MAEER's MIT College of Engineering, Pune
Department of Computer Engineering

Three Days Faculty Development Programme
(in association with BoS Computer Engineering, SPPU, Pune)

On

Laboratory Practice-I

(BE Computer Engineering Course 2015)

10th July 2018 - 12th July 2018

- Prof. Jayshree Aher(Ghorpade)

What is OpenMP?

OpenMP Is:

- An Application Program Interface (API) that may be used to explicitly direct multi-threaded, shared memory parallelism.

- Comprised of three primary API components:
  1. Compiler Directives
  2. Runtime Library Routines
  3. Environment Variables

- An abbreviation for: Open Multi-Processing
  - OpenMP consists of a set of compiler #pragmas that control how the program works. The pragmas are designed so that even if the compiler does not support them, the program will still yield correct behavior, but without any parallelism.
- You can compile them like this: g++ tmp.cpp -fopenmp

Goals of OpenMP:

- **Standardization:**
  - a variety of shared memory architectures/platforms
  - group of major computer hardware and software vendors

- **Lean and Mean:**
  - simple and limited set of directives for programming shared memory machines.
  - Significant parallelism can be implemented by using just 3 or 4 directives.

- **Ease of Use:**
  - Provide capability to incrementally parallelize a serial program, unlike message-passing libraries
  - 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

Few commands:

Syntax	Example	Explanation
#pragma omp parallel	*#pragma omp parallel { // Code inside this region runs in parallel. printf('Hello!\n'); }*	*The parallel construct starts a parallel block. It creates a team of N threads (where N is determined at runtime, usually from the number of CPU cores, but may be affected by a few things), all of which execute the next statement.*
#pragma omp parallel for if(parallelism_enabled)	*extern int parallelism_enabled; #pragma omp parallel for if(parallelism_enabled) for(int c=0; c<n; ++c) handle(c);*	*The parallelism can be made conditional by including a if clause in the parallel command.In this case, if parallelism_enabled evaluates to a zero value, the number of threads in the team that processes the for loop will always be exactly one.*
#pragma omp for	*#pragma omp for for(int n=0; n<10; ++n) { printf(' %d', n); } printf('.\n');*	*The for construct splits the for-loop so that each thread in the current team handles a different portion of the loop. This loop will output each number from 0...9 once. However, it may do it in arbitrary order. It may output, for example: 0 5 6 7 1 8 2 3 4 9.*
#pragma omp parallel for	*#pragma omp parallel for for(int n=0; n<10; ++n) printf(' %d', n); printf('.\n');*	*#pragma omp for only delegates portions of the loop for different threads in the current team. A team is the group of threads executing the program. At program start, the team consists only of a single member: the master thread that runs the program.To create a new team of threads, you need to specify the parallel keyword.*
#pragma omp parallel num_threads(3)	*#pragma omp parallel num_threads(3) { // This code will be executed by three threads. // Chunks of this loop will be divided amongst // the (three) threads of the current team. #pragma omp for for(int n=0; n<10; ++n) printf(' %d', n); }*	*You can explicitly specify the number of threads to be created in the team, using the num_threads attribute.*
#pragma omp for schedule(static)	*#pragma omp for schedule(static) for(int n=0; n<10; ++n) printf(' %d', n); printf('.\n');*	*static is the default schedule as shown above. Upon entering the loop, each thread independently decides which chunk of the loop they will process.*

Syntax	Example	Explanation
#pragma omp for schedule(dynamic)	*#pragma omp for schedule(dynamic) for(int n=0; n<10; ++n) printf(' %d', n); printf('.\n');*	*In the dynamic schedule, there is no predictable order in which the loop items are assigned to different threads. Each thread asks the OpenMP runtime library for an iteration number, then handles it, then asks for next, and so on.*
#pragma omp for schedule(dynamic, 3)	*#pragma omp for schedule(dynamic, 3) for(int n=0; n<10; ++n) printf(' %d', n); printf('.\n');*	*The chunk size can also be specified to lessen the number of calls to the runtime library:In this example, each thread asks for an iteration number, executes 3 iterations of the loop, then asks for another, and so on. The last chunk may be smaller than 3, though.*
#pragma omp for ordered schedule(dynamic)	*#pragma omp for ordered schedule(dynamic) for(int n=0; n<100; ++n) { files[n].compress(); #pragma omp ordered send(files[n]); }*The order in which the loop iterations are executed is unspecified, and depends on runtime conditions.This loop 'compresses' 100 files with some files being compressed in parallel, but ensures that the files are 'sent' in a strictly sequential order.*	*The order in which the loop iterations are executed is unspecified, and depends on runtime conditions.This loop 'compresses' 100 files with some files being compressed in parallel, but ensures that the files are 'sent' in a strictly sequential order.*
#pragma omp parallel for collapse(2)	*#pragma omp parallel for collapse(2) for(int y=0; y<25; ++y) for(int x=0; x<80; ++x) { tick(x,y); }*	*When you have nested loops, you can use the collapse clause to apply the threading to multiple nested iterations.*
#pragma omp parallel for reduction(+:sum)	*int sum=0; #pragma omp parallel for reduction(+:sum) for(int n=0; n<1000; ++n) sum += table[n];*	*The reduction clause is a special directive that instructs the compiler to generate code that accumulates values from different loop iterations together in a certain manner. It is discussed in a separate chapter later in this article. Example:*
#pragma omp sections	*#pragma omp sections { { Work1(); } #pragma omp section { Work2(); Work3(); } #pragma omp section { Work4(); } }*	*This code indicates that any of the tasks Work1, Work2 + Work3 and Work4 may run in parallel, but that Work2 and Work3 must be run in sequence. Each work is done exactly once.*
#pragma omp simd	*float a[8], b[8]; ... #pragma omp simd for(int n=0; n<8; ++n) a[n] += b[n];*	*SIMD means that multiple calculations will be performed simultaneously by the processor, using special instructions that perform the same calculation to multiple values at once.*

Syntax	Example	Explanation
#pragma omp simd collapse(2)	*#pragma omp simd collapse(2) for(int i=0; i<4; ++i) for(int j=0; j<4; ++j) a[j*4+i] += b[i*4+j];*	*The collapse clause can be added to bind the SIMDness into multiple nested loops. The example code below will direct the compiler into attempting to generate instructions that calculate 16 values simultaneously, if at all possible.*
#pragma omp simd reduction(+:sum)	*int sum=0; #pragma omp simd reduction(+:sum) for(int n=0; n<1000; ++n) sum += table[n];*	*The reduction clause can be used with SIMD just like with parallel loops.*
#pragma omp parallel for simd reduction(+:result)	*float sum(float* table) { float result=0; #pragma omp parallel for simd reduction(+:result) for(int n=0; n<1000; ++n) result += table[n]; return result; }*	*The for and simd constructs can be combined, to divide the execution of a loop into multiple threads, and then execute those loop slices in parallel using SIMD.*
#pragma omp task	*struct node { node *left, *right; };extern void process(node*);void traverse(node* p){ if (p->left) #pragma omp task // p is firstprivate by default traverse(p->left); if (p->right) #pragma omp task // p is firstprivate by default traverse(p->right); process(p);}*	*When for and sections are too cumbersome, the task construct can be used.*
#pragma omp declare target	*#pragma omp declare targetint x;void murmur() { x+=5; }#pragma omp end declare target*	*The declare target and end declare target directives delimit a section of the source code wherein all declarations, whether they are variables or functions/subroutines, are compiled for a device.This creates one or more versions of 'x' and 'murmur'. A set that exists on the host computer, and also a separate set that exists and can be run on a device.*
#pragma omp target teams	*#include <stdio.h> int main(void) { #pragma omp target teams { printf('test\n'); } return 0; }*	*This directive can be only used directly inside a target construct. The optional attribute num_teams can be used to specify the maximum number of teams created. The actual number of teams may be smaller than this number. The master thread of each team will execute the code inside that team.*
#pragma omp parallel default(none) shared(b)	*int a, b=0; // This code won't compile: It requires explicitly // specifying whether a is shared or private. #pragma omp parallel default(none) shared(b) { b += a; }*	*The default clause can also be used to set that all variables are shared by default (default(shared)). The most useful purpose on the default clause is to check whether you have remembered to consider all variables for the private/shared question, using the default(none) setting.*

Syntax	Example	Explanation
#pragma omp atomic	*#pragma omp atomic counter += value;*	*The atomic keyword in OpenMP specifies that the denoted action happens atomically. It is commonly used to update counters and other simple variables that are accessed by multiple threads simultaneously.*
#pragma omp critical(dataupdate)	*#pragma omp critical(dataupdate) { datastructure.reorganize(); } ... #pragma omp critical(dataupdate) { datastructure.reorganize_again(); }*	*The critical construct restricts the execution of the associated statement / block to a single thread at time.The critical construct may optionally contain a global name that identifies the type of the critical construct. No two threads can execute a critical construct of the same name at the same time.In this example, only one of the critical sections named 'dataupdate' may be executed at any given time, and only one thread may be executing it at that time. I.e. the functions 'reorganize' and 'reorganize_again' cannot be invoked at the same time, and two calls to the function cannot be active at the same time.*
#pragma omp flush(a,b)	*/* presumption: int a = 0, b = 0; */ /* First thread */ /* Second thread */ b = 1; a = 1; #pragma omp flush(a,b) #pragma omp flush(a,b) if(a == 0) if(b == 0) { { /* Critical section */ /* Critical section */ } }*	*The flush directive can be used to ensure that the value observed in one thread is also the value observed by other threads.In this example, it is enforced that at the time either of a or b is accessed, the other is also up-to-date, practically ensuring that not both of the two threads enter the critical section. (Note: It is still possible that neither of them can enter it.)You need the flush directive when you have writes to and reads from the same data in different threads.*
#pragma omp parallel for private(a) shared(b)	*int a, b=0; #pragma omp parallel for private(a) shared(b) for(a=0; a<50; ++a) { #pragma omp atomic b += a; }*	*This example explicitly specifies that a is private (each thread has their own copy of it) and that b is shared (each thread accesses the same variable).*
#pragma omp parallel firstprivate(a,c) shared(b) num_threads(2)	*#include <string> #include <iostream> int main() { std::string a = 'x', b = 'y'; int c = 3; #pragma omp parallel firstprivate(a,c) shared(b) num_threads(2) { a += 'k'; c += 7; std::cout << 'A becomes (' << a << '), b is (' << b << ')\n'; } }*	*If you actually need a copy of the original value, use the firstprivate clause.Now the output becomes 'A becomes (xk), b is (y)'.*

Assignments:

- Parallel Sorting Algorithms-For Bubble Sort and Merger Sort, based on existing sequential algorithms, design andimplement parallel algorithm utilizing all resources available.

- Parallel Search Algorithm-Design and implement parallel algorithm utilizing all resources available. for
  1. Binary Search for Sorted Array
  2. Breadth-First Search (tree or an undirected graph)

HPC Assignment:

- Parallel Sorting Algorithms-For Bubble Sort and Merger Sort, based on existing sequential algorithms, design andimplement parallel algorithm utilizing all resources available.

BUBBLE SORT:

N = 6

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

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

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:

```
#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;}
```

HPC Assignment:

- Parallel Search Algorithm-Design and implement parallel algorithm utilizing all resources available. for
  1. Binary Search for Sorted Array
  2. Breadth-First Search (tree or an undirected graph)

BINARY SEARCH:

```
#include<iostream>
#include<stdlib.h>
#include<omp.h>
using namespace std;
int binary(int *, int, int, int);

int binary(int *a, int low, int high, int key){
int mid;  mid=(low+high)/2;
int low1,low2,high1,high2,mid1,mid2,found=0,loc=-1;

#pragma omp parallel sections {

#pragma omp section     {
low1=low;   high1=mid;
while(low1<=high1)   {
   if(!(key>=a[low1] && key<=a[high1]))   {    low1=low1+high1;    continue;   }
   cout<<"here1";
   mid1=(low1+high1)/2;
   if(key==a[mid1])   {    found=1;    loc=mid1;    low1=high1+1;   }
   else if(key>a[mid1])   {   low1=mid1+1;   }
   else if(key<a[mid1])    high1=mid1-1;
}
}
```

BINARY SEARCH (cont…):

```
#pragma omp section     {
   low2=mid+1;   high2=high;
   while(low2<=high2)   {
      if(!(key>=a[low2] && key<=a[high2]))   {    low2=low2+high2;    continue;   }
      cout<<"here2";
      mid2=(low2+high2)/2;
      if(key==a[mid2])   {  found=1;    loc=mid2;    low2=high2+1;    }
     else if(key>a[mid2])   {   low2=mid2+1;   }
     else if(key<a[mid2])    high2=mid2-1;
  }     } }
 return loc;
}

int main(){
  int *a,i,n,key,loc=-1;
  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];       }
  cout<<"\n enter key to find=>"; cin>>key;
  loc=binary(a,0,n-1,key);
 if(loc==-1)  cout<<"\n Key not found.";
 else  cout<<"\n Key found at position=>"<<loc+1;
 return 0;
}
```

BREADTH FIRST SEARCH:

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

class node{   public:  node *left, *right; int data;};
class Breadthfs{  public:  node *insert(node *, int); void bfs(node *); };

node *insert(node *root, int data){                    / /inserts a node in tree
   if(!root) {    root=new node;  root->left=NULL;  root->right=NULL;  root->data=data;  return root; }
  queue<node *> q; q.push(root);
  while(!q.empty()) {
    node *temp=q.front();  q.pop();
    if(temp->left==NULL)  {     temp->left=new node;   temp->left->left=NULL;   temp->left->right=NULL
temp->left->data=data;    return root;  }
    else  { q.push(temp->left); }

   if(temp->right==NULL)  {     temp->right=new node;   temp->right->left=NULL;   temp->right->right=NULL;
temp->right->data=data;    return root;  }
   else  {  q.push(temp->right);  }
 } }
```

```
void bfs(node *head){
 queue<node*> q;  q.push(head);    int qSize;
 while (!q.empty())   {
    qSize = q.size();
    #pragma omp parallel for                / /creates parallel threads
     for (int i = 0; i < qSize; i++)    {    node* currNode;
      #pragma omp critical    {   currNode = q.front();     q.pop(); cout<<"\t"<<currNode->data;     }  / /prints parent node
     #pragma omp critical    {
       if(currNode->left)     q.push(currNode->left);              / /push parent's left node in queue
      if(currNode->right)     q.push(currNode->right);    }       / /push parent's right node in queue
  } }
}
int main(){
   node *root=NULL; int data; char ans;
   do {  cout<<"\n enter data=>";  cin>>data;
   root=insert(root,data);
  cout<<"do you want insert one more node?";  cin>>ans;
 }while(ans=='y'||ans=='Y');
  bfs(root);
 return 0;}
```

THANK YOU…