Introduction

- **What is OpenMP**
  - Open specifications for Multi Processing
  - **Long version**: Open specifications for MultiProcessing via collaborative work between interested parties from the hardware and software industry, government and academia.
  - OpenMP (Open Multi-Processing) is an application programming interface (API) that supports **multi-platform shared memory multiprocessing programming in C, C++, and Fortran.**
- **API components:**
  - Compiler directives
  - Runtime library routines
  - Environment variables
- **Portability**
  - API is specified for C/C++ and Fortran
  - Implementations on almost all platforms including Unix/Linux and Windows.
  - OpenMP is used for **parallelism within a (multi-core) node** while MPI is used for parallelism between nodes.
- **Standardization**
  - Jointly defined and endorsed by major computer hardware and software vendors.

Process

- A process contains all the information needed to execute the program
  - Process ID
  - Program code
  - Data on run time stack
  - Global data
  - Data on heap

  Each process has its own address space.
- In multitasking, processes are given time slices in a round robin fashion.
  - If computer resources are assigned to another process, the status of the present process has to be saved, in order that the execution of the suspended process can be resumed at a later time.

Thread

- **What is Thread**
  - A **process** is an instance of a computer program that is being executed. It contains the program code and its current activity.
  - A **thread** of execution is the smallest unit of processing that can be scheduled by an operating system.
  - **Differences between threads and processes:**
    - A thread is contained inside a process.
    - Multiple threads can exist within the same process and share resources such as memory.
    - The threads of a process share the latter’s instructions (code) and its context (values that
    - its variables reference at any given moment).
    - Different processes do not share these resources.

Threads

- Thread model is an extension of the process model.
- Each process consists of multiple independent instruction streams (or threads) that are assigned computer resources by some scheduling procedure.
- Threads of a process share the address space of this process.
  - Global variables and all dynamically allocated data objects are accessible by all threads of a process
- Each thread has its own run time stack, register, program counter.
- Threads can communicate by reading/writing variables in the common address space.

OpenMP Programming Model

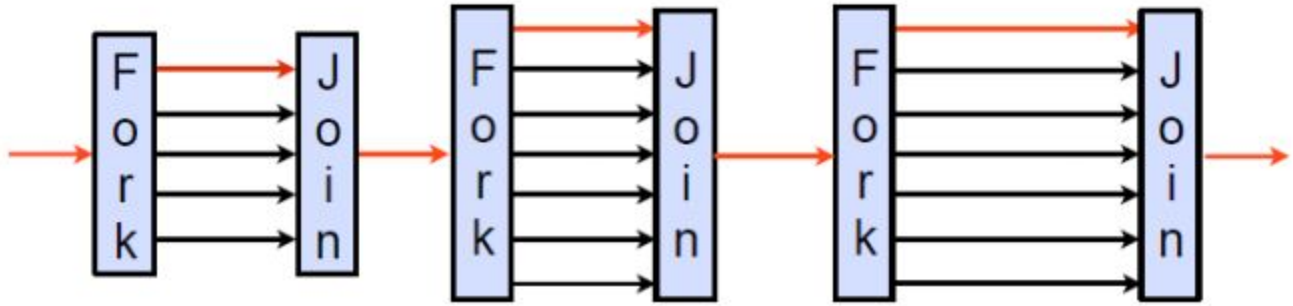
- Shared memory, thread-based parallelism
  - OpenMP is based on the existence of multiple threads in the shared memory programming paradigm.
  - A shared memory process consists of multiple threads.
- Explicit Parallelism
  - Programmer has full control over parallelization. OpenMP is not an automatic parallel programming model.
- Compiler directive based
  - Most OpenMP parallelism is specified through the use of compiler directives which are embedded in the source code.

OpenMP is not

- Necessarily implemented identically by all vendors
- Meant for distributed-memory parallel systems (it is designed for shared address spaced machines)
- Guaranteed to make the most efficient use of shared memory
- Required to check for data dependencies, data conflicts, race conditions, or deadlocks
- Required to check for code sequences
- Meant to cover compiler-generated automatic parallelization and directives to the compiler to assist such parallelization
- Designed to guarantee that input or output to the same file is synchronous when executed in parallel.

Fork-Join Parallelism

- OpenMP program begin as a single process: the *master thread*. The master thread executes sequentially until the first *parallel region* construct is encountered.
- When a parallel region is encountered, master thread
  - Create a group of threads by **FORK**.
  - Becomes the master of this group of threads, and is assigned the thread id 0 within the group.
- The statement in the program that are enclosed by the *parallel region* construct are then executed in parallel among these threads.
- **JOIN**: When the threads complete executing the statement in the *parallel region* construct, they synchronize and terminate, leaving only the master thread.



Master thread is shown in red.

OpenMP Code Structure

```
#include <stdlib.h>
#include <stdio.h>
#include "omp.h"

int main()
{
  #pragma omp parallel
  {
    int ID = omp_get_thread_num();
    printf("Hello (%d)\n", ID);
    printf(" world (%d)\n", ID);
  }
}
```

Set # of threads for OpenMP
In csh
setenv OMP_NUM_THREAD 8

Compile: g++ -fopenmp hello.c

Run: ./a.out

- “Pragma”: stands for “pragmatic information. A pragma is a way to communicate the information to the compiler.
- The information is non-essential in the sense that the compiler may ignore the information and still produce correct object program.

OpenMP Core Syntax

```
#include "omp.h"
int main ()
{
  int var1, var2, var3;
  // Serial code
  . . .
  // Beginning of parallel section.
  // Fork a team of threads. Specify variable scoping
  #pragma omp parallel private(var1, var2) shared(var3)
  {
     // Parallel section executed by all threads
       . . .
     // All threads join master thread and disband
  }

 // Resume serial code . . .
}
```

OpenMP C/C++ Directive Format

OpenMP directive forms

- C/C++ use compiler directives
  - Prefix: #pragma omp …
- A directive consists of a directive name followed by *clauses*

  Example: #pragma omp parallel default (shared) private (var1, var2)

OpenMP Directive Format (2)

General Rules:

- Case sensitive
- Only one directive-name may be specified per directive
- Each directive applies to at most one succeeding statement, which must be a structured block.
- Long directive lines can be “continued” on succeeding lines by escaping the newline character with a backslash “\” at the end of a directive line.

Number of Threads

- The number of threads in a parallel region is determined by the following factors, in order of precedence:
    1. Evaluation of the if clause
    2. Setting of the num_threads() clause
    3. Use of the omp_set_num_threads() library function
    4. Setting of the OMP_NUM_THREAD environment variable
    5. Implementation default – usually the number of cores on a node
- Threads are numbered from 0 (master thread) to N-1

Thread Creation: Parallel Region Example

- **Create threads with the parallel construct**

```
#include <stdlib.h>
#include <stdio.h>
#include "omp.h"

int main()
{
  int nthreads, tid;
  #pragma omp parallel num_threads(4) private(tid)
  {
    tid = omp_get_thread_num();
    printf("Hello world from (%d)\n", tid);
    if(tid == 0)
    {
      nthreads = omp_get_num_threads();
      printf("number of threads = %d\n", nthreads);
    }
  } // all threads join master thread and terminates
}
```

Clause to request threads

Each thread executes a copy of the code within the structured block

Pros & Cons - OpenMP

- **Pros of OpenMP-**
  - easier to program and debug than MPI
  - directives can be added incrementally - gradual parallelization
  - can still run the program as a serial code
  - serial code statements usually don't need modification
  - code is easier to understand and maybe more easily maintained
  - Incremental parallelism: can work on one part of the program at one time, no dramatic change to code is needed.
  - Unified code for both serial and parallel applications: OpenMP constructs are treated as comments when sequential compilers are used.
  - Original (serial) code statements need not, in general, be modified when parallelized with OpenMP. This reduces the chance of inadvertently introducing bugs.

- **Cons of OpenMP-**
  - can only be run in shared memory computers
  - requires a compiler that supports OpenMP
  - mostly used for loop parallelization
  - High chance of accidentally writing false sharing code.

Performance Expectations

- To expect to get a N times speedup when running a program parallelized using an OPENMP on a N processor platform. But its not….rarely it happens as,
  - **When a dependency exists**, a process must wait until th data it depends on is computed.
  - **When multiple processes share a non-parallel proof resource (like a file to write in),** their requests are executed sequentially. Therefore, each thread must wait until the other thread releases the resource.
  - A **large part of the program may not be parallelized** by OpenMP, which means that the theoretical upper limit of speedup is limited.
  - N processors in a symmetric multiprocessing (SMP) may have N times the computation power, but the **memory bandwidth usually does not scale up N times**.
  - Quite often, the **original memory path is shared by multiple processors** and performance degradation may be observed when they compete for the shared memory bandwidth.
  - Many other common problems affecting the final speedup in parallel computing also apply to OpenMP, like **load balancing and synchronization** overhead.

LABORATORY PRACTICE-I (GROUP A)

2. Vector and Matrix Operations-Design parallel algorithm to

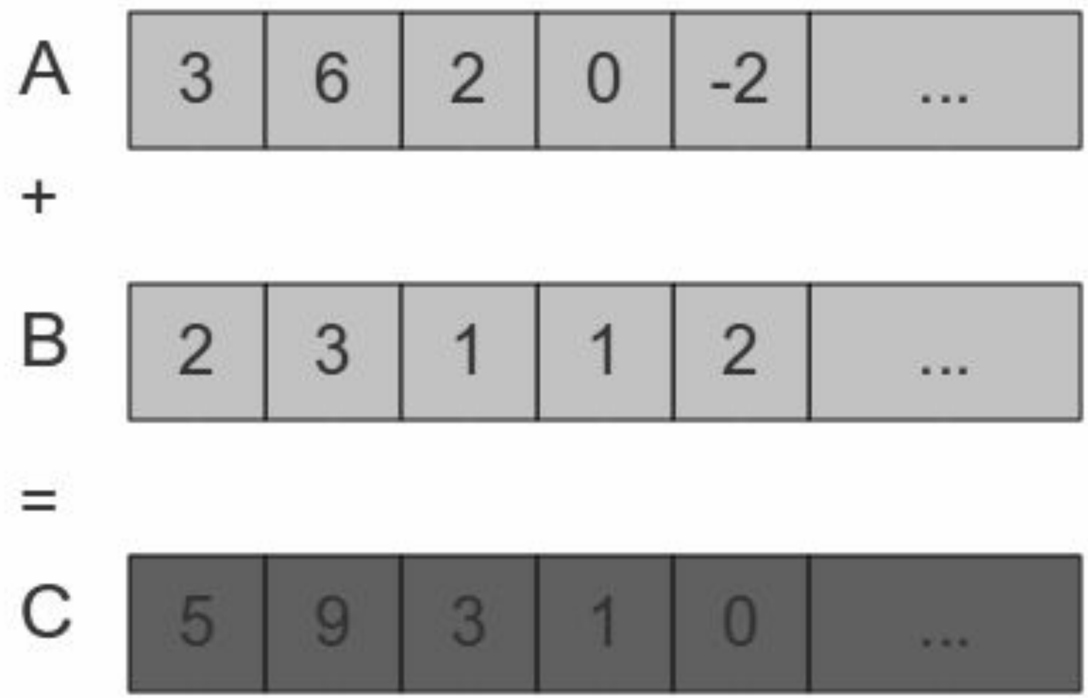
1. Add two large vectors
2. Multiply Vector and Matrix
3. Multiply two N × N arrays using n2 processors

3. Parallel Sorting Algorithms-

For Bubble Sort and Merger Sort, based on existing sequential algorithms, design and implement parallel algorithm utilizing all resources available.

2. Vector and Matrix Operations-Design parallel algorithm to

1. Add two large vectors



2. Vector and Matrix Operations-Design parallel algorithm to

2.Multiply Vector and Matrix

$$\begin{pmatrix} 2 & 3 \\ 4 & 5 \\ -1 & 6 \end{pmatrix} \times \begin{pmatrix} 4 \\ 7 \end{pmatrix} = \begin{pmatrix} (2 * 4) + (3 * 7) \\ (4 * 4) + (5 * 7) \\ (-1 * 4) + (6 * 7) \end{pmatrix} = \begin{pmatrix} 29 \\ 51 \\ 38 \end{pmatrix}$$

2. Vector and Matrix Operations-Design parallel algorithm to

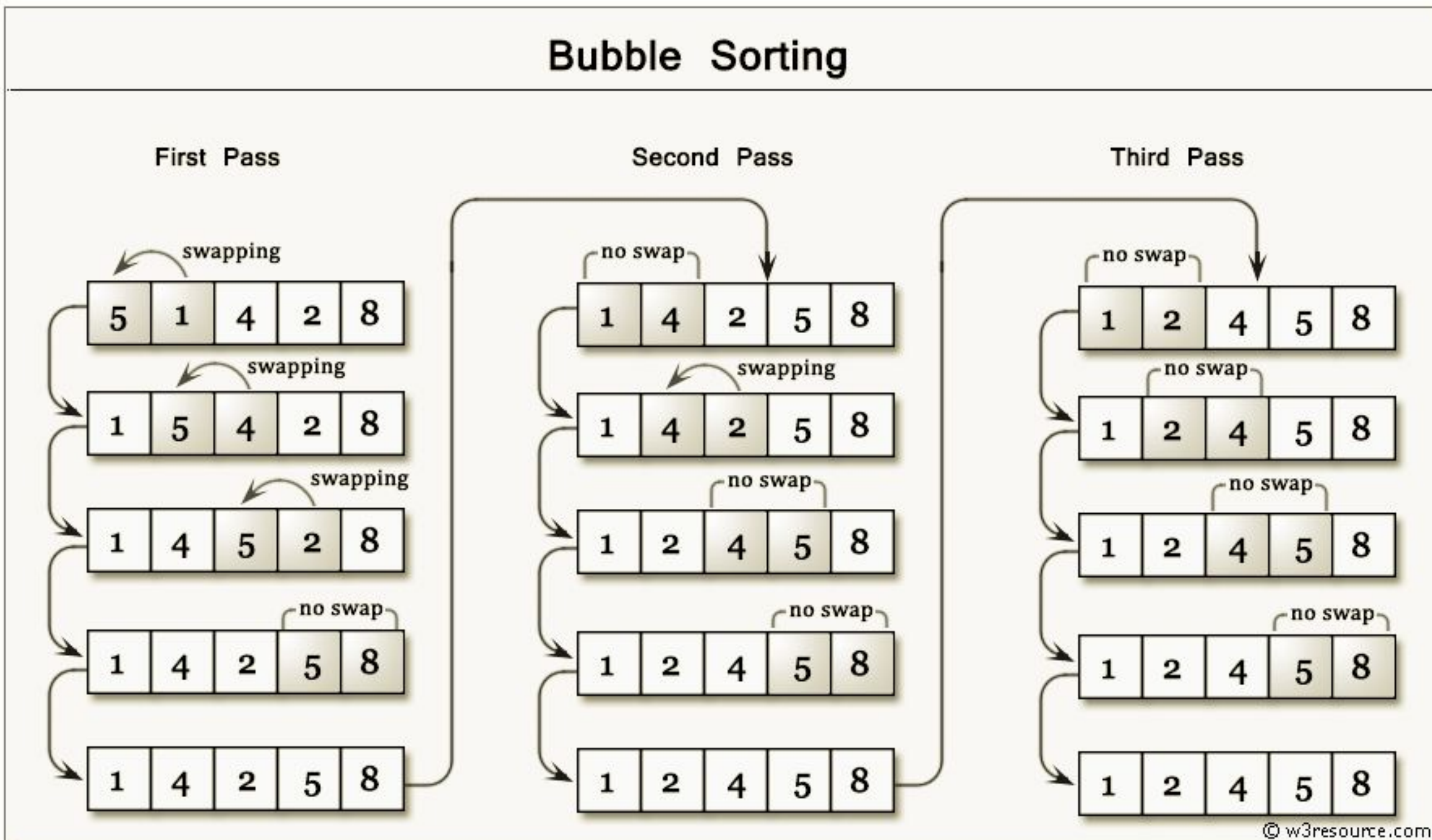
3.Multiply two N × N arrays using n2 processors

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \times \begin{bmatrix} j & k & l \\ m & n & o \\ p & q & r \end{bmatrix} =$$

$$\begin{bmatrix} aj + bm + cp & ak + bn + cq & al + bo + cr \\ dj + em + fp & dk + en + fq & dl + eo + fr \\ gj + hm + ip & gk + hn + iq & gl + ho + ir \end{bmatrix}$$

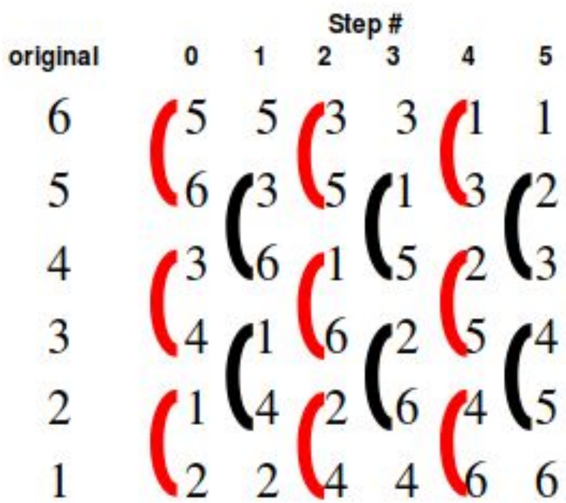
3. Parallel Sorting Algorithms-

For Bubble Sort and Merger Sort, based on existing sequential algorithms, design and implement parallel algorithm utilizing all resources available.



Odd-Even Bubble Sort

N = 6



N = 6

original	Step # 0	1	2	3	4
6	5	4	3	2	1
5	4	3	2	1	2
4	3	2	1	3	3
3	2	1	4	4	4
2	1	5	5	5	5
1	6	6	6	6	6

Non-threaded

original	Step # 0	1	2	3	4	5
6	5	5	3	3	1	1
5	6	3	5	1	3	2
4	3	6	1	5	2	3
3	4	1	6	2	5	4
2	1	4	2	6	4	5
1	2	2	4	4	6	6

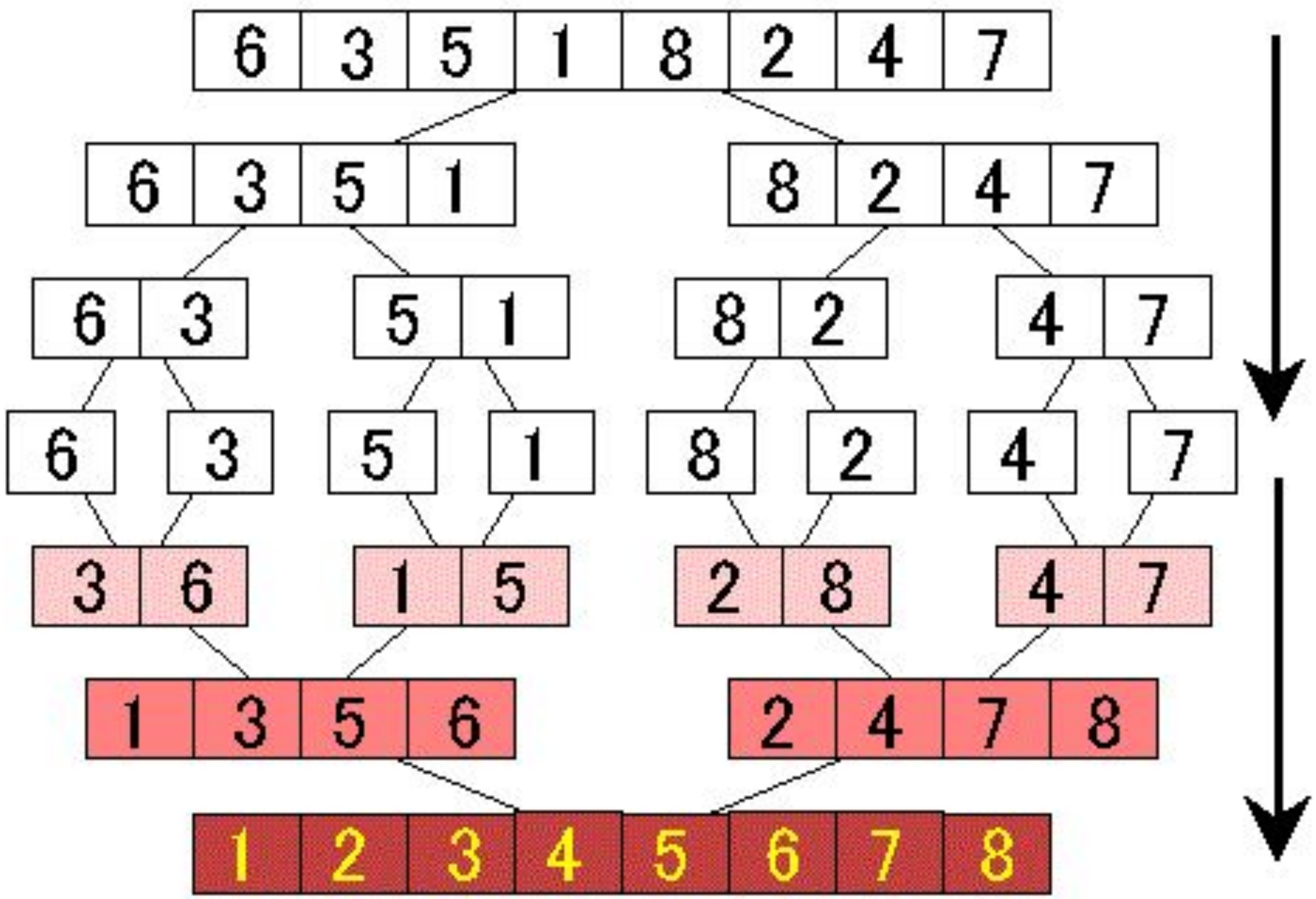
Threaded

- This is basically a variation of bubble-sort.
- This algorithm is divided into two phases- Odd and Even Phase.
- The algorithm runs until the array elements are sorted and in each iteration two phases occurs- Odd and Even Phases.
- In the odd phase, we perform a bubble sort on odd indexed elements and in the even phase, we perform a bubble sort on even indexed elements.
- Single Pass = One Even Phase + One Odd Phase

Bubble Sort Code:

```
#include<iostream>
#include<stdlib.h>
#include<omp.h>
using namespace std;
void bubble(int *, int);
void swap(int &, int &);


void bubble(int *a, int n)  {
 for(  int i = 0;  i < n;  i++ )  {
     int first = i % 2;
     #pragma omp parallel for shared(a,first)   / /Creates parallel threads to swap consecutive elements
     for(  int j = first;  j < n-1;  j += 2  ) {
          if(  a[ j ]  >  a[ j+1 ]  ) {
           swap(  a[ j ],  a[ j+1 ]  );
    }}}}


void swap(int &a, int &b){
 int test; test=a; a=b; b=test;                         / /swaps two variables
}


int main(){
 int *a,n;
cout<<"\n enter total no of elements=>"; cin>>n; a=new int[n];
cout<<"\n enter elements=>"; for(int i=0;i<n;i++) {  cin>>a[i]; }
bubble(a,n);
cout<<"\n sorted array is=>"; for(int i=0;i<n;i++) {  cout<<a[i]<<endl; }


return 0;}
```

Merge Sort



6 3 5 1 8 2 4 7
6 3 5 1
8 2 4 7
6 3
5 1
8 2
4 7
6
3
5
1
8
2
4
7
3 6
1 5
2 8
4 7
1 3 5 6
2 4 7 8
1 2 3 4 5 6 7 8

MERGE SORT:

```
#include<iostream>
#include<stdlib.h>
#include<omp.h>
using namespace std;


void mergesort(int a[],int i,int j);
void merge(int a[],int i1,int j1,int i2,int j2);


void mergesort(int a[],int i,int j){
   int mid;   if(i<j)   {
      mid=(i+j)/2;
         #pragma omp parallel sections      {                                    / /implements both left and right sections in parallel
         #pragma omp section        {           mergesort(a,i,mid);              }     / /parallelized left section
         #pragma omp section        {           mergesort(a,mid+1,j);            }    / /parallelized right section
      }
      merge(a,i,mid,mid+1,j);      }}         / /merge function outside parallel section to sort

void merge(int a[],int i1,int j1,int i2,int j2){
int temp[1000];      int i,j,k;   i=i1;      j=i2;      k=0;
while(i<=j1 && j<=j2)      {
      if(a[i]<a[j])      {         temp[k++]=a[i++];      }           / /copying lowest element from left or right subarray to temp array
      else      {         temp[k++]=a[j++]; }      }
while(i<=j1)      {      temp[k++]=a[i++];   }                     / /copying remaining elements from left subarray
while(j<=j2)      {      temp[k++]=a[j++];   }                     / /copying remaining elements from left subarray
 for(i=i1,j=0;i<=j2;i++,j++)   {      a[i]=temp[j];   }   }


int main(){   int *a,n,i;
cout<<"\n enter total no of elements=>";   cin>>n;   a= new int[n];
cout<<"\n enter elements=>";   for(i=0;i<n;i++)   {      cin>>a[i];   }
 mergesort(a, 0, n-1);
 cout<<"\n sorted array is=>";   for(i=0;i<n;i++)   {      cout<<"\n"<<a[i];   }        return 0;}
```