Research Paper: Fast Parallel Sorting Algorithms

Summary

This research paper proposes a parallel bucket sort algorithm that sorts the elements in O(log n) time. Though it requires that space should be greater than the product of time and processers. The serial algorithms require minimum amount of time to solve a problem, this bound of minimum time increases the space required. However, in case of parallel sorting algorithms, the tradeoff is between time and number of processors i.e. a minimum time is required to solve a problem given a bounded number of processors. This time can be reduced if increase the number of processors for computation in parallel sorting algorithm. The mentioned research paper first proposes parallel sorting algorithms in which the tradeoff is between time, space and number of processors, and at the expense of space, both time and number of processors can be decreased. The model assumes that all processors are synchronized and have access to a common memory location. The number of processors are represented by n, the numbers to be sorted are {0, 1, 2, …m-1} and the duplicates (if there are any) shall be discarded in the second algorithm. Each bucket will have its own memory space, which will total m. There will be n areas, where n is the quantity of input integers to be sorted. The processors (pi) with ci = j will leave marks within each region, j, signifying their existence. After that, they will look for the existence of (the marks of) additional active processors using a binary-tree search method. The lower ranking (i.e., the one with smaller index) processor will continue to operate while the higher ranking one deactivates if two processes find each other's presence (such discoveries will turn out to be simultaneous). As there are n locations per area, each processor can mark its location i at area ci. When the kth iteration is complete, a mark will be present at each place whose final k bits are zeros and whose other (log n) - k bits match the corresponding bits of an address of a processor that is now operating in that region. Each such site will then be shown whether any of the 2 k processors were previously operating there. If any of the n processors that were initially active in that area, i.e. if any of the n numbers to be sorted was j, the area bucket number, were present after log n iterations, the first location in that area will be marked.

Following is the algorithm showing common variable in capital while local memory variables are in lower case:

```
Algorithm l -- parallel bucket sort

Input: 0 <= i <= n - l A[j,t]=O

        0 <= j <= m- 1 B[j]=O

                ci E {0, 1 ..... m - l}, not necessarily distinct

Output:O <= i <= n- 1 A[j,i]=O

        O <= j <= m - 1 B[j] = min is.t. ci = j, O if none such


Let ek -- 0 ... 0 10 ... O, all bits 0 except the kth from the right
```

```
for all i do
        Let i = x~og ,~ ... x2x~ be the binary representation of i
        x ~ i x is the location that pi is marking
        A[ci, x] <- 1
        flag <- 1
        for k <- 1 step 1 until log n do
                begin
                buddy <- x • ek
                count ~ A[ci, buddy]
                if xk = i AND count ~ 0 then
                flag <- 0
                if xk = 1 AND count = 0 AND
                flag = 1 then begin
                A [ci, x] <- 0      if buddy is not active then there
                x <- buddy                 is no problem in using his
                A [ci, x] <- 1      space
        end
end
if flag = 1 then B[ci] ~ i
A[c,, x] ,.- 0
```

The next algorithm adheres to the same fundamental framework that our prior approach established. Instead of a straightforward mark bit, we will instead maintain track of the number of processors that were initially active in each block of indices of size 2 k.

Algorithm 2.1 -- parallel bucket sort (part 1)

```
Input: 0 <= i <= n - 1 A[j,t]=0
        0 <= j <= m - 1 B[J]=0
                ci E {0, 1 ..... m - 1} not necessarily distinct
Local
```

```
Output:          r = max k s.t. ~:1 t in same 2k-block as i
                 with t. < i and ct = ci
                 y = head of 2^r-block containing i


Output:          0 <= i <= n - 1 A[j,t] = 0 exceptA[ci, y]=#of t >= is.t;
                         Ct = Ci
                 0 <= j <= m - 1 B[j] == min i s.t. ci = j, 0 if none such
for all i do
flag <- 1
r <- log n
x <- i
                                         x is the head of the largest block
y <- i


A [ci. y] ~-- 1                  that pi has counted
                                         y is the location at which pl is
                                         accumulating that count
                                         A[c,, y] will hold # of t s.t.
                                         t_> iand ct = ci


for k <- 1 step 1 until log n do
begin
buddy <- x @ eh
count <- A [cl. buddy]
if xk = 0 then A[ci. y] <- A[ci. y]
        + count
else (Xk = 1)
begin
        x <- buddy
```

```
if count != 0
then if flag = 1 then [flag~--O;
r*--k- 1]
else null
else (count=O)
if flag = 1 then
begin
Alc~. x] ~- A[c, y]
A[ci, y] *"0
y <- X
end (of then block)
end (of else x~ = 1)
end (of for loop)
B[ci] <- A[ci, y]
```

We have calculated the frequency with which each of the h numbers appears in the numbers to be sorted. Assuming duplicate numbers will be maintained, we will now add up the counts (for each number ci) of all numbers larger than ci to get the true ordering of the numbers. This accumulation will be carried out similarly to how it was done in the past.

```
Algorithm 2.2--parallel bucket sort (part 2)
Input: from Algorithm 2.1
Output: 0 <_ i _< n - 1 D[i] = sorted position of c~ only for the first
instance of each c~
larger values of cl will have smaller values of D = (CA of k
s.t. ck>ci) + 1
for all i do
o[0 ,- 0
if flag = 1 then
begin Let ci = Wlog .... w2w~ be the binary representation of c~
flag2 ~-- 1
W<---Ci
Z ~"'Ci
for k ,-- 1 step 1 until log m do
begin
buddy ~- w • ek
count ~ B[buddy]
if Wk = 0 then
BIzl ,-- BIz] + count
else (wk = 1)
begin w ~-- buddy
if count # 0 then
flag2 ,.- 0
else if flag 2= 1 then
begin
B[w] *-- B[z]
B[z] ,.- 0
g~--w
end (of then block)
end (of else Wk=I)
end (of for loop)
Dlt] ,-- BIz] - .4[c,, y] + l
A[c, y] ,-- B[z]
B[z] ,--- 0
end (of if flag= 1) Algorithm 2.2--parallel bucket sort (part 2)
```

Input: from Algorithm 2.1

Output: 0 <_ i _< n - 1 D[i] = sorted position of c~ only for the first

instance of each c~

larger values of cl will have smaller values of D = (CA of k

```
s.t. ck>ci) + 1
for all i do
o[0 ,- 0
if flag = 1 then
begin Let ci = Wlog .... w2w~ be the binary representation of c~
flag2 ~-- 1
W<---Ci
Z ~"'Ci
for k ,-- 1 step 1 until log m do
begin
buddy ~- w • ek
count ~ B[buddy]
if Wk = 0 then
BIzl ,-- BIz] + count
else (wk = 1)
begin w ~-- buddy
if count # 0 then
flag2 ,.- 0
else if flag 2= 1 then
begin
B[w] *-- B[z]
B[z] ,.- 0
g~--w
end (of then block)
end (of else Wk=I)
end (of for loop)
Dlt] ,-- BIz] - .4[c,, y] + l
A[c, y] ,-- B[z]
B[z] ,--- 0
```

end (of if flag= 1)

The D-values of duplicates may now be assessed using the reverse of Algorithm 2.1's approach if we require that no more than one processor may access a place at once, not even for fetches. This is carried out in Algorithm 2.3 below.

Algorithm 2.3--paraUel bucket sort (part 3)

Input: from Algorithm 2.2

Output: 0 <= i <= n - 1 A[j,i] = 0

0 <= j < = m - 1 D[i] = sorted pos of ci

=#ofks.t. ck>ci

+ !=of k <=is.t. Ck = ci

```
for all i o
for k *-- (log n) - 1 step - 1 until 0 do
if r = k then
begin get value ofA [ci, 0] from buddy
x ~ y ~9 e,. location
D[i] ~ Alci. x] -Alcl. y] + 1
A[ci, y] *'-" A[ci, x]
end
else if r > k then
begin
x ~--y OR (/AND ek)
if x ~ y then begin
A[ci, x] *-- A[ci, y]
.4[c, yl ,---0
end (of then block)
end (of if r>k)
(end of for loop)
alc, y] ~-0
```

Note that Algorithm 2 (sequence of Algorithms 2.1, 2.2, 2.3) requires space S = O(mn), time and T = O(log n + log m), using n processors.

Here we present an algorithm to permute any n number of numbers in time O(log n). They are based on an extension of the algorithm by Gavril [8] that merges two of his linear ordered sets in O(log n) time. Algorithm 3, the first algorithm to do this, requires the use of n 3/2 processors.

Algorithm 3--paraUel sort using n a/2 processors

Input: 0 <_ i _< n - 1 ci E integers

Output: {ci} will be stably sorted, smallest first

1. Partition the n input numbers into n ~/2 groups, each having n ~/2

elements.

2. Within each group do

For each element, j, determine count[j] = (# of i such that c,<cj) +

(# of i<_j such that ci=cj). This can be done in time O(log n) using

n ~/2 processors per element (a total of n processors per group or n 3/2

processors in toto). The n ~/2 processors for element j will be assigned,

one to each element i in j's group, to compare c~ with cj. Summing

the results of these comparisons can be done in time O(log n).

3. Within each group do

Bucket sort the elements, using count[j] as the key for the jth

element in the group. This is done by: Ccoun~jl ",-- ci, where j and

count[j] are offsets (of value at most n ~/2) from the beginning of

each group. There will be no memory conflicts since the count[j]'s

within a group are all distinct. Steps 2 and 3 have effectively sorted

the elements within each group using an "Enumeration Sort" [1 I].

4. All elements do a binary search of the n ~/2 groups. That is, each

element(cj) in group g, has n ~/2 processors which are assigned, one to

each group, to do a binary search on the elements in a group (which

are sorted) so as to determine, for all groups k, the value of

count[j, k] = if k < g, # of elements i such that cl <-- cj

ilk = g,j

if k > g, # of elements i such that c~ < c i

where ci refers to the ith element in group k and cj is fLxed.

5. For all elements, j, evaluate count[j] = sum (over k) of count-

[j, k]. This can be done in time O(log n) and requires n ~/2 processors

per element for a total of n 3/2 processors.

6. Do a bucket sort on all n elements using count[j] as the key for the

jth element. Again, there will be no memory conflicts since count[j]

will be the rank of thejth element.

7. END of Algorithm 3.

We can see that Algorithm 3 takes O(log n) time and uses n 3/2 processors. Here is a simple modification of Algorithm 3 that uses the same time on the order of and requires only n 4/3 processors:

Algorithm 4---parallel sort using n 4/3 processors

1. Partition the n input numbers into n 2/3 groups each having n ~/3

elements.

2. Within each group do

For each element, j, determine count[j] = # of i such that c~ < cj)

+ (# of i _< j such that cl = ci).

3. Within each group do

Bucket sort the count[j]'s obtained in step 2. This will rearrange the

elements in rank order within each group.

4. Divide the n 2/3 groups into n '/3 sectors, each sector consisting ofn '/3

groups.

5. Within each sector do

For each element (j) in group g, do a binary search of each of the

n '/a groups inj's sector to determine, for all k, the value of

count[j, k] = if k < g, # of i in group k such that ci <- c i

ilk = g,j

if k > g, # of i in group k such that ci < cj.

Then, for each element j, evaluate count[j] = (# of i in j's sector

such that ci < cj) + (# of i <_ j in j's sector such that ci = cy). This

number is simply the sum (over k) of count[j, k].

6. Within each sector, do a bucket sort of the elements within the

sector using count[j] as they key for element./'. This will rearrange the

elements in rank order within each sector.

7. For all elements (j) in sector t, do a binary search of each of the n '/3

sectors to determine, for all k, the value of

count[j, k] = ilk < t, # ofi in sector k such that cz <- c i

ilk = t,j

if k > t, # of i in sector k such that ci < cj.

Then evaluate count[j] = the sum (over k) of count[j, k].

8. Do a bucket sort of all n elements.

9. END of Algorithm 4.