OV

Overview of Computational Science

Copyright (C) 1991, 1992, 1993, 1994, 1995 by the Computational Science Education Project

This electronic book is copyrighted, and protected by the copyright laws of the United States. This (and all associated documents in the system) must contain the above copyright notice. If this electronic book is used anywhere other than the project's original system, CSEP must be notified in writing (email is acceptable) and the copyright notice must remain intact.

1 Introduction

Presently there is no generally accepted definition of Computational Science. In broad terms it is about using computers to analyze scientific problems. Thus we distinguish it from computer science, which is the study of computers and computation, and from theory and experiment, the traditional forms of science. Computational Science seeks to gain understanding principally through the analysis of mathematical models on high performance computers. The term *computational scientist* has been coined to describe scientists, engineers and mathematicians who apply high performance computer technology in innovative and essential ways to advance the state of knowledge in their respective disciplines. More recently, computational science has begun to make inroads into other areas such as economics, music and visual arts.

We shall use the term "computational science" as a convenient shorthand for "computational science and engineering." A well-known characterisation of computational science activities was presented by K. Wilson in 1986 [1]. He summarized the characteristics of computational science problems in whatever discipline as those;

1. having a precise mathematical statement,
2. being intractable by traditional methods,
3. having a significant scope,
4. requiring an in-depth knowledge of science, engineering or the arts.

At that time Wilson also addressed a number of open issues concerning computational science, all of which are still open issues today and will be addressed in the following sections.

1.1 Is there a Computational Science Community?

The computational approach to doing science is inherently *multi*-disciplinary: it requires of its practitioners a firm grounding in applied mathematics and computer science in addition to a command of one or more scientific or engineering disciplines or in the high-tech arts. Thereby the role of computer science is similar to the role of mathematics in the mathematical sciences in that it provides the tools, ranging from networking and visualization tools to algorithms, that match modern computer architectures. Mathematics provides means to establish credibility of algorithms, such as error analysis, exact solutions and expansions, uniqueness proofs and theorems.

At this stage computational science may be thought of as a methodology common to a variety of sciences, which makes use of the same kind of tools. It is conceivable that computational science will one day be redefined as a discipline, following significant breakthroughs and as some of its major challenges are addressed. The computational science community is very diverse and includes researchers with a multitude of area-specific terminology and research methodologies. This community as a whole carries the responsibility to define quality research in this area and to set the standards for publications. The community has the task to assess what activities have value and to communicate results within this very diverse community. Furthermore, there is a need to develop training and education of future practitioners of computational science [3].

1.2 What Role do the Grand Challenges play in the Advancement of Computational Science?

Historically, studies of specific problems and corresponding breakthroughs have led to new scientific disciplines. Physics, Chemistry and Astronomy emerged out of Natural Philosophy following the important discoveries by Newton, LaVoisier and Galileo respectively. The type of problems in which computational science could achieve breakthroughs are generally referred to as 'grand challenges'. A nice definition of grand challenges stems from the Office of Science and Technology Policy:

> Grand Challenges are...fundamental problems in science and engineering, with potentially broad social, political, and scientific impact, that could be advanced by applying high performance computing resources.

As expected, there is a long list of grand challenge problems, including electronic structure of materials, turbulence, genome sequencing and structural biology, global climate modeling, speech and language studies, pharmaceutical design, pollution and dispersion, and many more. The grand challenges supported by the government under the High Performance Computing and Communications initiative are a select set of these.

In 1991 the U.S. Department of Energy created two High Performance Computing Research Centers to serve as intellectual homes for selected grand challenges and to conduct, manage and integrate the research activities necessary to enable progress toward their solution. To appreciate the scope of a grand challenge problem, consider the problem of modeling the global environment. Atmospheric scientists, applied mathematicians and computer scientists from several institutions including NCAR (National Center for Atmospheric Research) and the Oak Ridge National Laboratory collaborate in the CHAMMP project. Studying rainfall patterns is part of this effort. You may view some of their results on the WWW at URL:

- http://www.epm.ornl.gov/chammp/chammpions.html

1.3 How Significant is the Role of Algorithm and Computer Technology Developments?

Due to the continuous synergistic progress in algorithm development, computer technology, and computational science methodology, the scope of large-scale problems such as the one just mentioned can be continuously increased. Examples can be matched to the curve corresponding to the increase in computing power of Cray vector supercomputers.

At the time the Cray 1S emerged, codes forcasting the weather were accurate for no more than 12 hours. The Cray XMP raised that limit to 24 hours and plasma modeling in 2D became feasible. With the Cray 2 weather forcasting for 48 hours, modeling in chemical dynamics and estimate of the Higgs Boson Mass were addressed. The Cray YMP allowed 72 hour weather forcasting, and 2D nonlinear hydrodynamics. The C90 has made pharmaceutical design, and vehicle signature feasible.

Similarly, following the curve of Intel's massively parallel systems, there are examples of problems that could be addressed in an increasing scope as the has computing power increased. [2] In 1952 Hartree had recognized the importance of algorithm development. His vision was that:

> "With the development of new kinds of equipment of greater capacity, and particularly of greater speed, it is almost certain that new methods will have to be developed in order to make the fullest use of this new equipment. It is necessary not only to design machines for the mathematics, but also to develop a new mathematics for the machines."

Among examples of the most significant developments in algorithms are the widely used Metropolis, FFT and multigrid algorithms. The Metropolis algorithm grew out of physical chemistry in 1950's through attempts to calculate statistical properties of chemical reactions. It is now used in a wide range of areas, including astrophysics, many areas of engineering, and chemistry. The FFT (Fast Fourier Transform), an implementation of Fourier Analysis, is used in signal processing, image processing, seismology, physics, radiology, acoustics and many other areas. The more recent multigrid method for solving a wide variety of partial differential equations is now applied to problems in physics, biophysics and engineering. Since algorithms emerging in one discipline are often adapted to problems in other disciplines, it is important to propagate such research results throughout the community.

Computational scientists can now address problems that could not be solved one or two decades ago, and so computational science has emerged as a powerful and indispensable method of analyzing a variety of problems in research, product and process development, and other aspects of manufacturing. Computational inquiry, in the form of numeric simulation, complements theory and experimentation in engineering and scientific research.

Numeric simulations fill the gap between physical experiments and analytical approaches. Numeric simulations provide both qualitative and quantitative insights into many phenomena that are too complex to be dealt with by analytical methods and too expensive or dangerous to study via experiments. Some studies, such as nuclear repository integrity and global climate change, involve time scales that preclude the use of realistic physical experiments. Additional support for numeric simulation stems from the increasing frequency with which simulations are providing results of comparable accuracy to physical experiments.

Thus we can now address some of the outstanding issues, to which conventional approaches have proven inadequate, and we can formulate and investigate new questions which would not even be asked in the absence of our current and expected future computational tools and methods. Computational Science has moved to the leading edge of science and engineering, and it is likely to remain there for a long time to come. As its supporting computing and communications technologies pervade our society, computational science will increasingly be put to many uses, ranging from entertaining videogames running on VLSI chips to novel medical imaging equipment.

1.4 Does Computational Science have Language Requirements?

As Wilson pointed out, there has been a lack of an effective language to communicate new ideas and results in computational science. This is in part due to the diversity of the computational science community.

Currently the most popular languages for computational scientists are Fortran77, C, and C++. Fortran does not allow the programmer to structure programs so that they reflect the logical order of ideas involved in addressing a problem, and it is often complicated to delineate new contributions and modifications to existing large scientific Fortran codes. Due to this inflexibility Fortran appears to be loosing ground, even though F77 is much easier to use than C++, which is growing in popularity. From a computer science viewpoint, there is very little difference between C and Fortran, except that C has marginally better facilities for structuring programs than the original Fortran77. Much more significant are the differences between C and a parallel C, or between Fortran and a parallel Fortran, and also the differences between the various styles of parallel programming. These differences are already having a big impact on software development for computational science, and it is not at all clear which "new practices" are the best. But it seems inevitable that significant changes in programming practice will have to include an increased familiarity with parallel machines and thus parallel programming.

We can observe compelling reasons for developing a language that provides both, programming flexibility and structure. Numerous efforts to provide more practical programming languages have led to a variety of innovative developments. Fortran90 is now emerging as a powerful scientific programming language, incorporating some of the best features of F77 and

C. Developments in the area of literate programming strive for both flexibility and structure. The notion of literate programming, that like is grouped with like, and that logical structure takes precedence over details, is in line with present trends to move towards object-oriented languages such as C++. Thus it is possible to write pseudo object-oriented code using literate programming methodologies. One example is FWEB, where multiple languages are supported within a single file (C, C++, F77, RATFOR, Fortran90, TeX, etc.) providing the greatest amount of flexibility to the programmer. (Look for the upcoming chapter on Literate Programming).

The three main classes of parallel programming we can observe at this time also exhibit a variety of approaches to address the language issues:

data parallelism the easiest to learn, and arguably the easiest to write, debug, and tune, provided the problem maps well to this style. Languages for this type of parallelism are simple extensions of the corresponding sequential language. Examples are C$\star$, Dataparallel C, and MasPar's MPL as extensions of C; pc++ as an extesion of C++; and the parallel array operations in HPF and Fortran90.

parallel libraries runtime libraries such as PVM, P4, Linda, and MPI, which have procedures that can be called from any language. Users have to explicitly parallelize their code, and contend with synchronization problems. There are two main classes within this group – shared memory and message passing – but as far as programming language issues are concerned, both classes are implemented by augmenting an existing sequential language with library routines for creating and coordinating parallel tasks.

new high level languages with implicit parallelism the functional and logic programming languages often fall into this category. This approach requires programmers to learn a whole new programming paradigm, not just a new language sytnax, but adherents claim the effort will be worth it in the long run.

In summary we observe the lack of a common programming style among computational scientists and a resulting breakdown of efficient communication within the community.

1.5 Summary

Computational science has developed much like a complex organism It was born in the 1940s, cutting its teeth on the ballistics and nuclear weapons design problems of World War II. It went through its adolescence during the 1970s and 80s, where it began making an industrial impact in fields such as commercial aircraft design. It is just now reaching its prime of life and is poised to take advantage of rapidly developing computing and communications infrastructure to secure its role as a major contributor to national and world economies.

This timing is opportune. We are at a point, late in the industrial revolution, where further refinements to our current systems are costly and yield small returns. The conventional approaches to problem solving, theory and experiment, are quite mature. Most of the problems which are tractable by these approaches have already yielded to them. Yet

Table 1: Applications of Computational Science

Established	*Emerging*
Computational Fluid Dynamics	Biology
Atmospheric Science	Economics
Seismology	Materials Research
Structural Analysis	Medical Imaging
Chemistry	Animal Science
Magnetohydrodynamics	
Reservoir Modeling	
Global Ocean Modeling	
Environmental Studies	
Nuclear Engineering	

we are faced with a growing number of complex issues which require immediate attention, ranging from securing future sources of energy through managing a highly interdependent collection of national economies to understanding the impact of human activity on our global environment. There is increasing awareness that the scale and scope of the problems which computational science will ultimately address is such that success is dependent on the establishment of effective collaborations among government, academia and industry. It is only through a common, coordinated effort that adequate resources and skills can be brought to bear on such problems. Various applications disciplines are at different stages in their assimilation of computational science (see table 1). Some, like aerodynamics, have fully integrated it into their culture. Others, like oceanography, have more recently begun to recognize its potential as a consequence of pioneering efforts, such as global ocean modeling. In still other disciplines, such as many of the life sciences, much work remains to be done in determining the role of computational science. Despite the varying degrees to which computational science has currently penetrated particular applications disciplines, the evidence that it will have a fundamental impact on virtually all disciplines is clear.

2 Computational Science $\neq$ Computer Science

Computational science should not be confused with computer science. Computer science is the study of algorithms, languages, and machines for solving problems. It is related to, but distinct from, computer engineering, which focuses on the design and construction of computing machines. Computational science focuses on a scientific or engineering problem and draws from computer science and mathematics to gain an improved understanding of the problem area. Even though computational science is quite distinct from most present day computer science, many of the topics typically considered to be in the domain of computer science are of much value to the computational scientist. For example, when choosing a numerical algorithm to map to a particular computer architecture, the computational scientist must be aware of fundamental issues from areas such as data structures and software

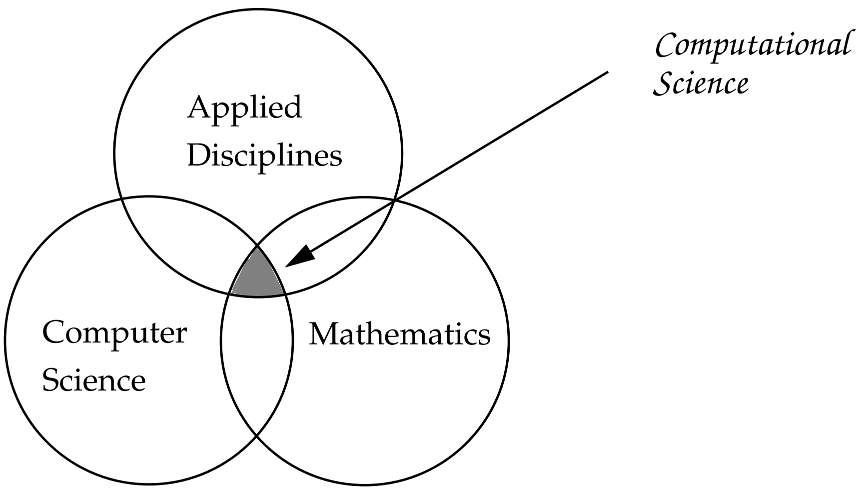

Figure 1: Computational Science

applied discipline. The effective computational scientist must also be familiar with leading edge computer architectures and the data structure issues associated with those architectures. A computational scientist must have a good understanding of both the analysis and implementation of numerical algorithms and the ways that algorithms map to data structures and computer architectures. A familiarity with visualization methods and options is also necessary for computational research. For instance, recently, scientific visualization for the preprocessing of data sets and the interrogation of massive amounts of computational results has become an essential tool of the computational scientist. Thus a computational scientist works in the intersection of (1) an applied discipline; (2) computer science; and (3) mathematics. Computational science is a blending of these three areas to obtain a better understanding of some phenomena through a judicious match between the problem, a computer architecture, and algorithms.

3 A Brief History of Computer Technology

A complete history of computing would include a multitude of diverse devices such as the ancient Chinese abacus, the Jacquard loom (1805) and Charles Babbage's "analytical engine" (1834). It would also include discussion of mechanical, analog and digital computing architectures. As late as the 1960s, mechanical devices, such as the Marchant calculator, still found widespread application in science and engineering. During the early days of electronic computing devices, there was much discussion about the relative merits of analog vs. digital computers. In fact, as late as the 1960s, analog computers were routinely used to solve systems of finite difference equations arising in oil reservoir modeling. In the end, digital computing devices proved to have the power, economics and scalability necessary to deal with large scale computations. Digital computers now dominate the computing world in all areas ranging from the hand calculator to the supercomputer and are pervasive throughout society. Therefore, this brief sketch of the development of scientific computing is limited to the area of digital, electronic computers.

The evolution of digital computing is often divided into *generations*. Each generation is characterized by dramatic improvements over the previous generation in the technology used to build computers, the internal organization of computer systems, and programming languages. Although not usually associated with computer generations, there has been a steady improvement in algorithms, including algorithms used in computational science. The following history has been organized using these widely recognized generations as mileposts.

3.1 The Mechanical Era (1623–1945)

The idea of using machines to solve mathematical problems can be traced at least as far as the early 17th century. Mathematicians who designed and implemented calculators that were capable of addition, subtraction, multiplication, and division included Wilhelm Schickhard, Blaise Pascal,[1] and Gottfried Leibnitz.

The first multi-purpose, i.e. *programmable*, computing device was probably Charles Babbage's Difference Engine, which was begun in 1823 but never completed. A more ambitious machine was the Analytical Engine. It was designed in 1842, but unfortunately it also was only partially completed by Babbage. Babbage was truly a man ahead of his time: many historians think the major reason he was unable to complete these projects was the fact that the technology of the day was not reliable enough. In spite of never building a complete working machine, Babbage and his colleagues, most notably Ada,[2] Countess of Lovelace, recognized several important programming techniques, including conditional branches, iterative loops and index variables.

[1] Pascal's contribution to computing was recognized by computer scientist Nicklaus Wirth, who in 1972 named his new computer language Pascal (and insisted that it be spelled Pascal, not PASCAL).

[2] Another pioneer with a programming language named after her. Naming languages after mathematicians is somewhat of a tradition in computer science. Other such languages include Russel, Euclid, Turning, and Goedel.

A machine inspired by Babbage's design was arguably the first to be used in computational science. George Scheutz read of the difference engine in 1833, and along with his son Edvard Scheutz began work on a smaller version. By 1853 they had constructed a machine that could process 15-digit numbers and calculate fourth-order differences. Their machine won a gold medal at the Exhibition of Paris in 1855, and later they sold it to the Dudley Observatory in Albany, New York, which used it to calculate the orbit of Mars. One of the first commercial uses of mechanical computers was by the US Census Bureau, which used punch-card equipment designed by Herman Hollerith to tabulate data for the 1890 census. In 1911 Hollerith's company merged with a competitor to found the corporation which in 1924 became International Business Machines.

3.2 First Generation Electronic Computers (1937–1953)

Three machines have been promoted at various times as the first electronic computers. These machines used electronic switches, in the form of vacuum tubes, instead of electromechanical relays. In principle the electronic switches would be more reliable, since they would have no moving parts that would wear out, but the technology was still new at that time and the tubes were comparable to relays in reliability. Electronic components had one major benefit, however: they could "open" and "close" about 1,000 times faster than mechanical switches.

The earliest attempt to build an electronic computer was by J. V. Atanasoff, a professor of physics and mathematics at Iowa State, in 1937. Atanasoff set out to build a machine that would help his graduate students solve systems of partial differential equations. By 1941 he and graduate student Clifford Berry had succeeded in building a machine that could solve 29 simultaneous equations with 29 unknowns. However, the machine was not programmable, and was more of an electronic calculator.

A second early electronic machine was Colossus, designed by Alan Turing for the British military in 1943. This machine played an important role in breaking codes used by the German army in World War II. Turing's main contribution to the field of computer science was the idea of the Turing machine, a mathematical formalism widely used in the study of computable functions. The existence of Colossus was kept secret until long after the war ended, and the credit due to Turing and his colleagues for designing one of the first working electronic computers was slow in coming.

The first general purpose programmable electronic computer was the Electronic Numerical Integrator and Computer (ENIAC), built by J. Presper Eckert and John V. Mauchly at the University of Pennsylvania. Work began in 1943, funded by the Army Ordnance Department, which needed a way to compute ballistics during World War II. The machine wasn't completed until 1945, but then it was used extensively for calculations during the design of the hydrogen bomb. By the time it was decommissioned in 1955 it had been used for research on the design of wind tunnels, random number generators, and weather prediction. Eckert, Mauchly, and John von Neumann, a consultant to the ENIAC project, began work on a new machine before ENIAC was finished. The main contribution of EDVAC, their new project, was the notion of a *stored program*. There is some controversy over who deserves the credit for this idea, but none over how important the idea was to the future of general purpose computers. ENIAC was controlled by a set of external switches and dials; to change the program required physically altering the settings on these controls. These controls also limited the speed of the internal electronic operations. Through the use of a memory that was large enough to hold both instructions and data, and using the program stored in memory to control the order of arithmetic operations, EDVAC was able to run orders of magnitude faster than ENIAC. By storing instructions in the same medium as data, designers could concentrate on improving the internal structure of the machine without worrying about matching it to the speed of an external control.

Regardless of who deserves the credit for the stored program idea, the EDVAC project is significant as an example of the power of interdisciplinary projects that characterize modern computational science. By recognizing that functions, in the form of a sequence of instructions for a computer, can be encoded as numbers, the EDVAC group knew the instructions could be stored in the computer's memory along with numerical data. The notion of using numbers to represent functions was a key step used by Goedel in his incompleteness theorem in 1937, work which von Neumann, as a logician, was quite familiar with. Von Neumann's background in logic, combined with Eckert and Mauchly's electrical engineering skills, formed a very powerful interdisciplinary team.

Software technology during this period was very primitive. The first programs were written out in machine code, i.e. programmers directly wrote down the numbers that corresponded to the instructions they wanted to store in memory. By the 1950s programmers were using a symbolic notation, known as assembly language, then hand–translating the symbolic notation into machine code. Later programs known as assemblers performed the translation task.

As primitive as they were, these first electronic machines were quite useful in applied science and engineering. Atanasoff estimated that it would take eight hours to solve a set of equations with eight unknowns using a Marchant calculator, and 381 hours to solve 29 equations for 29 unknowns. The Atanasoff-Berry computer was able to complete the task in under an hour. The first problem run on the ENIAC, a numerical simulation used in the design of the hydrogen bomb, required 20 seconds, as opposed to forty hours using mechanical calculators. Eckert and Mauchly later developed what was arguably the first commercially successful computer, the UNIVAC; in 1952, 45 minutes after the polls closed and with 7% of the vote counted, UNIVAC predicted Eisenhower would defeat Stevenson with 438 electoral votes (he ended up with 442).

3.3 Second Generation (1954–1962)

The second generation saw several important developments at all levels of computer system design, from the technology used to build the basic circuits to the programming languages used to write scientific applications.

Electronic switches in this era were based on discrete diode and transistor technology with a switching time of approximately 0.3 microseconds. The first machines to be built with this technology include TRADIC at Bell Laboratories in 1954 and TX-0 at MIT's Lincoln Laboratory. Memory technology was based on magnetic cores which could be accessed in random order, as opposed to mercury delay lines, in which data was stored as an acoustic wave that passed sequentially through the medium and could be accessed only when the data moved by the I/O interface.

Important innovations in computer architecture[3] included index registers for controlling loops and floating point units for calculations based on real numbers. Prior to this accessing successive elements in an array was quite tedious and often involved writing self-modifying code (programs which modified themselves as they ran; at the time viewed as a powerful application of the principle that programs and data were fundamentally the same, this practice is now frowned upon as extremely hard to debug and is impossible in most high level languages). Floating point operations were performed by libraries of software routines in early computers, but were done in hardware in second generation machines.

During this second generation many high level programming languages were introduced, including FORTRAN (1956), ALGOL (1958), and COBOL (1959). Important commercial machines of this era include the IBM 704 and its successors, the 709 and 7094. The latter introduced I/O processors for better throughput between I/O devices and main memory.

The second generation also saw the first two supercomputers designed specifically for numeric processing in scientific applications. The term "supercomputer" is generally reserved for a machine that is an order of magnitude more powerful than other machines of its era. Two machines of the 1950s deserve this title. The Livermore Atomic Research Computer (LARC) and the IBM 7030 (aka Stretch) were early examples of machines that overlapped memory operations with processor operations and had primitive forms of parallel processing.

3.4 Third Generation (1963–1972)

The third generation brought huge gains in computational power. Innovations in this era include the use of integrated circuits, or ICs (semiconductor devices with several transistors built into one physical component), semiconductor memories starting to be used instead of magnetic cores, microprogramming as a technique for efficiently designing complex processors, the coming of age of pipelining and other forms of parallel processing (described in detail in Chapter CA), and the introduction of operating systems and time-sharing.

The first ICs were based on small-scale integration (SSI) circuits, which had around 10 devices per circuit (or "chip"), and evolved to the use of medium-scale integrated (MSI) circuits, which had up to 100 devices per chip. Multilayered printed circuits were developed and core memory was replaced by faster, solid state memories. Computer designers began to take advantage of parallelism by using multiple functional units, overlapping CPU and I/O operations, and pipelining (internal parallelism) in both the instruction stream and the data stream. In 1964, Seymour Cray developed the CDC 6600, which was the first architecture to use functional parallelism. By using 10 separate functional units that could operate

[3] The term "computer architecture" generally refers to aspects of a computer's internal organization that are visible to programmers or compiler writers; see Chapter CA.

simultaneously and 32 independent memory banks, the CDC 6600 was able to attain a computation rate of 1 million floating point operations per second (1 Mflops). Five years later CDC released the 7600, also developed by Seymour Cray. The CDC 7600, with its pipelined functional units, is considered to be the first vector processor and was capable of executing at 10 Mflops. The IBM 360/91, released during the same period, was roughly twice as fast as the CDC 660. It employed instruction look ahead, separate floating point and integer functional units and pipelined instruction stream. The IBM 360–195 was comparable to the CDC 7600, deriving much of its performance from a very fast cache memory. The SOLOMON computer, developed by Westinghouse Corporation, and the ILLIAC IV, jointly developed by Burroughs, the Department of Defense and the University of Illinois, were representative of the first parallel computers. The Texas Instrument Advanced Scientific Computer (TI–ASC) and the STAR–100 of CDC were pipelined vector processors that demonstrated the viability of that design and set the standards for subsequent vector processors.

Early in the this third generation Cambridge and the University of London cooperated in the development of CPL (Combined Programming Language, 1963). CPL was, according to its authors, an attempt to capture only the important features of the complicated and sophisticated ALGOL. However, like ALGOL, CPL was large with many features that were hard to learn. In an attempt at further simplification, Martin Richards of Cambridge developed a subset of CPL called BCPL (Basic Computer Programming Language, 1967). In 1970 Ken Thompson of Bell Labs developed yet another simplification of CPL called simply B, in connection with an early implementation of the UNIX operating system. comment):

3.5 Fourth Generation (1972–1984)

The next generation of computer systems saw the use of large scale integration (LSI – 1000 devices per chip) and very large scale integration (VLSI – 100,000 devices per chip) in the construction of computing elements. At this scale entire processors will fit onto a single chip, and for simple systems the entire computer (processor, main memory, and I/O controllers) can fit on one chip. Gate delays dropped to about 1ns per gate.

Semiconductor memories replaced core memories as the main memory in most systems; until this time the use of semiconductor memory in most systems was limited to registers and cache. During this period, high speed vector processors, such as the CRAY 1, CRAY X–MP and CYBER 205 dominated the high performance computing scene. Computers with large main memory, such as the CRAY 2, began to emerge. A variety of parallel architectures began to appear; however, during this period the parallel computing efforts were of a mostly experimental nature and most computational science was carried out on vector processors. Microcomputers and workstations were introduced and saw wide use as alternatives to time–shared mainframe computers.

Developments in software include very high level languages such as FP (functional programming) and Prolog (programming in logic). These languages tend to use a *declarative* programming style as opposed to the *imperative* style of Pascal, C, FORTRAN, et al. In a declarative style, a programmer gives a mathematical specification of what should be computed, leaving many details of how it should be computed to the compiler and/or runtime system. These languages are not yet in wide use, but are very promising as notations for programs that will run on massively parallel computers (systems with over 1,000 processors). Compilers for established languages started to use sophisticated optimization techniques to improve code, and compilers for vector processors were able to vectorize simple loops (turn loops into single instructions that would initiate an operation over an entire vector).

Two important events marked the early part of the third generation: the development of the C programming language and the UNIX operating system, both at Bell Labs. In 1972, Dennis Ritchie, seeking to meet the design goals of CPL and generalize Thompson's B, developed the C language. Thompson and Ritchie then used C to write a version of UNIX for the DEC PDP–11. This C–based UNIX was soon ported to many different computers, relieving users from having to learn a new operating system each time they change computer hardware. UNIX or a derivative of UNIX is now a de facto standard on virtually every computer system.

An important event in the development of computational science was the publication of the Lax report. In 1982, the US Department of Defense (DOD) and National Science Foundation (NSF) sponsored a panel on Large Scale Computing in Science and Engineering, chaired by Peter D. Lax. The Lax Report stated that aggressive and focused foreign initiatives in high performance computing, especially in Japan, were in sharp contrast to the absence of coordinated national attention in the United States. The report noted that university researchers had inadequate access to high performance computers. One of the first and most visible of the responses to the Lax report was the establishment of the NSF supercomputing centers. Phase I on this NSF program was designed to encourage the use of high performance computing at American universities by making cycles and training on three (and later six) existing supercomputers immediately available. Following this Phase I stage, in 1984–1985 NSF provided funding for the establishment of five Phase II supercomputing centers.

The Phase II centers, located in San Diego (San Diego Supercomputing Center); Illinois (National Center for Supercomputing Applications); Pittsburgh (Pittsburgh Supercomputing Center); Cornell (Cornell Theory Center); and Princeton (John von Neumann Center), have been extremely successful at providing computing time on supercomputers to the academic community. In addition they have provided many valuable training programs and have developed several software packages that are available free of charge. These Phase II centers continue to augment the substantial high performance computing efforts at the National Laboratories, especially the Department of Energy (DOE) and NASA sites.

3.6 Fifth Generation (1984–1990)

The development of the next generation of computer systems is characterized mainly by the acceptance of parallel processing. Until this time parallelism was limited to pipelining and vector processing, or at most to a few processors sharing jobs. The fifth generation saw the introduction of machines with hundreds of processors that could all be working on different parts of a single program. The scale of integration in semiconductors continued at an incredible pace — by 1990 it was possible to build chips with a million components — and semiconductor memories became standard on all computers.

Other new developments were the widespread use of computer networks and the increasing use of single-user workstations. Prior to 1985 large scale parallel processing was viewed as a research goal, but two systems introduced around this time are typical of the first commercial products to be based on parallel processing. The Sequent Balance 8000 connected up to 20 processors to a single shared memory module (but each processor had its own local cache). The machine was designed to compete with the DEC VAX–780 as a general purpose Unix system, with each processor working on a different user's job. However Sequent provided a library of subroutines that would allow programmers to write programs that would use more than one processor, and the machine was widely used to explore parallel algorithms and programming techniques.

The Intel iPSC–1, nicknamed "the hypercube", took a different approach. Instead of using one memory module, Intel connected each processor to its own memory and used a network interface to connect processors. This *distributed memory* architecture meant memory was no longer a bottleneck and large systems (using more processors) could be built. The largest iPSC–1 had 128 processors. Toward the end of this period a third type of parallel processor was introduced to the market. In this style of machine, known as a *data-parallel* or SIMD, there are several thousand very simple processors. All processors work under the direction of a single control unit; i.e. if the control unit says "add `a` to `b`" then all processors find their local copy of `a` and add it to their local copy of `b`. Machines in this class include the Connection Machine from Thinking Machines, Inc., and the MP–1 from MasPar, Inc.

Scientific computing in this period was still dominated by vector processing. Most manufacturers of vector processors introduced parallel models, but there were very few (two to eight) processors in this parallel machines. In the area of computer networking, both wide area network (WAN) and local area network (LAN) technology developed at a rapid pace, stimulating a transition from the traditional mainframe computing environment toward a distributed computing environment in which each user has their own workstation for relatively simple tasks (editing and compiling programs, reading mail) but sharing large, expensive resources such as file servers and supercomputers. RISC technology (a style of internal organization of the CPU) and plummeting costs for RAM brought tremendous gains in computational power of relatively low cost workstations and servers. This period also saw a marked increase in both the quality and quantity of scientific visualization.

3.7 Sixth Generation (1990 –)

Transitions between generations in computer technology are hard to define, especially as they are taking place. Some changes, such as the switch from vacuum tubes to transistors, are immediately apparent as fundamental changes, but others are clear only in retrospect. Many of the developments in computer systems since 1990 reflect gradual improvements

Table 2: Network Speeds

		Transmission Time		
Name	*Speed (bits/sec)*	*24-bit Color Screen*	*Bible*	*Encyclopedia Britannica*
T3	45,000,000	0.5 sec	1.2 sec	60 sec
T1	1,544,000	15 sec	36 sec	30 min
56 kbps	56,000	7 min	16 min	13 hrs
14.4 kbaud	14,400	0.5 hr	1 hr	2 days

over established systems, and thus it is hard to claim they represent a transition to a new "generation", but other developments will prove to be significant changes.

In this section we offer some assessments about recent developments and current trends that we think will have a significant impact on computational science. This generation is beginning with many gains in parallel computing, both in the hardware area and in improved understanding of how to develop algorithms to exploit diverse, massively parallel architectures. Parallel systems now compete with vector processors in terms of total computing power and most expect parallel systems to dominate the future.

Combinations of parallel/vector architectures are well established, and one corporation (Fujitsu) has announced plans to build a system with over 200 of its high end vector processors. Manufacturers have set themselves the goal of achieving teraflops (10^{12} arithmetic operations per second) performance by the middle of the decade, and it is clear this will be obtained only by a system with a thousand processors or more. Workstation technology has continued to improve, with processor designs now using a combination of RISC, pipelining, and parallel processing. As a result it is now possible to purchase a desktop workstation for about \$30,000 that has the same overall computing power (100 megaflops) as fourth generation supercomputers. This development has sparked an interest in heterogeneous computing: a program started on one workstation can find idle workstations elsewhere in the local network to run parallel subtasks.

One of the most dramatic changes in the sixth generation will be the explosive growth of wide area networking. Network bandwidth has expanded tremendously in the last few years and will continue to improve for the next several years. T1 transmission rates are now standard for regional networks, and the national "backbone" that interconnects regional networks uses T3. Networking technology is becoming more widespread than its original strong base in universities and government laboratories as it is rapidly finding application in K–12 education, community networks and private industry. A little over a decade after the warning voiced in the Lax report, the future of a strong computational science infrastructure is bright. The federal commitment to high performance computing has been further strengthened with the passage of two particularly significant pieces of legislation: the High Performance Computing Act of 1991, which established the High Performance Computing and Communication Program (HPCCP) and Sen. Gore's Information Infrastructure and Technology Act of 1992, which addresses a broad spectrum of issues ranging from high performance computing to expanded network access and the necessity to make leading edge technologies available to educators from kindergarten through graduate school.

In bringing this encapsulated survey of the development of a computational science infrastructure up to date, we observe that the President's FY 1993 budget contains $2.1 billion for mathematics, science, technology and science literacy educational programs, a 43% increase over FY 90 figures.

4 The Modern High Performance Computing Environment

The first computer to be termed a "supercomputer" was the CDC 6600, introduced in 1966. Later model CDC 6600s had a peak performance rate of 3 million floating point operations per second, or 3 Megaflops (Mflops). Computers of the 1990s are capable of peak performance rates of one Gigaflop (one thousand Megaflops). Teraflop (one million Megaflops) performance rates are predicted by the turn of the century. Table 3 shows the peak performance rate for some representative machines. With this rapid increase in performance, it has long been recognized that the definition of the term supercomputer must be dynamic. More than just peak performance rates must be considered when designating a computer as a super computer. Other factors that must be considered include memory size and memory bandwidth.

Recently the term "supercomputer" has been displaced by the term "high performance computer" or "high performance computing environment". This shift in terminology has resulted from the recognition that, in a computational science setting, when real problems are being tackled (rather than just CPU benchmarks) it is the entire computing environment that must offer high performance, not just the CPU. In addition to a computer with a high computational rate and a large, fast memory, a high performance computing environment must include high speed network access, reliable and robust software and compilers, documentation and training and scientific visualization support.

In today's high performance computing environment, a computational scientist's routine activities rely heavily on the Internet. Activities include exchange of e-mail and interactive talk or chat sessions with colleagues. Heavy use is made of the ability to transfer documents such as proposals, technical papers, data sets, computer programs and images. A networked high performance computing environment provides the computational scientist access to a wide array of computer architectures and applications. Using `telnet` to connect to a remote computer (on which one has an account) on the Internet enables the computational scientist

Table 3: Peak Performance Rates of Selected Computers

Machine	*Number of Processors*	*Peak Performance (Mflops)*
CDC 6600	1	3
CDC 7600	1	10
CRAY 1	1	160
CYBER 205	1	400
nCUBE/10	1024	500
IBM 3090/VF	6	686
CRAY X-MP	4	940
CRAY Y-MP	8	2664
CM-2	65536	20000
CM-5		

to use all of the computational power and software applications of that remote machine. digits

References

[1] Wilson, K. G. *Basic Issues of Computational Science*, presented at the International Conference In Computational Physics, Trieste, October, 1986.

[2] Grand Challenges: High Performance Computing and Communications. The FY 1992 U.S. Research and Development Program. Supplement to the President's Fiscal Year 1992 Budget. p.7.

[3] Swanson, C. D. *Computational Science Education*, Cray Research Internal Working Document, September 1994.

[4] Decker, J.F., Johnson, G.M., *Computational Science: An Assessment and Projection*, Proceedings of the 2nd International Conference on Computational Physics, Beijing, September 1993, World Scientific.

[5] Goldstine, H. *The Computer from Pascal to von Neumann*. Princeton University Press, 1972. [`A detailed account of early developments in computing. Goldstine was himself a key figure in the ENIAC and EDVAC projects`].

[6] Hayes, J. P. *Computer Architecture and Organization*. McGraw-Hill, 1978. [`Out of date now as a text on computer architecture, but it has a very nice section on historical computers, including details on how Babbage's machine used the method of finite differences to calculate functions`].

[7] Hodge, A. *Alan Turing: The Enigma*. Simon and Schuster, 1983. [`A comprehensive biography of Turing; Hodge is a mathematician who provides a satisfying explanation of Turing's contributions to that field as well as his work on computers and artificial intelligence`].

[8] Slater, R. *Portraits in Silicon*. MIT Press, 1987. [`Biographical vignettes on over 30 influential figures in computer science and the computer industry from Babbage, Turing, and von Neumann to Seymour Cray, Bill Gates, and Ross Perot`].

[9] Wilson, K. G. *Grand Challenges to Computational Science*. Cornell University, May 1987.