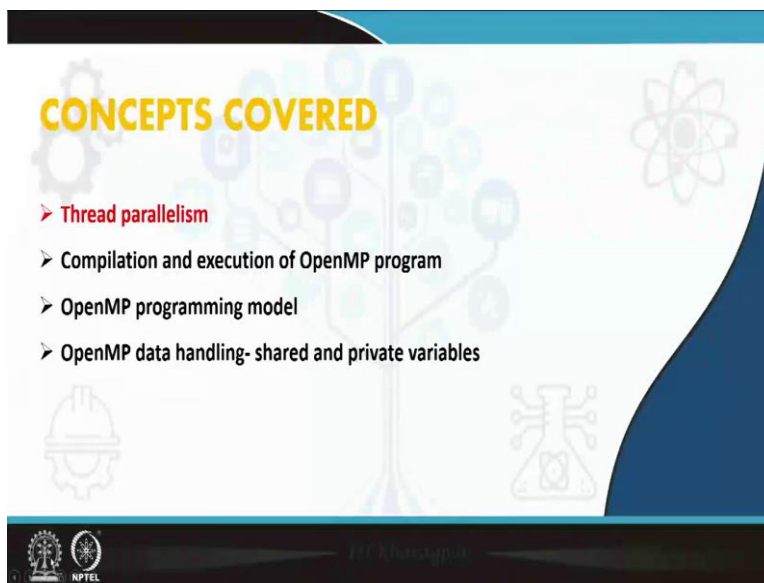


**High Performance Computing for Scientists and Engineers**  
**Prof. Somnath Roy**  
**Department of Mechanical Engineering**  
**Indian Institute of Technology, Kharagpur**

**Module - 02**  
**OpenMP Programming**  
**Lecture – 13**  
**Introduction to OpenMP (continued)**

Welcome, we are going through the lectures of the class ,High Performance Computing for Scientists and Engineers and this is the second module which is OpenMP programming and we are continuing with the lecture 13 which is continuation of the previous lecture, lecture 12 introduction to OpenMP. We will continue it for one more lecture . The subsequent lecture we will also discuss about Introduction to OpenMP.

(Refer Slide Time: 00:56)



So, in the previous lecture we have discussed about threaded parallelism. I gave some introduction to shared memory systems. So, we have looked into shared memory systems in much detail earlier, when discussing about parallel computing architecture, but I also try to give another view on introductory shared memory system.

This view is a programmer view on that, given a problem have to identify what are the parallel regimes, have to launch the threads and there will be some sequential part of the program. In the sequential part only one thread will be active, when one goes to the parallel part, we can launch number of threads; what will be the number of threads that

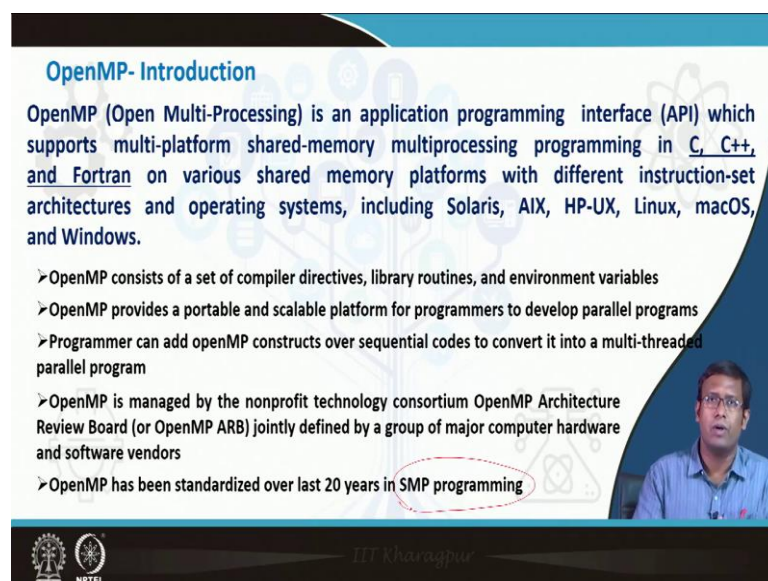
can be determined by the programmer and also determined by the person who is executing.

If there are some thumb rule for finding out number of threads for different cases, we will discuss that. So, there is a single thread which is active in the sequential part, then multiple threads are launched when the parallel part of the program is coming and then again when this parallel part is over, it will join the threads and one of the threads will be only active. We looked from a programmer's point of view in a sense we looked that how these threads can interact, how these threads can write data to memory systems, some features like this.

So, we will go more directly on to OpenMP in this lecture and we will have some introduction to OpenMP after that what OpenMP is, who developed OpenMP, these things will discuss. After that we will see, which is very very important, how we can compile an OpenMP program and execute an OpenMP program and how can we do it in our own computer.

What is the software we require to launch OpenMP? We will look into OpenMP programming model, followed by data share, data handling part in OpenMP. So, these two will be completed in this lecture and the subsequent lecture.

(Refer Slide Time: 03:15)



**OpenMP- Introduction**

OpenMP (Open Multi-Processing) is an application programming interface (API) which supports multi-platform shared-memory multiprocessing programming in C, C++, and Fortran on various shared memory platforms with different instruction-set architectures and operating systems, including Solaris, AIX, HP-UX, Linux, macOS, and Windows.

- OpenMP consists of a set of compiler directives, library routines, and environment variables
- OpenMP provides a portable and scalable platform for programmers to develop parallel programs
- Programmer can add openMP constructs over sequential codes to convert it into a multi-threaded parallel program
- OpenMP is managed by the nonprofit technology consortium OpenMP Architecture Review Board (or OpenMP ARB) jointly defined by a group of major computer hardware and software vendors
- OpenMP has been standardized over last 20 years in SMP programming

IIT Khargpur

NPTEL

OpenMP is an Application Programming Interface or API which supports multi-platform shared memory multiprocessing programming. It can be used in different platforms of shared memory multiprocessing programming, using C, C++ and Fortran and this is very important. OpenMP cannot be used directly, you cannot try to just write an OpenMP program, you have to write a C, C++ or Fortran program on which you can use OpenMP as library function calls and it's an API.

This can be used in various shared memory platforms with different instruction set architectures and different operating systems like Solaris which comes from Sun AIX, HP-UX, Linux, macOS, Windows. On different type of systems with different operating systems you can learn OpenMP jobs.

But these OpenMP programs are basically C, C++ or Fortran programs we have to first write a program using say C and then in which you can include OpenMP library calls and then you can get multiprocessor program using a shared memory system that will be the OpenMP parallel program.

OpenMP consists a set of compiler directive, library routines and environment variables. OpenMP provides a portable and scalable platform for programmers to develop parallel programs. Portable means; you can put it from one platform to another, you can develop it somewhere else, you can run it in another platform. You develop a program in your windows platform, you take it to a Linux server and run it there.

Its scalable, you can develop a program and test it up to say 20 processors; you go to a 64-processor system, you can still run this program. So, it can be scaled to larger number of processors. So, one important point is that, it consists of compiler directives, library routine and environment variables. So, when you use OpenMP you have to use some of the directives inside your program, some of the library routines which will be called, so some directives will go to the compiler, some library it will run and it also have to set some environment variables.

Programmer can add OpenMP constructs over sequential codes to convert to a multi-threaded parallel program. This is extremely important thing for scientific computing experts, if you have your own program to solve say gravity wave equations or finite element method for some beam bending, etcetera, you identify that this part can be parallelized, you can write OpenMP construct over your sequential code, over your C

code, you can write some OpenMP lines and convert it into a multithreaded parallel program.

So, parallelization of your of already existing sequential program is quite doable using OpenMP and that is one of the goals of OpenMP. We will see how OpenMP was developed and how its standardized and we can understand that, this has been one of the driving force of getting OpenMP available to different people across different fields, so that they can convert their sequential code to multi-threaded parallel program or you if you are developing your parallel program from scratch, you can write it something like a simple C code adding some of the OpenMP constructs.

OpenMP is managed by the nonprofit technology consortium. OpenMP architecture review board jointly defined by major computer hardware and software vendors. So, if we go back to history when parallel programming came and especially threaded programming came, some of the thread parallelism languages were very low-level language; you have to be an expert in computer programming or you have to be very well conversant in computer science field in order to develop the parallel program.

However, it has been seen that the major users of parallelism are not people who are experts in specifically computer programming, but people who are coming from background of science, engineering, finance, etcetera. So, then at certain point of time a consortium is formed between hardware and software vendors, because they are the people who has to sell it to a wider community, so, that they can use parallel architectures as well as software for using parallel programs.

So, they came and formed this OpenMP review board, and this board sits down very frequently and finalizes different releases in OpenMP. So, whatever OpenMP we see today has been developed over last couple of decades, and has been finalized by OpenMP architecture review board, and if somebody develops something, that is given to all the hardware and software vendors, made member of this review board. So, it is a nonprofit organization who manages it, but why is it given that? To keep OpenMP portable, if you develop it in a windows platform it should run in Solaris also.

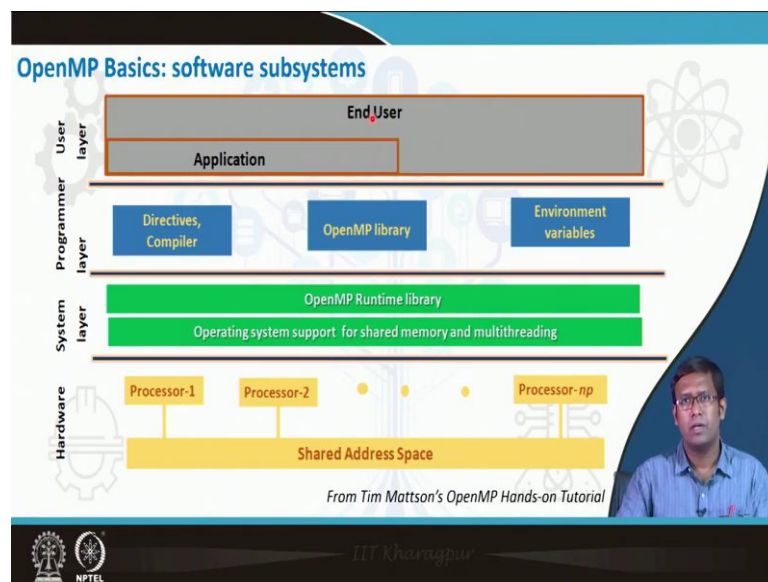
So, whatever developments coming from a group who are working with windows, this development must go to all the other software vendors. If you are developing something in HP, it must also run with dell. So, this is managed by OpenMP and this is specific

purpose is that in some sense users from other community will come and learn of OpenMP programming.

This has been standardized over last 20 years in SMP or symmetric multiprocessor programming and also for non-uniform memory access multi computer programming. OpenMP has been standardized and very recently; say in last couple of years OpenMP has also been developed for G P U computing. So, using OpenMP you can use you can launch multiple threads in in G P U and gateway utilization of G P G P S.

So, we will see it briefly when we will discuss about G P S that how OpenMP can be used for G P S, but right now we mostly focus on the symmetric multiprocessor or uniform memory access U M A type of SMP, and see how OpenMP can be used for that and this is simply, there is a common shared address space in which multiple C P Us are connected. Each C P U can access the data elements in that address space and update it and the C P Us are working in parallel. They are executing concurrent stream to update different locations of the memory element. This is the programming model which will follow in this part of this discussions.

(Refer Slide Time: 11:16)



So, and now if we come to the software sub systems of OpenMP that how OpenMP works in terms of the software development and software-based operation of a parallel program, OpenMP based program, how does it work? So, we can see that there is of

course some hardware, there is a shared address space memory that (DRAM) which is connected with multiple processors, and over this hardware OpenMP will work.

So, we go from a bottom up approach. So, the in this what will be required? Some executable will come and this executable will be running through all the processors connected with the main address space. So, we need an operating system which will support for shared memory access of different processors, and what we called as multi threading's that different threads are active over different processors connected to the same shared memory system has to be supported by the operating system.

When it is supported by the operating system, we will send the instructions to the operating system and that will be done by OpenMP runtime library. This is the systems layer, there is a software systems layer which is operational over the hardware layer. Now, comes the programmer layer, if you write your OpenMP program what will you do, so, that your OpenMP runtime libraries are called and they are interacting with the operating system that the jobs are launched in different processors concurrently connected with the same address space, and for that programmers will write the directives inside the program. When it will compile this will call the OpenMP libraries, programmer has to call the OpenMP library functions and the compile job will link to the OpenMP libraries, and ask the OpenMP runtime libraries to interact with the operating system to execute the program and also the programmer can set some of the environment variables.

Say, on how many processors it should run or how many threads will be active at one point of time? Which thread will be specifically asked to do something else? So, this sort of environments , the programmer can specify inside the code. So, this is programmers layer, and above that that once, this program is ready program, and it compiles and gets an executable, this can be used by the end user who is basically from an application area, who has a program say on computing some of the economic cost benefit matrix, and that is parallelized by the programmer which is run going through the systems and finally, launched in the hardware. The end user is only running the OpenMP program of that for the application he wants to solve.

So, this is typical hierarchy of OpenMP software subsystems, that user only knows that it has an application job which has to be worked. A programmer has taken the application,

wrote a program inside, has included the directives and which will call the library functions during compilation and will also set the environment variables and this will go to the when in the systems layer of the OpenMP library, runtime libraries will be active and will interact with the OS and the OS will launch the jobs, as multi-threaded jobs connected with the shared memory system. This is taken from Tim Mattson OpenMP hands on tutorial.

If you go to OpenMP annual review board website, the official website for OpenMP which is [OpenMP.org](http://OpenMP.org) will find a very good collection of materials including textbooks, including lecture notes and tutorials on OpenMP. You can self-learn OpenMP using those and there you will find Tim Mattson from IBM corporation has prepared hands on tutorial for OpenMP. This is very useful tutorial for learning OpenMP.

As I said, it is like learning a language if not a language, but it is like learning a paradigm of programming and only through the video lectures you cannot learn this. You have to get your hands dirty. You have to pick up some of the problems, best thing what I can show you is, how to write the parallel programs, but you have to pick up some of the problems and if you are actually interested in learning parallel programming, you have to start writing your own parallel program on that problem.

While doing that you will of course, face some of these difficulties which I bet I cannot clear through the video lectures. So, you have to go to an exercise of self-learning and some of the resources like Tim Mattson's tutorial and some more which I will mention later in this this lecture will be very important .

(Refer Slide Time: 16:57)

**Compilation and execution of OpenMP program**

**Installation:** GNU (gcc) provides support for OpenMP starting from its version 4.2.0. So if the system has gcc compiler with the version higher than 4.2.0, then it must have OpenMP features configured with it. OpenMP runs in C/C++/Fortran programs and also works with commercial compilers like Intel and PGI. For more details visit [www.openmp.org](http://www.openmp.org)

**Compilation**

Compiler	OpenMP compiler option	Default number of threads (if OMP_NUM_THREADS not set)
GNU (gcc, g++, gfortran)	-fopenmp	Number of available cores in the SMP
Intel (icc, ifort)	-openmp	Number of available cores in the SMP
Portland Group (pgcc, pgCC, pgf77, pgf90)	-mp	one thread

It is important to set the number of threads before execution of OpenMP codes

IIT Kharagpur

So, what becomes the next important step is , how to install OpenMP, if you have to say if you have to self-run OpenMP, if you have to run OpenMP programs, you need a platform where you can compile the OpenMP jobs. So, important part is installation, fortunately if you have GNU compiler or gcc compilers for C/ Fortran you already have OpenMP ,in case this G N U version is greater than 4.2.0.

So, starting from version 4.2.0, GNU has started including OpenMP in the in the gcc list. So, if you install gcc, automatically OpenMP is installed there, you can compile OpenMP programs. So, gcc version of 4.2.0 or higher must have OpenMP features configured in it, and but it has to be only with C, C ++, Fortran.

These are the three languages for which OpenMP support is given and why is these three languages? Because the users who require OpenMP which is mostly for some of the domains in which multi-threaded parallelization ,will find importance, they are mostly conversant with these three languages and most of the programs for scientific computing ,for engineering applications and also so many-many for economics or finance calculations are developed in C , C++ and Fortran.

Keeping that that in mind OpenMP.org ,the OpenMP annual review board or the vendors who came and formed the consortium for OpenMP made OpenMP supportable for only these three languages and if you have GNU, I said OpenMP is already there.



If you want to use it with some other compiler like Intel compiler or PGI compiler you can go to OpenMP dot org and see how to install it for the other compilers. OpenMP is free. So, if you have a commercial compiler like PGI, on top of that you can install OpenMP. Availability of an OpenMP that does not require paying from your site, it is a free software and it can be used with any other compiler .

So, for compilation you have your OpenMP program you use gcc, you have to write gcc the program name and -fopenmp is the compiler option. So, if you write gcc a.c -fopenmp will get an executable which is a thread parallelized OpenMP executable.

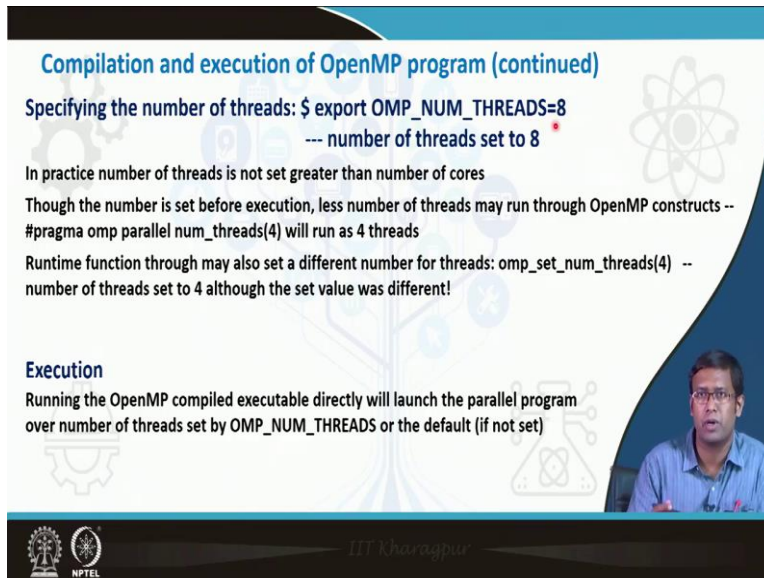
And, if you run this executable simply, it will run on the number of available cores in the SMP. In the SMP or symmetric multiprocessor or the computer in which you are using which is multiple processors connected with a same common address space, the number of processors will be the number of threads, if you just compile a program OpenMP program with fopenmp and run the executable.

If you want to launch some different number of threads and this is what an environment variable that how many threads will be active, you have to set OMP\_num\_threads externally or inside the program, in that case you can run it in different number of threads. If you use Intel compiler the compiler option is -openmp and it will again, if you do not set the environment variable, by default it will go to the number of cores of the SMP.

If you are using Portland groups compiler pgcc or pgf90, etcetera then the compiler option is -mp and it will be by default launched in one thread. So, it will be essentially a sequential program, if you have to launch it into multiple threads, the program is same, if you execute it just if you simply run the executable it will run as a sequential program, because it will go to launch only one thread.

If you have to launch it in multiple threads, you have to again use this environment variable, I will show you in a while and set the numbers. It is important to set the numbers of threads before execution of OpenMP codes or also you can change the number of threads while executing the OpenMP code.

(Refer Slide Time: 21:18)



**Compilation and execution of OpenMP program (continued)**

Specifying the number of threads: `$ export OMP_NUM_THREADS=8`  
--- number of threads set to 8

In practice number of threads is not set greater than number of cores  
Though the number is set before execution, less number of threads may run through OpenMP constructs --  
`#pragma omp parallel num_threads(4)` will run as 4 threads  
Runtime function through may also set a different number for threads: `omp_set_num_threads(4)` --  
number of threads set to 4 although the set value was different!

**Execution**  
Running the OpenMP compiled executable directly will launch the parallel program  
over number of threads set by `OMP_NUM_THREADS` or the default (if not set)

NPTEL IIT Kharagpur

So, it is important to set the number of threads. If you are using a Linux terminal the command `export OMP_NUM_THREADS = 8` will set the number of threads. Now, if you run your executable it will go to 8 processors.

So, build your executable using the compiler command `gcc program name and the compiler option -fopenmp`, you will build an executable. If you simply run the executable in a quad core machine, it will run in 4 processors, but if you set its `OMP_NUM_THREADS= 8`, it will launch 8 threads. As earlier it does run on 4 processor, it was if you run on a quad core machine it will by default run 4 threads, if you set this number to 8 the number of threads will be set to 8.

In practice, the number of threads is not greater than the number of cores. Why then there will be load balancing, there will be threads distributed in different processors, different threads will be distributed in different processor, some will get less, some will get more .

So, in practice we see how many cores are available and number of threads is not greater than number of cores. It is rather a better practice to set the number of threads, less than the number of cores, because some of the cores will be always active on some other job. So, just for running OS some core is active, for some of the graphics interface some core is active, etcetera

So, if you have an 8-core system, if you run the job in 6 cores, then these 6 cores are probably free if you are not running any other job. The other two cores are going into something else graphics or internets etc. These 6 cores are free and you can best utilize these 6 cores, if you run in all 8 cores some core is already doing some other work, so it will be slower there will be some job division ,some run time division in between your OpenMP program and some external program in that core.

So, one core will be slower than others; 7 cores are faster therefore, there will be some latency in that. So, best is to run in number of cores which is little less than the total number of cores available in the CPU. Though the number is set before the execution like by the environment variable export NUM\_THREADS is equal to 8 or if you do not set it is set to the default of the compiler, number of cores available for Intel and gcc and the thread number is set 1 for pgi.

So, though it is set before the execution either by export NUM\_THREADS or either by this or whatever is set by the compiler, but it is also possible to run some instances at less number of threads, and that can be done by writing ( this is an OpenMP directive after the directive you specify) num\_threads is equal to 4 then this job will go and run in 4 threads. I will show you an example that you can change the number of threads while executing the OpenMP job.

Runtime functions, though may also set as different number of threads like you can do while executing ;you can set omp\_set\_num\_thread =4 you can write this runtime function and that means the next parallel part will be executed in 4 processors unless, you change the numbers. Execution; running the OpenMP compiled executable directly will launch the parallel program.

For executing the OpenMP parallel program you have to do nothing just run the executable. Before that we believe you have set the number of threads, if it is not specified within the program. So, programmer can specify it within the program or the person who is applying it, with the application person can set the number of threads by export NUM variable

So, if it should be already set or it is a default and then if you just run the executable it will launch, the parallel program will be launched and wherever is the threaded part in the parallel program, wherever you have given some constructs like parallel and, they

will be launched in the variable that has already been set or in the default number of variables. So, we will see it as an example.

(Refer Slide Time: 26:10)

**Sample OpenMP program**

**Hello World program**

In c:

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

int main(int argc, char* argv[])
{
    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        printf("Hello World from thread id %d\n", id);
    }
}
```

OpenMP header file

Directive for parallel region with default no. of threads

Runtime library function, returns thread id

Parallel region ends

**Sample output**

```
(sommath@localhost OMP_Programs)$ export OMP_NUM_THREADS=8
(sommath@localhost OMP_Programs)$ ./a.out
Hello World from thread id 1
Hello World from thread id 3
Hello World from thread id 7
Hello World from thread id 5
Hello World from thread id 6
Hello World from thread id 2
Hello World from thread id 0
Hello World from thread id 4

(sommath@localhost OMP_Programs)$ export OMP_NUM_THREADS=5
(sommath@localhost OMP_Programs)$ ./a.out
Hello World from thread id 3
Hello World from thread id 0
Hello World from thread id 2
Hello World from thread id 4
Hello World from thread id 1
```

IIT Kharagpur

NPTEL

So, and we come to our first OpenMP program which is a Hello World program. Let us see what happens here it is a c program include starts with include omp.h, this is the header file required in any c OpenMP program. This specifies that this OpenMP program and links all the OpenMP libraries. Inside which the stdio.h, stdlib .h which is standard header files. int main (int argc, char\* argv ) and then we write the pragma omp parallel. This is a OpenMP directive, it shows that the next regime must be omp parallel regime and run in multiple threads. There will be multiple threads launched here.

So, write int id= omp\_get \_num\_threads(), printf (Hello World from thread id ); so, id which is the number of the thread number and it will write Hello World from this number of threads. This is a simple program you can write this c program, just copy it and run it in your computer if you have gcc.

So, how to run it, we will see how to run it. This is the OpenMP header file this is the directive for parallel region. So, whatever is within the brace will be run as multiple threads. What will be the number of threads? The number of threads is what is set by the num set variable by the environment variable.

Here to here is the parallel region and after this the parallel region ends, and the if you write anything after ,that will be a sequential operation, only one thread will be active here, multiple threads will not be active.

So, after this parallel the next place will be the parallel part of the program, multiple threads will be launched here. This is a runtime library functions we are calling about runtime libraries. During operation for each thread it will return, what is the id of this thread and this through the OpenMP runtime delivery functions will return the thread id.

So, if we execute it, export OMP\_NUM\_THREADS=8 ,we said the number of threads is 8 and then write a.out it will write Hello World from thread id and from different threads from 1, 0 to 7 there are 8 threads. So, from 0 to 7 till each thread will write Hello World. So, though only one printf option is given here there will be multiple threads each will write execute their own printf option.

If we change the number of threads, here it is 8, so 0 to 7; the first thread is thread id 0, the last thread is thread id 7, thread id is always starts from 0. So, if you run on 5 plus threads the thread id will be 0 to 5-1, 4. So, it will run from 0,1, 2, 3, 4 and we can see that the output of this threads are not in ascending order, because it is not a do loop, it is a parallel loop different processor are writing this.

So, different processors may have different latency to the io system in the screen and they can write it on their own; it cannot be synchronized very easily . There are some ways to synchronize it, but the if you actually use your parallel environment, utilize your parallel environment they will be written in some unknown order in some random order, because different processors are trying to write it, it is not synchronized as of now.

(Refer Slide Time: 30:20)

**Sample OpenMP program (continued)**

**Hello World program**

1. The outputs are not written sequentially
2. Can this code be run as threads less than the set number of threads?

If nothing is specified, number of threads is the default or set by export before execution of the program. Else, number of threads can be specified by (i) runtime function `omp_set_num_threads()` or by a clause of parallel construct directive

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

int main(int argc, char* argv[])
{
    #pragma omp parallel num_threads(4)
    {
        int id = omp_get_thread_num();
        printf("Hello World from thread id %d \n", id);
    }
    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        printf("Again Hello World from thread id %d \n", id);
    }
}
```

Parallel directive with clause: No. of threads set as 4

No. of threads set as default

```
(sommath@localhost OMP_Programs)$ export OMP_NUM_THREADS=8
(sommath@localhost OMP_Programs)$ ./a.out
Hello World from thread id 0
Hello World from thread id 3
Hello World from thread id 2
Hello World from thread id 1
Again Hello World from thread id 1
Again Hello World from thread id 0
Again Hello World from thread id 2
Again Hello World from thread id 4
Again Hello World from thread id 5
Again Hello World from thread id 6
Again Hello World from thread id 7
Again Hello World from thread id 3
```

one parallel loop ends, the other starts

So, we continue the Hello World program, as I said the outputs are not written sequentially, it is not one by one, they in some random order the threads have written their output and each thread is been executed in different processor, if we use multiprocessor system.

Can this code be run as threads less than the set number of threads? We have noticed that before running the threads we have executed the OpenMP library environment call set num threads is `omp_set_num_threads` is equal to 8 or 5; can we run it in different number of threads. We can see a simple example if nothing specified ,number of threads is the default or set by export before execution of the program, else the number of threads can be specified by runtime function `omp_set_num_threads` or by a clause. So, either by this function or by a clause parallel construct directive.

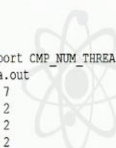
So, when you wrote that this section is parallel; `omp parallel` , we can also write the number of threads in a bracket and then this can be changed. So, programmer can change the number of threads. We will see the example that here we wrote `omp parallel num_threads 4`. That means, this parallel regime will be launched only in four threads and they later we wrote `omp parallel int id omp_get_num_threads`, but there is no clause is given after parallel.

So, what is the default number of threads, it will run on that number of threads. So, if we run this program ;this part will run in 4 threads, because the number of threads is set as 4.

After that, it will launch another parallel regime and again Hello World and this because we have not, we have set it as default and the default number of threads is 8 which is set by `omp_num_threads`, this will go to 0 to 7, this will go in 8 parts. Well interestingly we can see that one parallel part ends here which is in four processors, because we have given the clause `num threads 4`. The other parallel part starts and ends here. Therefore, there is one set of parallel part if we write any, if we put anything here that will not be a parallel part that will be executed as a sequential part and there will be another set of parallel part.

(Refer Slide Time: 33:54)

## Sample OpenMP program- Fortran



```

program Hello World
integer omp_get_thread_num
!$omp parallel
    id = omp_get_thread_num()
    write(*,*)'Hello World from thread =',id
!$omp end parallel
end

```

Fortran default i-n integer, rest real.

### Output

```

[somnath@localhost OMP_Programs]$ export OMP_NUM_THREADS=8
[somnath@localhost OMP_Programs]$ ./a.out
Hello World from thread = 7
Hello World from thread = 2
Hello World from thread = 2
Hello World from thread = 2
Hello World from thread = 2
Hello World from thread = 2
Hello World from thread = 2
Hello World from thread = 2

```

Variable id must be different for different variables, but being a s shared memory machine, Fortran treats it as a common shared valued Variable- Solution: declaring it as private to each thread. C does it by default for all variable used only in the parallel part

```

program Hello World
integer omp_get_thread_num
!$omp parallel Private(id)
    id = omp_get_thread_num()
    write(*,*)'Hello World from thread =',id
!$omp end parallel
end


```

### Output

```

[somnath@localhost OMP_Programs]$ ./a.out
Hello World from thread = 2
Hello World from thread = 5
Hello World from thread = 7
Hello World from thread = 1
Hello World from thread = 3
Hello World from thread = 6
Hello World from thread = 0
Hello World from thread = 4

```



## IIT Kharagpur

If we take a Fortran program; in Fortran; the program Hello World is written. Interestingly in Fortran we do not have to include any header file. A parallel loop in Fortran the braces are not there. So, parallel loop will start with `omp parallel` and it will end as `omp end parallel` and these are not exactly Fortran commands, so this started with exclamations which are comments in Fortran.

So, if you do not use the OpenMP compiler option then this will be treated as comments, it will not be a parallel part, it will be executed as sequential program, if you add it then it will be parallelized. And, in Fortran by default integers are given i,j,k,l,m,n, any variable starts with these letters are by default integer, but `omp_get_num_threads` returns as integer value, so it has to be externally specified as integer, it is not a by default integer, that is what is Fortran's semantics.

So, by default it is real, so we have to externally specify it as integer. It is `set export NUM_THREADS` is equal to 8 and when we ran it we saw that Hello World, it should run on 8 plus threads therefore, we will get Hello World from thread and this is thread id which is obtained by `omp_get_num_thread` that will give the number of the thread, so this will be 0 to 7; however, you see 7 and rest all are 2.

So, what we are seeing in C, it was coming 0 to 7, but here in Fortran it did not come 0 to 7. So, there must be something wrong in the program. This is very interesting, we will look into it in the next lecture, that variable the id which is writing id, it is a shared memory system. So, this id is being shared ,is in written in the common global address space. So, when each thread is trying to write id, it is writing the id, variable id which resides in the shared memory. Therefore, the value id must be same for all the threads.

So, what happened somewhere the last thread was active it got its own value id. Then the third thread which is thread number 2 is active and it wrote the value id to the shared memory location. The other threads while writing their the value of the id, that got the value id which is obtained by thread number 2; the third thread, because this is a shared memory variable, but it should have their own thread id. So, this thread id is also being shared and each one is writing the same thread id.

So, variable id must be different for different threads; however; its being a shared memory system if one thread writes something another threads might pick up the same value which is written by other threads, not its own value.



So, the solution will be declaring it is a private to thread. Instead of being a shared variable, id should be defined on top of the regular definition of the variables id, apart from that defined separately as a shared variable, as a private variable that. It is not shared; each thread has its own copy of the variable id.

In C anything used within the parallel loop not used outside the parallel loop is by default a private variable, in Fortran it is not. But it is a good practice to see which variables are private, because when the threads operate, they execute the same line for each. So, you cannot write for a thread id 1 its id 1, thread id 2 its id 2 so on.

It is the same variable id which all the threads will write, but in a shared memory system they will essentially end up in writing same value if not for one, but in generally they will essentially end up in writing same value for all the processors.

So, you have to define parallel private id, id will become a private variable to all the processors, and the processors will write Hello World from id. This id is parallel so each processor or each thread has its own value of id, and then you will get the variables 0 to 7.

So, though C because id is not called outside the parallel loop it. In C it will behave as a private variable, but in Fortran it is not so, but it is also a very good practice to identify the private variables. There are few in number, what are the private variables? The variables whose value will be different for different threads, and specify as the clause to the parallel regime that these variables are the private variables. Well we will look into some more important aspects of OpenMP program in next class.

Thanks.