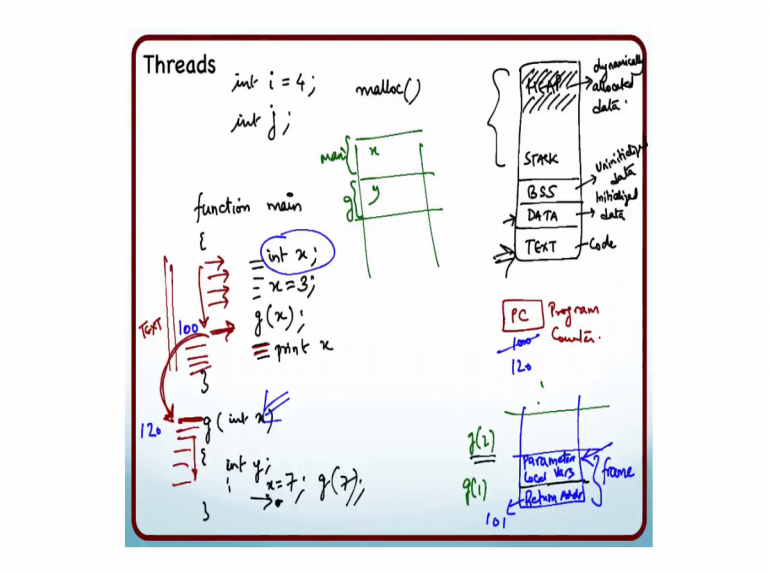
Introduction to Parallel Programming in OpenMP
Dr. Yogish Sabharwal
Department of Computer Science & Engineering
Indian Institute of Technology, Delhi

Lecture – 12
Program with Single thread

(Refer Slide Time: 00:02)



So, what are threads to understand threads, we will have to get a little bit in to operating systems. This will be important to really understand how parallel open mp program is work right shared memory programs work when you run a code there is some memory that it can access right what is that memory. So, the memory of a program is divided into various regions what are these regions. So, one is that text region, right, this is essentially where all your code resides after that there is the data segment and this essentially contains your initialized data.

So, what is initialized data? So, this is global variables that you may have defined and you have initialized. So, if you globally declare some variable int i and initialize it equal to 4, then this will actually be present in the data segment that is where it will be kept the point here is that the value 4 that you have specified in your code has to be stored somewhere. So, it stored in the data segment after this you have the BSS segment. This is essentially the base storage segment and this has your global uninitialized data.

So, if you have declared some variable int j and have not initialized it; so, we are talking about global variables over here. So, they could be file scope variables or static variables. So, all of these global data kept in the BSS the base storage segment and then the remaining memory is divided into 2 important data structures one is the stack and the other is the heap. So, let us understand heap first heaps essentially stories all the dynamically allocated data. So, whenever you use malloc or calloc or there are number of calls which allocate data dynamically whenever you use any of those calls that data essentially comes out of the heap as you allocate more and more data the heap keeps on growing and then you free data that spaces returned back to the heap.

So, what is this stack; let us say you have a function you have a function main, right and there are some instructions over here and after that lets say it call the function g and it call that with a parameter x; x is something it has declared over here, let us say int x and then it execute some more code and then you have function g here which takes a parameter int x and there is some code setting over here. Where is this code kept; this code is kept in the text segment; what the system does is it maintains something called a program counter. So, all of this is essentially residing in the text segment, right and it maintains a pointer to the current instruction that is being executed, right and as it executes the instruction it keeps on shifting this pointer.

Now what happens when it comes to the function g.

Student: (Refer Time: 03:27)

It is being executing these codes sequentially and now suddenly, it has to go somewhere else, it has to go to this code, these are instructions following in this main function and then maybe are the code for function g. So, it has to jump from this location to this location, right. So, that is done using some branch instructions. So, that causes the program counter to change and get the value which is the location where the function g starts from.

Student: So, do this jumping of the program counter cost time.

It costs very little time you are say best cost time, but very little time.

Student: we are doing to (Refer Time: 04:08) something there nearly programming (Refer Time: 04:17).

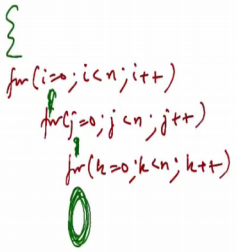
Yeah. So, its gets a tradeoff of code maintenance you says performance typically you should not have very large functions that become very difficult to maintain you should have small functions you should just know when you need when you need a function call when you do not; there are things like inline functions and you know macros that that you can define. So, inline functions are basically just substituted at that place and macros are also substituted, right.

So, they do not cause any branch instructions to happen in does a trade of between productivity and performance putting all your code and one function is going to be a nightmare to maintain your going to a income to lots of bugs.

Student: is the small performance that is it tradeoff (Refer Time: 05:11) absolutely care (Refer Time: 05:13).

It depends how much performance you are looking for; what are you trying to code up for, right. So, you remember what a matrix multiply looks like, right.

(Refer Slide Time: 05:24)



So, you have for i is equal to 0 i is less than n i plus plus, right for j equal to 0 j is less than n j plus plus for k equal to 0 k is less than n k plus plus, right, I am just assuming a simple case of n cross n times n cross n matrices multiplication, right.

If I have function calls over here, right, I would not care, right, I would not care trying to replace them if I have function calls over here I would not care if I have function calls over here I make care a little bit, but not of full lot, right, what I will definitely avoid is having function calls here this is the code that is getting executed the maximum number of times this is the innermost loop, right, this is executed n cube times this code is executed n square times this is executed only n times, right.

So, simple logic that I would try to make this code as efficient as possible and not bother to much about this; so, typically what you do is you use profilers and you figure out where your bottlenecks are and we just concentrate on that piece of code which is the most time consuming and its typically a small piece of code sitting inside some number of loops and you try to make that as efficient as possible and do not bother about there are should be go that is typically what do it, all right.

Let us come back. So, we were talking about that the main function is the executing and then it will count as the function call to g and now I have to change the value of the program counter program counter is a register which basically keeps stack of the current instruction being executed right. So, what do I do? Now I have to change the value of program counter, I am putting some values over here. So, let us say; this is location number 100 and this is location number 120, right. So, right now, the program counter has location 100 and I am just going to replace that with location 120, right.

So, now, what happens? I execute this function g, I execute all the instructions and once I am done, what happens; I am suppose to go back to 101, but I have replace the value of the program counter, right. So, I lost that value.

Student: (Refer Time: 07:43)

So, what is very important is that whenever i make a function call i store the value of the.

Student: hum

Return address, right; that is typically what goes into the stack. So, whenever a function call is made right there is a frame that is created in the stack and what do I store in this frame the first thing, I store is the return address this case, I will store the value 101 over here.

And then I change the value of program counter and then I start executing instructions from there, let us say that this function g, right, it has a variable int y. So, this y means to resides somewhere in the memory right and as long as I have not entered the function g this y variable does not exist. So, I do not need to have any memory location for it, right it is only when I come into the function g that space needs to be created for this variable y and I know that when this function is going to complete its execution, I would need this variable anymore, right. So, I am going to throw with this variable.

So, since I have created this stack frame; what is the life of this frame? The life of this frame is while the function g is executed; right once function g is done, I do not need this return address anymore; I am going to throw away this frame, right. So, what I am going to do is I am also going to store local parameters on to the stack, right.

So, whenever I invoke a function whatever variables are defined inside that function all those variables go into the stack memory for them is allocated on the stack in that particular stack frame, why cannot I have global space allocated for them. So, I could have created a space for function g, I could have created a space for function mean and I could have just storied by over here and you know whatever x over here and so on, right, I could have just done this do you see any problem with that.

Student: (Refer Time: 09:57) not needed

Student: (Refer Time: 10:04)

I am to leave with that load any other problem.

Student: So, if there are more variables then the number of registers.

So, variables I am not stored in the registers variables are stored in the memory space is created in the memory only when some computation needs to be not done on them at that time they are faced into the registers computation is performed and then that put back into the memory, right that is why the cache and everything comes in, but otherwise space is allocated in the memory in the main memory not in the registers.

So, I can define as many variables as I want irrespective of the number of registers; I have, yeah.

Student: we also have take care of that variable names in different function (Refer Time: 10:46).

Once you compiled your program, names have no meaning it says memory locations.

Student: hmm

Once the compiler has compiled a code in created the object code it only keeps stack of memory locations, it is going to replace this name y with 1002 wherever it occurs.

Student: (Refer Time: 11:07).

Exactly;

Student: recursion.

Recursion; so, in recursion, what happens you call the same function over and over and over again right now how can you allocate one space for g, I do not know how many times g is going to get invoked because early, right, I just do not know and you could write programs you have the same function gets invoked 1000 times, if I mean, I have return a recursive program to calculate factorial and I am call the factorial 1000 to get invoke 1000 times.

Student: (Refer Time: 11:36)

So, in this case, if we put it on the stack; so, every time the function g gets invoked and new stack frame gets created. So, this is for the first invocation of g 1 another for the second invocation of g 2 and so on. So, there will be one stack frame created for every invocation of the function and I will have a different copy of the variable what else goes into the stack let me explain by an example.

So, let us say I have set the value of x 2 3 and then I am worked g of x and now inside the function g, what I do is I set x equal to 7. Now when I come back and I print, I am not writing the full print command, but let us say I have print x; what is going to get printed.

Student: (Refer Time: 12:23).

3 or 7?

Student: 3.

3, what that means, is that whenever a function gets invoked a separate copy of the parameter is created. So, again where is this copy created is created on the stack this should not be local Params; this should be local variables, right.

So, what is the life of the parameters this copy int x created over here only exists during the life of g right when the function g ends we do not need this space anymore. So, when I am setting the value of x equal to 7 over here, it is actually modifying the value of the location which is in the stack over here right not the location wherever this was allocated. So, this is pretty much for goes into this stack, right.

So, now if this function g again mix a called to g again, right with let us say value 7 or something, then another stack frame of g will be created, right and these 2 stack frames are completely independent of each other of both of them are the functions g, but their independent of each other, right because now what will I do if I am again calling g over here. I will be storing the return address which is the next instruction after this, right because that is why I am suppose to return after the second invocation of g, right and then I will create another copy of int x and so on and local variables. So, all of that will be done on this stack frame.