9

BACK IN AMERICA 1987–2004

Distinguished professor at Syracuse – Birthday celebration in the former Danish West Indies – Becoming an American citizen – Parallel scientific computing – A personal supercomputer – Parallel cryptography – History of programming languages – The Computer Pioneer Award – Final words.

After the first year in Denmark, Milena and I knew that our family now belonged in the United States. Our children did not feel at home in Denmark. In May 1986, our daughter Mette announced that she was going back, no matter what! From then on, things happened quickly. The American Embassy in Copenhagen informed me that our residence permits (known as "green cards") had expired. We would only be allowed to return if we obtained new immigrant visas by the end of the year and returned to the United States no later than April 1987.

I immediately called Syracuse University (SU), in Central New York, where I knew the computer scientist John Reynolds, and asked if it would be possible to appoint me as full professor within two months—a process that normally takes six months. In September, Milena and I flew to Syracuse where I gave a talk. While Milena looked at houses, I met with the academic vice chancellor, Gershon Vincow, and the faculty of the School of Computer and Information Science. Two weeks later, the interim dean, Ernie Sibert, offered me an appointment as distinguished professor, an honorary title that had only been bestowed on two other professors at the university.

By February 1987, all four of us had returned to America. Milena and I moved into a white colonial on 5070 Pine Valley Drive in the small village of Fayetteville, a short drive from the university. In our backyard we have a large swimming pool (27 by 60 feet). The house lies in a beautiful valley surrounded by tree-topped hills. The unfenced lawns with trees make the neighborhood look like a park.

Syracuse is a city of about 160,000 people in the center of New York state, a five-hour drive from New York City. Until the 1920s the 363-mile long Erie Canal, extending from the Hudson river at Albany to Lake Erie at Buffalo, passed through downtown Syracuse. The climate is similar to the Danish one with tons of snow during the winter and plenty of rain during the spring and summer. The story goes that a Syracuse professor missed the summer one year—it fell on a Tuesday, and he was out of town.

After moving to the East Coast, we spent many vacations in the Caribbean. In November 1988, Milena and I celebrated our 50th birthdays in the U.S. Virgin Islands. We stayed a week at the Morningstar Beach Club on St Thomas. From 1666 to 1917, these islands were known as the Danish West Indies until the United States bought them from Denmark for 25 million dollars. Since the US is not supposed to have colonies, the islands now have the status of "unincorporated territory." In Charlotte Amalie you can still see Jørgen Iversen's Red Fort (1680), built by the first Danish settlers near King's Wharf, the Governor's House (1747) on Kongens Gade, and the Lutheran Church (1820) on Nørregade. Every day, cruise ships arrive at the West Indian Company Dock, and the tourists all head for the duty-free shops on Dronningens Gade.

One day we took the ferry from Redhook Bay to Cruz Bay, the only town on St. John, and hired a tour guide to drive us along Kongevejen through the tropical forest to Coral Bay, where King June and his last followers killed themselves after killing 76 whites and destroying 48 plantations during the slave rebellion in 1733. At Annaberg Plantation, the ruins of a Danish sugar mill, built in the 1780s, have been partially restored. On the north shore are some of the most beautiful beaches in the Caribbean.

We flew by seaplane to St. Croix. On the way from Christiansted to Frederiksted we walked through St. George Village, a tropical garden, landscaped around the ruins of a plantation, built by Governor General Peter Oxhold around 1815. The Whim Greathouse is another plantation, from 1803, completely restored with mahogany furniture and crystal chandeliers. Near the pier in Frederiksted lies Fort Frederik, completed in 1776. Here Governor von Scholten liberated the slaves on July 3, 1848. In Christiansted, we saw the Governor's Residence, Fort Christian, and the Danish Scale House. These yellow-and-white buildings reminded me of the old houses in Frederiksberg, Denmark.

Four years later, on May 19, 1992, I took the oath of citizenship of the United States at the Onondaga County Courthouse in Syracuse.

⋆ ⋆ ⋆

In the 1980s, the early programming problems of operating systems surfaced again in *parallel scientific computing* (also known as *computational science*): there was a serious need for machine-independent programming languages and algorithms. To understand this challenge, I spent five years writing portable parallel programs for typical programs in science and engineering.

As a first step, my student Anand Rangachari and I moved the parallel programming language Joyce from an IBM PC to an Encore Multimax 320, a multiprocessor with 18 processors and 128 Mbytes of shared memory (Brinch Hansen 1989). This machine was designed a few years before I joined SU. It was owned and operated by the Northeast Parallel Architectures Center (NPAC) at Syracuse University.

The only valid reason for using parallel programming in scientific computing is to tackle problems that require more computing power than you can get from a single processor. From that point of view, our experiments with the Multimax were somewhat academic. The Joyce compiler generated portable code which was interpreted by a kernel of 2,300 lines written in assembly language. In theory, the multiprocessor had the potential of making programs eighteen times faster. However, most of the potential speedup was wasted by the portable code, which was an order of magnitude slower than machine code.

Nevertheless, I learned a great deal from this first experiment about the problems of implementing a parallel programming language on a multiprocessor. The main decision issues were: (1) *load balancing*—the number of process scheduling queues required to balance the computational load evenly among the processors, (2) *synchronous communication*—the implementation details of processes exchanging messages through unbuffered channels, and (3) *mutual exclusion*—the number of software locks needed to prevent multiple processors from accessing the same queue or channel at the same time (without slowing the processors unnecessarily down). We settled these issues by performance measurements (Brinch Hansen 1988).

The Achilles heel of the multiprocessor concept was the empirical observation by Intel cofounder Gordon Moore (1979) that the density of integrated circuits had doubled every year since 1958. Moore's law predicted that by 1992 you would be able to buy 1,000 processors for the same price as 10 processors in 1985. And, since nobody believed that a shared memory machine could support that many processors efficiently, multiprocessor architectures appeared to have no future. The catch phrase at the time was that "multiprocessors do not scale up."

One way out of this dilemma was to give up the simple idea of a multiprocessor with shared memory in favor of a *multicomputer with distributed memory.* Such a parallel architecture consists of a bunch of microcomputers, each with its own local memory. The processor nodes communicate by sending messages to their nearest neighbors only through communication links. Each link is a "point-to-point" connection between exactly two nodes. The removal of the bottleneck created by shared memory greatly increased the performance of parallel computers. However, the occasional need to route some messages though a sequence of intermediate nodes made multicomputers far more difficult to program than multiprocessors. So simplicity was sacrificed for performance—what else is new in computing? This compromise has, I believe, doomed computational science to remain an extremely difficult form of programming for experts only.

While we were experimenting with multiprocessing, the possibility of multicomputing had already been explored by a Caltech group headed by physicist Geoffrey Fox and computer scientist Charles Seitz. Together, they pioneered a new parallel architecture known as the *hypercube* (Seitz 1985).

Let me explain what a hypercube is: In each corner of a cube, you place a microcomputer with its own memory. Then you turn each edge of the cube into a communication link that connects two processor nodes. This gives you a cube architecture in which each of the eight nodes can exchange messages with its three nearest neighbors only.

If you link each node in a cube with the corresponding node in another cube, you obtain a hypercube architecture with sixteen processors. And, if you link two of these hypercubes in the same manner, you get a hypercube with 32 nodes, and so on. The key insight is that whenever you double the number of processors, the increase in the number of communication links is only proportional to the previous number of processors. So, as microprocessors become cheaper, a hypercube scales nicely without letting the number of links grow out of bounds.

By October 1983 Chuck Seitz had constructed a 64-processor hypercube at Caltech. The message communication was handled by a slow software kernel, known somewhat grandiosely as the Crystaline Operating System, "although," as Geoffrey Fox pointed out, "it was never really an operating system." This parallel machine was no academic toy. From the beginning, Fox (1988) used the Cosmic Cube (as it was called) to solve substantial computational problems in science and engineering.

In 1985, before leaving Denmark, I was instrumental in obtaining funding for DIKU's first parallel computer, an Intel iPSC hypercube with 32 microcomputers. At the time, this machine was only the third of its kind acquired by European research institutions.

Although I recognized the invention of inexpensive supercomputers as a major breakthrough, I was never enamored of hypercube architectures. I felt that hypercube algorithms would be dominated by the problem of mapping problem-oriented process configurations onto a hypercube. That prediction turned out to be true, I think.

Parallel programs were often written in traditional programming languages, such as Fortran or C, extended with subroutines for parallelism. To my taste these programs were difficult to read and lacked the beauty that scientists expect of their own research. I was convinced that *the most important task in computational science was to make the programming of parallel computers easier.* This was, in my opinion, even more important than increasing computational power, and I felt that we should be prepared to sacrifice some performance to solve the programming problem.

At a Supercomputing Conference in Boston in May 1988, I looked (in vain) for the ideal parallel architecture of the future. Such a machine should, in my opinion, (1) use general-purpose microcomputers, (2) be expandable from tens to thousands of processors, (3) support different processor configurations (pipelines, trees, matrices, and so on) in a transparent manner, (4) handle process creation, communication, and termination by machine instructions that are only an order of magnitude slower than memory references, and (5) automatically balance the computational load among and route messages between the processors.

The first requirement ruled out NPAC's Connection Machine, in which 64,000 synchronous processors executed identical processes in lock step (Hillis 1985). The second one excluded multiprocessors. The third condition made hypercubes unsuitable. The only architecture that satisfied the first four requirements was a multicomputer known as the *Meiko Computing Surface.* No parallel computer satisfied the fifth condition.

In the summer of 1988, I traveled to Bristol, England, to visit Inmos and Meiko. At the Inmos research center, I met David May, the architect of the T800 transputer chip, a 32 bit VLSI microprocessor with 64 bit floating-point arithmetic. Four on-chip links enabled the transputer to exchange messages with four other transputers.

All programming of the transputer was done in the parallel programming language *occam*, which David May had based on Hoare's Communicating Sequential Processes (CSP). This language made it possible to define parallel processes that communicate by messages. Direct communication between two connected transputers was very fast (a few microseconds). Process creation and termination were also hardware operation. The transputer could switch from one occam process to another in 1 microsecond. There was no other processor like it in the world!

A group of Inmos employees had formed a small company, named Meiko, to build a multicomputer with transputer nodes. On my last day in Bristol, July 8, 1988, I had dinner with the chairman of Meiko, Miles Chesney. My appointment letter at SU specified that the university "will further purchase computer equipment as needed for your work in an amount not greater than $100,000." For that amount of money, Miles was prepared to sell me a Computing Surface with 20 transputers and 40 Mbytes of distributed memory. I told him that 20 transputers would not add anything new to my research in computational science, since I had already used a multiprocessor at NPAC with 18 processors. I would need at least 40 transputers to make multicomputer programming interesting. On the other hand, I had no problem with reducing the memory of each transputer to 1 Mbyte only. I also offered to make a Computing Surface at SU available to Meiko for demonstrations to potential American customers.

Being a risk taker, I asked Miles Chesney to leave a message at my hotel the next morning informing me if he was be willing to offer me a 40-node system for my money. If that was unacceptable, I would fly home empty-handed and look for another machine (although I could not think of any worthy alternative). When I woke up the next morning, there was a message from Miles accepting my request.

Back in Syracuse the university hosted an inaugural symposium in March 1989 to celebrate the opening of its new Center for Science and Technology. The themes of the symposium were parallel computers, neural networks, and intelligent systems. It was organized by Alan Robinson and me from Syracuse University together with Michael Arbib from the University of Southern California. On that occasion, the three of us were awarded the Chancellor's Medal for Outstanding Achievement. The nine invited speakers included Ralph Gomery from IBM, who reviewed the evolution of computing, David May from Inmos, who discussed the possibility of designing general-purpose parallel computers, and Geoffrey Fox from Caltech, who described major applications of parallel supercomputers. As the first speaker, I described

"The nature of parallel programming" without going into technical details (Brinch Hansen 1990).

In July, a Computing Surface was installed at SU right next to my office. It had 48 transputers, each with 1 Mbyte of memory. The transputers were linked by a switching network that could be reconfigured before program execution. After two months of initial problems with hardware, software, and documentation, I was able to run a trivial occam program that sorted 65,536 integers on 31 transputers. I was now ready to experiment with parallel scientific programs (Brinch Hansen 1995).

Although I knew nothing about numerical analysis, I thought that parallel solution of linear equations would be a useful programming exercise for a beginner. I chose the problem for the following reason: When a pipeline with p processors solves n linear equations, the parallel computer time for the numerical computation is of the order of n^3/p. A computer scientist would say that the numeral computation requires $O(n^3/p)$ time, while the input/output of the equations takes $O(n^2)$ time. If the problem size n is large compared to the machine size p, the relative overhead of processor communication is negligible. The high ratio of computation to communication makes the problem ideal for efficient parallel computing.

A colleague recommended Householder reduction as an attractive method for solving linear equations on a parallel computer. The main strength of the method is its unconditional numerical stability (Householder 1958). The more familiar Gaussian elimination is faster but requires a dynamic rearrangement of the equations, known as pivoting, which complicates a parallel program somewhat.

Unfortunately, I could not find a well-written, understandable explanation of Householder's method. Most textbooks on numerical analysis produced the so-called "Householder matrix" like a rabbit from a magician's top hat without explaining why it is defined the way it is. At that point, I stopped writing parallel programs and concentrated on sequential Householder reduction. After several frustrating weeks I was able to write a tutorial on Householder reduction. Two pages were sufficient to explain the purpose and derive the equation for Householder's matrix. I then explained the computational rules for Householder reduction and illustrated the method by a numerical example and a Pascal program.

I was beginning to think that others might have the same difficulty understanding this fundamental computation. So I submitted the tutorial to a journal that published it (Brinch Hansen 1992). One reviewer wrote that he "found the presentation far superior to the several descriptions I have seen in numerical analysis books." I quote this review not just because I like it, but because it was my first lesson about computational science: In order to understand a computation, I must first explain it to myself by writing a tutorial that includes a complete sequential program.

After studying parallel programming for 25 years it was not difficult for me to program a Householder pipeline in occam for the Computing Surface. To achieve approximate load-balancing, the pipeline was folded three times across an array of transputers, so that each transputer executed four pipeline processes. The folded pipeline solved 1000 equations on 45 transputers in 87 sec. The Computing Surface made the computation 32 times faster than it would have been on a single transputer. I was able to derive an elegant formula that predicted the parallel run time accurately as a function of the number of equations solved, the number of transputers used, and the number of times the pipeline was folded.

My next exercise was to compute the trajectories of n particles that interact by gravitation only. I considered the n-body problem to be particularly challenging on a parallel computer since it involves interactions among all the particles in each computational step. This means that every processor must communicate, directly or indirectly, with every other processor. My description of an n-body pipeline included a brief summary of Newton's laws of gravitation and a Pascal program for sequential n-body simulation.

It was a complete surprise for me to discover that the sequential Pascal programs for Householder reduction and n-body simulation had practically identical control structures. I suddenly understood that both of them are instances of the same *programming paradigm*: Each algorithm solves an *all-pairs problem*—a computation on every possible subset consisting of two elements chosen from a set of n elements. I did not find this insight mentioned in any textbook on numerical analysis or computational physics.

I now discarded both parallel algorithms and started all over. This time I programmed a general pipeline algorithm for all-pairs computations. This program was a parallel implementation of the common control structure. It provided a mechanism for performing the same operation on every pair of elements chosen from an array of n elements without specifying what the elements represent and how they "interact" pairwise.

I then turned the all-pairs pipeline into a Householder pipeline by using a few data types and procedures from the sequential Householder program. This transformation of the parallel program was completely mechanical and required no understanding of Householder's method. A similar transformation turned the all-pairs pipeline into an n-body pipeline.

On August 24, 1984, I made the following entry in the computer log book: "At midnight, I used 31 transputers to simulate 10,000 gravitational bodies in 47 sec/step!"

I had now found my research theme: I would explore the use of programming paradigms in parallel programming. In programming, the word "paradigm" is often used with a general (but vague) connotation, such as "the high level methodologies that we recognize as common to many of our effective algorithms." I used the term in a more narrow (but precise) sense: *A programming paradigm is a class of algorithms that solve different problems but have the same control structure.*

This was the beginning of my studies in computational science from the point of view of a computer scientist. I followed the advice of Geoffrey Fox to "use real hardware to solve real problems with real software." But, where the Caltech group concentrated on scientific applications for their own sake, I used them as realistic case studies to illustrate the use of structured programming in computational science.

In addition to all-pairs computations, I developed paradigms for tuple multiplication, divide-and-conquer, Monte Carlo trials and cellular automata. For each paradigm I wrote a general program that defined the common control structure. Such a program is sometimes called an algorithmic skeleton, a generic program, or a program template.

From a general parallel program I derived two or more *model programs* that illustrated the use of the paradigm to solve specific problems. A general program includes a few unspecified data types and procedures that vary from one application to another. A model program is obtained by replacing these data types and procedures with the corresponding data types and procedures from a sequential program that solves a specific problem. The essence of the programming methodology is that a model program has a parallel component that implements a paradigm and a sequential component for a specific application. The clear separation of the issues of parallelism and the details of application is essential for writing model programs that are easy to understand.

My own model programs solved typical problems in science and engineering: linear equations, n-body simulation, matrix multiplication, shortest paths in graphs, sorting, fast Fourier transforms, simulated annealing, primality testing, Laplace's equation, and forest fire simulation.

I ran these parallel programs on a Computing Surface configured as a pipeline, a tree, a cube, or a matrix of transputers.

⋆ ⋆ ⋆

I now turned my attention to the RSA cryptosystem,where large primes play an essential role in the encoding and decoding of messages (Rivest 1978). A user chooses two large random primes. These primes are used to compute a public encoding key and a secret decoding key. Both keys include the product of the primes. The user can receive encoded messages from anyone who knows the public key. But only the user (who knows the secret key) can decode the messages.

The crucial assumption is that it is feasible to generate large primes using a computer, but there is no known algorithm for finding the prime factors of large composite numbers in reasonable amounts of computer time. If that ever becomes possible, you will be able to break the code by factoring the public product of the secret primes.

At the time, the RSA cryptosystem was believed to be secure for keys of 150 decimal digits. The simplest way to find a 150-digit prime is to generate 150 random digits at a time, until you discover a prime. The probability that a 150-digit number is a prime is about 1 in 150 ln10. You must therefore expect to test about 350 numbers for primality before you find a prime. (Half of these tests can be skipped if you only examine odd numbers.)

So, the generation of primes is reduced to the problem of testing the primality of random numbers. Unfortunately, it is not feasible to determine whether or not a 150-digit integer is a prime by examining all the 10^{75} possible divisors (a truly astronomic number). The Miller–Rabin algorithm tests the same integer many times using different random numbers (Rabin 1980). If any one of the trials shows that a number is composite, then this is the correct answer. However, if all trials fail to prove that a number is composite, then it is almost certainly prime. The probability that the algorithm gives the wrong answer after, say, 40 trials is less than 10^{-24}.

This is far less than the probability of a computer error. A computer that performs one million operations per second, with the same probability of failure per operation, will fail once in thirty billion years. That is roughly the age of the universe since the Big Bang.

The advantage of using a multicomputer for primality testing is obvious. When the same random number has been broadcast to every processor, the trials can be performed simultaneously without any communication between the processors. Consequently, the processor efficiency is very close to 1 for non-trivial problems.

I programmed the Miller–Rabin algorithm in occam and used the Computing Surface to perform 40 tests of a 160-digit random number simultaneously on 40 transputers.

For the primality testing, I had to program multiple-length arithmetic. Most computers limit integer arithmetic to 32–64 bits, corresponding to 8–17 decimal digits. A larger integer must be represented by an array of digits, each occupying a single machine word. The arithmetic operations on multiple-length integers are serial operations that imitate paper-and-pencil operations.

I thought it would be easy to find a textbook that includes a simple algorithm for *multiple-length division* with a complete explanation. Much to my surprise, I was unable to find such a book. I ended up spending weeks on this "well-known" problem and finally wrote a tutorial that includes a complete Pascal algorithm (Brinch Hansen 1994). I mention this unexpected difficulty to illustrate what happens when a standard algorithm is not published as a well-structured program in an executable language.

Inspired by my use of a programming paradigm for primality testing, my student, Jonathan Greenfield, explored the development of distributed generic algorithms for RSA cryptography. He defined abstract algorithms in a variant of the parallel programming language Joyce. These algorithms were rewritten in the implementation language occam and tested on the Meiko Computing Surface. His PhD thesis was an appealing combination of the theory and practice of parallel computing. From the point of view of a computer scientist, it was an amazing feat to recognize five different aspects of the same application as instances of two simple paradigms for parallel computing. In addition, Jonathan's thesis was well-written and easy to understand. It was published as a volume in the Springer-Verlag Lecture Notes in Computer Science (Greenfield 1993)—a rare honor for a PhD student.

It had been fun to enter an interdisciplinary field, refresh my memory of mathematics and physics I learned as an undergraduate, study numerical analysis, and teach myself the art of multicomputer programming.

My one serious criticism of computational science was that it largely ignored the issue of precision and clarity in parallel programming that is essential for the education of future scientists. A written explanation is not an algorithm. A graph of computational steps is not an algorithm. A picture of a systolic array is not an algorithm. A mathematical formula is not an algorithm. A program outline written in non-executable "pseudocode" is not an algorithm. And, a complicated "code" that is difficult to understand will not do either.

Subtle algorithms must be presented in their entirety as well-structured programs written in readable, executable programming languages. This was my main reason for publishing model programs for computational science. I felt that the study of programming paradigms provides an architectural vision of parallel scientific computing.

My fifth book, *Studies in Computational Science: Parallel Programming Paradigms* was published in 1995. I wish I could say that this work influenced the way people program parallel computers, but—with the possible exception of my students—I don't think it did.

A graduate student, Anil Menon (1995), left this impression of my course on multicomputer programming:

> Over the last ten years, I've studied under many teachers and taken many courses. Dr. Brinch Hansen's course was unlike no other. He was interested in solving problems in parallel. I had no idea, even after five earlier courses, that it was so difficult. He took seven to eight different problems and showed by means of a series of beautiful and elegant programs, how one would go about writing parallel programs. His insights were often remarkable, for example, his deep idea that *process structures* were the correct way to reason and work with parallel processing, just as *data structures* are the key to sequential processing. Or the time he told us about the importance of constraints in the design process.
>
> Perhaps the conviction always evident in his presentation came from the fact that these programs were his own, and not copied off some standard book. Even now it mystifies me to some extent how he could reduce a really complex program to a series of subprograms each no more than a dozen lines, the whole piece elegantly connected.
>
> The course was especially enjoyable because Dr. Brinch Hansen is a character. He's passionate, outspoken, opinionated and intolerant of anything less than perfection. What a relief it was to find a professor who wasn't afraid to voice what he really felt about issues in computer science. None of that cowardly "on the one hand. . .on the other hand" balance with which the meek

> evade making choices. He was as opinionated about the state of NPAC, as he was about his language SuperPascal. I could go on and on: His rare sense of history, the remarkable perceptiveness with which he'd transform one problem into another etc. But perhaps the great physicist, Feynman put it best (though in a different context): "To do physics," he said, "you gotta have style". I believe it's true of computer science as well. Dr. Brinch Hansen does parallel programming in style, and for one great semester it was my privilege to learn by example.

⋆ ⋆ ⋆

I have always felt that professionals should study the history of their own field for the enjoyment and insight it gives. In 1978 and 1993, I attended two ACM conferences, which became milestones in the History of Programming Languages. The first conference (HOPL-I) covered the major languages of the 1960s (Wexelblat 1981). The program committee selected thirteen languages that had been in use for at least ten years, had significant influence, and were still in use. Each paper was presented by a pioneer who had played a key role in the development of the language. The following presentations were of special interest to me:

APL (Ken Iverson)
Algol (Alan Perlis and Peter Naur)
Basic (Tom Kurtz)
Cobol (Jean Sammet)
Fortran (John Backus)
LISP (John McCarthy)
PL/I (George Radin)
Simula (Ole-Johan Dahl and Kristen Nygaard)

On this occasion, Ole-Johan Dahl, Peter Naur, Alan Perlis and his wife Sydelle visited us in Altadena.

The second conference (HOPL-II) focussed on programming languages of the 1970s which "had significant influence on the theory or practice of computing" (Bergin 1996). This time the languages and speakers included:

Ada (Bill Whitaker)
C (Dennis Ritchie)
C++ (Bjarne Stroustrup)

CLU (Barbara Liskov)
Concurrent Pascal (Per Brinch Hansen)
Pascal (Niklaus Wirth)
Prolog (Alain Colmerauer and Philippe Roussel)
Smalltalk (Alan Kay)

The organizers set high technical and editorial standards. Historian Mike Mahony reviewed all the papers. Each author also worked with a technical expert who reviewed the various drafts. My own paper on "Monitors and Concurrent Pascal: A personal history" went through six drafts over a period of fifteen months.

In a "no holds barred" panel discussion, the following exchange took place (Bergin 1997):

> *Per Brinch Hansen:* I'm going to sit down, since you have already answered my question which is, "Is there a future for insecure, low-level languages like C, and huge, incomprehensible languages like Ada?" But, I wish to make a less loaded comment, which is that there may be differences of style between programming languages, but there ought to be some common idea of the minimal requirements, so we can all agree that we are looking at a programming language. And I think that's part of the problem.
>
> If you look at physics, for example, I would say that a theory ought to satisfy at least three requirements and so should a programming language. First, a notation, which is what a programming language is, is supposed to enable you to express a theory of computation, not necessarily a mathematical theory (although that would be ideal), but theories can also be helpful if they are informal, as in geology. In any programming language, you will recognize a set of abstractions that are machine-independent, but at a certain point those concepts break down. If you have overflowing arithmetic, your results become meaningless; and that goes for every one of them, that they only apply under certain conditions, which should be stated in the language manual. The requirement that a language should be secure is the simple requirement that a compiler and a computer should tell you, when the programming concepts break down. If we can't agree on that being a minimum requirement for all programming languages, then I think we are just using the same word for con-

venience to denote things that have very little in common. By that definition, C is not a programming language.

The second requirement is that a theory in physics must be simple. If a Niels Bohr can't comprehend it, or a [Richard] Feynman, then a committee of physicists won't be able to master it either. That boils down to the simple requirement that language manuals must be short, concise, and so must their compilers. By that definition, Ada is not a programming language. [laughter]

The third requirement was illustrated by the German physicist, [Wolfgang] Pauli, who once said to Bohr, "I have a crazy theory, you are going to like it!" To which Bohr responded: "It is not crazy enough!"

When I look at this conference, I do see a certain sameness in what we have done. Apart from these obvious violations of what programming languages should be, there is precious little difference between Fortran and Concurrent Pascal. They are mostly the same thing: x becomes $x + 1$. To me, it is not terribly interesting which languages will win, because that appears to be a study for sociologists, rather than computer scientists. What I like are the crazy paradigms we have seen, and there are two of those: Prolog and Smalltalk.

So I leave you with this question: Can we agree that a programming language must represent a theory of computation, that compilers and computers must check if the assumptions behind the abstractions apply when we run our programs, that the manuals must be short, and that the ideas must be crazy?

Dennis Ritchie: Could you please repeat the question? [laughter] I know what the question is. Is there a place for, in particular, C? Well, my guess is that there will not be any more significant low level languages—in other words, the niche is occupied—maybe that's just hoping. I guess the other response is that you have even stricter criteria than Jean [Sammet], whose criteria for considering what a language is, I think, are already too strict.

Niklaus Wirth: Is there any agreement among the four of you on the minimum requirements for us to call something a programming language other than the fact that it can change bits in a computer?

> *Dennis Ritchie:* Are you kidding? [laughter] No, of course, there is no agreement. That is the point. [laughter]
>
> *Bill Whitaker:* In particular, we didn't agree with YOU! [laughter]
>
> *Alan Kay:* I don't know, I think he hit it right on the head—I like the crazy part.

To understand what was going on here, I will quote what the biologist Francis Crick (1988) wrote about another "soft" science:

> [The work] tended to fall into a number of somewhat separate schools, each of which was rather reluctant to quote the work of the others. This is usually characteristic of a subject that is not producing any definite conclusions. (Philosophy and theology might be good examples.)

⋆ ⋆ ⋆

On May 8, 2002, I was awarded the IEEE Computer Pioneer medal "For pioneering development in operating systems and concurrent programming exemplified by work on the RC 4000 multiprogramming system, monitors, and Concurrent Pascal." In my acceptance speech (borrowing liberally from my own writing), I said (Brinch Hansen 2002):

> It is an unexpected pleasure for me to receive the first major award for the work I did from 1965 to 1975. I must confess, I was beginning to feel like Duke Ellington, who once said, "Fate doesn't want me to be famous too young." So, I thank the IEEE Computer Society for honoring me and making this speech necessary. And, I thank my friend, Jonathan Greenfield, for his tireless efforts in nominating me for the Computer Pioneer Award.
>
> Now, you should not for a minute imagine that I knew what I was doing as a young programmer. On two occasions, the work, you are honoring me for, almost came to nothing.
>
> In 1963, I graduated from the Technical University of Denmark without any programming experience (it was not yet being taught). There were (as far as I remember) no textbooks available on programming languages, compilers or operating systems.

With this background, I began my career as a systems programmer with Regnecentralen in Copenhagen. At age 29, I became head of software development for the *RC 4000 computer*. The senior manager of Regnecentralen, Niels Ivar Bech, gave me only one directive: "I need something new in multiprogramming!"

After a while, Jørn Jensen, Søren Lauesen, and I realized that we had no original ideas about multiprogramming. So, I told Bech: "We aren't getting anywhere. Is it all right with you if Jørn, Søren, and I spend a weekend at a country inn?" I wanted to give us one last chance. We had already agreed that we would either return with new ideas or give up and copy the best ideas we could find elsewhere. Bech immediately agreed (he had done the same thing when Regnecentralen's Cobol compiler project had come to a standstill).

It worked! The thought of returning to Regnecentralen without new ideas was simply unacceptable to us. Out of that weekend came the first ideas for the *RC 4000 multiprogramming system*, which introduced the now-standard concept of an operating system kernel.

Since 1970, I have been a computer scientist in the United States. While writing my textbook on operating system principles, I invented the *monitor notation*, which combines process synchronization with object-oriented programming.

At California Institute of Technology my goal was to develop a concurrent programming language with monitors. You would think it would be easy for me to extend Pascal with monitors. But I had no idea of how to do this. I remember sitting in my garden in Altadena, day after day, staring at a blank piece of paper and feeling like a complete failure. It took me two years to find reasonable solutions to most of the problems and make compromises which enabled me to ignore the most thorny issues.

In 1974, I distributed a description of the programming language *Concurrent Pascal*. I now understood what I was doing. One day the Caltech president, Harold Brown, came to my office and asked me to explain my research. After listening for half an hour, he said, "That sounds easy." I agreed because that was how I felt at the time. So, in the end, things turned out all right.

Let me conclude by quoting the biologist Francis Crick: "It's true that by blundering about we stumbled on gold, but the fact remains that we were looking for gold."

Thank you for your attention.

⋆ ⋆ ⋆

I am now sixty-six years old and close to retirement. My adult children left home many years ago after graduating from Syracuse University. My wife, Milena, received her second master's degree from SU and started a new career as a public librarian in Onondaga County.

I have been fortunate to live the creative life I dreamt of as a young man. It would have been easier for my family and colleagues, if I had been a more patient man, but you don't get to chose your temperament (or gifts for that matter).

I will end this programmer's story on a philosophical note by quoting Albert Einstein:

> In the light of knowledge attained, the happy achievement seems almost a matter of course, and any intelligent student can grasp it without too much trouble. But the years of anxious searching in the dark, with their intense longing, their alternations of confidence and exhaustion, and the final emergence into the light—only those who have experienced it can understand it.

Life has been good.