Syracuse University
SURFACE

Electrical Engineering and Computer Science Technical Reports

College of Engineering and Computer Science

12-1991

Parallel Divide and Conquer

Per Brinch Hansen
Syracuse University, School of Computer and Information Science, pbh@top.cis.syr.edu

Follow this and additional works at: http://surface.syr.edu/eecs_techreports

Part of the Computer Sciences Commons

Recommended Citation
Hansen, Per Brinch, "Parallel Divide and Conquer" (1991). *Electrical Engineering and Computer Science Technical Reports.* Paper 131.
http://surface.syr.edu/eecs_techreports/131

This Report is brought to you for free and open access by the College of Engineering and Computer Science at SURFACE. It has been accepted for inclusion in Electrical Engineering and Computer Science Technical Reports by an authorized administrator of SURFACE. For more information, please contact surface@syr.edu.

SU-CIS-91-45

Parallel Divide and Conquer

Per Brinch Hansen

December 1991

School of Computer and Information Science
Syracuse University
Suite 4-116, Center for Science and Technology
Syracuse, New York 13244-4100

Parallel Divide and Conquer [1]

PER BRINCH HANSEN

School of Computer and Information Science
Syracuse University, Syracuse, New York 13244, U.S.A.

December 1991

SUMMARY

We develop a generic divide and conquer algorithm for a parallel tree machine. From the generic algorithm we derive balanced, parallel versions of quicksort and the fast Fourier transform by substitution of data types, variables and statements. The performance of these algorithms is analyzed and measured on a Computing Surface configured as a tree machine with distributed memory.

KEY WORDS Parallel algorithms Programming paradigms Generic algorithms Divide and conquer Quicksort Fast Fourier transform Tree machine

INTRODUCTION

This is one of several papers that explore the benefits of developing generic algorithms for parallel programming paradigms that can be adapted to different applications [1–4]. In this paper we consider the *divide and conquer* paradigm for a parallel *tree machine* with distributed memory.

Divide and conquer is an elegant method for solving a problem: You divide the problem into smaller problems of the same kind, solve the smaller problems separately, and combine the partial results into a complete solution. The method is used recursively to split the problem into smaller and smaller problems until you reach a point where each problem is easy to solve.

This beautiful concept has led to fast sequential algorithms for sorting [5], Fourier transform [6], matrix multiplication [7], spatial proximity [8], convex hulls [9], and n-body simulation [10].

Parallelism is also a mechanism for splitting larger computations into smaller ones that can be performed simultaneously. For multicomputers, divide and conquer algorithms already exist for sorting, fast Fourier transform, and matrix multiplication [11].

[1]Copyright©1991 Per Brinch Hansen

We are more interested in the programming methodology of multicomputers than in solving specific problems. With this emphasis in mind we develop a generic divide and conquer algorithm for a tree machine. From the generic algorithm we derive balanced, parallel versions of *quicksort* and the *fast Fourier transform* by substitution of data types, variables, and statements. The performance of these algorithms is analyzed and measured on a *Computing Surface* with 31 transputers configured as a tree machine.

SEQUENTIAL PARADIGM

Since we are interested in principles rather than detail, we concentrate on divide and conquer algorithms with four simple properties:

1. A problem of size n and its solution are both defined by an array of n elements of the same type.

2. A problem of size 1 is its own solution.

3. A larger problem is solved by splitting it into two halves, which are solved separately.

4. A problem is solved by an in-place computation that replaces the elements of a single array by the corresponding solution without using additional arrays.

We begin by writing a sequential divide and conquer algorithm in the programming language Pascal (Algorithm 1).

```
type table = array [0..n-1] of T;

procedure solve(var a: table;
  first, last: integer);
var middle: integer;
begin
  if first < last then
  begin
    split(a, first, last, middle);
    solve(a, first, middle);
    solve(a, middle + 1, last);
    combine(a, first, last, middle)
  end
end
```

Algorithm 1

A complete problem and its solution are defined by an array of n elements of some type T. The procedure generally solves a subproblem in a *slice* of the array

a[first..last]

where

$$0 \leq \text{first} \leq \text{last} \leq \text{n}-1$$

A slice with one element only is left unchanged.

A larger slice is *split* into two smaller slices

$$\text{a[first..middle]} \qquad \text{a[middle+1..last]}$$

where

$$0 \leq \text{first} \leq \text{middle} < \text{last} \leq \text{n}-1$$

The subproblems are solved by recursive activations of the solution procedure, and the partial results are *combined* into a single solution to the original problem.

A complete problem is solved by transforming all the elements of an array a of size n

$$\text{solve(a, 0, n}-1)$$

The class of divide and conquer algorithms that we are considering is defined by Algorithm 1. The procedures for splitting problems and combining solutions depend on the nature of a specific application, such as sorting or Fourier transformation. We assume that split and combine define *in-place* transformations of a single array slice.

PARALLEL PARADIGM

The simplest parallel computer for divide and conquer computation is a *binary tree* of processors connected by communication *channels.* In Fig. 1 the processors and channels are shown as nodes and edges, respectively. The nodes at the top are the *leaves* of the tree. The single node at the bottom is the main *root.* Each node in the middle is the root of a subtree.

In the terminology of family trees, each root is called the *parent* of the two nodes immediately above it. These two nodes, in turn, are called the *children* of that parent.

Each node is connected to its parent by a *bottom* channel. In addition, each root is connected to its two children by a *left* and a *right* channel.

The main root inputs a complete problem from its bottom channel, splits it into two parts, and sends one part to its left child and the other part to its right child. The remaining roots repeat the splitting process. Eventually, each leaf inputs a problem through its channel, solves it, and outputs the solution through the same channel. Each root then inputs two partial solutions from its children and combines them into a single solution, which is output to its parent. Finally, the main root outputs the solution to the complete problem.

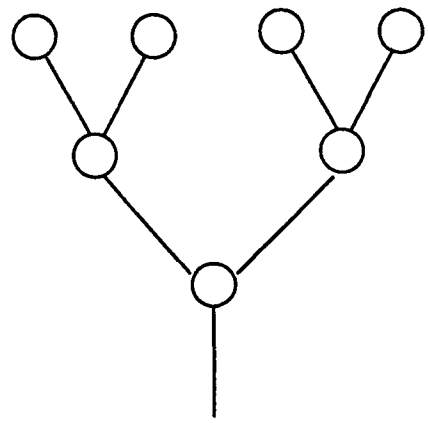

Fig. 1 A tree machine

The tree machine will be programmed in Pascal extended with statements for parallel execution and message communication.

The execution of k statements $S_1, S_2, \ldots, S_k$ as parallel processes is denoted

$$\textbf{parallel } S_1 | S_2 | \ldots | S_k \textbf{ end}$$

The parallel execution continues until every one of the k processes has terminated.

In this paper we assume that parallel processes run on separate processors without shared memory. Parallel processes communicate through synchronous channels only.

The input and output of an array slice $a[i..j]$ through a channel c are denoted

$$c?a[i..j] \qquad c!a[i..j]$$

A communication may include the bounds i and j

$$c?(i,\ j,\ a[i..j]) \qquad c!(i,\ j,\ a[i..j])$$

The tree machine activates a tree of processes that run in parallel on p processors (Algorithm 2). Initially the main root executes this procedure as a sequential process. A tree with more than one processor consists of a root and two subtrees running in parallel. Each subtree has $(p-1)/2$ processors. The tree processes continue to split themselves recursively into parallel subtrees and roots until they reach the point where each process is a leaf process.

```
procedure tree(p: integer; bottom: channel);
var left, right: channel;
begin
  if p > 1 then
    parallel
      root(bottom, left, right)|
      tree((p − 1) div 2, left)|
      tree((p − 1) div 2, right)
    end
  else leaf(bottom)
end
```

Algorithm 2

A root process inputs a problem, splits it into two problems, which are solved by its children, and outputs the combined result (Algorithm 3).

```
procedure root(bottom, left, right: channel);
var a: table; first, last, middle: integer;
begin
  bottom?(first, last, a[first..last]);
  split(a, first, last, middle);
  left!(first, middle, a[first..middle]);
  right!(middle + 1, last, a[middle + 1..last];
  left?a[first..middle];
  right?a[middle + 1..last];
  combine(a, first, last, middle);
  bottom!a[first..last]
end
```

Algorithm 3

A leaf process inputs a problem, solves it by means of the sequential divide and conquer algorithm, and outputs the solution (Algorithm 4).

```
procedure leaf(bottom: channel);
var a: table; first, last: integer;
begin
  bottom?(first, last, a[first..last]);
  solve(a, first, last);
  bottom!a[first..last]
end
```

Algorithm 4

Algorithms 1–4 define the behavior of a tree machine with p processors solving a divide and conquer problem in parallel. We have deliberately ignored the system-dependent details of processor allocation.

PARALLEL QUICKSORT

Our first divide and conquer example is the *quicksort* algorithm, which splits an array of integers into two parts and sorts the left and right parts separately. (Algorithm 5). The splitting is repeated recursively until the algorithm sorts a single element only (by an empty operation).

```
type table = array [0..n−1] of integer;

procedure quicksort(var a: table;
  first, last: integer);
var i, j: integer;
begin
  if first < last then
  begin
    partition(a, i, j, first, last);
    quicksort(a, first, j);
    quicksort(a, i, last)
  end
end
```

Algorithm 5

The *partition* algorithm selects an arbitrary key value from an array slice and splits the slice into two pieces with the property that no element in the left piece is larger than the key, and no element in the right piece is smaller than the key. Algorithm 6 uses the value of the middle element as the key.

```
procedure partition(var a: table; var i,
  j: integer; first, last: integer);
var ai, key: integer;
begin
  i := first; j := last;
  key := a[(i + j) div 2];
  while i <= j do
  begin
    while a[i] < key do i := i + 1;
    while key < a[j] do j := j - 1;
    if i <= j then
    begin
      ai := a[i]; a[i] := a[j];
      a[j] := ai;
      i := i + 1; j := j - 1
    end
  end
end
```

Algorithm 6

The run time of partition is $O(n)$. The average run time of quicksort is $O(n \log n)$. The worst case sorting time is $O(n^2)$.

Quicksort can be derived from Algorithm 1 by making the following substitutions:

1. Type T is replaced by type integer.
2. Procedure solve is renamed quicksort.
3. Indices middle and middle+1 are replaced by j and i.
4. Split is replaced by

   partition(a, i, j, first, last)

5. Combine is empty.

It follows immediately that a *parallel quicksort* is obtained by making the same substitutions in Algorithms 3 and 4.

Unfortunately, the partition procedure produces array slices of unpredictable sizes. In the best case two slices are of equal length. In the worst case the smallest size has one element only. The unpredictable nature of quicksort causes load imbalance on a multicomputer [11]. 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.

On a tree machine with 31 processors each leaf receives a problem of size $n/16 = 0.06n$, provided the splitting is balanced. However, if partition on the average splits a problem of size m into two problems of size $0.6m$ and $0.4m$, respectively, two of the leaves receive problems of size $0.6^4n = 0.13n$ and $0.4^4n = 0.03n$.

We will call this algorithm the *unbalanced parallel quicksort.*

Fortunately, quicksort can be balanced by using a different splitting algorithm (Algorithm 7).

```
procedure quicksort(var a: table;
  first, last: integer);
var middle: integer;
begin
  if first < last then
  begin
    middle := (first + last) div 2
    find(a, first, last, middle);
    quicksort(a, first, middle);
    quicksort(a, middle + 1, last)
  end
end
```

Algorithm 7

The balanced quicksort can be derived from Algorithm 1 as follows:

1. Type T is replaced by type integer.
2. Procedure solve is renamed quicksort.
3. Split is replaced by

```
middle := (first + last) div 2;
find(a, first, last, middle)
```

4. Combine is empty.

The **find** algorithm repeatedly partitions an array slice into smaller and smaller pieces of unpredictable sizes until it has formed two halves with given first, last, and middle indices (Algorithm 8).

```
procedure find(var a: table; first,
  last, middle: integer);
var left, right, i, j: integer;
begin
  left := first; right := last;
  while left < right do
  begin
    partition(a, i, j, left, right);
    if middle <= j then right := j
    else if i <= middle then left := i
    else left := right
  end
end
```

Algorithm 8

If a single partitioning of n elements takes n time units, the average time required to find the middle element is

$$n + n/2 + n/4 + \cdots + 1 = 2n - 1$$

For large n *find* is twice as slow as *partition*. That is why the balanced *sequential* quicksort is of academic interest only. (We remark in passing that it is possible to write an iterative version of the balanced quicksort without a stack!)

On a parallel tree machine, a sorting problem must be distributed evenly among the leaves to obtain the best possible performance. As a compromise we will use the find algorithm in the roots only and the partition algorithms in the leaves. The resulting algorithm is called the *balanced parallel quicksort*. Measurements show that it consistently runs faster than the unbalanced algorithm.

The previous arguments are valid only for the *average* behavior of parallel quicksorting. In the *worst* case both algorithms perform very poorly. The correctness of the standard quicksort, partition, and find algorithms is proven in [12, 13].

PARALLEL FFT

Our second example is the *fast Fourier transform* (FFT), which computes the frequency components of a signal that has been sampled n times [6]. The theory behind the FFT is explained in [14], which includes an FFT procedure written in standard Pascal. Here we omit details by extending Pascal with complex numbers and complex arithmetic.

The FFT is an in-line transformation of an array of n complex numbers (Algorithm 9). This algorithm should be used only when n is a power of two, if necessary by

padding the data with zeros up to the next power of two [15]. The array elements must be permuted in *bit-reversed* order before the FFT computation begins [14].

The transform of a single number is the number itself. The FFT splits a larger sequence into two halves, computes the transform of each half separately, and combines the two transforms of size $n/2$ into a single transform of size n.

```
type table = array [0..n−1] of complex;

procedure fft(var a: table; first,
  last: integer);
var middle: integer;
begin
  if first < last then
  begin
    middle := (first + last) div 2;
    fft(a, first, middle);
    fft(a, middle + 1, last);
    combine(a, first, last)
  end
end
```

Algorithm 9

Algorithm 10 defines the combination of two transforms into one. Since n is a power of two, this procedure does not require a *middle* parameter.

The run times of the combine and fft procedures are $O(n)$ and $O(n \log n)$, respectively.

The FFT can be obtained by making the following changes to Algorithm 1:

1. Type T is replaced by type complex.
2. Procedure solve is renamed fft.
3. Split is replaced by

   middle := (first + last) **div** 2

4. Combine is replaced by

   combine (a, first, last)

A *parallel fft* is obtained by making the same substitutions in Algorithms 3 and 4.

```
procedure combine(var a: table; first,
  last: integer);
const pi = 3.14159265358979;
var even, half, odd, j: integer;
  w, wj, x: complex;
begin
  half := (last − first + 1) div 2;
  w := (cos(pi/half), sin(pi/half));
  wj := (1.0, 0.0);
  for j := 0 to half − 1 do
  begin
    even := first + j;
    odd := even + half;
    x := wj*a[odd];
    a[odd] := a[even] − x;
    a[even] := a[even] + x;
    wj := wj*w
  end
end
```

Algorithm 10

COMPLEXITY

We will analyze the average performance of a parallel divide and conquer algorithm under the assumption that every problem is split into two subproblems of equal size.

The *sequential run time* $T(1,n)$ is the average time required to solve a divide and conquer problem of size n on a single processor. The processor inputs and outputs n data items in $O(n)$ time and transforms them in $O(n \log n)$ time. So

$$T(1,n) = n(a\log(n) + b) \tag{1}$$

where a and b are system dependent constants for computation and communication in a leaf processor.

The *parallel run time* $T(p,n)$ is the average time it takes to solve a problem of size n on a binary tree machine with p processors, where $p+1$ and n are powers of two.

The tree consists of a root and two subtrees. The root transforms n items in $O(n)$ time units. The communication between the root and its parent also takes $O(n)$ units. The communication between the root and a subtree will be included in the run time of the subtree. Each subtree uses $(p-1)/2$ processors to solve a problem of size $n/2$.

The root does not terminate until the subtrees have terminated. Since the subtrees solve problems of the same size in parallel, the parallel run time of the tree is

$$T(p,n) = T\left(\frac{p-1}{2}, \frac{n}{2}\right) + (b+c)n$$

where b and c are constants for communication and computation in a root processor.

This recurrence has the solution

$$T(p,n) = T(1, n/q) + 2(b+c)(n - n/q)$$

where

$$q = (p+1)/2$$

is the number of leaf processors.

So the parallel run time is

$$T(p,n) = (n/q)(a\log(n/q) + b) + 2(b+c)(n - n/q) \tag{2}$$

For $p = q = 1$ this formula reduces to Eq. (1).

We will use the abbreviations

$$T_1 = T(1,n) \qquad T_p = T(p,n) \qquad S_p = T_1/T_p$$

The *speedup* S_p defines how much faster a parallel divide and conquer algorithm runs on p processors compared to a single processor.

On a hypothetical tree machine of infinite size the parallel run times of the roots would be

$$(b+c)(n + n/2 + n/4 + \cdots) = 2(b+c)n$$

This is a lower bound on the parallel run time of a finite tree machine

$$T_{\min} = 2(b+c)n \tag{3}$$

It is also an upper bound on the cost of distributing a problem in a tree machine and collecting the results.

We can now rewrite Eq. (2) as follows

$$T_p = T_1/q + (1 - 1/q)T_{\min} - an\log(q)/q \tag{4}$$

The parallel run time T_p is the sum of three terms:

1. The first term T_1/q is the sequential run time T_1 divided by the number of leaf nodes q.

2. The second term $(1 - 1/q)T_{\min}$ is the parallel run time of the roots.

3. The last term $an\log(q)/q$ is small compared to T_p. It reaches its maximum value $an/2$ for $q = 2$ ($p = 3$). So

$$\frac{an\log(q)/q}{T_p} \le \frac{an/2}{T_{\min}} = \frac{a}{4(b+c)}$$

For $a = b$ and $c = 2a$ the last term accounts for less than 8% of the run time.

For large q, the parallel run time T_p approaches $T_{\min}$.

The speedup cannot exceed $T_1/T_{\min}$, that is

$$S_{\max} = \frac{a\log(n) + b}{2(b+c)} \tag{5}$$

Suppose we wish to achieve a speedup that is close to the maximum speedup in the following sense

$$S_p \geq S_{\max}/(1+f)$$

where f is a given fraction. This inequality can be also expressed as follows

$$T_p \leq (1+f)T_{\min}$$

Using Eq. (4) we can extend the inequality further

$$T_p \leq T_1/q + (1-1/q)T_{\min} \leq (1+f)T_{\min}$$

From the second part of the inequality we derive the condition

$$q \geq (T_1/T_{\min} - 1)/(1+f)$$

In other words

$$S_p \geq S_{\max}/(1+f) \quad \text{for } q \geq (S_{\max} - 1)/f$$

Since $S_{\max}$ is $O(\log n)$ this result shows that the tree machine achieves $O(\log n)$ speedup with $O(\log n)$ processors. Since the sequential run time is $O(n \log n)$, an $O(\log n)$ speedup reduces the parallel run time to $O(n)$.

For $a = b$, $c = 2a$, and $n = 2^{20}$, the maximum speedup $S_{\max} = 3.5$. For $p = 31$ the actual speedup $S_p = 3.1$ corresponding to $f = 0.16$. The last term in Eq. (4) is 4% of T_p only.

PERFORMANCE

For the performance measurements we replaced Algorithms 5 and 9 by the iterative quicksort and fft defined in [16, 14]. We reprogrammed the parallel algorithms in occam and ran them on a Computing Surface with T800 transputers configured as a binary tree machine. The input data were produced by a random number generator [17].

For *balanced parallel sorting* of 32-bit random integers we measured

$$a = 3.8\mu s \qquad b = 5.6\ \mu s \qquad c = 2a$$

Table I shows measured (and predicted) sorting times for $n = 131072$ integers (in seconds).

Table I
Parallel Balanced Quicksort

p	T_p		S_p	
1	9.25	(9.20)	1.00	(1.00)
3	6.02	(6.08)	1.54	(1.51)
7	4.56	(4.65)	2.03	(1.98)
15	3.96	(3.99)	2.34	(2.31)
31	3.63	(3.63)	2.55	(2.53)

The performance limits are

$$T_{\min} = 3.46\ s \qquad S_{\max} = 2.66$$

Table II shows measured run times for the *unbalanced parallel quicksort.* ΔT_p is the relative time difference between the unbalanced and balanced algorithms. The unbalanced sort is 20–37% slower and is rather erratic.

Table II. Parallel
Unbalanced Quicksort

p	T_p	S_p	ΔT_p
1	9.25	1.00	0%
3	8.20	1.13	36%
7	5.46	1.69	20%
15	5.41	1.71	37%
31	4.85	1.91	34%

For *parallel FFT* of 128-bit random complex numbers we found

$$a = 25\ \mu s \qquad b = 22\ \mu s \qquad c = a$$

Table III shows measured (and predicted) FFT times for $n = 32768$ complex numbers (in seconds).

Table III. Parallel FFT

p	T_p		S_p	
1	13.23	(13.01)	1.00	(1.00)
3	7.74	(7.64)	1.71	(1.70)
7	5.12	(5.15)	2.58	(2.53)
15	3.99	(4.01)	3.32	(3.24)
31	3.47	(3.50)	3.81	(3.72)

The performance limits are

$$T_{\min} = 3.08\ s \qquad S_{\max} = 4.22$$

The run times for the parallel FFT do not include the sequential permutation time of the array

$$8.5n\ \mu s = 0.28\ s \qquad \text{for } n = 32768$$

CONCLUSION

We have presented a generic divide and conquer algorithm for a binary tree machine. From this algorithm we have derived balanced, parallel algorithms for quicksort and fast Fourier transform.

For problems of size n a tree machine achieves $O(\log n)$ speedup using $O(\log n)$ processors. The modest speedup makes divide and conquer algorithms unsatisfactory for multicomputers with hundreds or thousands of processors.

The disappointing performance of parallel divide and conquer cannot be attributed solely to the overhead of processor communication. Even if communication was instant ($b = 0$), the maximum speedup of a balanced quicksort would still be $0.25 \log n$ only. No matter how many processors you use to sort a million numbers, they can do it only five times faster than a single processor.

Although the degree of parallelism grows exponentially as a tree machine repeatedly divides a problem, the wave of computation still spreads sequentially through the levels of the tree. In a large tree machine, the main root alone accounts for almost half of the parallel run time.

The parallel algorithms presented have two obvious limitations:

1. When a tree machine with p processors solves a problem of size n, every node holds an array of size n. Consequently, the problem size is limited by the memory of a single node. This limitation can be removed by having a large memory of $O(n)$ size in the main root and halving the memory size of each processor at each level in the tree. This limits the total size of the distributed memory to $O(n \log p)$.

2. After dividing a sorting problem into smaller parts, a tree machine uses only half of its processors (the leaves) to reduce the sorting time. If we had used a *hypercube* instead of a tree machine, we could have written an algorithm that divides a sorting problem evenly among *all* nodes.

This is a valid criticism of small tree machines, but not of larger ones. If you use a multicomputer for large scientific computations, you probably already have at least 32 or 64 processors. So, if you have to sort numbers you may as well use all the processors you have.

A hypercube with p processors can solve a divide and conquer problem in the same time as a tree machine with $2p - 1$ processors [18]. A tree machine with 31 transputers can sort a million numbers in 31 s. And a hypercube with 32 transputers solves the same problem in 29 s, which is only 7% faster.

On multicomputers with 32–64 processors, parallel tree algorithms are practically as fast as hypercube algorithms and are simpler to program.

ACKNOWLEDGEMENTS

The helpful remarks of Jonathan Greenfield are gratefully acknowledged.

REFERENCES

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

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

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

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

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

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

7. V. Strassen, 'Gaussian elimination is not optimal', *Numerische Mathematik*, **13**, 354–356 (1969).

8. J. L. Bentley and M. I. Shamos, 'Divide-and-conquer in multidimensional space', *ACM Symposium on Theory of Computation*, 220–230 (1976).

9. W. Eddy, 'A new convex hull algorithm for planar sets', *ACM Transactions on Mathematical Software*, **3**, 398–403, (1977).

10. J. Barnes and P. Hut, 'A hierarchical $O(N \log N)$ force-calculation algorithm', *Nature*, **324**, 446–449 (1986).

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

12. M. Foley and C. A. R. Hoare, 'Proof of a recursive program: Quicksort. *Computer Journal*, **14**, 391–395 (1971).

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

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

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

16. E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms.* Computer Science Press, Rockville, MD, 1978.

17. S. K. Park and K. W. Miller, 'Random number generators: good ones are hard to find', *Communications of the ACM*, **31**, 1192–1201 (1988).

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