G-BLAST: A Grid Service for BLAST

Purushotham Bangalore, Enis Afgan
Department of Computer and Information Sciences
University of Alabama at Birmingham
1300 University Blvd., Birmingham AL 35294-1170
{puri, afgane}@cis.uab.edu

Abstract – This paper described the design and implementation of G-BLAST – a Grid Service for one of the most widely used bioinformatics application Basic Local Alignment Search Tool (BLAST). G-BLAST uses the factory design pattern to provide application developers a common interface to incorporate multiple implementations of BLAST. The process of application selection, resource selection, scheduling, and monitoring is completely hidden from the end-user through the web-based user interfaces and the programmatic interfaces enable users to employ G-BLAST as part of a bioinformatics pipeline. G-BLAST uses an adaptive scheduler to select the best application and the best set of resources available that will provide the shortest turnaround time when executed in a grid environment. G-BLAST is successfully deployed on a campus and regional grid and several BLAST applications are tested for different combinations of input parameters and computational resources. Experimental results illustrate the overall performance improvements obtained with G-BLAST.

Keywords: grid, BLAST, scheduling, usability

1 Introduction

Basic Local Alignment Search Tool (BLAST) is a sequence analysis tool that performs similarity searches between a short query sequence and a large database of infrequently changing information such as DNA and amino acid sequences [8, 9]. With the rapid development of sequencing technology of large genomes for several species, the sequence databases have been growing at exponential rates [11]. Facing the rapid expanding target databases and the length and number of queries used in the search, the BLAST programs take significant time to find a match. Parallel computing techniques have helped BLAST to gain speedup on searches by distributing searching jobs over a cluster of computers. Several parallel BLAST search tools [13, 19] have been demonstrated to be effective on improving BLAST’s performance. mpiBLAST [19] and TurboBLAST [13] use database segmentation to distribute a portion of the sequence database to each cluster node. Thus, each cluster node only needs to search a query against its portion of the sequence database. On the other side, some researchers apply query segmentation to alleviate the burden of searching jobs [16, 18]. In query segmentation, a subset of queries, instead of the database, is distributed to each cluster node, which has access to the whole database. As for as the end-user of the BLAST application is concerned the final outcome and turnaround time is of interest and typical users do not really care which of the above techniques were used to generate the final results.

The majority of parallel BLAST applications, however, cannot cross the boundary of computer clusters, i.e., the communication among parallel instances of BLAST algorithms are limited among computing nodes with homogeneous system architectures and operating systems. This limitation heavily encumbers the development of cooperative BLAST applications across heterogeneous computing environments. Particularly when many universities and research institutes have started to build grids to take advantage of various computational resources distributed across the organization.

The emerging Grid computing technology [12] based on Service Oriented Architecture (SOA) [20] and Web Services provides an ideal development platform to take advantage of the distributed computational resources. Grid computing not only presents maximum available data/computing resources to BLAST search, but also shows its powerful ability on some critical issues, such as security, load balancing, and fault tolerance. Grid services [20] provide several unique features such as statefull services, notification, and uniform authentication and authorization across different administrative domains. The focus of this paper is to develop a grid service for BLAST by exploiting these unique features of grid services and provide ubiquitous access to distributed computational resources as well as hide the various details about application and resource selection, job scheduling, execution, and monitoring. One of the goals of this work is to provide a web-based interface through which users can submit queries, monitor their job status, and access results. The portal could then dispatch the queries to all the available computing resources according to a well-planed scheduling scheme that takes into account the heterogeneity of the resources and performance characteristics of BLAST on these resources based on the query length, number of queries, and the database used. An additional goal of this effort is to provide applications developers a common interface so that different implementations of BLAST could be easily
incorporated within the G-BLAST service. A scheduler can then select the best application (multithreaded, query split, or database split) for the available resources and dispatch the job to the appropriate resource(s). There is no need for the end-user to be concerned about which version of BLAST application was used as well as the computational resource(s) that where used to execute the application.

The rest of this paper is organized as follows. Section 2 presents an overall architecture of G-BLAST and described the various components. The experimental setup and deployment details used to test G-BLAST are provided in section 3. Other related work is described in section 4 and summary and conclusions are provided in section 5.

2 G-BLAST Architecture

The overall architecture of G-BLAST is illustrated in Figure 1. G-BLAST has the following four key components:

(a) G-BLAST Core Service: Provides a uniform interface through which a specific version of BLAST could be instantiated. This enables application developers to extend the core interface and incorporate newer versions of BLAST applications.

(b) User Interfaces: Provides web and programmatic interface for file transfer, job submission, job monitoring and notification. These interfaces support user interactions without exposing any of the details about the grid environment and the application selection process.

(c) Scheduler: Selects the best available resource and application based on user request using a two-level adaptive scheduling scheme.

(d) BLAST Grid Services: Individual grid services for each of the BLAST variations that are deployed on each of the computational resource.

2.1 G-BLAST Core Service

A BLAST Grid Service with a uniform Grid service interface is deployed on each of the computing resources. It is located between the Invoker and each implementation of BLAST programs. No matter what kind of BLAST programs are deployed on each resource, the BLAST Grid service should cover the differences and provide fundamental features. To facilitate developers to integrate individual BLAST instances into the G-BLAST framework, the BLAST Grid service defines the following methods for each instance:

1. **UploadFile**: Upload query sequences to a compute node.
2. **DownloadFile**: Download query results from the compute node.
3. **RunBlast**: Invoke corresponding BLAST programs on the compute node(s).
4. **GetStatus**: Return current status of the job (i.e., pending, running, done).
5. **NotifyUser**: Notify the user once the job is complete and the results are available.

With G-BLAST developers can easily add new BLAST services (corresponding to the BLAST programs and the computing resources supporting it) without modifying any G-BLAST core source code. In addition, developers can add new BLAST services on the fly, without interrupting any of the other G-BLAST services. To accommodate for such functionality, G-BLAST employs the creational design pattern “factory method” [22] to enable the invoker to call newly-built BLAST services without changing its source code. To integrate their corresponding BLAST programs into this framework, developers should create and deploy Grid services on each of the computing resources in the Grid.

As described in Figure 2, **Invoker** and **BLASTService** are two abstract classes representing the invoker in the G-BLAST service core and the BLAST services on computing resources, correspondingly. When a new BLAST service (e.g., mpiBLAST) is added into the system, the relevant invoker (mpiInvoker) for that service must be integrated as a subclass of the class Invoker. When the invoker wants to call the new BLAST service, it can first create an instance of mpiInvoker, then let the new invoker generate an instance of mpiBLAST by calling the member function CreateService(). Thus, the invoker does not need to hard-code instantiation of each type of BLAST services.

![Fig 1. Overall architecture of G-BLAST](image)

![Fig 2. Factory method for BLAST service](image)
This design pattern encapsulates the knowledge of which BLAST services to create and delegate the responsibility of choosing the appropriate BLAST service(s) to the scheduler (described in Section 2.C). The Invoker could invoke more than one BLAST service based on the availability of resources to satisfy user requirements.

2.2 User Interfaces

G-BLAST framework provides unified, integrated interfaces for users to invoke BLAST services over heterogeneous and distributed Grid computing environment. The interfaces summarize the general functionalities that are provided by each individual BLAST service, as well as cover the implementation details from the end users. Two user interfaces are currently implemented to satisfy different users' requirements. For users who want to submit queries as part of workflow, a programmable interface is furnished through a Grid Service. Service data and notification mechanism supported by Grid Services are integrated into the BLAST Grid service to provide statefull services with better job monitoring and notification. For users who want to submit each query with individual parameter settings and familiar with traditional BLAST interface, like NCBI BLAST [26], a web interface is implemented for job submission, monitoring, and file management.

G-BLAST exploits the notification mechanism [20] provided by grid services in two aspects. One aspect is the notification of changes by BLAST services to the scheduler. The other aspect is the notification of job completion to the end users. Both of these two instances strictly follow the protocol of notification. In notification of service changes, the BLAST services are the notification source, and the scheduler is the notification subscriber. Whenever the BLAST service on the computing node has any changes, the service itself will automatically notify the scheduler with up-to-date information. This mechanism keeps the scheduler updated with the most recent status of the BLAST service, therefore helps scheduler make informed decisions on the selection of computing resources. Notification for job completion has a similar implementation, except that the notification sink is the registered client program.

To facilitate users' using G-BLAST, a programming template is also provided to guide users' code their own client program for G-BLAST service invocation. Figure 3 demonstrates the major part of a client program that invokes G-BLAST service by creating Grid service handler, uploading query sequence(s) to the back-end server, submitting a query job, checking the job status, and finally retrieving the query results.

In addition to providing a programmable interface for the end user, the framework also provides a web workspace that supports the needs of a general, non-technical Grid user who prefers graphical user interface to writing code. The most common needs of a general user are file management, job submission, and job monitoring. File management is supported through a web file browser allowing users to upload new query files or download search result files. It is a simplified version of an FTP client that is developed in PHP. Job submission module is made as simple as possible to use, the user after naming the job for easy reference only provides or selects a search query file and chooses the database to search against. Application selection, resource selection, file mapping, and data transfer are handled automatically by the system. Finally, the job monitoring module presents the user with the list of his or her jobs. It includes a date range allowing the user to view not only the currently running jobs, but the completed jobs as well. When viewing the jobs, the user is given the name of the job, current status (running, done, pending, or failed) and job execution start/end time. Upon clicking on the job name, the user can view more detailed information about the query file, database used, and start and end time. The user is also given the option to re-submit a job with the same set of parameters or after changing one of the parameters.

// Get command-line argument as Grid Service Handler
URL GSH = new java.net.URL(args[0]);

// Get a reference to the Grid Service instance
GBLASTServiceGridLocator gblastServiceLocator = new GBLASTServiceGridLocator();
GBLASTPortType gblast = gblastServiceLocator.getGBLASTServicePort(GSH);

//Query sequence uploading
gblast.FileTransfer(inputFile, src, remote);

//Submit query as a job
gblast.BLASTRequest(blastRequest);
jobid=gblast.JobSubmit();

//Check query (job) status
gblast.JobStatus(jobid);

//Retrieve back the query result
gblast.ResultRetrieve(jobid);

Fig 3. Client program to invoke G-BLAST service

2.3 Two-level Adaptive Scheduler

Due to the heterogeneity of available resources in the Grid, system usability as well performance of the application can be drastically affected without an efficient scheduler service. Rather than developing a general purpose meta-scheduler that tries to schedule any application equally by using the same set of deciding factors, we have created a two-level application-specific scheduler that uses application and resource specific information to provide a high-level service for the end user (either in turnaround time, better resource utilization, or usability of the system).

The scheduler collects application specific information in the Application Information Services (AIS) [3], initially from the developer through Application Specification Language (ASL) [3] and later from application runs. ASL assists
developers to describe the application requirements and the AIS acts as a repository of application descriptors that can be used by a resource broker to select the appropriate application. ASL is much like RSL but from the application point of view. It is a language that provides a way for the application developer to specify the requirements imposed by the application. It specifies deployment parameters that have to be fulfilled during runtime such as required libraries, specific operating system, minimum/maximum number of processors required to run, specific input file(s) required, specific input file format, minimum/maximum amount of memory and disk space, type of interconnection network and so on. Unlike RSL, where only the end user specifies the requirements for their job, ASL allows the application developer or owner to specify requirements for allowing the application to be run (e.g., licensing, subscription fee) and thus is creating a contract between the user and the developer.

The scheduler uses this information for each of the subsequent decision makings when selecting the best available resource (say, resource resulting in shortest turnaround time, or the cheapest resource, or the most reliable resource). Once the user provides the necessary job information (as described by the JDF), the scheduler obtains a snapshot of the available resources in the Grid, and based on the information obtained from AIS, it automatically performs a matching between the JDF and ASL as to which of the available algorithms and available resources will result in desired performance. For more details on the inner workings of the scheduler please refer to [5, 6].

3 Deployment and Results

3.1 UABGrid

UABGrid is a campus grid that includes computational resources belonging to several administrative and academic units at the University of Alabama at Birmingham (UAB). The campus-wide security infrastructure required to access various resources on UABGrid are provided by Weblogin [28] using the campus-wide Enterprise Identify Management System [27]. There is diverse pool of available machines in UABGrid, ranging from mini clusters based on Intel Pentium IV processors and Intel Xeon based Condor pool to several of clusters made up of 64-bit AMD Opteron and Intel Xeon CPU’s. Each of the departments participating has complete autonomy over local resource administration, resulting in a true grid setup. Access to individual resources was traditionally made through SSH or command line GRAM tools, but more recently we have added a general purpose resource broker [7, 23] and a portal that facilitates resource selection based on user’s job requirements. Since we are focusing our work to BLAST, a common feature of all the resources in UABGrid is that they have BLAST and/or mpiBLAST installed and available for use. The sequence databases on local resources are updated daily by a cron job and formatted appropriately to speedup user query searches.

3.2 Experimental Setup and Results

The scheduler has to select not only the best resource among a set of available resources but also select the best version of the BLAST application to deliver the shortest turnaround time for any given user request. In order to develop the knowledgebase required to test the capabilities of the scheduler in delivering these goals we have executed these applications on diverse computer architectures that are representative of actual resources on UABGrid [4]. On each of these computer architectures three different versions of BLAST algorithm: multithreaded, query split, and database split are executed. Each version of the BLAST application is in-turn tested with three protein databases of varying sizes and three different query file sizes (varying number of queries and query lengths). In this section we provide some of key performance results to illustrate the impact of these various parameters on the overall performance of BLAST queries and describe how the adaptive scheduler uses these performance results to decide the appropriate BLAST application and the computational resources.

Three protein databases available from NCBI’s website (ftp://ftp.ncbi.nlm.nih.gov/blast/db/) are used as part of these experiments. The smallest database selected was yeast.nt, a 13 MB representing protein translations from yeast genome. As the medium size database, we selected the non-redundant protein database (nr). It is an 821 MB database with entries from GenPept, Swissprot, PIR, PDF, PDB, and RefSeq [2] and finally, the largest database selected was 5GB est database, or Expressed Sequence Tags. This is a division of GenBank, which contains sequence data and other information on “single-pass” cDNA sequences from a number of organisms [1]. These databases represent a wide range of possible sizes and have been selected to reflect the most commonly used databases by the scientists in order to provide a solid base for the scheduling policies developed as part of this application. 10,000 protein queries were run against three NCBI’s protein databases. Query input files are grouped into three groups based on the number of queries; small, medium and large. Small number of queries is anything less than 100 queries, medium is between 100 and 1,000, while large is anything over 1,000 queries.

For any given computer architecture and BLAST version, the performance depends on the following input parameters: individual query lengths, total number of queries, and the size of the database against which the search is performed. Results indicate that the execution time increases linearly as the length of individual queries increase. Figure 4 provides BLAST execution time for queries of varying lengths using nr database on three different architectures. Similar experiments with other databases indicate that the execution time increases correspondingly when the database size increases. These experiments also highlight the importance of CPU clock frequency on the overall execution time of a BLAST query as the query length increases.

The performance of query splitting and database splitting approaches were also compared on different architectures
with different query files and databases. Figure 5 provides the comparison between query split and database split approaches for the \( nr \) database using 10,000 query input file. From the diagram, we can observe that BLAST keeps a nearly linear speedup up to 16 CPUs regardless of the algorithm used. Overall performance results indicate that the query-splitting algorithm outperforms database-splitting algorithm by almost a factor of two. Use of different database shows similar results as long as the size of the database is less than the amount of main memory available. In the later case, the database-splitting algorithm outperforms the query-splitting algorithm due to the reduced I/O involved in trying to keep only a portion of the original database in memory.

![Execution time vs. Query length with \( nr \) database on 1000 queries](image)

**Fig 4.** BLAST application performance as a function of query length.

Testing the validity of resource selection involved submitting a number of equal jobs and varying resource availability. We varied the number of available CPUs as well as availability of resources of different architectures and capabilities. Table 1 shows the performance of multithreaded BLAST on different architectures for 10 queries of varying lengths with the \( nr \) database. These results indicate that CPUs with higher clock frequencies and larger caches, along with more memory, outperform their slower counterparts. In addition, hyper-threading seems to offer significant performance improvements on the Intel Xeon EM64T architecture.

![Fig 5. Direct comparison of execution time of query splitting and database splitting versions of the BLAST with varying number of processors.](image)

**Figure 6** shows execution of a G-BLAST job across multiple resources using 100 queries and \( nr \) database. Following a job submission, G-BLAST selects resources for execution and the input data is automatically distributed and submitted to those resources. The figure shows execution of the same job using 16, 8, 4, and 2 processors among selected machines. As can be seen, the load imbalance across resources is minimized, but not eliminated. This is generally due to the inconsistencies of performance of individual resource that was not predicted by the scheduler, as well as the contention of any one fragment with other jobs currently being submitted to the given resource and thus competing with the one another.

The use of grid technologies and the described scheduler has enabled G-BLAST to move beyond executing on any single resource and execute on users’ jobs on multiple resources simultaneously, thus realizing a shorter job turnaround time. The aim of the scheduler is to minimize job’s overall turnaround time. This is achieved through selection of resources to execute the job from the pool of available resources and selection of which algorithm to employ on each resource. These two main directives are further complicated by the need to minimize load imbalance across selected resources. Details on the scheduler implementation can be found in [6], while the results in paper focus on showing the ‘value added’ to a user employing G-BLAST. G-BLAST enables execution of a job across multiple resources simultaneously, and because this is typically not available, it is hard to provide a direct comparison of obtained results. As such, provided results focus on internal functionality of the scheduler while overall job runtimes of G-BLAST jobs and standard BLAST jobs can be derived from Figure 4.

![Table 1](image)

**Table 1.** Performance comparison on different processor types against \( nr \) database (1.1 GB) and input file with 10 queries.

<table>
<thead>
<tr>
<th>Processor Type</th>
<th>Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Xeon (2.66 GHz, 32 bit, 512 Kb L2, 2 GB RAM) – Dual processors</td>
<td>508 265 266</td>
</tr>
<tr>
<td>Intel Xeon (3.2 GHz, 64 bit, 2 MB L2, 4 GB RAM) – Dual processors</td>
<td>426 231 180</td>
</tr>
<tr>
<td>AMD Opteron (1.6 GHz, 64 bit, 1 MB L2, 2 GB RAM) – Dual processors</td>
<td>471 243 242</td>
</tr>
<tr>
<td>Macintosh G5 (2.5 GHz, 64 bit, 512 KB L2, 2 GB RAM) – Dual processors</td>
<td>382 198 ---</td>
</tr>
<tr>
<td>Sun Sparce E450 (400 MHz, 64 bit, 4 MB L2, 4 GB RAM) – Quad Processors</td>
<td>2318 1183 590</td>
</tr>
<tr>
<td>Sun Sparce V880 (750 MHz, 64 bit, 8 MB L2, 8 GB RAM) – Quad Processors</td>
<td>1211 615 318</td>
</tr>
</tbody>
</table>

All of the results presented above indicate the different intricacies that a typical end-user has to handle while executing BLAST in a grid environment. The scheduler encapsulates all these details and makes it easier for the end-user to take advantage of a grid environment. By analyzing these experiments, we were able to confirm that the choice the scheduler was making during algorithm selection was indeed accurate. Under the constraints of resource availability, it can be inferred from the above figures the overall time saved by the user when performing searches.
using G-BLAST. We observed that with average resource availability of 8 CPUs on the UABGrid, the maximum time saved by a user was around 75%, when compared to executing the same job on a scientist’s local, single processor workstation.

4 Related Work

4.1 BLAST on Grid

Several Grid-based BLAST systems have been proposed to provide flexible BLAST systems that could harness distributed computational resources. GridBLAST [24] is a set of Perl scripts that distribute work over computing nodes on a grid using a simple client/server model. Grid-BLAST [32] employs a Grid Portal User Interface to collect query requests and dispatch those requests to a set of NCSA clusters. Each cluster in the system is added and tuned to accept jobs in an ad hoc way. The major disadvantage of a non-service based system is that the computing resources cannot be integrated into the system automatically and human intervention is required to adapt a new version of BLAST and new computational resources. GT3 based BLAST [10] system, however, is based on web services programming model. A meta-scheduler is also used to farm out query requests onto remote clusters. Nevertheless, the job submission is still through traditional batch submission tools and does not exploit the benefits of SOA and Grid Services.

4.2 Scheduling

Due to the heterogeneity of resources as well as different application choices in the Grid, resource selection is a hard task to perform correctly. Unlike the local schedulers [29, 33] which have much of the necessary information readily available to them, grid meta-schedulers are dependent on the underlying infrastructure. The general meta-schedulers such as Nimrod-G [15], AppLeS [21], the Resource Broker from CrossGrid [17], Condor [25], and MARS [14] are helping the general user in alleviating some of the intricacies of resource selection by automating resource selection across the Grid through application and resource parameter pooling. Due to the mentioned heterogeneity of applications and resources, a general meta-scheduler simply does not have enough information and support from the middleware to perform the optimal resource selection. In order to accommodate this need, application-specific meta-schedulers based on application runtime characteristics are used in G-BLAST framework to adaptively schedule applications on the grid. Runtime information is also used by other software packages (ATLAS [31] and STAPL [30]) to determine the best application specific parameters on a given architecture.

5 Summary and Conclusions

The overall architecture for G-BLAST – a Grid Service for the Basic Local Alignment Search Tool (BLAST) was presented in this paper. G-BLAST not only enabled the execution of BLAST application in a grid environment but also abstracted the various details about selecting a specific application and computational resource and provided simple interfaces to the end-user to use the service. Using the factory design pattern multiple implementations of BLAST were incorporated into G-BLAST without requiring any change to the Core Interface. The two-level adaptive scheduler and the user interfaces used by G-BLAST enabled the process of application selection, resource selection, scheduling, and monitoring without requiring extensive user interventions. G-BLAST was successfully deployed on UABGrid and different BLAST applications were tested for various combinations of input parameters and computational resources. The performance results obtained by executing various BLAST applications (multithreaded, query split, database split) on different architectures with different databases and query lengths illustrated the role of the adaptive scheduler in improving the overall performance of BLAST applications in a Grid environment. In this paper, we have used BLAST as an example for performing local alignment search. We also plan to extend this architecture to other bioinformatics applications.

REFERENCES


[33] Zhou, S., "LSF: Load Sharing in Large-scale Heterogeneous Distributed Systems," In the Proceedings of Workshop on Cluster Computing, 1992