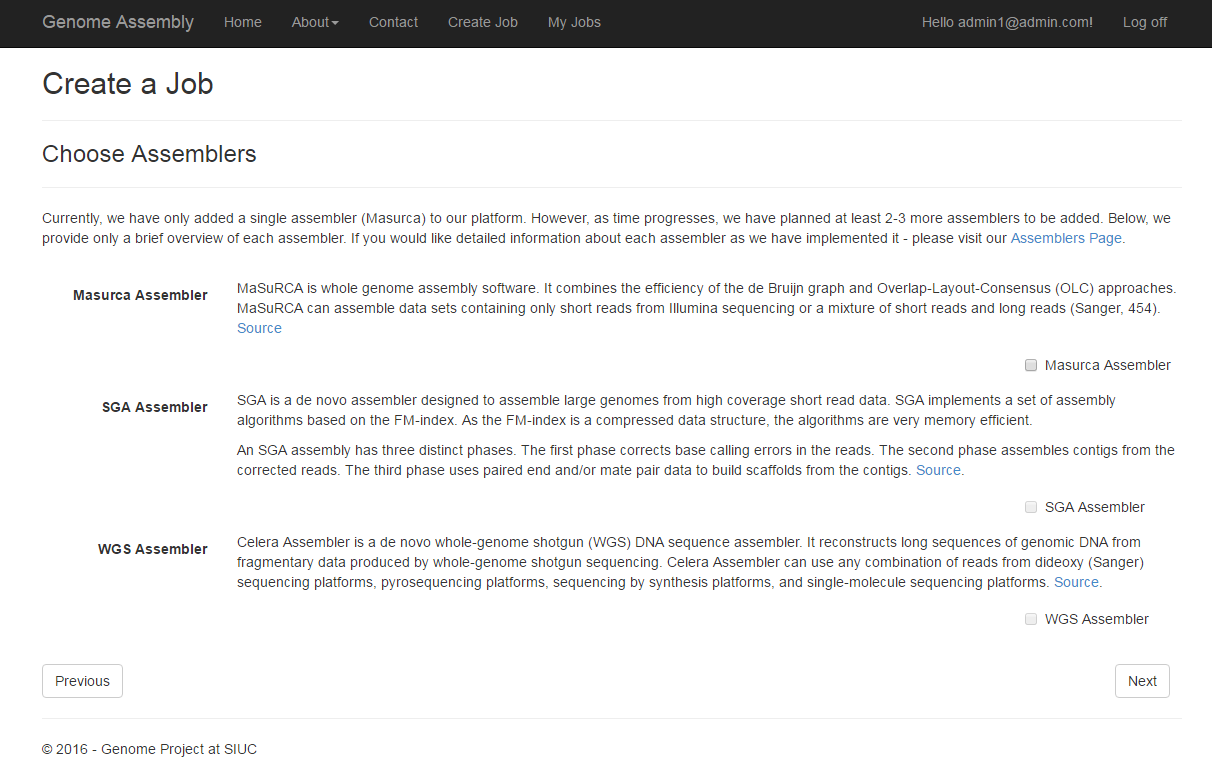
**Figure 5.7**

This screen allows the researcher to input their data specific information. Basically, this information will exist for all types of data that are possible to run through our system. It is interesting to note here, that the user is able to add more URLs which will create more textboxes. In addition, upon selecting either PE reads or Jump Reads, the other will disappear. This is because those options are mutually exclusive and the data cannot be both.

In other words, the steps throughout this process are all dynamic and will change depending on what kind of data that you choose.

If the user doesn’t enter in a data location, errors will be generated and the user will be unable to proceed to the next step until those errors have been rectified.

### **5.5.3 Step 3**



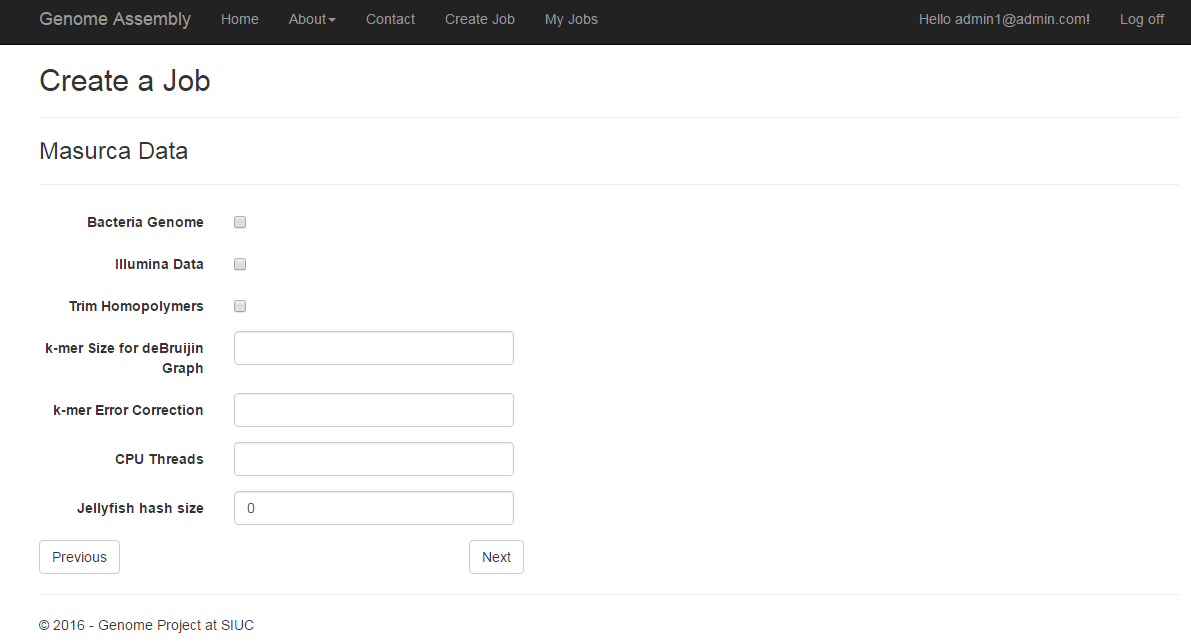
**Figure 5.8**

This step will allow the user to select which assemblers they want to run on their data. Depending on their choices at this step, there could be (in this case), 1 or possibly more steps to the wizard. In other words, if the user only selected Masurca, we don’t care about setting SGA or WGS parameters so we skip those steps.

The user must select at least one assembler to proceed to the next step in the wizard.

### **5.5.4 Step 4**

Assuming that we selected the MaSuRCA assembler alone, then we would have a fourth step.

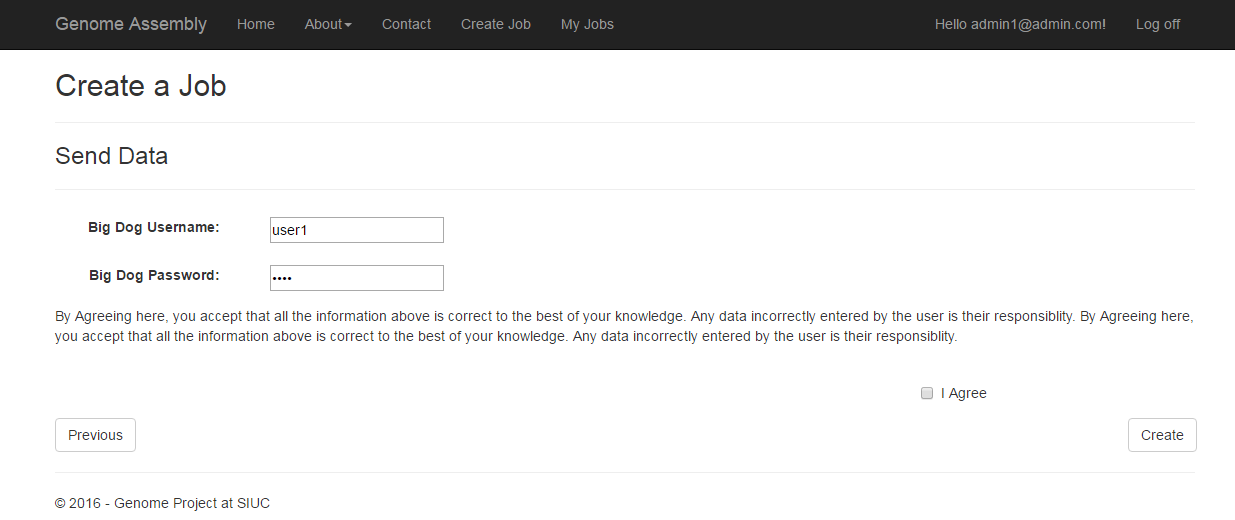


**Figure 5.9**

Here, the information entered is specific to the MaSuRCA assembler. Currently, the only required information to enter is the Jellyfish Hash Size. This is definitely subject to change. If nothing is entered, default values will be selected for the data. It is important to note that if Bacteria Genome is selected when the genome isn’t a bacterial genome, there could be unforeseen consequences and may not even work.

In the future, there will be descriptions of what each of these items are in detail so that it will be easy to figure out what the researcher should and should not enter.

### **5.5.5 Step 5**



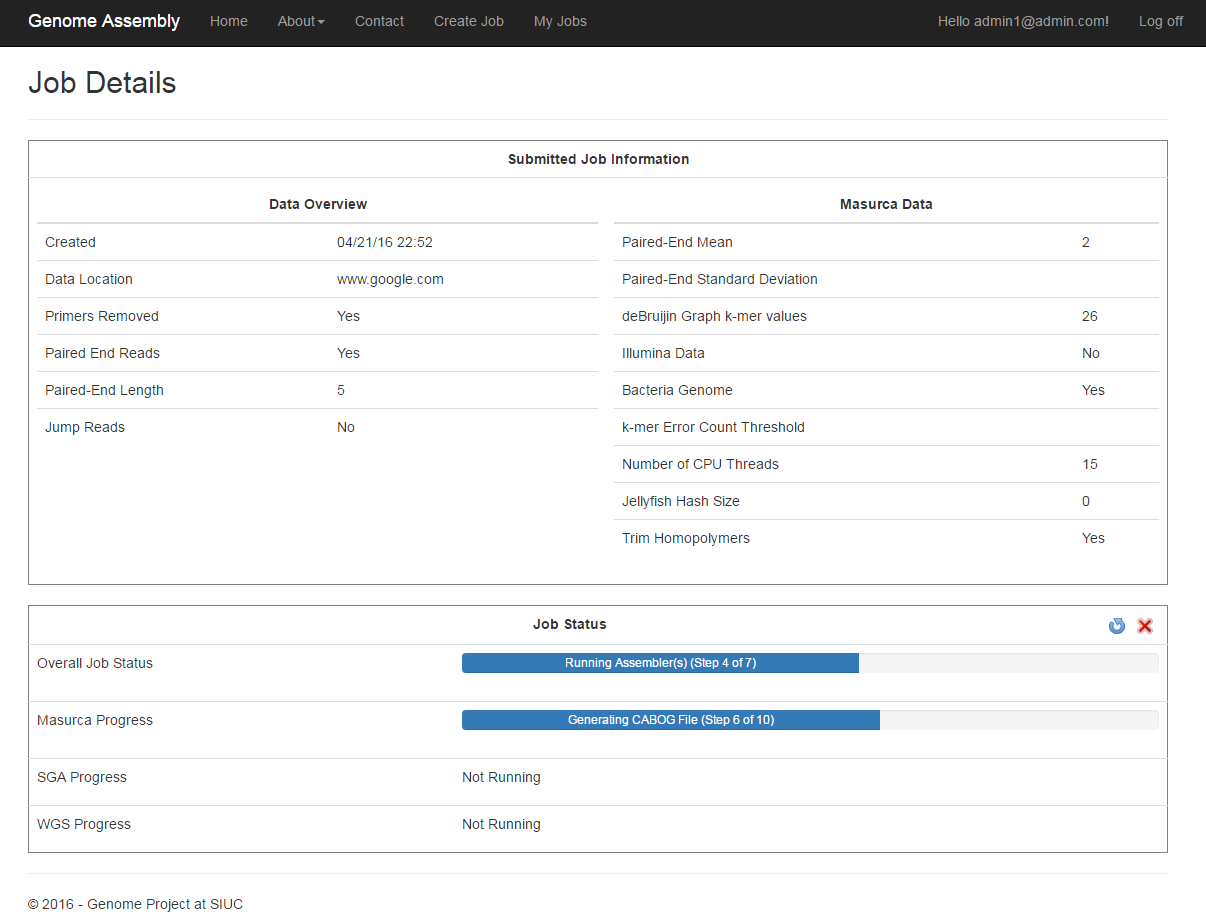
**Figure 5.10**

The final step of the wizard instructs the user to enter their SSH username and SSH password to BigDog into the two fields. These two things are not stored in any capacity after the connection to BigDog is completed.

Once this step completes successfully, a job will have been queued on BigDog and the user will be redirected to the details page of their just submitted job. If an error occurred anywhere along the process at all, the error will be returned to the user in detail at the top of this screen.

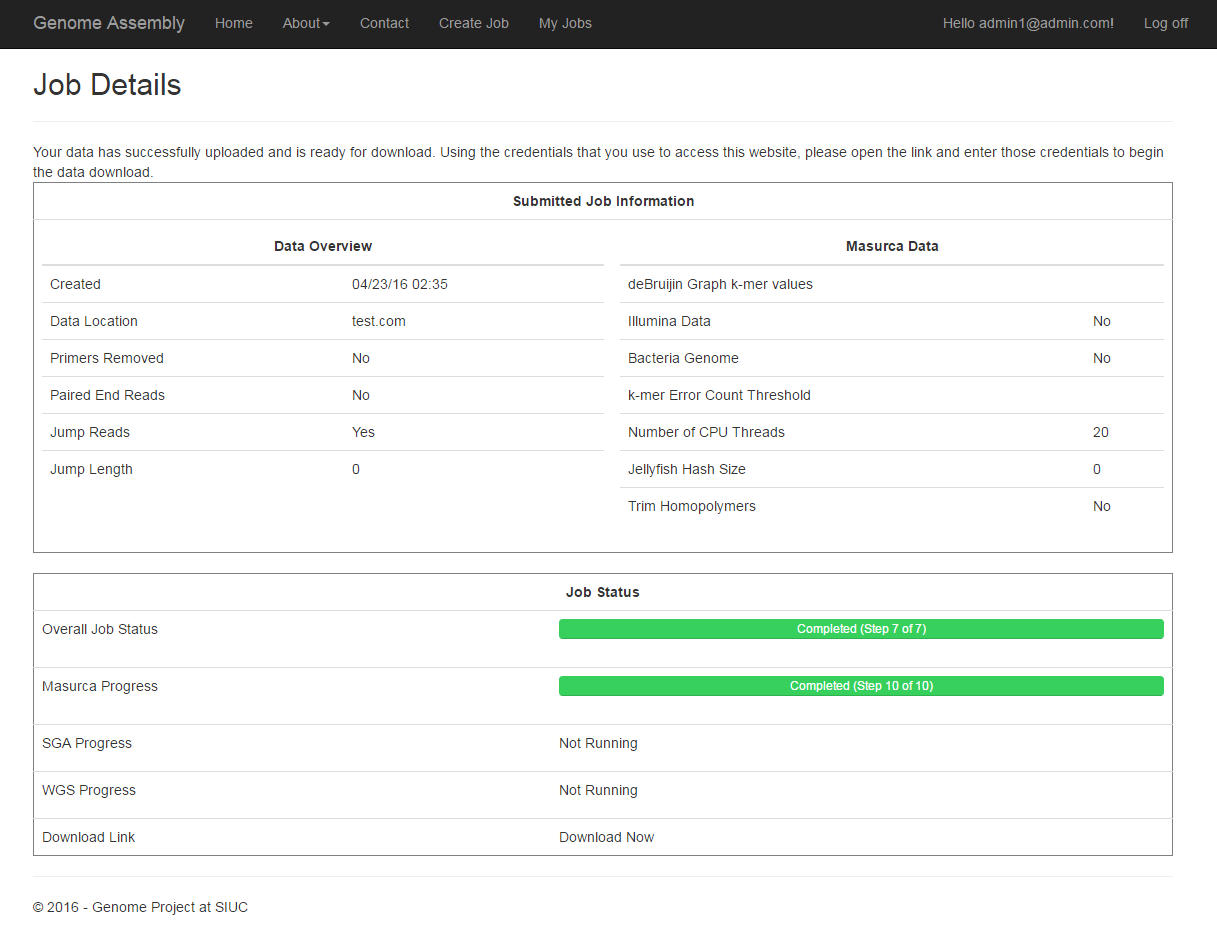
## **5.6 View Job Details Screen**

In this section there are a few variations on what a user will see. There are two main cases, currently running and completed job.



**Figure 5.11**

A currently running job, like above, will display the information about the job and a few options for the user in the Job Status box. On the right, there is an x and a refresh button. Upon selecting the cancel option, the user will be forced to use sign in with their SSH username and password as a stop gap to cancel their job. Updating the status will update the view of where the job currently is in the pipeline. It may take some time for the page to be updated.



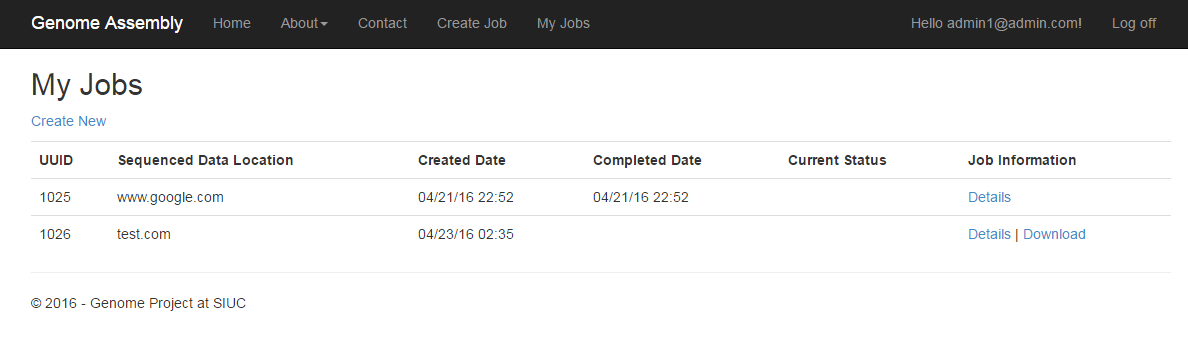
**Figure 5.12**

The above screen shows a successfully completed job. The status bars are green and say completed. Notice that the X and the refresh button are no longer visible. It wouldn’t make sense for them to be either. If the job were to complete unsuccessfully (through an error of some kind), then the status bars would be instead red where the error occurred.

Also take note of the Download Link section at the bottom. A link is displayed for the user to access their data straight from the details page. If you refer to Section 5.7, you can also see that a link is made available there as well for download if the job has completed.

This screen is subject to change and will likely see an overhaul in both form and function prior to the release of the final version of the platform.

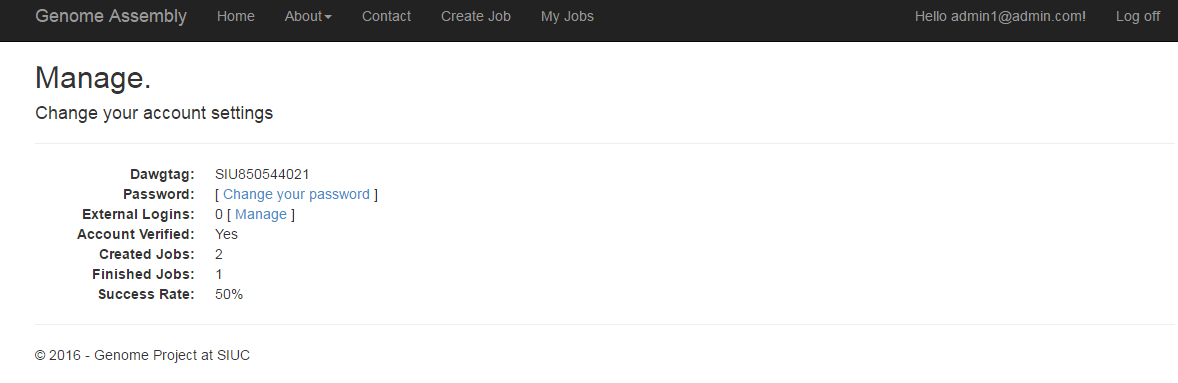
## **5.7 View Jobs Screen**



**Figure 5.13**

By selecting “My Jobs”, the user is greeted with a list of all of their jobs they have successfully submitted to BigDog. If the user hasn’t submitted any jobs, they will be met with a blank page saying there are no jobs to display and a link to display one if they so wish.

## **5.8 Manage Profile Screen**



**Figure 5.14**

This screen displays some information about the user. There are various statistics displayed as well as an ability to change their password. There isn’t anything too crazy on this screen.

However, there will likely be an option to delete their account and all their associated data as well. But that hasn’t been confirmed with the domain expert if that should be an option or not.

## **5.9 Verify Account Screen**

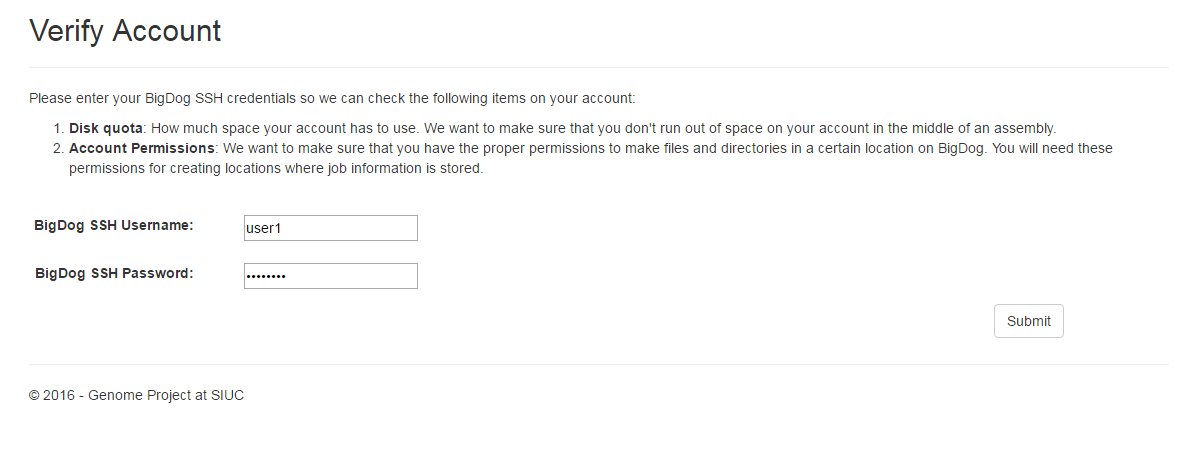
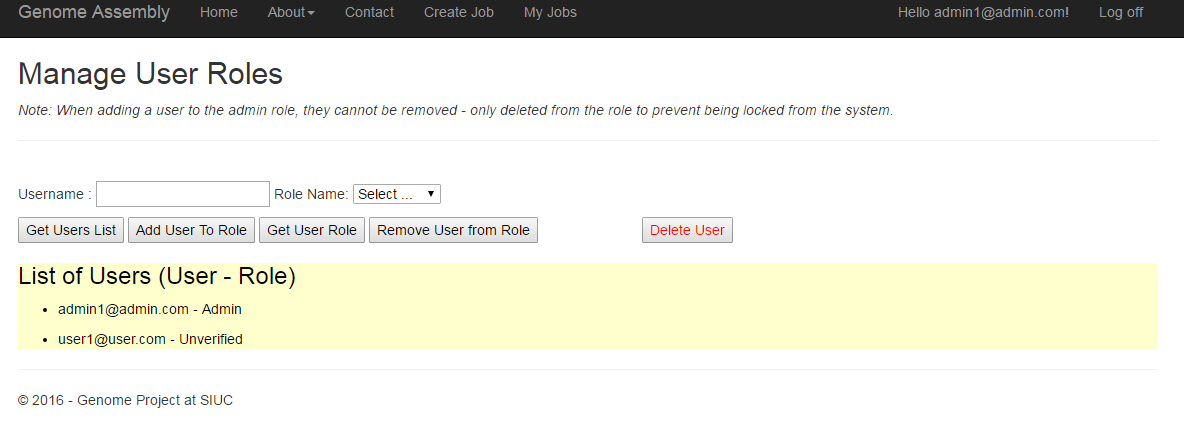


Figure 5.15

This screen asks the user to connect to BigDog to check the quota of their account and make sure they have the proper permissions to do the necessary tasks during job creation.

## **5.10 Manage Users Screen**



An admin has the capabilities to manipulate users such as deleting them among various other things. The screen above is an example of what the page looks like after clicking “get users list”. Typically that yellow box would not appear when the page is first loaded.

# **6. DETAILED DESIGN**

## **6.1 Registration and Login**

The first step of the process starts with the web interface and requiring the user to be on a campus IP. Provided the user is on the campus network, they will create an account on the website using a valid email and supplying their dawgtag as well. It is important to note that unless someone is registered and verified, they will be unable to access any website features.

Once they have created an account, their account will be marked as “unverified”. There are three user roles in the system. The default is unverified and they don’t have any real access to the website. They cannot create a job or perform any tasks. Then there is verified and those users have full use of the system. Finally there is the admin. The only additional task they have access to is the ability to delete users and change user roles.

By default, each user is marked as “unverified” which means an admin will need to change their status to “verified” to receive full access. Ultimately, the goal is to somehow tie into the SIUC AD to run verification of user accounts past the BioInfo group that would check if their email + dawgtag is in that group and then email their account with a verification email. In the meantime however, an admin must approve accounts prior to their use.

## **6.2 Verify Account**

After successfully being “verified” by an admin, a user must first run a check on their account. This involves connecting to BigDog over SSH. When connecting over SSH, we made use of a public library SSH.NET that allows ssh and sftp connections to be made.

Either by attempting to create a job or selecting “Verify Account”, they will be greeted with the verify account page (Figure 5.15).

When the user connects, we run two simple commands. The first of which is:

quota -vs | awk '{print $2}' | grep '[0-9][0-9]\*' | grep -o '[a-zA-Z]'

Essentially, this command will check the quota for the current user, getting the specific amount and parsing the number. Then, if needed, we convert the number from Mb to Gb. We imposed an artificial requirement of 50Gb needed to run a typical genome assembly job. This is subject to change based on test data and advisement from domain experts.

The second command we run is:

mkdir /share/scratch/Genome/testPermissions

The reason we do this is to check and see if they can create a file in the working directory. If you can recall, the first thing we do when submitting a job is to create a handful of directories under their account in our working directory. If the user is unable to write to the directory, then the job will always immediately fail.

If the user lacks permissions or sufficient quota, they will be directed to contact the team managing BigDog in order to satisfy the requirements. In the future, there will be a form the user can fill out to request a change to their account but that has not been made public.

After their BigDog account has been verified, the user will have the ability to create a job. A flag is set to the account that reflects this and is never re-evaluated.

## **6.3 Create Job**

After verifying the SSH account on BigDog, the user will have the ability to create a job. This is immediate. To create a job, simply select “Create Job” in the toolbar. The user will be greeted with Figure 5.6.

All information stored throughout the wizard for creating a job is stored in session and only submitted upon selecting the “submit” button.

As the user moves through the steps, the information that needs to be validated is validated with Javascript at runtime. Although we follow an MVC model and the model is type checked and against all other restrictions placed at that level, it will never reach that point. All validation is done at runtime and when the information is submitted, a valid model will always be submitted.

Validation is done for each step so when the next button is clicked, the current step is validated. So if there are any errors then they are displayed and the user is unable to move onto the next step.

In section 5.5.3, the assembler selection screen is discussed. It is important to note that depending on the assemblers chosen here, the next steps may be different. That is to say, the display of the different steps is dynamic and if a particular assembler is not chosen, then its values are nulled in the database.

The interesting backend work happens on the final step of the wizard. At this point, all information for the model will have been validated and is 100% correct. Once the user enters in their BigDog ssh credentials, the work begins.

Outlining what happens in an abstract manner,

1. The init script is created with the following:
   1. wget [data location supplied by user]
   2. …
   3. wget [data location supplied by user]
   4. ./assemble.sh
2. The assembler script(s) are created with the parameters supplied by the user.
3. A connection using the supplied credentials is made to BigDog over SSH.
4. The following directories are created:
   1. /Job[#]
   2. /Job[#]/Data
   3. /Job[#]/Config
   4. /Job[#]/Output
   5. /Job[#]/Log
5. Change directories to the Config Path.
6. Download the init script from the web server.
7. Download the assembler script(s) from the web server.
8. Change permissions of the Job folder.
9. Determine the load of the two compute nodes and choose the one with least load.
10. Change directories to the Output Path.
11. Add the job to the scheduler by running the init script.
12. Set the SGEID in the model.

The idea is that all of this will be done under the user’s own account. That means that the verification of their account is imperative to the success of a job due to having the correct permissions as well as the quota.

Taking each item in the list in order, the init script is created with simply the linux commands to download all of the data. This is so the data is only downloaded at job runtime to minimalize the amount of space usage prior to a job run. Then the assemblers are run in parallel (d).

The assembler script(s) are then created dynamically from the data supplied by the user. That means if only a single assembler is selected, then only a single assembler script will be generated.

These are the only two things we store locally initially. This is for data reliability and debugging if it needs to occur as well as a central staging point so the information isn’t being generated remotely which can cause a whole host of other potential issues.

The running theme is to keep the SSH window open as short as possible.

After the scripts are created, a connection to BigDog with the credentials supplied are made. A reminder that those credentials are never stored and are immediately discarded upon completion of the transaction.

Once a successful connection has been made, then the user creates all of the directories that will be used during the course of a job. The directories are very simply created by using:

mkdir [directory] -p

After creating all of the directories, the focus is moved to the config directory where the config scripts are downloaded so that they may be run. This is accomplished simply by using:

cd ../Config

wget [URL to scipts]

After the scripts are downloaded, then focus is again shifted to the output path after computing the relative node load of each node and changing the permissions for the folder so we are certain it is all viewable. This is done by simply using:

chmod 777 -R [job directory] // Change permissions

qstat –f | grep [node] | awk ‘{print $4;}’ // Get node load

cd ../Output // Change directories

Finally, the job is added to the scheduler by invoking the init script with the following linux command:

qsub –pe make 20 –V –e [Error Log Path] –o [Output Log Path] –b y –l hostname=[node] –N [ username[UUID]] ./init.sh

After the job is added to the scheduler, there is one last thing to do prior to returning back to the controller. The scheduler ID needs to be grabbed and assigned to the model. To grab the ID, the following command is run:

qstat –f –u \[SSH Username]\ | grep [username[UUID]]

Then a regex is run on the return value only grabbing the number.

Each command is run black-boxed meaning if any one command were to fail, then the entirety of the job is aborted and an error reflecting the nature of the failure is returned to the user on the webpage via the error variable.

Upon successful completion of the tasks outlined above, the job is only now added to the database. So if an error were to occur, the model would never make it to the database and the user would have to diagnose the issue or contact the appropriate individuals to fix the issue.

Once the controller returns, the user is redirected to the details page with a summary (Section 5.6) of the information that was entered and a brief overview of where the job is currently at in the computational process.

## **6.4 Update Job**

After a job has been created, it is important to keep the user informed as to its current status. That is not only to keep them well informed but also allow the system to notify them when the job has ultimately completed.

There are two methods that are used currently to update a particular job:

1. Background task
2. Manual task

The first method is particularly important as it allows the system to run an update regardless of user interaction. The second method explicitly updates the job regardless of the current background cycle.

There are pros and cons to both methods. For a background task, there is the following,

1. Pros
   1. Automated process.
   2. Updates all jobs that are currently running.
2. Cons
   1. Will be dumped by IIS since asynchronous tasks aren’t registered with the App Domain.
   2. Can potentially hang/exit in an undetermined manner resulting in possible data corruption.

It is important to note that the list isn’t exhaustive but outlines a couple major concerns that must be accounted for when using background tasks in a web environment. The biggest issue is that IIS recycles the App Domain every 29 hours which forces all tasks to cease and restart. However, background tasks aren’t registered with the App Domains so IIS doesn’t ever ask for the task’s permission if it can shut down.

That means that locks are required to force IIS to wait until the background task has completed before recycling. The update shouldn’t take too much time at all realistically so the time it is asked to wait shouldn’t be long at all. But this all adds additional complication to the code and forces it to function in ways not originally intended.

For a manual task, they are straight forward and left to the reader to determine. Really the benefit seen is that an individual job can be updated whereas with a background task bundles of jobs are updated. So time is saved and the user can get an immediate status update rather than waiting for the background task to come back around. But this requires the user to press a button which is a bit more involved than having to do nothing. Ajax is a solution to that problem and will likely be implemented in a later release.

The process that occurs when a job is updated is two-fold. First, the overall status needs to be determined and then assembler specific statuses need to be determined.

Both of the cases are carried out in a single general method that will run either once or looped. Instead of asking for credentials from the user, a special account is used to check the statuses of the jobs. This is the primary reason that the job directories need to be accessible to others so that the status can be determined as well as when the job has completed it can be packaged and sent off.

After a connection is made, the first thing that is checked is how many assemblers have been chosen. This is so the method can populate the step list that shows the overall steps (dynamic based on number of assemblers chosen) as well as know which assemblers to check for a status.

The step list is constructed in a hashtable as follows:

1. Program Queued
2. Data Conversion
3. Running Assemblers
4. Finished Assembler 1 of ?
5. Finished Assembler 1 of ?
6. …
7. Finished Assembler ? of ?
8. Data Analysis
9. Uploading Data
10. Complete

The assembler specific steps will vary but the masurca step list is stored in a hashset which is comprised of the step, the file name associated with that step, and the step description.

So after generating the appropriate lists, the first thing to check is if a job is running. That is done by running the following command:

qstat –j [SGEID]

If the job is running, then the next thing to check is if it has completed on any of the assemblers. So the directory focus is shifted to each assembler output directory under /Output in the Job directory. Then the hashset is looped to check for each filename and an assembler specific step is determined. If that step is the final step, then the assembler has completed.

We know if it has completed by checking the log name for “masurca\_finished.olog”. If that file exists, then it has finished. Otherwise, it is still running.

This is done for all of the chosen assemblers and their current steps are updated in the process.

If the job is NOT running, then the first thing to check is if it completed successfully or errored out. For both methods we do the same process with slight variation in the small details.

In order to determine if a job finished successfully, we run the following command:

find [Assembler Success Log]

This is done for all of the chosen assemblers. Now the job needs to be packaged and downloadable for the user. First, the data is compressed by using the following command:

zip -9 –y –r –q [OuputName] [Job[ID]]

Once the data has been compressed, then another connection is made from BigDog back to the fileserver. Creating an SFTP connection with a private key just like the update method, the connection is made.

Then the data is uploaded to the user’s folder on the web server’s FTP and a download link is generated based on where that location is and stored in the download link parameter in the model for the particular job. That signals to the rest of the platform that the job has indeed completed.

Finally, an email will be sent to the user who created the job that it has completed. This feature has not yet been implemented in the design.

It is important to note that throughout the process, the job’s details page is updated with its current status whether it is still running or if it just completed. The information gathered from the update job is reflected back to the user via the details page.

If a job happens to fail rather than succeed, all of the same measures are taken except that information regarding the failure is included in the email and an explicit note is included in the resulting data folder.

Another point to note is that regardless of whether it is a background task or manual, the exact same method is called.

## **6.5 Manage Users**

Access to the website is through the use of roles. Currently there are three roles that a user can have. They are only able to have a single role at a time. That means all roles are mutually exclusive when their access isn’t necessarily so. There are the following roles:

1. Admin (Administrator)
2. Verified
3. Unverified

Upon registering for an account, a user is set to unverified. This is the default role for any new users and those users must be approved by an administrator (Section 5.10). Future functionality would have this process automated by checking against the BioInfo group in AD and automatically admitting users from that group into the website.

An unverified user has little to no access to the website. They cannot do anything but wait to be verified.

A verified user can access all of the functions of the website save for manipulation of other users.

An admin can access all of the same things as a verified user but can also manipulate other user’s information.

There are several options that an admin has when it comes to managing users. They can change the roles of a user (remove them from a role, move them to another role), they can get the entire list of users using the system, they can get the roles of a particular user, and they can permanently delete a user.

The process of getting the users and a particular user’s role are straight forward. A call is made to the database and the appropriate items are returned.

However, adding a user from a role and removing them from a role are different. Since each role is mutually exclusive, that means if a user changes roles, then they must also be removed from their previous role.

In addition, moving a user to an admin makes it possible that they cannot be removed from that role. If there is a single admin left, then all others have either been removed from the role or deleted, then the remaining admin cannot be removed from the role or deleted. This is to prevent system lockout. Since it is the job of an admin to approve users, it would result in a lockout if the last admin is removed from that role.

The delete user option is also similar in that it won’t let an admin delete the last admin (themselves) if it comes down to it. In addition, prior to deleting a user, a dialog box pops up warning the user that the action results in non-recoverable data loss of the user account. The user is completely removed from the system without backup. Their submitted jobs will have been kept unless that policy is changed.

## **6.6 Analysis**

After a job has completely finished running on assemblers and before it is packaged for download by the researcher, data analysis is run on the resulting data. Although not fully implemented yet, once a second assembler is installed it will become a major priority to get the following up and running.

Using the outputs from SGA, WGS, and MaSuRCA we will run a comparison analysis to determine the “best” final output of the three algorithms. To do this, we will run a BLAST efficiency analysis which will provide a more complete picture than any one algorithm can produce.

Then we will assemble the contigs using a CAP3 assembly program to produce the final output of the analysis.

The goal is to also determine if there is also a better set of data parameters that produce less error-ridden results as well as a tighter fit in terms of DNA representing the true genome. This is a primary focus moving forward done through repetitive analysis of known genomes run through this algorithm with different parameters.

# **7. RESULTS**

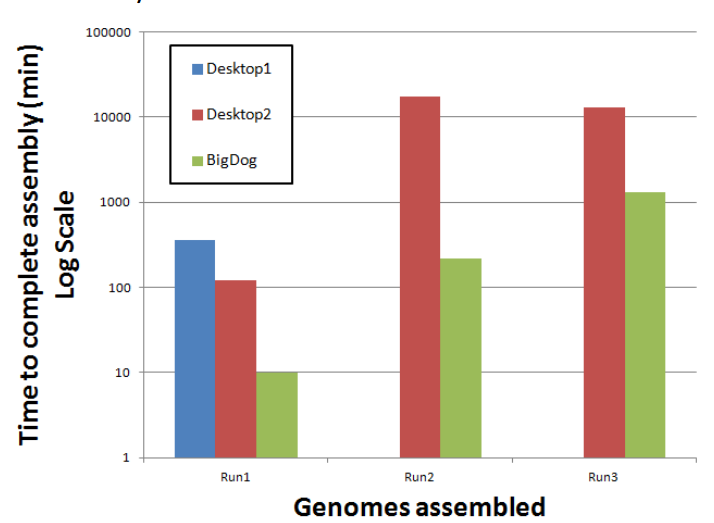
Running the data through BigDog has ultimately increased efficiency considerably. In a small case study, we selected three computers and three different data sets.

The following were the test device:

1. **Desktop 1**: Running Ubuntu with 4 CPU and 4GB RAM with Geneious version 7.1
2. **Desktop 2**: Running Windows 7 with 4 Xeon CPU and 64GB RAM with CLC Workbench version 5.6
3. **BigDog Node 24**: Running CentOS with 20 CPU and 750GB RAM with Masurca version 3.1

The following were the data sets:

1. **Rafflesia lagascae**: 454 pyrosequencing, DNA-genomic 2 samples, 2 million reads and 400 million DNA bases.
2. **Arabidopsis thaliana**: Single End Illumina HiSeq RNA-total 12 samples, 300 million reads, and 15 billion RNA bases.
3. **Rafflesia leonardii**: Genome in depth Paired-end Illumina HiSeq DNA-genomic 4 samples, 370 million reads, 11 billion DNA bases.

Running all three genomic data sets through each test device resulted in an incredible differential in terms of running time. On a logarithmic scale, the BigDog compute node 24 destroys the other two test devices with Desktop 2 not even able to complete data sets 2 and 3.

Prior to using BigDog to conduct the genomic assemblies, it would take upwards of a week to run data. But looking at the data below, desktop 2 took almost a complete week to compute data set 3 whereas compute node 24 was able to do it in little over half a day. The time differential is enormous. Consider the case in which the process can be parallelized between compute note 24 and 25. The time savings could be even further increased depending on the relative load of the nodes.

It is clear as to the need to interface with this system to provide speedy results.