106117003(Abhishek)

106117013(Ashutosh)

Problem:

To perform sorting on a CRCW and EREW computer model

Assumptions:

The number of processor cores available is less than or equal to 8

Algorithms:

PARALLEL ALGORITHMS:

procedure EREW SORT (S)

if |Sj| < k

then QUICKSORT (S)

else (1) for i = 1 to k - 1 do

PARALLEL SELECT (S, i|Si|/k) {Obtain mi}

end for

(2) S1 <- {sεS:s <= m1}

(3) for i = 2 to k - 1 do

S <- {sεS : mi-1 <= s <= mi}

end for

(4) Sk <- {sεS : s >= mk-1}

(5) for i = 1 to k/2 do in parallel

EREW SORT (Si)

end for

(6) for i = (k/2) + 1 to k do in parallel

EREW SORT (Si)

end for

```
end if.

procedure CRCW SORT(S)
Step 1: for i = 1 to n do in parallel
          for j = 1 to n do in parallel
            if (si > sj) or (si = sj and i > j)
              then P(i, j) writes 1 in c,
            else
              P(i, j) writes 0 in c,
            end if
          end for
        end for.
Step 2: for i = 1 to n do in parallel
          P(i, 1) stores si in position 1 + ci of S
        end for.

procedure PARALLEL SELECT (S, k)
Step 1:
if |S| <= 4
  then PI uses at most five comparisons to return the kth element
else
  (i) S is subdivided into |S|1-x subsequences Si of length |S|^x each, where
  1 <= i >= |S|1-x, and
  (ii) subsequence Si is assigned to processor Pi.
end if.
Step 2: for i = 1 to |S|^1-x do in parallel
          (2.1) {Pi obtains the median mi, i.e., the fISil/2lth element, of its associated
          subsequence}
          SEQUENTIAL SELECT (Si, [|S|/2])
```

(2.2) Pi stores mi in M(i)

end for.

Step 3: {The procedure is called recursively to obtain the median m of M}

PARALLEL SELECT (M, [|M|/2]).

Step 4:

The sequence S is subdivided into three subsequences:

L={siεS: si<m},

E={siεS: si =m}, and

G={siεS: si>m}.

Step 5: if|L|>=k then

PARALLELSELECT(L, k)

else if |L|+|E| >= k then return m

else PARALLEL SELECT (G, k - |L| - |E|)

end if

Endif.

SEQUENTIAL ALGORITHM:

(All the parallel flows are removed)

procedure EREW SORT (S)

if |Sj| < k

then QUICKSORT (S)

else (1) for i = 1 to k - 1 do

SEQUENTIAL SELECT (S, i|Si|/k) {Obtain mi}

end for

(2) S1 <- {sεS:s <= m1}

(3) for i = 2 to k - 1 do

S <- {sεS : mi-1 <= s <= mi}

end for

```
(4) Sk <- {sεS : s >= mk-1}
(5) for i = 1 to k/2
      EREW SORT (Si)
    end for
(6) for i = (k/2) + 1 to k
      EREW SORT (Si)
    end for
end if.
```

```
procedure CRCW SORT(S)
Step 1: for i = 1 to n
          for j = 1 to n
            if (si > sj) or (si = sj and i > j)
              then P(i, j) writes 1 in c,
            else
              P(i, j) writes 0 in c,
            end if
          end for
        end for.
Step 2: for i = 1 to n
          P(i, 1) stores si in position 1 + ci of S
        end for.
```

```
procedure SEQUENTIAL SELECT (S, k)
Step 1: if |S| < Q then sort S and return the kth element directly
        else subdivide S into |S|/Q subsequences of Q elements each (with up to Q-1 leftover elements)
        end if.
Step 2: Sort each subsequence and determine its median.
Step 3: Call SEQUENTIAL SELECT recursively to find m, the median of the ISI/Q
```

medians found in step 2.

Step 4: Create three subsequences S1, S2 and S3 of elements of S smaller than, equal to, and larger than m, respectively.

Step 5: if |S1|>= k then {the kth element of S must be in S1}

call SEQUENTIAL SELECT recursively to find the kth element of S1

else if |S1| + |S| >= k then return m

else call SEQUENTIAL SELECT recursively to find the (k-|S1|- |S2|)th element of S3

end if

end if.

CODE:

| crcw.py |

```
import multiprocessing as mp

from joblib import Parallel, delayed

import seqentializer as sq

import time

import math

import numpy as np


print("No. of Processors : ", mp.cpu_count())

# declare  n and array global ..

n = 10

array = np.random.randint(1, 10**6, n)

ci = np.zeros(n, dtype=int)


# sequential function ..
```

```
def sequential():
    c = np.zeros(n, dtype=int)
    sq.sequilizer(0)

    # step 1 start..
    for i in range(n):
        for j in range(n):
            if (array[i] > array[j] or (array[i] == array[j] and i > j)):
                c[i] = c[i] + 1
            else:
                c[i] = c[i] + 0

    print(array)
    print(c)
    # step 1 end ..

    # step 2 start ..
    final_array = np.zeros(n, dtype=int)
    for i in range(n):
        final_array[c[i]] = array[i]

    print(final_array)

# parallel function ..

def fun(i, j):
    if (array[i] > array[j] or (array[i] == array[j] and i > j)):
        ci[i] = ci[i] + 1
    else:
```

```
        ci[i] = ci[i] + 0


def givevalue(i, final_array):
    final_array[ci[i]] = array[i]


def parallel(array, n):
    # step 1 start
    Parallel(n_jobs=-1, require='sharedmem')(delayed(fun)(i, j)
                                        for i in range(n)
                                        for j in range(n))
    # step 1 end ..


    # step 2 start ..
    final_array = np.zeros(n, dtype=int)
    Parallel(n_jobs=-1, require='sharedmem')(delayed(givevalue)(i, final_array)
                                        for i in range(n))
    #step 2 finished ..
    print(ci)
    print(final_array)
if __name__ == '__main__':


    seq_start = time.time()
    sequential()
    seq_req = time.time() - seq_start
    print(seq_req)
    parallel_start = time.time()
    parallel(array, n)
    parallel_req = time.time() - parallel_start
    print(parallel_req)
```

```
    print(parallel_req / seq_req)
    print(seq_req / parallel_req)
```

| erew.py |

```
import multiprocessing as mp
from joblib import Parallel, delayed
import seqentializer as sq
import time
import math
import numpy as np
from parallelselection import make_selection_parallel
from matplotlib import pyplot as plt

N = mp.cpu_count()
n = 160
x = math.log(n / N, n)
k = int(pow(2, math.ceil(1 / x)))
print("No. of cpu :",N)
print("No of element in arr : ",n)
print("Value of x : ",x)
print("Value of k :",k)

# Quick sort code .......................................................
def partition(arr, low, high):
    i = (low - 1)  # index of smaller element
    pivot = arr[high]  # pivot
```

```
    for j in range(low, high):

        if arr[j] <= pivot:
            i = i + 1
            arr[i], arr[j] = arr[j], arr[i]

    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return (i + 1)

def quickSort(arr, low, high):
    if low < high:
        pi = partition(arr, low, high)
        quickSort(arr, low, pi - 1)
        quickSort(arr, pi + 1, high)

# Quick sort code finished ....................................................

def sequential_select(arr,pos):
    temp = arr.copy()
    temp.sort()
    return temp[pos]

def erewseqential(array):
    if len(array)<=k:
        quickSort(array,0,len(array)-1)
    else :
        # step - 1 start
        m = list()
        for i in range(1,k):
```

```
    m.append(sequential_select(array,i*int((math.ceil(len(array)/k)))-1))
print(m)

# Step 2 start ..
new_array = list()
for i in range(k):
    new_array.append(0)
another_array = list()
for i in array :
    if i <=m[0]:
        another_array.append(i)
new_array[0] = another_array

# Step - 3 start
for i in range(1,k-1):
    another_array = []
    for j in array :
        if j > m[i-1] and j<=m[i] :
            another_array.append(j)
    new_array[i] = another_array

# Step  - 4 start
another_array = []
for i in array :
    if i > m[k-2] :
        another_array.append(i)
new_array[k-1] = another_array
print(new_array)
# Step - 5 start
```

```
        for i in range(int(k/2)):
            erewseqential(new_array[i])

        for i in range(int(k/2),k):
            erewseqential(new_array[i])

        temp = 0
        for i in range(len(new_array)):
            for j in range(len(new_array[i])):
                array[temp] = new_array[i][j]
                temp+=1
def erewparallel(array):
    if len(array)<=k:
        quickSort(array,0,len(array)-1)
    else :
        # Step 1 start ..
        m = list()
        for i in range (1 , k):
            m.append(make_selection_parallel(array,i*(math.ceil(len(array)/k))- 1, x))
        print(m)

        # Step 2 start ..

        new_array = list()
        for i in range(k):
            new_array.append(0)
        another_array = list()
        for i in array :
            if i <=m[0]:
```

```
            another_array.append(i)
        new_array[0] = another_array

        # Step - 3 start
        for i in range(1,k-1):
            another_array = []
            for j in array :
                if j > m[i-1] and j<=m[i] :
                    another_array.append(j)
            new_array[i] = another_array

        # Step  - 4 start
        another_array = []
        for i in array :
            if i > m[k-2] :
                another_array.append(i)
        new_array[k-1] = another_array
        print(new_array)

        # Step - 5 start
        Parallel(n_jobs= -1 , require='sharedmem')(delayed(erewparallel)(new_array[i]) for i in
range(int(k/2)))
         Parallel(n_jobs= -1 , require='sharedmem')(delayed(erewparallel)(new_array[i]) for i in
range(int(k/2),k))
       temp = 0
        for i in range(len(new_array)):
            for j in range(len(new_array[i])):
                array[temp] = new_array[i][j]
                temp+=1
```

```
if __name__=='__main__':
    arr = np.random.randint(1, 10**6, n)
    array = arr.copy()
    print("Initial Array : ",array)

    sequential_start = time.time()
    sq.sequilizer(1)
    erewseqential(array)
    sequential_end = time.time() - sequential_start
    print(sequential_end)
    print(array)

    array = arr.copy()
    parallel_start = time.time()
    erewparallel(array)
    parallel_end = time.time() - parallel_start
    print(parallel_end)
    print(array)
    plt.bar([2],[sequential_end*10],label="Seqential_EREW",color='g',width=1)
    plt.bar([4],[parallel_end*10],label="Parallel_EREW", color='r',width=.5)
    plt.legend()
    plt.xlabel('Diff. Algo.')
    plt.ylabel('Time(in 10*sec)')
    plt.title('EREW Implementation')
    plt.show()
    speed_up = sequential_end / parallel_end
    print("speed_up : ", speed_up)
```

| ANALYSIS |

CRCW SORTING:

t(n) = 0(1)

The sequential algorithm takes O(n^2).

p(n) = n^2 , the cost of procedure CRCW SORT is

c(n) = 0(n^2), WHICH IS NOT OPTIMAL.

EREW SORTING:

t(n) = cn^x + 2*t(n/k)

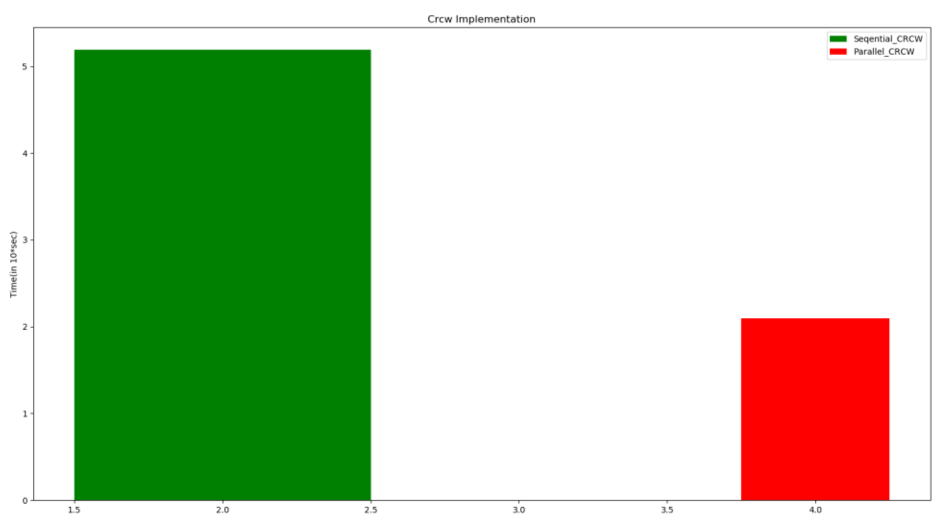
= O(n^x*log n).

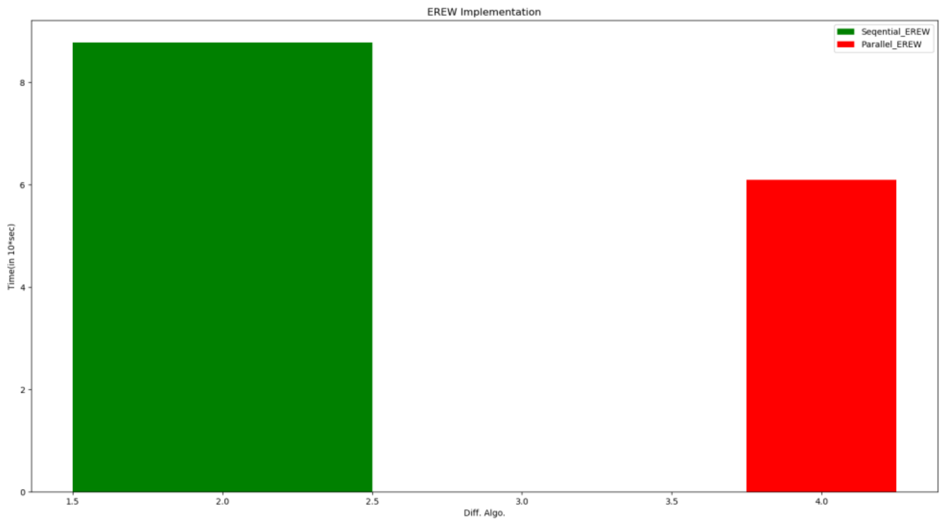
The sequential algorithm takes O(nlogn)

p(n) = n^1-x, the procedure's cost is given by c(n)

= p(n) x t(n) = O(n log n)

| RESULTS AND COMPARISON |





As we see in the above graphs:

Speedup in CRCW(approximately) = 3.75 [n = 10]

Speedup in EREW(approximately) = 2.15 [n = 16]

Conclusion:

Thus, sorting in CRCW and EREW models successfully implemented and performance metrics compared.