The Nature of Parallel Programming[*]

(1989)

Parallel programming is the art of writing programs for computers that perform many operations simultaneously. This essay discusses the nature of parallel programming without going into technical details. It uses a sorting problem to illustrate what it means to solve a problem in parallel, how we write parallel programs, how parallel computers execute them, and how fast they run. The author expects that scientific users of parallel computers may find ease of programming more important than maximum performance. He suggests ways of making this possible.

Asking the right questions

As a computer scientist I have been fascinated by parallel programming since I first encountered it twenty-five years ago. I welcome this opportunity to explain the essence of my field to scientists and engineers.

Parallel programming is the art of writing programs for computers that perform many operations simultaneously. Parallel computers with tens and hundreds of processors are already commercially available. Researchers are now working on computers with thousands of processors. Programming these machines sounds like an exciting idea until you try it. It is often too complicated, but for the wrong reason: Most of our programming languages and computer architectures do not really support parallelism as well as they could.

In this essay I will discuss the nature of parallel programming without going into technical details. It seems natural to begin by asking some fundamental questions:

- What does it mean to solve problems in parallel?

[*]P. Brinch Hansen, The nature of parallel programming. Inaugural Symposium at Syracuse University, Syracuse, NY, March 1989. In *Natural and Artificial Parallel Computation*, M.A. Arbib and J.A. Robinson, Eds. The MIT Press, Cambridge, MA, (1990), 31–46. Copyright © 1990, Massachusetts Institute of Technology.

- How do we write parallel programs?
- How do parallel computers execute such programs?
- How fast can parallel programs run?
- Can we make parallel programming easier?

I will try to answer these questions by stripping away the inessentials and penetrating to the core of the problem.

One step at a time

A well-chosen example is often an important source of insight. I will use a sorting problem to illustrate the ideas of parallel programming. Once you understand these ideas, the example becomes merely a detail in the great scheme of things.

Bridge players often sort their hands by picking up one card at a time and inserting it where it belongs. This is the simplest way to sort a small number of cards. But, if you are sorting thousands of cards, there are much faster methods.

One of them is called *merge sorting.* As early as 1945 John von Neumann wrote computer programs for merge sorting. Let me describe how you would use this method to sort eight numbers manually:

Write each number on a separate card and place the cards in front of you in any order

$$\boxed{4}$$
$$\boxed{1}$$
$$\boxed{6}$$
$$\boxed{2}$$
$$\boxed{8}$$
$$\boxed{5}$$
$$\boxed{3}$$
$$\boxed{7}$$

Pick up two cards at a time and put them down as an ordered pair of cards. You now have four ordered pairs

$$\begin{array}{cc} \boxed{1} & \boxed{4} \\ \boxed{2} & \boxed{6} \\ \boxed{5} & \boxed{8} \\ \boxed{3} & \boxed{7} \end{array}$$

Take the first two pairs

$$\begin{array}{cc} \boxed{1} & \boxed{4} \\ \boxed{2} & \boxed{6} \end{array}$$

and merge them into a single, ordered sequence of four cards

$$\boxed{1}\ \boxed{2}\ \boxed{4}\ \boxed{6}$$

Then combine the last two pairs into an ordered sequence

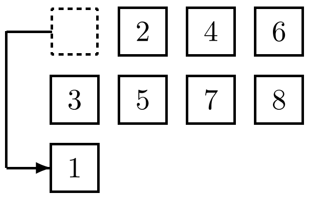
$$\boxed{3}\ \boxed{5}\ \boxed{7}\ \boxed{8}$$

Finally merge the two ordered sequences of four cards each into a single, ordered sequence of eight cards

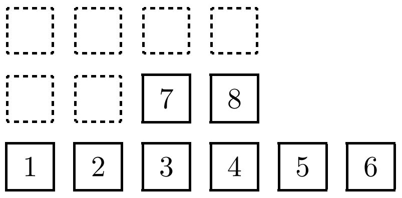
$$\boxed{1}\ \boxed{2}\ \boxed{3}\ \boxed{4}\ \boxed{5}\ \boxed{6}\ \boxed{7}\ \boxed{8}$$

This completes the sorting.

Let me explain the merging more carefully. The first card in the merged sequence is the smallest of the eight cards. Since the two original sequences are ordered, the smallest card is the first card of one of these sequences. Take this card and place it below the two sequences as the first card of the merged sequence

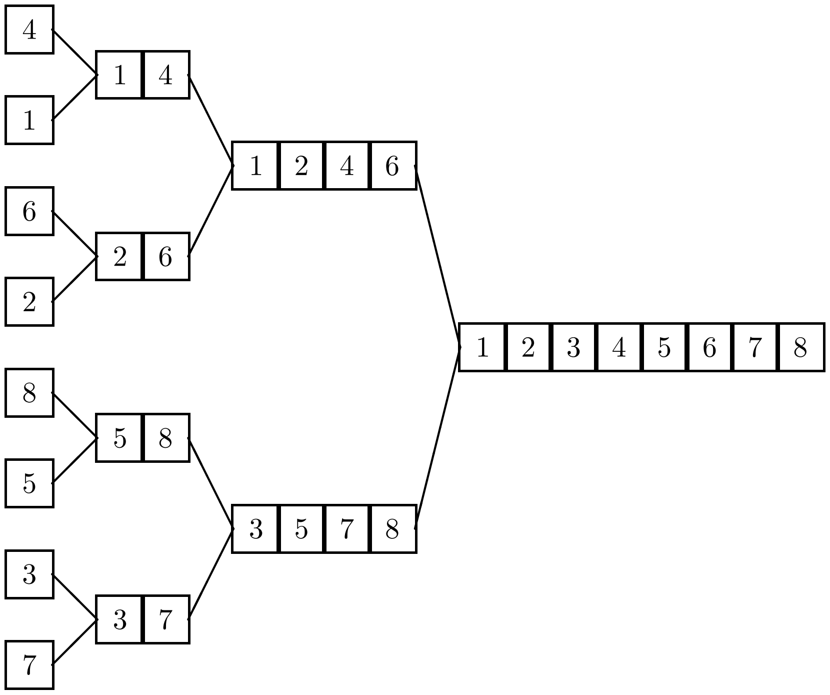


Continue to remove the smallest remaining card and add it to the merged sequence until one of the original sequences is empty



Then add the rest of other sequence to the merged one.

The mergesort works by repeatedly merging shorter, ordered sequences into longer ones. Eight sequences of length 1 are merged into four sequences of length 2, which, in turn, are merged into two sequences of length 4, and finally into one sequence of length 8. You can picture the sorting process as a tree of merging steps

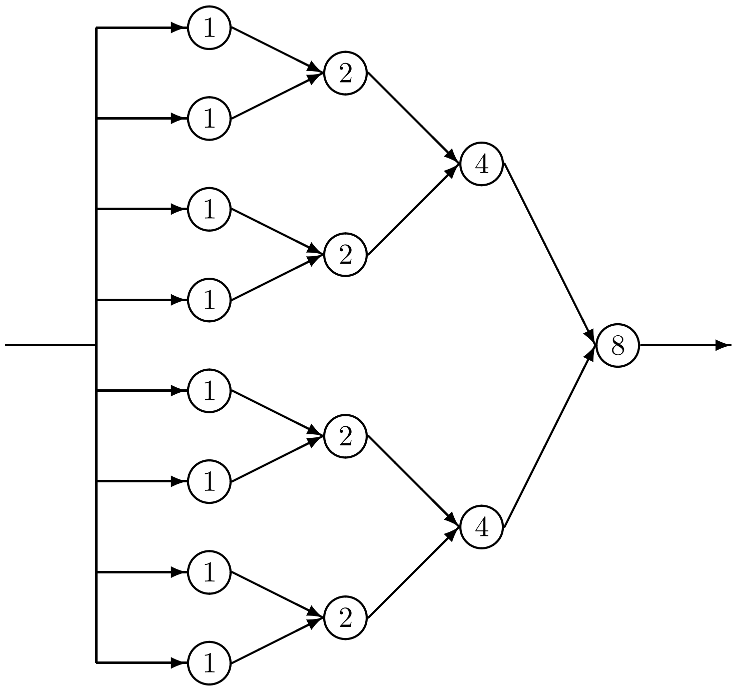


I have described merge sorting as a *sequential process* performed one step at a time. This is indeed how it would be done on a traditional computer. Merge sorting can, however, be speeded up by performing the merging steps simultaneously on a parallel computer.

Running in parallel

The mergesort solves a problem by dividing it into smaller instances of the same problem. The subproblems can be solved independently of one another. This property makes the algorithm well-suited for parallel execution.

We can build a parallel computer that sorts eight numbers. This machine is organized as a *tree*. It consists of 15 processors connected by 16 communication channels. The processors and channels are drawn as circles and arrows



The eight processors on the left are the *leaves* of the tree. The single processor on the right is the *root* of the whole tree. Each processor in the middle is the root of a smaller tree within the larger one.

The eight numbers move from left to right in the tree. Each leaf receives a single number from a shared channel and sends it to its successor in the tree. Each root receives two sequences of numbers from its predecessors and sends them as a merged sequence to its successor. The main root sends the eight numbers through a channel is ascending order. Each processor merges either 1, 2, 4 or 8 numbers as shown in the circles.

The processors operate in parallel. Processors which are at the same vertical *level* in the tree communicate simultaneously with their neighbors. Each root holds only two numbers at a time. When a root has sent a number to the right, it immediately receives another one from the left. Meanwhile its successors can process the previous number.

In general when a tree machine sorts N numbers, we will simplify the discussion a bit by assuming that N is a power of two. In other words, N is a number in the series

$$1, 2, 4, 8, \ldots, 1024, \ldots$$

Since the number of processors doubles from one level to the next, the total number of processors in the tree machine is $1 + 2 + 4 + \cdots + N$. This adds up to $2N-1$. So the tree machine needs 2047 processors to sort 1024 numbers. If N is large the number of processors is almost 2N.

In practice we do not always have two separate processors and channels for each of the sorted numbers. We often have a parallel computer with a much smaller number of processors and channels. We use these processors and channels to *simulate* a large number of slower processors and channels. This simulation is a crucial part of the implementation of a programming language for parallel programming.

Getting down to fundamentals

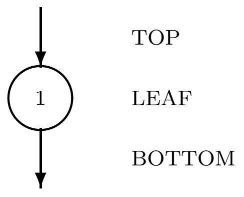
A parallel computation may involve millions of small steps. The mind obviously cannot comprehend such a multitude of simultaneous events in detail. We must impose order on the complexity by describing it in terms of a small number of general concepts. The most important *abstractions* in parallel programming are processes and communication.

A *process* is an abstract model of a computation. A sequential process is a sequence of steps which take place one at a time. A parallel process is a set of processes performed simultaneously. And a *communication* is a transfer of data from one process to another. These concepts are the essence of parallel programming. The rest is detail.

From now on we will view the parallel mergesort as a tree of processes. Whether these processes run on real or simulated processors is a technical detail.

When you have discovered powerful thinking tools, it becomes essential to express them in a *concise notation*. For parallel computations we need a programming language which can describe individual processes and combinations of processes precisely. I will not discuss the merits of particular programming languages. Instead I have invented a simple notation which will give you the flavor of a parallel language.

The simplest processes in a parallel merge tree are the sequential leaves. A leaf is connected to two channels.



All leaves behave in the same way. We can therefore write a single procedure that describes the behavior of these identical processes. In a programming language this procedure might look as follows

```
LEAF(top, bottom) =
  1.  receive(x, top);
  2.  send(x, bottom).
```

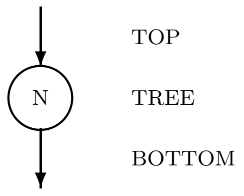
The notational details are unimportant. The procedure consists of two numbered steps. Each step describes an action performed by a leaf:

1. Receive a number x through the top channel.

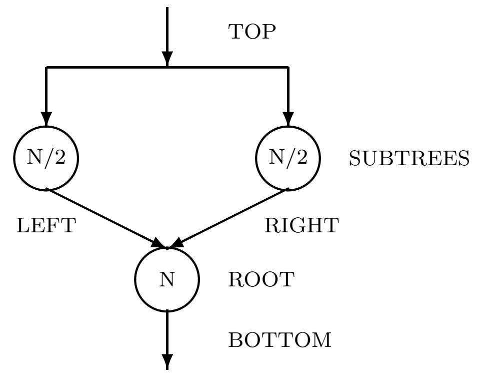
2. Send the same number through the bottom channel.

When a leaf has done this, it terminates and ceases to exist.

From a user's point of view a sorting tree is a single process that receives N numbers through one channel and sends them in ascending order through another channel.



A closer look reveals that a sorting process takes one of two forms. A tree that "sorts" one number only is just a single leaf. A tree that sorts more than one number consists of a root process and two smaller trees connected by a left and a right channel.



In this picture each subtree is viewed as a single process. Internally it may be composed of other processes, but right now we choose to ignore these details.

The following procedure defines the behavior of a tree that sorts N numbers:

```
TREE(N, top, bottom) =
  if N = 1 run LEAF(top, bottom)
  if N > 1 run
          TREE(N/2, top, left),
          TREE(N/2, top, right),
          ROOT(N, left, right, bottom)
        in parallel
```

Again the programming symbols are not important. Here is what they mean:

1. If N = 1 a tree is just a single leaf. The effect of the command

   run LEAF(top, bottom)

   is to activate a leaf process with access to the top and bottom channels of the tree. When the leaf terminates, the tree ceases to exist.

2. If N > 1 a tree splits into two subtrees and a root process running in parallel. When all three processes have terminated, the whole tree disappears.

The above procedure defines a tree in terms of smaller trees. A parallel process which is defined in terms of other processes of the same kind is called a *recursive process.*

A root is a sequential process that receives two ordered sequences from a left and a right channel and sends a merged sequence through a bottom channel. I will omit the programming details of this process and describe it in English.

```
ROOT(N, left, right, bottom) =
  1. Receive the first left and right numbers.
  2. Send the smaller of the two through the
      bottom channel and replace it by the
      next number (if any) from the same
      left or right sequence;
  3. Repeat step 2 until the left or right
      sequence is empty;
  4. Copy the rest (if any) of the other
      left or right sequence.
```

Parallelism is a mechanism for splitting larger computations into smaller ones which can be performed simultaneously. A notation for recursive processes is essential in a parallel programming language. The reason is simple. In a highly parallel program it is impractical to formulate thousands of processes with different behaviors. We must instead rely on repeated use of a small number of behaviors. The simplest problems that satisfy this requirement are those that can be reduced to smaller problems of the same kind and solved by combining the partial results. Recursion is the natural programming tool for expressing these *divide and conquer* algorithms.

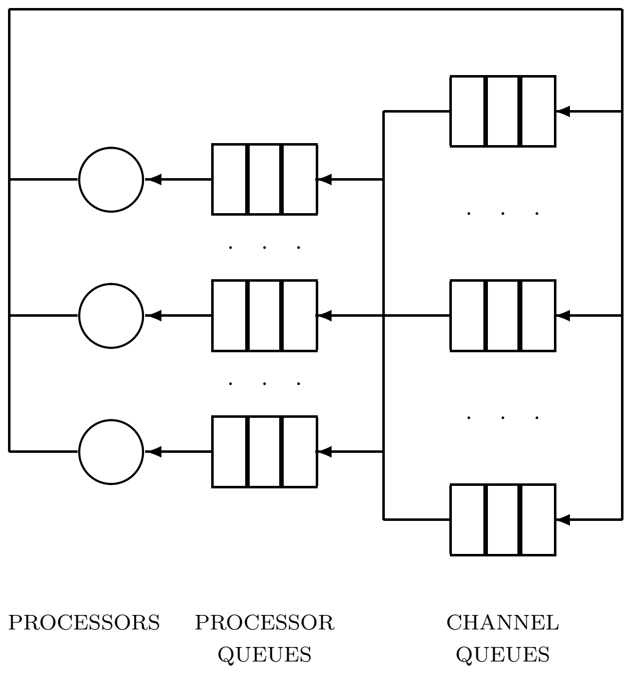
A good programming language has an air of economy and an element of surprise. The economy comes from using a small number of concepts: processes, channels and communication. The surprise is the elegance and utility of recursive, parallel processes. This wonderful concept can be used not only for sorting, but also for fast Fourier transforms, N-body simulation, computational geometry and matrix multiplication on parallel computers.

Hidden complexity

A programming language should hide irrelevant details of computer hardware and support more abstract models of computation efficiently. You will immediately appreciate the significance of this requirement if you catch a glimpse of what really happens when a parallel computer executes a program.

In a parallel computation the number of processes often exceeds the number of physical processors. This is only too obvious when you run thousands of processes on a parallel computer with ten processors only. Programs that are more parallel than the computer itself are executed by switching the processors rapidly between processes to give the illusion that they are executed simultaneously on a slower, parallel computer.

The simplest kind of parallel computer is a *multiprocessor* which consists of tens of processors connected to a common memory. A language implementer views a multiprocessor as a *queueing system* with a finite population of customers (the processes) and multiple servers (the processors).



In the common memory each process is represented by a small block of memory called a *process record*. This record holds the parameters and local variables of the process. Each processor has a separate queue of processes that are ready to run. The queue is a list of process records chained together.

An idle processor removes a process from its queue and executes it until the process, for example, is ready to send a message through a channel. The processor then puts the process in a queue associated with the channel. Immediately afterwards the processor resumes the execution of another process from its own queue.

When a process is ready to receive from the same channel, a message is copied from the record of the sending process to the record of the receiving process. The delayed process is then moved from the channel queue to one of the processor queues. Sooner or later the corresponding processor resumes the execution of the process.

To achieve the highest performance of a parallel computer it is important to divide a computation evenly among the processors, so that all of them can work at full speed whenever possible. This is called *load balancing.*

On a multiprocessor it is easy to balance the load, if the processors share a table defining the lengths of all processor queues. When a processor removes a process from a channel queue, it scans the table and puts that process in the shortest processor queue. Load balancing is in effect achieved by letting communicating processes *migrate* from processor to processor.

If several processors simultaneously attempt to manipulate the same queue, they must be forced to do it one at a time in unpredictable order. So parallelism introduces an element of chance in computation. The study of machines with *nondeterministic* behavior is still a fertile area of research in computer science.

A well-designed programming language enables the programmer to ignore these implementation details of processes and communications. However, the programmer cannot ignore the efficiency of the language implementation.

Limits to parallelism

The parallel mergesort is not particularly efficient. To understand why, we need a theoretical model of its performance.

The most critical performance figures for a highly parallel program are the execution times of process activation, communication and termination. We will assume that each of these steps takes exactly one unit of time. This is a reasonable approximation for the parallel mergesort written in the programming language Joyce and executed on a multiprocessor (the Encore Multimax).

It is customary to compare the running time T_1 of a parallel program on a single processor with its running time T_p on p processors.

For large N the *serial running time* is approximately

$$T_1 = N(L + 5) \text{ units}$$

where L is the number of process *levels* in the tree. T_1 includes the activation and termination of 2N processes and the communication of N numbers through all levels in the tree. I cannot go into further details here. You will find them in Brinch Hansen (1989b). For mathematically inclined readers: It turns out that L = logN + 1, where logN is the binary logarithm of N.

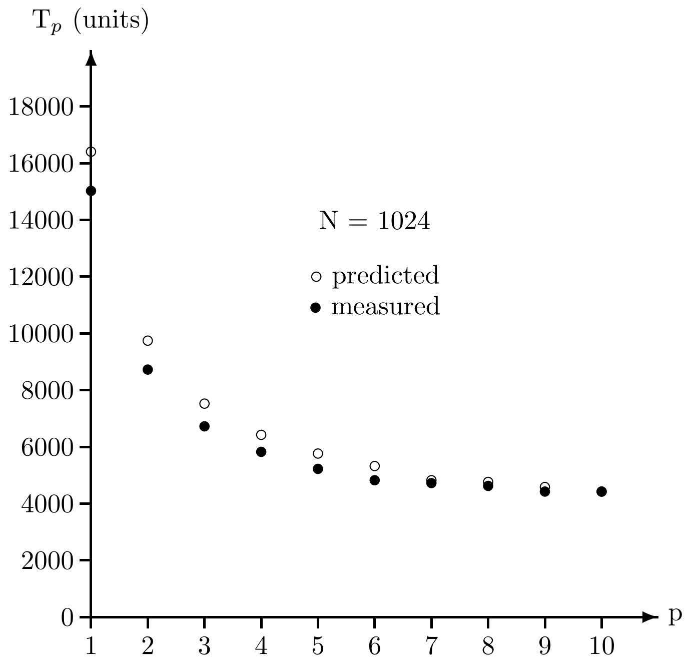
A tree that sorts 1024 numbers has 11 process levels. It takes 16384 time units to run the sorting on a single processor.

The *parallel running time* is approximately

$$T_p = N(3 + (L + 2)/p) \text{ units}$$

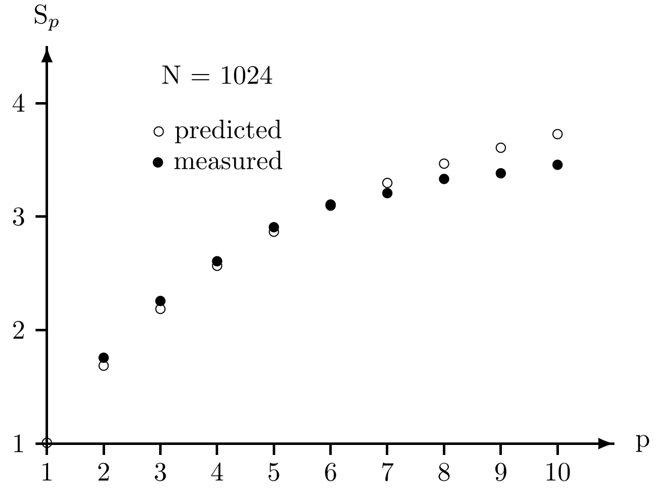
Most of the steps are now executed p times faster. But there are 3N steps which cannot be speeded up by the use of multiple processors. These serial steps can be attributed to the initial creation of the process tree and the sequential communications of the root. (See the paper cited above.)

The following figure shows the predicted and measured running times (in time units) for sorting 1024 numbers on 1 to 10 processors. The run times were measured on the Multimax.



The *speedup* $S_p = T_1/T_p$ defines how much faster a program runs on p processors compared to a single processor. Ideally p processors should make a parallel program run p times faster. If the speedup is less than p, it means that some processors are idle part of the time.

The next figure shows the predicted and measured speedup of the parallel sorting of 1024 numbers. The approximate model is fairly accurate.



No matter how many processors you use, parallel merge sorting cannot be speeded up by more than

$$S_{max} = (L + 5)/3$$

For N = 1024 the maximum speedup is five only. The limiting factor is the number of serial steps in the parallel algorithm. This is known as *Amdahl's law.* Many other parallel algorithms have similar limitations.

As you add more processors, the algorithm runs slightly faster, but wastes more and more processor time. In practice one should probably stop adding more processors when

$$S_p = p/2$$

since more than half of the processing capacity will be wasted beyond this point. Consequently the speedup of the parallel mergesort is limited to

$$S_p = (L + 8)/6 \qquad \text{for } p = (L + 8)/3$$

For N = 1024 the sorting program runs only three times faster on six processors. And it does not pay to use more processors. This modest speedup is acceptable for parallel computers with tens of processors, but not for thousands of processors.

After this brief discussion of the nature of parallel programming it is time to draw conclusions.

Looking ahead

As we move from tens to thousands of processors, our parallel algorithms will often be unable to run that much faster. There is only one way out of this problem: We must perform numerous experiments with new algorithms until we know how to use highly parallel computers well.

Scientific computer users, who are primarily interested in getting numerical results fast, will constantly have to reprogram new parallel architectures and may become increasingly frustrated at the difficulty of doing this.

Parallel programs are often written in the conventional languages Fortran and C extended with subroutines for parallelism. To my taste these programs are difficult to read and lack the beauty which scientists expect of their own research. This state of affairs puts a scientist in an unreasonable dilemma: should you study the unnecessary complexity of existing programs or reinvent similar ones?

I am convinced that *the most important task in computational science is to make the programming of parallel computers easier*. This is even more important than increasing computational power, and we should be prepared to sacrifice some performance to solve the programming problem. With this important goal in mind, I propose three requirements for the next generation of parallel hardware and software.

- *Requirement 1*: Parallel programs must be written in abstract notations that hide irrelevant hardware detail and express parallelism concisely.

The essence of parallel computing is process creation and communication. These basic operations are implemented in software on most parallel computers. Consequently they are an order of magnitude slower than subroutine calls in Fortran. Due to the lack of hardware support for the process concept, the parallel mergesort is only slightly faster than the best sequential method for sorting (quicksort)!

- *Requirement 2*: Process creation and communication must be hardware operations which are only an order of magnitude slower than memory references.

In the future we can expect to see highly parallel programs which use a mixture of process structures simultaneously. Sorting trees and image meshes may, for example, coexist. A computation may also change its process structures from one phase to another. When a parallel program spawns numerous processes with changing topologies it is not meaningful to ask the programmer to specify on which processor each process should run. This leads me to the last requirement.

- *Requirement 3*: Most parallel computers must be able to distribute the computational load automatically with reasonable efficiency.

The driving force

I will end on a personal note. Parallel programming is not just about computation. It is about beautiful ideas that happen to be useful. The study of parallelism is driven by the same powerful ideas as the rest of science and mathematics. They are the concepts of *number*, *form*, *arrangement*, *movement* and *chance*. In mathematics, these notions led to arithmetic, geometry, combinatorics, calculus and probability. In parallel programming, they reappear as data, processes, networks, communication and nondeterminism.

The most enjoyable thing about computer programming is the insight it provides into the deep similarities of all creative endeavors.

Acknowledgements

This essay has been improved by valuable advice from Gideon Frider, Milena Hansen, Carlos Hartmann, Erik Hemmingsen, Anand Rangachari and Ernest Sibert.

This work was conducted using the computational resources of the Northeast Parallel Architectures Center (NPAC) at Syracuse University, which is funded by DARPA, under contract to Rome Air Development Center (RADC), Griffiss AFP, NY.

Digging deeper

Athas, W.C., and Seitz, C.L. 1988. Multicomputers: Message-passing concurrent computers. *IEEE Computer 21*, (August), 9–24. (A status report on a research project.)

Barnes, J., and Hut, P. 1986. A hierarchical O(NlogN) force calculation algorithm. *Nature 324*, (December), 446–449. (A parallel divide and conquer algorithm for the N-body problem.)

Bitton, D., DeWitt, D.J., Hsiao, D.K., and Menon, J. 1984. A taxonomy of parallel sorting. *ACM Computing Surveys 16*, (September), 287–318. (Includes sorting algorithms that are much faster than the parallel mergesort.)

Brinch Hansen, P. 1987. Joyce—A programming language for distributed systems. *Software—Practice and Experience 17*, 1 (January), 29–50. (Explains Joyce by examples.) *Article 18.*

Brinch Hansen, P. 1989a. A multiprocessor implementation of Joyce. *Software—Practice and Experience 19*, 6 (June), 579–592. (A detailed explanation of how parallel processes and communication channels are implemented.) *Article 19.*

Brinch Hansen, P. 1989b. Analysis of a parallel mergesort. School of Computer and Information Science, Syracuse University, Syracuse, NY. (An exact performance model of the parallel mergesort.)

Dally, W.J. 1987. *A VLSI Architecture for Concurrent Data Structures.* Kluwer Academic Publishers, Norwell, MA. (Proposes a programming language and a parallel computer based on message passing.)

Fox, G., Johnson, M., Lyzenga, G., Otto, S., Salmon, J., and Walker, D. 1988. *Solving Problems on Concurrent Processors*, Vol. 1. Prentice-Hall, Englewood Cliffs, NJ. (Explains how many scientific problems can be solved on hypercube architectures. Chapter 11 outlines a parallel divide and conquer algorithm for the fast Fourier transform.)

Gehani, N., and McGettrick, A.D. 1988. *Concurrent Programming.* Addison-Wesley, Reading, MA. (A collection of classical papers on parallel programming languages.)

Gustafson, J.L. 1988, The scaled-size model: A revision of Amdahl's law. *Supercomputing 88*, International Supercomputing Institute, St. Petersburg, FL, 130–133. (Reports impressive performance of parallel computations by redefining the meaning of speedup. The problem size is scaled up in proportion to the number of processors to finesse the limitations of Amdahl's law.)

Hillis, W.D. 1985. *The Connection Machine.* The MIT Press, Cambridge, MA. (A clear description of a revolutionary idea: A parallel architecture with 64,000 synchronous processors executing identical processes in lock step.)

Hoare, C.A.R. 1985. *Communication Sequential Processes.* Prentice-Hall, Englewood Cliffs, NJ. (A mathematical theory of communicating processes.)

Jenkins, R.A. 1989. New approaches in parallel computing. *Computers in Physics 3*, (January–February), 24–32. (A recent survey of parallel architectures.)

Kahn, G., and MacQueen, D.B. 1977. Coroutines and networks for parallel processes. In *Information Processing 77*, B. Gilchrist, Ed. North Holland Publishing, Amsterdam, The Netherlands, 993–998. (Presents an elegant parallel language with recursive processes.)

May, D. 1988. The influence of VLSI technology on computer architecture. *Supercomputing 88*, International Supercomputing Institute, St. Petersburg, FL, 247–256. (Includes a short overview of the nonrecursive language occam and the Transputer processor which has machine instructions for process creation, communication and termination.)

Preparata, F.P., and Shamos, M.I. 1985. *Computational Geometry: An Introduction.* Springer-Verlag, New York. (Chapter 3 describes a parallel divide and conquer algorithm for finding the convex hull of a finite set of points in the plane.)

Quinn, M.J. 1987. *Designing Efficient Algorithms for Parallel Computers.* McGraw-Hill, New York. (A short introduction to parallel sorting and searching, matrix multiplication, and graph algorithms.)

Whiddett, D. 1987. *Concurrent Programming for Software Engineers.* Halstead Press, New York. (Explains the three major paradigms for parallel programming: Monitors, message passing and remote procedures.)