Calculating All Pairwise Similarities from the RCSB Protein Data Bank: Client/Server Work Distribution on the Open Science Grid

TR-09-03

December 10, 2009

RENCI Technical Report Series
http://www.renci.org/techreports

Chris Bizon, Renaissance Computing Institute,
University of North Carolina at Chapel Hill

Andreas Prlic, RCSB PDB Protein Data Bank
University of California, San Diego

Calculating All Pairwise Similarities from the RCSB Protein Data Bank: Client/Server Work Distribution on the Open Science Grid

Chris Bizon, Renaissance Computing Institute, University of North Carolina at Chapel Hill

Andreas Prlic, RCSB PDB Protein Data Bank University of California, San Diego

Introduction

Proteins can have various degrees of similarity. If two proteins show high similarity in their amino acid sequence, it is generally assumed that they are closely evolutionary related. With increasing evolutionary distance the degree of similarity usually drops, but proteins can still show similar activity in the cell and have an overall similar 3D structure, even if the sequence similarity is low. The detection of such remote similarities is important in order to infer functional and evolutionary relationships between protein families and is a core technique used in protein structure bioinformatics. The goal is to establish regions of equivalence between two or more molecules.

The RCSB Protein Data Bank (PDB) is a leading primary database that provides access to experimentally determined protein structures, nucleic acids, and complex assemblies. PDB is a vital part of the infrastructure supporting biomedical science worldwide and is used by around 200,000 unique scientists per month.

While protein sequence comparisons can be computed quickly, the calculation of protein structure alignments is much more time consuming. The RCSB PDB has recently started to add new tools to the site, that allow users to quickly identify protein sequence neighbors and run pairwise protein structure comparisons. In order to allow users to also quickly identify more distant 3D relationships the goal of this project is to provide a pre-calculated set of all vs. all 3D protein structure alignments.

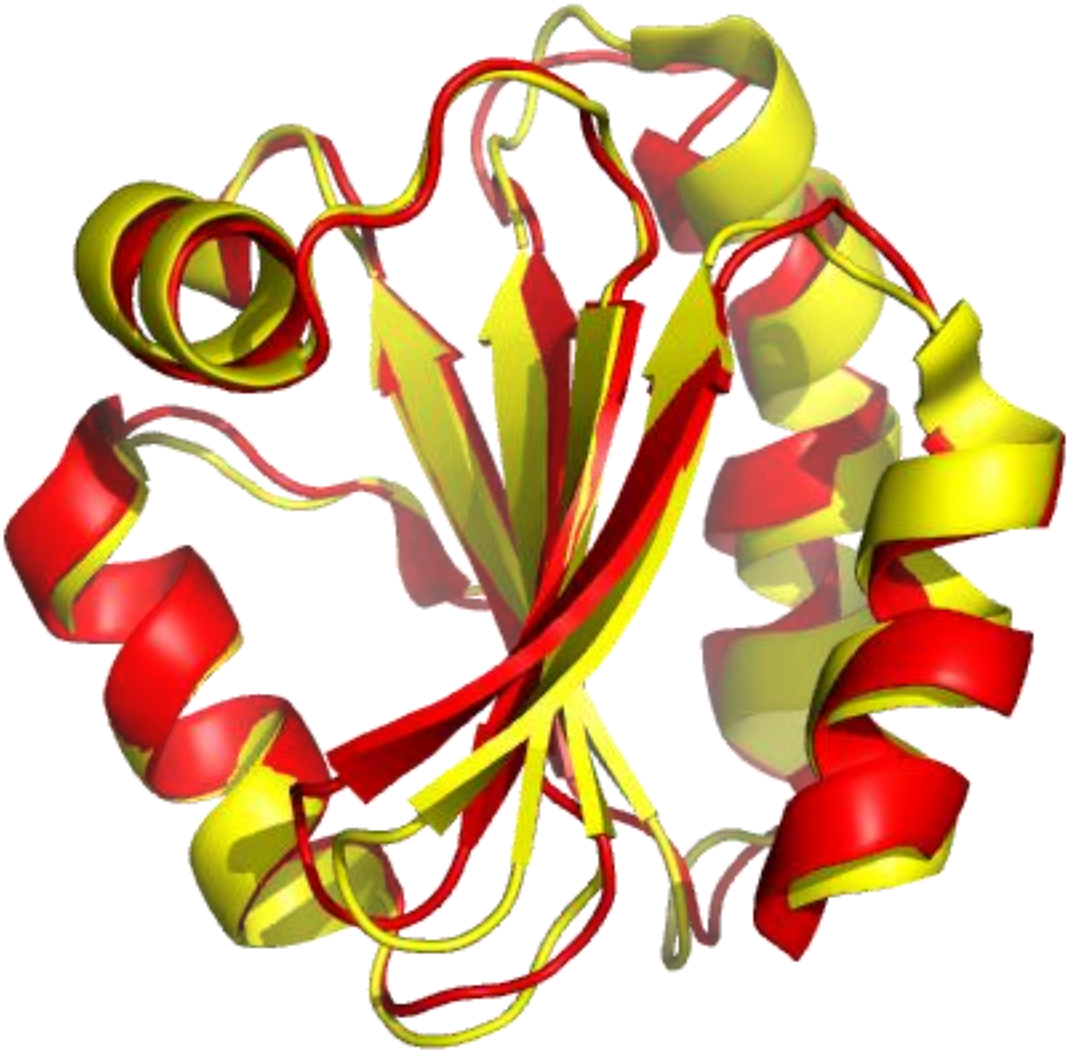


Figure 1: Structural alignment of thioredoxins from humans and the fly Drosophila melanogaster. The proteins are shown as ribbons, with the human protein in red, and the fly protein in yellow. Generated from PDB IDs 3TRX and 1XWC. (Image taken from http://en.wikipedia.org/wiki/Structural_alignment)

Conceptually, calculating many pairwise similarities is no more complicated than calculating a single similarity. However, the size of the calculation will dictate the practical means that must be used to solve it. Given that 61000 protein structures inhabit the PDB, with approx 130.000 chains, the initial estimate for the number of similarities would be 8.4×10^9 comparisons. This nr of comparisons can be reduced, by using obvious sequence similarities and selecting representative structures from sequence clusters. Assuming that proteins with very similar primary sequences will also have similar three dimensional structures, we can let a single protein stand in for a group of proteins with similar sequences. This is accomplished by using Blastclust to calculate the sequence clusters (detailed strategy documented on PDB web site). Additionally, because alignments are symmetric, giving the same result for protein A vs protein B and protein B vs protein A, we can reduce the number

of required similarity calculations to 140 million. Additional complexity is added by multi-domain proteins. For these it will be necessary to provide alignments for the whole sequence as well as for the composite domains. Estimating that each pairwise calculation will require 2 seconds leads to an overall estimate of ~500 CPU weeks.

Calculation of these pairwise similarities, then, is structured as a series of short calculations that are independent of one another. However, the sheer number of such calculations leads to a prohibitively long run time. This combination of factors makes the present problem a good candidate for a solution via distributed computing, in which a large number of processors each solve a small part of the matrix independent from one another.

In particular, we have used the Open Science Grid (OSG) as the platform for performing these calculations. The OSG, funded by the Department of Energy and National Science Foundation, is the US contribution to the worldwide Cyberinfrastructure to support data analysis for the Large Hadron Collider. It comprises tens of thousands of processing cores across more than 80 university and government labs. One component of the OSG is the NSF funded Engagement Program, whose purpose is to bring the power of the OSG to new science domains beyond high energy physics. The Engagement team has successfully demonstrated providing millions of CPU hours via OSG to a diverse set of projects in numerous science domains.

A Client/Server Solution on the Open Science Grid

The Simplest Approach and Its Problems

The basic questions that must be answered in creating a distributed computing application are:

1. How will the work be divided among available compute nodes?
2. How will data be staged to and from the compute nodes?

Each protein in the PDB can be exported as a single file called a PDB file. In the simplest implementation of pairwise similarity calculation on the OSG, we would map one pair of PDB files to each OSG job. In other words, each cell in the similarity matrix would be calculated by a different job. The two PDB files would be staged to the remote processing core at runtime and cleaned up after the job completed. Though conceptually straightforward, this solution is not practical for several reasons. Most importantly, each job has a latency. This latency arises from finding available processors that match the job requirements, waiting in queues, and staging input and result files. To maximize the work-to-latency ratio (assuming fixed job latencies), we prefer longer jobs. In practice, we typically aim for jobs that last between 4 and 12 hours. This gives a good work/latency ratio, while staying below any time limits imposed by owners of OSG resources.

The next stage in sophistication, then, would be to bundle up a group of PDB files, stage it into the remote processor, and calculate all similarities between these protein. Creation of such statically defined jobs is a standard paradigm in work on the Open Science Grid, but in the case of the current problem, it has several shortcomings.

The first, and most fundamental difficulty is the management of such a large computation. Assuming that we could solve the work-balancing issue described below, we would divide the work into 12600 jobs each running for 8 hours. Each job requires numerous files, such

as submit files, execution scripts, and input and output data files. A vast number of files would be stored to disk and have to be organized and coallated in some way. All of this data management is further complicated by the fact that these files will not be on the users' home machine, but on a submit host for OSG, increasing the distance between the user and the calculations. Finally, there would be a need to monitor the overall state of this computation. While the infrastructure for submitting jobs would withstand the large number of jobs, it does a poor job of helping users understand the decomposition of their work into jobs. A user does not know, without extra work, what portion of the total work has been submitted, has succeeded or failed. This management stands as a further obstacle to a researcher who simply wishes to perform calculations, not manage distributed computations.

The second difficulty in static job definition is in the creation of equal-length jobs. if each pairwise protein similarity calculation took an equivalent amount of time, we could easily divide PDBs among jobs with the assurance that each job would take a similar amount of time to complete. However, the similarities being calculated are dependent upon the size of the proteins; the proteins in the PDB range from small proteins consisting of 20 amino acids, up to several hundred amino acids long protein chains. If no attention is paid to this variance, we will by chance create jobs with many large proteins, causing jobs that run longer than allowed by resource owners. To compensate for this, we might include fewer PDB files in each job. While long jobs would now fall under execution limits, short jobs would be too short, and latency would again become an important factor in the overall wall time of the calculation. The problem of load balancing is made still more difficult by the heterogenatity in computing resources found in the OSG.

The third difficulty is the repetitive staging. Each protein in the calculation is going to be compared with every other protein. Regardless of how work is split up, this means that the same protein PDB file will be staged with many jobs. Potentially, many of these jobs will run at the same site, so that repeatedly staging the same PDB file represents a waste of network and latency time.

A Client/Server Approach

One approach that addresses these difficulties is to replace statically-defined jobs with a dynamic client/server model of manging work coupled with pre-staging the PDB data; See Figure 2.

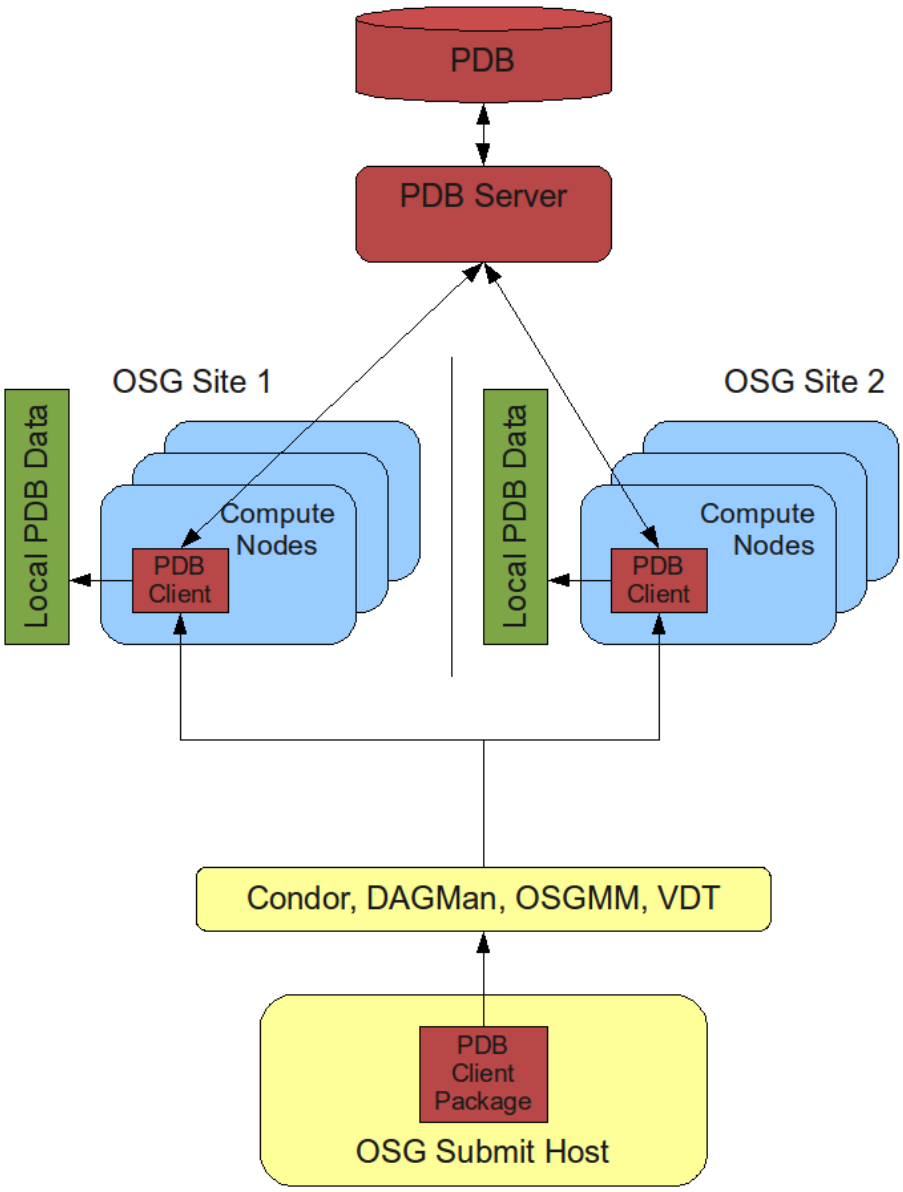


Figure 2: A client/server approach simplifies workflow and data management for the PDB problem. Standard OSG technology is used to provision the PDB client package from the OSG submit host to OSG compute nodes. The PDB client contacts the external PDB server, and is given a chunk of work to do using pre-loaded PDB data. The results are returned directly to the PDB server.

A server runs outside of the OSG infrastructure. It can access a database that contains information about all pairwise calculations that need to be run, as well as results for those

that have already been performed. The code that runs on OSG processors is a client for this server. When one such clients starts, it contacts the server and requests work. The server knows what work is required, and sends to the client a list of proteins to compare. The client then performs calculations on the pairs indicated by the server and returns the results to the server. If the client has run beyond a fixed amount of time (usually about 12 hours), it terminates; otherwise, it requests further work from the server. The client/server model effectively decouples the process of workflow control and the process of resource allocation; workflow is controlled by the server, while the OSG stack allocates the client resources independent of the server.

With this model, the balancing problem vanishes. The client requests further work until it reaches a fixed time limit. The client will never underrun its time (except in the case of failures), and will never overrun its time by more than the amount of time required to service one request. All clients, then, will have the same lifetime, within a window whose length is bounded by the size of one work request to the server. Making the size of each request too small will lead to many requests, potentially overburdening the server. In practice, as long as there is a separation of scales between the most atomic unit of work and the requested walltime for a client, a balance can be struck during the tuning portion of development. Note also that this dynamic balancing automatically deals with differences in processor speed, since we are specifying the time spent on each processor rather than the amount of work done there.

More importantly, job management is vastly simplified with the client server model. All inputs and outputs all pass through and are stored on the server; the state of the computation is known exactly rather than being inferred by looking at job inputs and outputs stored on disk in many different directories. The ability to deal with failures is similarly improved. When statically defined jobs fail, the user must either resubmit the entire collection of jobs, or create a new run containing only the failed jobs, or identify only the particular calculations that failed, and build a new run of new jobs only containing these calculations. Those three options trade off user time in diagnosis with computer time in recalculating successful work, with no optimal solution. In the client server approach, the server is independent of jobs and runs, and simply operates the unit of work that is parceled to clients. If a client crashes, or is evicted, whatever successful work has already been reported to the server becomes part of the solution, no matter the fate of the client. If a run is successful except for particular calculations, only those calculations will be later recalculated with neither a waste in computation or a waste in user time. Further, the server can aggregate failures, identifying work that fails repeatedly for a legitimate reason from the random failures endemic to distributed computing.

Furthermore, the job management burden is reduced for the user because they can host the server on their own machine. In the simple statically-defined case described above, a user must double stage data in and out of the OSG. First, the data must be moved from their local machine to a submit host. The data flows from there to OSG processors for computation and results are staged out to the submit host. From the users' perspective there is an extra management step in getting useful results back to their home institution. In the server-managed paradigm, data flows directly from and to the users' server. The submit host is still used to start a set of jobs and monitor whether they are running, but this resource managment is significantly easier than the data and work management handled by the server.

The problem of repetitive staging could be addressed without recourse to the client/server model. Nevertheless, the new model does make implementation of the solution more straightforward. In particular, we make use of shared filesystems on OSG resources.

Many OSG resources provide a file system that can be used to store results to save from run to run, called OSG_DATA. We make use of this system by first pre-loading all PDB files into a specified location on OSG_DATA. When a client requests work from the server, it receives not a full PDB file, but a reference to a PDB file that has been preloaded into OSG_DATA. The client then retrieves the PDB file from the local file system, saving network traffic. However, the PDB changes over time, with several hundred records added or modified every week. The client may find, therefore, that it is asked to use a PDB that has not been preloaded. In this case, the client can download the PDB file from the server and add it to the cache of files in OSG_DATA. Subsequent clients needing that PDB at that site will then find the PDB in the normal location.

Implementation

The solution described in a general sense above is implemented against the infrastructure background of the Open Science Grid and tools provided by the Engagement Virtual Organization. In particular, the solution is composed of three software packages. The first is a web server running at the PDB, which is capable of doling out jobs and receiving results. The second is a ODB client that is capable of communicating with the server and performing the alignment calculations. The third component is the group of scripts that interact with the Open Science Grid infrastructure to provision the PDB client to the appropriate OSG compute nodes. We shall discuss these in the reverse order, following the order in which a typical run operates.

Work on the OSG is organized into jobs and runs. A job describes a unit of work to be staged into a compute node, run, and staged out. A run is a collection of these jobs, possibly containing interdependencies. In the present case, there are no interdependencies, so a run is merely a collection of jobs that can run independently of one another. Given the existence of server and client packages, the first problem is getting the client package to a large number of OSG compute nodes where it will be able to run. The client will require certain features of the OSG node. In particular, the node must have sufficient memory to run the client, it must have a java runtime available, and we only want to run on sites where the PDB data has been preloaded. Finally, the client will need to be able to make HTTP connections to the server.

Resources on the Open Science Grid are controlled entirely by their owners, implying that no particular set of circumstances can be guaranteed. At one site, outgoing connections may be allowed, while being forbidden at another. Similarly, some sites may have a shared filesystem allowing data preloading, while other sites may not. Finally, software versions and hardware vary widely. To deal with this variation and make sure that the client is provisioned to sites where it will be effective, we make use of the Open Science Grid Match Maker, or OSGMM (http://osgmm.sourceforge.net/). OSGMM gets site information from the Resource Selection Service (ReSS), and augments it with information from OSGMM's verification jobs to produce a final site description. This allows users to submit OSG jobs using the widely known mechanism of condor submit scripts, and take advantage of the powerful Condor matchmaking facilities.

```
requirements    = ( (TARGET.GlueCEInfoContactString =!= UNDEFINED) \
                  && (TARGET.Rank > 300) \
                  && (TARGET.OSGMM_MemPerCPU >= (500 * 1000)) \
                  && (TARGET.OSGMM_CENetworkOutbound == TRUE) \
```

```
                    && (TARGET.OSGMM_SoftwareGlobusUrlCopy == TRUE) \
                    && (TARGET.EngageSoftware_Java_v15 == TRUE) \
                    && (TARGET.OSGMM_Data_PDB == TRUE) \
                    && ( isUndefined(TARGET.OSGMM_Success_Rate_andreas) \
                          || (TARGET.OSGMM_Success_Rate_andreas > 75) ) \
                )
```

Listing 1: The condor submit script defines requirements for the PDB job. In particular, we include information about required memory, that the site allows outward network traffic, and that we have access to globus-url-copy, java, and the preloaded PDB data. Further, we only want sites that have performed well in recent tests by the OSGMM (leading to a high Rank), and those that have a high success rate for this user's jobs.

We submit one condor script with such requirements for each job. The collection of jobs into a run is handled by the DAGMan meta-scheduler for condor. DAGMan allows condor jobs to be managed at a higher level, including groups of jobs whose dependencies form a directed acyclic graph. Here, the dependencies are nil, but DAGMan provides a simple mechanism for managing the multiple concurrent jobs as a single run. When the user submits the dagman script to condor, it becomes aware of the multiple job-level condor scripts and begins submitting them. When an OSG job is started from one of these condor jobs, it is first matched with a usable resource as described above. Once this is accomplished, the job is transferred to the OSG site, where it enters a local provisioning system, and eventually lands on a compute node.

In the condor submit script, we define an executable and tell condor that we need to transfer this executable to the compute node. This remote execution script, then, is the code that will be run on the remote OSG node; its stdout and stderr will be automatically returned to the submit host upon job completion or failure. Our remote script has several responsibilities. First, it surveys the node on which it landed, running uname, ulimit, env, and catting /proc/cpuinfo and /proc/meminfo. The output of these commands is invaluable in the case when a job fails.

The second responsibility of the remote execute script is to bring the PDB client from its home on the submit host. The client is retrieved with GridFTP by the globus-url-copy command. While the client could be sent to the compute nodes directly with the execution script, we have found that GridFTP scales better in staging data to multiple sites simultaneously, and we use it for anything other than the execution script itself. Note also that this allows updates to the client with no change to the machinery of submitting jobs. In principle, the client could indeed be hosted along with the PDB server, but we do require the presences of a GridFTP server. Practically, this means that the submit host is the most likely location from which to serve the client.

Once the client has been pulled from the submit host, the third responsibility of the execution host is to run the client. Finally, once the client has run, it is the responsability of the execute script to clean the temporary directory in which we have been running. Many sites will not immediately delete this work directory when a job finishes, in case the user is relying on the results of one job feeding another. Since we have no dependencies, we remove the directory to preserve disk space on the site. Note that in a normal OSG run, we would include stageing out data as one of the resonsabilities of the remote execution script, but in the current case, the returning of results is handled by the client directly.

The client is a Java implementation of the FatCat algorithm (http://fatcat.burnham.org/ Yuzhen Ye & Adam Godzik. Flexible structure alignment by chaining aligned fragment pairs

allowing twists. 2003. Bioinformatics vol.19 suppl. 2. ii246-ii255.). It communicates with the server via a simple RESTful - XML over http protocol. The client accepts a command line argument describing how long it should run, the location of the local PDB data files, and an ID. The client asks the server for work, performs the work, and returns the results back to the server. It then requests more work, until it passes the prescribed time limit, at which point it exits. When the server requests a certain alignment, it does not send the PDB files to the client. Instead, the PDB client looks for the PDB files in the location indicted by the command line argument. The ID is constructed from a run ID, a job ID, and the name of the OSG site. It is transmitted with messages to the server, allowing the server to track the amount of work performed per run or job, as well as how much work is done by particular OSG sites.

The server is based on Apache Tomcat. It does not attempt to keep track of the state of clients. Instead, it simply serves work as requested and accepts results. Client IDs are recorded, but only for accounting purposes. Because the server knows what jobs it has received results for and what work it has sent out, it can easily coordinate the jobs simply by not sending out the same set of work to multiple clients. In addition, the server can be instructed to send out a kill signal in response to a request for work. A client receiving this signal shuts down with a successful exit signal, so that it is not rescheduled by condor. This kill signal was implemented in response to the observation that clients sometimes did not stop when a job was removed via condor.

The communication interface between the server and client is driven by the requirement that all messages must be client initiated HTTP. While many sites will allow outgoing-initiated messages, most will not allow incoming-initiated messages for security reasons. Further, the OSG sites usually reside behind firewalls with many ports blocked by default, but most open port 80 for HTTP traffic.

Performance Tuning

With these software components in place, we can set several parameters to increase the performance of the application and discover the rate-limiting step. Chasing bottlenecks is aided by the ability to analyze a distributed run as described below. In the condor submit scripts, we specify a minimum amount of RAM as 500MB. We want this number to be as small as possible, since this will disallow the smallest number of processors, increasing our throughput. To this end, the client was optimized to minimize the amount of memory required; with a 500MB limit, very few processors are removed from the pool of allowed resources. In a normal OSG application we would also tune the length of a run by portioning the data evenly across processors. With the present client/server model, this is not required, since each client will simply run for a fixed amount of time and then exit. We simply set the maximum allowed time to 50% higher than we expect the client to run.

With jobs able to make effective use of OSG resources, the remaining performance tuning concerns the client/server interaction, and the server backend. Because there is a single server doling out work, that server can easily become a bottleneck. If 1000 clients are running concurrently, each working for one minute before requesting new work, the server will have to deal with approximately 1000/60 ~ 17 communications per second. This indicates several features. First, we will be able to reduce the spacing of communications by increasing the amount of work given to each client. Second, we will need to optimize the rate at which the server can store results and hand out the next portion of work. Finally, it

is likely that the server will, in the end, be the rate limiting step, rather than the number of processors that we can access.

The OSG allows to calculate a large amount of data in short time. When setting up the server to consume the results it is important to also develop a backup strategy for this incoming data. Copying several hundred GB of data across file systems takes a while. At the time of writing we are still working on improving the backup strategy for the server and are exploring MySQL replication and binary file system replication strategies.

Results

Overall Metrics

A total of 140 million alignments have been calculated for the all vs all comparison. Out of this total, 122 Mio. have been calculated on the OSG, the rest on other resources. The calculations on OSG have consumed ~102000 CPU hours. This includes initial test runs in order to scale up the central submission server. The average calculation time for a pairwise alignment was about 3 sec per alignment, which is more than was estimated in the beginning. Reasons are probably inefficiencies in the initial runs (both server and client), which has been fixed in the course of the process. The second half ran faster than the estimate.

The capacity at the present is to align 10-20 mio alignments per day. Test runs with 1000 parallel jobs caused occasional write-bottlenecks on the central submission server. As such, usually 800 jobs were submitted to OSG during a run.

Run PostMortem: dagmanalyze.py

As a run proceeds, condor creates several log files indicating significant events in the run, such as jobs starting, completing, being evicted, and so on. Though the information is relatively complete, it is not organized in a useful way. In particular, we would like to be able to easily answer these questions:

1) How many concurrent processes are running or have run?
2) Did jobs spend more time running or in the queue?
3) Were jobs evenly balanced in run time?
4) Were there a significant number of failures?
5) Were failures concentrated at a particular site?
6) How were jobs distributed across sites?

To aid in providing answers to such bird's-eye view questions, we have created dagmanalyze.py, a simple python script that parses the dagman log file and produces several charts using the matplotlib library. The following images are those created by dagmanalyze from a run consisting of 300 jobs.

The overall contours of the run are shown in Figures 3 and 4. In figure 3, each job within the run is a horizontal line, with the horizontal axis indicating wall time. When the job enters the queue, the line begins in blue. When the job starts running, the line becomes red. If the job is sent to the queue again, the line becomes red. We can see immediately that there are several jobs that did not complete until well after the majority. Furthermore, we can see that these are cases where the job re-entered the queue at least once. Figure 4

displays similar information but aggregated across jobs. In figure 4, the blue line traces the number of jobs queued at any moment, while the red line shows the number of jobs running. In this figure, we can see the rate at which jobs move from queued to running, and the number of running processors that are sustained over time. In this case, we see that we can simultaneously run as many processors as we have asked for, suggesting that we could increase the number of processors on a subsequent run.

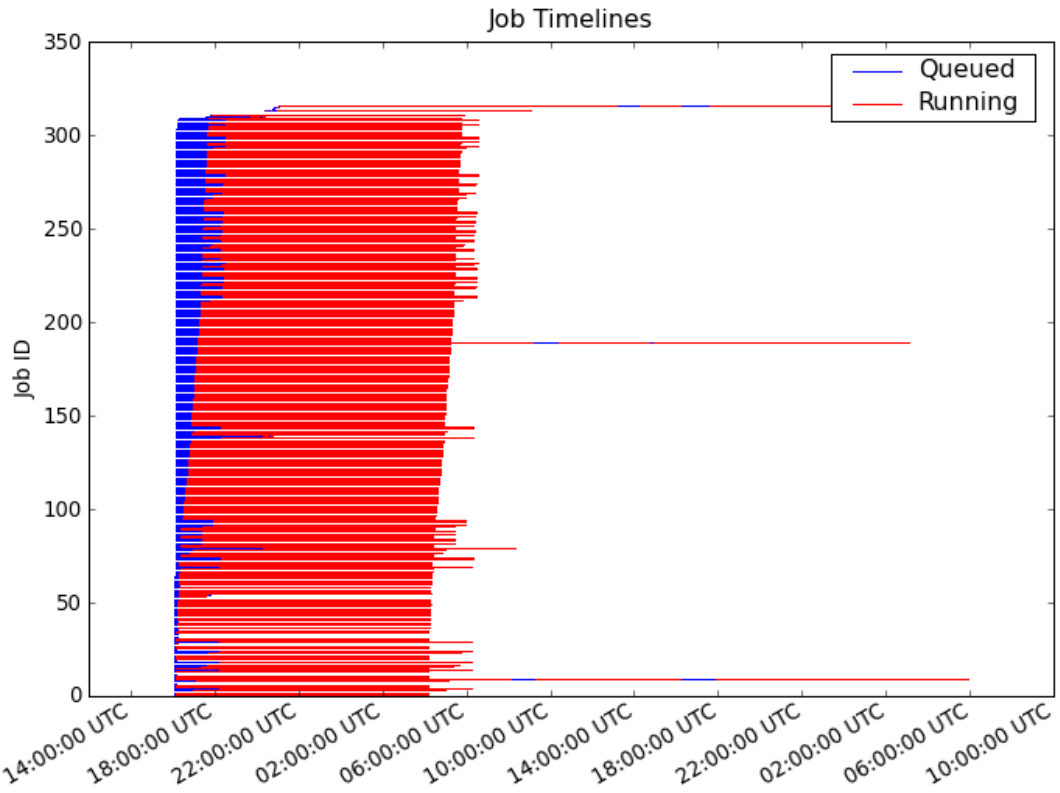


Figure 3: Job timelines for a run of 300 jobs.

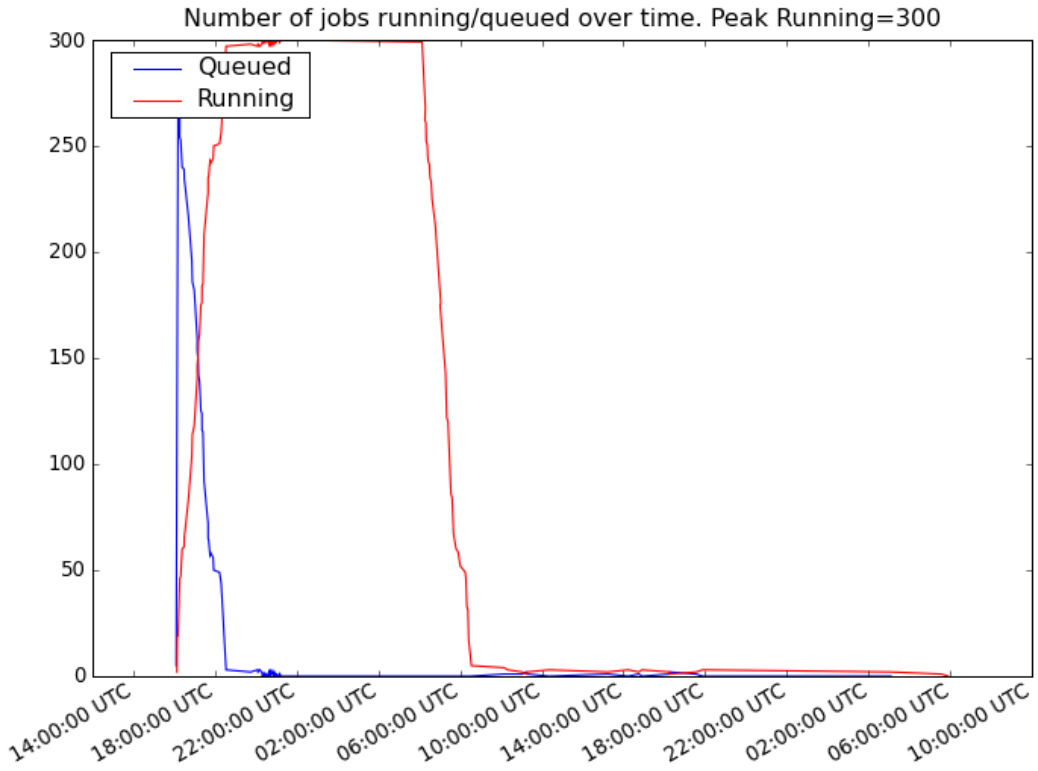


Figure 4: Queued and running jobs over time indicate scale up time and processor limitations.

Figure 5 displays histograms of run, queue, and wall times for the jobs in the run. Because the client is time-based, we can see a very tight distribution of run times. A distribution of queue times exists, but on the whole, the queue is adding about 1/7 of the total wall time per job.

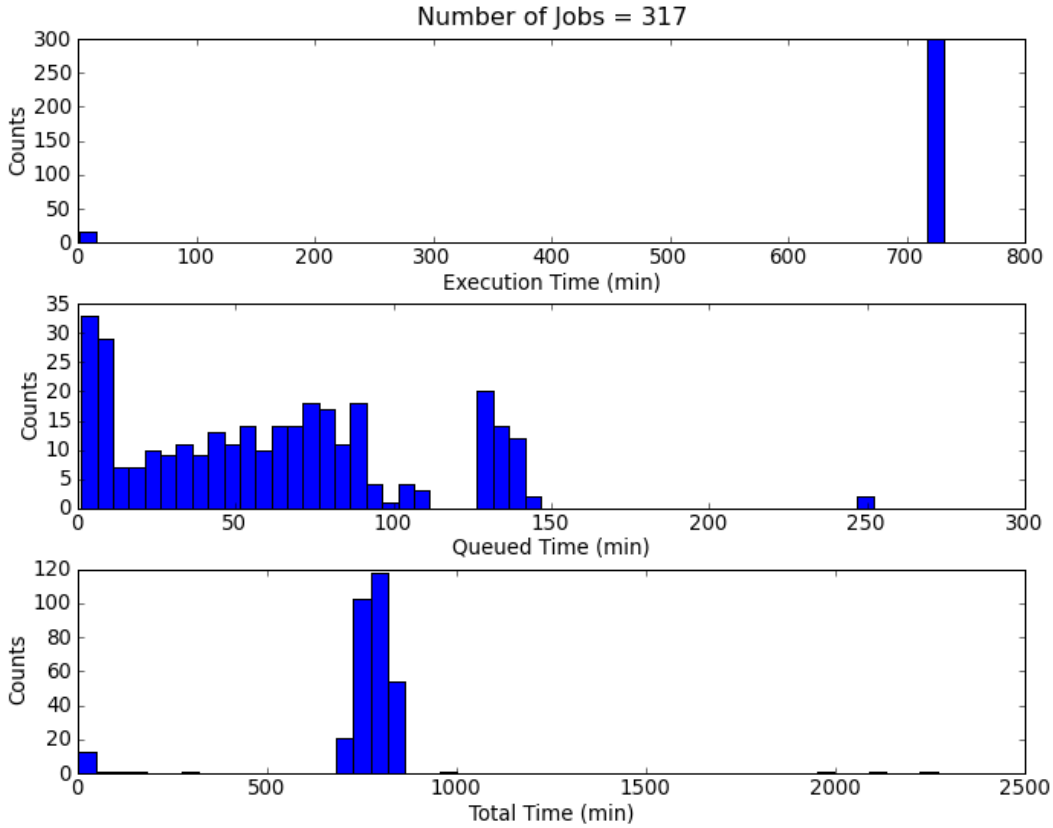
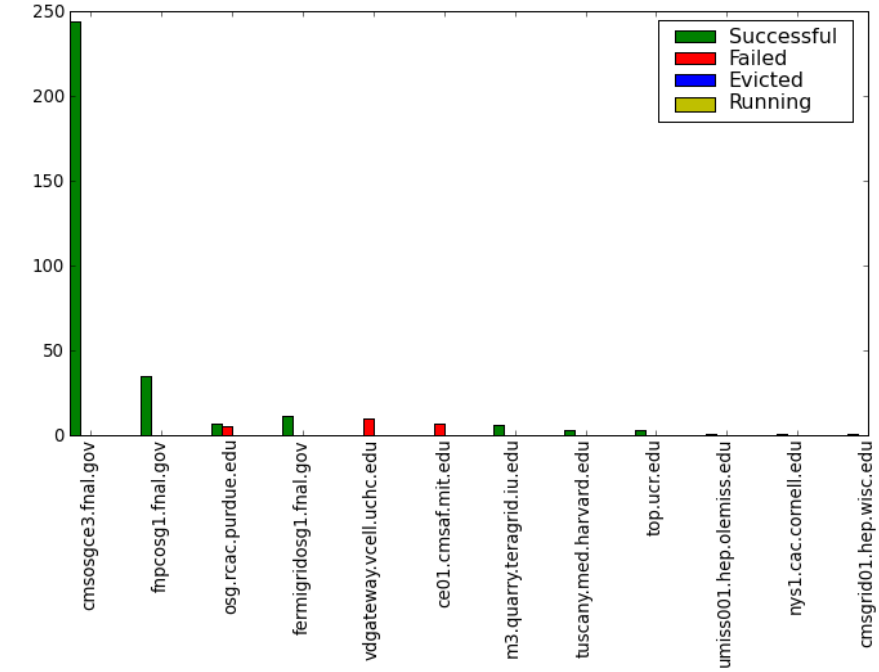
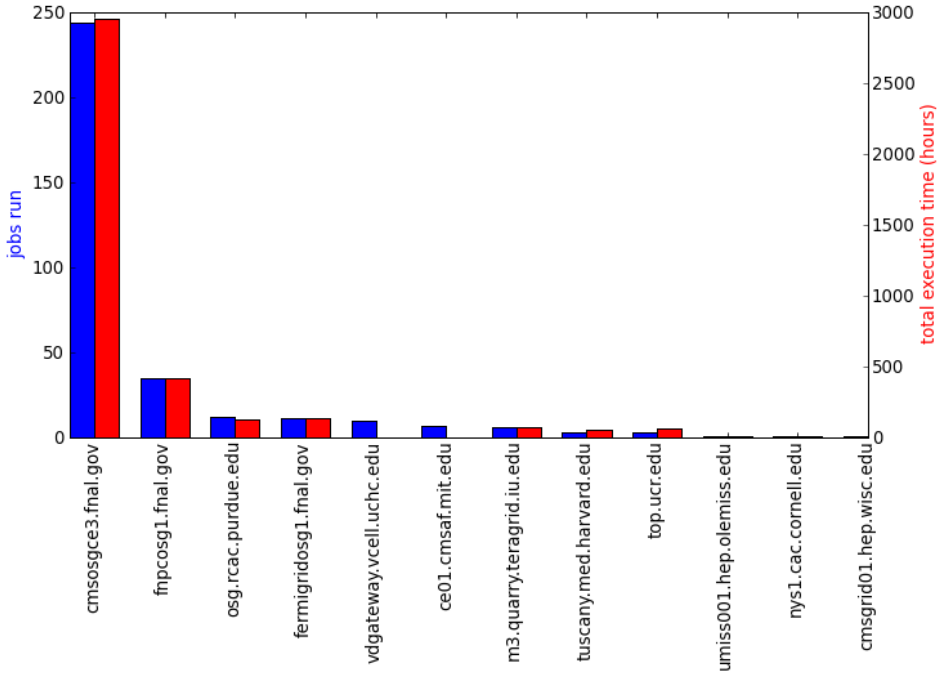


Figure 5: Histograms of run times, queue times, and wall times are used to diagnose workload balance and latency cost.

Figures 6 and 7 display the disposition of runs across different OSG sites, with slightly different focus. Figure 6 shows the outcomes of runs at each site, so that we can look for sites with high failure or eviction rates. In this case two sites have a 100% failure rate, indicating that there may be a problem, *e.g.* with memory limits. Figure 7 is geared towards determining which sites are contributing the most CPU time. For this run, the Fermilab sites are contributing the lion's share of the CPU hours.





Figures 6 and 7: Site dependent statistics are used to determine site-dependent problems and overall effectiveness.

Finally, figure 8 shows the number of times that each job must be run. In many cases, though not the present case, particular jobs will be associated with particular data. If there is a problem with one chunk of data, then that job will fail repeatedly. This histogram is a tool for helping find out whether there are jobs that are failing repeatedly. In this case, most jobs are requiring only a single run to complete successfully, with decreasing numbers needing to run more frequently.

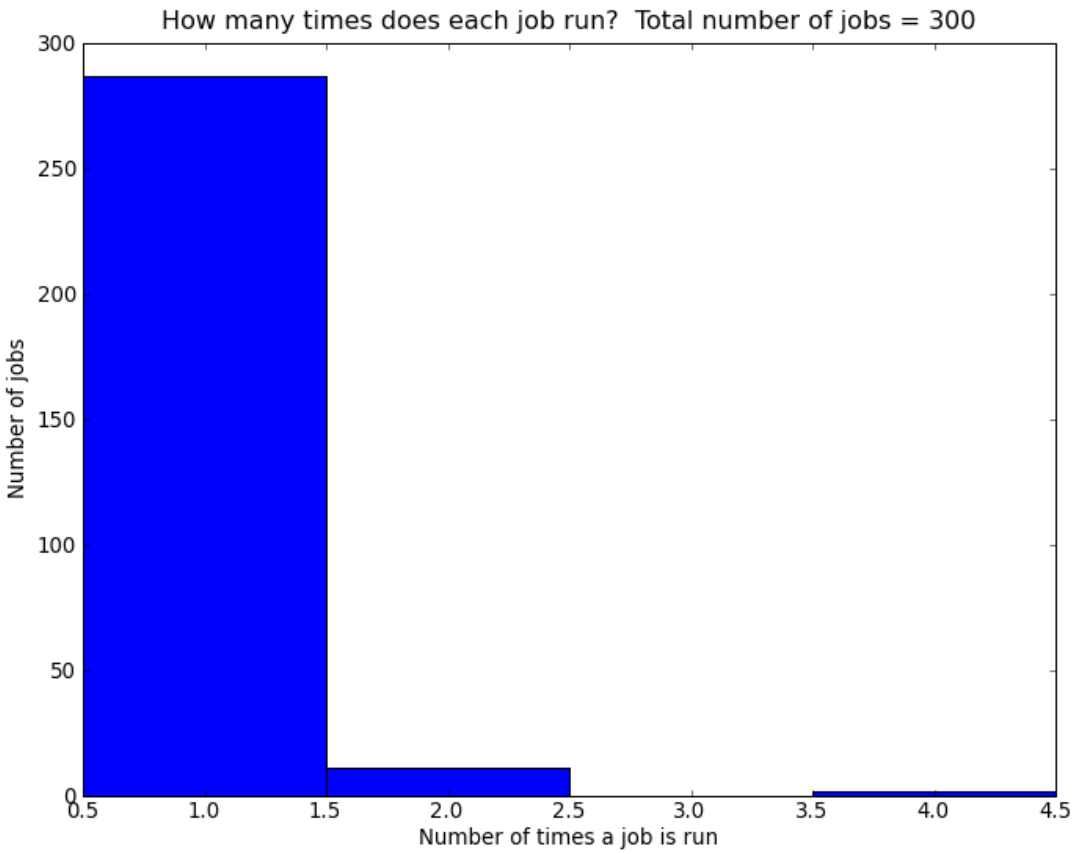


Figure 8: The number of jobs that must be multiply restarted indicates an extra computational cost, which can be further diagnosed and improved upon.

Limitations of the server/client approach

The client/server approach introduces a real bottleneck into the system via communication with the server, which must deal with many clients simultaneously. In the current implementation, this also becomes the overall work bottleneck. The limiting factor is the rate at which the central database management system (MySQL is being used here) can process the incoming data. Some develoment time had to be spent during this project on setting up an optimized infrastructure that can scale. At the present the following steps have been implemented in order to provide optimal I/O performance:

- MySQL contains a table that lists all alignment pairs to be computed. Once a pair has been calculated the row for this alignment is updated with summary scores that give a quick estimate if the similarity between the two proteins is significant.
- The detailed alignment files are stored on hard drive, that is physically independent of the one that contains the MySQL database files.
- Caching: SQL access is reduced by caching data in memory. E.g. the list of alignment that have not been calculated is kept in memory and a large batch of new alignments is loaded into memory when required.
- Batch processing: A job that is run on OSG sends back results back to the centralized server every 400 alignments (roughly every 15 min.). The SQL update statements are performed as a batch SQL update as well.
- Thread synchronization: In order to avoid too many parallel incoming connections from writing to the database and file system at the same time, all incoming requests are thread synchronized and only 8 parallel write/update operations are allowed at the same time. The nr 8 matches the number of available CPUs on the system.
- The MySQl query optimizer has been investigated to make sure the correct indexes are used for accessing the data. This allowed to identify a probably MySQL specific issue that has been affecting performance. If multiple indexes are available for a column, mysql arbitrarily selects one of them. This can affect select and update speeds, e.g. if multi column indexes should be used, but MySQL uses a one-column index instead. In order to avoid this, all one-column indexes have been removed from our schema, and all DB access is through the multi column indexes.

As mentioned ealier, with these optimisations in place, the central server is at the present capable of communicating efficiently with 800+ concurrent jobs. If 1000 jobs are being run in parallel occasional write bottlenecks can cause clients to have to queue, resulting in less efficient use of the available CPU.

A useful feature during the development of the server was a "kill switch" I.e. if a critical error happens on the central server that requires immediate stop of all calculations, it can send a kill signal to all jobs, resulting in an their termination. This was used during the initial development of the server to identify and remove components that had bad performance. The kill switch can also be used as an emergency break. If the queue of waiting clients gets too long, selected clients can get killed in order to ensure that the overall system is used efficiently and that clients only have to spend a minimal amount of time waiting for responses from the server.

Conclusion

We have described the use of a client/server architecture for managing large calculations in the open science grid. While the client/server approach offers benefits in terms of workload distribution, its main advantage is in workflow management. The main disadvantage is the bottleneck introduced through the centralized server. Some development time had to be invested in order to come up with a server that can hold up to the scale of calculations that are possible on the grid.

New users of a system such as the OSG are often distracted from their computation by the need to learn how that system manages workflow. In cases where the workflows are large or complicated, the workflow management itself can itself consume significant user time through the staging and managing of files.

In standard OSG usage, the user often works on the submit host rather than their home

machine, and manages workflows by managing files on disk; state of multi-run computations is managed implicitly by the presence of absence of files on disk.

In the client/server approach, workflow management is decoupled from processor provisioning. This allows the user to operate largely from their own computers, using the submit host only to request more processors. Significantly, computation state is managed actively; at any time the full state of the computation can be retrieved from the user-owned database.

Choices about the use of a client/server model in any given application must balance the advantages of workflow distribution and management with the disadvantages of increased bottlenecks and server development.