PGL SERVER: DEVELOPMENT OF A STAND-ALONE SERVER-BASED EARTH-MODEL LIBRARY FOR SEISMIC MONITORING

Lee A. Jensen, James R. Hipp, Rose Q. Villanueva, Daniel S. Myers, Breanna Ammons, and Jack H. Gauthier
Sandia National Laboratories

Sponsored by National Nuclear Security Administration
Office of Nonproliferation Research and Development
Office of Defense Nuclear Nonproliferation
Contract No. DE-AC-04-94-AL85000

ABSTRACT

Sandia National Laboratories is developing a server version of the Parametric Grid Library (PGL), a shared-object library that allows storing and accessing Earth models for the National Nuclear Security Administration (NNSA) Knowledge Base. Creation of a server version of the PGL—PGL Server—allows several improvements when compared with the existing version of the PGL, which bundles an Earth model directly with an application.

1) Computational efficiency. From a practical viewpoint, currently when an application calls or starts up (i.e., instantiates) the PGL, there is a significant startup delay while the PGL reads in the Earth-model data. This delay can occur frequently if the application is not always in use. The PGL Server has the advantage of always having the Earth-model data available, thus there is no delay. However, the PGL Server does accrue delays from message passing.

2) Platform independence. A server-based Earth model allows applications to talk to the server across a network; thus the Earth model and the applications can run in a heterogeneous computing environment.

3) Architectural flexibility. A server-based Earth model can support many architectures, including client-server architectures, service-oriented architectures, and distributed-objects architectures. A server-based Earth model can also support parallelism, either with a single server for multiple applications or multiple servers for one or more applications.

Several issues are still being addressed with the PGL Server. These issues include assessing network communication performance between applications and the PGL Server, managing application state in order to manage simultaneous requests from multiple applications, and evaluating parallel distributed architectures that allow the PGL Servers to scale for large problems and large numbers of applications. Architectural issues are also being addressed, including the most appropriate service level for an Earth model. The PGL Server used at its lowest level (atomic components) can require many small messages to be passed. But defining a server at a higher interface level (e.g., the level where event-locations are calculated) might require only a few large messages (a server-based event locator would have the Earth model instantiated with it). Because of the message-passing overhead, a higher-level server could provide performance improvements in many circumstances.
OBJECTIVE

We are developing a server version of the PGL called the PGL Server. The PGL is a shared-object library used in seismic analysis and seismic monitoring to store, access, and manipulate geophysical data representing the Earth. With the PGL Server, we want to improve a delay problem when the PGL is instantiated. In addition, we want to use the PGL Server to investigate advanced computing architectures that might be used in future seismic-monitoring systems.

RESEARCH ACCOMPLISHED

Background

The PGL is a shared-object library used to store, access, and manipulate geophysical data representing the Earth—spatial dimensions (lat, lon, and depth), density, travel time of seismic waves, the difference between observed and actual travel times, etc (Hipp et al., 2005). These “Earth models”—the geophysical datasets—are used in seismic monitoring to identify, locate, and measure the yield of nuclear explosions. There are many Earth models and they are often updated.

The PGL can be used in research to build Earth models and in operations to monitor seismicity in near real time. It works with many Earth models, and it incorporates features that allow it to be readily modified. It is used by many applications, including tools to create and modify Earth models (e.g., KBCIT) and seismic-event locators (e.g., LocOO). It is an object-oriented program, written in C++, designed to be flexible, modifiable, and computationally efficient. The PGL contains approximately 250,000 lines of code, of which approximately 150,000 lines are source code. It has almost 200 classes, with several thousand public methods.

Currently, an application must instantiate the PGL directly. This situation requires that the PGL be on the same machine as the application, and that the application be in the same language as the PGL (although applications written in Java can also access the PGL through special Java Native Interface calls). This situation also typically necessitates a lengthy startup delay while the PGL reads the Earth model dataset and constructs the appropriate objects. This “startup problem” can occur frequently if the application uses the PGL in a repetitive start/stop execution cycle.

The PGL Server was initially conceived as a means of circumventing the startup problem. A PGL Server runs all of the time; thus, the Earth model is always loaded and the objects always instantiated. An application—or several applications—can access the PGL immediately, whether the application is in constant use or not. Figure 1 shows the fundamentally different way that the PGL and the PGL Server can be used.

Figure 1. Comparison of current the PGL implementation with a simple the PGL Server implementation.

Although the initial impetus for a server version of the PGL was the startup problem, the PGL Server also offers the opportunity of investigating more advanced architectures that could be used in a seismic-monitoring system. Server versions of software products are seeing widespread use in industry, for example enterprise databases, which are often used in seismic monitoring, are typically server-based products. Architectures built on server-based products
(e.g., service-oriented architectures [SOA] and web services [SOA over the Internet]), are of increasing interest (Binstock, 2005; Reuters, 2006). These advanced architectures are being developed because they offer advantages in cost, reliability, upgradeability, and performance. As a component within an advanced architecture, the PGL Server could offer similar benefits to seismic monitoring.

1. Cost: the PGL Server can run on a variety of platforms, allowing use of commercial-off-the-shelf (COTS) hardware. Also, applications can be located on the same or different machines. The ability to operate in a heterogeneous computing environment not only allows reduction in capital costs, but protects against obsolescence.

2. Availability, reliability, and maintainability. Multiple PGL Servers could be implemented redundantly so that if one fails, the job can be completed by another. Such a configuration could also allow a machine running the PGL Server to be disconnected from the network without failure of the system, thus simplifying maintenance.

3. Upgradeability: Legacy code is difficult to upgrade for two reasons. First, it usually requires single-supplier hardware. Second, it usually requires a transition that cannot be easily performed within the existing system. The PGL Server is platform independent. And multiple PGL Servers can be readily deployed, including upgraded versions, so that transitions can be done easily.

4. Design flexibility: As a stand-alone component with access through an application programming interface (API), the PGL Server can fit into many system designs, including client-server, distributed-objects, and SOA. This flexibility is important because many applications have need of an Earth model, and these applications might have different design requirements; therefore, the PGL Server would not interfere with design decisions. In this manner the PGL Server also allows ease of upgrading.

5. Data and parameter control: The separate, stand-alone nature of the PGL Server allows different applications to have the same or different datasets or parameters.

6. Performance. Not only could the PGL Server fix the startup problem, but increased speedup and higher throughput could be achieved by implementing multiple distributed PGL Servers or by having a PGL Server running on a faster machine. Different configurations are possible, as discussed later.

Several issues need to be addressed before an efficient and effective PGL Server can be realized. These issues include understanding (1) the design of a PGL Server, (2) the delays incurred from message passing, (3) how a PGL Server should keep track of multiple client applications, (4) what is the most appropriate service level for an Earth model, and (5) how to best implement parallelism. These issues are interrelated. We have conducted and continue to conduct experiments to examine these issues. This paper presents our work and conclusions.

**Basic Design of the PGL Server**

The PGL Server is a stand-alone, continuously running program. It instantiates one or more Earth models from File Databases (FDB) as specified by clients accessing the models. In addition to the objects instantiated from the FDB, the PGL Server also contains a set of stub classes and methods that match one-to-one with PGL classes and methods. These stubs contain the processes necessary to create, send, receive, and decipher messages. In the case of the version of the PGL Server that we are currently using, the messages are in eXtensible Markup Language (XML) and the message passing is handled by XML-Remote Procedure Call (XML-RPC). The PGL Server is written in C++ and uses an XML-RPC library for C++.

XML-RPC was chosen as the message-passing protocol for this work after we performed a survey of existing distributed-system software. The XML-RPC protocol makes use of two well-established technologies: XML and HTTP. XML-RPC has several desirable properties. In particular, the protocol is lightweight and is specifically designed to emphasize compatibility between different platforms and programming languages. This is important, since we have many Java applications that need to communicate with the C++ PGL library.
When a request is generated, XML-RPC parses the relevant information—such as the name of the procedure, argument types, and argument values—and generates an XML document with all of the necessary information. This document is transmitted to the server using the HTTP protocol. After the server receives the message, it parses the XML document to reconstruct all the information it needs to complete the request. In order to handle complex objects, the PGL Server creates an object map to retain the information on any instantiated object. In this map, the PGL Server saves the object and a unique identifier as its key. This key, in most situations, is provided by the client. Usually the client is the one creating objects (e.g., it can use Java’s toString() method to create a unique key), which it then sends to the server for the object map. This work is all done in the stub classes, so use of XML-RPC is completely transparent to the user, and the user interface to PGL is almost, if not exactly, the same.

A client using the PGL Server can run on the same computer or on a different computer. It can be written in any computer language that has support for the same message passing protocol as the PGL Server. Among other languages, XML-RPC has libraries for C/C++, Java, Lisp, and a number of scripting languages including Perl, Python, PHP, and Tcl (XML-RPC, 2003; Apache, 2006). The client must have a set of stub classes and methods that represent the PGL classes and methods that it uses. These stubs are similar to the PGL Server stubs; they contain the processes to create, send, receive, and decipher messages.

Currently, a version of the PGL Server exists that runs on Sun servers with the Solaris operating system, and clients have been written that run on Sun servers, Linux servers, and personal computers.

Figure 2 contains an illustration of how a PGL Server interacts with the PGL and with client stubs.

Figure 2. Basic design of the PGL Server.

Because the PGL has over 5,000 methods, programming all of the necessary stubs is a daunting task, compounded by the prospect of making stubs for different interfaces (e.g., XML-RPC, sockets, and Java Native Interface [JNI]). The PGL also has several Java-based clients, which interface to PGL in a manner similar to that shown in Figure 2, but using Java Native Interface. For the PGL Server project, we also developed a method for automatically generating the PGL interface stubs. Automatic stub generation consists of three parts: Doxygen (an open-source documentation tool) (van Heesch, 2006), a specialized parser, and the generating programs themselves.

First, the Doxygen program scans each of the PGL source files and produces an XML document summarizing the information in each class: methods, members, arguments, etc. Next, this XML document is processed by a specialized parser that examines each XML document and produces a tree data structure containing the same information as the raw XML, but in a more convenient form. Finally, the tree is processed using a standard pre-order traversal. As it encounters each new class, the code-generating program creates a new source file, and applies its rules to create XML-RPC interfaces to each of the PGL functions.
As mentioned previously, the XML-RPC protocol has no innate method of dealing with complex objects. The PGL library is object-based, so this is an obstacle to fully implementing the new system. Our solution to this problem is to create all objects in pairs, with one object created on the server-side, and one on the client-side. The server-side object is the “real” copy that maintains its state and actually executes any RPC requests. The client-side object is only a stub, and is used only to supply an interface from the client to the server. The two objects are linked by a common handle, created during the construction process. When an RPC request involves an object, the client will collect the appropriate stub object and extract the handle linking it to a server-side object. The handle is sent with the other data as part of an XML-RPC request. When the server unpacks the handle, it uses a hash-table data structure to look up the corresponding “real” object. This server object performs any requested work, saves its state, and is then re-hashed until it is needed again. This approach does require more sophisticated methods to manage object life-cycles, so that objects are correctly discarded when their useful lives are over, but this functionality is already part of the PGL, so we do not have to manage object life-cycles explicitly.

**Delays Incurred by Message Passing**

One of the potential problems with the PGL Server (and indeed all server-based software and SOAs) is the overhead associated with message passing. We are interested in understanding the best method for implementing message passing (a lower-level technique, such as sockets, or a higher-level system, such as XML-RPC). Delays incurred by message passing can also be mitigated by choosing the appropriate level of service of an Earth model. This issue is discussed below.

Important variables in this problem are the format of the message, size of the message, the compression of the message, and the platforms that the client and PGL Server are running on. Preliminary work showed that binary (unformatted) messages without any compression are transferred the most efficiently. Binary messages are shorter and generally require less processing than formatted (ASCII or unicode) messages, although the communicating processes must know how to decipher the message. Our investigation confirmed fewer large messages are more efficient than numerous small messages. This preliminary work also showed that compression only helped with formatted messages, and then only with a low level of compression. Additional preliminary work also showed that Fast Infosets (Sandoz and Pericas-Geertsen, 2005), a binary XML implementation, show promise in speeding message passing under some circumstances.

For the work presented here, we programmed a client in Java that communicates with the PGL Server. The client is based on an original test code for the PGL. The client calls the same methods using the same PGL interface as the original, but it can be running on the same or a different machine. The Java client sends method parameters to the server via XML-RPC, and the C++ PGL Server sends responses back to the client in the same way. The major complication occurs when the original test code passes pointers to large datasets. Pointers have no meaning on a different machine, so what the pointers are pointing to - generally large data vectors - have to be passed back and forth.

To address this complication, the client calls the PGL stub classes (Figure 2). The purpose of these stubs is to execute the PGL method calls via XML-RPC, performing whatever manipulations are necessary to make communication between it and the server as simple as possible. An object map on the server side retains the information on any instantiated object. In this map, the server saves the object and a unique identifier as its key. This key is provided by the client, using Java’s toString() method, which it then sends to the server for the object map. This work is all done in the stub classes, so use of XML-RPC is completely transparent to the user, and the user interface to the PGL is almost if not exactly the same.

The first step of the performance testing is to determine the baseline performance. Of particular importance is the total amount of time spent doing XML-RPC-related processing. The baseline times are used below to adjust the the PGL Server times to determine the additional fraction of time spent on XML-RPC-related processing. To determine the baseline times, the original C++ test code, directly connected to the PGL, was timed performing operations on various sized data vectors. For the test, the PGL was actually exchanging six vectors of doubles of the indicated size between the client and server: two containing interpolatory coordinate information and four containing interpolation results vectors for some arbitrary attributes. The results of the baseline PGL performance testing are given in Figure 3.
The same tests were then conducted with the PGL Server and the Java client. The data messages were passed as uncompressed binary payloads, as the preliminary testing indicated that this type of message would give the best performance. Performance testing was conducted with the PGL Server running on a Sun server. The Java client communicating with the PGL Server was tested running on the same Sun server, a Linux server and a Windows XP workstation. The network connection to the Sun server was 100 Mbps.

The results of the testing are presented in Figure 4. The plot shows the fraction of XML-RPC-related processing time required for various message sizes. The metric shown on the Y-axis was calculated as the time it took to execute with the PGL Server minus the baseline time (Figure 3), normalized by the baseline time: \[ \frac{\text{XML-RPC time} - \text{baseline time}}{\text{baseline time}} \]. The result is the additional fraction of time spent doing XML-RPC related processing, such as parsing and serializing XML documents and sending these over the network.

As shown in Figure 4, in all cases XML-RPC related processes reduce performance of the PGL. This finding is especially true when dealing with short data-vector lengths, with the total run times taking more than twice as much time as the baseline. Note however, that the total run times for these short data-vector lengths are typically small fractions of a second. For larger amounts of data, the performance suffers by only 20 to 40 percent.
These results indicate that there is a penalty associated with using the PGL Server. However, the additional time needed for passing short data vectors is small in absolute terms, and the additional time needed for passing long data vectors is small in relative terms. And these delays are much less than the typical 30-second or more startup time for the PGL with a large Earth-model FDB.

**How the PGL Server Should Keep Track of Multiple Client Applications**

Server software should be able to handle multiple clients. Thus, the software should be able to save the state of each process for the clients that are being served. However, the PGL was not designed with this specific capability as it was originally intended to be bound to a single client for its entire execution lifetime. In the future, however, we plan to modify the PGL Server to support a message queue that tags each client’s requests with a unique identifier. The PGL Server will then use the identifier with an object map to determine various “state” settings associated with a particular client. With this approach the PGL Server will be able to change its state settings, process a request, and store the results for each client that it is serving. The client will be able to request these tasks be performed in separate asynchronous functional calls, if desired, without having to worry about other clients modifying the server’s state and thus producing a result that is different than the original request submitted by the client.

**The Most Appropriate Service Level for an Earth Model**

An outstanding architectural issue is whether the most appropriate service level for an Earth model is the PGL Server. Consider that the PGL Server, used at its lowest level (atomic components), can require many small messages to be passed. But defining a server at a higher interface level (e.g., the level where event-locations are calculated), might require only a few large messages. In this case, a server-based event locator would have the PGL and the Earth model instantiated with it. Because of the message-passing overhead, a higher-level server could provide performance improvements in many circumstances. This architectural issue is illustrated in Figure 5.

![Figure 5. Comparison of architectures at the PGL/Earth model level and the event-locator level.](image)

To address this issue, we have programmed a server version of the event-locator LocOO (Ballard, 2003). LocOO takes as input data for one event and (based on a given FDB) returns the location. LocOO calculates locations by repeated successive queries to the PGL, continually refining its estimate of the event’s hypocenter and epicenter and the error volumes associated with these locations. Many requests for small collections of data are made. The version of LocOO that is typically used instantiates its own copy of the PGL to access Earth-model data, so these numerous requests to the PGL carry little overhead. LocOO can also use the PGL Server, but when it does the numerous message-passings that occur cause substantial overhead.
Contrast this situation with LocOO Server. An instance of the PGL is created within LocOO Server when LocOO Server is started. LocOO Server takes the same input data as LocOO. LocOO Server then calculates the location by repeated successive queries to the PGL, but because the PGL is bound directly to LocOO Server on the same machine, the communication overhead is minimal.

We are currently defining tests to quantify the differences between using the PGL Server and LocOO Server in typical location problems. We are also interested in looking at implementing parallelism in the PGL Server and LocOO Server (see the next section), because LocOO is often used to locate or relocate hundreds of events. Location, however, is only one specific problem where Earth-model data are needed. Earth models are also used in phase identification, magnitude, discrimination of explosions from earthquakes, etc. It is possible that a next-generation data center for seismic event monitoring could have Earth models available at several different service levels within the processing system.

How to Best Implement Parallelism

Parallelism, the execution of two or more similar processes concurrently, is one technique for improving processing speed and achieving higher processing throughput in a system. It is possible that next-generation data centers for seismic event monitoring will have to deal with many difficult processing issues (e.g., 3D Earth models). At this time, we have only dealt with parallelism primarily at a theoretical level. However, the PGL Server and other server-based components of a seismic event monitoring system lend themselves immediately to parallelism. Here we discuss some ideas concerning parallelism, including running the PGL Server in parallel processes.

There are two ways of implementing parallelism in an Earth-model tool such as the PGL Server: “internal” parallelism, where subprocesses within a single instance of the PGL Server can execute in parallel, and “external” parallelism, where multiple instances of the PGL Server work on the same job in parallel. Examples of these types of parallelism are outlined in Figure 6. Note that in the figure the job being processed is implied within the PGL Server. In both types of parallelism, a single job is split into parts, the parts are worked at the same time, and then the results from each part are collected (the parts of the job cannot be dependent on one another).

Internal parallelism is the type of parallelism that is usually envisioned. Several software systems have been developed to implement this type of parallelism, including Message Passing Interface (MPI) and Parallel Virtual Machine (PVM). Although not strictly necessary, this type of parallelism tends to work best on multi-processor computers or a network of homogeneous computers with exotic network hardware. We believe there are processes that the PGL performs that are amenable to distributed parallel processing (e.g., loop oriented numerical processes such as matrix solvers). In addition to implementing MPI parallelism within the PGL for multiprocessor machines, we are investigating how to implement the PGL Server parallelism using a network of heterogeneous computers and standard gigabit Ethernet hardware.

More germane to this paper is how to implement parallelism using multiple instances of the PGL Server. Two basic questions arise: (1) Would there be an improvement in speedup or throughput; and, (2) how would multiple PGL Servers be controlled? This second question includes issues such as how would the job be divided among multiple servers, how would load balancing be performed, how many servers would be running, and how would errors be handled (e.g., fault tolerance). Answering the second question involves theoretical work and creating prototypes of controllers. In our initial approach, each the PGL Server instance has an agent that communicates with agents at each client to manage these issues. This work is in the formative stages, and we will only address the first question here.

Using the Condor batch-job-farming software (Condor, 2006), we investigated the extent of the speedup afforded by using multiple instances of the PGL on different processors. The problem was to visualize a travel-time model by using kriging to interpolate on a grid of given resolution (e.g., two degree spacing over some extent). By varying the grid resolution and the number of processors, it was possible to investigate performance improvements afforded by using Condor to facilitate parallel processing with the PGL. In this particular test with the interpolation of grid points was split between 9 processors, we see parallel performance bettering single processor performance at a grid resolution of 0.5 degrees (approximately 2,000 total grid points in this test). The performance improvement approaches linear speedup when the grid resolution is 0.025 degrees (approximately 750,000 total grid points in this test).
test). These results are shown in Figure 7. Thus, to answer the first question, we have shown that multiple instances of the PGL (or PGL Server) can improve speedup under certain circumstances within a seismic monitoring system.

Figure 6. Potential ways of implementing parallelism with the PGL Server.
CONCLUSIONS AND RECOMMENDATIONS

We currently have a version of the PGL Server that uses XML-RPC for communications. The PGL Server can be used to solve the PGL startup problem. In the process of developing this version of PGL Server, we are also developing an automatic method for generating the PGL Server stub libraries (e.g., using XML-RPC, sockets, JNI). We also developed a server version—LocOO Server—for the event-location program LocOO.

With these programs, we investigated issues associated with advanced system architectures, in particular SOAs. We conducted performance testing that indicated the best way to conduct message-passing and the best form of the messages. Using LocOO Server, we have begun to investigate the structure of different architectures, including whether it is better to have a service level at the Earth model or at the locator. This preliminary work suggests that different processes might require different service levels. In anticipation of more computation-intensive operations being required for seismic monitoring in the future, we began to study ways of parallelizing Earth models. A preliminary finding from this work is that many PGL calculations scale linearly with the number of PGL instances (e.g., PGL Servers) running.

Much of this work is in the initial stages. We would like to implement a method for saving state within the PGL Server. We would like to better understand how to manage multiple PGL Servers and how to best implement parallelism with the PGL Server. Additionally, we would like to test the concepts that we are developing on real-world problems.

In performing this work, we understand that architecture development is not accomplished by introspection, and it is not just a paper study. Development of an advanced architecture requires construction of prototypes and testing the behaviors of these prototypes in realistic situations. Ultimately, we believe that the knowledge that we have gained from this work can be applied to the design of an advanced-architecture for a seismic monitoring system.

ACKNOWLEDGEMENTS

We would like to thank David Gallegos, Mark Harris, and Chris Young; without their support and guidance this work would have never been accomplished.

REFERENCES


