Assignment No B5
Roll NO:

- **Title:**
  Implement n-array search algorithm using OPENMP

- **Problem Definition:**
  Implement n-array search algorithm using OPENMP

- **Learning Objective:**
  1. To understand n-array search algorithm
  2. To understand implementation OPENMP

- **Learning Outcome:**
  Successful implement n-array search using openmp.

- **Software Requirement:**
  1. Free open source
  2. gcc/g++

- **Hardware Requirement:**
  1.64bit processor.
  2. Multicore processor

- **Theory:**
  N array algorithm is extension of Binary Search algorithm. Searching a list for a particular item is a common task. In real applications, the list items often are records (e.g. Student records), and the list is implemented as an array of objects. The goal is to find a particular record, identified by name or an ID number such as a student number. Finding the matching list element provides access to target information in the record - the student's address, for example. The following discussion of search algorithms adopts a simpler model of the search problem - the lists are just arrays of integers. Clearly, the search techniques could generalize to more realistic data.

  The concept of efficiency (or complexity) is important when com- paring algorithms. For long lists and tasks, like searching, that are repeated frequently, the choice among alternative algorithms becomes important because they may differ in efficiency. To illustrate the concept of algorithm efficiency (or complexity), we consider two common algorithms for searching lists: **linear search** and **binary search**.

1. Introduction:

In this assignment we are implementing n-array search by using binary search.

2. Related Concepts:

- Linear Search:

What is the simplest method for searching a list of elements? Check the first item, then the second item, and so on until you find the target item or reach the end of the list. That’s linear (or sequential) search. Assume an array called list, filled with positive integers. The task is to search for the location of the number that is the value of a variable target.

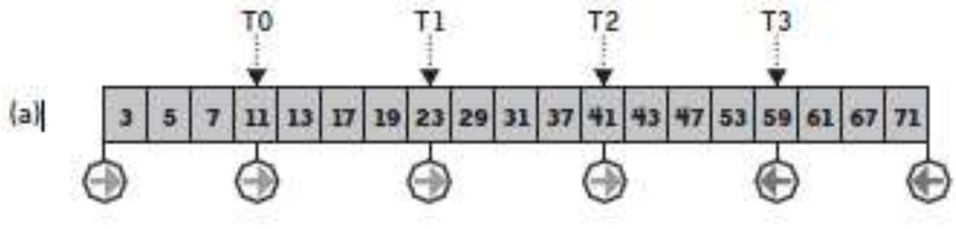
- Binary search:

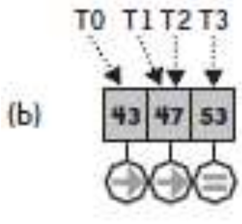
The strategy of binary search is to check the middle (approximately) item in the list. If it is not the target and the target is smaller than the middle item, the target must be in the first half of the list. If the target is larger than the middle item, the target must be in the last half of the list. Thus, one unsuccessful comparison reduces the number of items left to check by half! The search continues by checking the middle item in the remaining half of the list. If it’s not the target, the search narrows to the half of the remaining part of the list that the target could be in. The splitting process continues until the target is located or the remaining list consists of only one item. If that item is not the target, then it’s not in the list. Here are the main steps of the binary search algorithm, expressed in pseudo-code (a mixture of English and programming language syntax).

- n-ary search:

If we have N threads available, we can develop a concurrent N-ary search. This search identifies N well-spaced points within the search array bounds and compares the key of the corresponding records to the search key. Each thread does one of the N comparisons. There are three possible outcomes from these comparisons. The first is that the item of interest is found and the search is complete; the second is that the item key examined is less than the search key; the third is that the item key examined is greater than the search key.

If no search key match is found, a new, smaller search array is defined by the two consecutive index points whose record keys were found to be less than the search key and greater than the search key. The N-ary search is then performed on this refined search array. As with the serial version of binary search, the process is repeated until a match is found or the number of items in the search array is zero. A pictorial description of this algorithm is shown in Figure:





Given the sorted array of prime numbers in Figure(a), let’s say we want to determine whether the value 53 is in the array and where it can be found. If there are four threads (T0 to T3), each computes an index into the array and compares the key value found there to the search key. Threads T0, T1, and T2 all find that the key value at the examined position is less than the search key value. Thus, an item with the matching key value must lie somewhere to the right of each of these thread’s current search positions (indicated by the circled arrows).

Thread T3 determines that the examined key value is greater than the search key, and the matching key can be found to the left of this thread’s search position (left-pointing circled arrow). Notice the circled arrows at each end of the array. These are attached to “phantom” elements just outside the array bounds. The results of the individual key tests define the subarray that is to become the new search array.

Where we find two consecutive test results with opposite out- comes, the corresponding indexes will be just outside of the lower and upper bounds of the new search array.Figure(a) shows that the test results from threads T2 (less than search key) and T3 (greater than search key) are opposite. The new search array is the array elements between the elements tested by these two threads. Figure(b) shows this sub array and the index positions that are tested by each thread. The figure shows that during this second test of element key values, thread T3 has found the element that matches the search key (equals sign in circle). Consider the case where we want to find a composite value, like 52, in a list of prime numbers.

The individual key results by the four threads shown in Figure

(a) would be the same. The sub array shown in Figure

(b) would have the same results, except that the test by T3 would find that the key value in the assigned position was greater than the search key (and the equals sign would be a left-pointing arrow). The next round of key comparisons by threads would be from a sub array with no elements, bounded by the array slots holding the key values of 47 and 53. When threads are confronted with the search of an empty search space, they know that the key is not to be found. This is obviously more complex than a simple binary search. The algorithm must coordinate the choices of index positions that each thread needs to test, keep and store the results of each test such that multiple threads can examine those results, and compute the new search array bounds for the next round of key tests. From this quick description, it's clear that we'll need some globally accessible data and, more importantly, we need a barrier between the completion of the key tests and the examination of the results of those –o88itests.

3. Concept Of Programming Language Used:

OpenMp (Open Multi-Processing) is an API that supports multi-platform shared memory multiprocessing programming. It is a Directive based approach with library support. Targets existing applications and widely used languages like Fortran API, C,C++ API. OpenMP API consists of:

1. Compiler Directives and Programs which contains

1.1) Control structures

```
#pragma omp parallel [clause ...]  newline
      if (scalar_expression)
      private (list)
      shared (list)
      default (shared | none)
      firstprivate (list)
      reduction (operator: list)
      copyin (list)
      num_threads (integer-expression)
structured_block
```

1.2) Work sharing

a. For loop :

```
#pragma omp for[clause
clause[]. . . ]
  for loop
```

b. DO Loop:

```
!ompdo[clauseclause[]. . . ]
doloop
```

```
[!omp end do[nowait]]
```

c. SECTIONS Directive

```
#pragma omp sections [clause ...]  newline
    private (list)
   firstprivate (list)
    lastprivate (list)
 reduction (operator: list)
    nowait
 {
 #pragma omp section   newline
  structured_block
  #pragma omp section   newline
 structured_block
 }
```

d. SINGLE Directive.

```
#pragma omp single [clause ...]  newline
private (list)
firstprivate (list)
 nowait
   structured_block
```

1.3) Synchronization

a. MASTER Directive:

```
#pragma omp master newline
 structured block
```

b. CRITICAL Directive:

```
 #pragma omp critical [ name ] newline
 structured block
```

c. BARRIER Directive:

```
 #pragma omp barrier newline
```

TASKWAIT DIRECTIVE:

```
  #pragma omp taskwait newline
```

ATOMIC DIRECTIVE

```
#pragma omp atomic newline
   statement expression
```

FLUSH DIRECTIVE

```
#pragma omp flush (list) newline
```

ORDERED DIRECTIVE

```
#pragma omp for ordered [clauses...]
(loop region)
      #pragma  omp  ordered  newline
       structured  block
  (endo of loop region)
```

1.4) Data scope attributes:

i. Private:
   private (list)

ii. Firstprivate:
   #pragma omp parallel firstprivate(list)

iii. Lastprivate
   #pragma omp parallel last-private(list)

iv. Shared
   shared (list)

v. Reduction
   reduction (operator: list)

2.)Runtime subroutines/functions

2.1)Control and query routines:

1. number of threads
   void omp_set_num_threads(n);

2. nested parallelism
   void omp_set_num_threads(n);

2.2)Lock API

1. Initialize lock
void omp_init_lock(omp_lock_t *lock);
void omp_init_nest_lock(omp_nest_lock_t *lock);

2. Destroy lock
void omp_destroy_lock(omp_lock_t *lock);
void omp_destroy_nest_lock(omp_nest_lock_t *lock);

3. Set lock
void omp_set_lock(omp_lock_t *lock);
void omp_set_nest_lock(omp_nest_lock_t *lock);

4. Unset lock
void omp_unset_lock(omp_lock_t *lock);
void omp_unset_nest_lock(omp_nest_lock_t *lock);

5. Test lock
int omp_test_lock(omp_lock_t *lock);
int omp_test_nest_lock(omp_nest_lock_t *lock)

3.)Environment variables:

3.1). schedule type

```
setenv OMP SCHEDULE ”guided, 4”
setenv OMP SCHEDULE ”dynamic”
```

3.2)max threads

EXAMPLE:

```
setenv OMP NUM THREADS 8
```

3.3)nested parallelism EXAMPLE:

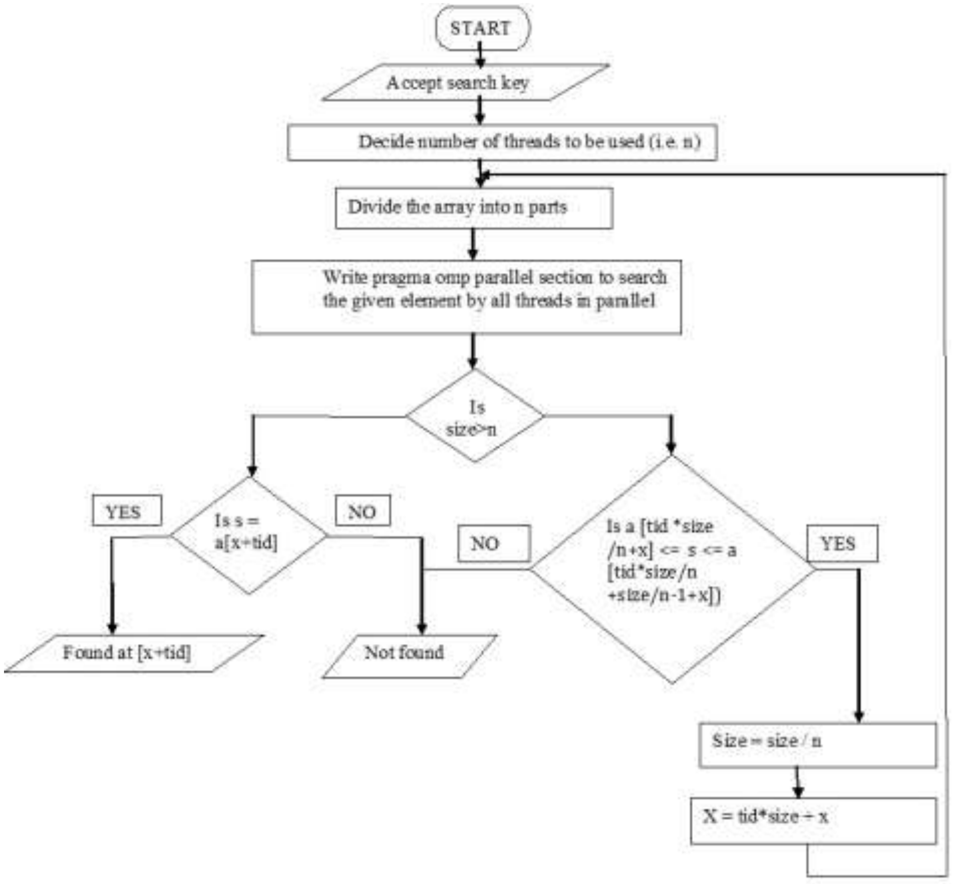
```
setenv OMP NESTED TRUE
```

4. Algorithm:

· n-ary:

```
1. Accept search key from user
2. decide number of threads to be used (i.e. n)
3. divide the array into n parts
4. write pragma omp parallel section to search the given
element by all threads in parallel
5. if size of array ¿ n then check whether search key lies
between a[tid*size/n+x] and a[tid*size/n+size/n-1+x]
5.1 set size = size/n
5.2 set x=tid*size+x
5.3 goto step 3
6. if size of array ¡= n then check whether search key=a[x+tid]
6.1. print ’element found at [x+tid]
6.2. else print ’not found’
```

5. Flowchart:


START
Accept search key
Decide number of threads to be used (i.e. n)
Divide the array into n parts
Write pragma omp parallel section to search the given element by all threads in parallel
Is size>n
Is s = a[x+tid]
YES
NO
Found at [x+tid]
Not found
Is a [tid *size /n+x] <= s <= a [tid*size/n +size/n-1+x])
NO
YES
Size = size / n
X = tid*size + x

6. Mathematical Model:

In class info and info1,
Let U ={D,ND,S,F,f,In,Out}
where
U: Universal set
D: Deterministic data values i.e. is initial values
ND: Non deterministic data values i.e. values given by user.
S: Cases for success
F: Cases for failure
f: nary search(int,int);
In: array,number of threads,search element
Out: found,position,not found.

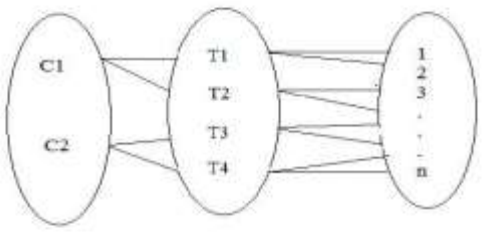


Figure 1: Venn Diagram

7. Specification requirement specification:

Specification	Description
Project scope	To understand pragma openmp to find a number in array using nary search
Functional requirements	Number is compared with lower and upper bound of each part of array by every thread
Design and implementation	Whole array should be divided among number of threads and this needs to be done 4 elements remain in array
Machine specification	32/64 bit CPU, 4GB RAM

8. UML:

- Use Case Diagram:

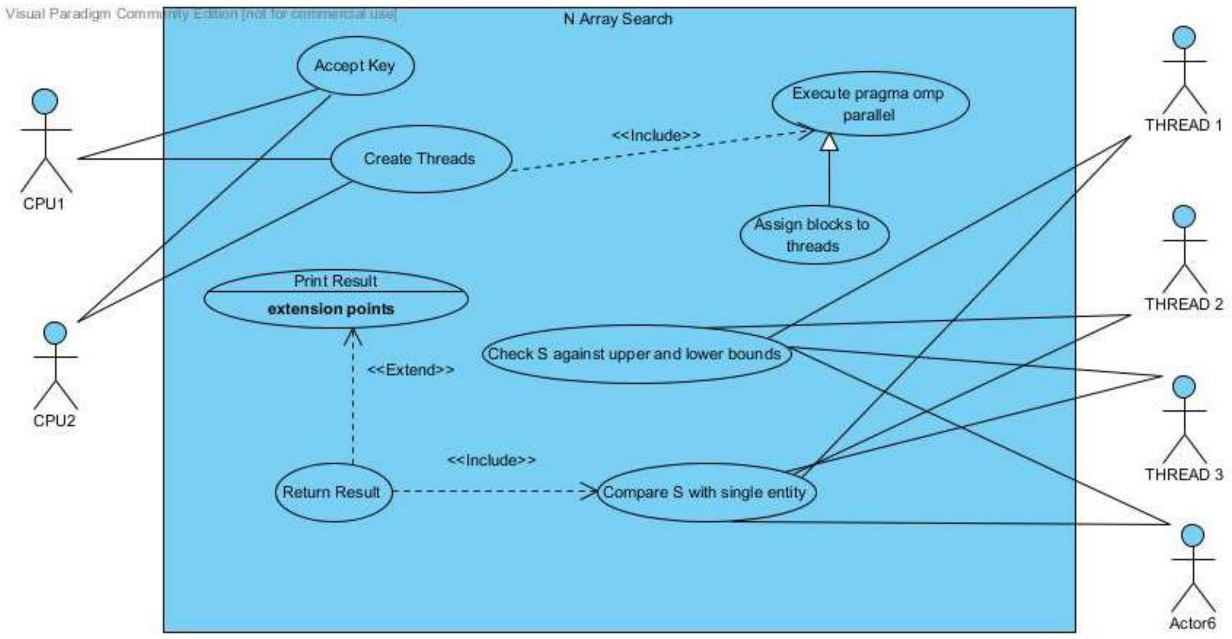


Figure 2: Use Case Diagram

- Activity Diagram:

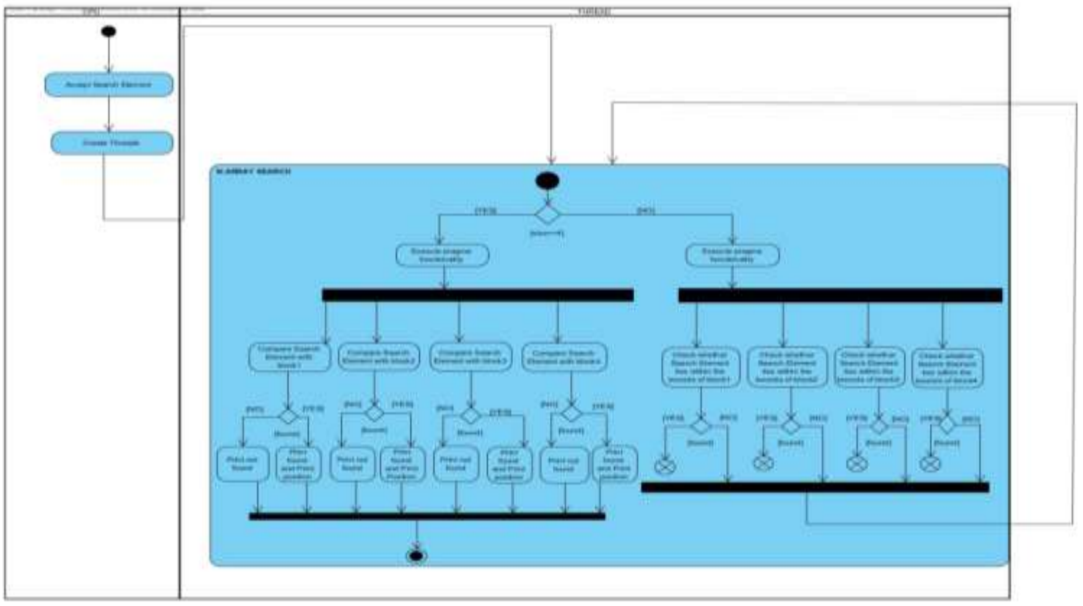
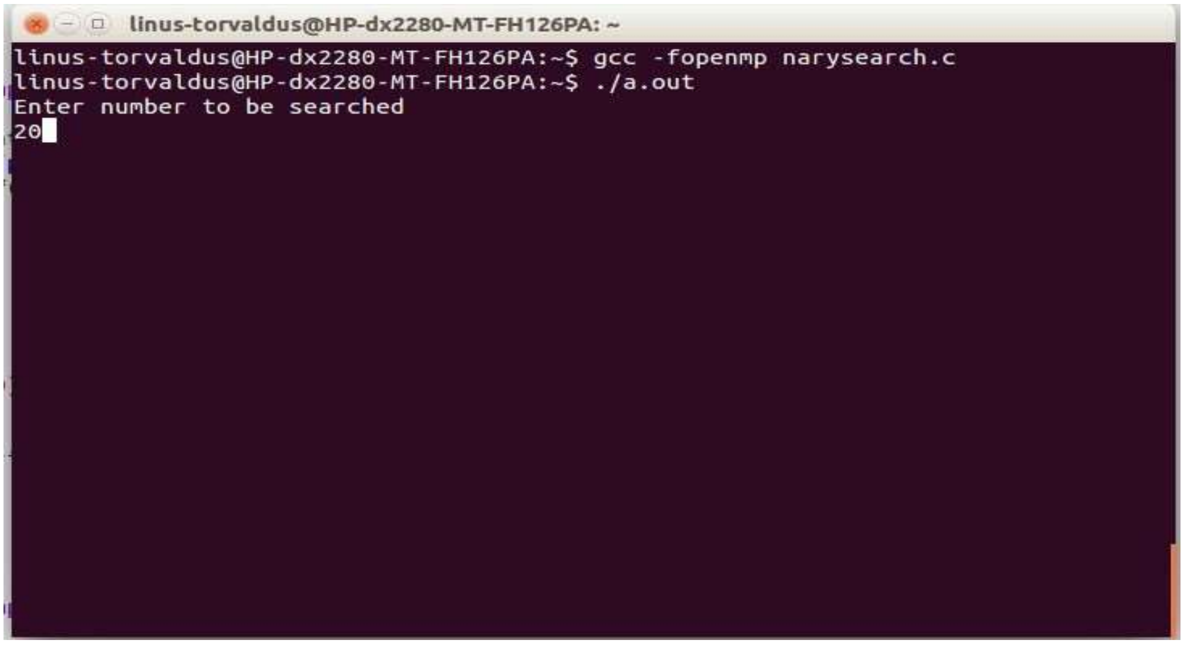


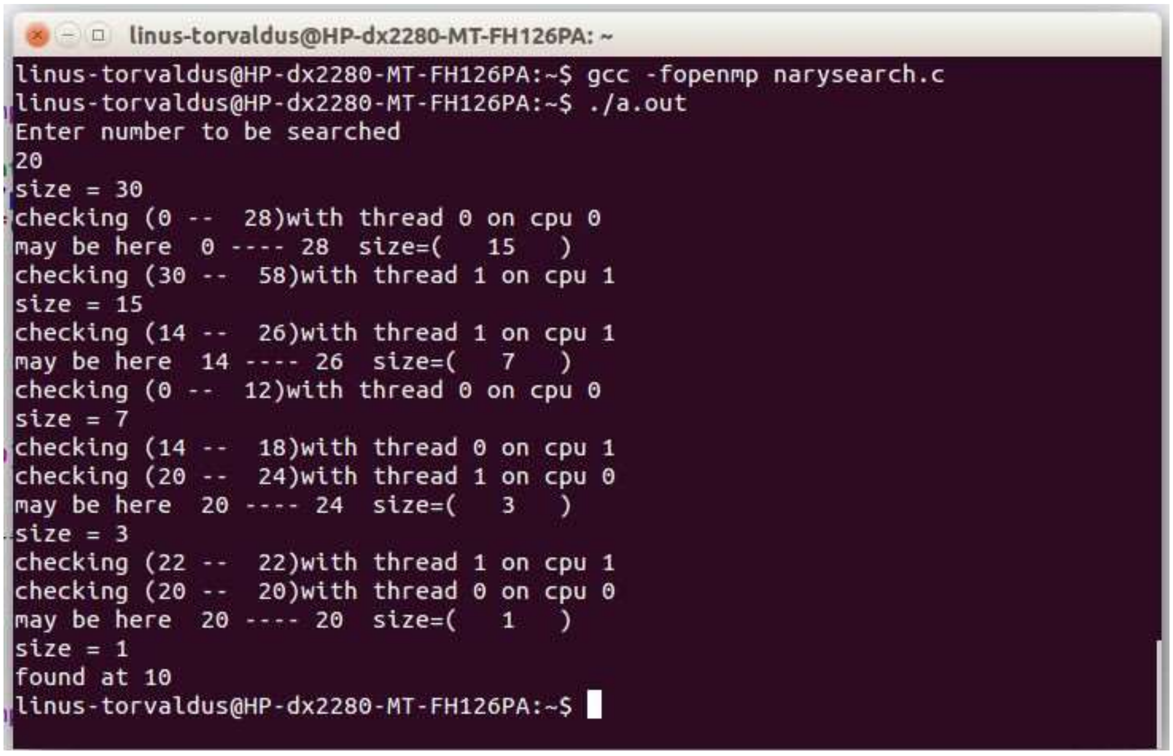
Figure 3: Activity diagram

- Input: { Enter Number to be searched }



```
linus-torvaldus@HP-dx2280-MT-FH126PA:~$ gcc -fopenmp narysearch.c
linus-torvaldus@HP-dx2280-MT-FH126PA:~$ ./a.out
Enter number to be searched
20
```

- Output: {size,checking with thread number on cpu number,found,not found }



```
linus-torvaldus@HP-dx2280-MT-FH126PA:~$ gcc -fopenmp narysearch.c
linus-torvaldus@HP-dx2280-MT-FH126PA:~$ ./a.out
Enter number to be searched
20
size = 30
checking (0 --  28)with thread 0 on cpu 0
may be here  0 ---- 28  size=(   15   )
checking (30 --  58)with thread 1 on cpu 1
size = 15
checking (14 --  26)with thread 1 on cpu 1
may be here  14 ---- 26  size=(   7   )
checking (0 --  12)with thread 0 on cpu 0
size = 7
checking (14 --  18)with thread 0 on cpu 1
checking (20 --  24)with thread 1 on cpu 0
may be here  20 ---- 24  size=(   3   )
size = 3
checking (22 --  22)with thread 1 on cpu 1
checking (20 --  20)with thread 0 on cpu 0
may be here  20 ---- 20  size=(   1   )
size = 1
found at 10
linus-torvaldus@HP-dx2280-MT-FH126PA:~$
```

- Conclusion:

Thus we have studied and implemented nary search through binary search and using multithreading.

- Course Outcome:

Course Outcomes	Achieved Outcome
CO I : Ability to perform multi-core, Concurrent and Distributed Programming.	
CO II : Ability to perform Embedded Operating Systems Programming using Beaglebone	
CO III :Ability to write Software Engineering Document.	
CO IV :Ability to perform Concurrent Programming using GPU.	

Table 1: Course Outcomes

- FAQs:

i. Why is sched.h header used?
ii. What is openmp?
iii. Which header file is included to use openmp?
iv. What is pragma omp parallel?
v. What is narray search?
vi. Why is omp get thread num() used?
vii. How to find total number of thread?
viii. How many threads are created in pragma omp?
ix. What does function nary search() do?
x. What does sched getcpu() do?