

Challenges in Large Scale Distributed Computing: Bioinformatics

Terry Disz, Mike Kubal, Robert Olson, Ross Overbeek, Rick Stevens

Argonne National Laboratory

The University of Chicago

The Fellowship For the Interpretation of Genomes

disz@mcs.anl.gov, olson@mcs.anl.gov, mkubal@mcs.anl.gov, ross@thefig.info,

stevens@mcs.anl.gov

Abstract

The amount of genomic data available for study is increasing [1] at a rate similar to that of Moore's Law [2]. This deluge of data is challenging bioinformaticians to develop newer, faster and better algorithms for analysis and examination of this data. The growing availability of large scale computing grids coupled with high-performance networking [3] is challenging computer scientists to develop better, faster methods of exploiting parallelism in these biological computations and deploying them across computing grids.

In this paper, we describe two computations that are required to be run frequently and which require large amounts of computing resource to complete in a reasonable time. The data for these computations are very large and the sequential computational time can exceed thousands of hours. We show the importance and relevance of these computations, the nature of the data and parallelism and we show how we are meeting the challenge of efficiently distributing and managing these computations in the SEED [4] project.

1. Problems in Modern Bioinformatics

1.1. Protein Structure

In Texas, folks put a lot of thought and effort into making good chili. So to win the “Annual Texas Chili Cook-Off” and the \$10,000 prize money is no small feat. One needs the right recipe, choicest ingredients, and proper cooking equipment. A few years back however, a guy won and he didn’t possess any of those things. One of the pleasures of attending the Texas Chili Cook-Off is sampling all the contestants’ efforts. This fellow visited each booth for a sample, but instead of savoring it, he saved it. And when he got back to his booth with his samples, he dumped them into his empty pot. It was not until he had been declared the winner that anyone realized he didn’t have any cooking equipment other than his one pot.

In the world of bioinformatics, a lot of folks are going to painstaking efforts to “make good chili”, and we should all be grateful to them. But sometimes, we get the best results by integrating the output from several of these efforts. The SCOPmap[5] program takes this approach. SCOPmap applies several sequence and structure comparison tools to a query protein and combines the results to find homologs, sometimes remote homologs, already in the SCOPmap database, and thus determine the structural classification of the protein and possible evolutionary links between families of proteins. The four sequence comparison tools used, listed in increasing order of sensitivity of remote homologs are BLAST[6], RPS-BLAST, PSI-BLAST[7], and COMPASS[8]. The structure comparison tools used are MAMMOTH[9] and DaliLite[10].

Since remote homologs can have similar amino acid residue conservation patterns, but lack overall sequence similarity SCOPmap contains two additional programs that look to identify remote homologs by conservation analysis which requires the output from both the structural and sequence comparison tools. SCOPmap was able to make the correct structural classifications for 81.6% of the non-trivial, those not able to be detected by gapped BLAST, protein domains tested.

Possessing the structural classifications for as many of the proteins in our database as possible will be a valuable resource in our efforts to accurately annotate protein functions. In addition to helping find proteins that may perform the same function as our query protein, despite a low degree of overall sequence similarity, the structural information will help us disambiguate protein families into more discrete sets. Having more well-defined protein

families will lead to better annotations with other annotation tools and approaches. Even after the function of a protein is determined, the structural classifications will aid researchers in determining how the protein actually performs its job.

The compute resources for any one of sequence or structure applications can be significant when applied to a large number of proteins, integrating them all together can make waiting for the results a project for three generations of bioinformaticians. Running SCOPmap on one of our machines (Macintosh G5) for just one protein out of the million in our database takes about an hour. To run SCOPmap for all of our proteins would take approximately 100 years on our single machine.

The good news is that the project of finding the SCOP classification for each protein in our database can be broken down into independent parallel jobs suitable for solving on a large computational grid.

1.2. Sequence Similarities

Recent analysis by Gordon Pusch of Fellowship for the Interpretation of Genomes forecasts that the 1000th microbial genome will become available sometime in 2008. For the field of comparative genome analysis this prospect is both daunting and mouth-watering. Since many similar genes, protein functions, cellular processes and structures are conserved across diverse sets of organisms, the greater the number of genomes included in the analysis, the more genes with previously unknown functions can be assigned a potential function. The number of similarities detected increases proportionally to the square of the number of genomes used [11]. According to a recent statistical model, we have not yet begun to “exhaust the power of comparative genome analysis,” and a significant amount of the protein sequence space still needs to be explored [12]. The benefits of possessing the similarities for as many genes as possible are not limited to determining the function of genes that are similar to others. Functionally related genes (genes that participate in the same process) often are clustered together in the DNA. In organism A, 3 out of 4 genes in a cluster may have similarities to a cluster of genes in organism B. The gene in organism A with no similarity to anything else has an unknown function. By comparing the genomic context (adjacency of genes, direction of transcription, fusion of two genes in one organism to one gene in another organism) of the gene clusters in organism A and organism B, it may be possible to map the function of

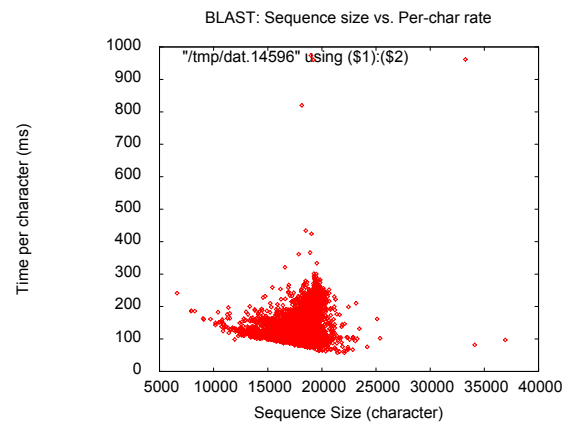
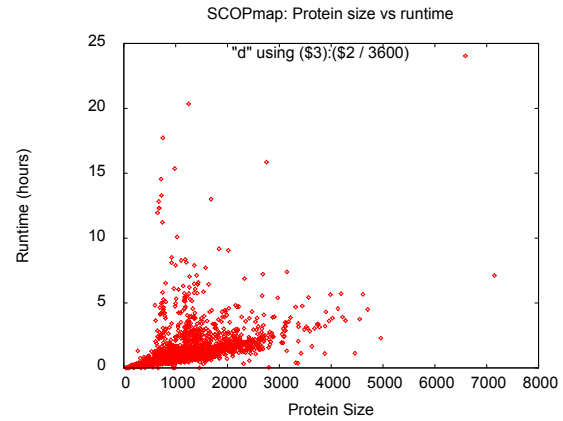
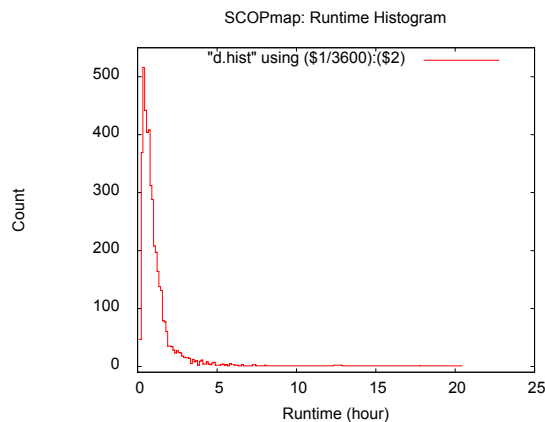
a gene in organism B to a gene in a cluster with no sequence similarity in organism A.

With all the benefits of similarities, it is no wonder that after each new genome is sequenced and has its genes called, the first thing everyone wants to know is how similar is every gene in this new genome to every gene in every other genome in the database, and how similar every gene in every organism in the database is to every gene in this new genome. In the SEED, an open-source framework for comparative analysis and genome annotation, the current set of similarities just described is about GB in size.

1.3. Characteristics of the problems

1.3.1. SCOPmap

The SCOPmap calculation, while itself rather complex, appears to the user as a simple shell script invocation. It takes as input a PDB file [13] containing the sequence data for the gene we are considering, and generates a file containing the structural classification information. The data transfer requirements for this are hence minimal, but the SCOPmap installation with its component databases is rather large, around seven gigabytes, and requires careful installation. An individual invocation of the SCOPmap program can take anywhere from a few minutes to many hours. In following figures we see a scatter plot and histogram relating the size of the input sequence (here, from genes in *Escherichia coli* K12) to the observed runtime. The majority of the runs are within a few hours of runtime, but there are a number of outliers requiring significantly long runtimes (see the following for a histogram of the runtime data); these outliers become significant when we consider approaches for monitoring long-term job execution.



1.3.2. Similarity Computations

The similarity computation is likewise simple to execute. Given a database of known sequence data (which has generally been curated to remove redundant sequences, retaining only the longest sequence containing zero or more shorter sequences which are suffixes of this longest sequence. We call this longest sequence the *principal synonym* for it and its suffixes. Our *nonredundant database*, or *NR*, contains just the principal synonyms. For a set of sequences on which we wish to compute similarities to sequences in the NR, we need only invoke the BLAST [6] application with the correct parameters. Due to the sheer volume of sequence data, this can be a hugely time-consuming task. Unlike the SCOPmap computation, the expected runtimes for the similarity computation can be fairly accurately predicted. On modern cluster nodes, we have measured BLAST runtimes of around 180 milliseconds per character of input sequence when comparing to a NR containing around 2.5 million sequences. This means that an initial computation of similarities of all 2.5 million sequences against each other could take nearly five years on a single processor.

2. Problem Solving Approach

The larger problem we now face is overcoming the challenges in attempting to solve a large SCOPmap or similarity calculation on a large computational grid.

These bioinformatics jobs can be viewed as many related but distinct computations. Related in that they use the same base data sets and the same code and that the output from them is collected together into a unit. Distinct in that each computation can proceed independent of any other computation (although there are optimizations to be had with the use of the common data sets). Some jobs can require tens of thousands of individual computations, the serial computation of which could take thousands of hours. While the time per computation is similar throughout the job, there are some significant exceptions as noted in the SCOPmap discussion.

Ideally, we would be able to send these jobs into a grid-based scheduler and have that scheduler work with the resource managers at each cluster to dispatch the individual computations as efficiently as possible, something along the lines of a Community Scheduler [14]. While there are proposals and experiments with such a facility [15], in general we are far from being able to do that on the clusters we are able to access.

Typically, the scheduler interface at the clusters we use is a variant of, or similar to the Portable Batch System [16], first developed in 1994 at NASA. This system is primarily a resource manager – it matches user resource requests against available resources and policy for an individual cluster and queues jobs accordingly. A resource request is basically the number of nodes required and for how long. The types of jobs usually run on these clusters are much smaller than what we are interested in; the 2004 mean for all jobs on our Jazz cluster was 10 nodes and 2.94 hours per request.

Several thousand jobs launched into a traditional batch scheduler will overrun the available resources ending in job failure. In addition, the sheer number of computations we must perform for a given job require that we schedule computations across multiple clusters. Therefore, the lack of a grid based scheduler across our target clusters requires that we develop our own methods for workflow management, job management and submission.

The usual job scheduling on these clusters is “push” based, i.e., the scheduler acts as a master controller, queuing and dispatching work to compute resources. The underlying model is of a job consisting of a closely related set of computations that start and finish together. If any one of the processes for a job fails, the entire job is considered to have failed. The user asks for more time than is

needed to avoid termination of the job before completion. While the user is only billed for the time actually used, one can’t help but regret the loss of the remainder of the available time if the user had more work to be done.

Our computations don’t follow this model; in fact, it would be quite costly to lose progress on all computations if any one of them failed. When a computation is completed, there is almost always more work that could be done in the remainder of the allocated time, if only one could get somehow obtain the unit of work.

Because of these job differences and the complexity in having a central scheduler maintain job and resource state across multiple clusters and the difficulty of trying to launch tens of thousands of computations in this manner, we choose to implement a simpler and more efficient model that is more in keeping with our requirements.

Self scheduling tasks have been the subject of research and writings for many years[17] and more recently utilized in bioinformatics computations[18]. The idea is to have computational processes ask for work from a “pool of work” when they are ready, rather than having the master scheduler ascertain their readiness state and then push work out to them. This simplifies the task of the owner of the work to one of only maintaining the state of the work to be done and not the state of the computational resources.

When we schedule a job onto a cluster, we are scheduling a “worker”, that works independent of any other workers on the cluster. Each worker is given the address of a server we call an “AskFor” [17] server which dispenses units of work upon request from a worker. A worker stays active over the entire allocated time, continually asking for and processing work, making better utilization of allocated time. Since a worker will not quit before the allocated time has run out, it is invariable that some work will not be completed. The server keeps track of work and will reschedule uncompleted work. If the size of allocated time to time per task is sufficiently large, this overhead is an acceptable price to pay for the higher utilization we get of our allocated time.

2.1. Resources

The resources that we are primarily targeting to solve these problems are loosely coupled clusters of computers. Since the computations are entirely independent of each other once dispatched, we do not rely upon the existence of a fast interconnect between the nodes in a cluster (apart from its effect on shared file system performance, which we will discuss later).

The first of our two primary resources is the Jazz cluster at the Laboratory Computing Resource Center [19] at Argonne National Laboratory. Jazz comprises 350 compute nodes, each with a 2.4 GHz Intel Xeon processor. Half the nodes have 2GB RAM, half have 1GB RAM. The cluster has 10TB of shared disk served as Global File System (GFS) [20] volumes, with another 10TB shared as Parallel Virtual File system (PVFS) [21] volumes.

The heavy lifting in our computations is provided by the second of our resources, the TeraGrid [3] [22] distributed cluster. The TeraGrid is a large distributed cluster spread over nine sites across the United States interconnected via a 40 gigabit per second network. For the purposes of our work, we consider each of the TeraGrid sites as a different cluster; the integration of them into an overall distributed cluster does not affect the technology used in the application.

The TeraGrid cluster located at Argonne National Laboratory is typical of the TeraGrid sites. It comprises 62 nodes with dual 1.3 GHz Intel Itanium 2 (IA64) processors and 4GB RAM, and 96 nodes with dual Intel Xeon 2.4GHz processors and 4GB RAM. The cluster has 15 TB General Parallel File system (GPFS) shared disk, and 5 TB PVFS shared disk. For the experiments described here, we only used the IA64 nodes.

2.2. Our implementation

Our implementation directly models the pool-of-work abstraction. Each job is divided into a number of explicitly defined, independent work units. A central broker, the AskFor [17] server, manages the dispatching of these work units to workers; monitors and updates a record of the state of the work units; and handles the collection of output data upon completion of work units.

We have built three distinct systems implementing this model as we gained experience with it in real-life operation.

The first version (MultiBlast) is dedicated to the large-scale computation of protein sequence similarities via the BLAST sequence comparison tool. It implements a simple self-scheduling worker infrastructure. The manager is a persistent server implemented in the Python scripting language. It maintains the work unit state in memory, writing snapshots of that state to disk for use in later restarts of the manager. The worker programs are each implemented in Python, and maintain a persistent TCP connection to the server for use in obtaining new work units and in writing results back to the server. The workers use the blastall program to compute similarities between the input sequences as

distributed by the manager and a nonredundant database that was manually configured.

The second system is dedicated to the SCOPmap computation. The worker/manager communication no longer relies upon persistent TCP connections, but rather uses XMLRPC messages directed from the workers to the manager. The entire infrastructure is asynchronous: the manager state is no longer kept in memory, but rather in tables in a relational database. The manager itself is not a persistent program, but implemented as a set of XMLRPC message handlers hosted by an Apache web server that act upon the state kept in the database.

Each request from a worker is of the form *get_work(job_name, worker_id)*. The manager maintains multiple distinct jobs; each worker picks the job from which it wishes to obtain work. The worker identifier passed to the manager is used to match the results for a particular piece of work to the worker responsible for its computation.

The third system is intended to be a more general-purpose tool, suitable at the least for both the BLAST and SCOPmap computations. It is also constructed to use web service messaging between workers and the manager (via SOAP instead of XMLRPC in this case). The manager again maintains state in a relational database, and is hosted in an Apache web server. The database schema is considerably more complex than that used in the SCOPmap system, largely due to the support for file staging.

3. Challenges

3.1. File Staging and Cluster Optimizations

One of the obstacles to the use of the MultiBLAST system as originally built is the requirement that the search database be manually installed on each cluster that is to participate in the computation. We address this in the latest software by incorporating the concept of *cluster work*. A computational job can specify that there are pieces of work that must be done once per cluster, and that this work must be completed before *noncluster* work can proceed. In the similarities computation, we define a cluster work item for each job that causes the search database to be downloaded to shared file system space in the cluster, and the BLAST *formatdb* indexing program to be invoked on it.

For such a mechanism to work, the manager must know the cluster affiliation for each worker. We currently require a manual registration of each cluster with the manager, which results in the assignment of

a cluster identifier. When each worker starts running, it registers with the manager, providing it with its cluster identifier.

3.2. Network considerations

The manager/worker architecture clearly requires a reliable network between the components. It must have high enough bandwidth in order to allow the downloads of any large database files required to succeed in a reasonable amount of time, and be able to sustain the large flow of connections from the workers to the manager. Note, however, that the system is resilient to transient failures of the network since no persistent connections are required. If these transient failures persist longer than the interval the manager uses to time out inactive workers, however, it is possible for completed work to be thrown away by the manager because it comes from workers declared out of commission. This can be solved by analysis of failure patterns in the worker/manager communications and by adjusting the dead-worker timeout if necessary (concurrently with working with the network administration staff to determine the cause of failure of the network, of course).

The reliance on workers' ability to connect to the manager also requires one to consider the effect of firewalls or other network security measures. Clearly, the manager must be accessible to the worker hosts; either by being located outside any firewalls or by having conduits to it installed in any firewall between the manager and the workers. The design of the system does not require individual workers to be open to incoming connections; all communication is initiated by workers. We have encountered large clusters in the TeraGrid system that block outgoing access from compute nodes; we are currently working with the site administrators to solve this problem.

3.3. Managing complexity

Conceptually, each of the distributed computation management systems is quite simple. A manager maintains state of work units, and workers are delivered work upon request, compute the results for that work, and return the results.

In operation, however, the simplicity vanishes in a snowstorm of details. We will discuss the challenges that we faced with each system, and how their resolution led to succeeding design choices.

The first MultiBLAST system has on the whole been quite reliable; we have used it for production similarity computation in support of the SEED[4]

genome annotation system since 2003. The primary limitation that it has is that its use is largely manual: the user is responsible for staging the nonredundant (NR) database onto each cluster that participates in the computation and must ensure that the invocation of the worker programs on the cluster have the correct path to the NR database.

The use of the persistent worker/manager TCP connections is also a potential source of trouble. In a very large computation, it is possible that resource limits may be exceeded on the server, in terms of the number of active connections supported by the operating system. In addition, if a connection is disrupted for any reason (perhaps a glitch in a wide area network), the client will interpret that disruption as a signal to exit.

The persistent connection does however provide a key advantage: the manager can know immediately if a worker has terminated for some reason and that the work unit the worker was processing has to be returned to the pool of available work. Without this immediate feedback, the manager must rely upon periodic updates from the workers to determine when a worker has terminated unexpectedly. Note that even with the persistent connections, it is possible for a worker to hang in a way that the connection is not dropped: for complete reliability, the manager must have a mechanism to determine the state of workers.

The SCOPmap computation system was built without such a mechanism under the assumption that for the limited scope of the problem being addressed that it would not be an important issue. In practice, however, it turned out that some of the pieces of work took upwards of 20 hours to execute (the standard timeslot we used in requesting node time from the system schedulers). Without manually logging into the compute nodes and checking the system load, it was impossible to tell if a client was dead, hung, or processing a very long computation.

We have addressed this issue in the new system by introducing worker heartbeats. Each time any web service method is invoked on behalf of a worker, the database record corresponding to that worker is updated with the current timestamp. A "heartbeat" method is also defined which has the sole effect of updating the timestamp. While a long-running computation is executing, the heartbeat method is periodically invoked (currently, on a 5-minute basis). Given this information, the manager can determine which worker records correspond to workers that are no longer active. If there are pieces of work which are marked as currently being computed on these workers they can be returned to the pool of available work.

Even with the mechanism in place to detect defunct workers, the worker code is designed to detect failure wherever possible and proactively notify the manager when a piece of work must be aborted. To this end, the worker code carefully manages signal handling, detecting the occurrence of fatal signals and invoking the work failed web service method for the current piece of work. In order to maintain as much order as possible, signals are masked during the execution of the web service interactions with the manager; any signals that arrive during the masked time are deferred by the application until the point at which signals are re-enabled.

3.4. Server scaling issues

We need this distributed computation infrastructure to support very large computations. For instance, a computation being used in the development of a major SEED release requires the computation of pair wise similarities between roughly 2.4 million sequences. Using a conservative estimate of 180 milliseconds per character of input sequence (this estimate derives from observing a large number of executions of work units from this computation), a sequential execution of the problem would take roughly 4.8 years. With the resources we appear to be able to realistically obtain on the combination of the Jazz and TeraGrid clusters, we should be able to complete this computation in under two weeks.

Doing so, however, requires managing a large number of individual work units. In order to strike a balance between managing scheduling and process startup overhead (which argues for large pieces of work), and the real-life limitations of working with production batch scheduling systems where one cannot assume infinitely long job execution (which argues for small pieces of work), we are currently dividing the work into roughly 20,000 character blocks, which translates to around 30-60 sequences per work unit, taking roughly 30-90 minutes to execute, depending on the actual size of the work unit and the capability of the node in use. Thus, the manager has to keep track of the current state of over 42,000 individual pieces of work, the input sequence corresponding to that data, and the output similarities.

We have not attempted to be overly clever in this system, and apply brute force methods to the solution. We use a well-connected Linux server as the manager. It is a powerful machine, with four Intel Xeon 2.8GHz processors and 4GB of RAM. The database is hosted in a Postgres sql database server configured with a large number of shared buffers available. In order to avoid any database table

consistency problems, we completely sequentialize all access to the database via a POSIX semaphore in the SOAP service code. This has the effect of potentially increasing the latency of requests; however, the attempts we made to portably use native database transaction and locking methodologies have not yet succeeded. In practice, our choice of work unit size and the fact that there is typically little contention for the database lock result in this not being a significant problem.

4. Results and Future Work

We have implemented three different versions of an AskFor-based manager-worker infrastructure. The original MultiBLAST has been used for 18 months as a production code in support of computing sequence similarities for the SEED project. We regularly run the MultiBLAST across the Jazz cluster at Argonne and the University of Chicago TeraGrid cluster, and have consumed over 32,000 node-hours on the Jazz cluster alone.

We have used the AskFor-based SCOPmap computation engine to compute SCOPmap protein structure information for *E. Coli* K12 on the Jazz and UC TeraGrid clusters. This computation took roughly 3.5 days on an aggregate of 196 nodes across the two clusters. The corresponding serial computation would have taken over 176 days.

We are in the process of computing a large BLAST similarity run on a collection of clusters including Jazz, TeraGrid clusters at UC, SDSC and NCSA, and the new Teraport cluster also at the University of Chicago. This job comprises over 48,000 individual pieces of work, and is running smoothly on the newest MultiBLAST infrastructure.

We believe the results of these three systems, including production and experimental usage, validate the AskFor model for use in high-volume distributed computations.

In the future, we envision such as system to be an efficient back-end processing engine for novel interactive applications in computational biology and bioinformatics. A centralized AskFor manager which buffers a queue of work units for presentation to a large-scale distributed computational engine is a useful component in the decoupling of the complexities of large distributed computers from the smaller-scale, more serial in nature biology applications. We envision a number of applications of this technology, including

- Allowing users to quickly compute similarity values for private or newly-

sequenced genomic data for use in annotation and other analysis.

- Providing high-capacity processing access to simple scripting language-based applications through the use of web service interfaces to the AskFor manager.
- Powerful bioinformatics portals that expose sophisticated analysis modules to users without detailed knowledge of the computing systems providing the back-end processing.

5. Acknowledgements.

We gratefully acknowledge use of "Jazz," a 350-node computing cluster operated by the Mathematics and Computer Science Division at Argonne National Laboratory as part of its Laboratory Computing Resource Center.

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W-31-109-ENG-38.

The submitted manuscript has been created by the University of Chicago as Operator of Argonne National Laboratory ("Argonne") under Contract No. W-31-109-ENG-38 with the U.S. Department of Energy. The U.S. Government retains for itself, and others acting on its behalf, a paid-up, nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.

References:

1. Genbank. 2005<http://www.ncbi.nih.gov/Genbank/genbankstats.html>.
2. Apweiler, R., et al., *Managing core resources for genomics and proteomics*. Pharmacogenomics, 2003. **4**(3): p. 343-50.
3. *The TeraGrid Project*. 2004www.teragrid.org.
4. Overbeek, R., Stevens, R., Disz, T., *The SEED: A Peer to Peer Environment for Genome Annotation*. CACM, 2004. **47**(11): p. 46-50.
5. Cheek, S., et al., *4SCOPmap: automated assignment of protein structures to evolutionary superfamilies*. BMC Bioinformatics, 2004. **5**(1): p. 197.
6. Altschul, S.F., et al., *Basic local alignment search tool*. J Mol Biol, 1990. **215**(3): p. 403-10.
7. Altschul, S.F., et al., *Gapped BLAST and PSI-BLAST: a new generation of protein database search programs*. Nucleic Acids Res, 1997. **25**(17): p. 3389-402.
8. Sadreyev R, G., N, *COMPASS: a tool for comparison of multiple protein alignments with assessment of statistical significance*. J Mol Biol, 2003. **326**: p. 317-336.
9. Ortiz AR, S.C., Olmea O, *MAMMOTH (matching molecular model comparison obtained from theory): an automated method for model comparison*. Protein Sci, 2002. **11**: p. 2606-2621.
10. Holm L, S.C., *Dali: a network tool for protein structure comparison*. Trends Biochem Sci, 1995. **20**: p. 478-480.
11. Overbeek, R., et al., *The use of gene clusters to infer functional coupling*. Proc Natl Acad Sci U S A, 1999. **96**(6): p. 2896-901.
12. Eddy, S., *A Model of the Statistical Power of Comparative Genome Sequence Analysis*. PLoS Biol, 2005. **3**(1): p. e10.
13. Berman, H.M., et al., *The Protein Data Bank*. Nucleic Acids Res, 2000. **28**(1): p. 235-42.
14. *Community Scheduler Framework*. 2005<http://sourceforge.net/projects/gcsf/>.
15. Mausolf, J., *Use Community Scheduler Framework to implement grid meta-schedulers*. 2004<http://www->

- 106.ibm.com/developerworks/grid/library/gr-meta.html.
16. Koonin, E.V., Galperin, M.Y., *SEQUENCE - EVOLUTION - FUNCTION. Computational Approaches in Comparative Genomics*. 2002, Boston: Kluwer Academic Publishers. 488.
 17. E. Lusk, R.O., et. al., *Portable Programs for Parallel Processors*. 1987: Holt, Rinehart, and Winston.
 18. A.J. Chakravarti, G.B., M. Lauria, *Self-Organizing Scheduling on the Organic Grid*. International Journal on High-Performance Computing Applications, 2005.
 19. Overbeek, R., M. Fonstein, M. D'Souza, G. D. Pusch, and N. Maltsev, *Use of contiguity on the chromosome to predict functional coupling*. In *Silico Biol*, 1998. **2**(1): p. 93-108.
 20. Osterman, A.O.R., *Missing genes in metabolic pathways: A comparative genomics approach*. *Current Opin. Chem. Biol.*, 2003. **7**: p. 1-14.
 21. Gaasterland, T., and Selkov, E., *Reconstruction of metabolic networks using incomplete information*. *ISMB*, 1995. **3**: p. 127-135.
 22. Mervis, J., *Advanced computing. NSF launches teragrid for academic research*. *Science*, 2001, 2001. **293**(5533): p. 1235-7.