Grande Applications and Java

Geoffrey Fox
Indiana University
Computer Science, Informatics and Physics
Community Grid Computing Laboratory,
501 N Morton Suite 224, Bloomington IN 47404
gcf@indiana.edu

Java Grande

Soon after Java became popular for Internet applications, it received attention in many other areas. In fact Java's original thrust was client side "applets" and today its dominant use is in the large servers running enterprises – the "business middleware". The original client side use is of secondary importance. There has been substantial work looking at "Java for high performance computing" (http://www.javagrande.org) and this has been the subject of workshops held annually over the last seven years including the latest event JGI'02 http://charm.cs.uiuc.edu/javagrandeIscope/ held at Seattle from November 3-5 2002. This meeting series joined forces with another (ISCOPE) largely aimed at looking at object-oriented methods in science and engineering with an emphasis on the use of C++. Together they addressed object oriented methods in Grande applications; we use term Grande to describe a very broad class of problems which are "large" in some sense. This can include large massively parallel simulations as well as scalable approaches to large scale distributed systems – an area we have discussed earlier as part of Grid and Peer-to-Peer computing columns.

Several important research topics are included in the JGI (Java Grande-ISCOPE) agenda divided into three broad categories -- "Java on the Node", object-oriented scientific computing and distributed systems. In more details, issues are

1. Java for numerical computing with possible language and runtime issues.
2. High performance Java Compilation
3. Object oriented (largely C++ today) libraries
4. Parallel Java and C++ environments
5. Java based distributed computing and Grid environments

Java on the Node

When Java Grande started, the performance penalty for using Java was severe – upto two orders of magnitude slower than equivalent Fortran or C++ codes. Now much progress has been made and the penalty depends on the application but is typically at most a factor of two on scientific applications. NIST has developed a benchmark SCIMARK http://math.nist.gov/scimark2/ which averages several common scientific kernels including the Fast Fourier Transform and matrix algebra. The current top score is over 300 megaflops for the IBM Java VM (Virtual Machine) on an Intel architecture PC. A Java benchmark suite was developed at EPCC and used to compare Java with Fortran and C in http://www.epcc.ed.ac.uk/overview/publications/research_pub/2001_pub/conference/jgflangc

omp_final.ps.gz with very favorable results for Java reported at JGI'01. Much of this improvement has come from better compiler techniques including those reported at JGI meetings. It is interesting to note that Java uses dynamic "Just-in-Time" methods and its compilers typically have rather different structure from those for other familiar languages.

The Java Grande community has discussed in detail (http://math.nist.gov/javanumerics/) the key reasons why Java node performance cannot easily match that of Fortran and C++. These include

1. Java Floating point rules
2. Nature of Java Arrays
3. Overhead of "small objects" as in the "complex data type"
4. Immature scientific libraries
5. Lack of parameterizable types to remove need for each method to be replicated for each argument type (real, double, complex etc.)

Substantial progress has been since 1995 in allowing Java more flexibility in floating point representations. Initially the language insisted that Java should not only run on all computers but always give exactly the same answer. Actually this goal is quite interesting in applications (such as control of medical instruments) where exact reproducibility could be critical. However it is not natural for floating point arithmetic where different CPU's have different rounding characteristics even when operating with basically the same IEEE representation. The Java Grande forum's numerics working group was successful in getting Java's original strict rules relaxed with an optional *strictfp* modifier to enforce the original semantics. Java still cannot take advantage of several floating point accelerations – including the fused multiple-add instruction on some high performance CPU's. However the working group effort to add a *fastfp* modifier to allow this is no longer being pursued. Perhaps current performance is good enough.

Java arrays are built as objects and have nontrivial overhead from this – they are not just a list of sequential entries in memory. Further similar overhead occurs in using Java representations as objects for complex arithmetic or for say a three dimensional velocity vector (or whatever physical structure describes the basic entities in the simulation. Here the solution could involve either language or compiler optimizations (or better both) to remove the object overheads. It is getting harder and harder to change Java as it becomes so extensively used in main stream software systems. Thus most attention has been paid to compiler strategies to recognize and remove overheads. Much progress has been made but much of the advanced compiler technology remains only in research compilers as major vendors see little demand from the scientific community for better performance. The Java Grande community is pursuing "Fortran style arrays" in Java with JSR 83 (Java Specification Request) for such multi-arrays.

The next release of Java should include "generic types" partly addressing the parameterizable type issue listed above. Two other language enhancements of interest to scientific computing are "operator overloading" (so one could for instance add complex objects with the + notation and not with a non-intuitive add method) and "value classes" (objects represented by value and not by reference). These are not likely soon. One interesting recent announcement from Visual Numerics is the availability of a high quality Java version JMSL http://www.vni.com/products/imsl/jmsl.html of their well known IMSL scientific library.

Java performance on the node is reasonable today. Further improvement is possible in areas of importance to computational science and engineering. This requires users to show more interest and demand that new supercomputers offer Java compilers with the capabilities that have been proven in research but not yet deployed.

Parallel Java

Java for parallel computing involves both the node issues discussed above and similar parallelism support to that needed in other languages. There are several activities in message passing (such as Java bindings of MPI), shared memory openMP and other compiler based approaches to parallelism (for the Indiana University work in this area see http://www.hpjava.org). We can discuss these in later columns but in a naïve summary we can say that Java is as good or better than other languages in support for parallelism. Java has excellent communication libraries which are getting better on each release. Further the language is expressive and supports rich parallel constructs. Interesting results were reported at JGI'02 by researchers at Los Alamos and Rice developing large scale scientific applications from scratch in Java and using advanced compiler optimizations. Good performance was obtained for the CartaBlanca http://www.lanl.gov/projects/CartaBlanca/overview.html code for heat transfer and multi-phase flow and the Parsek particle in the cell code. Different coding styles (from "Fortran in curly brackets" to fully exploiting the object-oriented paradigm with fine grain objects) were investigated.

Is this a good idea? Should we lose some performance in exchange for the more powerful object-oriented style and robust software engineering of Java? Is the additional complexity of C++ important or perhaps Microsoft's C# is the answer? C++ and C# should get better performance than Java but so far they have not been extensively adopted in scientific computing.

Distributed and Enterprise Java

Traditionally distributed systems have been the largest topic in Java Grande meetings as they builds on Java's natural integration with the Internet. Much of the research in this area can be thought of as Grid and/or Peer-to-Peer computing which we have discussed in earlier columns. We will of course return to these central areas several times in the future. Here we will discuss briefly an area highlighted in the keynote talk of Pratap Pattnaik of IBM. He stressed the critical importance of robust enterprise systems which are today largely being built in Java. Enterprise architectures are built around (Grid-like) architectures with individual components linked by messaging subsystems. This architecture is replacing the use of monolithic systems built around huge mainframes. Such approaches are perhaps inevitable with growth of distributed enterprises and commodity server systems. However "modularity" and natural distributed support comes at a serious cost in "management". Such enterprise solutions will inevitably grow in size as Moore's law 's continued progress leads to smaller unit systems with increased performance coming from adding more CPU's. This is the Grande Enterprise (or Grid) problem. How can we manage and make robust such a decentralized system of growing size. This challenge requires perhaps new algorithms and new software aimed at building what is called "Autonomic" systems by IBM (http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf). Autonomic computers should sense their environment and respond properly to unexpected and often

erroneous input. They should be tolerant of faults in themselves and others – a characteristic sought perhaps not so successfully through generations of human societies. We expect that as the lower-level technical problems are solved (better node and communication performance for example), autonomic computing issues such as the robustness and management of Grande systems will become new foci of research. Here the different architecture styles of Peer-to-Peer and Grid systems must be refined and melded to address these challenges. Note how the scaling to more and more computers affects differently parallel and distributed computing. In the parallel case, the scaling usually involves the SPMD (the same program on each node) case and so the key problem is maintaining performance as the number of nodes scales up. In distributed computing, one typically scales heterogeneously with each node running distinct codes; individual node or communication channel is important but not so critical as in the parallel case. Rather the key scaling and performance problems stem from the heterogeneity and unpredictability of the system.