Parallel Computer Architecture

Hemangee K. Kapoor
Department of Computer Science and Engineering
Indian Institute of Technology Guwahati

Week - 02
Lecture - 11

Lec 11: Examples

Hello everyone, we are doing module 1 Introduction to parallel architectures. In today's lecture, we are going to quickly have a look of few examples about how can we go about writing parallel programs. That being not the focus of the subject, we will just get a glimpse of how that is done so that you understand how everything falls into place. Okay. So we have been using some of the array elements as the running example throughout this introduction module. So I will use the same one to explain things here. So you have example of sum of array elements, we have an array A of big size and we need to add the elements into a final sum. Okay.

So if you want to write a sequential program, how will you write that? You will have a for loop from 0 to max size, that is the array size and you will sequentially add the values. Okay. So we will declare a local sum variable to be 0 and this is the array which we are going to add and there is a for loop. So this loop is going to sequentially sum from i equal to 0 to max size and eventually it is going to print the answer. Right? So this is a sequential program or pseudo code for the sequential method of adding the array elements. Okay.

Now we will move and understand how can I do this in parallel. Okay. We have seen two parallel paradigms shared memory and message passing. So we will first try to write a program using the shared memory paradigm. Okay. Our shared memory paradigm says we need to share some variables across the processes and we have multiple threads of processes to compute this final answer. Okay. So I will first need to decide how many threads or how many processes I have for doing this operation then we will have to divide the work across all of them and then once the local work of every process is done we need to combine it.

So that is the overall logic. Okay. So let us start by first dividing the array. This is the array A and I am going to divide it into small pieces to the number of processes. So how will we do this? I have the array size from 0 to max size. Right. So this was the max size and suppose nproc is the variable which tells me the number of processes I have for doing this job. Okay.

So each process will get how much data to compute. So each process will get the $MAX\ size/nproc$. So this is the amount of data each process will have to compute. So 0, the first process starts from 0 and goes up to suppose let me call this as a small variable w, Okay. So for work and the first process works on the first w elements. The second one starts from $w + 1$ and goes up to 2w and so on. Okay.

So this is how we are going to partition the data and each process will then have a for loop on this construct, for loop on this much amount, for loop on this much amount and so on. So we have parallel for loops running, so parallel programs will run and what will they do? Each of those for loops will look similar to this one in a sequential program, but it will not add to this sum variable, it will add to its local sum variable. Okay. So what is shared information inside this? So what is shared? My complete array A is shared, the big array because each process can go to these individual partitions and is able to read the data within its partition, hence this array is shared. In addition, the final answer sum is also shared. Because every process is supposed to come and add its partial sum inside this final sum. And what are the private variables if at all a process needs? So every process will have its local sum, so I will say mysum. Okay.

So my sum is a private variable for every process and maybe some internal loop variables and so on. Alright. So this first process is going to compute mysum of 0, this next process will compute mysum of 1 and so on. I am just putting the suffix to understand which process it comes from. Alright. Once this mysum is computed, it needs to be added to the final sum. Okay. So once this for loop is done, once you come out of the sequential for loop in the individual process, we need to take the my sum and add to this global variable.

So this global variable sum which is shared, it is in the shared memory, it is stored somewhere and each process is going to come here with its mysum and add it. Okay. So there are several such mysums coming which will want to add the value to this variable. Do you see a problem in this? Or rather we need to take care of something because if multiple of those come and when I want to add the mysum to this global variable sum, it might happen that first I need to read the sum, then we need to add and then we need to write. Okay. So first this will be read out from the shared memory, the local sum will be added to the sum and then the final answer will be written. So in case it happens that all of these parallel programs read the sum variable maybe simultaneously or one after the other, they will have their individual answers of sums and the one who ends up writing the value in the last or in the end to the main memory would be the final answer which may or may not be correct because we are not controlling the order in which the sum is accessed and probably multiple processes are accessing sum and reading it in parallel, modifying independently and then writing it in parallel or one by one they will write but

the sums are computed in parallel.

So how do I guarantee correctness of this? Right. So for this we need to do something more. So I hope till here it is clear, we have taken the shared data, we have distributed the data across the processes, every process computes the local sum in a for loop, once it is done it has to add it to the global variable. Now for the correctness of the final answer, we need to make sure that every process adds the sum one at a time, that is if one process is accessing sum and modifying it the other process should not. So what is this thing called? If you can recollect your operating system scores, this is called changing a variable within a critical section. Okay. So we need to establish a critical section in which the sum variable will get changed. Okay.

So we need a critical section in which I am going to update the sum variable and to establish a critical section what do we need normally? We need a lock so that we lock, we perform the action and then we unlock. So this is how a critical section works, you first lock the lock variable, change the variable sum and then unlock the lock variable. So we need to do this. So addition of mysum to the sum will happen in this blue portion within the critical section. Okay. So this will guarantee that the final sum will be really the correct value. Okay.

Then in the sequential program if you see I have initialized the sum variable to 0, this has to be initialized to 0 because it will have some garbage value in the beginning and if we do a cumulative addition you might get a wrong answer. So when I divide this program into multiple processes is it guaranteed that the sum is initialized to 0 because this is running in one process and the other processes are simply doing the for loop onwards work. Alright. So is $sum = 0$ completed or not we do not know. So for guaranteeing this what we could do is we shall start adding the sum value only when every process has finished or every thread has finished because that will guarantee that the first thread which was making $sum = 0$ has also finished executing and sum is really initialized to 0. Okay. So this way we can guarantee sum has become 0 and then we can add it. Okay.

So for this what should we do? We want to make sure that every thread or every process has finished before we enter this critical section. This is done using a new construct called barrier that is called barrier synchronization. So there is a variable which I will use with this synchronization construct which guarantees that when I insert a barrier in the program all the threads have to wait for every other thread to finish before they can cross the barrier. So even if a thread finishes as soon as it encounters the barrier instruction it waits until other parallel threads have finished and they have reached the barrier. So at this point we can guarantee that every thread has finished post this action

we will enter the critical section, do the summation of the variables mysum equal to this and eventually print the output.

So with this explanation let us see the program. So this is a neater way of writing the same thing. I will just explain pieces of code here. So I would call this pseudocode because some abstraction is used in the coding. So this part is the shared variables. Okay.

So the sum is a shared variable, array is the shared variable then the lock variable used for critical section and the join variable used for the barrier synchronization. So these four are my shared variables. Now what happens within every program? Each program or each thread has a mysum variable. It has a range of values it is supposed to compute on. Right? We divided the array A into small pieces of data.

So each process has a low and a high index in that big array which it is supposed to add. So this low and high are those ranges. And this is just some expression which helps you to find out what is the low variable depending on the process ID. So if you have 0 to n, 0 to nproc number of processes you can quickly do the maths and find out how do you identify low and high of every process. Okay. So this low and high are algebraically identified using this formula.

Once you have this low and high each process is going to run a for loop from low to high and compute the local sum in my sum. Once this is completed as we discussed we need to wait for every other thread to finish. Hence this barrier keyword is put here on the variable joinVar. So using this barrier synchronization, we guarantee that all threads will complete this instruction and go to the next step only when every other thread has finished. So once we cross the barrier we need to enter the critical section.

So this is the critical section in which we add the sum variable. So for critical section first we need to lock the Lockvar which is a shared variable I am using and then compute the sum and then unlock the variable. Okay. So with this method using barrier and critical section we can compute the sum of array elements in parallel. So that is using a shared memory paradigm. The same explanation is written on this slide so I would not read it through.

So whatever I explain is again written here for your future reference and the barrier and the critical section locks everything can be implemented using read write that is load store and read modify write type of instructions provided by the ISA and we are going to see details of this and implementation of how this can be done in future lectures or future modules of this subject. Okay. Right. So moving on we are going to repeat that exercise using message passing. So if you recollect what is message passing instead of every

process being able to access shared memory it is going to send messages to each other using the send and the receive keywords. Okay. So let us start thinking how you would want to do this work. So you have a big array A and that array was shared in the shared memory program but now I am not permitted to share because my paradigm is message passing.

So there has to be an initiator of doing this. So who will initiate? So let us say I have a master process who initiates this work and this master process is going to have the array. So it is the owner of the array. It chops the array and hands over pieces of this array to different processes. Okay. So the array has to be physically transferred to the other process and not shared across everybody. Okay.

So once every process is given this piece of array each process is again going to run the for loop to compute the local sum and then we need to do something more. Okay. So let us start and try to arrive at a solution. So this is the partitioned array and I am calling process ID equal to 0 as my master process. So this is the master process. It chops this array and gives one array to each definitely keeping one for itself. Right.

So this one is given to PID 1 and so on. 2, 3 and so on. So depending on how many processes you have you equally divide the work and then assign it. So PID 0, this is the master process. What is its job? Its first job is to send the partition to every process.

So it has to send this information to all the PIDs starting from 1 to nproc that is the total number of processes. Okay. So every process, it is going to execute the send system call and this send will be matched by each receiver. So if this is my PID 1 this is the process. This process has to execute a receive system call. Right. It has to execute receive which will essentially copy this.

If you remember the FIFO based synchronization between send and receive so the master is sending this chunk of data. It goes through a channel or through a FIFO and reaches PID 1. Similarly it will send the next chunk of data to PID 2 and so on. Right. So it is going to execute several send calls and every send call what does it contain? It contains the data that is A of 0 to M or whatever. Then the amount of data it is sending, to whom it is sending and the tag because we want to identify that there is a matching send and receive. Okay.

So here we are going to send the data then the size of the data we are transmitting then to whom and the tag ID or the identifier which we were talking about. Okay. So this much information will go in every send call and a matching receive what is it going to do? The receive here is going to receive the data because it is it will pick up this data

which is coming. It knows how much data to pick up, from whom to pick up and then the matching tag ID. So once the tag ID matches it will receive the data. Got it? So using this send and receive we have transmitted the information.

Assume that every process runs its local for loop computes the mysum. Now what happens after it has finished? Once every process finishes mysum there is no facility that they can go to the shared memory and update the sum variable. They need to report back to the master. Now reporting to the master is again using the send and receive pair of communication. Okay.

So let us see what happens in this. So I will say this is within the other compute processes. Right? So compute the local sum and then the local sum needs to be sent to the master. Send the local sum to the master. Again using similar tuple I will give the value which that is the local sums value, the size of that data. If it is an integer that much size of int will come here.

Here you are going to write PID 0 because that is the master process and then the tag. Now remember this tag here and the tag here, so these two tag values are different because they are assigned to different tasks. Okay. So that is the importance of using the tag ID in a send receive protocol. So with this all the PIDs from 1, 2, 3, 4 and so on, they will execute the send and PID 0 will execute the receive. It will receive this information one by one and as it keeps on receiving one information at a time, where is the value coming in? The first variable here.

So this the tuples first entry is the value. So as soon as it receives this value it is going to add it to its local variable sum. So the sum variable is in the master process and the master process here PID 0, it is going to execute the sum. Sum it is going to do this with mysum, but what is this mysum? It is coming from all the sends from the other processes. Okay. So this my sum will have different value coming from every send receive signal.

So I will just run through the pseudocode again. So this is the array, then PID 0 is the master process. The master process first divides the data array into chunks and gives it to the different processes. See this is the send call. What is the send call doing? It takes one piece and this piece size is decided by this arithmetic expression. So using this expression you know what size from which index to give to which process.

So this A[k], so the address is given because we are copying a big chunk of data and it is given to process ID i because our processes start from 1 to nproc. So this is the receiving process ID which is from 1 to nproc. Okay. So the receiver ID is given here

and this is the tag because we are doing some transactions related to data. So I have just used MESG1 as the tag as a sample here.

So with this all send has completed. Then what does every process do? It will compute its local sum. So this local sum is computed here, Okay, till this point. My sum in your range of values compute the my sum. Every process completes this. Once the my sums are completed they need to be sent to the master.

So sending to the master is that if the process ID is not equal to 0 because master has process ID equal to 0. If the current process ID is not equal to 0 only then it is supposed to send. So look at this if and else. This is if that is this is the master. I am not going to go into this loop but I have to go into the else part because once I finish the mysum I directly reach here because this process is not the master process.

It comes and executes a send, sends the value to process ID 0. getting it? Process ID 0 because that is the master process and remember, a new tag is used here because this is a different type of a transaction. Okay. So all the other child processes or support processes whatever you call they are going to execute this send system called sending the local sums. Now what happens at the master? Let us go back to the left hand side.

The master has first sent complete information. So master has executed the send given the work to others. It also has to do its own work. So it does its own work here using its own for loop and once it finishes its for loop, it has to start receiving the answers from others. So it is going to execute this blue part of the code because if it is the master process it has to receive the mysum from every other process and then add it to the global variable sum. So if you see this send and this receive are matching because they share the MESG2 as the tag ID. Right.

So their tag IDs are sent so they are matching send receive calls. So this is what the master does. Compute its local sum and then receive sums from everybody else, add it to the global variable and the other processes what do they do? They have to receive the partition of information, partition of data from process ID 0. From pre ID 0 they receive the data and then run their for loop to compute the mysum and transmit it to the master process. Okay. So this is how using send and receive we are able to do sum of the array elements.

The same explanation is written on the slide. Okay. So with that I think you got an idea how can we actually use shared memory and message passing. So this the current slide is simply going to list some possibilities or some existing methods of doing that. Okay. So parallel programming methods, so this is not an exhaustive list there are many variants

and many new things coming up. So which you can learn in some other subject. But I am just making a list to keep the context in that for distributed and shared memory systems we normally use C and its extensions. Okay. So normally C and its extensions are used for message passing, the MPI interface is very popular it can be used in shared as well as distributed memory systems.

If you remember shared memory system is normally used for centralized memory and distributed where each node has its local memory. Okay. In the global sense everything is a shared memory system but MPI can be applied on both methods. And the implementation is based on libraries, functions and macros. For doing shared memory based communication we can create threads using pthreads that is the pthreads library. Again library support is required for doing this.

Then we have OpenMP and we also have OpenCL for GPU systems. So these are some examples of how you can write parallel programs which come with good amount of documentation and so on. So if you are interested you can try out some things. However as part of the parallel architecture subject we are going to look at the design aspects and not the programming aspects. So overall as we complete this module 1 we have seen a big picture of the parallel architectures and then in the end we conclude that end of the day there is a memory communication, memory to memory copy is going to happen. Parallel programs are going to access shared memory so we need to manage coordination, communication, synchronization, correctness, consistency all this needs to be managed when I have a big shared memory exposed to so many processes. Okay.

So in future topics we are going to see how do we manage multiple levels of memory?, how does copying of data or sharing of data takes place? and so on. So with this we complete the first module introduction to parallel architectures. Thank you.