JOURNAL OF COMPUTATIONAL PHYSICS 73, 41–58 (1987)

Supercomputer Environments for Science Applications

BRENDAN MCNAMARA

John von Neumann Center, P. O. Box 3717, Princeton, New Jersey 08543

Received July 15, 1986; revised March 4, 1987

Current implementations of Roberts' rules for programming for supercomputers are reviewed. The coming Class VII computers will require more powerful software tools to realize their full potential, with more use of knowledge systems, symbolic manipulation, automated programming, parallel processing, parallelized graphics, and an integration of high-performance work stations into the supercomputing environment. The need for these approaches is illustrated with simple examples.

1. INTRODUCTION

Among Keith Roberts' many interests were standards for writing software. Roberts' rules [1] for software style are still very useful and bring clarity and ease to Fortran programming. They were motivated by the need to communicate clearly with other scientists and later workers on a project, and by the continuing need for portability of programs. The environment for large-scale computing has changed dramatically with the advent of supercomputers 1000 times faster than the Class II–III computers available then and with workstations costing \$10–30 K for personal computers of the same power and with high-resolution color graphics. Keith Roberts' standards had the effect of clarifying the steps needed to do successful computing and the goal of this article is to identify some of the new tools and methods which will make the full power of the modern environment accessible to the ordinary user.

The emerging Class VII supercomputer with 300 million words of memory and peak computation rates of 10,000 megaflops is capable of doing problems which scientists can describe but for which the algebraic effort in developing the equations is enormous and for which the coding will take many man-years. Though many codes can be expanded to take advantage of the Class VII capability, the machines have outstripped our ability to program them for the most advanced problems. There is a clear need for a new, Fifth, generation of software tools which are simpler to use even than Fortran in describing scientific problems.

Roberts' rules have been well received by a small number of people in the Magnetic Fusion community but many people still program with almost no com-

0021-9991/87 \$3.00

ments, inscrutably formatted input, and very cryptic output. Clearly this works if only the final conclusions are to be communicated to one's colleagues, but it will not work for large-scale code development efforts. It is common in commercial, systems, and engineering control applications to program to rigid rules described as "software engineering" but these approaches are rarely used in scientific applications. There is a median in program documentation and some of the most valuable elements of Roberts' rules will be given later. A basic program shell written in this style will be discussed and should be a useful teaching aid as well as a good starting point for many codes.

The environment we have to work in is partly due to what we demand and partly due to advances in computer science, with the fascinating array of alternatives now offered. The real progress in creating user friendly software has come from the personal computer manufacturers like Apple, Commodore, and IBM. Efforts are currently in progress at many computer centers to bring this level of commonsense capability to the supercomputer environment. The Class VI supercomputers, led by the Cray 1 series, have executed a tremendous number of scientific problems with the barest minimum of software. With some persuasion, a whole new generation of increasingly intelligent tools can be produced for the Class VII computers and the environment around them.

The architectures of scalar, vector, and graphics computers with the additional complication of parallel processing offer different methods for running problems and interacting with the results. It is a daunting prospect to try to become expert in all the possibilities before solving a scientific problem. However, it is usually found that, providing a problem has been coded with some reasonable understanding of the objectives of vector and parallel processing, the amount of code which finally has to be tuned to the architecture is about 1 % of the total! This is not surprising since the architectures and compilers were already designed with scientific calculations in mind. The obvious implication is that only a few users in each community need be highly skilled at tuning codes and the rest of us should concentrate on clarity and communication in writing our programs. I will therefore not attempt to discuss code optimization further.

Having argued the needs, we begin with a review of Roberts' rules and some current implementations of a basic program shell. This shell is useful for both a user and as a target environment for automatically generated code. The starting point for computer-generated code for many scientific applications could usefully be an expert system. We make the case for this with some obvious examples. The examples are then used as a basis to develop the expectations of a symbolic manipulation system and of packages to do code generation.

Essential to almost every scientific calculation is the ability to display the results graphically and, from a supercomputer calculation, this really means hundreds of plots per run. This implies that a plot should be as easy to code as a WRITE statement and we describe implementations of high-level packages which do this. An advantage of such high-level packages is that they lead to natural extensions to parallel processing where a calculation may proceed on several processors while

graphical output is being generated in real time in parallel on another. The final discussion is of the workstations, which complete the supercomputing environment. These are indeed self-contained computer centers and it is essential for different groups to develop tool kits which make their problem sets easy to handle without learning everything about the device. Although these workstations are often used as a stand alone resource, they are increasingly a vital part of the supercomputing scene.

2. Current Implementations of Roberts' Rules

The original Olympus system [1] is available from Computer Physics Communications. Many of the features were appropriate to Fortran IV and have in effect been included in Fortran 77 and modern Fortran systems. The system is used by a number of people at Livermore and at the Princeton Plasma Physics Lab as a way of structuring a code. I have used a simpler version of the system, modified to utilize the interactive features of the Livermore systems [2], and have transported the ideas to the Cyber 205 and Sun workstations at JVNC. The listing reproduced in Listing 1 is the key part of the Fortran Applications Driver [3] or FAD, which embodies many of Roberts' rules in a highly practical form.

This simple code shell provides the following features and others which are described fully elsewhere. [3].

(i) The MASTER routine runs many steps of many cases of a problem. The user describes the specifics of the problem through (a) the COMMON and DATA statements which layout the variables and a test case, (b) the user routine VALUES to complete the problem initialization, and (c) the routines to be written by the user to run a step, STEPON, (d) control principal printed and plotted results, OUTPUT, and (e) check for end or convergence conditions, TESEND.

(ii) Data input is by NAMELIST, which, although not a "standard" in Fortran, still comes with every major Fortran system. The input file or data stream is instantly intelligible and, with the character variable, "COMMENT" can be fully described by the user.

(iii) The first part of the input should be a description of the purpose of the run—often hard to discern weeks later from the output! This is handled by the FAD library routine, LABELR, which read up to 10 lines of text and prints it on the output and graphics channels.

(iv) Initialization of input and output files, graphics, multiprocessing, timing, and terminal interaction is all automatic through the library routines PICKIO and PSETUP. It is up to the local implementers of the system to write these routines—certainly it is not the responsibility of the user!

(v) The library routine TTYMES links the code to the terminal to look for interrupts and to make appropriate responses. Any key will stop the code at the

```
C               THE JOHN VON NEUMANN CENTER
C               FORTRAN APPLICATION DRIVER (FAD)
C
*COMDECK,FADCOM
C******         DO NOT CHANGE THESE COMMON BLOCKS!!!!
C******         THEY ARE NEEDED BY ALL FAD'S AT JVNC
      COMMON /FADMSTR/ ICASE, IEND,IHALT, IQUIT,
     .  ISTEP, ISW(50), ITIME,ITTY,
     . LABEL(100),LABLINS,LGRAPH,LIN,LOUT,LTTY
      COMMON /FADPAR/ PI
      CHARACTER*8 DATE,DROPFIL,INFAD,OUTFAD,TYME
     . ,NUSER
      CHARACTER*80 COMMENT
      COMMON /FADFILES/ COMMENT,DATE,DROPFIL,INFAD,OUTFAD,TYME
     . ,NUSER
      COMMON WORK(10000)

*COMDECK,FADYOU
C               PUT YOUR OWN COMMON BLOCKS HERE
C

*COMDECK,FADATA
      NAMELIST /MAIN/ COMMENT,IEND,IHALT, IQUIT,ISW,ITTY,LIN,LOUT
      NAMELIST /TEST/ COMMENT,ICASE, ISTEP
C               INSERT YOUR DATA STATEMENTS HERE
*DECK,FADUPD

C******         DUMMY MAIN PROGRAM NEEDED FOR USE WITH PRIVATE LIBRARIES.
      CALL MASTER(0)
C               STOP THE BEAST!
      STOP 666
      END
C
      SUBROUTINE MASTER(IPICK)
C-----------------------------------------------------------------
*CALL,FADCOM
C
C               OPEN THE BASIC INPUT/OUPUT FILES
      CALL PICKIO
C               INITIALIZE THE SCIENCE GRAPHICS PACKAGE
      IF(LGRAPH.NE.0) CALL PSETUP
C
C               2. LABEL OUTPUT AND INITIALIZE EACH CASE
C-----------------------------------------------------------------
      CALL LABELR
201   ICASE= ICASE+ 1
      ISTEP= 0
      IF(LIN.NE.0) CALL LISTIO(1)
      IF(ITIME.NE.0) CALL ALLTIM(LOUT,'2. I/O OPENED. DATA READ')
      IF(IQUIT.NE.0) GO TO 701

C               3. SET PROBLEM ARRAYS
C-----------------------------------------------------------------
C
      CALL VALUES
      IF(ITIME.NE.0) CALL ALLTIM(LOUT,'3. PROBLEM ARRAYS SET')
C
C               4. PRINT DESCRIPTION OF THIS PROBLEM
C-----------------------------------------------------------------
C
400   IF(LGRAPH.NE.0) CALL LISTIO(LGRAPH)
      I= 1
      CALL OUTPUT(I)
      IF(ITIME.NE.0)
     . CALL ALLTIM(LOUT,'4. PROBLEM DESCRIPTION COMPLETED')
```

LISTING 1. The Fortran application driver.

```
C
C              5. MAIN LOOP OF THE COMPUTATION
C_____________________________________________________________
C
500     ISTEP= ISTEP+ 1
        CALL STEPON
        IF(ITIME.NE.0) CALL ALLTIM(LOUT,'5.1 STEPON JUST COMPLETED')
C
C              5.2 OUTPUT FROM EACH STEP
C_____________________________________________________________
C
        IF(ISTEP.LE.1) CALL LISTIO(-2)
        I= 2
        CALL OUTPUT(I)
        IF(ITIME.NE.0)
         CALL ALLTIM(LOUT,'5.2 STEPON DIAGNOSTICS COMPLETED')
C
C              5.3 CHECK IF LOOP HAS ENDED
C_____________________________________________________________
C
        CALL TESEND
        IF(ITIME.NE.0) CALL ALLTIM(LOUT,'5.3 END CONDITIONS TESTED')
        IF(ITTY.GE.1) CALL TTYMES
        IF(IEND.EQ.0) GO TO 500
C
C              6. FINAL OUTPUT FROM THE PROBLEM
C_____________________________________________________________
        CALL LISTIO(-1)
        CALL LISTIO(-2)
        I= 3
        CALL OUTPUT(I)
        IF(ITIME.NE.0) CALL ALLTIM(LOUT,'6. CASE OUTPUT FINISHED')
        IF(ITTY.GE.2) CALL TTYMES
        IF(IPICK.EQ.2) RETURN
C
C              7. CHECK IF MORE PROBLEMS TO BE RUN OR IS IT TIME TO QUIT
C_____________________________________________________________
C
        IF(IQUIT.EQ.0) GO TO 201
701     I= 4
        CALL OUTPUT(I)
        CALL ALLTIM(LOUT,'7. END OF THIS RUN')
        IF(LGRAPH.NE.0) CALL PCLOSE
C
C              8. DO CHECKPOINT/RESTART TO RECOVER ERRORS EASILY.
C_____________________________________________________________
C
        IF(IHALT.EQ.0) THEN
        CLOSE(UNIT= LIN)
        CLOSE(UNIT= LOUT)
        RETURN

        ELSE
        CALL HALTGO
        IEND= 0
        IQUIT= 0
C               REOPEN THE BASIC INPUT/OUPUT FILES
        CALL PICKIO
C               REINITIALIZE THE SCIENCE GRAPHICS PACKAGE
        IF(LGRAPH.NE.0) CALL PSETUP
        CALL LABELR
        CALL LISTIO(1)
        IF(IPICK.EQ.3) RETURN
        GO TO 400
        ENDIF

        END
```

LISTING 1—*Continued.*

next time step and it will display the TEST namelist. The MAIN namelist can then be reset or the code can be allowed to continue or forced to quit.

(vi) The whole run is carefully labelled with time, date, user number, and any other information which can be usefully gleaned from the system by routine ALLTIM.

(vii) The routine LISTIO is the essential link between the data input and the code variables and so is always needed in the user's source. It is not listed here, but also contains a list, in alphabetic order, of the meaning of each of the FAD variables. Users are encouraged to do likewise as they develop a new code.

(viii) The code can be made to stop at HALTGO. This library routine closes all the output files and saves the current copy of the dropfile. The code can then be restarted exactly at this point with new input and output files to continue an interesting run or to switch on detailed diagnostics for one final timestep.

(ix) The trivial-looking main program is there as a starting point for the loaders, which usually will not pick a main routine from a binary library. The idea is that when you have 100 routines in a code and you only need to work on a couple of them, it is easy to turn the whole code, except for this main program and the routines still being developed, into a binary library. Your text editing is then simpler and listings are much smaller.

The coding presented here is deceptively simple since many difficult pieces of system-related coding have been buried in the library routines. However, I have found it easy to duplicate the effects on a number of systems and, having done it once, these very useful features are forever available. This code shell is a useful teaching tool for new programmers or even for new users on a system. It also allows colleagues to help each other more readily with coding problems as the top-most structure is common. Portability is made easier since the system is easily reproduced by local experts and is already available for Cray and Cyber computers [2, 3].

3. Job Control

Even the simplest job requires many steps, such as block substitutions, vectorization, compilation, load, run, print, and plot, and it is the task of the "Job Control Language" to access and run all the pieces. Most such languages have accreted rather than been designed and the most creative is surely UNIX. Even so, the commonest sequence of activities is to change a few lines of Fortran or Data and rerun the whole sequence and look at the results. All the intermediate activity should be buried into a single command and most users find ways to do so. This has been done in the "Friendly Fortran" package [2] which has a controller to do all of the obvious sequences with a one-line request.

At the John von Neumann Center things were a little harder because one may sit

at a high-end work station, running UNIX, to talk to the VAX-8600 front-end system, running VMS, to submit jobs to the Control Data Cyber-205, running the batch system, VSOS. This is to be rationalized by replacing the Cyber-205 by and ETA-10, running UNIX, and it will then not always be necessary to work on the VMS system.

In the meantime, the idea of having commands to do whole sequences of common activities was extended to allow VMS users to interact with procedures, with the family name of PEP, which write appropriate VSOS job controls for submission to the Cyber [4]. The session (slightly shortened for this paper) to update, compile, load, run, print, and plot from a version called TAG of the Fortran Application Driver shell is shown in Table I. The PEPCLGO command is interactive, uses much of the availabe software options on the Cyber, and finishes with instructions on what to do next. The cycle of "edit–submit–inspect results" thereafter requires only one statement, @ TAG, to drive jobs forth and back to the Cyber. This is about as friendly as one can make a batch supercomputer.

This is but one example of the range of PEP commands which in fact drive all the available Cyber utilities in their commonest mode. Thus, the extension .UPD for the Fortran source told PEP that it is in Update form and has to be preprocessed before being given to the compiler. The compiler is run with the set of options recommended for most users. The Loader is instructed to invoke the graphics library, when channel 63 is used, and the mathematical and symbolic dump libraries are always invoked as standard resources. The few answers to the PEPCLGO questions generated a 50-line, customized procedure to run the job.

Even on the UNIX system one needs similar omnibus commands and a way to generate them easily. The family name, PEP, will be changed to represent the

TABLE I

Running a VAX Procedure to Write Cyber-205 JCL

```
JVNCC> pepclgo tag /lib= fadlib
Your "tag" for your files on the 205 will be TAG
Found default Fortran Source TAG.UPD;1
Is this the one you want?: yes
Enter time limit in STU's (< CR> = 60): 1000
Enter FORTRAN unit # of text input file - < CR> when done: 5
Found default Input file TAG.DAT5;1
Is this the one you want?: yes
Enter FORTRAN unit # of text input file - < CR> when done:
Unit # of (next) output text (print/graphics) file - < CR> when done: 6
Unit # of (next) output text (print/graphics) file - < CR> when done: 63
Unit # of (next) output text (print/graphics) file - < CR> when done:

PEPCLGO made a command file for submitting your job to the Cyber 205.
1. Submit your job by entering "@TAG"
2. Wait for the TAG.LOG file to show up.
3. When it does, the following files should be in your directory:
  TAG.LIS     Fortran compiler listing.
  TAG.MAP     Load map
  TAG.OUT(n)  Text outputs, n= Unit #
```

machine on which the job is to run and whose compiler and libraries and so on may be different. To run on the ETA one builds the appropriate command set with "etaclgo tag" a Unix command available on a work station or on the ETA-10. The jobs are then run by typing "tag" to the ETA-10 or "submit tag" to send the job up from a workstation. If the job is small enough the identical Fortran may well run on a SUN workstation but the commands would then be "sunclgo tag" to create the local command shell.

As with the FAD, the shells produced by these commands provide an excellent starting point for more exotic sequences and serve the user as an instant manual on how to write UNIX shellscripts or "make" files.

This may be an oversimplification of the tasks in writing a new code and running it on (several) machines. However, this is very often quite sufficient and almost always adequate as a starting point. Before developing ideas of the further potential, let us complete the simplified tool kit with a discussion of graphics, another area which Keith Roberts emphasized as essential to progress in computational research.

4. Simplified Science Graphics

It is obvious that most of the output from large scientific codes should be graphical. At the same time, pictures without numbers convey little scientific knowledge and unlabelled pictures remain unintelligible to any but the author. A good science graphics package should provide publication-quality graphics, without requiring the user to learn hundreds of routines, and yet should have easy access to color, Greek and mathematical fonts, multiple displays, and other features of modern computer graphics. Unfortunately, different disciplines have very different needs and while many physicists are content with curves and contour maps to diagnose their mathematical models, chemists and engineers demand three-dimensional views of molecules and structures as a key part of their understanding. Here we describe simple packages which are suitable for physicists and trust that others may adapt the philosophy.

The first system, GRAFL-II [5], is widely used in the Magnetic Fusion community and drives the Livermore libraries TV80LIB and GRAFLIB and also CERNLIB. The system is easy to use and leads naturally to codes with exclusively graphical output where enough scales, labels, contour heights, and text provide all the needed numbers. A plot command is as simple as a WRITE statement and more useful in many cases. The second package, which is briefly described here, is very similar with some enhancements.

The first difficulty with most graphics libraries is to find out how to put several plots on a page—or more usefully, on a screen—without all the titles overlapping each other. This is obviated in our package at JVNC by allowing the user to pick a

layout of one to three pictures across a page and up to six rows of pictures down. A typical setting may be

```
CALL PLAY(121)
```

which would give one picture on the first row, two on the second, and one on the third. The default is 22. Each plot call then needs an argument, LAY, to select the picture being used. An important feature of the selection process is that any picture may be overplotted with lots of lines, contours, markers, and text, but when a new picture is chosen any attempt to return to a finished picture causes a new page to be started for that plot. This prevents accidental overplotting and is one of many safeguards built into the package.

The rest of the plotting package is illustrated by the call to plot a curve:

```
CALL PCURVE ("LINLOG", XAXIS, GAMMA, NX, LAY
."Functions of X$", "X$", "; g < (X) $")
```

The first argument chooses the type of scaling in each direction. The use of a character string here and elsewhere makes the coding easy to follow. The scales are selected from the NX values of XAXIS and GAMMA which need not be monotonic. The routine checks these arguments and, if they are unplottable, gives error messages on the picture. The title, x axis label, and y axis label MUST be given in the call—though "$" is acceptable. This (almost) forces users to produce labelled plots. The odd construct ; $g <$ is to change fonts to Greek for a gamma and back to Roman for the rest of the caption. Finally, a second call to PCURVE with the same LAY will overplot this next curve in a different line style, a different color, and with a different marker. All these features cycle automatically through a menu which can of course be controlled with calls to PSTYLE, PCOLOR, or PMARK. The key point is that the default is to provide multicolor output which is also easily read in black and white.

The package provides curves, contours, three-dimensional surfaces, markers, text, and vector fields and other types of plots are easily added. The example in Fig. 1 shows a fairly elaborate plot and the one statement which generated it. The package is itself only a driver for the underlying library which, in this case, is ISSCO's DISSPLA library. The package definition is easy to implement to drive any other such library and makes codes more portable as well as easier to write.

It is important to realise that the first objective of these packages is to make it easy to make hundreds or thousands of plots from Fortran codes. This is quite different from the need to take a small amount of data and make one beautiful picture, a need which is met ideally with P.C. and workstation packages. Three-dimensional plots also require a different treatment because the viewing angle and distance are so important and often cannot be known beforehand. Again, modern workstations provide most of the needed capability to examine a single picture at leisure from any angle.

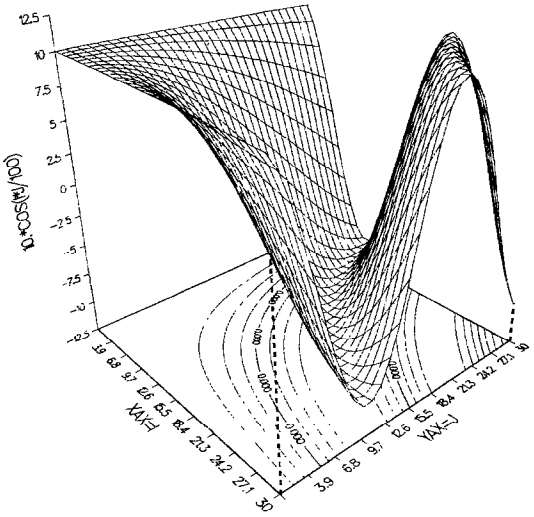

FIG. 1. A 3D surface suspended over its own contour map. This is generated by one statement: CALL

We have described three sets of tools to simplify the starting point for doing supercomputing—the FAD, the PEP procedures, and some simplified science graphics. For many people this may already be sufficient as they may need to run large, commercial, or public domain packages which do all of their modelling and then need quick supplementary calculations and graphics to complete the work. For others, these tools are the first step towards some huge enterprise and the rest of this paper will develop ideas to simplify that development.

5. KNOWLEDGE SYSTEMS

So-called expert systems [6] have so far been employed in limited applications and directed at people with minimal computer experience. Their development and application in supercomputing is essentially nil, yet this approach is immensely important for supercomputing. The field is new and the constructs and languages required have not yet matured. Natural language and even the mathematics we use in science is highly context dependent and so the present approaches to building expert systems require that a finite body of knowledge, with its own syntax and terminology, be coded. The systems are more like an interactive text book in a topic or an automated decision analyser than a human expert. They sometimes perform better than the expert because they do not tire and always give the same weight to a decision branch.

An obvious candidate for a very useful expert system for supercomputing applications is in solving ordinary differential equations—ODEs. The system should

know the characteristics of all the common solution methods and be able to interact with an algebra system which could estimate eigenvalues for the set of equations and detect stiff systems. The user's requirements on speed and accuracy of the solutions strongly affect the choice of method and should be part of the expert system. The final output of a session with the ODE expert system would be either advice on the best library routines or programs to use or a suggested method to be programmed. In some cases a more important output would be experiential advice on the nonlinear properties or long-term accuracy of the methods. Much of this would be obtained from experts in ODEs who would contribute their knowledge and experience to the scientists who would program the system.

Such an expert system would have to perform on a scale comparable to a good text on ODEs [7] and the contents of the major subroutine libraries [8, 9]. Clearly the task is considerable, but since ODEs represent the commonest form of modelling for scientists and engineers, it should be worthwhile. Similarly, an expert system for statistical analysis of experimental data would be of great value. These two systems could set the standards, the formats, and the expectations for future systems for supercomputing.

6. Symbolic Manipulation

This is by now a mature field with, for example, several versions of MACSYMA [10] and a wide distribution of REDUCE [11]. What has not yet emerged is a coherent library of applications packages to use these facilities easily in many areas. The notation and algebraic devices used in fluid or solid mechanics, magnetohydrodynamics, kinetic theory of gases and plasmas, or quantum mechanics are sets of techniques which can be easily codified in these algebra systems. It is now essential that this be carried through for the Class VII supercomputing applications for multidimesional problems.

A good demonstration that all this is feasible is given in the Ph.D thesis by M. C. Wirth [12]. The thesis described a vector and dyadic analysis package which could then generate component equations expressed in any of 14 orthogonal coordinate systems. Sets of scalar partial differential equations could then be reformatted in terms of variables on a discrete multidimensional mesh and the differential operators turned into difference operators to generate sets of finite difference equations. These equations could then be expanded in Taylor series to produce the error terms in the difference representation and show whether the equations were diffusive or dispersive in their error propagation. Alternatively, the equations could be Fourier analysed in local fashion to evaluate the stability properties of the schemes. Higher-level programs could generate complete alternating-direction-implicit difference schemes in two or three dimensions. The thesis stopped at the point of turning all the component expressions into Fortran coding to solve the complete problem. Similar work by G. Cook [13] went on to generate well-optimized code for Cray computers. These seminal theses serve as an existence

proof that such packages can and have been written but they do not yet represent a finished system for general use.

The design of a symbolic system for ODEs is worth outlining, as the need for it is so universal. Ordinary differential equations include Hamiltonian mechanics, non-linear mechanics, eigenvalue problems, rate equations for many kinds of zero-dimensional modelling, and so on. Each application requires a different treatment of the equations and so the symbolic tools should be designed in different, context-dependent sets. Consider a generic set of ODEs

$$dX/dt = F(X, C, t)$$

where X, F, C are vectors of some lengths and where some constants are defined by additional relations, $C = C(P)$, involving parameters, P. The functions, F, may also be defined in subsidiary equations and the boundary conditions are specified by $X = X(C, P, t = 0)$. Other relations may also be specified to define intermediate results or functionals of the answers which are needed for the studies. The point is that the total problem may require a large number of relations which first have to be checked for internal self-consistency.

The entry point to the symbolic system is first through the mechanism to enter all the relations. This system will also demand names and plain text descriptions of the quantities which can then be used to make the final coding intelligible and the graphical output readable.

Another tool could do dimensional analysis and simplify the equations to find a minimal parameter set which determines the system. This tool could also change units or rescale results as required.

The next important tool is a cracker which will identify dependent and independent variables and do the consistency checks. This tool will also assign Fortran names to the variables and to the arrays for storing all requested intermediate and final answers.

At this point, the system has a complete, symbolic description of the equations and algorithms to be used. It remains to write a functioning code.

7. Automated Computer Programming

An automated system is only useful if it generates many more lines of code than are input. Utilities for translating the output of MACSYMA or REDUCE programs into FORTRAN have been written many times, though production of vectorizable code is still a topic for development. The more difficult part of the task is to handle the—often convoluted—logic of a set of algorithms. In the case of the sets of ODEs discussed here, the approach would be to write complete program shells within the Fortran Application Driver for each problem type. These would be a set of "context-dependent" shells with the main loops and branches in place and interactive points for the user to make appropriate decisions on what functions to

store, or pictures to plot. Some of this was designed and implemented in a pilot project [14].

The tools to be offered include commands to be used anywhere in the program, for logic controls like "while," "until," and so on. There are many advantages in the symbolic programming approach over a compiler: The type of a variable can be allowed much greater scope, including operator functions. All the properties of a variable need not be displayed explicitly whenever it is used. A simple example would be a plotting command like

plot B V P against T

The plot command would find the physical units, the DIMENSION of the variables, the descriptions of the variables, and use the simplified science graphics to produce the needed plots, fully labelled and annotated.

Programming constructs can be developed to reflect the need of particular disciplines, producing a language which knows the context of crystallography or biomedicine or plasma physics. The final output language of the automated programming may be Fortran-8X, or C, or Pascal, but many scientist would never see this and the main programming language of the future could become these families of science-based symbolic operators.

The substantial reduction in the number of lines of programming needed for each problem has other consequences. The programs are much easier to modify and colleagues can use them and add to them much more easily. The coding effort needed to restructure a code in Fortran in order to try out a new algorithm is often a deterrent to doing so, and one ends up patching unsatisfactory algorithms. Symbolic programs would be much easier to change and libraries of techniques and operators much easier to build, making the choice of algorithms for nonlinear problems more experimental. An ultimate step in the chain is to allow the computer to make changes to symbolic programs and run them in response to very abstract questions.

A futuristic programming environment is outlined in Fig. 2: The user enters at the Expert Systems level and uses the A.I. codewriter to write the code or to modify an existing symbolic code from a library. When the results are sufficiently interesting to be published, a paper is written—again on the computer system—and

Scientific Information Base. It seems only remotely possible, at present, to supply a "Knowledge Extractor" which will allow the successful scientist to encode his knowledge also in the same fashion as the Expert System from which he started. It is certainly possible to insert good references to the latest products into the Expert Systems. The (Sixth Generation) capability to have the computer do significant self-programming is a source for further speculation. Let us conclude with a short list of more accessible possibilities which are now available or readily produced.

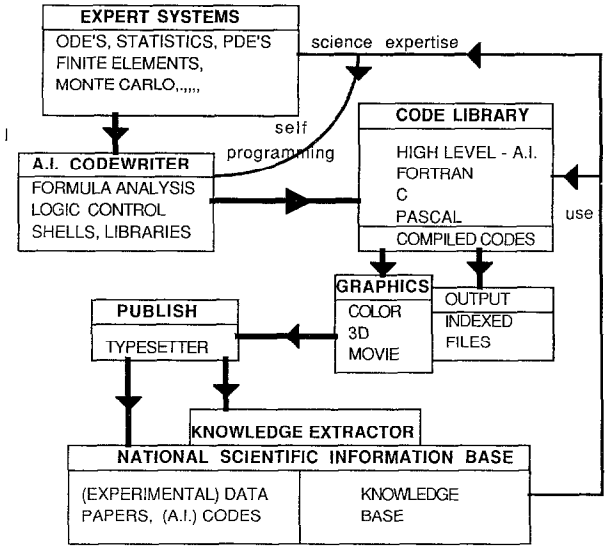

FIG. 2. Use and capture of scientific knowledge. This shows the flow of a problem from the expert systems advisor to the artificial intelligence codewriter to a compiled Fortran code, results, a publication, and storage in a National Scientific Information Base.

8. THE CLASS VII COMPUTING ENVIRONMENT

The price/performance ratio for both supercomputers and workstations is dropping rapidly, so rapidly that a qualitative change is under way in how we do computing and a quantitative change in the number of people who have access to such facilities. At one end, millions of personal computers have been sold and have served to train a large fraction of society in doing computing. At the other end, the technology of large-scale integrated circuit production is soon to allow hundreds or thousands of today's supercomputers on the market at a readily affordable level. The changes are so great that it is hard to predict what it will mean for scientific computing and impossible to know how it will affect society. However, we can define the "Class VII supercomputing environment," the elements of which are illustrated in Fig. 3 and described further below.

It includes a Class VII supercomputer, a high-speed (1–500 megabits/sec) network, and a workstation with 3D color graphics and a fully interactive operating system. The purpose of the workstation is to supply all the input, output, graphics, text editing, and typesetting for the user, and management of codes and results on the supercomputer. These workstations allow the user to run several processes at once, and also to view them in separate windows on the screen! The windows may be separately attached to different computers or merely to the

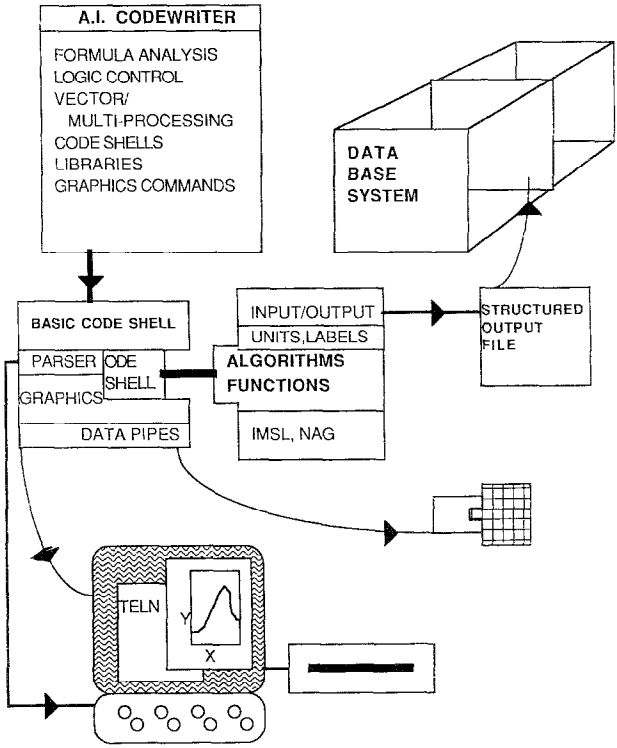

FIG. 3. Automated Programming. This illustrates some capabilities of an artificial intelligence codewriter, the contents of the basic code shell, and the coupling to a real problem. The completed code can interact with a user, another code, or a structured data base system for storing results.

workstation. The advantage is that all the events the user has initiated can be monitored at whatever rate they run. This makes a computing session very efficient for the user. (The disadvantage is that this can be exhausting for the user!)

A Workstation WIMP Interface

The workstations already offer many software packages which make their functions easy to manage. It would be nice to have the same level of functionality and ease of use in scientific codes and a useful tool would be a general-purpose. workstation WIMP. The WIMP starts as a window with a number of menu and icon boxes around it. The user picks on "menu 1," which first asks to be named—a fusion physicist might call it "B-Fields." A second pick of this menu would then display an empty set of options for the user to fill in. A simple syntax would allow the user to indicate input from a file, setting of a variable or array, or choice of a graphical image to be displayed in the window. In a short session the user would fill in this starter WIMP and build a front end to a complex code with Windows, Icons, Menus, and Prompts. A further session with the customized WIMP is now

able to build a complete—and more error free—data set, perhaps in Namelist format, for running the code. For example, correctly constructing the description of a large molecule for a dynamical study is clearly a sufficiently difficult task that such tools are essential. This sort of data entry is now offered in a number of commercial packages for running Chemistry codes [15], but could be available to all.

Generalized Data Pipes

How should data be passed to or from a code on a supercomputer? The modern compilers produce a complete "symbol table" with the location of every variable in the code. The Symbolic or Dynamic Debugging packages use these tables to inspect the code at any time. A small modification of these packages should allow the user to set any variable, in any routine, at any time, without coding of appropriate READ statements. A more powerful option is to design a Parser which will accept a more general data description with symbolic or more natural language capability. This has been done many times for particular codes but should be part of the general supercomputing environment.

An extension of this idea allows codes to exchange results readily. Often, one needs someone else's code as input or for output of results and a lengthy interface design process takes place. With access through the symbol tables of each code a very general "data pipe" is easily established. This sort of facility would be a simple addition to the capabilities of the Fortran Applications Driver.

Structured Data Files

These techniques work well for codes which require small amounts of data and parameters to define a problem and then run a long time. The output is likely to be much more voluminous, including large arrays and a great deal of graphics. A defect of present systems is that the filing and storage of results is done simply, with access to files whose names can be as short as a single character! A more natural corollary to input through symbol tables is to use structured output files which contain an index and descriptions of the data contained. Again, this is a facility to be included with the FAD and to be utilized by computer-generated codes.

Parallel Processed Graphics

The volume of results from a supercomputer brings us back to graphics: Graphical output is often in hundreds or thousands of pictures and viewing them should be done with intelligent selection mechanisms or with much faster devices able to run at movie speeds. The production of graphical output never affects the algorithms—unless the user could see them in real time and react—and therefore is a prime candidate to be run in parallel on a multi processing machine. The aims of intelligent selection and parallel processing begin to be met through the simplified graphics already described and were partially achieved in an extension to GRAFL-II.

The idea is to output the arguments of calls to the simplified graphics to a file for later processing or, on a parallel machine, to shared memory for processing into

graphics images in parallel. The production code or the problem module then contains no graphics library. The output file is 10–50 times smaller than the picture file, which contains all the fonts, tick-marks, and so on which go with a picture. This compact file is also much easier to store for later study. The GRAFL-II extension uses Namelist style output which is also readily comprehensible with a text editor

The post or parallel processor then takes the graphics text or data and produces pictures. It was easy to write as the simplified graphics has only a few routines and is a higher-level language. The processor is also a complete production graphics editor: Pictures can be picked out by character strings in their titles. Previous results can be recalled, captions changed, and indeed any feature of any picture altered. In interactive mode the user can create new pictures or insert whole other data bases to be plotted. Another advantage is that the picture language is the same and the call

```
CALL PCURVE ('LINLOG', XAXIS, GAMMA, NX, LAY
.'Functions of X$', 'X$', '; g < (X) $')
```

becomes

```
&PLOT
    PCURVE = 1 LL = 'LINLOG' X = 1. 2. 3. F = 2.3E – 4 2.7E – 4 6.8E – 3
    NX = 3 LAY = 2 TITTLE = 'Functions of X$' XAX = 'X$' YAX = '; G < (X) $'
&END
```

to plot a curve with the indicated three points. The variable names had to be fixed for interaction with the processor. This suggests how a parser might replace the namelist syntax by a more compact and operational form including symbolic names.

The post processor is a natural way to make movies. Consider a calculation of 50 times steps with 20 different pictures at each step. The following commands should generate a 50-second movie of the "Magnetic Field" alone:

```
&PLOT
COMMENT = 'STRIP OUT THE B-FIELD INTO A NEW FILE'
OUT = 'BFIELD' FIND = 'MAGNETIC' PLOT = 'NO'
&END
&PLOT
COMMENT = 'OPEN OUT FOR INPUT'
INPUT = 'BFIELD'
&END
&PLOT
COMMENT = 'MAKE POST PROCESSOR INTERPOLATE 16 FRAMES
BETWEEN PICTURES'
FILL = 16 PLOT = 'YES'
&END
```

The post processor does three separate tasks or calls to STEPON: Pick out the pictures by name and put them in a separate file. Close that file for output and reopen it for input. Read the file BFIELD two pictures at a time and interpolate linearly in the data to generate 16 smoothly varying pictures. This is only possible because the graphics were preserved as structured data, not as a processed "metafile." The namelist structure is somewhat inefficient for large volumes of data but is readily replaced.

The parallel graphics processor can have similar capabilities of both intelligent selection and multiple output streams. In its simplest operation, a code using the simplified graphics need only be loaded with a different, parallel processing library to run parallel. This could be done for all users as they migrate from single to multiprocessing supercomputers.

9. Concluding Remarks

The results and references given here are chosen to make the case that supercomputing needs to be made easier as the machines get more powerful. The next generation machines have outstripped our ability to utilise them fully. Keith Roberts would have had a lot to say about it.

References

1. J. P. Christiansen and K. V. Roberts, *Comput. Phys. Commun.* 7, 245 (1974).
2. B. McNamara and P. A. Willman, "FFP: Friendly Fortran Programming," UCRL-88387, Lawrence Livermore National Laboratories, 1982 (unpublished).
3. "The Fortran Applications Driver," John von Neumann Center, 1987 (unpublished).
4. "User Guide to the John von Neumann Center," 1986 (unpublished).
5. B. McNamara and P. A. Willmann, "GRAFL-II: User Oriented Science Graphics," UCRL-20467, Lawrence Livermoore National Laboratories, 1985 (unpublished).
6. A. Barr and E. A. Seigenbaum, *The Handbook of Artificial Intelligence* (W. Coffman, Los Altos, CA, 1982).
7. C. W. Gear, *Numerical Initial Value Problems in Ordinary Differential Equations* (Prentice–Hall, Englewood Cliffs, NJ, 1971).
8. IMSL, Houston, TX.
9. NAG: The Numerical Algorithms Group Inc., Downers Grove, IL.
10. MACSYMA, The Mathlab Group, Laboratory for Computer Science, MIT, Cambridge, MA, 1983.
11. A. C. Hearn, "REDUCE Users Manual," UCP-19, University of Utah, Salt Lake City, 1973 (unpublished).
12. M. C. Wirth, "On the Automation of Computational Physics," UCRL-52996, Lawrence Livermore National Laboratories, 1980.
13. G. Cook, "Development of an MHD Code for Axisymmetric, High-Beta Plasmas with Complex Magnetic Fields," UCRL-53324, Lawrence Livermore National Laboratories, 1982.
14. T. H. Einwohner and B. McNamara, Automation of Fortran programming for solving differential equations by MACSYMA, in *10th Conference on Numerical Simulation of Plasmas, General Atomic, San Diego, CA*, 1983.
15. Polygen Corp., Waltham, MA (1987).