Parallel Computer Architecture

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

Week - 02
Lecture - 10

Lec 10: Message Passing Paradigm

Hello everyone. We are doing module 1, Introduction to Parallel Architectures. This is lecture number 10. Here, we are looking at the message passing paradigm. Okay. So, inter-process communication can be done using shared memory as in the previous lecture and using message passing that we are going to consider in this lecture. So, what is message passing? Sending a message from one processor to the other and this is not directly possible for the processor to do without the help of the operating system and the compiler because the compiler or the libraries and the OS help or give APIs which enable the process to talk to another process in the form of sending messages.

So, this is similar to sending a letter to your friend. So, if you are sending a letter, first you write the letter, put it in an envelope and then send it across some medium to your friend, but along with that on the envelope, you need to write down the address of the receiver. So, this whole thing has to be done and the process is need not do it itself. So, it takes help from the OS and the library calls for the actual implementation of message passing.

And what is the interconnect here? Any generic global interconnect or the local interconnect, but the there is a tight banding, sorry, there is a tight integration between the sender and the receiver because this is not a generic computer to computer interconnect, but it is a processor to processor interconnect. Hence, there is a tight integration available or required for this type of communication. What are the user level instructions? Simple send and receive calls. Okay. So, a tight integration because it is a processor to processor communication and then done using send and receive system calls. So, let us look at send and receive calls.

Send as the word says, I am going to send, but what are we sending? We are going to send data, from where? From a buffer in the processes local address space. So, this is a process in its local address space, there is some data. So, this data in my buffer, local buffer has to be transmitted. Now transmitted to whom? That is the receiving processes ID. So, we need to tell the ID of that process and mostly this process is on a remote processor. Okay.

So, the send is going to send content of a local buffer to a destination process ID and the matching end is receive where the receiving process has to receive something from a sender process. So, it should know that it is going to receive from so and so process that is the process ID has to be known and with the receive system call, it also has to give a location in its local memory to store the data. Okay. So, send if I repeat, you have to have a local buffer which has to be transmitted and then to whom? Similarly, receive has to specify the sending process and a local buffer into which I am going to collect the received data item. Okay. So, there may be multiple such messages going on the network, process P1 is sending multiple messages and P2 is receiving from P1, but some messages of P1 are not destined for P2 or they are in a different context. So, if I am sending for example, data array, later on I am sending a character array in a different function.

So, depending on the context, the send and receive have to do a matching between each other. Hence, every message needs a tag associated so that we know that the send which is happening is matched with the corresponding receive and hence each of them has an associated identifier or a tag with it. Okay. So, this is required so that the receiver can match correctly the sender and the receiver or rather the messages are correctly mapped. Okay. So, combination of the send and a matching receive. So, we need a matching receive.

This is equally important. This will accomplish a pairwise synchronization between the two. So, these two processes are now pairwise synchronized with each other and what happens at the end of the synchronization? Memory to memory copy happens because the send sends from a local buffer onto this message and the receive receives from this into its local memory. So, a memory to memory copy occurs by this pairwise synchronization. So, send and receive enable us to do this. Okay.

So, here I have depicted a process P. It has the local address space. Everything is local. There is no shared address space here and the pink portion is the data it wants to send to the process Q. And so, the method used is it uses the system call send and it says I am sending the variable X.

So, this is the data. Okay. Data means address of the data. This is address of the data here, whatever data is stored. To whom? This is to whom. So, this goes to process Q and this is some tag that is the identifier.

So, when it executes this, it goes over the interconnect and reaches here. So, process Q also executes a receive call. It has to execute a receive call, otherwise this cannot be completed. What does it say? It says the local buffer into which it is going to receive the

data item. So, it says this is my bucket inside this come and pour your data.

So, that is the bucket inside which the data will come here and who is sending it? So, this is P. So, receive this data only from P and nobody else. So, sender and the tags match. Okay, and then compare the tag. So, the sending tag and the receiving tag should match and the data from process P gets copied into the address space of process Q. Okay. So, that was the basic example.

Now we need to worry about synchronization. Now what is synchronization? There are two processes P and Q talking with each other and a tight synchronization means a send of P should match to the receive of Q. Alright. So, if I am sending suppose, I am sending this pen from here to there, if I just throw this pen, what will happen to it if there is nobody to catch it? It will fall down essentially, this message will get lost. So, what is it? I am throwing an object and I need the receiver to catch that object. Okay. So, if I throw this object, this is the ball I am throwing, there should be somebody to catch this ball.

This is called synchronization. If there is no synchronization, this ball will fall down essentially, the message will be lost. Okay. So, to establish this, there is a requirement that the send from here should exactly match with the receive on the other. That is both the processes should be able to execute send and receive at almost similar times, a very difficult thing to agree, but this is logically you need to establish this. Now, how will you establish this? It is the sender is on one node, receiver is on another node, there is an interconnect in the middle, you are sending this message.

So, this message will eventually reach the receiver and the receiver should be ready to get or receive that information. So, the receiving process should execute the matching receive call. Until then the message is not set to be completed. One could do this using variety of implementations. So, if I move on, I say that a process has completed the send.

So, the send from a process that is this pen which I am sending from here to other process or other person, this event of sending has completed when? When will you say that it has completed? When my other counterpart has captured the pen, I have I have captured that message. Okay. So, the receiver has executed. So, complete transfer of data has happened from the sender to the receiver. The second is the send buffer from the receiver is now available to be reused because the process P was sending a variable X and when it says that I have sent that variable X, now this location is free to be reused or replenished with a newer value. So, send will complete under all these circumstances that the receiver has completed.

My sending buffer is now available for reuse and when is the request accepted? So,

depending on the semantics, we can say that this is the type of synchronization I am expecting between send and receive. Similarly, for the receive, when do I say that the receiver has completed? When it should match with a send message. Right. You cannot receive nothing. So, you need to receive something which somebody is sending. Hence, we need to wait for a matching send to occur.

So, matching send must occur for a receive to complete and then the other option is if the process is busy and it can afford to wait for some time, it could say that I will simply post the receive that is I will say that I want to receive this and parallely I am able to do something else. So, depending on the facility or your design, you can have different ways of synchronization and every variant has got different semantics and different implementations. So, you need to fix what is your semantics and accordingly go for implementation. Message passing has been there earlier. It was used in more formal languages like CSP, communicating sequential processes.

This has been using the send receive synchronized, a tightly synchronized way of communication. Then Occam and OS functions like the socket functions have been using send and receive based synchronization. In such systems, the parallel programs which are talking to each other are very well structured because most of the time they are doing the same work because we will only send and receive information when we are executing almost similar code. So, they are executing almost same amount of works, working on similar data items and the ID that is the process ID for sending and receiving is also kind of known because you have process 0 to process 10 and so you know the process IDs before you can begin. So, at runtime there is no worry to find out who is the receiver process ID. Okay.

So, such parallel systems are very well structured. So, hence the synchronization becomes possible and doable. So, what were the methods of doing this earlier? So, we are worrying about synchronization, when does send finish? when does receive finish? how is it implemented? So, one way of implementing it is a FIFO. If I go back to my diagram, how do I do this? I am throwing the ball in the air and somebody else is going to catch it, but if there is nobody to catch or it is going to take more time or the interconnect is so far that I cannot throw from one end to the other in one go, it has to go through a channel. Okay.

So, I need to establish a channel here. Right. So, this is a person who has to establish a channel. So, the ball goes inside this channel and then eventually this person is going to catch it. So, this is nothing but a FIFO because the messages should go in the order they entered. So, that is the first in first out method which was implemented earlier.

For a FIFO based implementation as you understand the network has to be point to point because this channel which we drew is a fixed channel from one node to the other node. It does not take a detour, it does not go via any other node. It is a fixed point to point network for implementing FIFO. Okay. So, we have been using FIFO to send the data. Here the network topology is important because there has to be a direct connection.

The sender writes the data item into a link or into the FIFO and the receiver reads from that FIFO. FIFOs cannot be very long because if you have more storage, it is going to take more hardware realistic, more time and so on. So, the FIFOs are normally having lesser buffer space and if this buffer space becomes full, the sender has to wait until the receiver has consumed all the data items. Okay. So, FIFOs are small, hence the senders will block after the buffer space finishes and hence this is called a synchronous message passing. So, synchronous message passing because the sender will block when the buffer space finishes. Okay.

So, there is a limitation that you cannot have a lot of pending messages to be sent. Okay. So, to improve on this, the earlier methods also use DMA. Okay. So, before that, let us quickly see how a FIFO is implemented. Here between P and Q, this is one channel to send. So, you can pack some number of balls inside this which will be received and here similarly some items here. Okay.

So, this is this way. So, there is a one directional FIFO from P to Q and another direction, opposite direction FIFO from Q to P. So, that is between two processes. Okay. So, here the network is point to point. Okay. So, P to P network. So, if I want to extend this left hand side design to a bigger network, we can use the right hand side design which is hypercube network.

Here the naming convention of the nodes is that each node is connected to the other which differs by one bit position. So, this if you see this one, the bit position, this one and this one, these bit positions are different. Similarly, when this gets connected to that, the MSB bit is changing. This is connected to this, the bit number from the MSB side bit 0 and bit 1. Okay. So, this is how they are connected and there is a direct link between them.

So, this is the link on which the FIFO can be implemented. This is the FIFO implementation for a bigger network. So, they are bidirectional link where the nodes can communicate with each other, but the disadvantage here is that this node 1 cannot send data to this node. Okay, because there is no direct connection between the two. So, FIFO comes with this limitation. So, we can improve this of course, using different types of networks or topologies. Okay.

So, before that FIFO was allowing you only small amount of data transfer, the senders were blocked. So, to improve on this, the FIFO got replaced with a more versatile and robust DMA that is direct memory access space transfers. So, DMA is a special device, it has comes with a specific controller which is capable of reading data from your memory and transmitting it on an I/O device. So, it copies directly from the memory of the processor without the intervention of the CPU.

So, CPU is not involved in this. So, DMA is in a way non-blocking because the sender can simply say that this is the amount of data I want to send, it initiates the transaction and then the sender process can continue doing its other operations. It can also start another parallel send or it can do some computation because the DMA controller takes charge of sending the data from this node to the other node. So, the sender is in a way free here once it initiates the transaction. Similarly, the receiver also need not be blocking because it need not wait for the data to come. It simply says that I am ready to receive the data that is it initiates the receive by specifying a buffer that you receive this data into this buffer and subsequently the receive can receiver process can continue with other tasks.

So, overall this is a non-blocking communication. And when the DMA is able to transfer data from the sender to the receiver, eventually the data is copied into the receiver's buffer and the receiving process is then intimated about the send receive to be completed. So that was FIFO and DMA based message passing. Okay. So, what were the later developments? Here the FIFO or the DMA both models require that the program be written in that way. If I am dealing with the FIFO, my program should work like that because I need to know where the FIFO, what is the FIFO address, what is the capacity of this FIFO. If it is a DMA, I should invoke correct system calls to initiate the DMA, copy the data or send the data and so on.

So, the programs had to be written in that way. But we want more generic system. We also want to remove the limitation that I should be able to copy data across any node and not only to my physical neighbors. So, the exact point to point physical connection between the two nodes is no longer a limitation. We do not want this to happen. So, how to do this? So, when I want to transfer data across multiple nodes from here to here. Okay.

So, suppose this is my sender S and my receiver is somewhere here. Okay. So, earlier this was not possible, but now this S can send, sender can send to the receiver via multiple other nodes. Correct? So, you could find the shortest path or the best latency path or this less busy path. So, these are the different algorithms which we will not go

into details, but from S to R if you want to go, you might come here, then you might come here, then you might go there. Okay. So, this is one way S can send to R which was earlier not possible in the FIFO based communication.

But with this, we need to modify the system. Why? Because the node, this node 011 0100. This is not the destination. So, this node needs the capability to not consume the data, but forward the data. So, it should be able to forward the data and not eat it itself. Right?

So, this is called a store and forward based communication. So, this intermediate node should be able to store the information and also forward it if that is not the destination. Okay. So, that is the store and forward based communication. If the data is not destined for a particular intermediate node. The downsides of this is of course, I am able to connect any node, but the latency increases because it now depends on the number of hops I am going to travel. Okay.

So, the more hops we cover, the more latency gets added. So, this was a more generic design where we were able to connect from any node to any other node, not necessarily to physical neighbors. It had a disadvantage of the latency which depended on the number of hops. So, overall what do we draw from this? The more emphasis is now on the interconnect topology. So, the better the topology, we can control the latency of transfer. Okay.

So, that was message passing. Now, we will discuss about convergence between the shared memory and the message passing paradigm. Okay. So, as the hardware has evolved, there is the boundary between the hardware and software is getting blurred. So, with this, we are able to do message passing on shared memory architectures and vice versa. So, traditional message passing can now be supported by machines which are designed for shared memory. And how would they do that? Using some shared buffer space because message can be implemented by, you can imagine a FIFO or a DMA.

So, it needs some place to get the message that is the data and take it somewhere. So, this message so called can be mapped to a local address space which is shared across all the processes. So, when we execute a send system call, so what is send? Send sends our data from here to another process. So, this send is nothing but writing the data into a buffer. Okay. User has said that send this data to so and so process. But in the background, the send will be implemented using the write to a shared buffer and what does the receive happen? Receive simply reads the data from the shared storage. Okay. As simple as that.

So, message passing has a send and a receive, you implemented using earlier say using FIFO. Now the send is implemented by write into a shared location and receive is implemented using a read from the shared location. However, because there is no real FIFO, no real connection or channel established, we need to make sure in this case that when the receiver goes to the shared buffer, the data is actually in it. So, for this we need to guarantee that synchronization should be there. Okay. So, we need flags or locks to make sure that the messages have arrived before we actually go and read the data.

So, okay so this was message passing on shared memory systems. The second one is message passing. Sorry. The second one is shared memory on a message passing machine. So, a message passing machine also essentially creates a global address space because end of the day, we all talk through the memory, only thing is the semantics and the abstraction is different, but the address space is shared at the end. So, there is a global address space through which we will communicate. So, when you do a read, the read in a shared address space is realized by sending a message from this processor's address space to that processor's address space. Okay.

So, after the read is done in a shared memory program, it is translated to a send or a receive message depending on the action to be done. And this all is transparent from the user. So, user does not know this. So, given a type of a machine and a different type of a programming model, the compiler or the system make sure that we do the translation of the semantics so that one type is able to run on the other type. So, message passing architectures look very similar to NUMA because yes, sending a message there if the receiver does not happen synchronously or if the interconnect is far, overall your access time is not uniform.

So, it is non-uniform memory access. This network interface which is there in every node, which connects the node to the interconnect is very tightly integrated with the cache and the memory controller. Why? Because either it be shared memory architecture or it be message passing architecture. End of the day, the interconnect, either the send call or the load store, they need to copy the data from the interconnect to the memory. So, there has to be a tight integration between the network interface and either the cache or the main memory controller. Right. So, what does this network interface which is tightly integrated do? It is able to observe the cache misses because it is so close to the cache and the memory controller that it observe the cache misses.

And if this cache miss needs to be served from a remote node, for example, then it has to generate a network transaction to reach the remote node and execute the read or write of the data. And similarly, in a message passing architecture, when the message is coming, is being sent or being received, we are essentially carrying data and bringing

data. Now, this brought in data needs to be stored in the local memory and the network interface should also be able to read from the local memory. Okay. So, this I O device is tightly connected to the memory system so that the data can be directly transferred from the user's address space to the network. Okay. So, overall, if you see that the large scale, that is NUMA based or distributed based shared memory systems are ultimately message passing because within a node, you could do local read, write and load stores.

But when you need to communicate or read write data to a remote node, you have to go through the interconnect and going over the interconnect would require some type of a message passing. So, overall, it is a mixture local nodes can easily do load stores, but remote nodes would require message level transactions to get implemented. Alright. So, convergence is happening between message passing and shared memory. If you scale it up, you can have multiple such nodes or clusters of machines interconnected with each other so that we can build bigger and bigger systems with this convergence. We are talking of say 10 nodes talking with each other using either method, but I can pack these 10 nodes into one cluster and connect several such clusters over a good interconnect and have a bigger bigger system design, okay using more better interconnects. Okay.

So, to summarize, we have seen message passing, we have seen shared memory, we have seen that there is a convergence happening between the two. So, we can have a generic architecture being able to implement both the programming models and these paradigms are very well defined. They have well defined data sharing methods, communication methods and synchronization. So, to implement them, you need specific definitions and very nicely implemented communication and synchronization methods without which you cannot execute correct programs. And the underlying machine is nothing but a set of computers, a communication assist that is the network interface, which is able to connect to a scalable network. Okay.

So, what do you have? You have a compute node, a communication assist or the network interface and plus a scalable network. Okay. So, this is overall what we are interested in. And while we implement this, we need to understand how does the communication assist interact with the network and with the compute node with its memory, how does it affect the cache design, where does the address transition happen and so on. So, there are many aspects related to memory which we need to understand for this whole implementation to work properly. Okay.

So, but we will see that later. So, this finishes this lecture. Thank you. Thank you.