Title : Write a Parallel program (using OpenMP) for Binary Search in Sorted Array

Theory:

1)OpenMP:

OpenMP (Open Multi-Processing) is an [application programming interface](https://en.wikipedia.org/wiki/Application_programming_interface) (API) that supports multi-platform [shared memory](https://en.wikipedia.org/wiki/Shared_memory_architecture) [multiprocessing](https://en.wikipedia.org/wiki/Multiprocessing) programming in [C](https://en.wikipedia.org/wiki/C_(programming_language)), [C++](https://en.wikipedia.org/wiki/C++), and [Fortran](https://en.wikipedia.org/wiki/Fortran), on most platforms, [instruction set architectures](https://en.wikipedia.org/wiki/Instruction_set_architecture) and [operating systems](https://en.wikipedia.org/wiki/Operating_system), including [Solaris](https://en.wikipedia.org/wiki/Solaris_(operating_system)), [AIX](https://en.wikipedia.org/wiki/IBM_AIX), [HP-UX](https://en.wikipedia.org/wiki/HP-UX), [Linux](https://en.wikipedia.org/wiki/Linux), [macOS](https://en.wikipedia.org/wiki/MacOS), and [Windows](https://en.wikipedia.org/wiki/Microsoft_Windows). It consists of a set of [compiler directives](https://en.wikipedia.org/wiki/Compiler_directive), [library routines](https://en.wikipedia.org/wiki/Library_(computing)), and [environment variables](https://en.wikipedia.org/wiki/Environment_variable) that influence run-time behavior.

OpenMP is managed by the [nonprofit](https://en.wikipedia.org/wiki/Nonprofit_organization) technology [consortium](https://en.wikipedia.org/wiki/Consortium) OpenMP Architecture Review Board (or OpenMP ARB), jointly defined by a group of major computer hardware and software vendors, including [AMD](https://en.wikipedia.org/wiki/AMD), [IBM](https://en.wikipedia.org/wiki/IBM), [Intel](https://en.wikipedia.org/wiki/Intel), [Cray](https://en.wikipedia.org/wiki/Cray), [HP](https://en.wikipedia.org/wiki/Hewlett-Packard), [Fujitsu](https://en.wikipedia.org/wiki/Fujitsu), [Nvidia](https://en.wikipedia.org/wiki/Nvidia), [NEC](https://en.wikipedia.org/wiki/NEC), [Red Hat](https://en.wikipedia.org/wiki/Red_Hat), [Texas Instruments](https://en.wikipedia.org/wiki/Texas_Instruments), [Oracle Corporation](https://en.wikipedia.org/wiki/Oracle_Corporation), and more.

OpenMP uses a [portable](https://en.wikipedia.org/wiki/Software_portability), scalable model that gives [programmers](https://en.wikipedia.org/wiki/Programmer) a simple and flexible interface for developing parallel applications for platforms ranging from the standard [desktop computer](https://en.wikipedia.org/wiki/Desktop_computer) to the [supercomputer](https://en.wikipedia.org/wiki/Supercomputer).

Comprised of three primary API components:

* Compiler Directives
* Runtime Library Routines
* Environment Variables

**Goals of OpenMP:**

* **Standardization:**
  + Provide a standard among a variety of shared memory architectures/platforms
  + Jointly defined and endorsed by a group of major computer hardware and software vendors
* **Lean and Mean:**
  + Establish a simple and limited set of directives for programming shared memory machines.
  + Significant parallelism can be implemented by using just 3 or 4 directives.
  + This goal is becoming less meaningful with each new release, apparently.
* **Ease of Use:**
  + Provide capability to incrementally parallelize a serial program, unlike message-passing libraries which typically require an all or nothing approach
  + Provide the capability to implement both coarse-grain and fine-grain parallelism
* **Portability:**
  + The API is specified for C/C++ and Fortran
  + Public forum for API and membership
  + Most major platforms have been implemented including Unix/Linux platforms and Windows

**OpenMP Programming Model :**

**Shared Memory Model:**

* OpenMP is designed for multi-processor/core, shared memory machines. The underlying architecture can be shared memory UMA or NUMA.
* Because OpenMP is designed for shared memory parallel programming, it largely limited to **single node** parallelism. Typically, the number of processing elements (cores) on a node determine how much parallelism can be implemented.

1. Binary Search:

Binary search is the search technique which works efficiently on the sorted lists. Hence, in order to search an element into some list by using binary search technique, we must ensure that the list is sorted.

Binary search follows divide and conquer approach in which, the list is divided into two halves and the item is compared with the middle element of the list. If the match is found then, the location of middle element is returned otherwise, we search into either of the halves depending upon the result produced through the match.

Binary search algorithm is given below.

BINARY\_SEARCH(A, lower\_bound, upper\_bound, VAL)

* **Step 1:** [INITIALIZE] SET BEG = lower\_bound  
  END = upper\_bound, POS = - 1
* **Step 2:** Repeat Steps 3 and 4 while BEG <=END
* **Step 3:** SET MID = (BEG + END)/2
* **Step 4:** IF A[MID] = VAL  
  SET POS = MID  
  PRINT POS  
  Go to Step 6  
  ELSE IF A[MID] > VAL  
  SET END = MID - 1  
  ELSE  
  SET BEG = MID + 1  
  [END OF IF]  
  [END OF LOOP]
* **Step 5:** IF POS = -1  
  PRINT "VALUE IS NOT PRESENT IN THE ARRAY"  
  [END OF IF]
* **Step 6:** EXIT

**Code:**

Parallel Binary Search Program:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <time.h>

#include <omp.h>

#include <unistd.h>

#define SIZE 1048576

void setUp(int a[], int size);

void tearDown(double start, double end, int a[], int size);

void setUp(int a[], int size){

int i;

srand(time(NULL));

for (i = 0; i<size; ++i) {

a[i] = i;

//printf("%d\n",a[i]);

}

//printf("Complete\n");

return;

}

void tearDown(double start, double end, int a[], int size) {

int sorted = 1;

int i;

printf("Time to execute: %f\n", end-start);

//for (i = 0; i < size-1; ++i) {

//sorted &= (a[i] <= a[i+1]);

//}

//printf("Array sorted: %d\n", sorted);

//#pragma omp parallel

//{

// #pragma omp master

//{

// printf("Num threads: %d\n", omp\_get\_num\_threads());

//}

//}

}

/\*

int parallelSearch(int arr[], int l, int r, int x,int num\_process)

{

int step=(r-l)/num\_process;

int flag[num\_process];

if(l<r) return -1;

#pragma omp parallel for

for(int i=0;i<num\_process;i++){

flag[i]=0;

if(x<arr[l+(step\*i)]){

flag[i]=1;

}

else if(x==arr[l+(step\*i)]){

return l+(step\*i);

}

}

//#pragma omp parallel for

for(int i=0;i<num\_process-1;i++){

{

if(flag[i]!=flag[i+1]){

return parallelSearch(arr, l, l+(step\*i), x, num\_process);

}

}

return -1;

}

\*/

int sequentialSearch(int arr[], int l, int r, int x,int num\_process){

for(int i=l;i<r;i++){

if(arr[i]==x){

return i;

}

}

return -1;

}

int parallelSearch(int arr[], int l, int r, int x,int num\_process)

{

int step,flag[num\_process+1],f;

flag[num\_process]=0;

while (l <= r)

{

if(r-l<=4){

for(int i=l;i<r;i++){

if(arr[i]==x){

return i;

}

}

return -1;

//printf("Zoned in");

}

step=(r-l+1)/num\_process;

//flag[num\_process];

f=-1;

// printf("L:%d R:%d \n",l,r);

// printf("STEP:%d \n",step);

#pragma omp parallel for

for(int i=0;i<num\_process;i++){

flag[i]=0;

f=-1;

if(x>arr[l+(step\*i)]){

// printf("x(%d) > jump(%d) \n",x,arr[l+(step\*i)]);

flag[i]=1;

}

else if(x==arr[l+(step\*i)]){

f=l+(step\*i);

}

}

if(f!=-1)return f;

/\*

printf("Flag:");

for(int i=0;i<num\_process+1;i++)

{

printf("%d",flag[i]);

}\*/

// printf("\n\n");

//sleep(2);

#pragma omp parallel for

for(int i=0;i<num\_process;i++)

{

if(flag[i]!=flag[i+1]){

l=l+(step\*i);

r=l+step;

}

}

}

return -1;

}

int binarySearch(int arr[], int l, int r, int x)

{

if (r >= l)

{

int mid = l + (r - l)/2;

// If the element is present at the middle

// itself

if (arr[mid] == x)

return mid;

// If element is smaller than mid, then

// it can only be present in left subarray

if (arr[mid] > x)

return binarySearch(arr, l, mid-1, x);

// Else the element can only be present

// in right subarray

return binarySearch(arr, mid+1, r, x);

}

return -1;

}

int main() {

//omp\_set\_nested(1);

//omp\_set\_dynamic(0); // Explicitly disable dynamic teams

//omp\_set\_num\_threads(1); // Use 4 threads for all consecutive parallel regions

int x=431;

int a[SIZE];

//int temp[SIZE];

double startTime, endTime;

int num\_threads;

#pragma omp parallel

{

#pragma omp master

{

num\_threads = omp\_get\_num\_threads();

}

}

printf("Parallel Thread Count:%d \n",num\_threads);

setUp(a, SIZE);

startTime = omp\_get\_wtime();

printf("Found at %d",parallelSearch(a,0,SIZE-1,x,num\_threads));

endTime = omp\_get\_wtime();

tearDown(startTime, endTime, a, SIZE);

printf("\n\nSerial SEARCH RESULTS:%d \n",num\_threads);

setUp(a, SIZE);

startTime = omp\_get\_wtime();

printf("Found at %d",binarySearch(a,0,SIZE-1,x));

endTime = omp\_get\_wtime();

tearDown(startTime, endTime, a, SIZE);

}

Time Analysis:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Searching Algorithm | Input Size (n) | Sequential Time | Parallel Time | Efficiency |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Binary Search  (Key= 54) | n=1024 | 1.153 | 1.542377 | 0.7477 |
| n=2048 | 1.673 | 1.236108 | 1.353 |
| n=4096 | 1.075 | 0.933585 | 1.150 |

Input: Key to be Searched:54

Output:

Original Array = 4096

New Array Size = 3576

Block size per processor = 3576/5 =715

Key found at Index = 15 by processor no = 0 .