3. Computers, Codes, and Engineering

3.1 General Comments

Success in using the methods of computational aerodynamics also depends on an ability to use computers effectively. In this chapter we present some guidelines for the effective use of computing systems. Software development and computing systems are called the *tarpits of engineering* by Brooks.[1] He describes the problems of software development through analogy with the ancient tarpits. Figure 3-1, from Brooks' book, shows the prehistoric beasts, completely bogged down in the primordial ooze. Almost all computer software development jobs get bogged down in a similar quagmire. His book of essays on software engineering is required reading in many places. Unfortunately for the beginner, a true appreciation of the essays comes only with experience. Reading Brooks's book may help avoid the stickiest traps. Students should understand that software development and maintenance/support costs completely overshadow the cost of computer hardware.

A great,
but copyrighted,
figure of the grand creatures, bogged down in the tar.

Figure 3-1. Mural of the La Brea Tarpits, by C. R. Knight, from Brooks[1].

Before providing detailed suggestions for code development and use, a couple of comments based on previous experience with students is in order:

i. Accuracy. Students are told a lot about roundoff error. In making students understand that computers process finite length numbers, the message students get is that computers aren't accurate. Rubbish. Roundoff error has become a favorite excuse (aerodynamicists frequently use unknown Reynolds number effects in a similar fashion). Don't accept three place accuracy. The computer is not a slide rule. More than likely, poor accuracy indicates bugs. At the very least it denotes poor numerical practice, which should be fixed before you find the case where the code goes entirely wrong. The issue here is not whether the theory warrants plotting the output to more than three places, the issue is whether the code is correct. When roundoff error is a problem, it usually arises as the result of taking the difference of two large numbers. In aerodynamics, most algorithms operate accurately using 64-bit arithmetic. This is single precision on scientific machines (Cray), but is known as double precision on commercial machines (IBM). Double precision must be specified to use 64-bit arithmetic on those computers. Investigation of the accuracy of a particular computer is left to the reader as an exercise.

A Story There were all sorts of rumors about the poor accuracy of the trigonometric functions in Applesoft, the Apple version of BASIC. So after I got an Apple][computer, and converted an airfoil analysis program that I had developed for a programmable calculator, I was willing to accept 2~3 place accuracy. After all, it was BASIC, and this method relied heavily on trig functions. Telling this story at work, a colleague winked knowingly, shook his head and told me to find the bug. I gave it one more try. The result? I found the bug. In fact, I learned something important: the order of precedence of operations in BASIC (the unary minus in particular) as compared to FORTRAN. It was just luck that the test case agreed as closely as it did. The moral? More often than not, it's not machine roundoff that causes poor accuracy!

ii. Disdain for "canned" programs. This may reflect some instructor's attitudes. Students should realize they won't be the authors of most programs they use. At best they will be making modifications and fixing bugs. They may be combing existing codes to develop design systems. Current codes are often the result of many man-years of development work (even hundreds of man-years for some engineering codes). Adopt a positive approach to using other people's codes. This requires learning how to use the code, demonstrating a desire to make it work on your specific problem (this often requires considerable ingenuity), and knowing how to check code accuracy against other results. You must gain confidence in a code before making an engineering decision that may mean millions of dollars, possibly lives and the future of the company (an exaggeration, but not an excessively large one). Because of the importance of maintaining software integrity, many engineers are not even allowed to modify the source codes they use. Frequently they won't even be allowed to see the code.

iii. Time. Trying to use a computer program for the first time takes more time than you expect. Working with computers is a sequential (and intense) process. It's very hard to skip steps. When you know you are going to need to use a program, try it out as soon as you can. Don't delay. Brooks[1] poses the question (and answer): “How do computer projects get a year behind schedule? One day at a time.” There are almost always unexpected delays. Letting problems slide is a sure recipe for disaster. This is especially true when a student waits until the night before an assignment is due to try out a program for the first time. Usually, difficulties can be easily resolved if you can contact someone, or if you can step back and calmly reassess the situation. That’s hard to do in the middle of the night with a deadline looming. This also holds true for developing codes. Code development is deceptively time consuming. A good rule of thumb states that the last 10% of the code development work takes at least 90% of the time.

iv. UNIX. UNIX is the current operating system of engineering. Learn UNIX and the **vi** editor. Without this skill you won’t be an effective engineer. This is the only operating system that is nearly universal. It’s used on all workstations and most advanced computing machines.

v. It’s a dynamic world.[*] Computational aerodynamics codes are always changing. Every new problem seems to require code extensions. One problem with computer science majors working on codes (or directing software projects) is their assumption that codes are “finished”. In aerodynamics, if a code is used it’s never finished. Someone will always need one more modification. Be prepared to change your code. This also means paying attention to defining versions of programs as well as backing up your programs. In addition, scientific computing is in a period of rapid change. After a long period of thinking in terms of sequential, or “scalar,” computations, most aerospace engineers now have access to computers which offer increased performance through advanced computer architectures. To use these architectures effectively requires using algorithms and software designed to take advantage of the specific machine capability. Examples are vector and parallel processing. Engineers graduating today will be using massively parallel computing machines over a significant portion of their careers. Computational aerodynamics requires that you stay abreast of scientific computing developments.

3.2 Introduction to Software Engineering

The process of developing and maintaining computer programs is known as software engineering. This field is developing approaches to code development that are intended to delay getting bogged down in the tarpits described by Brooks. Before proceeding, we need to outline the elements of software engineering and provide an overview of the proper approach to developing a code which will prove useful after the original programmers have gone on to other projects. Our discussion is based on Chapter 13 of the book by Darnell and Margolis[2] and Chapter 11 of

* The choice of FORTRAN, C or C++ as your programming language is frequently an issue (sometimes an emotional one!). Usually the particular circumstances dictate which language to use. If you know FORTRAN or C, there are many books available to help you learn the other. You can gain proficiency with a few evenings’ study and some practice. Studying code is also valuable, and if the circumstances require you know it, that usually means that lots of code is already available to study. Essentially, FORTRAN is important because many, many existing aerodynamics codes are written in FORTRAN. C is important because there are more graphics and data acquisition software tools written in C. In either case, object oriented programming will be used by engineers in the future, and, if you program, you will continually learn new programming methods. C++ is a little harder to learn if you are used to FORTRAN or C, but may be better for constructing large, complicated, multidisciplinary systems. Although Java is not yet relevant for scientific computing, it could become important because of its rapid development and cross-platform capability. Always be prepared for change.

the book by Stroustrup.[3] The problem is that real software systems are incredibly complex. The "problem analysis, overall program design, documentation, testing, maintenance and management dwarfs the actual writing and debugging of code."[3] Software development is done by people, and relies on common sense and personal commitment. While we list numerous activities below, software development procedures must not be allowed to discourage creativity.

Darnell and Margolis break software engineering into the following elements:

- *Product specification:* The product (program) that is going to be developed must be defined before work starts. Unless the goal of the project is specifically and realistically defined the project will fail. How it looks to the user must be defined before the details of the code required to implement the solution are designed. The user needs to be involved at this point to make sure there is no misunderstanding about exactly what and how information goes in, and exactly what comes out of the program. Vague language must be avoided. Use of "fast" or "easy to use" as specifications will inevitably result in arguments, and possibly lawsuits, when the product is delivered. Nevertheless, the specification will likely be revised as the customer and designers interact during the development. Availability of a new tool results in a change in the process as soon as it's used. The specification includes an abstract of the problem, the equations to be solved, the input and input interface, the operation of the program as it appears to the user (screen design, subroutine calls and argument lists, *etc*.), output file descriptions, error messages, and plans for future extensions.
- *Software design*: Once the product is defined, the software can be designed. This includes the major divisions of functionality, the major data structures, and the numerical/computational algorithms to be used. Quoting Stroustrup,[3] "The most fundamental problem in software development is complexity. There is only one basic way of dealing with complexity: Divide and conquer." Thus the problem should be split up. But to be effective, the communication of the various pieces requires that the interface between pieces be well designed. The result will be a program with a clean internal structure and clear connections.
- *Project planning and code estimation*: The estimation of code development time is a major problem. Even experienced programmers usually grossly underestimate the time a software job will take. One of the problems is the enormous difference in productivity between programmers. Brooks continues to be the key source of insight in this area.[1] One of the keys to tracking software schedules is to use specific measurable milestones. Typically the schedule can be broken into:

  1/3 product specification and scheduling
  1/6 coding
  1/4 component testing and early system testing
  1/4 complete system integration and testing

- *Software tools for software production*: To improve productivity use tools available on your system. These include *lint* programs to double check source code, *profilers* to evaluate where the time is being spent in programs, and tools to examine the function call tree. Most systems have *make* routines[4] which make sure that the latest versions of routines are being used without compiling the entire code after every change. Learn how to use these tools.

- *Debugging techniques*: Software development suffers from poor productivity. Research is continually being directed toward ways to improve productivity. Development and use of debugging techniques is one area where we can expect continual improvement. Compilers generally include debuggers. Take advantage of the best debuggers available.
- *Testing*: Software validation is a difficult job. "The program that has not been tested does not work."[3] The code must be correct, and it must be usable. The developers need to establish a set of test cases to use during code development. Once the accuracy of the code is established, the usefulness of the code is evaluated by having others use it. This usually involved an alpha and a beta test group. The alpha group is usually part of the organization that developed the code, while the beta group is usually made up of customers for the product. Users that call the developers frequently to complain are often selected to be part of the beta group. It is amazing how many problems these groups can unearth. Although bugs are never completely eliminated, the problems found in the testing process can quickly reduce the initial bugs that are found after the product is released. Special considerations for computational aerodynamics codes will be described below.
- *Performance analysis*: particularly in computational aerodynamics, the time and memory required to solve problems must be defined and evaluated compared to other methods. The question is always going to be asked. You must have an answer, and it will help determine if the new code will be competitive.
- *Documentation*: The methods and the code should be documented separately. The product specification should include most of this information. The user's manual should strike a balance between being too long, discouraging use, and being too short, so that it doesn't help the user. Generally the user's manual and overview documentation should be written by a new user of the code. The developers and long-time users do not bring the viewpoint of a new user to the documentation, and usually do a poor job. Documentation should include sample input and output files for the key cases and options available to the user.
- *Source control and organization*: As the code development effort proceeds the code changes become hard to keep track of, especially when the work is done by a team. Once the initial code development effort is completed, the code will be changed much less frequently, often by people not on the original development team. Without a formal process to track the history of the code changes and to ensure that the proper version is distributed there will be problems. In UNIX there is a system known as sccs/rcs which can be used.[5] Commercial products are available to help do the so-called version control. Use of version control requires self-discipline that is difficult to do in a student environment. But development of good habits from the beginning will greatly improve the development of of a professional approach to software engineering.

Above all, any code should be developed for:

- Readability
- Portability
- Maintainability.

When designing a computer solution to a problem, it is important to make sure that the problem is completely defined before the computer programming begins. Evin Cramer of Boeing recently described the proper procedure.[6] Most code projects should adopt a team approach, where the team consists of the core team, an extended team, and the customers. The extended team represents consultants who provide fast answers to questions that arise in the development process. The customer, or user, needs to be involved from the beginning to provide specifications and to make sure that the final product solves the right problem and the interface fits the user expectations.

Tackling the problem, it is important to keep the parts of the problem separate. First define the engineering problem. Next, look for a solution method. Once the solution method is selected, then develop the mathematical definition of the problem precisely. Only now should the code development effort start. In designing the computer code she suggested that modern simulation methods be used, and that simulation (analysis) methods be kept separate from the optimization formulation and strategy. This approach results in a modular and easily upgradeable code.

3.3 Specific Approaches to Code Development

Now we provide methods for code development applicable to computational aerodynamics. Perhaps the most important requirement is to use a disciplined approach. Because programming does not obey a specific natural law, the programmers must establish the process. At the heart of effective programming is self-discipline and personal responsibility. One approach to good programming practice has been developed by Watts Humphrey.[7] Since most code projects will be done by teams, consider the minimum team effort to be divided between interface design, numerical methods coding and code verification.

Considering specifics, engineers should develop a clean, coherent style for their coding. When engineers write poor code they provide the computer science majors with evidence to support their claim that engineers shouldn't be allowed to touch source codes. That's okay if the computer science people work for you. If they take control of the organization, and engineers depend on them for software support, it isn't. Engineers can also work on codes. It simply requires a good common sense approach and self discipline.

The code segments in Figures 3-2 and 3-3 illustrate both the old fashioned, terrible, coding style frequently found in codes written by engineers, and a modern, good, code style. Consider first Figure 3-2. This atrocious example actually exists in a series of widely used codes.

In this example:

- variables names don't mean anything
- the statement numbers are out of order
- the logic is virtually impossible to follow
- computed **go to**s and **arithmetic if**s are used almost exclusively*
- there is no white space or structure to the statements

I was assigned to modify this code (this segment is part of over 2000 lines of similar code) as one of my first assignments in an aerodynamics development group after graduating from school.

* **arithmetics if**s have been declared obsolescent in FORTRAN 90, see Section 3.10.

In contrast, consider Figure 3-3, an example of code in a modern program. Here, the code appears well structured, white space is well used, and the variable names seem to be systematically defined. It's a good example of current code practice. I have also modified this code. Although the program is much longer, the job was much easier.

```
   68 IF(1.-UFUT(K)/RK-1.0E-3)69,70,70
   69 IF(TFUT(K)/RK**2-1.0E-6)71,70,70
   70 IF(K-I(6))72,72,71
        .
      IF(I(9 ))76,76,110
  110 DIV=1./(X+XSTEP-ORDIN(Z,A(2),XFUT))
      IF(I(9 )-1)76,76,77
   77 DIV=SLOPBL(Z,A(2),I(5),XFUT)/ORDIN(Z,A(2),XFUT)
      IF(I(9 )-2)76,76,1055
 1055 DIV=ORDIN(Z,A(2),XFUT)
   76 I1=I(6)+1
      IF(I(6).GE.(I8-1)) GO TO 73
      I(6)=I8-1
   73 I61=I(6)+1
       .
      DO82 J=1,I61
      Y=1.+.25*(UFUT(J+1)+UFUT(J))*(WFUT(J+1)+WFUT(J))
   82 V(J+1)=(UFUT(J+1)*V(J)-Y*(TFUT(J+1)-TFUT(J))-.5*(WFUT(J+1)+WFUT(J)
     1)*(TFUT(J+1)+TFUT(J))*Y*(UFUT(J+1)-UFUT(J))*.5-.25*(UFUT(J)+UFUT
     2(J+1))**2*A(1)*(VDLPDX*(1.-Y*AY/A(12)/(Y-1.))+DIV))/UFUT(J)
   94 A(5)=TAU0/RK2/(1.+RK5)
      I625=I(6)/4
      DO 1056 J=I625,I6
      IF(UFUT(J)/RK-.995)1057,1056,1056
 1057 L=J
      RL=FLOAT(L)
 1056 CONTINUE
          .
      TAU0=TAU0+DTW
      IF(TAU0)9411,9411,9412
 9411 IQ=1
      GO TO 28
 9412 ALPHA=ALPHA+DA
      A162=2.*A(16)
      IF(ABS(DU/UFUT(1))-A(16))1042,1042,1041
 1042 IF(ABS(DTW/TAU0)-A(16))1043,1043,1041
 1043 IF(ABS(DA/ALPHA)-A162)  1044,1044,1041
 1044 WFUT(1)=F7
      TFUTP=TFUT(1)
      TFUT(1)=(TAU0+ALPHA *A(1))/(1.+.5*UFUT(1)*WFUT(1))
      IF(TFUT(1).GT.0.0) GO TO 1039
      RMAX=WR*RMAX
      GO TO 34
1039  CONTINUE
      IF(ABS(1.-TFUT(1)/TOR0)-2.*A(16))1045,1046,1046
```

Figure 3-2. An example of a terrible programming style.

```
c
      if (ivisc(3).gt.0) then

            tau   = vmu(j,i)*const/vol(j,i)*sk(j,i,4)**2
            vnorm = ub(j,i)*sk(j,i,1)+vb(j,i)*sk(j,i,2)+wb(j,i)
            dcx   = dcx+tau*(ub(j,i)-vnorm*sk(j,i,1))
            dcz   = dcz+tau*(wb(j,i)-vnorm*sk(j,i,3))
            dcy   = dcy+tau*(vb(j,i)-vnorm*sk(j,i,2))

                         end if
c
      chdl   =  chdl+abs(sk(j,i,3))*sk(j,i,4)
      swetl  =  swetl+sk(j,i,4)
      cxl    =  cxl+dcx
      cyl    =  cyl+dcy
      czl    =  czl+dcz
   50 cml    =  cml-dcz*(xa-xmc)+dcx*(za-zmc)
      xas    =  xas/float(jte2-jte1)
      yas    =  yas/float(jte2-jte1)
      zas    =  zas/float(jte2-jte1)
      cds    =  cxl*cosa+czl*sina
      cls    = -cxl*sina+czl*cosa
      cms    =  cml
      chds   =  chdl
      swets  =  swetl
      cl     =  cl+cls
```

Figure 3-3. An example of good programming style.

Kernighan and Plauger[8] have written a book containing basic rules for good programming practice. Their book should be read before starting to do serious programming. We repeat some of their rules here:

- Write clearly - don’t be too clever
- Choose variable names that won’t be confused
- Write first in an easy-to-understand pseudo-language; then translate into whatever language you are using
- Modularize. Use subroutines
- Write and test a big program in small pieces
- Make input easy to proofread
- Make sure all variable are initialized before use
- Don’t stop at one bug (keep looking!)
- Don’t test floating point numbers for equality
- Make it right before you make it fast*
- Don’t sacrifice clarity for small gains in efficiency
- Make sure comments and code agree
- Don’t just echo the code with comments—make every comment count

* With proper planning and code design, these shouldn’t have to be contradictory requirements.

Consider also the following rules from Roache[9] which are directed toward CFD:

- Start simple
- Debug and test on a coarse mesh first
- Print out "enough" information:
  - some at each step
  - lots sometimes
  - print good diagnostic functionals
- Always check on the finest mesh possible before releasing code (this is part of the "testing at the boundaries" requirement)
- Test convergence to machine accuracy
- Try to check all option combinations in a production program*
- Check convergence/stability over the widest possible range of parameters
- Test accuracy against:
  - exact solutions
  - approximate solutions
  - experimental data
- Avoid unnecessary hardware dependence

Additional comments on style and language peculiarities of FORTRAN are discussed in the book of *Numerical Recipes*.[10] Here we consider only FORTRAN. Although other languages are becoming more popular for engineering computing, most existing code is written in FORTRAN, and knowledge of FORTRAN is required in computational aerodynamics.

Other good practice:

- Avoid system-dependent code.** Any useful code will be put on different systems. The user's own system will change. Over the long term, system dependencies almost always cause more trouble than the apparent short term gain.
- An exception to the system-dependent rule: Consider using standard math libraries. Computing centers have libraries of mathematical subroutines available for the solution of most standard math problems. These programs are written by professionals, and take advantage of machine-specific advantages of a particular system. They help you avoid numerical accuracy problems. However, *never* use one of these subroutines in your code without first using it in a pilot code on a problem you can use as a check, to make sure you understand how the program is supposed to be used. The text by Kahaner, et al.[11], provides examples of this approach, and a disk of useful subroutines.
- Don't get carried away with the computer science possibilities. Concentrate on the specific development job. Keep it simple. Many very bright engineers have lost the forest for the trees when working on computer codes.

* This is essentially impossible to do with commercial codes, where millions of option combinations may be possible. That's why code design is so important. However, any code should be tested as much as possible. Certainly, a set of standard test cases must be developed to check code modifications (fixing one bug often results in the addition of another).

** An important exception is code written to take advantage of vectorization and parallel processing. If you are using a computer with these features, the code should be modified to use the machine specific techniques to achieve maximum computing speed. However, the compute-intensive portion of any program is usually a small part of the overall code. That's the only portion that should be made machine specific.

Specific programming standards have been established at VPI for instruction in FORTRAN in ESM 3074. Generally these are good rules. However, considering that new engineers with jobs in computational aerodynamics will mainly work with existing codes, students should be exposed to widely used, although poor, programming practices.

Many of the ESM 3074 guidelines duplicate items given above. Several require comments:

Item	Comment
• No FORTRAN 77 extensions allowed	• OK for learning FORTRAN. But some are standard. Should know NAMELIST.* Workstation manufacturers had to add NAMELIST as one of the first upgrades to their systems. It's a very nice way to handle input, and is widely used in existing aerospace engineering codes.
• ALL variables type declared.	• Although good programming practice, it is highly unusual to see this in most existing FORTRAN programs. Students should understand that programs that do not type-declare variables are not "wrong".
• FORMAT statements to be placed together, just before END statement	• This is a holdover from the days of cards. Sometimes it is inefficient at display terminals. FORMATs are best near the WRITE statements (but not obscuring executable code). Another way is to use another series of statement numbers, *e.g.* put all FORMATS in a 2XXX series.
• All DO loops end in CONTINUE	• A little harsh, just make code clear. Indent nested loops. But FORTRAN 90 requires this.
• GO TO statement	• Guideline allows. Use of GO TO should be minimized, if not eliminated. Use IF.. THEN.. ELSE for clarity.
• COMMON	COMMON is still widely used. You should know how they work. Especially since many problems occur due to errors in COMMON block use.
• DIMENSION	Still widely used.
• EQUIVALENCE	Still seen in some codes. Should know. Don't use in computational aerodynamics.
• Computed GO TO/Arithmetic IF**	Still seen, know what they are, don't use.
• Hollerith formats**	Should know, no one would ever use again.

* NAMELIST is standard in FORTRAN 90, see Section 3.10.

** Also declared obsolescent in FORTRAN 90, see Section 3.10.

3.4 Debugging Your Own Code

There is art and talent involved in debugging programs. However, experience is also an important ingredient. This is detective work. One of the problems is the difficulty in distinguishing between errors in an analysis and the code implementation. In general:

- To find out what’s wrong, you need to know what’s right. Check carefully.
- Use modern systems which include debuggers. Learn to use them! Typically they are part of software tools packages to enhance code development productivity. Consider using a *lint* program to examine your source files. These programs were originally developed for C programs, but now there are usually equivalent packages for FORTRAN (FTNCHEK is available on good UNIX systems).
- As a last resort if you don’t have debugging tools, use *lots* of write statements.
- Plot your results! EVERYWHERE.
- Stop and think, be patient and don't panic.
- Don’t wait until the whole code is written, test small parts separately.

3.5 Looking at Other People’s Codes

When you do have the source code, take time to look at the structure of the program and sketch the flow of the information. Also study the program for style. This is the best way to find ways to improve your own programming style. Try to get a feel for the organization of the computation. Make a chart of SUBROUTINE and COMMON Block structure, as well as external I/O unit use. Figure 3-4 illustrates an example of a program tree that I make to help understand program structure. Clearly this is not a conventional flowchart, it’s useful! Table 3-1 provides an example of the related COMMON block map. Both of these provide a basis for finding a bug or modifying a code.

In executing the code (even it you don’t have the source), observe carefully where things start to go wrong. Study your input data carefully. Try changing your input data. With complicated analysis work, start simple and build up, *i.e.*, if an entire airplane is going to be analyzed, do the wing, then the fuselage, and then put them together, *etc.* Identify when the results become “strange”, and always have a mental model of the expected flowfield and a “back of the envelope” idea of what the answer should be.

3.6 Getting help.

Sometimes it helps to let someone else look at the code. One typical problem is not seeing a misspelled variable name. If you wrote the code, and have been staring at it for an hour, you might not see it. You can frequently overcome this problem by looking at an XREF, and spotting a variable used only once. If you want someone else to look at your code, *bring it all*:

- the *exact* source code that was run
- *all* code and input documentation
- the *exact* input file

- *detailed* description of the method of running
- *all* the output, and exact system messages.

Without this information you are wasting someone's time. More often than not, after collecting all the information in preparation for getting help, you'll either find your mistake, or more likely, the next step in the debugging process will become obvious.

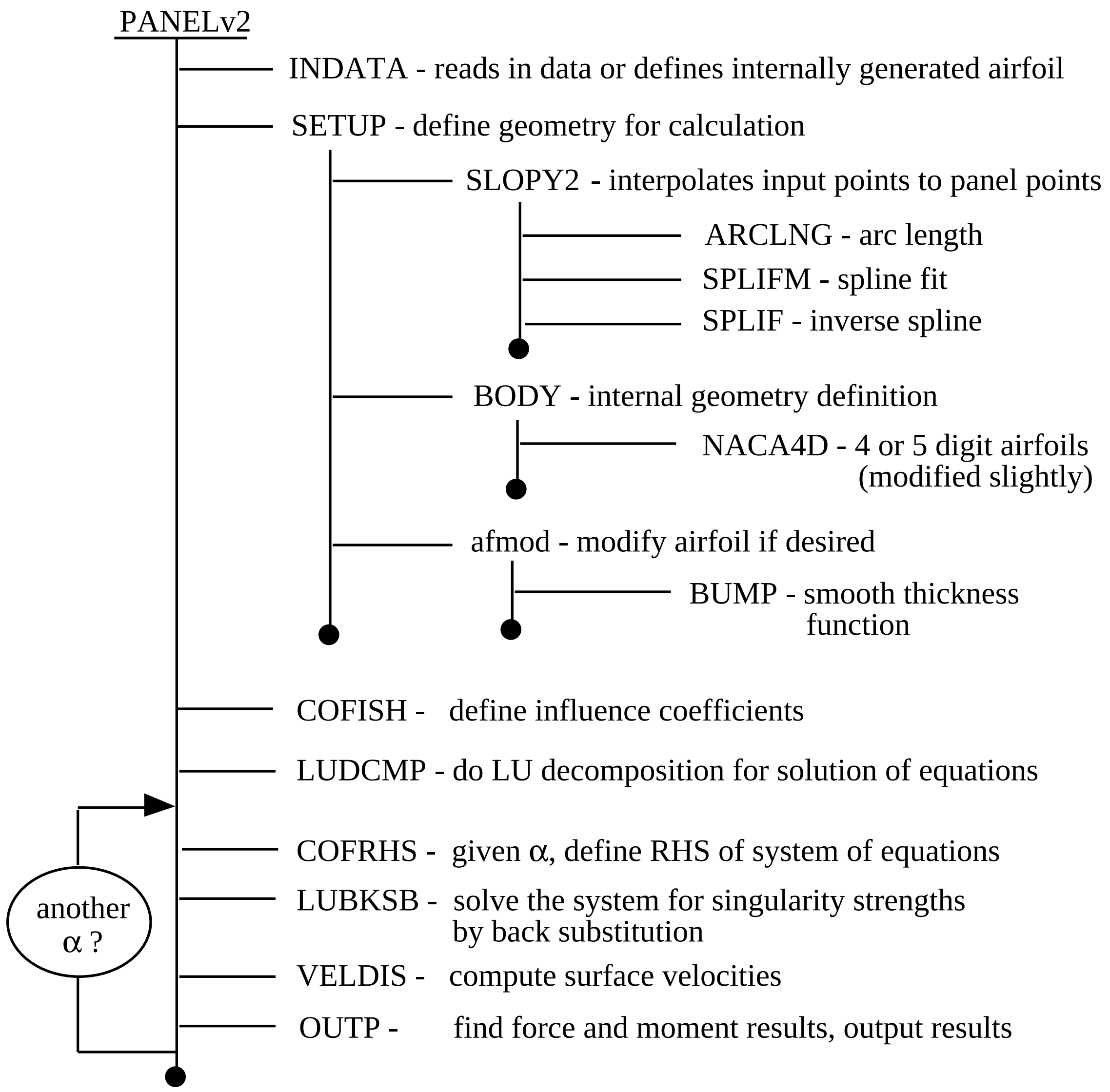
PANELv2
INDATA - reads in data or defines internally generated airfoil
SETUP - define geometry for calculation
SLOPY2 - interpolates input points to panel points
ARCLNG - arc length
SPLIFM - spline fit
SPLIF - inverse spline
BODY - internal geometry definition
NACA4D - 4 or 5 digit airfoils (modified slightly)
afmod - modify airfoil if desired
BUMP - smooth thickness function
COFISH - define influence coefficients
LUDCMP - do LU decomposition for solution of equations
COFRHS - given α, define RHS of system of equations
LUBKSB - solve the system for singularity strengths by back substitution
VELDIS - compute surface velocities
OUTP - find force and moment results, output results
another α ?

Figure 3-4. Routine tree for a small program to help understand the program structure.

Table 3-1

Typical COMMON block map (program PANELv2)

Routine Names	COMMON Block Names						
	BOD	NUM	PAR	COF	ORD	IOU	CPD
PANELv2	X	X		X		X	
INDATA	X		X		X	X	
SETUP	X	X			X	X	
SLOPY2							
ARCLNG							
SPLIFM							
SPLIF							
BODY			X				
NACA4D			X				
afmod						X	
bump							
COFISH	X	X		X			
LUDCMP							
COFRHS	X	X		X			
LUBKSB							
VELDIS	X	X		X		X	X
OUTP	X	X				X	X

3.7 Cost, Time, and Money

A serious effort using computational aerodynamics methods will take a significant investment in your time. Try to make sure that it will payoff. Spend time before starting work assessing whether your approach will produce the results you need. Which method will produce the information in time for it to be useful? There are still many situations where codes can't produce reliable information in time - although in many cases they can.

Next, develop a sense for what the required execution time and resources are to do a calculation. Insight into how long a job takes and how much core storage is required are also keys to effective use of computational aerodynamics. How long will it take to get your results? How much will it cost? Figure 3-5 appeared on my desk one day over a decade ago. Computing deadlines are still with us. Consider any special problems. You may need too much of the machine and be put into a special (slow) queue. Trying to do code work at the last minute is always dangerous. Two stories illustrate that Murphy's Law applied to software work. Once, after getting overtime pay approved and a priority to get fast turnaround, a team of six engineers arrived on Saturday morning, only to discover that the code wouldn't run because a systems programmer took the "clock" routine off the system for the weekend. The disturbance this caused was unbelievable. A contract deadline on an important Air Force contract hung in the balance. We had to get the job done. This is an example where the computer systems people can wreck the job. A more common occurrence is to have systems programmers move codes to different disk packs. Naturally, the engineer working against a deadline will discover this in the middle of the night, with no way to get help.

Don't take getting a new code too lightly. Even when you don't write the code, and think it will be easy to use, before you can use the code you will have to make a surpringly big investment in time before you have enough confidence in the results to use the code to make engineering decisions.

Asking a code developer how long his code takes to run may not make much sense without getting the qualifiers. A code developer may quote a time for a grid too crude to use for an application, or without reaching a reliable level of convergence. What's the batting average? Realize that an advanced code may require many submissions on a single case to get everything straightened out. In those cases, the CPU time of the last submission that produced the final result is not meaningful. You need to consider the CPU time (and calender time) of all the runs leading up to the final run that produced the desired results.

A cartoon of unknown origin:

"People ask me if it's tough meeting a deadline every day.

Frame 1: Man at a terminal, supposed to be a newspaperman

Sure it is,

Frame 2: He reponds

but we *aerodynamicists* thrive on pressure."

Frame 3: Man being carried on a stretcher into the computer room

Figure 3-5. Computational aerodynamics always means deadlines.

3.8 Validation/Verification

Always check a new code against test cases for which you know the correct answers before starting to use it for the analysis cases which made you get it. It is amazing how many times engineers, under time pressure, try to skip this step. Trying to use a code on a project without having prior experience with it is a sure recipe for disaster.

The process of establishing credibility in a computer code has come to be known as code verification/validation/certification. Trying to establish code accuracy on a scientific basis is noble goal, but is a very tricky proposition. Attempts to correlate code results with experimental data on high performance designs that push the flowfield to extreme conditions often shows the importance of using codes and experimental together to understand the relative importance of competing flowfield features.

Establishing a scientific basis for "certifying" codes for use has only recently started to receive significant attention.[12-14] In fact, the semantics of this area are still the subject of discussion, see for example the paper by Roache.[15] In general:

- code verification means solving the specified equation right
- code validation means solving the right equation (modeling the physics correctly)
- code certification means establishing the range of applicability of a validated/verified code

The wide range of ideas about what constitutes code validation is clear in the difference of content in papers addressing the subject. Among numerous papers, the one by Aeschliman, *et al*[16] deserves consideration.

In practice, code users should develop a library of test cases. For a particular code the sensitivity to the key solution control parameters should be well understood. They should also get to know the tricks used in computational aerodynamics. Be wary of code validation cases presented by the code developer. One old trick: a famous transonic test case included data at a span station that showed a double shock. For years the code developers presented results for this case without including this span station of data: the most interesting data on the wing! They got poor agreement at that station, so they ignored the data at that station. Another trick: researchers often compare results with data and not other theories. They may present comparisons of results for Euler or Navier-Stokes solutions without including computations for those same cases from simpler theories. The impression: that they are presenting results for cases that couldn't previously be computed. The truth: often small disturbance, full potential, and full potential/boundary layer methods are able to demonstrate results as accurate as the solution of the more exact equations.[*]

Sometimes using the complete equations precludes the use of enough grid points or sufficient run time to obtain fully converged results using the Euler or Navier-Stokes equations. Thus a more approximate method with improved resolution may be better. Also, some numerical

[*] One of my colleagues who develops CFD codes objects to this statement. However, my obervations at national meetings with numerous presentations on CFD methods continue to confirm this view.

methods used to treat more exact equations introduced more numerical errors than results obtained using better numerical techniques with more approximate equations. This last problem is disappearing with the development of improved numerical methods.

More discussion is presented in Chapter 14, Using Computational Aerodynamics: Review and Reinforcement. There we will provide details of the issues associated with code validation for computational aerodynamics. Appendix B provides references to cases that can be used to validate calculations.

3.9 Presenting Analysis Results: visualization, time, grids and convergence, etc.

Can you understand your results by looking at a table of numbers on a computer screen? Make maximum use of graphics to examine your results. Try to look at all the results. Computer graphics and computational flow visualization provide powerful means of examining results. But be cautious. Contour plots can provide a good means of assessing sharpness of shocks, presence of wiggles, and for checking that there are no incorrect flow gradients next to farfield boundaries. The original standard computational flow visualization graphics package is PLOT3D,[17] which originated at NASA Ames Research Center. PLOT3D has now been absorbed into a newer program: FAST. Computational flow visualization is part of a rapidly growing area known as *scientific visualization*. This area has recently been reviewed by Edwards.[18]

Watch out however, fancy color flowfield plots can be deceiving. Detailed surface pressure and force and moment comparisons should be made with appropriate experimental data and other calculations to assess a code's accuracy. Beware of pretty pictures. They are extremely valuable, but they do not provide the quantitative assessment required for engineering decision making. In a recent article on this subject,[19] we find this statement: "According to Tufte,[*] the danger in the movieland of computer-generated graphics lies in 'dequantification': the numbers get lost in a fascination for shapes and effects. Scales are dropped, meaningless colors are added…." Honest, effective presentation of results requires skill. This is an important part of computational aerodynamics. Globus and Raible have assembled a satirical look at the misuse of visualization in "13 Ways to Say Nothing with Scientific Visualization."[20]

To demonstrate the integrity of the calculation you must always present plots of the convergence history and results of grid convergence studies. Even though you may not be able to afford to converge iterative procedures to machine accuracy for all your calculations, you should demonstrate the effect of not doing this by presenting examples of not doing this for cases representative of the current problems you are studying. Examples of convergence with the number of panels, mesh points, and iterative solution convergence are presented throughout the rest of the text.

[*] Edward R. Tufte has written two fascinating books on graphical presentations: *The Visual Display of Quantitative Information* (1983), and *Envisioning Information*(1990). Both books are published by the Graphics Press, Cheshire, Connecticut. They present a more mature (and honest) view of graphics presentation than is available in most technical publications.

3.10 Modern Computing Developments

Computer architectures for scientific computing are entering a period of rapid change. Traditional scalar processing available on personal computers (and many standard mainframes) is being replaced by vector, parallel, and massively parallel machines. To use these machines effectively, the user must develop methods (and program them) to exploit the specific advantages of these machine architectures. An introduction to these issues for computational fluid dynamics has been given by Rizzi and Enquist,[21] and in an entire AGARD volume.[22] The field is changing quickly. Some of the latest information is contained in a book based on a recent conference edited by Simon.[23] A review of the situation, including descriptions of two current advanced computers, was given in a recent BYTE article.[24] Advances in computational aerodynamics are being driven by the High Performance Computing and Communications (HPCC) Project. An overview of this program is described in the paper by Holst, Salas and Claus.[25] . Here we review a couple of aspects of the computing hardware and languages important to computational aerodynamics.

The key issue in advanced computing is how to increase the computation speed. Several ways of measuring speed are important. This includes the basic processor speed, the size of memory, and the speed of the data transfer through the machine. Although the "raw" computation speed is misleading, it is nevertheless used to quantify the speed of computers. With advanced computers there is a large difference between the peak speed and the maximum sustainable speed obtained in practice. The following table shows the large increases in speed obtained over the last thirty years.

The basic speed is quoted in floating point operations per second, or flops. They are defined as:

flops	or	defined as:
1 million	$1\text{x}10^6$	Megaflop or Mflop
1 billion	$1\text{x}10^9$	Gigaflop or Gflop
1 trillion	$1\text{x}10^{12}$	Teraflop or Tflop

The goal of the HPCC Project is to demonstrate one Teraflop of sustained computing speed.

Typical results obtained by famous machines as initially released have been:

year	machine	speed
1964	CDC 6600	1 Mflop
1968	CDC 7600	4 Mflops
1976	Cray 1	27 Mflops
1983	Cray X-MP	70 Mflops

The current speed record was set recently at Sandia Labs using an Intel Paragon computer. Thus remarkable gains are being made. The history of advances is shown in Fig. 3-6. The top line is the peak performance including advanced approaches (vector and parallel architectures). The bottom curve shows that the scalar, or serial, computing method is starting to reach its limit.

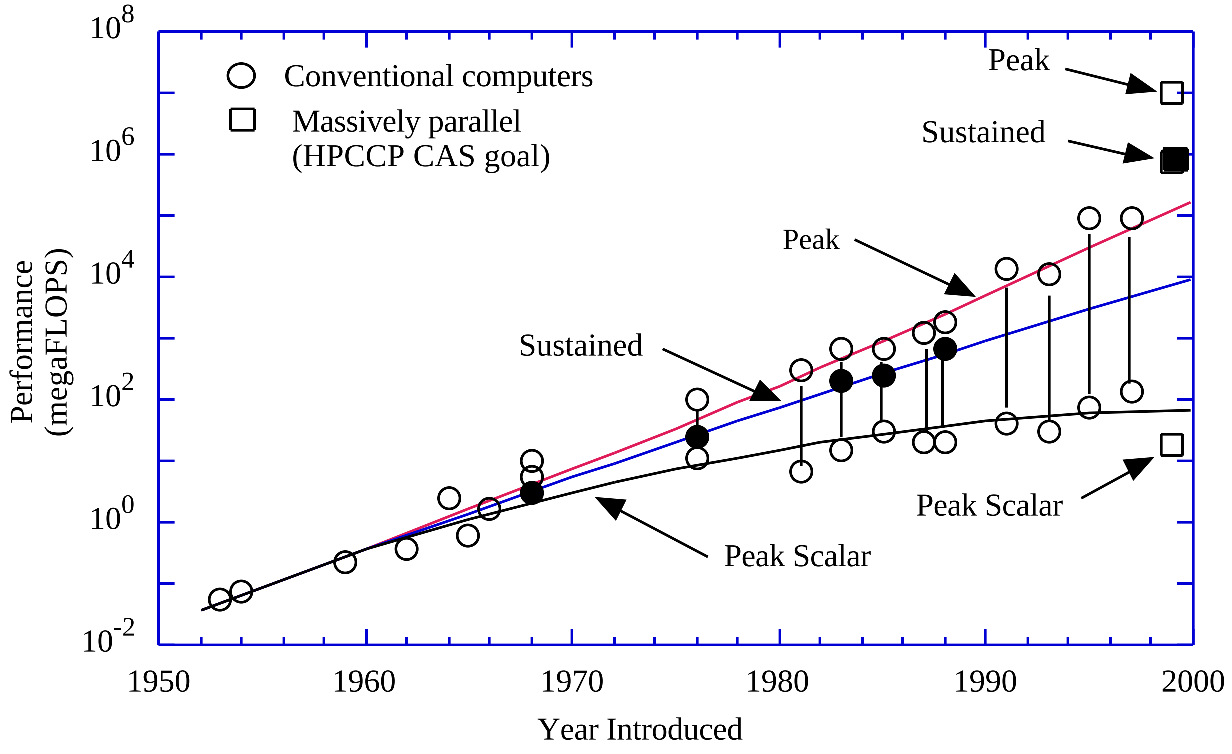

Figure 3-6. The history of computer speed increase and current goals.[25]

Vector Computing

Essentially, the original sequential computing architecture is being replaced by machines which can perform calculations simultaneously. The first step in this direction was the use of *pipelining*. In performing specific operations, the computer may take several *clock cycles* (the basic measure of time on a computer) to complete the instruction. During part of this time some of the CPU may be sitting idle. The idea of pipelining is to take advantage of the idle cycles to begin the next instruction before the machine has completed the previous instruction. This can produce a speedup in the throughput of the CPU. Typically, this procedure is applied to arrays, where the same operation is repeated, and is implemented in DO loops. This is known as code vectorization. To be effective, the operations must allow for simultaneous calculations. If the calculations are not independent, it may not be possible to *vectorize* the loop.

The VPI IBM 3090 computer (**vtvm1**) has vector capability. As an example of the potential of vector processing, the program **vtest**, presented here in Fig. 3-7, was run with several different compiler options (it is in caps because it was copied from the IBM screen).

The following results were obtained using the **fortvs2** compiler:

compiler command	execution time (hundredths of a second)
fortvs2 vtest	889
fortvs2 vtest (novector	892
fortvs2 vtest (novector opt(2)	252
fortvs2 vtest (novector opt(3)	251
fortvs2 vtest (vector opt(3)	33

```
C
C     VECTOR PROCESSING CHECKS - W.H. MASON, FEB. 1, 1992

      PARAMETER (ILOOP = 9000, JLOOP = 1000)
      DIMENSION A(ILOOP),B(ILOOP),C(ILOOP)

      CALL TIMEON

      DO 5 I = 1,ILOOP
      A(I) = I
      B(I) = 2.*I
    5 C(I) = 0.01*I

      DO 20 J = 1,JLOOP
      DO 10 I = 1,ILOOP
   10 A(I) = A(I) + B(I)*C(I)
   20 CONTINUE

      CALL TIMECK(N)

      WRITE(6,100) N
  100 FORMAT(/5X,'TIME = ',I4,3X,'IN HUNDRETHS OF SECONDS'/)

      STOP
      END
```

Figure 3-7. A sample code used to illustrate benefits from vector processing.

These results provide several messages. Clearly, use of the vector compiler option results in a significant reduction in computing time. However, the standard scalar execution optimization option (novector) also makes a large difference in execution time. This simple example illustrates the potential of vectorized computing. Practical cases would not produce this much improvement. This example also illustrates the potential for presenting misleading comparisons. Comparison of unoptimized scalar results to optimized vectorized results overpredict the effects of vectorization by more than a factor of three. Discussions of vectorization can be found in monographs[26] and computer manuals.[27]

Parallel Computers

Another approach to increasing processing speed is to perform calculations on several processors simultaneously. This is known as parallel computing. The recent article by Miel[28] provides a good overview. There are two approaches of interest:[28] “arrays of processing elements operating in unison with a single program (SIMD or Single Instruction Multiple Data), and arrays of cooperating computers running independently with distinct program memories (MIMD or Multiple Instruction Multiple Data).” Parallel computing is becoming practical, and a number of parallel processor computers are now available. Research is currently being conducted to understand how to do computational aerodynamics easily on these machines. Within the very near future computational aerodynamicists will be using these machines routinely. This will require the development of new computer languages and solution algorithms. A major government initiative, the High Performance Computing and Communications (HPCC) program, is addressing these problems. Expect rapid progress. Current practical aspects of parallel computing in aerodynamics are discussed in the newsletter of the Numerical Aerodynamic Simulation Program, located at NASA Ames.

In computational aerodynamics two other aspects need discussion. First, the machines can be used in either “coarse” or “fine” grain parallelization modes. The course-grained mode is of particular interest in aerodynamic and multidisciplinary design. Here, many solutions are required using the same program with different inputs. This is done to find the sensitivity of the design to various design variables. Thus the same code is run on each different processor or node at the same time. This approach is the easiest way to exploit the capability of parallel computing. Fine grain parallel computing requires that the code be changed to make a single calculation using numerous nodes.

The other aspect of concern in parallel computing is scalability. Here, the issue is whether the speedup obtained using a small number of processors can be extrapolated to cases where a large number of processors are used. Experience shows that the performance achieved with a small numbers of processors, say twenty to thirty, does not scale up linearly when hundreds or thousands of processors are used. One standard computer science rule-of-thumb, Ahmdahl’s Law, says that the speedup decreases to a finite limit, which depends on the fraction of the code where serial computations are required. Figure 3-8[25] shows that if even small parts of the code require sequential computation, the speedup using parallel processing will not increase without limit. In the figure R is the fraction of the code requiring serial computation and N is the number of processors used. If none of the code requires serial computation, then $R = 0$, and the linear trend is maintained. Otherwise, a slowdown is inevitable. Some computational scientists are currently trying to demonstrate that for CFD this “law “ is not valid, and the trend can be shown to be approximately $R = 1/N$.

Experience at Virginia Tech using the coarse-grain approach is illustrated in Fig. 3-9.[29] Here we show results obtained by a student, after considerable effort, on the Virginia Tech Intel Paragon parallel computer. The results were obtained for some typical aerodynamics programs. The results look good for this low number of processors. One of the bottlenecks in obtaining increasing speedup with increasing numbers of processors is the use of disk IO by some codes written for older machines.

Finally, note that progress is being made making calculations on networks of workstations. Supercomputing is moving from very large, very expensive machines to distributed processing on machines that individually are much smaller and cheaper.

Language evolution

After years of little change, the computing languages are changing also. There are two reasons for this. First, the C language introduced many desirable features. Many of these have been adopted in FORTRAN 90, which also includes as standard many extensions that were so common that most users thought they were part of FORTRAN 77. Extensions that are now standard include NAMELIST, IMPLICIT NONE and INCLUDE statements. Other new features are: standard calls for date and time routines, new symbols for relational operators, long variable names, and many new intrinsic functions. In addition, several of the worst features of FORTRAN are being declared obsolete, and are likely to be dropped in the future. Free format code is now allowed, so FORTRAN 90 code will not necessarily look like FORTRAN code has in the past. The new capabilities may lead to codes that are 50% shorter than current programs. The second reason for change is the emergence of parallel computers. Although FORTRAN 90 will not address parallel computing explicitly, it provides a basis for compiler builders to develop extensions for use in parallel computing.

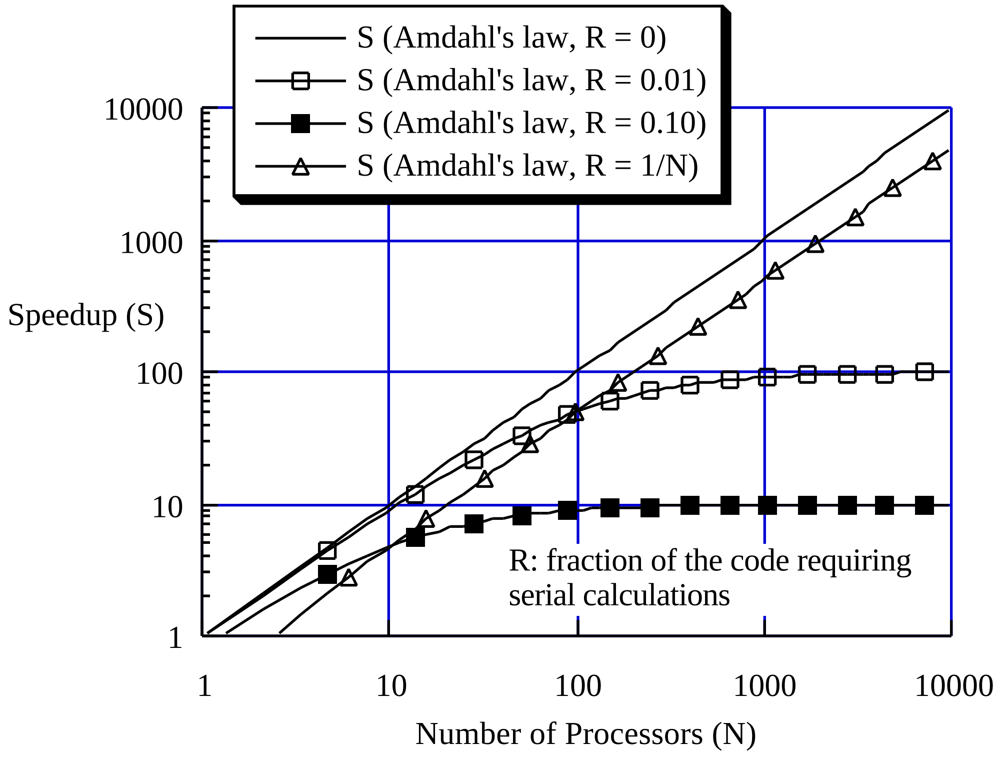

Figure 3-8. Theoretical speed up due to parallelization.[25]

FORTRAN 90 is much bigger than FORTRAN 77, and some effort will be required to learn the new features. One book to read in making the transition is by Kerrigan.[30] In addition, the issues for languages specifically designed for parallel computing are addressed in a recent ICASE Report.[31] The two flavors of FORTRAN being developed for parallel computing are High Performance FORTRAN (HPF) and FORTRAN-S. A standard FORTRAN for parallel computing will not be available for some time. The HPF research is being conducted at the Center for Research on Parallel Computation at Rice University.*

* A web viewer can use the URL: gopher://softlib.rice.edu/ to access reports from Rice. Other reports are available in HTML format via URL: http://softlib.rice.edu/ . A list of reports is available by sending email to softlib@cs.rice.edu. In the message of the body type send trlist.ps. You will get a postscript file with the list of re-

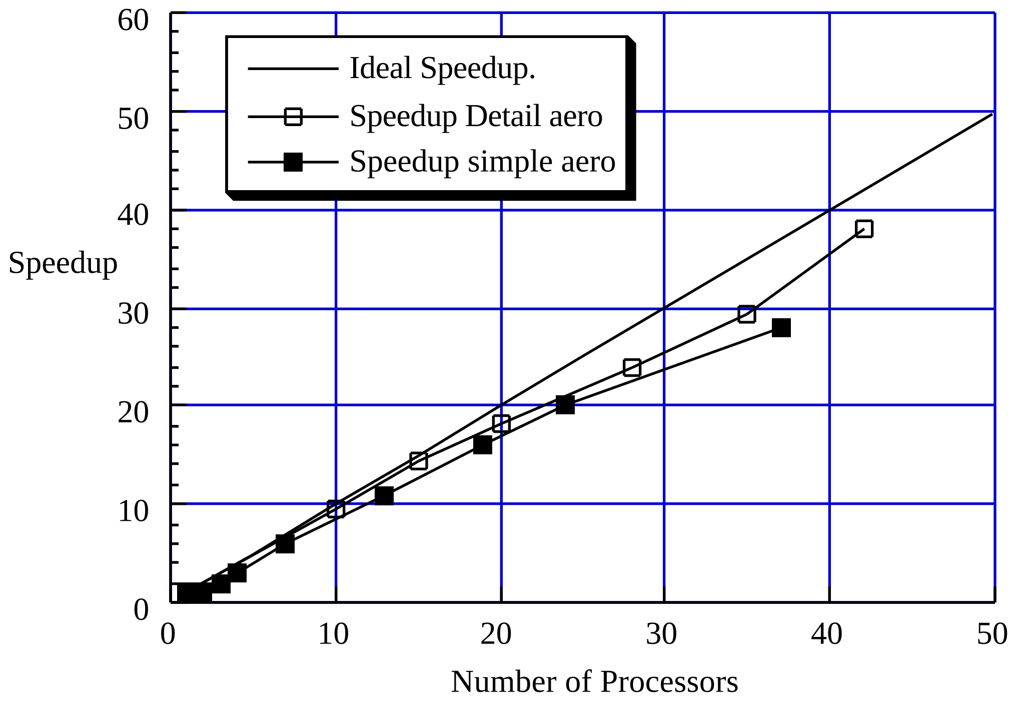

Figure 3-9. Experience at Virginia Tech with course-grained parallel computing.[29]

To keep abreast of current developments in hardware and language developments, students should use the internet. In particular, the web pages associated with NASA's Numerical Aerodynamic Simulation facility (NAS) provide information on current developments. The NAS activity can be reviewed at

http://www.nas.nasa.gov/home.html.[*]

See the NAS Newsletter in particular. Appendix F also has addresses home pages containing the latest information on computational developments for aerodynamicists.

ports.

[*] The addreses are subject to change, this is a very dynamic environment. However, with the addresses of the various pages listed in Appendix F, you should be able to locate these pages.

3.11 Exercises

1. Determine the accuracy of your computer/compiler. Find the size of the smallest number that the computer can distinguish from 1, *i.e.*, what is the smallest value of ε such that it produces the correct result when testing for $1 + \varepsilon > 1$? Recall that to have meaning, the computer and compiler, including the version number, must be included in your summary of results.

   Hint: try a program similar to the following:

```
c
c     compiler precision
c
      write(6,100)
      epsmch     = 1.0

       do 10 i    = 1,999
          epsmch  = epsmch/2.0
          eps1    = 1.0 + epsmch
          if(eps1 .le. 1.0) go to 20
   10    write(6,110) i,epsmch

   20 continue

      write(6,120) epsmch

  100 format(/5x,'Estimate of computer/compiler accuracy'/
     1       /9x,'i',9x,'eps')
  110 format(5x,i5,5x,e14.7)
  120 format(//5x,'Approx. machine zero is ',e14.7)

      stop
       end
```

2. A practical matter: Do not put TAB characters in your FORTRAN code or a data set to be read in by a FORTRAN code. Some editors do this automatically in the default mode. Some compilers allow TABS in source code, many do not. If you have TABS in your code, this severely limits the portability of the code. In this exercise, find out how your editor treats tabs. Write a program to read in a simple data set in 6F10.5 format. Determine what happens if you use TABS to put data in the correct column. Understand this now as an isolated test case, before using codes described later in this text. This will avoid a lot of late night frustration.

3.12 References

1. Brooks, Frederick P., Jr., *The Mythical Man-Month*, Addison-Wesley, Reading, 1975, an Anniversary Edition with additional chapters, 1995.
2. Darnell, P.A., and Margolis, P.E., *C: A Software Engineering Approach*, 2nd ed., Springer-Verlag, New York, 1991.
3. Stroustrup, B., *The C++ Programming Language*, 2nd edition, Addison-Wesley, Reading, 1991.
4. Oram, Andrew, and Talbott, Steve, M*anaging Projects with make*, O'Reilly & Associates, Sebastopol, CA, 1991.
5. Bolinger, Don, and Bronson, Ted, *Applying RCS and SCCS*, O'Reilly & Associates, Sebastopol, CA, 1995.
6. Cramer, E.J., "Airplane Performance Optimization: A Design Case Study," AIAA Paper 95-0465, January 1995.
7. Humphrey, Watts S., *A Discipline for Software Engineering*, Addison-Wesley, Reading, 1995.
8. Kernighan, B.W., and Plauger, P.J., *The Elements of Programming Style*, McGraw-Hill, New York, 1974.
9. Roache, Patrick J., *Computational Fluid Dynamics*, Hermosa Press, 1972.
10. Press, W.H., Teukolsky, S.A., Vettering, W.T., and Flannery, B.P., *Numerical Recipes in FORTRAN: The Art of Scientific Computing, Second Edition*, Cambridge University Press, Cambridge, 1992 (C version also available).
11. Kahaner, D., Moler, C., and Nash, S., *Numerical Methods and Software*, Prentice-Hall, Englewood Cliffs, 1989.
12. Sacher, P.W., "Technical Evaluation Report on the Fluid Dynamics Panel Symposium on Validation of Computational Fluid Dynamics," AGARD Advisory Report No. 257, May 1989.
13. Melnik, R.E., Siclari, M.J., Barber, T., and Verhoff, A., "A Process for Industry Certification of Physical Simulation of Codes," AIAA Paper 94-2235, June 1994.
14. Melnik, R.E., Siclari, M.J., Marconi, F., Barber, T., and Verhoff, A., "An Overview of a Recent Industry Effort at CFD Code Certification," AIAA Paper 95-2229, June 1995.
15. Roache, P.J., "Verfication of Codes and Calculations," AIAA Paper 95-2224, June 1995.
16. Aeschliman, D.F., Oberkampf, W.L., and Blottner, F.G., "A Proposed Methodology for Computational Fluid Dynamics Code Verfication, Calibration, and Validation," International Congress on Instrumentation iin Aerospace Simulation Facilities (ICIASF), July 18-21, 1995, wright=Patterson AFB, OH.
17. Walatka, P.P., Buning, P.G., Pierce, L., and Elson, A., "PLOT3D User's Manual," NASA TM 101067, March, 1990.
18. Edwards, David E., "Scientific Visualization: Current Trends and Future Directions," AIAA Paper 92-0068, Jan. 1992.
19. Patton, P., "Up From Flatland," *New York Times Magazine*, January 19, 1992, pg.61.
20. Globus, A., and Raible, E., "13 Ways to Say Nothing with Scientific Visualization," available as a postscript file from the www page of the National Aerodynamic Simulation (NAS) Facility.

21. Rizzi, A., and Enquist, B., "Selected Topics in the Theory and Practice of Computational Fluid Dynamics," *Journal of Computational Physics*, 72, 1-69 (1987).
22. Neves, K.W., in Chap 2, "Hardware Architecture" in *Computational Fluid Dynamics: Algorithms and Supercomputers*, AGARD-AG-311, 1988.
23. Simon, H.D., ed., *Parallel Computational Fluid Dynamics*, The MIT Press, Cambridge, 1992.
24. Sharp, O., "The Grand Challenges," *BYTE*, Feb. 1995, pp. 65-71.
25. Holst, T.L., Salas, M.D., and Claus, R.W., "The NASA Computational Aerosciences Program—Toward Teraflops Computing," AIAA PAper 92-0558, January 1992.
26. Schofield, C.F., *Optimising FORTRAN Programs*, Ellis Horwood, Ltd., Chichester, 1989.
27. ___, "Vectorization with VS FORTRAN and ESSL," Doc. FT16, User Services Department, Virginia Tech Computing Center, June 12, 1990.
28. Miel, G., "Supercomputers and CFD," *Aerospace America*, January 1992, pp. 32-35,51.
29. Giunta, Anthony A. , Narducci, Robert , Burgee,Susan L. , Grossman, Bernard , Haftka, Raphael T., Mason,William H. , and Watson, Layne T. "Variable-Complexity Response Surface Aerodynamic Design of an HSCT Wing," AIAA Paper 95-1886, AIAA 13th Applied Aerodynamics Conference, San Diego, CA, June 21, 1995 , Pages 994-1002 of Proceedings.
30. Kerrigan, James F., *Migrating to Fortran 90*, O'Reilly & Associates, Inc., Sebastopol, 1993. (the example code used in the book is available electronically for free)
31. Bodin, F., Priol, T., Mehrotra, P., and Gannon, D., "Directions in Parallel Programming: HPF, Shared Virtual Memory and Object Parallelism in pC++", ICASE Report No. 94-54, June 1994. (also NASA CR 194943)