Software Development around a Millisecond

Geoffrey Fox

Introduction

In this column we consider software development methodologies with some emphasis on those relevant for large scale scientific computing. There have been many projects aimed at improving the environment for scientific computing. These address the preparation and execution and debugging of scientific code. A scientific programming environment must address some of the key features that differentiate this from "commodity" or "business" computing for which many good tools exist. These special features include

1) Importance of floating point arithmetic
2) Importance of performance and need to support scalable parallel implementations
3) Very particular scientific data structures and utilities (such as mesh generation)
4) Integration of simulations with distributed data

The last area has been discussed extensively in this column as it is a major feature brought by the Grid to scientific computing. This article focuses on it again and generalizes to a discussion of technologies needed to integrate modules together. From the beginning of computing, we have struggled to identify the best way to develop self contained modules that then can be linked together to make bigger applications. Subroutines, methods, libraries, objects, components, distributed objects, services are all to some extent different ways of packaging software for greater re-usability. Here we are looking at the service model for software modules.

Criteria for choosing Software

In developing programming environments one needs to address the problem from two points of view. Firstly one needs to identify requirements and then as best one can identify the very best architectures and technologies to address them. Then one has the harder but perhaps more important issue – what is the "life-cycle model". How do we maintain the environment and update it as the underlying computing infrastructure underneath it marches on with major architecture and preferred vendor changing on the few year time scale driven by Moore's law for the performance improvement of the hardware. The "life-cycle" issue is particularly important in areas like scientific or high performance computing where the market is not large enough to support all the nifty capabilities that one needs. This affects both hardware and software but here we focus on the latter. One may 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 your system are often much harder to obtain unless one can find a sustainable commercial market for it. For this reason, once should look at existing commercial approaches and use them wherever possible. This will sometimes mean choosing a less than optimal solution but we argue that a supportable "almost good enough" solution is typically preferable to an optimal unsustainable system. Suppose your latest parallel computing software tool requires the application developer to modify or annotate their code. Maybe they do this and it works well but when the next generation hardware is

installed, the tool is not upgraded. The user must then start again on the parallelization. For this reason I tend to recommend that one use the rather painful explicit message passing approach to parallelization. It is supported by a sustainable technology, MPI even though there are higher level approaches such as openMP and HPF which offer more productive but not clearly sustainable portable programming models.

The Rule of the Millisecond

In this article we explore the "rule of the millisecond" which says that one should use commodity (Internet, peer-to-peer or Grid) programming models for those parts of one's programming that can tolerate millisecond or longer delays. First let us revisit the Grid or Web Service programming model.

Grid and Web Service Programming

We phrase our discussion in terms of the model of Grid programming that we briefly covered in the March/April 2003 column. This described a two-level programming model for the Grid and more generally the Internet shown in fig.1. At the lower level, there is "microscopic" software controlling individual CPU's and written in familiar languages like Fortran, Java and C++ . We assume that these languages generate "nuggets" or code modules and it is making these nuggets associated with a single resource that "traditional" programming addresses. To give examples illustrated in fig.1, the nugget could be the SQL interface to a database, a parallel image processing algorithm or a finite element solver. This well understood (but of course still unsolved) "nugget programming" must be augmented for the Grid by the integration of the distributed nuggets together into a complete "executable" exemplified in fig. 1(c). 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 by RMI (Remote Method Invocation). We described in the September/October 2002 column, how the Grid programming model and support technologies are very different from those for parallel computing which at first glance must tackle similar problems in integrating nuggets together. Thus if one is simulating a physical system, one typically divides into regions with each region simulated in a different node of the parallel computer. The software simulating each region becomes the nuggets and they must communicate together in a fashion analogous to fig. 1(c). Parallel computing is characterised by relatively small very frequent messages operating between synchronized processing nodes. This requires high bandwidths but more importantly low

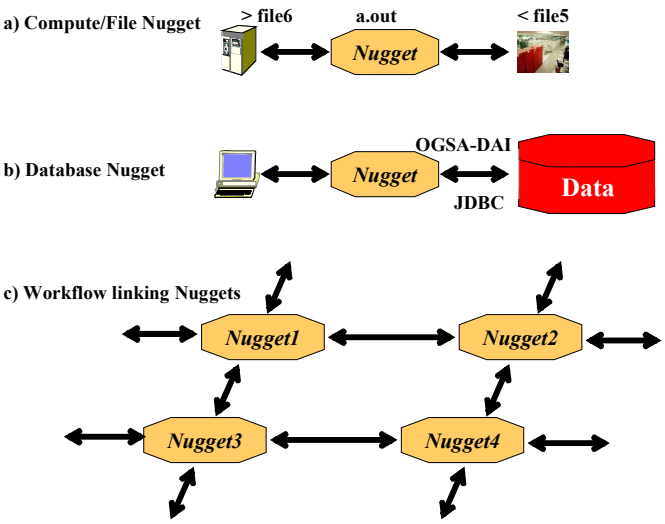

Fig. 1: Two-level Grid Programming Model

latency. For a typical node with gigaflop per second performance, one might use gigabyte per second internode communication bandwidth and 0.01 millisecond latency. In the corresponding Grid situation, one's nodes are more loosely coupled and one might expect similar bandwidth to the parallel computing case but much higher latency. As discussed in the September/October 2002 column the Grid latencies are determined by network transit times which are 100's of milliseconds for transcontinental links. Of course the different network performance and the different programming models is reflected in different requirements for those applications suitable for parallel and distributed systems. Parallel computers support nuggets coming typically from domain or data decomposition of single large application. Grid systems support loosely coupled components and functional decompositions of problems. The loosely coupled case is illustrated by the analysis of billions of events from accelerator collisions where each event can be processed independently with only access to the raw event data and accumulation into common statistical measures linking the nuggets together. Functional decompositions such as image processing of satellite data are often latency insensitive as information can be pipelined through different filters as it travels from the 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 insure the development of excellent programming tools and runtime. Already standards like BPEL (Business Process Execution Language) and Web Service Choreography are being developed in OASIS and W3C respectively; these involve the commercial heavy hitters including IBM, Microsoft, Oracle and Sun. This is workflow area is critical for commercial applications and their use in science was for example surveyed in a recent workshop at Edinburgh http://www.nesc.ac.uk/action/esi/contribution.cfm?Title=303.

When can Grid Programming Tools be used?

Now the Grid applications were characterized by hundreds of millisecond delays. However this represents some typical inter-enterprise or global network delay. The actual overhead in the software is characteristic of single CPU network processing times, process switching times, response of a typical servlet-based Java server or perhaps thread or process invocation times. These are more like a millisecond for a typical network response. Thus Grid programming tools can be used in any application where around a millisecond delay is acceptable. These uses need not be network based but include the linkage of software components within a single CPU. We can identify several application classes where the millisecond delay is acceptable.

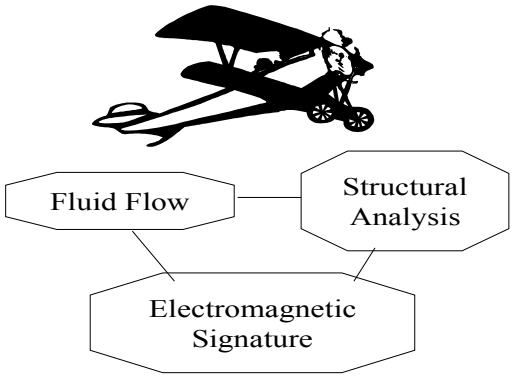

Fig. 2: Designing a Stealth Biplane with coupled simulations

The basic distributed service integration typical of Grid programming includes as we stated in the introduction the linkage of multiple data and computing components. The

data category includes streaming sensors, file systems, and databases. The computing category include both on-the-fly filtering and large scale simulations with perhaps real time assimilation of data from the Grid. This style of problem is similar to "application integration" in enterprise computing and multidisciplinary applications in scientific computing. The former class could include the integration of human resources, marketing and customer sales and satisfaction databases. The latter class is exemplified by the integration of fluid structural and electromagnetic signature simulations shown in fig. 2 and needed in the design of stealth aircraft. However essentially all fields of computational science need some sort of this "code coupling" for advanced applications. This coupling could be loose as in "run program A" and feed results into program B or "close" where two or more programs A and B exchange data interactively throughout the simulation. Loose coupling can always use the Grid programming model and if the amount of data exchanged is large enough to mask the latency, the close coupling case can also use these commodity technologies. This type of integration is used in most so-called "problem solving environments" or PSE's that offer domain specific portals to access needed services. PSE's need some sort of software bus to integrate the services and applications they control. We suggest the Grid Programming model will become the technology of choice here and replace either Java or Python coupling as used often today.

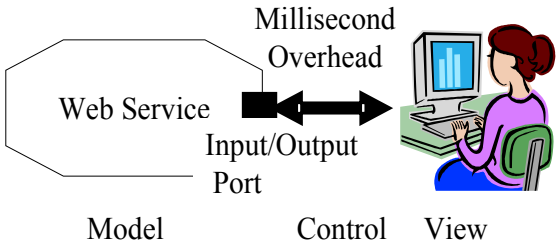

Fig.3: MVC Application built using explicit messaging for control

There is another very different class of millisecond tolerant applications where one can use the Web service approach to building interactive applications. These exploit the observation that people don't do very much in a millisecond and so one can use these technologies to build "traditional desktop or client" applications such as Word or PowerPoint using this approach. One can put the "meat" of the application 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 MVC (model-view-control) paradigm for applications and was used in our discussion of Grid Computing Environments and portlets in the March/April 2003 column. MVC is very old but typically is not implemented with explicit messaging. Building integrated applications like our current desktop applications gives higher performance than the Grid programming model. However computers keep getting faster and the modularity and power of the Grid approach gets more attractive as Moore's law increases our client's

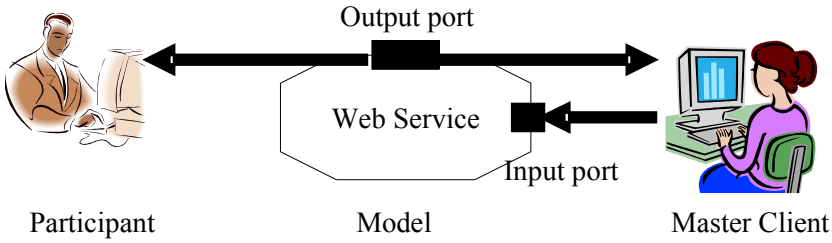

Fig. 4: Collaboration using shared output port of a MVC Message-based Application

performance. The Grid approach allows applications to be easily retargeted to different client platforms – Windows, Linux, MacOS, PDA, Kiosk or an interface supporting access for the disabled. Figure 4 shows another obvious application to support collaboration with a single "model" web service driving multiple "views" or clients. We think these ideas could provide a compelling software architecture for a new generation of clients as we might see developed to support Linux or cellphone clients. Note the architecture of figures 3 and 4 works wherever the client and Web service are placed – they can be separated by the 100's of milliseconds characteristic of continental networks. However they only give acceptable interactive performance if the model and view are run on the same or nearby machines.

There are other commodity technologies like the use of scripting languages such as Python and Perl which can tackle some of the above applications. One can expect support to be good for both Grid and scripting approaches. In fact as scripting tends to use method calls and not explicit messaging, the intrinsic overhead can be substantially lower than our millisecond for the Grid case. We can counter with two arguments in favor of the Grid model – firstly message based interfaces give greater modularity than method calls which allow side effects. Secondly the message based model allows Grid or Web service implementation if that should be desired. Thus we suggest that there are perhaps different regimes characterized by the typical transaction time of a module:

- A millisecond or greater: Grid and Web Service based technologies for linking modules.
- 10 microseconds or less: high performances technologies such as MPI
- 10 to 1000 microseconds: scripting and other object-based environments. Further refinements of the millisecond rule can help to refine this third regime and we will return to it later.

Conclusions

This article has espoused the view that one should use sustainability of software as a major criterion in building software systems. We have introduced and exemplified "the rule of the millisecond" to define cases where commodity Grid and Web service technologies can be used. We gave three application areas -- "conventional MVC" applications, Problem solving Environments and code coupling were these ideas could be applied. We suspect one can derive more refined such rules to clarify better when it is really needed to use specialized and hard to maintain technologies. We also note that computers are still getting faster and so the overhead of commodity technologies is getting less and less relevant in many cases. This should lessen the need for specialized approaches.