Editor: Geoffrey Fox, gcf@indiana.edu

SOFTWARE DEVELOPMENT AROUND A MILLISECOND

By Geoffrey Fox

IN THIS INSTALLMENT, I CONSIDER SOFTWARE-DEVELOPMENT METHODOLOGIES, EMPHASIZING THOSE RELEVANT TO LARGE-SCALE SCIENTIFIC COMPUTING. MANY PROJECTS AIM TO IMPROVE THE SCIENTIFIC COMPUTING ENVIRONMENT, INCLUDING scientific code preparation, execution, and debugging. But a scientific programming environment must address some of the key features that differentiate it from commodity or business computing for which many good tools exist. These special features include

- floating-point arithmetic,
- performance and support for scalable parallel implementations,
- specific scientific data structures and utilities (such as mesh generation), and
- integrating simulations with distributed data.

We have discussed the last area several times in this department because it is a major feature that the Grid brings to scientific computing. This article focuses on it again, discussing technologies needed for software integration, which is part of the overall struggle to develop self-contained modules that link together to create larger applications. Subroutines, methods, libraries, objects, components, distributed objects, and services are different ways of packaging software for greater reusability. Here, we will look at the service model for software modules.

Criteria for Choosing Software

We must address the problem of developing programming environments from two points of view. First, we must identify requirements and then—as best as we can—identify the best architectures and technologies to address them. Then two difficult but more important issues surface: What is the lifecycle model? How do you maintain the environment and update it as the underlying computing infrastructure driven by Moore's law marches on with major architecture and preferred vendor changes on a few years' timescale?

The lifecycle issue is particularly important in areas such as scientific and high-performance computing where the market is not large enough to support all the nifty capabilities that we need. This affects hardware and software, but I focus on the latter here.

Though you might have a great idea for a new parallel-computing tool and obtain funds to develop an initial and, perhaps, highly successful prototype, ongoing funds to refine and maintain the system often are much harder to obtain unless you can find a sustainable commercial market for it. Thus, you should look at existing commercial approaches and use them wherever possible. Sometime, this means choosing a less-than-optimal solution, but a supportable almost-good-enough solution typically is preferable to an optimal-though-unsustainable one.

For example, suppose your latest parallel-computing software tool requires an application developer to modify or annotate his or her code. Let's say he or she does it, and the new code version works well. But when the next-generation hardware is installed, the tool isn't upgraded, and thus the code won't run at all. A developer then must start again on its parallelization. For this reason, I usually recommend using the rather painful explicit message-passing approach to parallelization. It is supported by a sustainable technology (message-passing interface; MPI; www.mpi-forum.org/) even though there are higher-level approaches such as openMP (www.openmp.org) and high-performance Fortran (HPF; www.crpc.rice.edu/hpff/), which offer more productive but not clearly sustainable portable programming models.

My millisecond rule says that you should use commodity (Internet, peer-to-peer, or Grid) programming models for those parts of your programming that can tolerate millisecond or longer delays, or, more precisely, latencies. To see how this fits, first, let's revisit the Grid, or Web service, programming model.

Grid and Web Service Programming

I'll frame this discussion with the Grid programming model that I covered

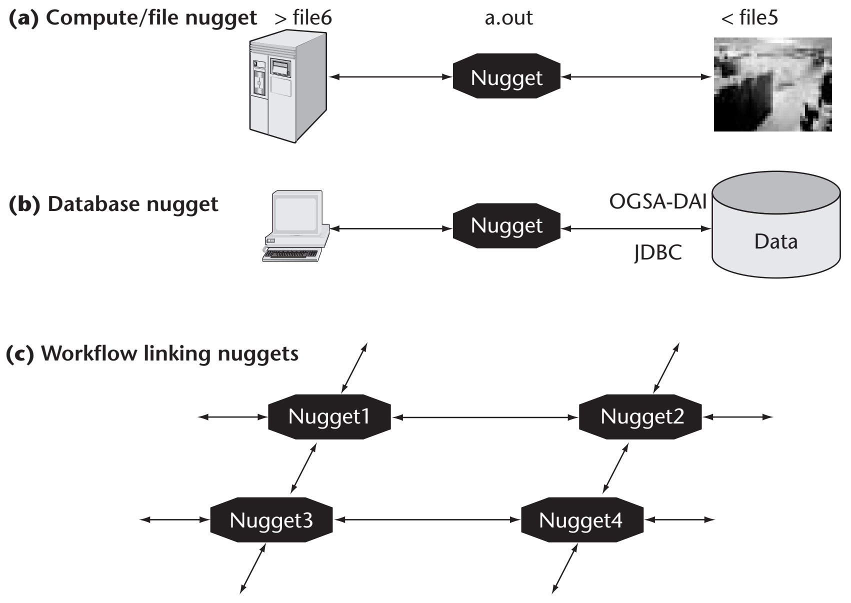

Figure 1. Two-level Grid programming model exposing the nuggets programmed at the Grid level. Nuggets can be software modules or any form of electronic resource. (a) File and program nuggets capturing a simple job reading and writing a file, (b) a nugget representing a database and its access, and (c) a workflow linking four nuggets, with input and output data streams between them.

briefly in the March/April 2003 Web Computing department ("Grid Computing Environments," pp. 69–72). The article described a two-level programming model for the Grid and, more generally, the Internet (see Figure 1). At the lower level, microscopic software is written in familiar languages such as Fortran, Java, and C++ to control individual CPUs. We assume that these languages generate nuggets, or code modules; associating these nuggets with a single resource is what traditional programming addresses. Practical examples of the architecture Figure 1 illustrates could be an SQL interface to a database, a parallel image-processing algorithm, or a finite-element solver.

After creation, this nugget with a well-understood (but, of course, still unsolved) programming model must be augmented for the Grid by integrating the distributed nuggets together into a complete executable exemplified in Figure 1c. For the Internet, nuggets are Web services; for the Grid, they are Grid services; for Corba, nuggets are distributed objects; and for Java, they are Java objects communicating via remote method invocation (RMI).

In the September/October 2002 installment of this department ("Message Passing: From Parallel Computing to the Grid," pp. 70–73), I described how the Grid programming model and support technologies are very different from those for parallel computing, which, at first glance, seem to be tackling similar nugget-integration problems. Thus, if you are simulating a physical system, you typically divide it into regions and simulate each region in a different parallel-computer node. The software simulating each region becomes the nuggets, and it must intercommunicate as Figure 1c shows.

Parallel computing is characterized by relatively small-sized but very frequent messaging among synchronized processing nodes. This requires high bandwidths but, more importantly, low latency. For a typical node with gigaflop-per-second performance, you might use gigabyte-per-second internode communication bandwidth and 0.01-ms latency.

In a corresponding Grid situation, your nodes are more loosely coupled, and you might expect a bandwidth similar to the parallel-computing case but with a much higher latency. As discussed in the September/October 2002 article, network transit times (hundreds of milliseconds for transcontinental links) determine Grid latencies. Of course, different network performance and programming models reflect different requirements for those applications suitable for parallel and distributed systems. Typically, parallel computers support nuggets coming from a single large, application domain with data decomposition. Grid systems support loosely coupled components and functional decompositions of problems.

A good example of a loosely coupled case is the analysis of billions of events from accelerator collisions. Each event can be processed independently, needing only access to the raw-event data and then accumulation into common statistical measures to link the nuggets together.

Functional decompositions, such as satellite-data image processing, often are latency-insensitive because information can be pipelined through different filters as it travels from source to final store. These latency-insensitive applications can cope with the hundreds or thousands of milliseconds' communication latencies implied by a Grid implementation.

However, we are not studying this issue here. Rather, we assume that for such Grid applications, the synergy with the commercial Web service area will ensure the development of excellent programming tools and runtime environments. Already, the Organization for the Advancement of Structured Information Standards (OASIS; www.oasis-open.org) and the World Wide Web Consortium (W3C; www.w3.org) are developing standards such as the Business Process Execution Language (BPEL) and Web Service Choreography, respectively. These involve the commercial heavy hitters, including IBM, Microsoft, Oracle, and Sun. This workflow area is critical for commercial applications, and a recent e-Science Workflow Services workshop (www.nesc.ac.uk/action/esi/contribution.cfm?Title=303) surveyed their use in science.

When to Use Grid Programming Tools

Though Grid applications are characterized by delays of hundreds of milliseconds, the delay time includes some typical inter-enterprise, or global, network delays. The actual software overhead is characteristic of single-CPU-network processing time, process-switching times, responses of typical servlet-based Java servers, or, perhaps, thread- or process-invocation times. These times are more like a millisecond for a typical server response, thus, you can use Grid programming tools in any application in which an approximately 1-ms delay is acceptable. These uses need not be network based but can include linking software components within a single CPU. We can identify several application classes in which a millisecond delay is acceptable.

As I said earlier, the basic distributed service integration typical of Grid programming includes the linkage of multiple data and computing components. The data category includes streaming sensors, file systems, and databases. The computing category includes on-the-fly filtering and large-scale simulations, with, perhaps, real-time Grid data assimilation.

This class of problem is similar to application integration in enterprise computing and multidisciplinary applications in scientific computing. The former class could include human resources integration, marketing, and customer sales and satisfaction databases. Figure 2 illustrates examples of the latter class—needed in stealth-aircraft design—by the integration of fluid structural and electromagnetic signature simulations. However, essentially all fields of computational science need some sort of code coupling for advanced applications. The coupling could be loose, as in "run program A and feed results into program B," or close, in which two or more programs exchange data interactively throughout the simulation.

A loose-coupling case always can use the Grid programming model. If the quantity of data exchanged is large enough to mask the latency, a close-coupling case also can use these commodity technologies. This type of integration often appears in so-called problem-solving environments (PSEs), which offer domain-specific portals to access needed services. PSEs need some sort of software bus to integrate the services and applications they control. I believe that the Grid programming model will become the technology of choice and replace today's often-used Java or Python coupling.

There is another, very different, millisecond-tolerant application class in which you can use the Web service approach to build interactive applications. This class exploits the observation that people don't do very much in a millisecond, so you can use these technologies to build traditional desktop or client applications such as Word or PowerPoint. You can put the application's core into a Web service and use messaging to transport user input (such as mouse and keyboard events) as messages from the user to the service. This is the so-called model–view–control (MVC) paradigm for applications, which appeared in our discussion of Grid computing environments and portlets in the March/April 2003 installment (see Figure 3).

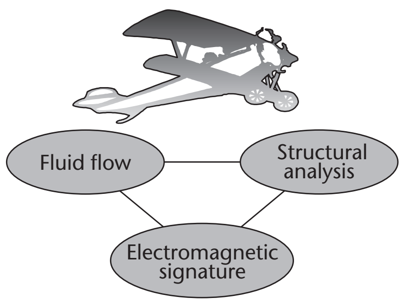

Figure 2. Designing a stealth biplane with coupled simulations and structural properties. We can achieve this by simulating the three characteristics shown and optimizing system parameters affecting them all.

MVC has been around a long time, but it's not usually implemented with explicit messaging. Building integrated desktop applications like our current ones provides higher performance than the Grid programming model, but as Moore's law continues to make computers faster, the Grid's modularity and power becomes more attractive. The Grid approach also lets us easily retarget applications to different client platforms—Windows, Linux, MacOS, PDA, Kiosk, or another interface supporting access for the disabled.

Figure 4 shows another obvious application to support collaboration with a single model: Web services driving multiple views, or clients. I think these ideas could be a compelling software architecture for a new generation of clients we might see developed to support Linux or cell-phone clients. Note that the architecture in Figures 3 and 4 works wherever clients and Web services are placed—they can be separated by the hundreds of milliseconds characteristic of continental networks, for example. However, they only give acceptable interactive performance when the model and view are run on the same machine or nearby machines.

Other commodity technologies—

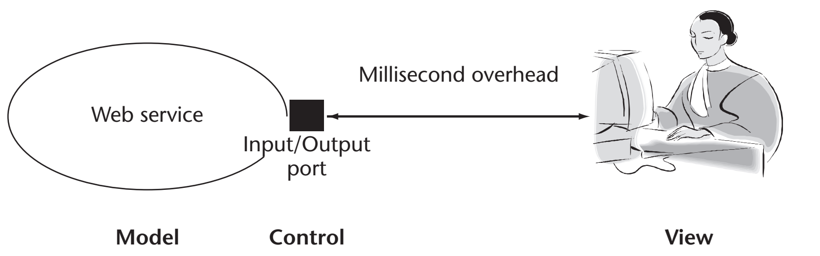

Figure 3. A model–view–control application using explicit messaging for control. The Web service holds the logic of a desktop application, and communicates by explicit messages with a user-interface module, which handles display rendering and the catching and transferring of user input and mouse and keyboard events to the Web service.

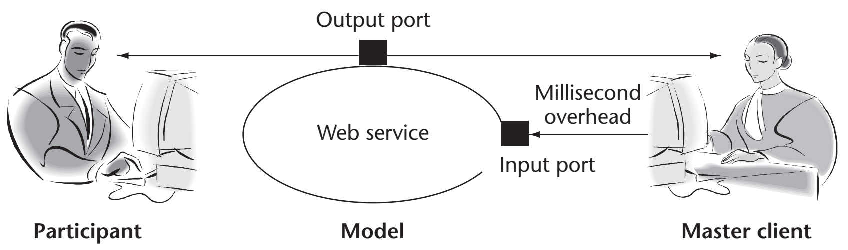

Figure 4. Collaboration using a shared output port of a model–view–control message-based application. This extends the scenario of Figure 3 by adding two users. As before, one sends user input to the Web service and receives information to generate the display. These correspond to input and output user-facing ports on the Web service. This figure also shows a second collaborative user who shares the output user-facing messages, and receives synchronous display updates from the first user.

scripting languages such as Python and Perl—can tackle some of the applications I just discussed. You can expect commercial software support to be good for both Grid and scripting approaches. In fact, because scripting tends to use method calls and not explicit messaging, the intrinsic overhead can be substantially lower than the millisecond Grid-case guideline. We can counter with two arguments in favor of the Grid model: first, message-based interfaces give greater modularity than method calls, which allow side effects, and, second, the message-based model lets you implement Grid or Web services if you need them. Thus, I suggest that perhaps, three different regimes are characterized by a module's typical transaction time:

- a millisecond or greater—Grid- and Web-service-based technologies for linking modules,
- 10 microseconds or less—high-performances technologies such as MPI, and
- 10 to 1,000 microseconds—scripting and other object-based environments.

Further refinements of the millisecond rule can help refine the third regime; I will address this at another time.

Inevitable computer hardware improvements will change the millisecond rule by shortening it, which could be very important in enabling new architectures and message-based applications. For example, the current millisecond rule was a "10-millisecond rule" five to 10 years ago. Then, this overhead was comparable to user interface times and thus made message-based user interfaces difficult. As the overhead continues to diminish, we can expect other currently infeasible applications to become possible. Another area previously mentioned and ready for future development is extending the rule to identify the region in which scripting languages can interpolate between message-based and high-performance architectures. CiSE

Geoffrey Fox is director of the Community Grids Lab at Indiana University. Contact him at gcf@indiana.edu; www.communitygrids.iu.edu/IC2.html.

Submissions: Send one PDF copy of articles and/or proposals to Francis Sullivan, Editor in Chief, fran@super.org. Submissions should not exceed 6,000 words and 15 references. All submissions are subject to editing for clarity, style, and space.

Editorial: Unless otherwise stated, bylined articles and departments, as well as product and service descriptions, reflect the author's or firm's opinion. Inclusion in *CiSE* does not necessarily constitute endorsement by the IEEE, the AIP, or the IEEE Computer Society.

Circulation: *Computing in Science & Engineering* (ISSN 1521-9615) is published bimonthly by the AIP and the IEEE Computer Society. IEEE Headquarters, Three Park Ave., 17th Floor, New York, NY 10016-5997; IEEE Computer Society Publications Office, 10662 Los Vaqueros Circle, PO Box 3014, Los Alamitos, CA 90720-1314, phone +1 714 821 8380; IEEE Computer Society Headquarters, 1730 Massachusetts Ave. NW, Washington, DC 20036-1903; AIP Circulation and Fulfillment Department, 1NO1, 2 Huntington Quadrangle, Melville, NY 11747-4502. Annual subscription rates for 2004: $42 for Computer Society members (print only) and $48 for AIP society members (print plus online). For more information on other subscription prices, see www.computer.org/subscribe or http://ojps.aip.org/cise/subscrib.html. Back issues cost $20 for members, $96 for nonmembers.

Postmaster: Send undelivered copies and address changes to Circulation Dept., *Computing in Science & Engineering*, PO Box 3014, Los Alamitos, CA 90720-1314. Periodicals postage paid at New York, NY, and at additional mailing offices. Canadian GST #125634188. Canada Post Publications Mail Agreement Number 0605298. Printed in the USA.