Title : Write a Parallel program (using OpenMP) for parallel merge sort

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. Merge sort:

Merge sort parallelizes well due to use of the divide-and-conquer method. Several parallel variants are discussed in the third edition of Cormen, Leiserson, Rivest, and Stein's [*Introduction to Algorithms*](https://en.wikipedia.org/wiki/Introduction_to_Algorithms). The first of these can be very easily expressed in a pseudocode with [fork and join](https://en.wikipedia.org/wiki/Fork%E2%80%93join_model) keywords:

// *Sort elements lo through hi (exclusive) of array A.*

**algorithm** mergesort(A, lo, hi) **is**

**if** lo+1 < hi **then** // *Two or more elements.*

mid = ⌊(lo + hi) / 2⌋

**fork** mergesort(A, lo, mid)

mergesort(A, mid, hi)

**join**

merge(A, lo, mid, hi)

This algorithm is a trivial modification from the serial version, and its speedup is not impressive: when executed on an [infinite number of processors](https://en.wikipedia.org/wiki/Analysis_of_parallel_algorithms), it runs in Θ(*n*) time, which is only a Θ(log *n*) improvement on the serial version. A better result can be obtained by using a parallelized [merge algorithm](https://en.wikipedia.org/wiki/Merge_algorithm), which gives parallelism Θ(*n* / (log *n*)2), meaning that this type of parallel merge sort runs in

Θ ( ( n log ⁡ n ) ⋅ ( log ⁡ n ) 2 n ) = Θ ( ( log ⁡ n ) 3 ) {\displaystyle \Theta \left((n\log n)\cdot {\frac {(\log n)^{2}}{n}}\right)=\Theta ((\log n)^{3})} Such a sort can perform well in practice when combined with a fast stable sequential sort, such as [insertion sort](https://en.wikipedia.org/wiki/Insertion_sort), and a fast sequential merge as a base case for merging small arrays.

Merge sort was one of the first sorting algorithms where optimal speed up was achieved, with Richard Cole using a clever subsampling algorithm to ensure *O*(1) merge. Other sophisticated parallel sorting algorithms can achieve the same or better time bounds with a lower constant. For example, in 1991 David Powers described a parallelized [quicksort](https://en.wikipedia.org/wiki/Quicksort) (and a related [radix sort](https://en.wikipedia.org/wiki/Radix_sort)) that can operate in *O*(log *n*) time on a [CRCW](https://en.wikipedia.org/wiki/CRCW) [parallel random-access machine](https://en.wikipedia.org/wiki/Parallel_random-access_machine) (PRAM) with *n* processors by performing partitioning implicitly.[[15]](https://en.wikipedia.org/wiki/Merge_sort#cite_note-15) Powers[[16]](https://en.wikipedia.org/wiki/Merge_sort#cite_note-16) further shows that a pipelined version of Batcher's [Bitonic Mergesort](https://en.wikipedia.org/wiki/Bitonic_sorter) at *O*((log *n*)2) time on a butterfly [sorting network](https://en.wikipedia.org/wiki/Sorting_network) is in practice actually faster than his *O*(log *n*) sorts on a PRAM, and he provides detailed discussion of the hidden overheads in comparison, radix and parallel sorting.

**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 mergesort\_parallel\_omp (int a[], int size, int temp[], int threads)

{

if ( threads == 1) { mergesort\_serial(a, size, temp); }

else if (threads > 1)

{

#pragma omp parallel sections

{

#pragma omp section

mergesort\_parallel\_omp(a, size/2, temp, threads/2);

#pragma omp section

mergesort\_parallel\_omp(a + size/2, size - size/2, temp + size/2, threads - threads/2);

}

merge(a, size, temp);

} // threads > 1

}

void mergesort\_omp(...) {

#pragma omp parallel

#pragma omp single

mergesort\_parallel\_omp(...)

}

void mergesort\_parallel\_omp (int a[], int size, int temp[])

{

#pragma omp task

mergesort\_parallel\_omp(a, size/2, temp);

mergesort\_parallel\_omp(a + size/2, size - size/2, temp + size/2);

#pragma omp taskwait

merge(a, size, temp);

}

void mergesort\_parallel\_omp (int a[], int size, int temp[])

{

if (size < size\_threshold) {

mergesort\_serial(a, size, temp);

return;

}

#pragma omp task

mergesort\_parallel\_omp(a, size/2, temp);

mergesort\_parallel\_omp(a + size/2, size - size/2, temp + size/2);

#pragma omp taskwait

merge(a, size, temp);

}

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 .