12-1991

Do Hypercubes Sort Faster Than Tree Machines?

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

SU-CIS-91-44

Do Hypercubes Sort Faster Than Tree Machines?

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

Do Hypercubes Sort Faster than Tree Machines? [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 balanced, parallel quicksort algorithm for a hypercube and compare it with a similar algorithm for a binary tree machine. The performance of the hypercube algorithm is measured on a Computing Surface.

KEY WORDS Parallel algorithms Quicksort Hypercube Tree machine

INTRODUCTION

Quicksort is probably the most widely used sequential method for sorting an array [1, 2]. On the average it sorts n items in $O(n \log n)$ time. In the worst case the sorting time is $O(n^2)$. The unpredictable nature of the algorithm makes it difficult to write an efficient, parallel quicksort for a multicomputer [3, 4].

In this paper we develop a *balanced, parallel quicksort* for a *hypercube* and compare it with a similar algorithm for a binary *tree machine* [4]. The performance of the hypercube algorithm is measured on a Computing Surface.

SEQUENTIAL QUICKSORT

The standard quicksort splits an array of integers in two parts and sorts the left and right parts separately. The splitting is repeated recursively until we are sorting single elements only (Algorithm 1).

[1]Copyright©1991 Per Brinch Hansen

```
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 1

In general, the algorithm sorts a *slice* of an array *a*

a[first..last]

The familiar *partition* algorithm [2] splits the slice into two smaller slices

a[first..j]   a[i..last]

where

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

HYPERCUBE SORTING

Initially we will discuss parallel sorting on a cube with 8 processor nodes only (Figure 1).

Each node can communicate with its three nearest neighbors through bidirectional channels. (The dotted channels are not used during parallel sorting). A fourth channel (shown for node 0 only) connects a node with the environment of the cube.

Since partition generally produces subproblems of unpredictable lengths, it may cause severe imbalance on a multicomputer. Later we will show how to balance a parallel quicksort. In the following we just assume that the nodes somehow always split sorting problems into smaller problems of equal (or nearly equal) size.

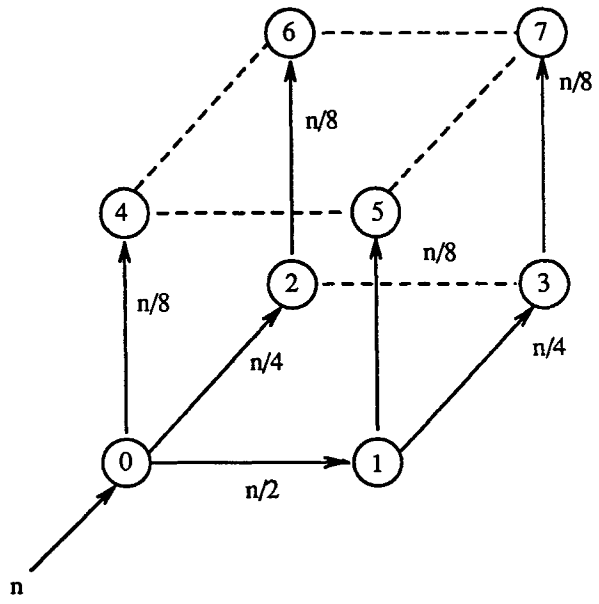

Fig. 1. Data distribution in a cube.

The cube sorts n numbers in three phases:

Splitting

1. Node 0 inputs n numbers, splits them into two halves, sends one half to node 1, and keeps the other half.

2. Node 0 splits half of the numbers into two fourths, sends one fourth to node 2, and keeps the other fourth. Simultaneously, node 1 splits the other half, sends one fourth to node 3, and keeps the other fourth.

3. Nodes 0, 1, 2, and 4 simultaneously send one eighth of the numbers to nodes 4, 5, 6, and 7, respectively, and keep the remaining eighths.

Sorting

4. All nodes work in parallel while each of them sorts one eighth of the numbers.

Combining

5. Node 0 inputs $n/8$ sorted numbers from node 4 and combines them with its own numbers into a sorted sequence of size $n/4$. At the same time, nodes 1, 2, and 3 communicate with nodes 5, 6, and 7, respectively, and form sorted sequences of size $n/4$.

6. Nodes 0 and 1 simultaneously input $n/4$ sorted numbers from nodes 2 and 3, respectively, and form sorted sequences of size $n/2$.

7. Finally, node 0 inputs $n/2$ sorted numbers from node 1, combines them with its own numbers and outputs a sorted sequence of size n 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.

In general, a hypercube has p processors, where p is a power of two

$$p = 2^d$$

The exponent $d = \log(p)$ is called the *dimension* of the hypercube. For a cube $p = 8$ and $d = 3$.

It is helpful to view a hypercube as a hierarchical system, where each *level* consists of a subset of the nodes. A cube has four levels of nodes

level	nodes
0	0..0
1	1..1
2	2..3
3	4..7

A sorting problem is distributed through the cube, one level at a time. First, the node at level 0 inputs a problem, then the node at level 1 inputs a subproblem, followed by the nodes at level 2, and finally the nodes at level 3.

In general a hypercube has $d + 1$ node levels.

```
function level(k: integer): integer;
var j, kmax: integer;
begin
  j := 0; kmax := 0;
  while kmax < k do
  begin
    j := j + 1; kmax := 2*kmax + 1
  end;
  level := j
end
```

Algorithm 2

Algorithm 2 defines the level number of node k, where

$$0 \leq \text{level}(k) \leq d \quad \text{for } 0 \leq k \leq p - 1$$

We will program a hypercube node in Pascal extended with statements for message communication. The input and output of an array slice $a[i..j]$ through a channel c are denoted

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

The communications include the bounds i and j.

Each node is connected to its environment and nearest neighbors through a local array of channels

type channels = **array** [0..d] **of** channel

A particular local channel is used to transmit a problem of a given size. For a cube we have

channel number	problem size
0	n
1	$n/2$
2	$n/4$
3	$n/8$

In general, channel number i carries a problem of size $n/2^i$, where $0 \leq i \leq d$.

The channel through which a node inputs a sorting problem is called its *bottom* channel. For the cube we have

nodes	problem size	bottom channel
0..0	n	0
1..1	$n/2$	1
2..3	$n/4$	2
4..7	$n/8$	3

In general, the index of the bottom channel is equal to the level of the corresponding node.

Algorithm 3 defines the behavior of hypercube node k for $n \geq p$. This is a *balanced, parallel quicksort.* It maintains load balance by using the well-known *find* algorithm to split array slices in half [5]. However, since find takes twice as long as partition, we use it during the splitting phase only. For the sorting phase, we use the standard quicksort (Algorithm 1).

If we use partition instead of find in Algorithm 3, we get an *unbalanced, parallel quicksort.* Measurements show that such an algorithm is slower than the balanced sort, and rather unpredictable.

```
procedure node(k: integer; c: channels);
var bottom, first, last, middle, i: integer;
begin
  bottom := level(k);
  c[bottom]?(first, last, a[first..last]);
  for i := bottom + 1 to d do
  begin
    middle := (first + last) div 2;
    find(a, first, last, middle);
    c[i]!(first, middle, a[first..middle]);
    first := middle +1
  end;
  quicksort(a, first, last);
  for i := d downto bottom + 1 do
    c[i]?(first, middle, a[first..middle]);
  c[bottom]!(first, last, a[first..last])
end
```

Algorithm 3

COMPLEXITY

The *parallel run time* $T(p, n)$ is the average time required to sort n numbers on a hypercube with p processors, where n and p are powers of two, and $n \geq p$.

An initial node inputs n numbers and splits them into two halves. Later the same node combines the two sorted halves and outputs n sorted numbers. The node inputs, splits, combines, and outputs the n items in time $(b + c)n$, where b and c are system dependent constants for communication and balanced splitting.

The initial node, which belongs to, say, the left half of the hypercube, sends $n/2$ items to the right half of the hypercube (see Fig. 1). The two halves of the hypercube now run in parallel. Each half uses $p/2$ processors to sort $n/2$ numbers. So the parallel run time of the complete hypercube is

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

This recurrence has the solution

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

Eventually, each node inputs, sorts, and outputs n/p numbers in time

$$T(1, n/p) = (n/p)(a \log(n/p) + b)$$

where a and b are system dependent constants for unbalanced splitting and communication.

Using the abbreviation $T_p = T(p, n)$ we have

$$T_p = (n/p)(a \log(n/p) + b) + 2(b + c)(n - n/p) \tag{1}$$

The *sequential run time* T_1 is the average time it takes to sort n numbers on a single processor. For $p = 1$ Eq. (1) reduces to

$$T_1 = n(a \log(n) + b) \tag{2}$$

On a hypothetical hypercube of infinite size, the parallel run time of the split and combine phases is

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

The time

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

is a lower bound on the parallel run time T_p.

The parallel run time can now be expressed as

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

For a large hypercube T_p approaches $T_{\min}$.

The *speedup* $S_p = T_1/T_p$ defines how much faster the sorting algorithm runs on p processors compared to a single processor. The speedup cannot exceed $T_1/T_{\min}$, that is

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

If $a = b$, $c = 2a$, and $n = 2^{20}$, the maximum speedup $S_{\max} = 3.5$. For $p = 32$ the actual speedup $S_p = 3.3$ only.

PERFORMANCE

For the performance measurements we replaced Algorithm 1 by the iterative quicksort defined in [6]. We reprogrammed the parallel quicksort in occam and ran it on a Computing Surface with T800 transputers configured as a hypercube. The four channels of a transputer limits the hypercube to a maximum of 8 nodes.

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

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

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)
2	6.10 (6.08)	1.52 (1.51)
4	4.64 (4.65)	1.99 (1.98)
8	3.96 (3.99)	2.34 (2.31)

The performance limits are

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

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 20–36% slower and somewhat erratic.

Table II. Parallel,
Unbalanced Quicksort

p	T_p	S_p	ΔT_p
1	9.25	1.00	0%
2	8.31	1.11	36%
4	5.58	1.66	20%
8	5.24	1.77	32%

HYPERCUBES VERSUS TREE MACHINES

In [4] we analyzed parallel sorting on a Computing Surface configured as a binary tree machine. The only difference between the performance models of a hypercube and a tree machine is that for the tree machine the number of nodes p is replaced by the number of leaf nodes

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

This difference is easy to understand. On a hypercube every one of the p nodes sorts. On a tree machine, sorting is done by the q leaves only.

However, the performance limitations T_{min} and S_{max} are the same for a hypercube and a tree machine.

Since a tree machine with $2p - 1$ processors has p leaves, it sorts as fast as a hypercube with p processors. In other words

$$T_{\text{cube}}(p, n) = T_{\text{tree}}(2p - 1, n) \qquad (6)$$

This relationship is confirmed by the measurements reported here and in [4].

In the following we compare a hypercube with p processors and a tree machine with $p-1$ processors when both machines sort n numbers. The time difference between these machines is

$$\Delta T(p,n) = T_{\text{tree}}(p-1,n) - T_{\text{cube}}(p,n)$$

If we replace p by $p/2$ in Eq. (6) we get

$$\Delta T(p,n) = T_{\text{cube}}(p/2,n) - T_{\text{cube}}(p,n)$$

which can be rewritten as follows using Eq. (4)

$$\Delta T(p,n) = (T_1 - T_{\min} - an(\log(p) - 2))/p$$

For $p \geq 4$ the following inequality holds

$$\Delta T(p,n) \leq (T_1 - T_{\min})/p$$

Since $T_{\text{cube}}(p,n) \geq T_{\min}$, the relative time difference is bounded as follows

$$\begin{aligned} \Delta T(p,n)/T_{\text{cube}}(p,n) &\leq \Delta T(p,n)/T_{\min} \\ &\leq (T1/T_{\min} - 1)/p \end{aligned}$$

In short

$$\Delta T(p,n)/T_{\text{cube}}(p,n) \leq (S_{\max} - 1)/p$$

Table III compares parallel sorting on medium-sized hypercubes and tree machines for $a = b$, $c = 2a$, and $n = 2^{20}$, where $S_{\max} = 3.5$. A hypercube with 32–64 nodes is only 3–6% faster than a tree machine with 31–63 nodes.

Table III

p	$\Delta T(p,n)$
16	12%
32	6%
64	3%

CONCLUSION

We have developed a balanced, parallel quicksort for a hypercube and compared it with a similar algorithm for a binary tree machine. The performance of the hypercube quicksort was measured on a Computing Surface.

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 we found that a hypercube with 32 or more nodes sorts only marginally faster than a tree machine of the same size. The reason is simple. On a large tree machine, the sorting time of the leaf nodes is smaller than the data distribution time of the root nodes. So there is not much to be gained by reducing the sorting time further.

ACKNOWLEDGEMENT

The author thanks Jonathan Greenfield for helpful suggestions.

REFERENCES

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

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

3. 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.

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

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

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