ACADEMIC PRESS

Available at www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT®

J. Parallel Distrib. Comput. 63 (2003) 578–589

Journal of Parallel and Distributed Computing

http://www.elsevier.com/locate/jpdc

Necessity is the mother of invention: a simple grid computing system using commodity tools

Daniel S. Myers[a,b] and Michael P. Cummings[b,*]

[a] Department of Mathematics and Computer Science, Pomona College, Claremont, CA 91711-7004, USA
[b] The Josephine Bay Paul Center for Comparative Molecular Biology and Evolution, Marine Biological Laboratory, Woods Hole, MA 02543-1015, USA

Received 8 April 2001; revised 8 November 2001; accepted 8 November 2002

Abstract

Access to sufficient resources is a barrier to scientific progress for many researchers facing large computational problems. Gaining access to large-scale resources (i.e., university-wide or federally supported computer centers) can be difficult, given their limited availability, particular architectures, and request/review/approval cycles. Simultaneously, researchers often find themselves with access to workstations and older clusters overlooked by their owners in favor of newer hardware. Software to tie these resources into a coherent Grid, however, has been problematic. Here, we describe our experiences building a Grid computing system to conduct a large-scale simulation study using "borrowed" computing resources distributed over a wide area. Using standard software components, we have produced a Grid computing system capable of coupling several hundred processors spanning multiple continents and administrative domains. We believe that this system fills an important niche between a closely coupled local system and a heavyweight, highly customized wide area system.

Keywords: Apache; Distributed computing; Grid computing; HTTP; Java; Linux; Perl; SQL; UNIX; XML-RPC

1. Introduction

Access to sufficient resources is a significant barrier to scientific progress for many researchers facing large computational problems. To fulfill their computational requirements, researchers can attempt to gain access to large-scale computational resources. Often, these resources are housed in university-wide or federally supported computer centers. These centers were established, in part, with the aim of providing such resources to a broad range of users. However, there are significant barriers to the use of these large-scale resources, including ignorance of their availability and proper use, their poor fit to certain problems of interest, and the long time requirements of their application/review/approval cycles. At the same time, many researchers find themselves with access to numerous individual workstations and older clusters (Beowulf [2] and other types) that are typically overlooked and forgotten by their owners and administrators in favor of newer and faster hardware. Here, we describe our experiences building a Grid computing system to conduct a large-scale simulation study using "borrowed" computing resources distributed over a wide area. The system was motivated primarily by our need for flexibility; we found ourselves in need of an adaptable, light-weight system to take advantage of diverse and scattered resources. In order to increase the robustness of the system and to speed its development, we used existing software wherever possible.

One prerequisite for making use of borrowed computing assets, and for maintaining cordial relationships with colleagues, is accepting the computing assets "as is" and requiring minimal assistance from those from whom the assets are obtained. Many suitable computing systems are informally administered, and often administration duties fall on people with other primary responsibilities. An alternative approach of engineering our application to work in commercial systems such as distributed.net [30], Entropia [31], or United Devices [32] was not practical for our application or our budget. Another possible alternative, use of the Globus Toolkit [9] or a

*Corresponding author.
E-mail addresses: dmyers@pomona.edu (D.S. Myers), mike@mbl.edu (M.P. Cummings).

0743-7315/03/$ - see front matter
doi:10.1016/S0743-7315(03)00004-2

similar system, required greater control of computational resources than was available to us; when asking for access to informally administered resources, there is a major difference between asking, "May I have an account on your system?", and, "May I have an account on your system, and will you install and test the Globus Toolkit?" The computer resource niche we are striving to occupy is somewhere in between a single or functionally uniform domain with respect to administration and Grid capabilities, and a parasitic computing model [1].

Here, we describe a lightweight Grid system to integrate heterogeneous computing resources that may be administratively and geographically disparate. Resources used for an initial project ranged from a Linux laptop in France to a large Beowulf cluster in Utah. We have successfully used the system to complete a large-scale simulation study that addresses a specific problem in phylogenetic analysis. While some of the presentation here is in the context of a specific simulation problem, the Grid system we describe is general and can be applied to a broad range of problem types. We begin by establishing the context with a brief description of the scientific problem motivating the endeavor. We then describe the Grid system we developed to satisfy the computational needs of our problem in context of basic features common to most Grid systems. Lastly, we briefly discuss performance considerations and possible future work.

2. Scientific context

Phylogenetic analysis seeks to infer the relationships among organisms based on their characteristics (e.g., DNA sequences, morphology). One of the most important developments in phylogenetic analysis in the last decade has been the application of Bayesian methods, and in particular, Markov Chain Monte Carlo methods [14], to infer relationships. One of the posited advantages of Bayesian methods in phylogenetic analysis is that posterior probability values for relationships are generated as a consequence of the analysis. These posterior probability values provide a measure associated with each node in a phylogenetic tree that is interpreted as the support of the data for that particular relationship. In maximum likelihood or other competing methods of phylogenetic analysis, a corresponding measure of the support of the data for particular relationships is the proportion of bootstrap replicates supporting a node. The bootstrap is a well-known statistical technique [7], one of a number that have developed in the context of computational statistics, and was first applied to phylogenetic inference by Felsenstein [8]. The bootstrap requires re-sampling the original data with replacement to generate pseudo-samples, and these in turn are used to re-estimate statistics of interest. Summarizing these re-estimates yields information, in this case a measure of support for particular phylogenetic relationships.

An open and important question in phylogenetic analysis is the exact relationship of bootstrap values to posterior probability values. Unfortunately, an analytical solution is not readily apparent. Furthermore, while the theory of each measure is largely independent it has been posited that they should be fully equivalent [6]. However, very limited empirical observations suggest that these statistics may differ [13,18]. Therefore, a need exists for a thorough and well-conceived study comparing the behavior and relationship of the bootstrap and posterior probability measures.

To examine this problem we conducted a simulation study over a well-known model space, the four-taxon model, which has been used by others to investigate properties of different methods of phylogenetic analysis [10,11]. A four-taxon tree has five branches (four external branches and one internal branch). This model space can be viewed as a two-dimensional matrix with the abscissa representing the lengths of three branches (two external and the internal) and the ordinate representing the lengths of two branches (the other two external). Branch lengths are proportional to the expected difference accumulated along the branch. Following Gaut and Lewis [10] we used branch lengths ranging from 0.02 to 0.74, which in 0.02 increments produces a matrix of 37^2 ($= 1369$) elements (Fig. 1).

For each element in the model space we simulated four DNA sequences to give branch lengths corresponding to the parameters of the four-taxon tree model space. Each sequence was 1000 nucleotides in length and followed a realistic model of nucleotide substitutions. These simulated DNA sequences were made using a slightly modified version of the program evolver, part of the PAML package [19]. Each simulated sequence was then used as data to infer the phylogenetic relationships using the two methods: maximum likelihood with the bootstrap using the program PAUP* [17], and Bayesian analysis using the program MrBayes [12]. The observations of interest for maximum likelihood were the proportions of bootstrap replicates (out of 2000) supporting each of the three possible fully resolved topologies (one correct, two incorrect), and the observations of interest for the Bayesian analysis were the posterior probabilities associated with each topology calculated from a sample of 2000 (of 50,000) Markov Chain Monte Carlo steps.

The number of replicates (sets of four simulated DNA sequences) for each of the two analysis types (maximum likelihood/bootstrap and Bayesian/Markov Chain Monte Carlo) in each element of the model space was 1000, a number chosen to achieve reduced variance and desired levels of power in subsequent statistical testing.

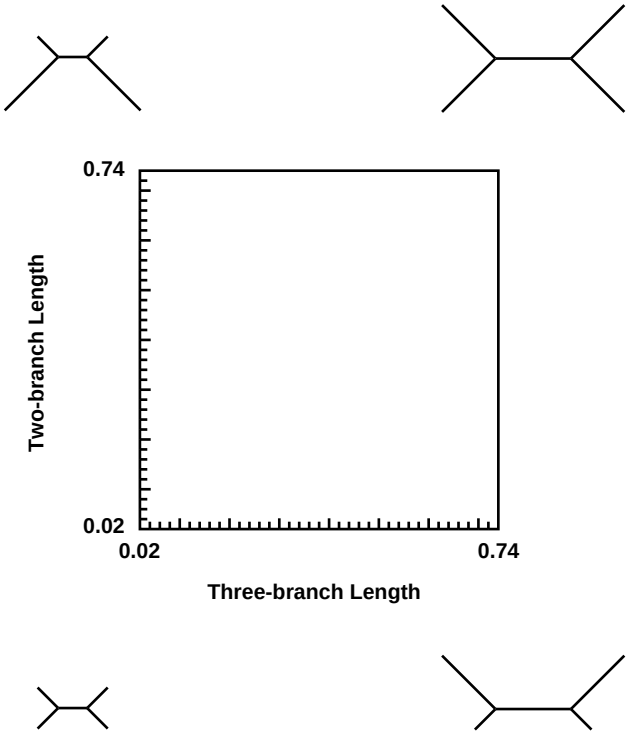

Fig. 1. The analytical model space presented as a two-dimensional matrix. The abscissa represents the lengths of three branches (two external and the internal) and the ordinate represents the lengths of two branches (the other two external). Branch lengths are proportional to the expected difference accumulated along the branch. Branch lengths on trees shown are not to scale.

Therefore, the total number of analyses completed was 2.738×10^6 ($= 1369$ model elements $\times 1000$ replicates $\times 2$ analysis types), ignoring the individual bootstrap replicates and Markov Chain Monte Carlo steps for each set of simulated sequences. As each analysis in the problem space was wholly independent of any others, parallelization was an obvious computational model with atomization to the level of individual maximum likelihood or Bayesian analysis of each set of simulated sequences. Indeed, this simulation falls within the class of problems referred to as embarrassingly parallel. Some previous large-scale computer-based studies in phylogenetics [4,5,15] have used Massively Parallel Processing systems such as multiple Thinking Machines Corporation Connection Machine CM-5 systems. However, due to a combination of factors including a lack of access to large-scale, high-performance computational resources; budget limitations; and time constraints, we made the decision to develop our own Grid system to take advantage of resources available to us. The results of the simulation to compare bootstrap and posterior probability values are more fully described in another paper [3].

Table 1
Computer architectures (processor family) and operating systems meeting the system constraints

Architecture	Operating System
Alpha	Linux, Tru64 UNIX
SPARC	Solaris
UltraSPARC32	Solaris
UltraSPARC64	Solaris
SGI_n32	IRIX
SGI_64	IRIX
x86	Linux, FreeBSD
RS/6000	AIX
IA-64	Linux

3. Implementation

3.1. System constraints

For this simulation study, the range of computing system architecture and operating system combinations was dictated by one application, PAUP* [17], which is used for the maximum likelihood/bootstrap analyses. PAUP* is not available as source code, and hence we were constrained to those architecture and operating system combinations for which PAUP* binaries are available. Furthermore, to simplify system design we restricted ourselves to versions of UNIX and similar operating systems (e.g., Linux), for reasons explained in more detail later. Even subject to these two constraints, several computer architecture and operating system combinations were suitable for use in project (Table 1).

3.2. General design of the grid system

The topology of the Grid is a loose star (hub and spokes) structure. The center of the star (hub), is the coordinating server for the project, and the points of the star (the ends of the spokes), are the remote machines running as clients. In the initial study, we used HyperText Transfer Protocol (HTTP) for client-server communication, a relational database management system (RDBMS; specifically MySQL) on the server to store work to dispatch and results received, and Perl to tie the components together. A new version of the software, developed after completion of the initial project, largely replaces HTTP with eXtensible Markup Language-Remote Procedure Call (XML-RPC) [29], and abstracts the central database behind a Java [21] interface; any data structure capable of implementing the required interface may thus replace MySQL.

3.3. System requirements

3.3.1. Operating system

As previously mentioned, we made the decision to restrict computers in our Grid to those using UNIX or

UNIX-like operating systems (e.g., Linux). This decision was motivated primarily by our desire for simplicity; writing for one target reduced programmer workload, and UNIX-like operating systems provide feature sets that best complement our design. For example, they have robust tool sets (e.g., native tar and gzip), are the native environment for Perl, and have robust multitasking facilities, including support for process priority levels. This latter feature (embodied by `nice`) allows us to courteously make use of computing resources by running our processes at a much lower priority than those of other users. While we have restricted our Grid to computers using UNIX or UNIX-like operating systems, the general features of our design can accommodate computers using other operating systems as long as the necessary functionality is met either by the operating system itself or through Grid-supplied capabilities.

Use of Perl as the backbone of the client side of the Grid provided several advantages. First, in the spirit of ad-hoc Grid computing, it allowed us to rapidly prototype and deploy the system for actual use. Second, Perl has excellent support for interfacing with external binaries, including easy access to functions such as fork and exec and superb regular expression features for processing of program output. Third, Perl has a robust library for HTTP communication, LWP (libwww-perl) [26], which allowed us to avoid writing our own implementation. Fourth, Perl is able to easily install add-on modules and libraries in directories outside of the main system repository; this allowed us to easily add missing modules using only the privileges of a local user. Finally, and, most importantly, Perl provides a portable and uniform interface to host systems; Perl developers have addressed the subtle-but-important differences between versions of UNIX-like operating systems, thus making it possible to run the framework code on a large range of platforms. The analytical programs specific to our application (evolver, MrBayes and PAUP*) are all compiled C, and thus the use of Perl for the framework did not have a significant impact on the speed at which the clients could perform analyses. While we chose Perl for these reasons, any language that provided an HTTP library and the other features described above would be suitable.

3.3.2. Client

The only requirements for a client are a UNIX-like operating system, Perl, and a collection of Perl library modules (LWP and its dependencies). These minimal requirements contribute to the flexibility of our system: only a normal user account is required, and any user provided with an appropriate client script and authentication may participate in the Grid. The role of client system administrators is thus restricted to at most providing us with a local account, in the event that no local user is available to run the client for us.

3.3.3. Server

Requirements on the server side are more numerous, but since we have full control over the server we may install software as desired. For the version of the software used in the initial study, the requirements were as follows. First, an Structured Query Language (SQL) database supported by the Perl DBI [24] (library for database access) is assumed to be available. Second, an Apache/mod_perl Web server [20,23] is assumed to be installed (mod_perl is a module for the Apache server that allows the use of Perl to modify and extend the server). Administrative control over the Web server is required, as the HTML::Mason Perl module [27] for embedding Perl code into Web pages and a custom Apache authentication handler must be installed.

A version of the software written after the conclusion of the initial study re-implemented the server in Java, so server-side requirements are consequently limited to a Java environment, plus any secondary programs (relational databases, etc.) used by particular projects running on the Grid.

3.3.4. Account

Only an ordinary user account is necessary for using the Grid system on client machines; no root privileges are necessary. Once an account is established on a client, it is accessed via `ssh`, a small Perl script (the grid client) is transferred to the client via `scp`, and the script is executed. In the spirit of "fire and forget", all subsequent communication between the client and server is performed automatically for the duration of the project, except in certain cases of local system failure (e.g., a reboot, NFS failure). For the most part, the client is able to cope gracefully with failures; the response to a network failure, for example, is simply to sleep for a specified interval and attempt to continue.

3.4. System processes

From a functional perspective, the Grid system can be viewed as having three major groups of processes: user, grid_client, and project. The user level processes are those necessary for the initial, and typically the only, human interaction with the client. The grid_client processes initialize client–server interaction and locally maintain (e.g., get, check, update, delete) files retrieved from server. The project processes include all of the application specific steps. The specific processing steps for each group of processes for the Grid system used in the initial project are described as pseudocode as follows.

3.4.1. user level processes

Performed once per client unless system reboot.

login to client
    copy grid_client Perl script to local (client resident) directory
    execute grid_client
logout

3.4.2. grid_client processes

get client architecture and operating system information
until client is terminated
  if a code package exists locally
    request code package MD5 value from server
    on server
      determine name of file to send based on client architecture
      lock file to send
      send file
  endif
  if MD5 values do not match **or** no local package present
    request code package from server
    on server
      determine name of file to send based on client architecture
      lock file to send
      send file
  endif
  unpack code package
  execute file within code package
enduntil
delete files

3.4.3. project processes: executed once per invocation by grid_client process

Steps presented here are for the initial study.

request analysis parameters from server
  on server
    lookup parameters for next analysis to run
    add record to the in_progress table
    increment ''replicates processing'' counter for analysis
    send parameters to client
execute analysis programs
send results back to server
  on server
    add record to the results table
    delete appropriate record from the in_progress table
    decrement ''replicates processing'' counter
    increment ''replicates completed'' counter
exit

The overall design of the Grid system is purposefully general. The user level processes would work for any Perl script or other program executable on a client. The initialization and file maintenance functions of the grid_client are project independent. The package processes could be anything of interest. Therefore, the system is very easily modified for use in solving other problems in which a similar computational model is appropriate. Work completed after the initial study had finished has reinforced the distinction between project-independent and project-specific code, thus making it easier to use the Grid for other projects.

3.5. Basic features

3.5.1. Client–server communications

In any Grid system common communications protocols must be established between clients and sever. Here, we employ the Transmission Control Protocol/Internet Protocol (TCP/IP) standard. TCP/IP guarantees reliable data delivery, albeit with significant overhead. However, for a system such as ours, which does not presume the availability of efficient, reliable networking and does not send large amounts of data between client and server, TCP/IP is ideal. Communications are conducted at different speeds throughout the system; clients used in our initial study had network connections ranging from local 100Base-T Fast Ethernet to Digital Subscriber Line (DSL), and T1, DS3, and Internet2 links were also used.

Communicated over the TCP/IP connections are HTTP POST requests that constitute the messages between server and clients. By using client-initiated HTTP requests for communication, we virtually guarantee that intervening firewalls will not block our messages; outbound requests to port 80 are rarely blocked. Secure Shell (SSH), a secure method for accessing user accounts on remote systems, is also initially used to distribute the client scripts.

The organization of the client–server communications is simple but sufficient. In the version of the software used in the initial study, the central server has four Common Gateway Interface (CGI) scripts (these are actually HTML::Mason files that allow Perl code to be mixed with HTML) that accept POST requests from clients. The scripts are job_dispatch.mason, request.mason, post.mason, and error.mason.

job_dispatch.mason sends code packages to the clients in the Grid. When they access this script, clients supply a mandatory ''arch'' CGI parameter that encodes their operating system name and processor type. They also supply an ''operation'' parameter, which is set to ''file'' or ''checksum.'' In the first case, the actual job package file is sent as type application/x-gzip. In the second case, a one-line file containing the 128-bit Message Digest 5

(MD5) value [16,25] of the package file is sent as type text/plain.

request.mason is the work unit dispatch side of the server. Clients supply a single parameter, hostname, when they access it. The response is a document of type text/plain containing four lines. Each line contains a key: processor, which specifies whether MrBayes or PAUP* should be used; brlen1, which specifies the three-branch length in the model (abscissa in model space; Fig. 1); brlen2, which specifies the two-branch length (ordinate in model space; Fig. 1); and seed, which specifies the seed to be passed to the psuedo-random number generator in evolver, which generates the sequences. Lines begin with a hash (#), and keys and values are separated by a colon and a space. Because clients receive a seed to use to generate their input rather than the input itself, we both increase the parallelization and decrease the bandwidth necessary to send work units.

post.mason accepts results for completed work units. Clients post CGI parameters for the three topology counts (the data of interest) and the ID of the work unit they are submitting. The response from the server is either "Result Received." or an error message beginning with "Error: ".

Finally, error.mason receives error messages from the clients. Clients call it with two CGI parameters: job_id indicates the ID of the work unit during which the error occurred, and errmsg is the error string to record.

Clients required on the order of 2–3 min, on average, to process a work unit (assuming a processor roughly equivalent to a Pentium-III), so each client generally communicated with the server every 2–3 min. Messages were on the order of tens of bytes, but the protocol was not optimized for terseness. The required information in a work unit consisted of an integer job ID, the type of analysis to perform (which could have been represented with a single bit), two decimal branch lengths (requiring two digits after the decimal), and an integer random seed. Result units contained an integer job ID, a string identifying the host and three floating-point topology counts. A request for a work unit contained a string identifying the host.

In a version of the software written after the initial project was completed, these CGI scripts were replaced with XML-RPC, but the logic remains largely unchanged.

3.5.2. Authentication

To properly control participation in a Grid system, a means of authenticating participating clients must be established. Here, a list of allowed clients is maintained on the server, and client machines are added to the list via a Perl script. In the present implementation of this Grid system, authentication is based on client IP addresses: when a client requests work or reports a result, Apache checks that its IP address is in the permit list. Authorized clients are assumed to return valid results. This is an admittedly simple system, but it is sufficient to prevent casual crackers from posting bogus results. The use of IP addresses means that the authentication list does not take advantage of the Domain Name System (DNS) and must thus be updated manually if addresses change; however, use of the DNS would require at least two reverse lookups (and thus non-trivial network and time overhead) for each work unit.

In any parallel computing system with asynchronous computing and communications, the design must incorporate features to ensure concurrent assignment of jobs and recording of results. In the version of the Grid system used for the initial project, the use of Apache on the server automatically allowed it to communicate with multiple clients; as a Web server, Apache is built to handle hundreds of simultaneous connections. However, Apache could only simulate multiplexed access to the database because we used the MyISAM (Indexed Sequential Access Method) MySQL [22] table type, and this table type is non-ACID (Atomicity, Consistency, Isolation, Durability) compliant. Thus, when dispatching work units, each Apache server process had to acquire an exclusive lock on one of the database tables, and this effectively serialized access to the database. The Apache processes spent very little time in these critical sections, however, and thus this potential bottleneck tended to be unimportant in practice: the servers spent far more time sending and receiving data than they did accessing the database, and the system scaled to over 135 simultaneous clients without issue. However, MySQL support for transactions has improved since the end of the initial project, so this problem would not be an issue for future projects.

3.5.3. Architecture specific binaries

In a Grid system incorporating computational resources that are heterogeneous with respect to architecture and operating system, the server must ensure that client systems always receive the proper binaries. The Perl script executed on each client includes a system call to the UNIX command `uname`, which returns information about the architecture and operating system. When requesting a code package file or MD5 value, clients encode information specifying their architecture and operating system into the request URL as a CGI parameter. This parameter is received by the server and interpolated into the filename of the code package to send. On the server side, code packages are kept in their un-archived form in directories outside of the document root of the Web server; symbolic links in each directory pull in shared, platform-independent files from a common location, while architecture-specific binaries reside in their respective package directories.

Scripts exist to generate the archived files (using `tar`) from the directories and to replace the archive files in use by the server (the latter script using cooperative file locking with the Web server in order to ensure that code packages may be safely replaced while the Web server is active).

3.5.4. Client-side security

If a Grid computing system is to incorporate computers over which its operators do not have direct administrative responsibility, it is not only necessary to convince the administrators of those systems that their participation will not compromise the security of their systems, but it is also necessary to make design decisions so as to insure that the conviction is justified. In this Grid system, the primary danger would be that an attacker could fool the client into accepting and running arbitrary code with the privileges of a local user. At present, clients do receive an MD5 value of the code to run, but this adds little security and still leaves them open to a DNS-poisoning attack. The solution to this problem is to add digital signing of the code sent to the clients (see Section 5).

Otherwise, we believe the security risk to clients to be extremely low. Because the client process runs with the privileges of a normal user, even the worst case of arbitrary code execution only results in a remote user exploit, not a remote root exploit, and the security measures presumably in place on the host system would prevent further escalation. Moreover, if increased client-side security is desired the client-based processes could be run in a chroot jail [38], thus preventing the client from accessing the rest of the file system and limiting the damage a cracker could cause.

3.5.5. Server-side security

The primary security risk on the server side is that an attacker might attempt to submit bogus results; as authentication is at present based only on the IP address of the client and correctly formatted CGI (or XML-RPC) requests can be easily crafted, this would not be terribly difficult to do. We assumed that no one would have reason to intentionally corrupt our results, so we deemed the extra complexity of strong client authentication to be unnecessary. Higher-profile studies could conceivably come to the opposite decision and utilize additional security features (see Section 5).

Otherwise, as the server only receives numeric data from the clients, does not execute unknown code on their behalf, and uses a modern, safe language, we believe that the chance of a security breach on the server as a result of this system to be exceedingly low.

3.5.6. System monitoring

An important administrative need of any Grid system is to easily monitor the system to assess performance and identify problems. In the version of the software used in the initial study, the Perl module that contains the functions to manipulate the database includes several that provide monitoring information, including statistics on the clients participating (hostname, work units completed, average time per unit, date and time of last unit submission, number of errors generated), overall progress (work units completed and currently processing for each point in the model space), current results (average results for each point using data to date), and errors (error message, client/work unit involved, time received). These data are returned as Perl data structures, and thus they are easily converted into text files, e-mails, or Web pages. We used a single large access-controlled Web page to display all these data, which allows us to monitor the system from any Web browser; we also generated graphical representations of the work units/hour data for each host to track trends in the system.

In the revised version of the software, the server periodically dumps non-domain-specific statistics (i.e., everything listed above save current results) to a text file; external programs may format it for display. This process executes without the direct participation of the projects themselves, thus obviating the requirement for individual Grid projects to rewrite such code.

Additionally, clients locally log which work units they are tasked to and what the results for those units were; this facilitates fine-grained debugging of clients as necessary.

During the initial study, the system monitoring provided information that was very helpful in the day-to-day use of the Grid system. System monitoring allowed us to quickly identify clients that had problems (e.g., were down or had been rebooted) and to initiate corrective steps, to assess the relative performance of different computer systems for our specific applications, and to get updated estimates of the completion time for our simulation, which in turn was useful in planning our other activities on the project. Furthermore, the summary of results to date provided scientifically relevant information that stimulated much thought and discussion well in advance of project completion.

3.5.7. Error handling

In order to use administratively separate and possibly unreliable computers, a Grid system must be able to gracefully cope with error conditions. In our system all errors are logged, thus allowing easy identification of problematic clients. We approach the problem of errors preventing completion of analyses by simply rescheduling, with identical starting parameters (i.e., random seeds), any work units for which results are not returned within a reasonable length of time. A periodically run script identifies such units, purges them from the outstanding-work list, and adds them to the tail of an

error queue in the database; work units at the head of this queue have priority over other units for dispatch to clients.

Additionally, clients themselves may send error reports to the server; for example, if a process returns "can't happen" results (if, for example, it is killed while processing), the client will send an "impossible result" message to the server. This has happened frequently during the study: one of our clients was unable to run one of the problem-specific application programs correctly, so roughly 50% of its results were "invalid." While the automatic-culling system, above, would have rescheduled the jobs, client-initiated error messages allow both faster rescheduling of work units and a better insight into the causes of errors.

4. Performance considerations

For the scope of this section we define performance to be analysis throughput in units of replicates completed per unit time, other similar measures could be used. Performance in other contexts (e.g., security, robustness) is implicitly included in previous sections describing system features.

One of the major advantages of this Grid system is that it is based on popular, well-understood standards. Thus, we were able to avail ourselves of high-quality implementations of relevant software. For example, our use of Apache/mod_perl and MySQL on the server side means that the server is unlikely to impose a practical limit on the maximum number of hosts that can participate at any one time; even with our relatively un-optimized implementation, our server (dual 700 MHz Pentium III SMP, 512 MB RAM) was able to handle upwards of 135 client processors simultaneously without perceptible decrease in performance. Should the server become a bottleneck, the entire spectrum of well-known HTTP/Apache/mod_perl/MySQL optimizations would be available, including proxy servers and clustering [33]. However, independent benchmarks of Apache/mod_perl and MySQL showed that, given a trivial database query (and our database queries were relatively trivial), 656.6 requests s^{-1} could be processed on a modest server (dual 450 MHz Pentium III SMP, 1 GB RAM) [40]. It thus seems unlikely that server CPU would be limiting for projects of the scale envisioned.

A second practical limitation stems from the bandwidth available to the host server. Conceivably, it would be possible to have enough clients participating in the Grid that the coordinating server would be unable to either serve code packages or dispatch/receive work units in a timely manner. Saturation due to the delivery of code packages to clients is both the least serious and easiest to resolve of these problems. It is least serious because code packages only need to be sent infrequently; unless a code package has changed since last downloaded, clients use existing copies. Bottlenecks would thus only come from the dispatch of new code packages and would be transient. Nonetheless, were this to become a significant problem, a mirror system could be constructed to distribute the load amongst multiple servers; code packages are static files and servers serving them have no need to communicate amongst themselves. Additionally, rsync [39] could be used in lieu of HTTP to distribute code packages; in those cases where only part of the code package changes, rsync would economize bandwidth by only transmitting the changes to clients. Finally, a special client for systems with shared file systems (e.g., appropriately configured Beowulf clusters or clusters of workstations [COWs]) could be designed that would allow multiple client processes to share a single code package, thus reducing network traffic.

Were the transmission of work units and their results to become a bottleneck, some relief could be realized by decreasing the verbosity of the protocol; in the initial study, we optimized for legibility and not to minimize bandwidth used. Beyond that, a hierarchical system of servers could be set up to handle work units as proposed for code packages above, but this would be a significantly more complex undertaking, as these servers would have significant synchronization requirements. However, we believe that bandwidth saturation is unlikely: even assuming 500 bytes per work unit dispatched or result reported (for our initial project, the figure was far lower), a 1.544 megabit s^{-1} T1 line could handle 386 simultaneous messages. However, because not all clients communicate with the server simultaneously, we would actually be able to support far more than 386 of them. We thus conclude that, except for very large projects, the network capacity to the server is unlikely to be a significant bottleneck.

The greatest practical limitation on throughput comes from server-side RAM exhaustion; at 10–15 MB each, multiple Apache/mod_perl server processes can quickly devour available memory on a server. However, there are multiple solutions to this problem (see Section 5).

Despite the above considerations, our greatest limitation on throughput has come not from the Grid infrastructure but rather from a lack of clients; we have simply been unable to obtain access to a sufficient number of computers to run up against the limits of our Grid system. The initial project to which the Grid system was applied ran on 163 unique clients distributed over a wide area (Fig. 2), used over 135 processors simultaneously, processed up to 4064 replicates h^{-1}, and consumed over 15.68 wall-clock-years of CPU time.

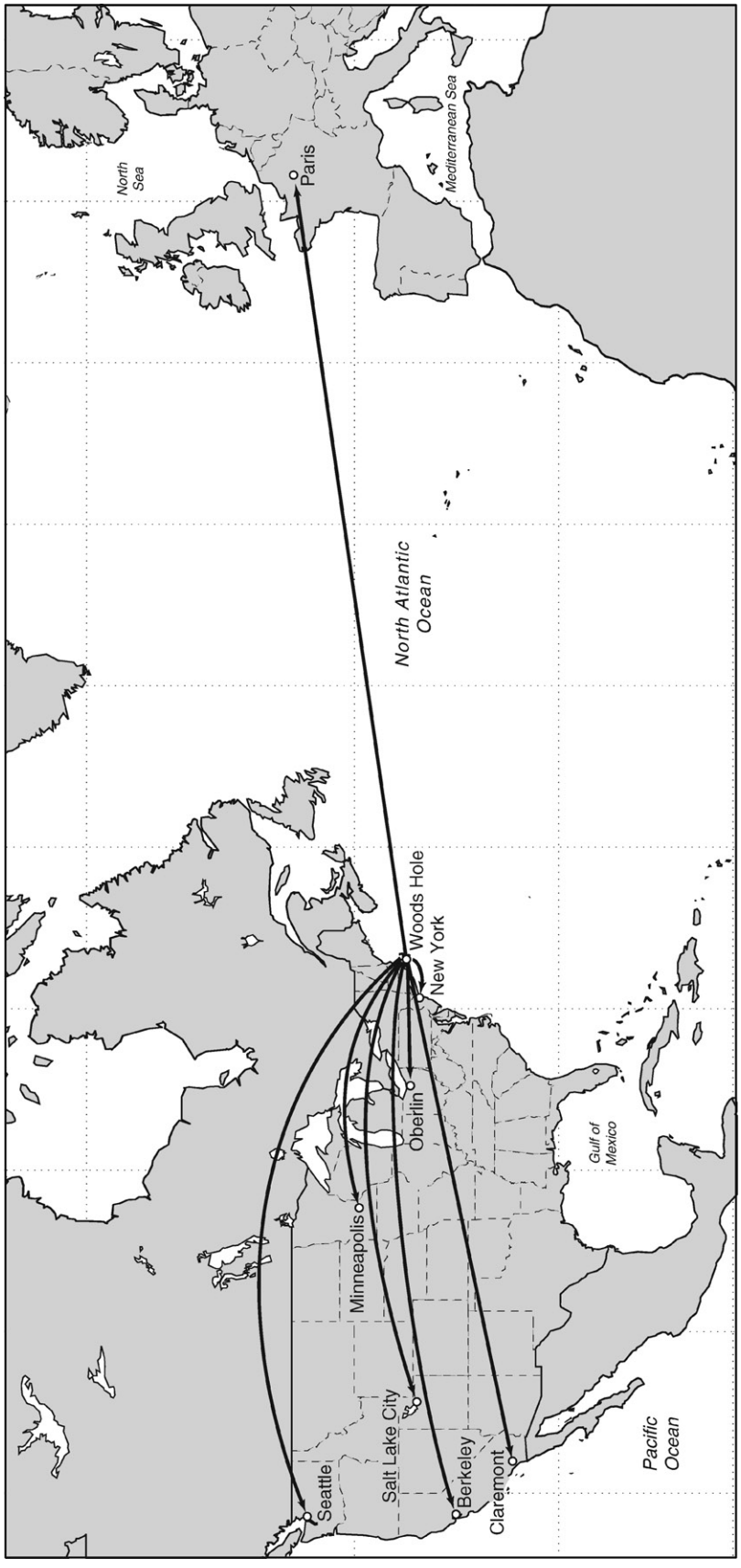

Fig. 2. Map of a portion of the northern hemisphere showing the general geographic distribution of clients and server used for the initial project with the Grid system.

5. Future work

As alluded to previously, work performed since the completion of the initial project described here has focused on re-implementing the server in Java. Java support for classes allows a rigorous distinction to be drawn between the code of the Grid server and the domain-specific code for each project. At the time of this writing, we have built a mostly functional prototype of the Java server. Grid computing projects take the form of classes implementing an interface; they provide a few basic methods (get the next work unit, receive a result unit, get the URL for the code to be downloaded by the client, and handle a lost work unit), and the provided server class handles the rest (assignment of clients to projects, generation of host statistics, and garbage collection of lost work units). The server runs either using a Web server supplied by the XML-RPC library from the Apache Software Foundation [34] or as a Java servlet; in either case, the one-client, one-thread model is easier to work with than the one-client, one-process model provided by the Apache/Perl solution used previously.

Areas requiring continued work are listed below.

5.1. Secure communications

As mentioned above, the protocol used for client–server communication is vulnerable to being tricked into executing arbitrary code by attacks on the DNS. Thus, it would be desirable to add digital signing of the code sent to the clients. A pure-Perl implementation of the OpenPGP standard exists [35–37] and could be used to add this functionality.

Additionally, as authentication is IP-address based, the server is vulnerable to dispatching work to and accepting results from unauthorized clients. Also, communications between the clients and the server are not secure. HyperText Transfer Protocol, Secure (HTTPS) would trivially solve the latter problem, and, while Secure Sockets Layer (SSL) certificate-based authentication would render the system slightly more complex (certificates would have to be generated and distributed), the ability to use it would nonetheless be a useful addition to the Grid system.

5.2. SQL transaction support

While the Java implementation of the server does not, strictly speaking, require the use of a relational database (in the previous version, a RDBMS provided the easiest way to handle the synchronization issues associated with a multi-process model), a database likely still represents the most effective way to store data for long-running projects. Servers might crash over the course of a multi-month project, and a RDBMS provides an easy way to ensure that data are not lost in such a case. Particularly, with introduction of the InnoDB [28] table type, MySQL [22] has acquired support for SQL transactions, which allow one to execute groups of SQL statements in atomic blocks (either all or none of the statements execute, thus preserving database integrity in the event of hardware failures).

5.3. A little language

While a RDBMS may be the best technical solution for server-side storage, writing the Java Database Connectivity (JDBC) and SQL code to interface to them is at best tedious and can be quite difficult for a programmer without prior exposure. One interesting area for future work would be the development of a small language to describe the fields of work and result units; this language could then be compiled to Java classes able to fetch and store those units, possibly verifying that results returned fit given criteria. This would significantly lower the barrier for use of the Grid system.

5.4. Validity checking

In the initial study, results that "look right" (i.e., are within known mathematical limits) are assumed to be correct. Even if the Grid system were to be enhanced with strong authentication, it would still be vulnerable to incorrect results that are internally generated. For example, if a C program used to actually execute an analysis had a bug that gave incorrect but properly formed results, those results would be accepted. Thus, it would be convenient to be able to cross check results between clients. For example, every nth unit from each client might be re-dispatched to a different client and the result compared with what was originally obtained; in the case of a disagreement, a third client would be used to break the tie. Such a system might be unnecessary and undesirable to use throughout a study, as once a sample of results from a client had been established as valid, continued testing simply decreases overall throughput; however, cross checking would be extremely useful in the early stages of a study or when new clients are introduced.

6. Conclusions

We have described a simple Grid system that we have developed and are using for large-scale simulation studies with "borrowed" computing resources distributed over a wide area. Based on commodity software components, our Grid system implements the necessary functionality of basic computational Grid systems. Unlike some alternatives, our Grid system requires

neither application modification nor services from client system administrators beyond user account creation. The system has displayed excellent performance in terms of scaling, security, robustness, and throughput. Given its modular and extensible design, we believe that this system provides a model for use in problems that can avail themselves of a computational Grid of a loose star topology using client-initiated communications and UNIX or UNIX-like operating systems. Furthermore, for appropriate problems, this Grid model provides the means to utilize computing assets that are largely forgotten or overlooked.

Acknowledgments

We thank our friends and colleagues for providing computing resources used in this project: Robert Campbell, Monica Riley, Laura Shulman and Mitchell Sogin, Marine Biological Laboratory; Kenneth Halanych and Arthur Newhall, Woods Hole Oceanographic Institution; Dale Clayton, David Reed and Center for High Performance Computing, University of Utah; Computer Science Clinic Program, Harvey Mudd College; James Marshall, Pomona College; David Danziger, Oberlin College; Michael Fister, Intel Corporation; Joe Felsenstein and Mary Kuhner, University of Washington; Ernest Retzel, Kevin Silverstein, and Computational Biology Centers, University of Minnesota; Lawrence Leung, University of California, Berkeley; and Aaron Grogan. We also thank Maile Neel and the anonymous reviewers for useful comments on the manuscript, and both Ken Rice and SETI@Home for providing stimulating examples of distributed computing. Tzu-Yi Chen, Pomona College; and Geoff Kuenning, Harvey Mudd College provided comments during the revision process.

References

[1] A.-L. Barabási, V.W. Freech, H. Jeong, J.B. Brockman, Parasitic computing, Nature 412 (2001) 894–897.
[2] D.J. Becker, T. Sterling, D. Savarese, J.E. Dorband, U.A. Ranawak, C.V. Packer, Beowulf: a parallel workstation for scientific computation, in: Proceedings of the 24th International Conference on Parallel Processing, Oconomowoc, WI, 1995, pp. 11–14.
[3] M.P. Cummings, S.A. Handley, D.S. Myers, D.L. Reed, A. Rokas, K. Winka, Comparing bootstrap and posterior probability values in the four taxon case, Syst. Biol., submitted.
[4] M.P. Cummings, S.P. Otto, J. Wakeley, Sampling properties of DNA sequence data in phylogenetic analysis, Mol. Biol. Evol. 12 (1995) 814–822.
[5] M.P. Cummings, S.P. Otto, J. Wakeley, Genes and other samples of DNA sequence data for phylogenetic inference, Biol. Bull. 196 (1999) 345–350.
[6] B. Efron, E. Halloran, S. Holmes, Bootstrap confidence levels for phylogenetic trees, Proc. Natl Acad. Sci. USA 93 (1996) 7085–7090.
[7] B. Efron, R.J. Tibshirani, An Introduction to the Bootstrap, Chapman & Hall, New York, 1993.
[8] J. Felsenstein, Confidence intervals on phylogenies: an approach using the bootstrap, Evolution 39 (1985) 783–791.
[9] I. Foster, C. Kesselman, Globus: a toolkit-based Grid architecture, in: I. Foster, C. Kesselman (Eds.), The Grid: Blueprint for a New Computing Infrastructure, Morgan-Kaufmann, Los Atlos, CA, 1999, pp. 259–278.
[10] B.S. Gaut, P.O. Lewis, Success of maximum likelihood phylogeny inference in the four-taxon case, Mol. Biol. Evol. 12 (1995) 152–162.
[11] J.P. Huelsenbeck, D.M. Hillis, Success of phylogenetic methods in the four-taxon case, Syst. Biol. 42 (1993) 247–264.
[12] J.P. Huelsenbeck, F. Ronquist, MRBAYES: Bayesian inference of phylogenetic trees, Bioinformatics 17 (2001) 754–755.
[13] A.D. Leaché, T.W. Reeder, Molecular systematics of the eastern fence lizard (*Sceloporus undulatus*): a comparison of parsimony, likelihood and Bayesian approaches, Syst. Biol. 51 (2002) 44–68.
[14] P.O. Lewis, Phylogenetic systematics turns over a new leaf, Trends Ecol. Evol. 16 (2001) 30–37.
[15] S.P. Otto, M.P. Cummings, J. Wakeley, Inferring phylogenies from DNA sequence data: the effects of sampling, in: P.H. Harvey, A.J. Leigh Brown, J. Maynard Smith, S. Nee (Eds.), New Uses for New Phylogenies, Oxford University Press, Oxford, 1996, pp. 103–115.
[16] R.L. Rivest, RFC 1321: The MD5 Message-Digest Algorithm, Internet Activities Board, 1992.
[17] D.L., Swofford, PAUP* 4.0 beta 8 *Phylogenetic Analysis Using Parsimony (and Other Methods), Sinauer Associates, Sunderland, MA, 2002.
[18] L.A. Whittingham, B. Slikas, D.W. Winkler, F.H. Sheldon, Phylogeny of the tree swallow genus, *Tachycineta* (Aves: Hirundinidae), by Bayesian analysis of mitochondrial DNA sequences, Mol. Phylo. Evol. 22 (2002) 430–441.
[19] Z. Yang, PAML: a program package for phylogenetic analysis by maximum likelihood, Comput. Appl. Biosci. 15 (1997) 555–556.
[20] http://www.apache.org/
[21] http://java.sun.com/
[22] http://www.mysql.com/
[23] http://perl.apache.org/
[24] http://dbi.symbolstone.org/
[25] http://www.perldoc.com/perl5.6.1/lib/Digest/MD5.html, http://www.faqs.org/rfcs/rfc1321.html
[26] http://www.cpan.org/modules/by-module/LWP/, http://www.perldoc.com/perl5.6.1/lib/LWP/, http://www.linpro.no/lwp/
[27] http://www.masonhq.com/
[28] http://www.innodb.com/
[29] http://www.xmlrpc.com/
[30] http://www.distributed.net/
[31] http://www.entropia.com/
[32] http://www.ud.com/
[33] http://perl.apache.org/guide/
[34] http://xml.apache.org/xmlrpc/
[35] http://rhumba.pair.com/ben/perl/openpgp/
[36] http://www.landfield.com/rfcs/rfc1991.html
[37] http://www.gnupg.org/rfc2440-1.html
[38] http://www.linuxdoc.org/HOWTO/Chroot-BIND-HOWTO.html
[39] http://www.rsync.org/
[40] http://www.chamas.com/bench/hello_bysystem.html#hellodb

Daniel S. Myers is currently a fourth-year undergraduate pursuing a B.A. degree in computer science at Pomona College, Claremont, CA, and is a Research Assistant in the laboratory of Michael Cummings at

the Josephine Bay Paul Center for Comparative Molecular Biology and Evolution, Marine Biological Laboratory, Woods Hole, MA. His interests include neural networks and parallel computing, with an emphasis on their application to problems of biological relevance. Recent research has included the application of the genetic algorithm to the problem of evolving training algorithms for Kohonen Self-Organizing Feature Maps.

Michael P. Cummings received a B.Sc. degree in botany from the University of California, Davis, and a Ph.D. degree in biology from Harvard University. He received an Alfred P. Sloan Foundation Postdoctoral Fellowship in Molecular Studies of Evolution and has done postdoctoral research at the University of California, Berkeley, and the University of California, Riverside. He is currently Assistant Scientist, and Director of the Workshop on Molecular Evolution, at the Josephine Bay Paul Center for Comparative Molecular Biology and Evolution, Marine Biological Laboratory, Woods Hole, MA. His research interests are in the areas of molecular evolutionary genetics, including population genetics, systematics, genomics and bioinformatics.