1997

Future Scientific Software Systems

John R. Rice
Purdue University, jrr@cs.purdue.edu

Report Number:
97-007

FUTURE SCIENTIFIC SOFTWARE SYSTEMS

John R. Rice

Department of Computer Sciences
Purdue University
West Lafayette. IN 47907

CSD-TR 97-007
January 1997

FUTURE SCIENTIFIC SOFTWARE SYSTEMS

John R. Rice
Department of Computer Sciences
Purdue University

January 24, 1997

The Applications

There is an enormous array of scientific applications to be solved. As our computers, algorithms, and software improve some of them come into the feasible range, are discovered, and attempted. The size, complexity, and difficulty of applications that we can see now greatly exceed our solution capabilities for the next 50–100 years. Consider animals; each living cell has thousands of different kinds of chemical *gizmos* (molecules) that make it work. Some are simple, some are extremely complex. There are more of these in a single cell than there are people in the U.S. And there are more cells in an average animal than there are people in the U.S. I conclude that the accurate simulation of animals is beyond thinking about at this time, it involves simulating the behavior and interactions of perhaps 100 quadrillion (10^{16}) gizmos.

Another such application is the ab initio computation of material properties. One starts with models of atoms and their interactions. With Teraflop computers one may be able to handle a few thousand atoms. But, except for a few crystalline materials, there is a complex microstructure with gaps, impurities, and micro-crystals with different sizes and orientations. On a larger scale there is a granular structure that is equally important; the micro-grains coalesce into grains or fibers which then coalesce to form "materials" as we normally think of them. The nature of these structures (and the material properties) depends critically on how the materials are made. It is clear that we cannot, in the 21st Century, simulate all the atoms in an interesting piece of material; we probably will not even be able to simulate all the micro-grains.

The Problem Solving Power

We thus have an unlimited expanse of new problems to solve but there is a huge increase in power coming in the next decade or two from improvements in the hardware and algorithm environments (see the sidebar – Appendix A – for forecasts). The limiting factor for future scientific software systems seems to be writing the software itself. There are few hard, reliable data points about trends in software productivity but everyone seems to believe there has been a very slow increase in the productivity of writing programs in conventional languages (Fortran, C, Ada, Java, ...). I

forecast that it is very likely that the cost of such raw software will remain high. Further, the reliability of such software is not improving (see the article in this issue by Les Hatton). If new applications require new raw software and if this forecast is correct, then the future for scientific software systems is bleak. The entire problem solving process will be dominated by (and bogged down in) software development.

I argue by considering four examples from my own experience that this bleak view of software development is misleading. It is only correct if one insists on writing new, raw code instead of using application systems.

- **Text Processing Software.** I wrote a book using *TEXT90* in 1968, I have used *troff*, *LaTeX*, *vi*, etc. after that and I now use *Word* whenever possible. The work to use *TEXT90* was about 10 times that of using *Word* and the quality was at least 100 times (perhaps 1000 times) worse. A sample of the best possible *TEXT90* output is given photographically in Figure 1 as modern text processing software cannot reproduce the low quality of *TEXT90*.

- **Elliptic Partial Differential Equations.** The decade old, high level systems *ELLPACK* [4] and *DEQSOL* [6] reduced programming effort by a factor of about 100 [3]. Current versions of these systems have much more improved programming efficiency.

- **Graphics.** My first experience with plotting software was in 1961 and I have used many systems since. I am still very impressed when I see *Excell* or *Matlab* graphics and think back to spending days to get much worse plots than I can now get in minutes.

- **Symbolic Computing.** I struggled using *FORMAC* in 1965 until I realized I did not understand the problem I was trying to solve. Using *Maple* or *Mathematica* now is easily an order of magnitude faster and these systems are two orders of magnitude more powerful.

One can describe these successes as having two parts: (1) a collection of problem solving and information processing components, (2) a natural system to use the components. This sounds like a software library plus a library routine delivery system, the original (1951) software reuse idea for a software parts technology. This obviously good idea has had mixed success. There have been numerous large scale library efforts, more have failed than succeeded. The idea has succeeded in the sense that there are now many thousands of well tested, high quality scientific software routines. The idea has failed in the sense that almost everyone starts from scratch instead of searching libraries, many newer systems and libraries contain unreliable and/or inefficient routines that repeat mistakes discovered 20 or 30 years ago. Thus I conclude that libraries alone have not been enough to improve software productivity significantly.

I forecast that programming productivity for conventional languages will continue to be static, that software reuse via libraries will continue to grow slowly, and that scientific applications software will experience strong growth. The paradigm of problem solving environments [1, 2] will provide a very significant increase in programming productivity in science and engineering, an increase of two or three orders of magnitude.

The commulative effect of these forecasts for hardware, algorithms, and programming is staggering; in 20 years scientists working on large, difficult problems could have 10 million times the

Figure 1: An example of the best output possible in 1968 from *TEXT90*.

current solving power and it could be a hundred to a thousand times easier to do the programming. I see the most likely causes for shortfalls in the forecasts to be (1) *Practitioners are slow to adopt better algorithms.* Good reasons for this slowness include: the loss of huge investments in existing programs, the uncertainty of the performance of new algorithms (and initial implementations will be much less than optimal), the re-education of users, the fragility of large software systems. (2) *The systematic and scientific testing of scientific software is immature.* It is hard to measure the quality (accuracy, reliability, efficiency, programming cost) of software and without such measures the case for changing software is unclear.

Application Trends

There are some existing directions in scientific computing that will clearly continue:

- *More accuracy.* Better approximations to the continuum will be made (finer grids, more terms in expansions). Better approximations to the physics models will be made (terms added, linear models replaced by nonlinear ones).
- *Optimization and design.* The models will be used to design (and test) new things, more parameters will be used, more complex parameters (shapes, response functions) will be optimized.

Then there are three new application domains that will move into the feasible range:

- **Multi-physics.** Applications will be attempted with several different important physical phenomena (fluid flows, material strength, heat flows, phase changes) and realistically complex geometry.
- **Multi-scale.** Applications will be attempted with two, perhaps three, scales in space and/or time.

Both of these domains require more than just increases in computing power; new numerical, analytical, and software methods are required.

- **Model discovery.** Science has a long tradition of fitting mathematical models to experimental data, e.g., the Law of Gravity, lift curves for airfoils. Much more complex phenomena can be modeled systematically by more than mathematical formulas, e.g., chemical reactions with multiple species, physical properties of composite materials.

Software Trends

The software trend toward integrated problem solving environments will continue to gain strength. I foresee a related trend toward application area specific software methodologies. Just as mechanical, electrical, and chemical engineering have different mathematical methods, models, and techniques (starting at about the sophomore year in college), such disciplines will develop different software methods, data structures, and techniques. The "general science" problem solving environment of high school will fragment into many as one moves to more and more advanced applications.

Some new trends in scientific software that I foresee developing are:

- **Personalization.** Scientific software systems will become much more sensitive to the context of the user, the computing environment, and the computational task. Software libraries, language dictionaries, telephone directories, file systems will be tailored to the context. The all-inclusive massive information resources will be available through the net but not resident nor represented in the user's own environment. Memory systems will have the capacity to store every word one reads or hears and I/O systems will be able to capture this information. Aliasing will become pervasive; "The example I had in the 1999 IEEE Transactions article", "The guy who called from Intel last month", and "The machine Thomas used at Cal Tech last year" will be recognized as precise path names.
- **Dynamic problem solving.** The traditional paradigm of *program*, *compile*, *load*, *run*, *view results* will fade away. Applications can, and will, reconfigure everything as new data arise, new difficulties are met, new resources emerge, and new priorities are imposed. A good deal of the new computing power coming on line will be consumed in this task. The comfortable practice of a system managing many independent processes will fade away. Hard and soft deadlines ("real time constraints") will be generated and met by applications of all kinds.

- **Automatic software optimization.** Compiler-like analyses will spread to much higher levels of programming than just the language module. Heroic efforts will be made in reducing indirection in code, in simplifying software interfaces, in testing alternate paths, or reorganizing data and programs, in replacing interpretation by compilation, etc. The high cost of software development is an irresistible force in a software parts technology, but software parts and portability can generate large overheads. Once software performance is identified as a problem, then such optimizations will be initiated automatically and dynamically.

- **Hybrid real and simulated systems.** Simulation is practical when (a) the underlying physics is well understood and (b) the physical phenomena are expensive, inaccessible, hard to visualize, immeasurable, etc. Otherwise real systems are used. Combining virtual reality (simulation) and physical reality as in a flight simulator will become common. Artificial models will be used for things that have no physical existence, e.g., flow of information, evolution of language, or weight of debt.

References

[1] R. Boisvert and J.R. Rice, From scientific software libraries to problem solving environments, *IEEE Comp. Sci. Engr.*, **3**, Fall (1996), 44–53.

[2] E. Gallopoulos, E.N. Houstis, and J.R. Rice, Computer as thinker/doer: Problem solving environments for computational science, *IEEE Comp. Sci. Engr.*, **1**, Summer (1994), 11–23.

[3] John R. Rice, Programming effort analysis of the ELLPACK language, in *Advances in Computer Methods for Partial Differential Equations*, (Vichnevetsky and Stepleman, eds.), IMACS, Rutgers University, (1979), 28.

[4] John R. Rice and Ronald F. Boisvert, *Solving Elliptic Problems Using ELLPACK*, Springer-Verlag (1985), 497 pages.

[5] John R. Rice, *Numerical Methods, Software and Analysis*, Second edition, Academic Press (1992), 720 pages.

[6] Y. Umetani, et al., Numerical simulation language DEQSOL (Japanese), *Trans. Info. Proc. Soc.Japan*, **26**, (1985), 168–180.

APPENDIX A: **SIDEBAR**

PAST AND FUTURE HARDWARE AND ALGORITHM ENVIRONMENTS

Hardware performance trends have been remarkably regular and persistent over a long period. In 1972 I collected hardware performance projections and consolidated them into a 25 year forecast for top-of-the-line computers. The forecast and actual value are remarkably close:

	1970 Actual	*1970 Forecast for 1995*	*1995 Actual*
Speed	100 K Flops	6 G Flops	6 G Flops
Main Memory	200 Kwords	20 Mwords	50 Mwords
Secondary Memory	200 Mwords	750 Gwords	500 Gwords
Adds per penny	5 Million	2 Billion	1.5 Billion

I now forecast that processor cycles will become a little faster, processors will become much more complex, and processors will become much more numerous; the result will be that desktop power will increase by a factor of 1000+ in the next 20 years. Further, I forecast that the ratio of memory size to processor power will increase some, the ratio of network performance to processor power will increase a lot, and sensory I/O devices (visual, audio, tactile) will be much better. I conclude the 2015 hardware environment will increase in raw computing resources by a factor of 1000–5000.

As applications become more difficult and complex, algorithm performance becomes more critical. As a benchmark consider computing the temperature distribution inside a common engine block. This is a well behaved, general, well understood elliptic problem for which many methods have been proposed over the years. I estimate the time and memory to solve this problem to with 0.1% accuracy using a 100 MFlop machine to be:

*Standard Practice**	*Time*	*Memory*
1945–55	3 Myears	800 Mwords
1955–65	200 years	5 Mwords
1965–1995 Direct Solve	5 hours	50 Mwords
1965–1995 Iteration	1 hour	300 Kwords

*See [5], Section 10.3.C for a description of the algorithms used and other assumptions made for this forecast (this table corresponds to the first four rows of Table 10.6).

Much more efficient algorithms are already known for this problem but these are not yet standard practice (there are considerably more complicated to program); they can reduce the resources needed to solve this benchmark problem to about two seconds and 50 Kwords. I forecast that these algorithms will be adopted, improved, and, most importantly, extended to applications with much more irregular behavior. I conclude that the 2015 algorithm environment will increase in raw solution efficiency by a factor of 1,000–10,000.