Model Programs for Computational Science: A Programming Methodology for Multicomputers[*]

(1993)

We describe a programming methodology for computational science based on programming paradigms for multicomputers. Each paradigm is a class of algorithms that have the same control structure. For every paradigm, a general parallel program is developed. The general program is then used to derive two or more model programs, which solve specific problems in science and engineering. These programs have been tested on a Computing Surface and published with every detail open to scrutiny. We explain the steps involved in developing model programs and conclude that the study of programming paradigms provides an architectural vision of parallel scientific computing.

1 Introduction

For the past three years I have studied *computational science* from the point of view of a computer scientist (Brinch Hansen 1990b–1992f). I have followed the advice of Geoffrey Fox (1990) 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 have used them as realistic case studies to illustrate the use of *structured programming* in computational science.

My research explores the role of *programming paradigms* in parallel computing. 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" (Nelson 1987). I will use

[*]P. Brinch Hansen, Model programs for computational science: A programming methodology for multicomputers. *Concurrency—Practice and Experience 5*, 5 (August 1993), 407–423. Copyright © 1993, John Wiley & Sons, Ltd.

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.

I have studied paradigms for all-pairs computations, tuple multiplication, divide-and-conquer, Monte Carlo trials and cellular automata (Brinch Hansen 1990c, 1991a, 1991d, 1992d, 1992f). For each paradigm I have written a general program that defines the common control structure. Such a program is sometimes called an *algorithmic skeleton*, a *generic program*, or a *program template* (Cole 1989; Brinch Hansen 1991b).

From a general parallel program I derive two or more *model programs* that illustrate 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 solve 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 have run these parallel programs on a *Computing Surface* configured as a pipeline, a tree, a cube or a matrix of *transputers*.

It has 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 is that it largely has 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 (Forsythe 1966; Ignizio 1973; Wirth 1976; Brinch Hansen 1977; Dunham 1982; Press 1989). This has been my main reason for publishing model programs for parallel scientific computing.

In the following, I will describe parallel programming paradigms and explain why I use different programming languages and computers for publication and implementation of model programs. I will also outline the steps involved in developing model programs based on paradigms. Finally, I will argue that the study of programming paradigms provides an architectural vision of parallel scientific computing.

2 The Computing Surface

When I started this research, I knew that my programs would soon become obsolete unless I wrote them for parallel architectures of the future. So I had to make an educated guess about the direction in which hardware and software technology would move parallel architectures during the 1990s.

By 1989 I had tentatively formulated the following requirements for a general-purpose parallel computer of the future (May 1988, 1990; Valiant 1989; Brinch Hansen 1990a):

1. A parallel architecture must be expandable from tens to thousands of processors.
2. A parallel computer must consist of general-purpose processors.
3. A parallel computer must support different process structures (pipelines, trees, matrices, and so on) in a transparent manner.
4. Process creation, communication, and termination must be hardware operations that are only an order of magnitude slower than memory references.
5. A parallel computer should automatically distribute the computational load and route messages between the processors.

The first three requirements eliminated multiprocessors, SIMD machines, and hypercubes, respectively. The only architecture that satisfied the first four requirements was the *Computing Surface* (Meiko 1987; McDonald 1991). No parallel computer satisfied the fifth condition.

In the summer of 1989 a Computing Surface was installed at Syracuse University. It is a *multicomputer* with 48 processors that can be extended to 1000 processors. Every processor is a T800 *transputer* with one or more megabytes of memory. The transputers are connected by a communication network that can be reconfigured before program execution. Direct communication is possible only among connected transputers, but is very fast (a few microseconds only). Process creation and termination are also hardware operations.

The programming tool is the parallel programming language *occam 2* (Inmos 1988; Cok 1991). This language makes it possible to define parallel processes that communicate by messages.

3 The All-Pairs Pipeline

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 in parallel, the numerical computation requires $O(n^3/p)$ time, while the input/output takes $O(n^2)$ time. If the problem size n is large compared to the machine size p, the 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 familiar Gaussian elimination is faster but requires a dynamic rearrangement of the equations, known as *pivoting*, which complicates a parallel program somewhat (Fox 1988).

Unfortunately, I could not find a well-written, understandable explanation of Householder's method. Most textbooks on numerical analysis produce Householder's matrix like a rabbit from a magician's top hat without explaining why it is defined the way it is. At this 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 (Brinch Hansen 1990b). Two pages were sufficient to explain the purpose and derive the equation of 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. One of the reviewers 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 too difficult for me to program a *Householder pipeline* (Brinch Hansen 1990c). The parallel program was written in occam for the Computing Surface. I used a coarse-grain pipeline to reduce communication overhead. To achieve approximate *load-balancing*, the pipeline was folded three times across an array of p transputers (Brinch Hansen 1990d). Figure 1 shows the *folded pipeline.* The squares and lines represent pipeline nodes and communication channels, respectively. Each column represents a single transputer that executes four parallel nodes.

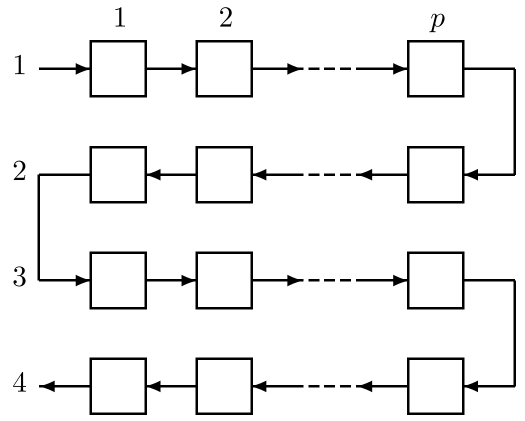

Figure 1 A folded pipeline.

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 (Brinch Hansen 1991a). Others have solved the same problem using a ring of processors (Ellingworth 1988; Fox 1988).

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 have not found 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 (Brinch Hansen 1990c). This program is a parallel implementation of the common control structure. It provides 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.

Later I discovered that all-pairs pipelines were described informally by Shih (1987), and Cosnard (1988), but without concise algorithms.

I had now found my *research theme*: the use of parallel programming paradigms in computational science.

4 The Multiplication Pipeline

After programming a subtle parallel program, I looked for the simplest problem that would illustrate the benefits of developing generic algorithms for parallel programming paradigms that can be adapted to different applications.

This time I chose *matrix multiplication*, which can be pipelined in a straightforward way as shown in Fig. 2 (Kung 1989).

First, the rows of a matrix a are distributed evenly among the nodes of the pipeline. Then the columns of a matrix b pass through the pipeline

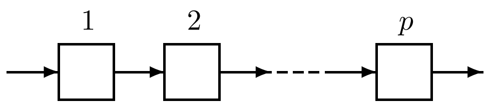

Figure 2 A simple pipeline.

while each node computes a portion of the matrix product $a \times b$. Finally, the pipeline outputs the product matrix.

For sequential algorithms it is well-known that matrix multiplication is similar to the problem of finding the *shortest paths* between every pair of nodes in a directed graph with n nodes (Cormen 1990).

After studying both algorithms, the unifying concept seemed to me to be an operation that I called *tuple multiplication*: the product of two n-tuples a and b is an $n \times n$ matrix c. The matrix elements are obtained by applying the same function f to every ordered pair consisting of an element of a and an element of b, that is $c_{ij} = f(a_i, b_j)$.

In the case of matrix multiplication, the tuple elements are rows and columns, respectively, and every function value is the dot product of a row and a column. The input to the shortest paths problem is the adjacency matrix of a graph. The output is a distance matrix computed by a sequence of tuple multiplications. In every multiplication, tuple a consists of the n rows of the adjacency matrix, while tuple b consists of the n columns of the distance matrix. The function value $f(a_i, b_j)$ defines the shortest path length found so far between nodes i and j of the graph.

The task was now obvious. I wrote a paper that defined a pipeline algorithm for tuple multiplication. I briefly explained matrix multiplication and the all-pairs shortest-path problem by means of Pascal algorithms. I then transformed the parallel program into pipeline algorithms for the two applications by defining the data types of the tuples and the corresponding variants of the function f. After rewriting the parallel programs in occam, I analyzed and measured their performance on the Computing Surface (Brinch Hansen 1991b).

5 The Divide and Conquer Tree

My third paradigm was a parallel *divide-and-conquer* algorithm for a binary tree of processor nodes (Browning 1980; Brinch Hansen 1991d). Figure 3 shows the *tree machine*.

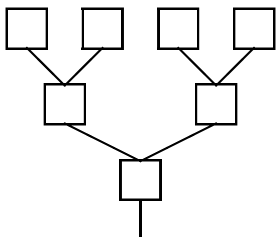

Figure 3 A tree machine.

The root node of the tree inputs a complete problem, splits it into two parts, and sends one part to its left child node and the other part to its right child node. The splitting process is repeated at higher levels in the tree. Eventually, every leaf node inputs a problem from its parent node, solves it, and outputs the solution to its parent. Every parent node in the tree inputs two partial solutions from its children and combines them into a single solution, which is output to its parent. Finally, the root node outputs the solution to the complete problem.

A problem and its solution are both defined by an array of n elements of the same type. The element type and the procedures for splitting problems and combining solutions are the only parts of the parallel algorithm that depend on the nature of a specific problem. Consequently, it was easy to transform the general algorithm into parallel versions of *quicksort* (Hoare 1961) and the *fast Fourier transform* (Cooley 1965; Brigham 1974; Brinch Hansen 1991c).

The emphasis on the common paradigm enabled me to discover an unexpected similarity between these well-known algorithms. After programming an iterative version of the fast Fourier transform, I suddenly realized that it must also be possible to write a *quicksort without a stack!* In standard quicksort, the *partition* procedure divides an array into two slices of unpredictable sizes. Why not replace this algorithm with the *find* procedure (Hoare 1971) and use it to split an array into two halves? Then you don't need a stack to remember where you split the array.

On the average, *find* is twice as slow as *partition*. That is probably the reason why a balanced quicksort is seldom used for sequential computers. However, on a multicomputer, the unpredictable nature of standard quicksort causes severe *load imbalance* (Fox 1988). If the two halves of a tree machine sort sequences of very different lengths, half of the processors are doing most of the work, while the other half are idle most of the time. As a compromise, I used *find* in the parent nodes and *partition* in the leaf nodes. Measurements show that the *balanced parallel quicksort* consistently runs faster than the unbalanced algorithm.

I selected the divide-and-conquer paradigm to demonstrate that some parallel computations are inherently *inefficient*. The average sorting time of n elements is $O(n\log n)$ on a sequential computer. A tree machine cannot reduce the sorting time below the $O(n)$ time required to input and output the n elements through the root node. So, for problems of size n, the *parallel speed-up* cannot exceed $O(\log n)$. No matter how many processors you use to sort, say, a million numbers, they can do it only an order of magnitude faster than a single processor. This modest speed-up makes divide-and-conquer algorithms unsatisfactory for multicomputers with hundreds or thousands of processors.

6 The Divide and Conquer Cube

I have never been enamored of *hypercube* architectures (Seitz 1985). I felt that hypercube algorithms would be dominated by the problem of mapping problem-oriented process configurations onto a hypercube. This prediction turned out to be true, I think (Fox 1988). Hypercubes can probably be made reasonably easy to use if they are supported by a library of programming paradigms that hide the mapping problem from scientific users. But I pity the professional programmers, who will have to cope with the awkward details of paradigm implementation.

In the future, most parallel architectures will almost certainly support automatic routing of messages between any pair of nodes. Although the hardware architecture may be a hypercube, this structure will be transparent to the programmer, who will define abstract configurations of nodes connected by virtual channels (May 1988; Valiant 1989). In the meantime, reconfigurable multicomputers are a reasonable compromise.

On a general-purpose multicomputer, a programmer may, of course, choose the hypercube as a programming paradigm in its own right. So, I was curious to find out if a hypercube sorts faster than a tree machine. Figure 4 shows a *cube* with eight processor nodes.

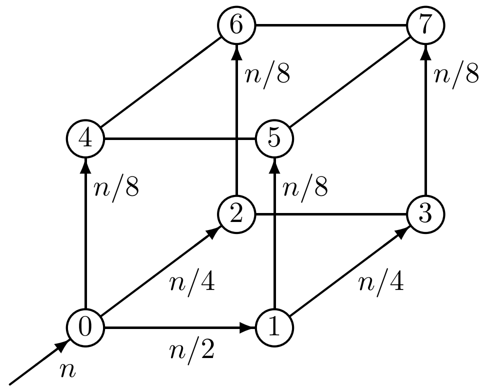

Figure 4 Data distribution in a cube.

First, node 0 inputs n numbers, splits them into two halves, sends one half to node 1, and keeps the other half. Then nodes 0 and 1 each split their halves into quarters. Finally, nodes 0, 1, 2, and 3 each keep an eighth of the numbers and sends the other eighths to nodes 4, 5, 6, and 7. All the nodes now work in parallel while each of them sorts one eighth of the numbers. Afterwards, nodes 0, 1, 2, and 3 each input a sorted sequence of size $n/8$ from their "children" and combine them with their own numbers to form sorted sequences of size $n/4$. Nodes 0 and 1 repeat the combination process and form sorted sequences of size $n/2$. At the end, node 0 outputs n sorted numbers to its environment.

A larger hypercube follows the same general pattern of splitting a sorting problem into smaller problems, solving them in parallel and combining the results. Needless to say, the sorting algorithm can easily be replaced by a fast Fourier transform.

On a hypercube, every node sorts a portion of the numbers. However, on a tree machine, sorting is done by the leaf nodes only. In spite of this, I found that a hypercube with 32 or more nodes sorts only marginally faster than a tree machine of the same size. This conclusion was based on a performance model verified by experiments (Brinch Hansen 1991e). The reason is simple. On a large tree machine, the sorting time of the leaf nodes is small compared to the data distribution time of the remaining nodes. So there is is not much gained by reducing the sorting time further.

7 Parallel Monte Carlo Trials

Monte Carlo methods are algorithms that use random number generators to simulate stochastic processes. Probabilistic algorithms have been applied successfully to combinatorial problems, which cannot be solved exactly because they have a vast number of possible solutions.

The most famous example is the problem of the *traveling salesperson* who must visit n cities (Lawler 1985). No computer will ever be able to find the shortest possible tour through 100 cities by examining all the 5×10^{150} possible tours. For practical purposes, the problem can be effectively solved by *simulated annealing* (Kirkpatrick 1983; Aarts 1989). This Monte Carlo method has a high probability of finding a near-optimal tour of 100 cities after examining a random sample of one million tours.

Fox *et al.* (1988) have shown that a hypercube can solve the traveling salesperson problem by simulated annealing. Allwright and Carpenter (1989) have solved the same problem on an array of transputers. These algorithms use parallelism to speed up the annealing process.

After a while I noticed that papers on simulated annealing often included remarks such as the following: "Our results [are] averaged over 20 initial random tours" (Moscato 1989). When you think about it, it makes sense: Due to the probabilistic nature of Monte Carlo methods, the best results are obtained by performing the same computation many times with different random numbers.

The advantage of using a multicomputer for *parallel Monte Carlo trials* is obvious. When the same problem 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.

A straightforward implementation of the Monte Carlo paradigm requires a *master* processor that communicates directly with p *servers*. Each processor performs m/p trials. The master then collects the m solutions from the servers. Unfortunately, most multicomputers permit each processor to communicate with only a few neighboring processors. For p larger than, say, 4, the data must be transmitted through a chain of processors. The simplest way to do this is to use a *pipeline* with p processors controlled by a master processor (Brinch Hansen 1992d).

I used this paradigm to compute ten different tours of 2500 cities simultaneously and select the shortest tour obtained (Brinch Hansen 1992a).

My second application of the paradigm was *primality testing* of a large integer, which is of considerable interest in *cryptography* (Rivest 1978). It is not feasible to determine whether or not a 150-digit integer is a prime by examining all the 10^{75} possible divisors. 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}.

I programmed the Miller–Rabin algorithm in occam and used the Monte Carlo paradigm to perform 40 tests of a 160-digit random number simultaneously on 40 transputers (Brinch Hansen 1992b).

For primality testing, I had to program multiple-length arithmetic. These serial operations imitate the familiar paper-and-pencil methods. 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 1992c). 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.

8 Parallel Cellular Automata

A *cellular automaton* is a discrete model of a system that varies in time and space. The discrete space is an array of identical cells, each representing a local state. As time advances in discrete steps, the system evolves according to universal laws. Every time the clock ticks, the cells update their states simultaneously. The next state of a cell depends only on the current states of the cell and its nearest neighbors.

John von Neumann (1966) and Stan Ulam (1986) introduced cellular automata to study self-reproducing systems. John Conway's game, *Life*, is undoubtedly the most widely known cellular automaton (Gardner 1970). Another well-known automaton simulates the life cycles of sharks and fish on the imaginary planet *Wa-Tor* (Dewdney 1984).

Cellular automata can simulate continuous physical systems described by *partial differential equations.* The numerical solution of, say, Laplace's equation by grid iteration is really a discrete simulation of heat flow performed by a cellular automaton.

Fox *et al.* (1988) described a Wa-Tor simulator for a hypercube. Numerical solution of Laplace's equation on multicomputers has been discussed by Barlow and Evans (1982), Evans (1984), Pritchard *et al.* (1987), Saltz *et al.* (1987), and Fox *et al.* (1988).

I developed and published a model program for parallel execution of a cellular automaton on a multicomputer configured as a *matrix* of processors (Fig. 5).

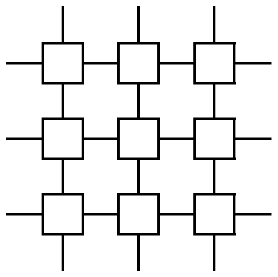

Figure 5 Processor matrix.

The combined state of a cellular automaton is represented by an $n \times n$ *grid.* The $q \times q$ processors hold *subgrids* of size $n/q \times n/q$. In each step, every node exchanges boundary values with its four nearest neighbors (if any). The nodes then update their subgrids simultaneously. At the end of a simulation, the nodes output their final values to a master processor that assembles a complete grid.

The shared boundary values raise the familiar concern about time-dependent errors in parallel programs. Race conditions are prevented by assigning a binary *parity* to every grid element. *Even* and *odd* elements are updated in two separate phases (Barlow 1982). Deadlock is prevented by letting every node communicate simultaneously with its four neighbors (Brinch Hansen 1992f).

The only parts of the parallel program that vary from one application to another are the possible *states* of the cells and the state *transition function.*

I have used a 6×6 matrix of transputers to simulate a *forest fire* on a

480×480 grid. Every element represents a tree that is either alive, burning, or dead. In each step, the next state of every tree is defined by probabilistic rules (Bak 1990).

I used the same paradigm to solve *Laplace's equation* for equilibrium temperatures in a square region with fixed temperatures at the boundaries. Every element represents the temperature at a single point in the region. In each step, the temperature of every interior element is replaced by a weighted sum of the previous temperature and the average of the surrounding temperatures (Brinch Hansen 1992e). In numerical analysis, this method is known as *successive overrelaxation* with *parity ordering* (Young 1971; Press 1989). The parallel program used 6×6 transputers to solve the heat equation on a 1500×1500 grid.

9 Program Characteristics

After studying the paradigms separately, it is instructive to consider what they have in common.

I was surprised by the *specialized* nature of some of the paradigms (Table 1). It may well be that some of them apply to only a small number of problems. To me that is a minor concern. The essence of the programming method is that you attempt to write two or more programs simultaneously. The intellectual discipline required to do this seems almost inevitably to produce well structured programs that are easy to understand.

The model programs illustrate programming methods for a variety of multicomputer *architectures* (Table 2). The reconfigurable Computing Surface was ideal for this purpose.

If a parallel architecture is not reconfigurable, it may be necessary to reprogram some of the paradigms. However, since all instances of a paradigm have the same sequential control structure, you know that if you can implement any one of them on a parallel architecture, the rest will turn out to be variations of the same theme.

Every program has a parallel component that implements a *paradigm* and a sequential component for a specific *application* (Table 3). The paradigm typically accounts for 60% of a program and is the most difficult part to write.

To make the programs *readable*, I divided them into short procedures of 10–20 lines each. No procedure exceeds one page of text (Table 4).

I have always found that a good *description* of a program is considerably

Table 1 Paradigms.

Program	Paradigm
Annealing	Monte Carlo trials
Primality	Monte Carlo trials
Multiply	Multiplication
Paths	Multiplication
Householder	All-pairs
N-body	All-pairs
FFT tree	Divide-and-conquer
Sorting tree	Divide-and-conquer
Sorting cube	Divide-and-conquer
Laplace	Cellular automata
Forest fire	Cellular automata

Table 2 Architectures.

Program	Architecture
Annealing	Pipeline
Primality	Pipeline
Multiply	Pipeline
Paths	Pipeline
Householder	Pipeline
N-body	Pipeline
FFT tree	Tree
Sorting tree	Tree
Sorting cube	Cube
Laplace	Matrix
Forest fire	Matrix

longer than the program text (Table 5). Fifteen years ago, I put it this way: "Programming is the art of writing essays in crystal clear prose and making them executable" (Brinch Hansen 1977).

Table 6 illustrates the *performance* of the model programs on a Computing Surface in terms of the size of the problems solved and the speedup S_p achieved by p processors running in parallel.

Table 3 Program lengths.

Program	Paradigm (lines)	Application (lines)
Annealing	150	200
Primality	150	520
Multiply	150	30
Paths	150	90
Householder	190	120
N-body	190	130
FFT tree	140	100
Sorting tree	140	80
Sorting cube	170	80
Laplace	280	30
Forest fire	280	60

Table 4 Procedure lengths.

Program	Lines/procedure		
	Min	Aver	Max
Annealing	1	12	34
Primality	3	15	43
Multiply	6	12	28
Paths	6	14	28
Householder	8	18	50
N-body	2	12	26
FFT tree	6	14	24
Sorting tree	6	13	24
Sorting cube	6	13	29
Laplace	3	12	31
Forest fire	3	13	31

Table 5 Program descriptions.

Program	Program (pages)	Report (pages)
Annealing	8	20
Multiply	5	10
Householder	7	40
FFT tree	6	30
Laplace	7	40

Table 6 Program performance.

Program	Problem size	p	S_p
Annealing	400	10	10
Primality	160	40	40
Multiply	1400×1400	35	31
Householder	1250×1250	25	20
N-body	9000	45	36
FFT tree	32768	31	4
Sorting tree	131072	31	3
Sorting cube	131072	8	2
Laplace	1500×1500	36	34

10 Programming Languages

As I was describing my first parallel paradigm, I became disenchanted with occam as a *publication language.* To my taste, occam looks clumsy compared to Pascal. (I hasten to add that I prefer occam to its competitors, Fortran, C, and Ada.)

At the time, no programming language was suitable for writing elegant, portable programs for multicomputers. As a compromise, I used *Pascal* extended with *parallel statements* and *communication channels* as a publication language.

To avoid dealing with the obscure behavior of incorrect programs on a multicomputer, I tested the parallel programs on a sequential computer. Since my publication language was not executable, I rewrote the model programs in an executable Pascal dialect that includes parallel statements and conditional critical regions. I used conditional critical regions to implement message passing and tested the programs on an IBM-PC with 64 Kbytes of memory.

When the parallel programs worked, I rewrote them in occam, changed a few constants, and used them to solve much larger problems on a Computing Surface with 48 transputers and 48 Mbytes of distributed memory. Sometimes I used *Joyce* to run the same computation on an Encore Multimax, a multiprocessor with 16 processors and 128 Mbytes of shared memory (Brinch Hansen 1987, 1989). The manual translation of correct readable programs into occam or Joyce was a trivial task.

The ease with which I could express the model programs in three different programming languages and run them on three different computer architectures prove that they are eminently portable.

The development of an executable publication language was a long-term goal of my research. One of my reasons for writing the model programs was to identify language features that are indispensable and some that are unnecessary for parallel scientific computing.

The published paradigm for the tree machine includes a recursive procedure that defines a tree of processes as a root process running in parallel with two subtrees. A notation for *recursive processes* is essential for expressing this idea concisely (Brinch Hansen 1990a). After using Joyce, I found the lack of recursion in occam unacceptable.

So far I have not found it necessary to use a statement that enables a process to poll several channels until a communication takes place on one of them. I have tentatively adopted the position that *non-deterministic* *communication* is necessary at the hardware level in a routing network, but is probably superfluous for scientific programming. It would be encouraging if this turns out to be true, since polling can be inefficient (Brinch Hansen 1989).

In practice, programmers will often be obligated to implement programs in complicated languages. However, wise programmers will prefer to develop and publish their ideas in the simplest possible languages, even if they are expected to use archaic or abstruse *implementation languages* for their final software products. Since it is no problem to rewrite a model program in another language, it is not particularly important to be able to use publication and implementation languages on the same machine.

Nevertheless, I must confess that the relentless efforts to adapt the world's oldest programming language for parallel computing strike me as futile. A quarter of a century ago, Alan Perlis boldly selected *Algol 60* as the publication language for algorithms in *Communications of the ACM*. In response to his critics, he said: "It is argued that more programmers now know Fortran than Algol. While this is true, it is not necessarily relevant since this does not increase the readability of algorithms in Fortran" (Perlis 1966).

Present multicomputers are difficult to program, because every program must be tailored to a particular architecture. It makes no sense to me to complicate hard intellectual work by poor notation. Nor am I swayed by the huge investment in existing Fortran programs. Every generation of scientists must reprogram these programs if they wish to understand them in depth and verify that they are correct. And the discovery of new architectures will continue to require reprogramming in unfamiliar notations that have not been invented yet.

11 Research Method

It took me a year to study numerical analysis, learn multicomputer programming, select a research theme, understand the development steps involved and complete the first paradigm. From then on, every paradigm took about one semester of research.

I followed the same steps for every paradigm:

1. Identify two computational problems with the same sequential control structure.

2. For each problem, write a tutorial that explains the theory of the computation and includes a complete Pascal program.
3. Write a parallel program for the programming paradigm in a readable publication language.
4. Test the parallel program on a sequential computer.
5. Derive a parallel program for each problem by trivial substitutions of a few data types, variables and procedures, and analyze the complexity of these programs.
6. Rewrite the parallel programs in an implementation language and measure their performance on a multicomputer.
7. Write clear descriptions of the parallel programs.
8. Rewrite the programs using the same terminology as in the descriptions.
9. Publish the programs and descriptions in their entirety with no hidden mysteries and every program line open to scrutiny.

The most difficult step is the *discovery* of paradigms and the *selection* of interesting instances of these paradigms. This creative process cannot be reduced to a simplistic recipe. Now that I know what I am looking for, I find it helpful to browse through books and journals on the computational aspects of biology, engineering, geology, mathematics and physics. When I see an interesting problem I ask myself: "Is there any way this computation can be regarded as similar to another one?" *Luck* clearly plays a role in the search for paradigms. However, as the French philosopher Bernard de Fontenelle (1657–1757) once observed: "These strokes of fortune are only for those who play well!" So I keep on trying.

12 Conclusions

I have described a collection of model programs for computational science. Every program is a realistic case study that illustrates the use of a paradigm for parallel programming. A programming paradigm is a class of algorithms that solve different problems but have the same control structure. The individual algorithms may be regarded as refinements of a general algorithm that defines the common control structure.

Parallel programming paradigms are elegant solutions to non-trivial problems:

1. Paradigms are *beautiful* programs that challenge your intellectual abilities and programming skills.

2. A programming paradigm is a *unifying* concept that reveals unexpected similarities between algorithms and raises new questions about familiar algorithms.

3. Viewing a parallel algorithm as an instance of a paradigm enables you to *separate* the issues of parallelism from the details of application. This sharp distinction contributes to program clarity.

4. Every paradigm defines an effective *programming style* that becomes part of your mental tool kit and enables you to apply previous insight to new problems (Nelson 1987).

5. Paradigms serve as case studies that illustrate the use of *structured programming* in scientific computing (Dijkstra 1972).

6. A commitment to *publish* paradigms as complete, executable programs imposes an intellectual discipline that leaves little room for vague statements and missing details. Such programs may serve as *models* for other scientists who wish to study them with the assurance that every detail has been considered, explained, and tested.

7. Model programs may also teach students the neglected art of program *reading* (Wirth 1976; Mills 1988).

8. Parallel paradigms capture the essence of *parallel architectures* such as pipelines, trees, hypercubes and matrices.

9. Parallel programs based on the same paradigm can be *moved* to different architectures with a reasonable effort by rewriting the general program that defines the common control structure. The individual programs can then be moved by making minor changes to the paradigm (Dongarra 1989).

10. Since a paradigm defines a whole class of useful algorithms, it is an excellent choice as a *benchmark* for parallel architectures.

11. A collection of paradigms can provide valuable guidance for *programming language design* (Floyd 1987). If the paradigms are rewritten in a proposed notation, the readability of the programs will reveal whether or not the language concepts are essential and concise.

After using this programming methodology for three years, the evidence strikes me as overwhelming: *The study of programming paradigms provides an architectural vision of parallel scientific computing!*

Acknowledgements

I am grateful to Coen Bron, Mani Chandi, Peter Denning, Jack Dongarra, Paul Dubois, Geoffrey Fox, Jonathan Greenfield, Tony Hey, Harlan Mills, Peter O'Hearn, Les Valiant and Virgil Wallentine for their constructive comments on earlier drafts of this paper.

References

Aarts, E. and Korst, J. 1989. *Simulated Annealing and Boltzmann Machines.* Wiley, New York.

Allwright, J.R.A. and Carpenter, D.B. 1989. A distributed implementation of simulated annealing. *Parallel Computing 10*, 335–338.

Bak, P. and Chen, K. 1990. A forest-fire model and some thoughts on turbulence. *Physics Letters A 147*, 5–6, 297–299.

Barlow, R.H., and Evans, D.J. 1982. Parallel algorithms for the iterative solution to linear systems. *Computer Journal 25*, 1, 56–60.

Brigham, E.O. 1974. *The Fast Fourier Transform.* Prentice Hall, Englewood Cliffs, NJ.

Brinch Hansen, P. 1977. *The Architecture of Concurrent Programs.* Prentice Hall, Englewood Cliffs, NJ.

Brinch Hansen, P. 1987. Joyce—A programming language for distributed systems. *Software—Practice and Experience 17*, 29–50. *Article 18.*

Brinch Hansen, P. 1989. A multiprocessor implementation of Joyce. *Software—Practice and Experience 19*, 579–592. *Article 19.*

Brinch Hansen, P. 1990a. The nature of parallel programming. In *Natural and Artificial Parallel Computation*, M.A. Arbib and J.A. Robinson, Eds. The MIT Press, Cambridge, MA, (1990), 31–46. *Article 20.*

Brinch Hansen, P. 1990b. Householder reduction of linear equations. School of Computer and Information Science, Syracuse University, Syracuse, NY. Revised version in *ACM Computing Surveys 24*, (June 1992), 185–194.

Brinch Hansen, P. 1990c. The all-pairs pipeline. School of Computer and Information Science, Syracuse University, Syracuse, NY.

Brinch Hansen, P. 1990d. Balancing a pipeline by folding. School of Computer and Information Science, Syracuse University, Syracuse, NY.

Brinch Hansen, P. 1991a. The n-body pipeline. School of Computer and Information Science, Syracuse University, Syracuse, NY.

Brinch Hansen, P. 1991b. A generic multiplication pipeline. School of Computer and Information Science, Syracuse University, Syracuse, NY.

Brinch Hansen, P. 1991c. The fast Fourier transform. School of Computer and Information Science, Syracuse University, Syracuse, NY.

Brinch Hansen, P. 1991d. Parallel divide and conquer. School of Computer and Information Science, Syracuse University, Syracuse, NY.

Brinch Hansen, P. 1991e. Do hypercubes sort faster than tree machines? School of Computer and Information Science, Syracuse University, Syracuse, NY.

Brinch Hansen, P. 1992a. Simulated annealing. School of Computer and Information Science, Syracuse University, Syracuse, NY.

Brinch Hansen, P. 1992b. Primality testing. School of Computer and Information Science, Syracuse University, Syracuse, NY.

Brinch Hansen, P. 1992c. Multiple-length division revisited: A tour of the minefield. School of Computer and Information Science, Syracuse University, Syracuse, NY.

Brinch Hansen, P. 1992d. Parallel Monte Carlo trials. School of Computer and Information Science, Syracuse University, Syracuse, NY.

Brinch Hansen, P. 1992e. Numerical solution of Laplace's equation. School of Computer and Information Science, Syracuse University, Syracuse, NY.

Brinch Hansen, P. 1992f. Parallel cellular automata: A model program for computational science. School of Computer and Information Science, Syracuse University, Syracuse, NY.

Browning, S.A. 1980. Algorithms for the tree machine. In *Introduction to VLSI Systems*, C. Mead and L. Conway, Eds. Addison-Wesley, Reading, MA, 295–313.

Cok, R.S. 1991. *Parallel Programs for the Transputer.* Prentice Hall, Englewood Cliffs, NJ.

Cole, M.I. 1989. *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press, Cambridge, MA.

Cooley, J.W. and Tukey, J.W. 1965. An algorithm for machine calculation of complex Fourier series. *Mathematics of Computation 19*, 297–301.

Cormen, T.H., Leiserson, C.E. and Rivest, R.L. 1990. *Introduction to Algorithms*, The MIT Press, Cambridge, MA.

Cosnard, M. and Tchuente, M. 1988. Designing systolic algorithms by top-down analysis. *The Third International Conference on Supercomputing*, Vol. 3, International Supercomputing Institute, St. Petersburg, FL, 9–18.

Dewdney, A.K. 1984. Sharks and fish wage an ecological war on the toroidal planet Wa-Tor. *Scientific American 251*, 6, 14–22.

Dijkstra, E.W. 1972. Notes on structured programming. In *Structured Programming*, O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, Eds. Academic Press, New York.

Dongarra, J.J. and Sorenson, D.C. 1989. Algorithmic design for different computer architectures. In *Opportunities and Constraints of Parallel Computing*, J.L.C. Sanz, Ed. Springer-Verlag, New York, 33–35.

Dunham, C.B. 1982. The necessity of publishing programs. *Computer Journal 25*, 61–62.

Ellingworth, H.R.P. 1988. Transputers and computational chemistry: an application. *The Third International Conference on Supercomputing*, Vol. 1, International Supercomputing Institute, St. Petersburg, FL, 269–274.

Evans, D.J. 1984. Parallel SOR iterative methods. *Parallel Computing 1*, 3–18.

Floyd, R.W. 1987. The paradigms of programming. In *ACM Turing Award Lectures: The First Twenty Years, 1966–1985*, R.L. Ashenhurst and S. Graham, Eds. ACM Press, New York, 131–142.

Forsythe, G.E. 1966. Algorithms for scientific computing. *Communications of the ACM 9*, 255–256.

Fox, G.C., Johnson, M.A., Lyzenga, G.A., Otto, S.W., Salmon, J.K. and Walker, D.W. 1988. *Solving Problems on Concurrent Processors*, Vol. I, Prentice-Hall, Englewood Cliffs, NJ.

Fox, G.C. 1990. Applications of parallel supercomputers: scientific results and computer science lessons. In *Natural and Artificial Parallel Computation*, M.A. Arbib and J.A. Robinson, Eds. The MIT Press, Cambridge, MA, 47–90.

Gardner, M. 1970. The fantastic combinations of John Conway's new solitaire game "Life." *Scientific American 223*, 10, 120–123.

Hoare, C.A.R. 1961. Algorithm 64: Quicksort. *Communications of the ACM 4*, 321.

Hoare, C.A.R. 1971. Proof of a program: Find. *Communications of the ACM 14*, 39–45.

Householder, A.S. 1958. Unitary triangularization of a nonsymmetric matrix. *Journal of the ACM 5*, 339–342.

Ignizio, J.P. 1973. Validating claims for algorithms proposed for publication. *Operations Research 21*, 852-854.

Inmos Ltd. 1988. *occam 2 Reference Manual*. Prentice Hall, Englewood Cliffs, NJ.

Kirkpatrick, S., Gelatt, C.D. and Vechi, M.P. 1983. Optimization by simulated annealing. *Science 220*, 671–680.

Kung, H.T. 1989. Computational models for parallel computers. In *Scientific Applications of Multiprocessors*, R. Elliott and C.A.R. Hoare, Eds. Prentice Hall, Englewood Cliffs, NJ, 1–15.

Lawler, E.L., Lenstra, J.K., Rinnooy Kan, A.H.G. and Shmoys, D.B., Eds. 1985. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. Wiley, Chichester, England.

McDonald, N. 1991. Meiko Scientific Ltd. In *Past, Present, Parallel: A Survey of Available Parallel Computing Systems*, A. Trew and G. Wilson, Eds. Springer-Verlag, New York, 165–175.

May, D. 1988. The influence of VLSI technology on computer architecture. *The Third International Conference on Supercomputing*, Vol. 2, International Supercomputing Institute, St. Petersburg, FL, 247–256.

May, D. 1990. Towards general-purpose parallel computers. In *Natural and Artificial Parallel Computation*, M.A. Arbib and J.A. Robinson, Eds. The MIT Press, Cambridge, MA, 91–121.

Meiko Ltd. 1987. *Computing Surface Technical Specifications*. Meiko Ltd., Bristol, England.

Mills, H.D. 1988. *Software Productivity*. Dorset House, New York, NY.

Moscato, P. and Fontanari, J.F. (1989) Stochastic vs. deterministic update in simulated annealing. California Institute of Technology, Pasadena, CA.

Nelson, P.A. and Snyder, L. 1987. Programming paradigms for nonshared memory parallel computers. In *The Characteristics of Parallel Algorithms*, L.H. Jamieson, D. Gannon, and R.J. Douglas, Eds. The MIT Press, Cambridge, MA, 3–20.

Perlis, A.J. 1966. A new policy for algorithms? *Communications of the ACM 9*, 255.

Press, W.H., Flannery, B.P., Teukolsky, S.A. and Vetterling, W.T. 1989. *Numerical Recipes in Pascal: The Art of Scientific Computing.* Cambridge University Press, Cambridge, MA.

Pritchard, D.J., Askew, C.R., Carpenter, D.D., Glendinning, I., Hey, A.J.G. and Nicole, D.A. 1987. Practical parallelism using transputer arrays. *Lecture Notes in Computer Science 258*, 278–294.

Rabin, M.O. 1980. Probabilistic algorithms for testing primality. *Journal of Number Theory 12*, 128–138.

Rivest, R.L., Shamir, A. and Adleman, L.M. 1978. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM 21*, 120–126.

Saltz, J.H., Naik, V.K. and Nicol, D.M. 1987. Reduction of the effects of the communication delays in scientific algorithms on message passing MIMD architectures. *SIAM Journal on Scientific and Statistical Computing 8*, 1, s118–s134.

Seitz, C.L. 1985. The Cosmic Cube. *Communications of the ACM 28*, 22–33.

Shih, Z., Chen, G. and Lee, R.T.C. 1987. Systolic algorithms to examine all pairs of elements. *Communications of the ACM 30*, 161–167.

Ulam, S. 1986. *Science, Computers, and People: From the Tree of Mathematics*, Birkhäuser, Boston, MA.

Valiant, L.G. 1989. Optimally universal parallel computers. In *Scientific Applications of Multiprocessors*, R. Elliott and C.A.R. Hoare, Eds. Prentice Hall, Englewood Cliffs, NJ, 17–20.

von Neumann, J. 1966. *Theory of Self-Reproducing Automata.* Edited and completed by A.W. Burks, University of Illinois Press, Urbana, IL.

Wirth, N. 1976. *Algorithms + Data Structures = Programs.* Prentice Hall, Englewood Cliffs, NJ.

Young, D.M. 1971. *Iterative Solution of Large Linear Systems.* Academic Press, New York.