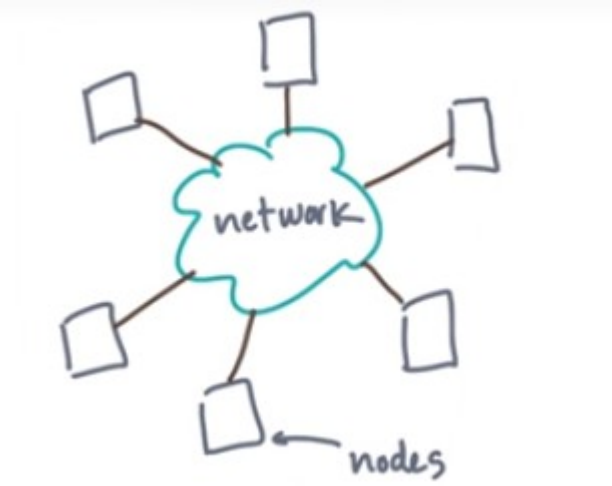
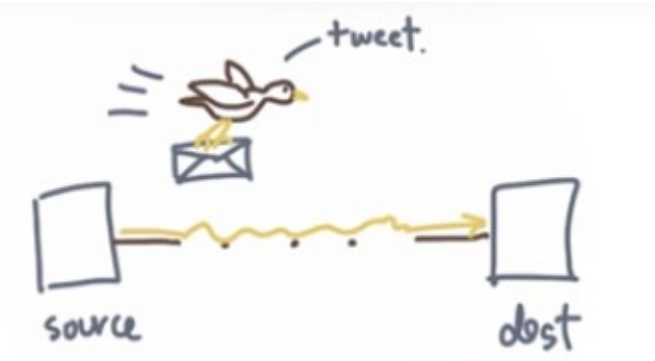
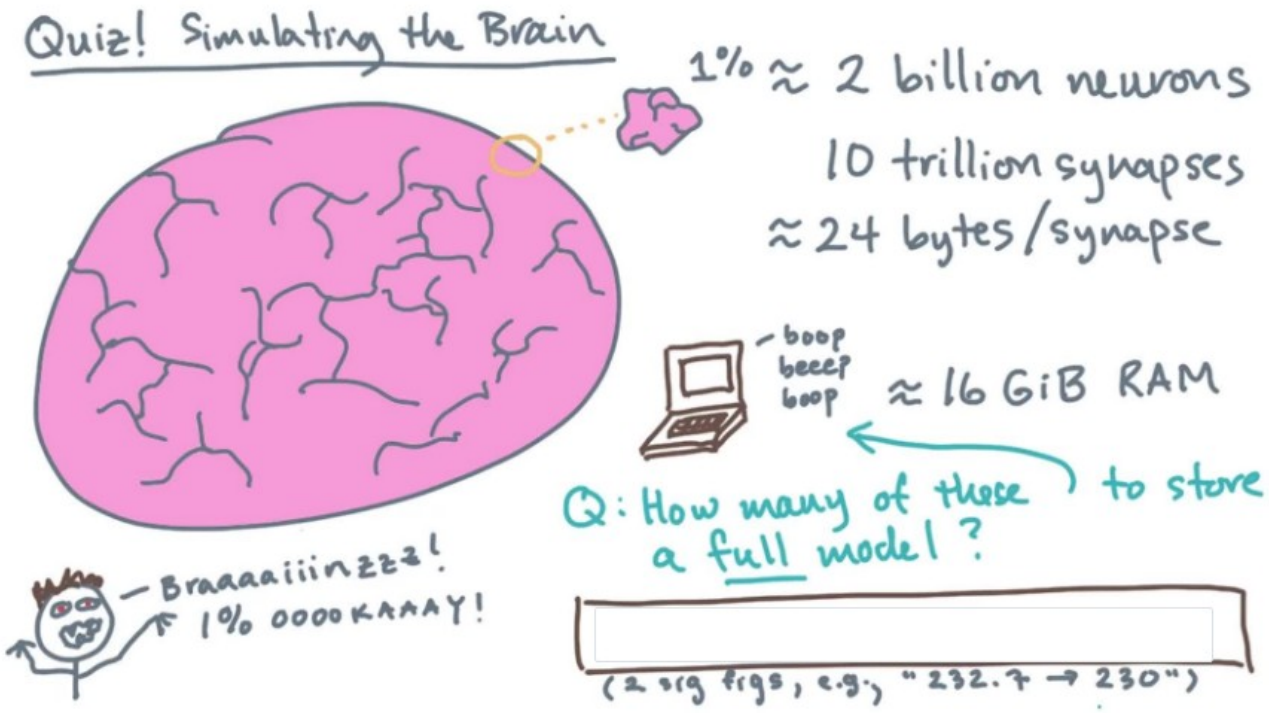
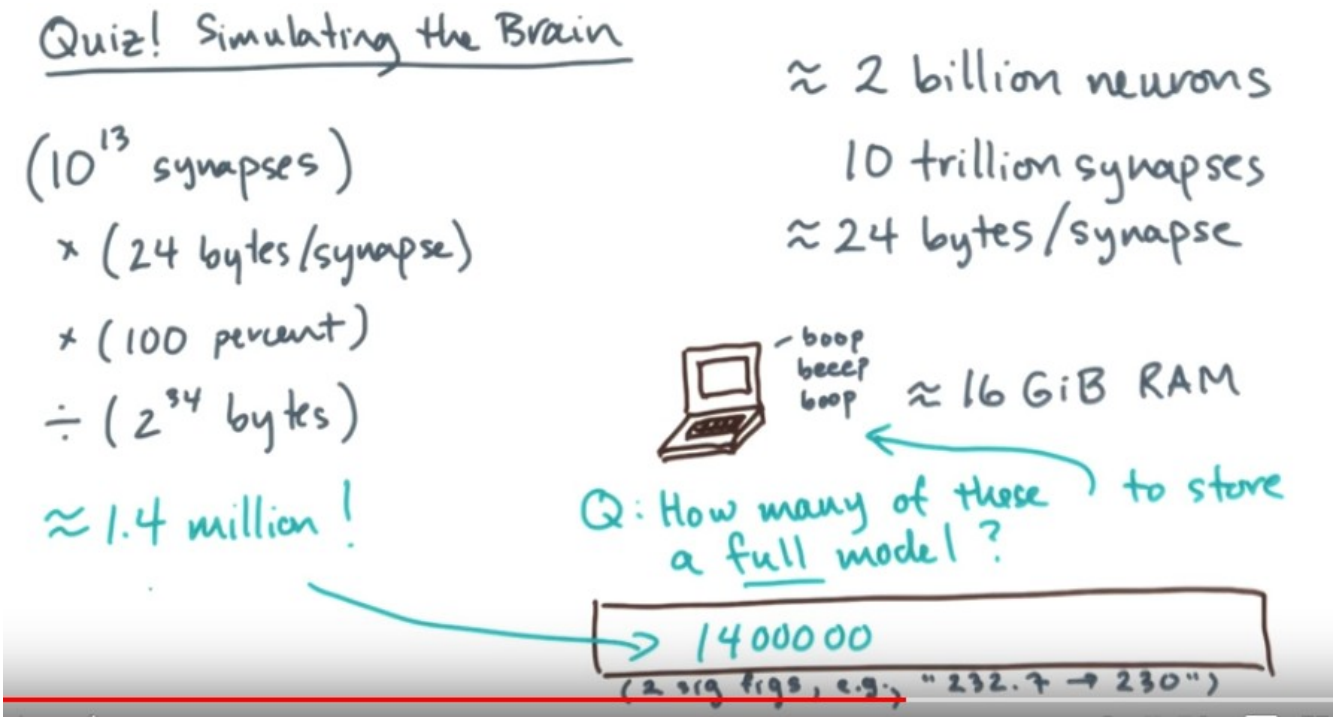




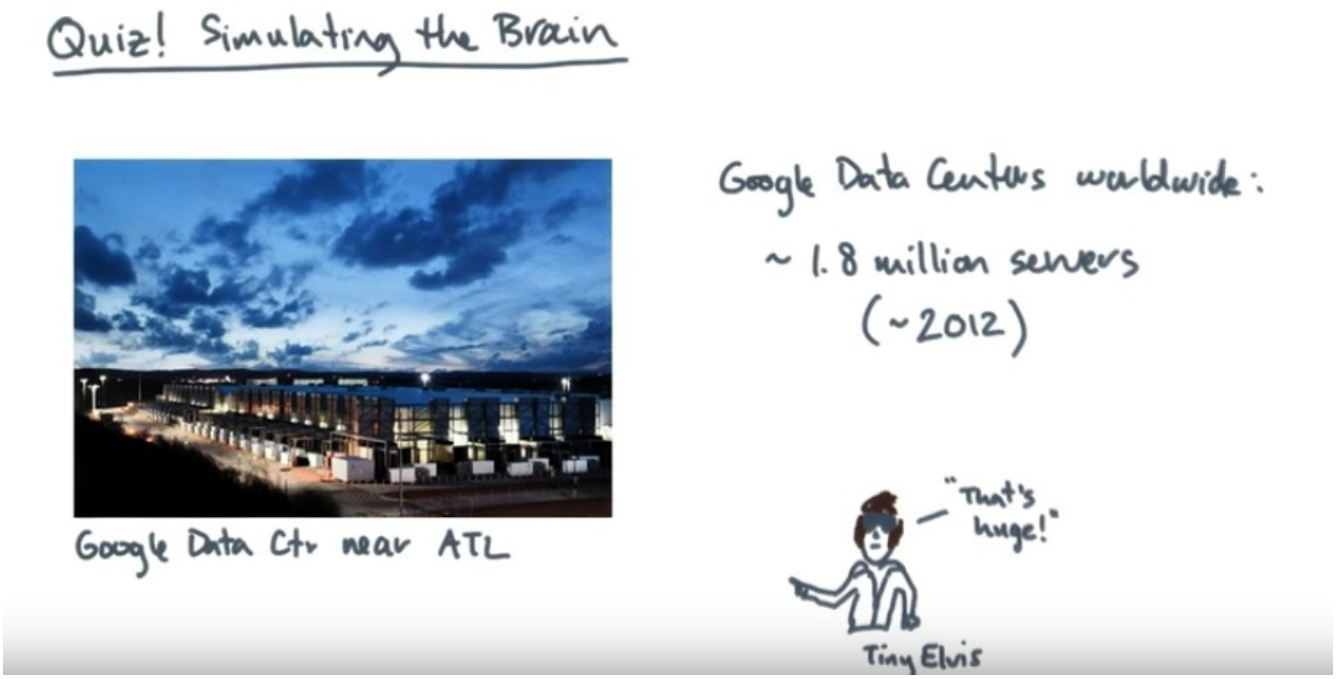
1. You're now ready to think about really big computations, computations where, for instance, the problem can't fit in the memory of a single computer. Or calculations that need so many operations, it would take hundreds of years to finish them using only a single computer. Now when the computation is really that big, you might ask, can I harness the collective power of many computers? But to design an efficient algorithm for such a machine, you need a suitable abstract machine model. And that's the goal for this lesson, for you to develop just such a model. This particular model and its variants go by many names, which you are bound to encounter as you read the literature on your own. These names include the Network Model, the Distributed Memory Model, the Message-Passing Model, the Communicating Sequential Processes Model and probably a zillion others that I've forgotten. Now personally, I like **message-passing** and for reasons you'll soon see, alpha beta. Now if any of that seems confusing, don't worry the basic abstraction is really simple. You have a bunch of computers collectively carrying out a computation. To coordinate, they communicate with one another by sending mass messages back and forth. This message-passing style stands in stark contrast to shared memory. There, processes or more likely threads, communicate by reading and writing shared variables. Now before you start, I think it's really important to think big. That's because distributed memory computations are all about solving problems that are really, really huge. How huge, you ask? Well, let's start with a quick warm up exercise to get you thinking about computations at the right scale.



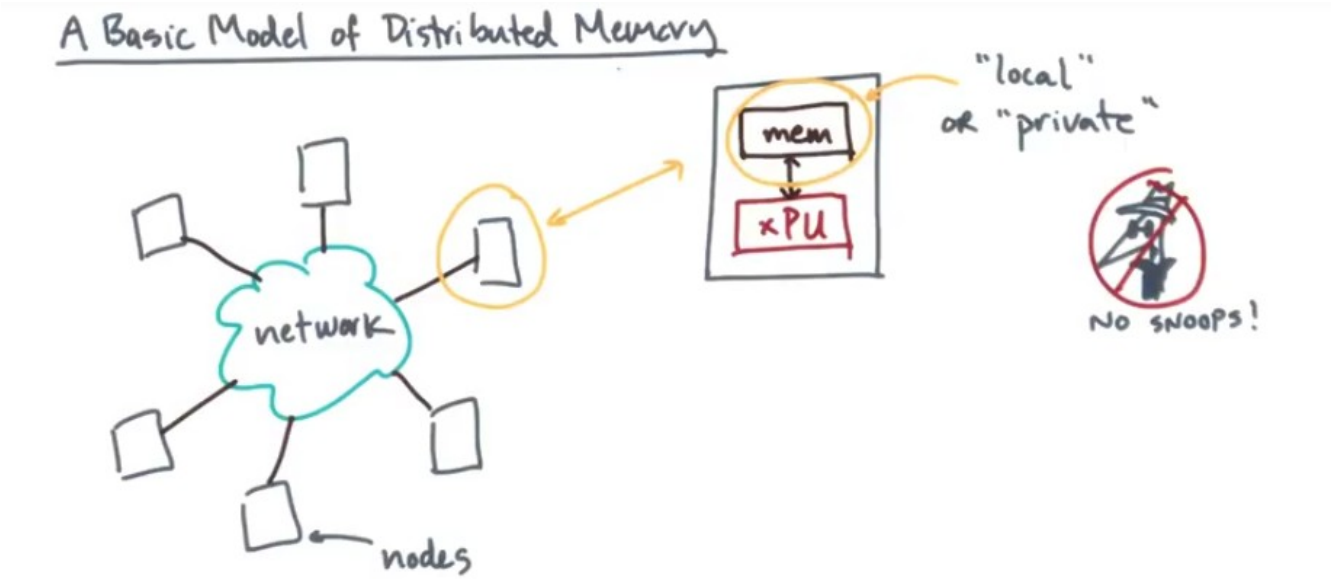
2. You may have seen a recent news story about a team of Japanese and German researchers who wrote a program to simulate the human brain. Brains! And in fact, their program didn't simulate the entire brain, just 1% of it over a short period simulated time. Now 1% of the brain is about 2 billion neurons and 10 trillion synapses. Very roughly speaking, neurons and synapses form a kind of network where neurons are like nodes, and synapses are like edges. The synapses serve as kinds of communication channels connecting the neurons. Now as it happened in the simulation, these synapses, because there are so many more of them, accounted for most of the storage and computation in the model. And in particular, the reason these researchers needed to run their simulation on a super computer is because every synapse required about 24 bytes of storage. I think you can imagine that 24 bytes times ten trillion is a lot of bytes. Now, suppose I gave you a computer work station with 16 gibibytes of main memory. If you don't know what a gibibytes is, please see the instructor's notes. Here's my question to you. How many of these 16 gibibytes machines would you need to store a full model? That is, all 100% of the brain. I want you to type your answer in this box, and please enter it to 2 significant figures. So, for example, if you thought the answer was 232.7, you would just write 230.



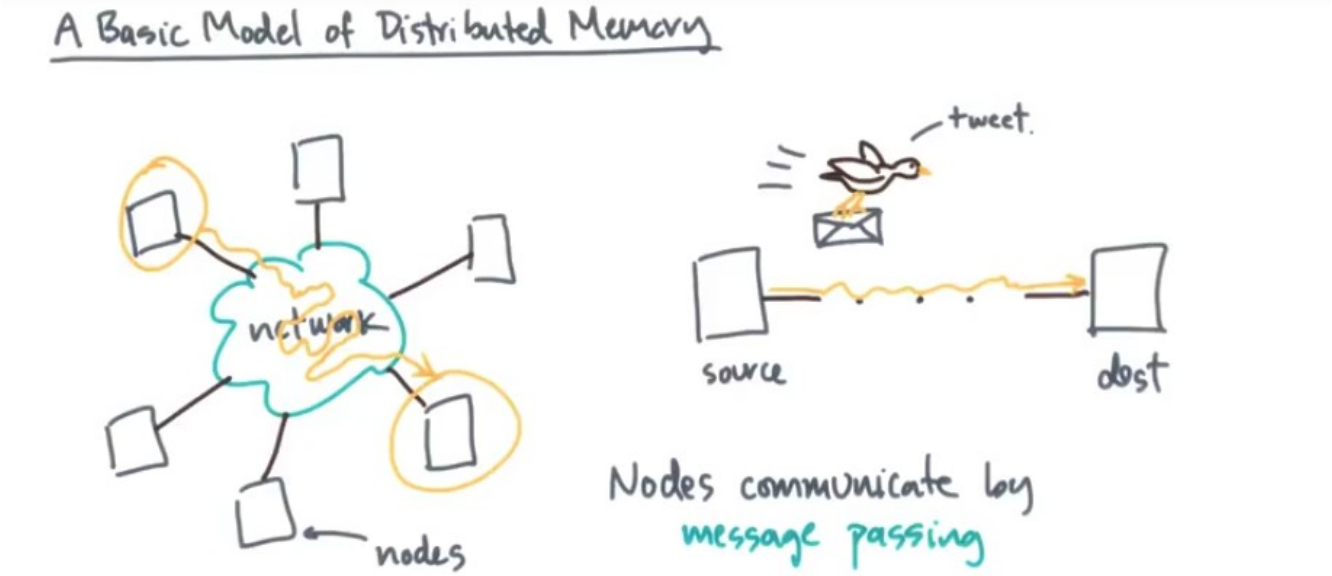
3. I hope you found this calculation to be a no brainer. [NOISE] Okay, so here's the answer that I came up with. 1.4 million computers. So, how did I get that? Well, 10 trillion is ten to the 13th. That's the number of synapses. Each synapse needs 24 bytes. And that's for 1% of the brain, so to get 100% we multiply by 100 so that's the total storage required by the model. Now each workstation had 16 GiB of RAM, and 16 GiB, as it turns out, is 2 to the 34th bytes, so dividing this by 2 to the 34th, you should get something like 1.4 million machines. So, how many computers is that? A heck of a lot. The super computer with the largest number of compute nodes is probably Sequoia, which is an IBM machine at Lawrence Livermore lab. That's of June 2014. The sequoia machine has about 98,000 nodes.



So what if we took a look at data centers. According to a 2012 analysis of Google's data centers using satellite maps. Someone with a lot of time on their hands estimated that all of Google's data centers might have about 1.8 million servers. Again that's of 2012. So, they probably have more. 1.8 million just barely big enough to hold the whole problem. Hey that's huge.

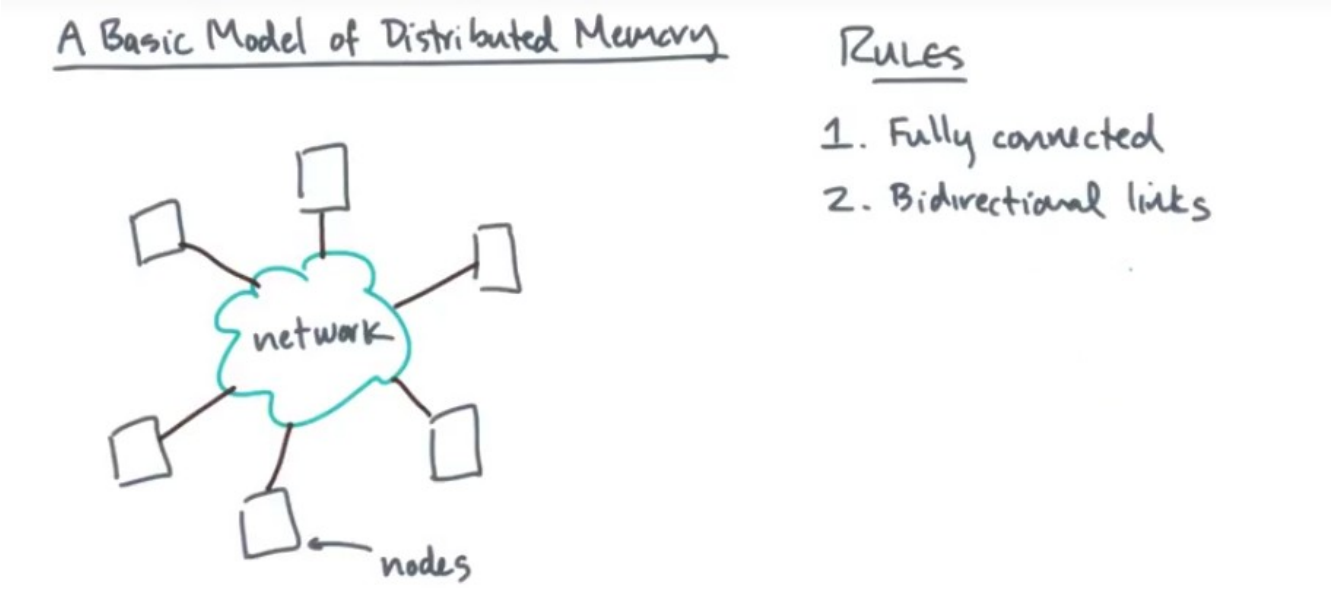


4. To start designing algorithms for a cluster or a supercomputer, you're going to need a machine model. How about this one? In this model, a machine is a collection of nodes connected by some kind of network. Each node consists of a processor connected to a private memory. By private, I mean that the node can only directly read or write its own memory. It can't directly access the memory of other nodes. I'll refer to this type of machine as a Distributed Memory machine. Now this abstraction of private memories is critical. It implies that to share data, nodes will have to send messages to one another.

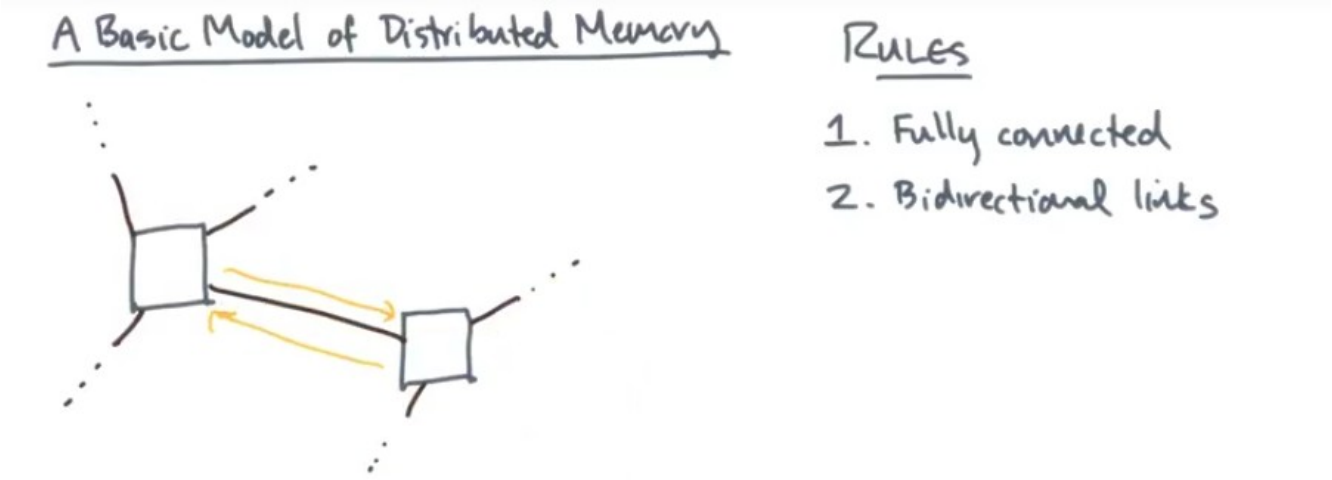


So for example, suppose this node wants to send data to this node. The only way to do that is the sender or source has to package up a message and then put it on the network. So this message will

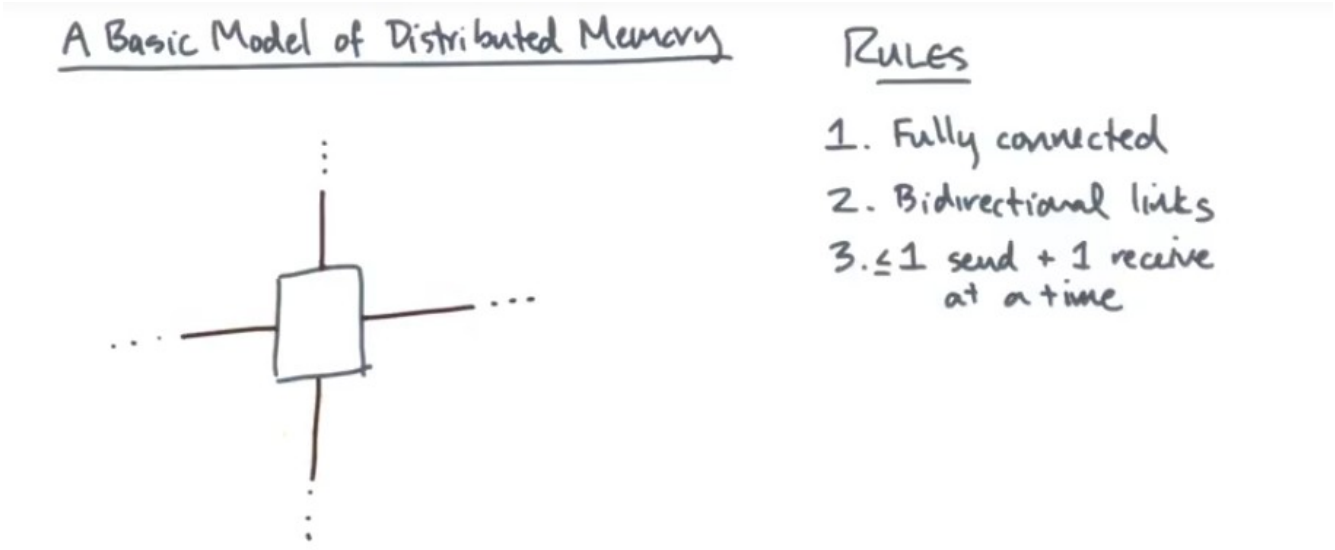
have to find some path through the network to get from the source node to the destination node. In contrast to shared memory where we read and write shared variables, this(即 send message 的 model, 不是 shared memory model) style of parallel communication is called message-passing.



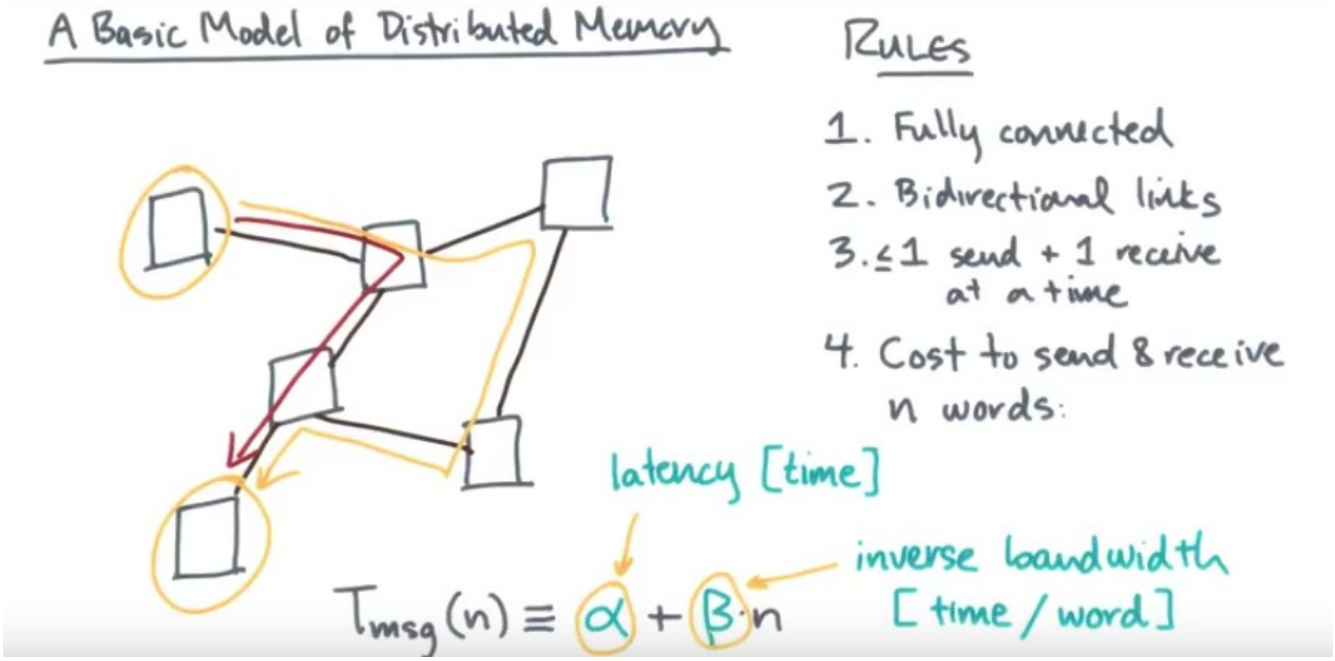
Now this machine model has a few simple rules. The zeroth rule of the model is you never talk about the model. By that I mean, you need to master and internalize these rules. Okay, so what are the real rules? The first rule is that you should assume the network is fully connected. That means there's always a path from any node to any other node in the network. The second rule is that the network links are bi-directional.



What does that mean? Well, suppose I have two nodes in the machine, and they're connected by a link. Bi-directional means that the link can carry a message in both directions at the same time. So, while one message is going this way, another message can be going this way.

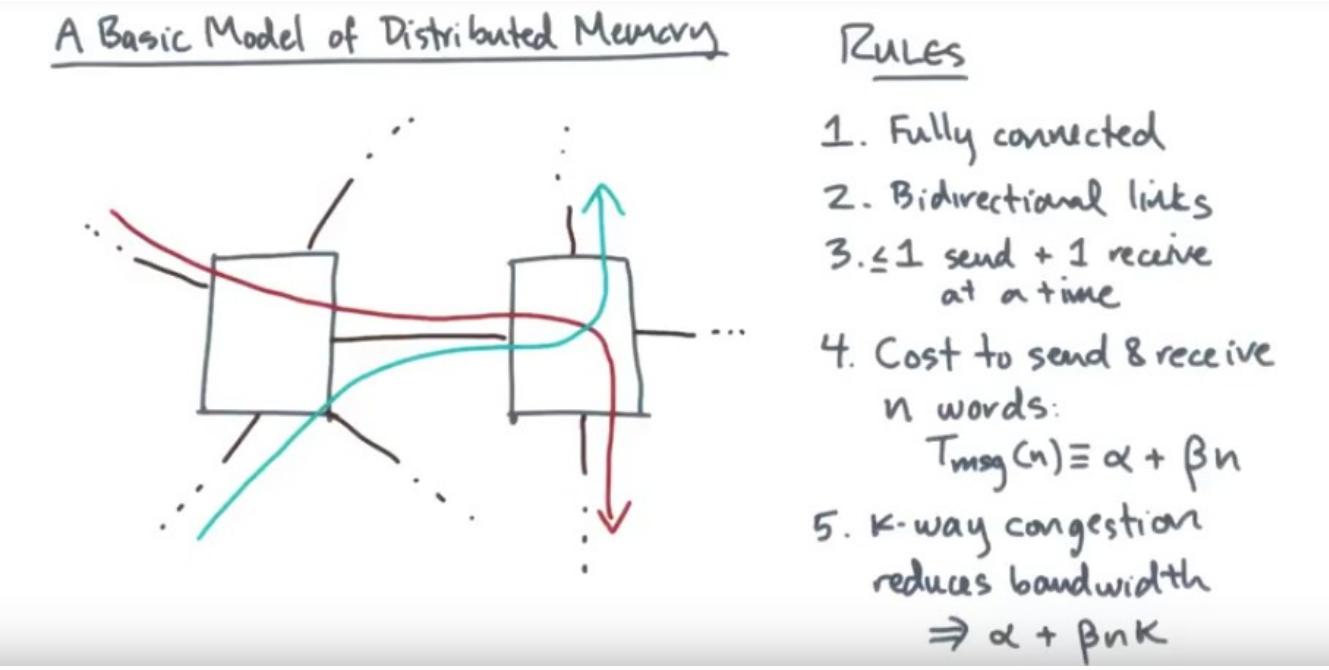


The third rule is that you will allow a node to concurrently perform at most, 1 send and 1 receive at a time. This rule is important because it effects the cost of communication. So for example suppose this node wants to send the message on each of its outgoing links. In order for this node to send four messages it's going to have to send them one at a time. By contrast it could do one send and one receive simultaneously.

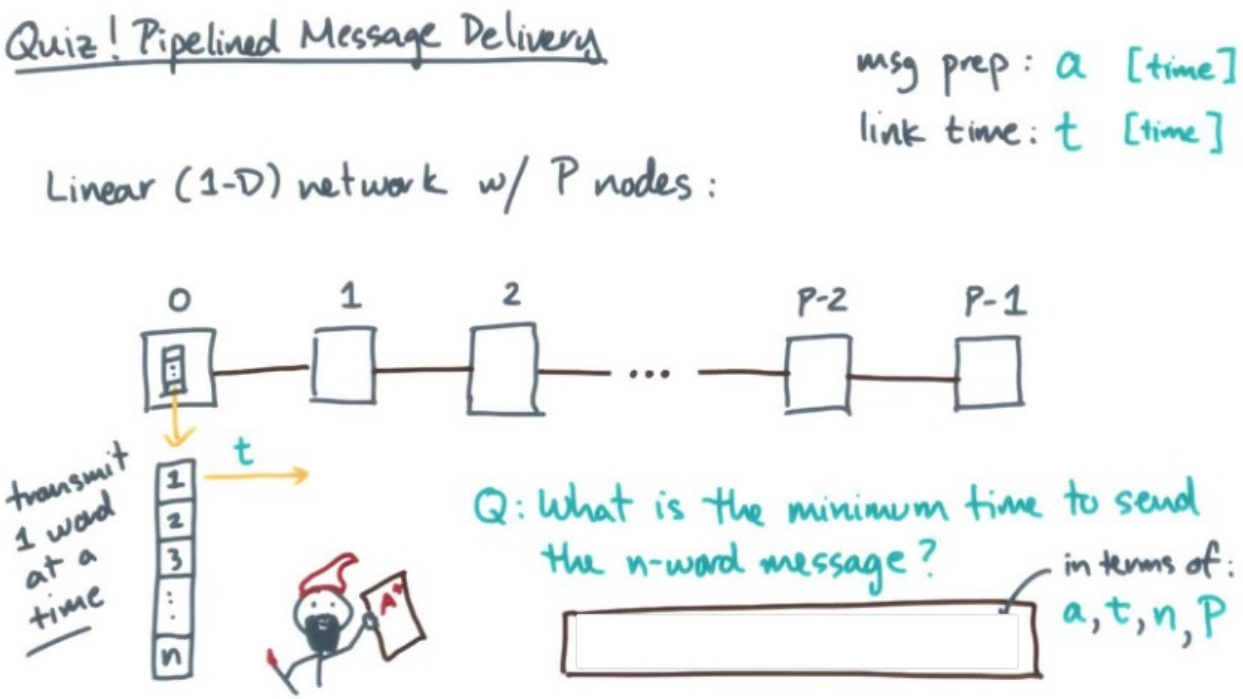


The fourth rule is about the cost of a simultaneous send and receive of n words. So suppose this our computer, and let's further supposed that **this node wants to send a message to this node.** There are several different paths, here is one path and here's another. **Regardless of which path** the message takes, rule number four says the following. The time to send **the message** if it **contains n words**, is a constant alpha plus another constant beta times the size of the message. In other words, the cost of sending the message is linear in the message size. Now, it may seem strange that this cost, somehow, is independent of the path that's taken. Now, that's not entirely true. And I'll clarify a little bit later in the lesson. For the time being, let's accept the formula, as stated. Now, the formula has two terms. The

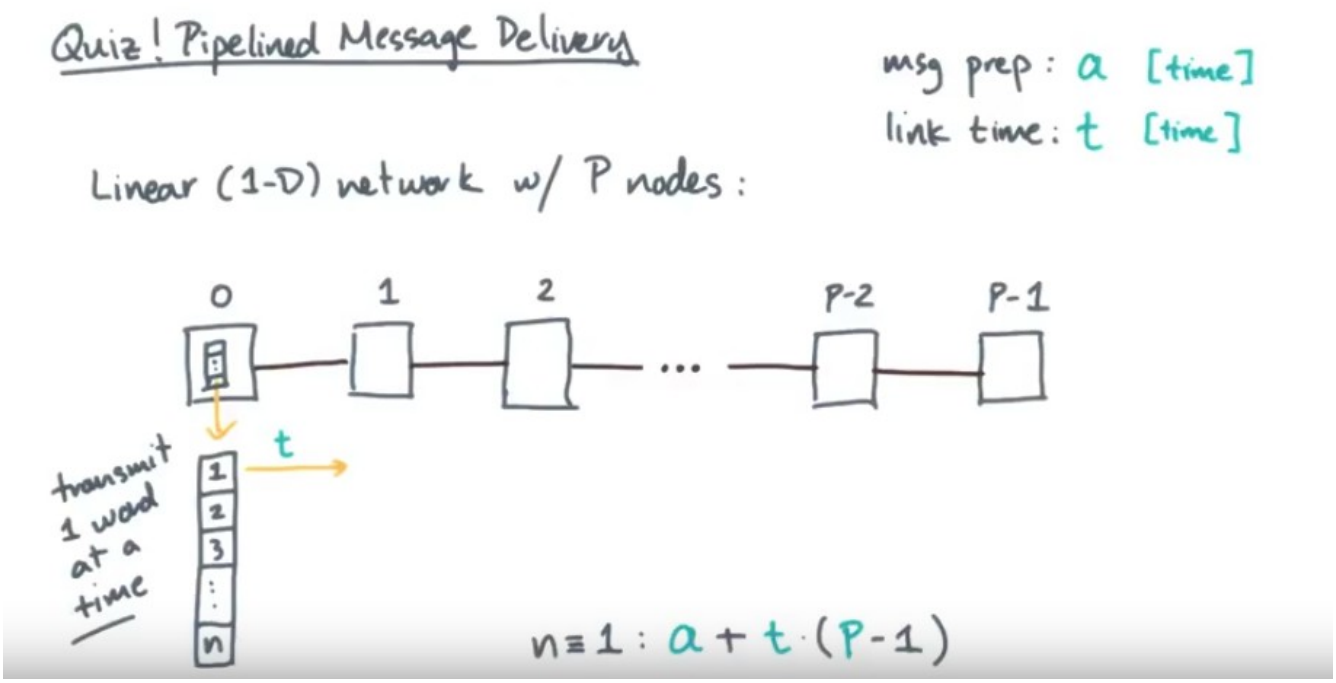
first term is called the latency, and it has units of time. It's a fixed cost that you pay no matter how large the message is. The second term has a parameter beta. Beta has a name. We'll call it the inverse bandwidth. It has units of time per word. You'll think a little bit more about where this cost model really comes from momentarily, so just sit tight. One subtlety is, that this fourth rule really only applies when there are no messages competing for links.



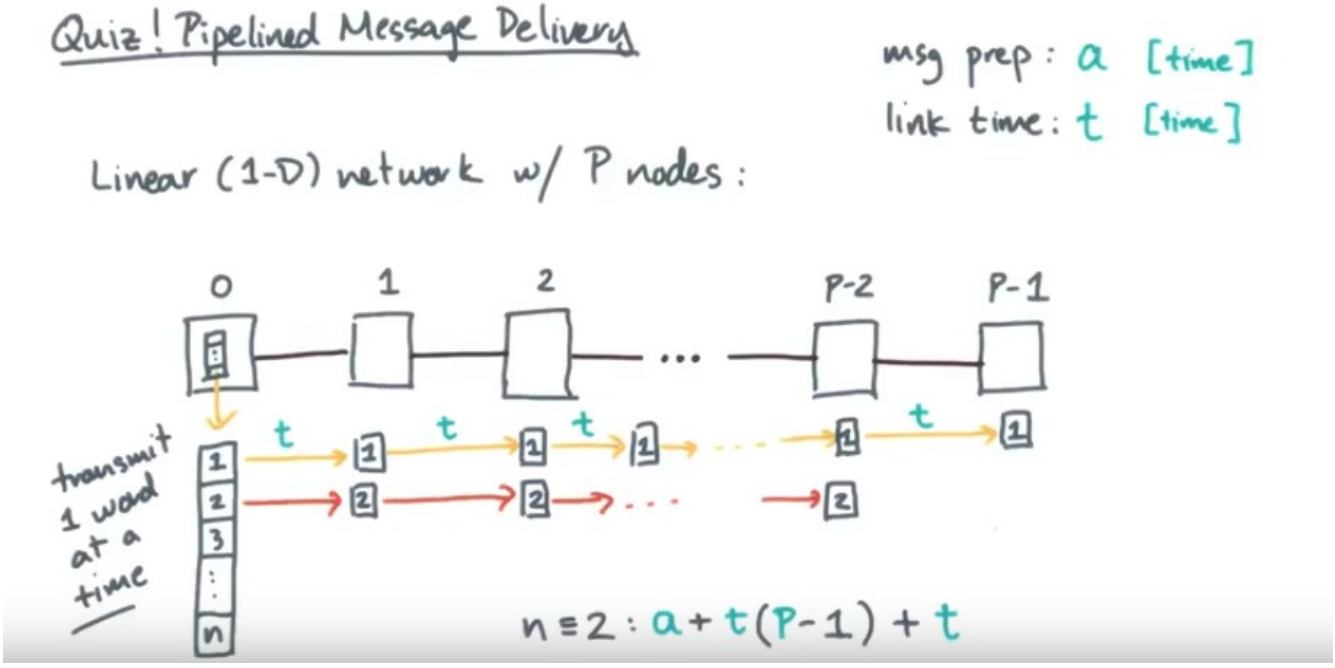
So we have a fifth rule which tells us what to do when messages are trying to go over the same link, at the same time. This situation is called congestion. This rule says if there are k messages simultaneously competing for a link, then in terms of the cost of the message, will change only the beta term and in particular we'll multiply it by k. For instance, let's consider this link that connects these two nodes. The last rule governs what happens if two messages try to go over the same link simultaneously. So suppose there's a red message going this way and a blue message going this way. The last rule says that the effective cost of this operation is the same as if the data transmission part, which is the beta term, is serialized over the link. So if these two messages are being transmitted in parallel Instead of observing a parallel execution time of alpha plus beta n, supposing that the messages are the same size n. What you'll see instead is a cost of alpha plus beta times n times 2. Okay, so take a second to look at these rules and commit them to memory.



5. I admit that I pulled this alpha-beta model out of thin air. So you should be asking yourself, where does this model come from? Let's consider a linear or 1-D network having P nodes. Now suppose node 0 has a message of size n that it wants to transmit to the last node in the network. Let's say it does that by taking the message, breaking it up into n individual words and transmitting one word at a time. Now, let's assume this step of preparing the message for transmission costs you a fixed amount of time, little a. Then let's suppose that any time a word goes from one node to another that it incurs t units of time. My question to you is, what is the minimum time to send this n-word message? Type your answer here. Now your answer should be symbolic, so you should write it algebraically in terms of a, little t, n, and p. There's a little grading gnome running behind the scenes that will do its best to compare what you enter automatically against our proposed solution.

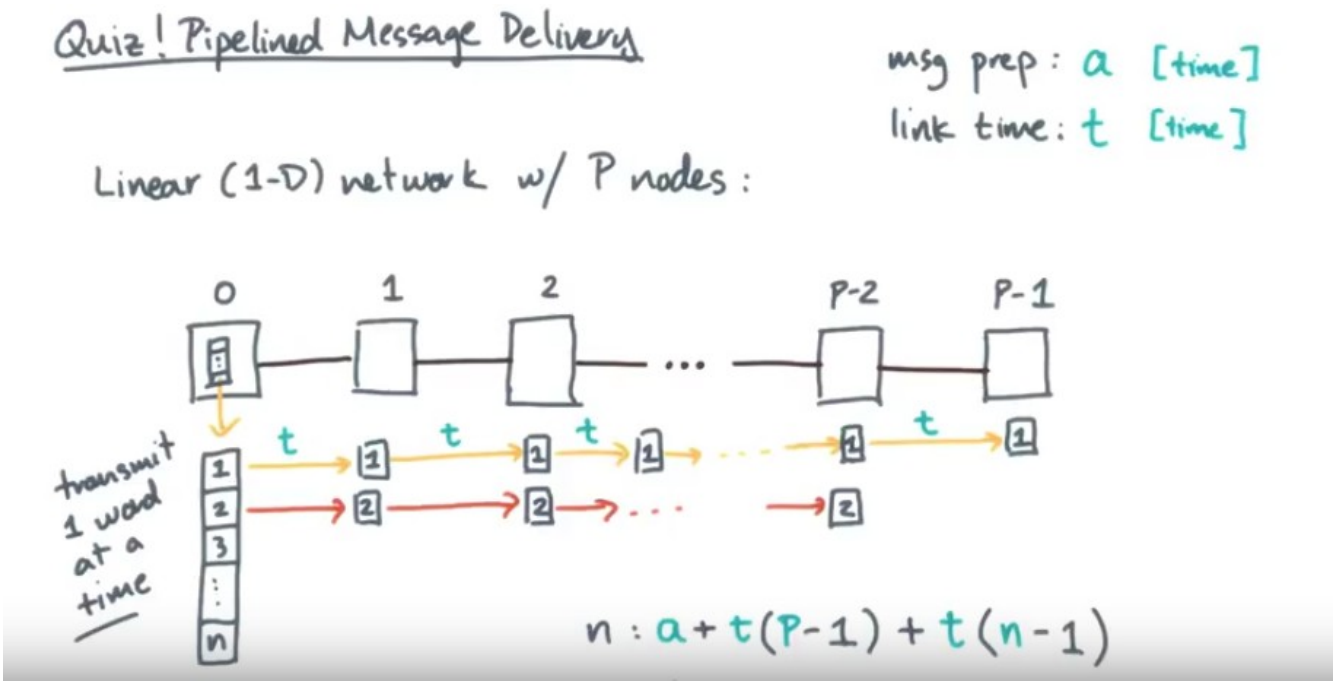


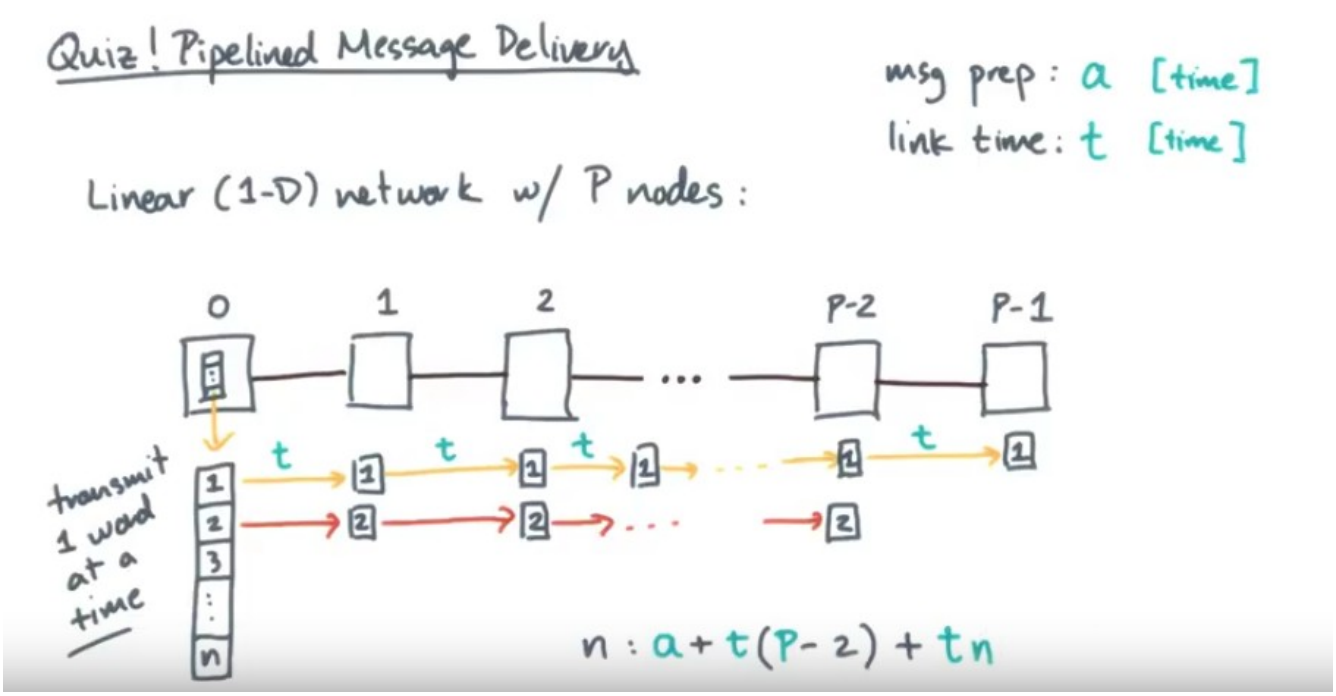
6. Okay, how do we figure this out? Suppose you were sending a message of just one word instead of n. That is, suppose n=1. Then the time would be this fixed startup time plus the time to traverse all the links which is t(P-1).



Now suppose instead of sending one word, we're actually sending two words. So we'd pay the startup time, little a, as before. Then the first word would hit the network and find its way to node 1. Eventually, it arrives. Then in the second step, word 1 continues to node 2, while the second word starts its way from 0 to node 1. Now this step and this step happen simultaneously, so we only see the cost of this step. And this continues. Eventually, the first word makes it all the way down the network to the destination. The second word always lags one step behind. So what does that mean? That means

to send two words, we pay the same cost as sending one word, plus an additional T units of time for the second word to go the last hop. I think you can see how this would work for n = 3, n = 4, and so on. So for any n, there's the time to send the first word plus the time for the remaining words to arrive at the destination.

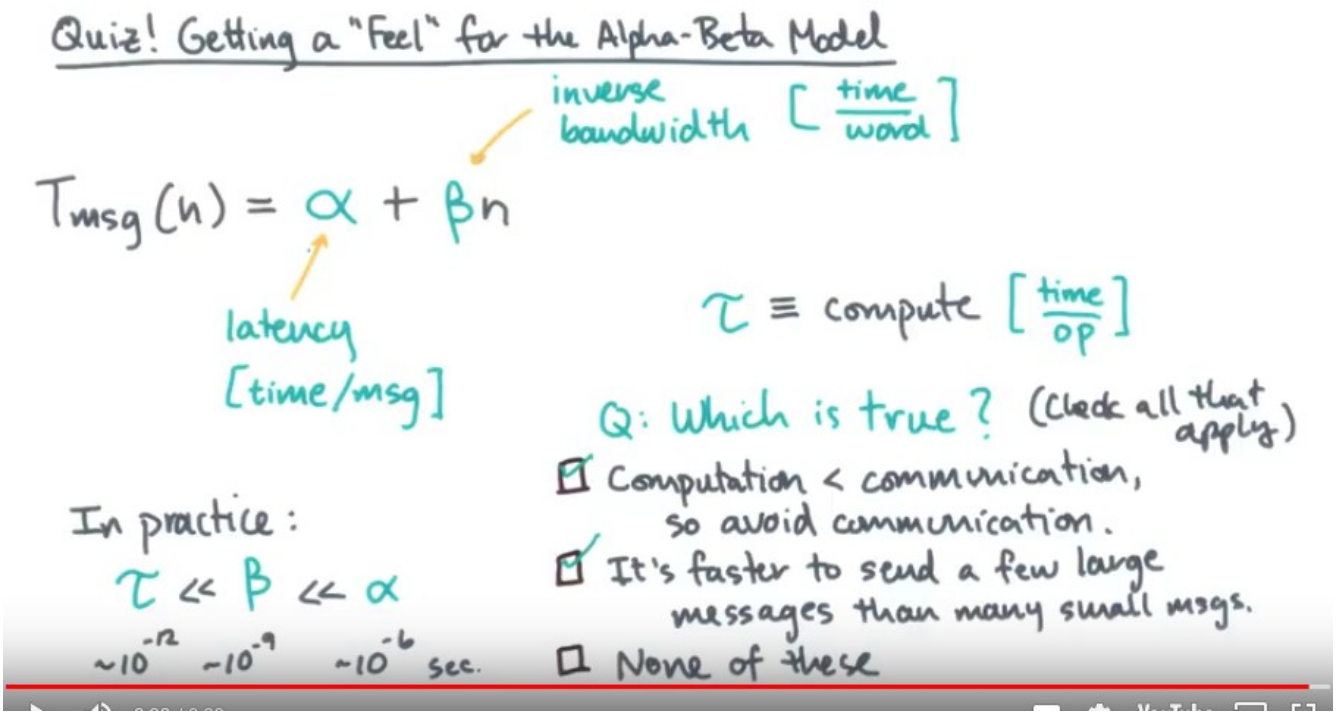




Let me reorganize this algebraically into this form. Now this formula has three terms. There's one that depends on P, and there's one that depends on n. And then of course, there's the startup time. So let's interpret these. The first term is just the message startup overheard. What about the second term? In practice, the time t effectively measures the message transmission time along a link. So you can interpret the second term as a kind of wire delay in going across the network. And finally, the third

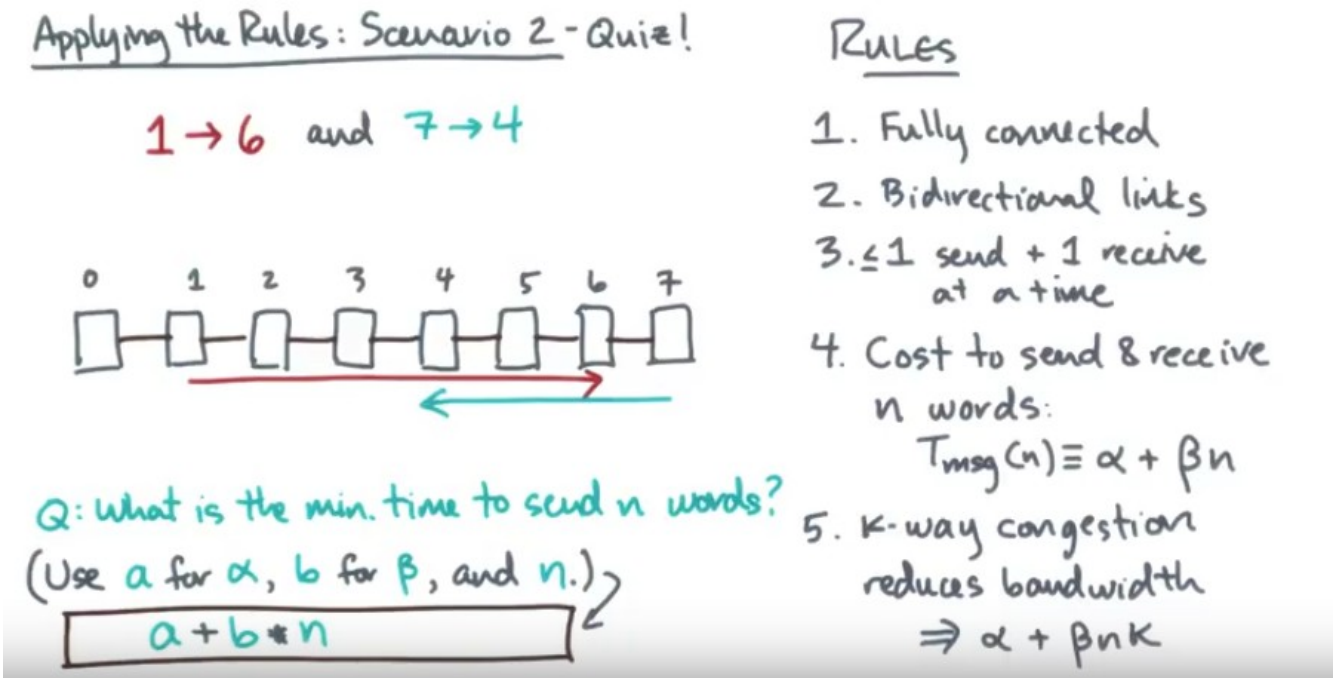
term captures a dependence as a function of a message size. And as it happens, even when P is very large, let's say tens or hundreds of thousands, it's actually still small, or only comparable to little a. Little a is typically associated with a software overhead for preparing messages for delivery. In fact, a is typically many orders of magnitude larger than t. So we typically treat these two terms as being effectively a constant. That's what becomes alpha in our alpha-beta model. And then beta is something that's sort of proportional to one over t. Anyway, this is a really rough analysis. The textbook actually has a more detailed explanation. So head to the instructor's notes for our reference. My hope is that if Alpha-Beta seemed mysterious before, hopefully you have a little bit better of an intuition for it now.

7. With the Alpha-Beta Model, an algorithm has three costs. The message cost defines the first two. It's essentially the latency, alpha, and the inverse bandwidth, beta. Now our algorithm will, of course, do other kinds of basic operations like comparisons and arithmetic. Let's denote the cost per compute operation by tau. Now in practice, it turns out the following is true. Tau is typically much less than beta, which is in turn much less than alpha. To give you an idea, you should be thinking of something like 10 to the minus 12 seconds, versus 10 to the minus 9 seconds, versus 10 to the minus 6 seconds. So something like three orders of magnitude in between. Let's suppose you buy this. Here's my question. Which of the following claims is true? And I want you to check all that apply. Is it true that computation is less than communication, so you should avoid communication? Or is it true that it's faster to send a few large messages, rather than many small messages? Or is neither of these statements true?



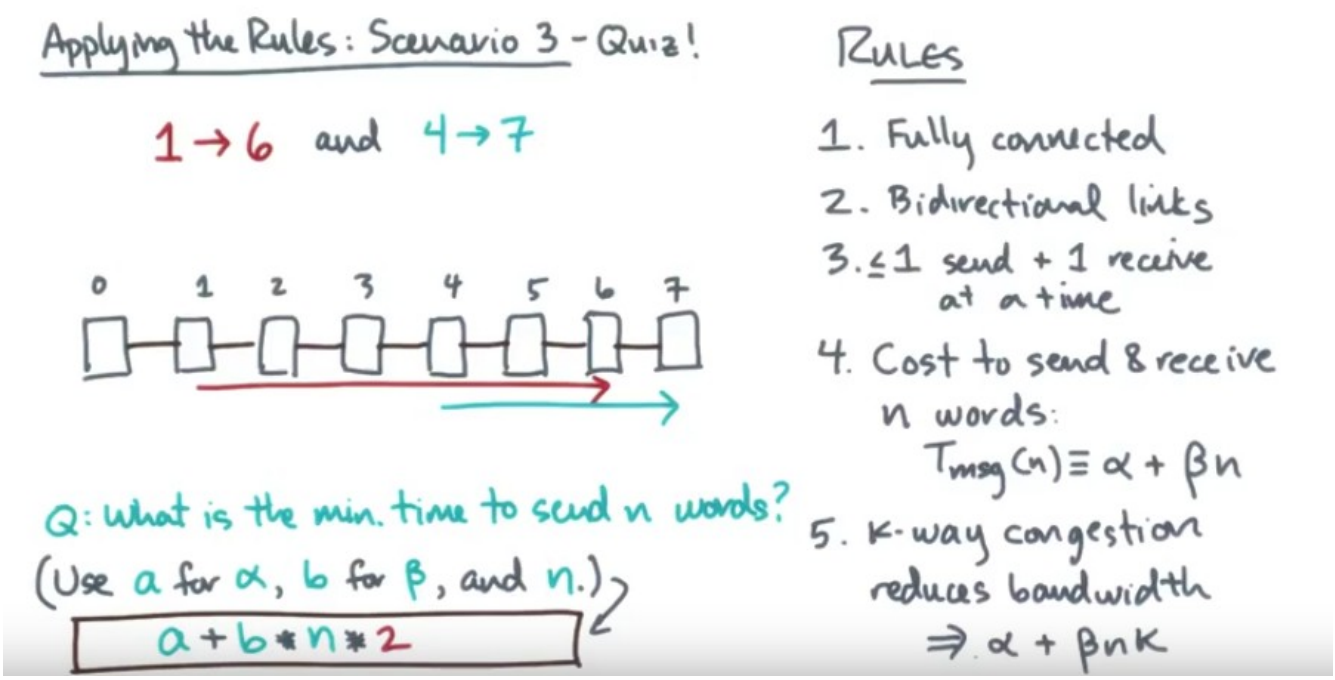
8. The answer is that both the first and the second statements are true. Computation (τ) is much less than communication (α, β), so you should probably avoid communication. It's an unfortunate fact of life that many computer scientists are very good at this. The second statement is essentially about the relationship between beta and alpha. So think about that. According to this, in looking at this formula, it's evident that you can deliver something like on the order of, I don't know, 1,000 words or bytes in the same time that it takes just to start a message. So, that suggests it would be better to send a few large messages rather than many small messages.

9. Recall the five rules of message passing and consider the following scenario. Suppose I have a linear network of eight nodes. So suppose that node zero wants to send a message to node two, at exactly the same time that node six wants to send a message to node three. How much time will it take for these two simultaneous messages to transpire? Here's the path from node zero to node two and here's the path from node six to node three. Notice that these messages occupy independent paths in the network, therefore, they can happen simultaneously. So if the messages are both of size n then the time is just alpha plus beta times n. Now I want you to try.



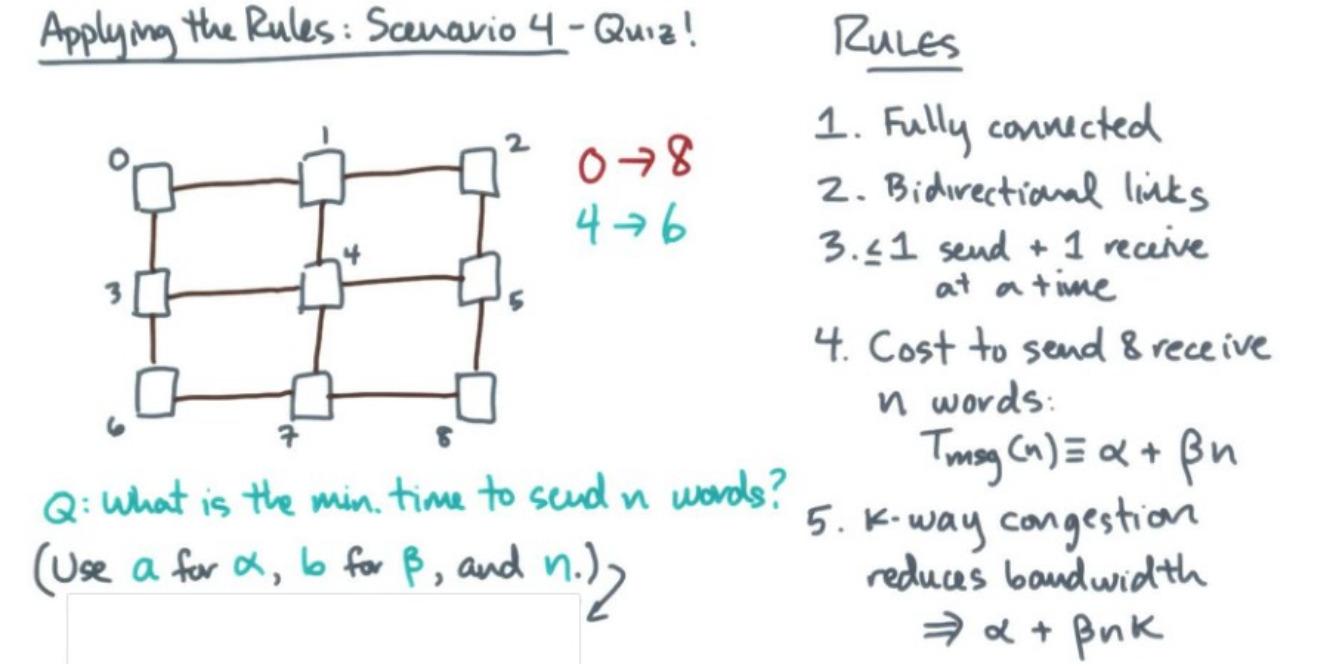
10. Recall the five rules of message passing and consider an example on the following linear network which has eight nodes. Suppose one wants to send a message to node six at exactly the same time node seven wants to send a message to node four. What's the minimum time it will take for these two messages? I want you to type your answer in this box. And I want you to do it symbolically. So for alpha use little a, for beta use little b, and for n you can just use n.

11. Let's start with an observation. 1 going to 6 occupies this path, whereas 7 going to 4 occupies this path. Now the two paths overlap, but they're going in opposite directions. So by rule number 2, which is the existence of bidirectional links, these messages can be carried at the same time. So you should have written something like a + b * n.

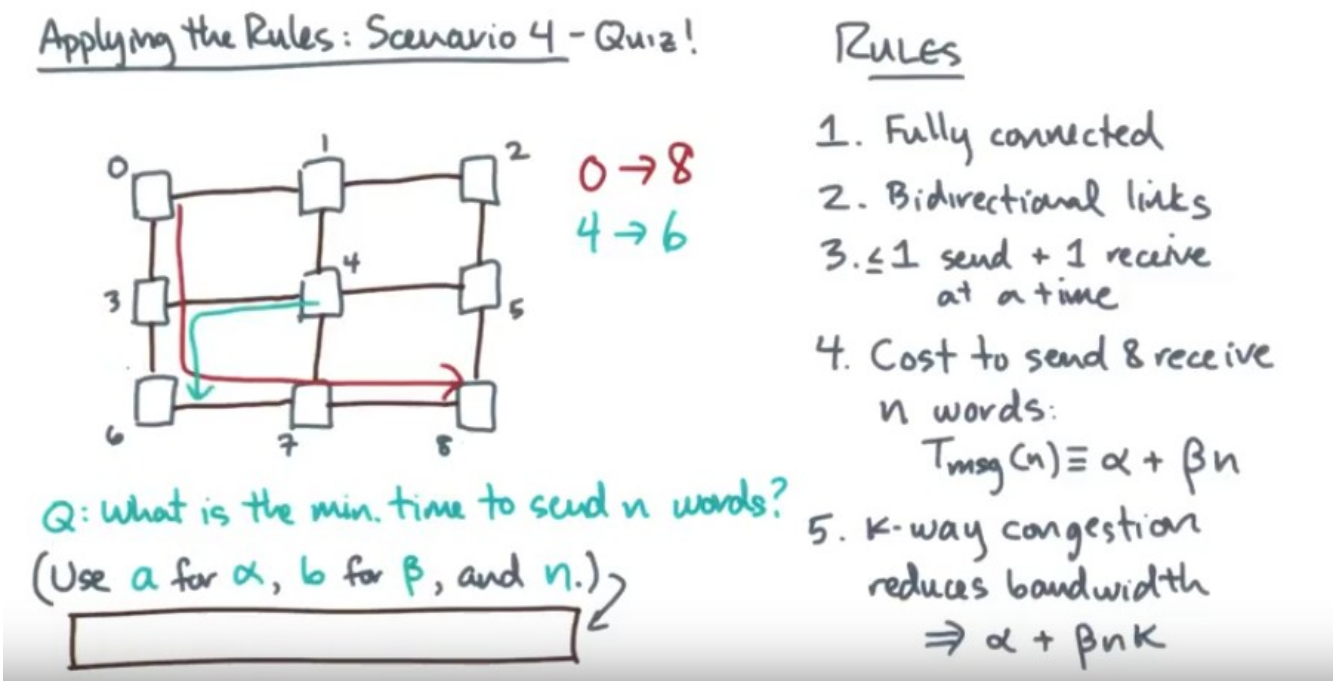


12. Recall the five rules of message passing. I want you to think about the following scenarios on a linear network with eight nodes. Suppose node 1 wants to send a message to node 6 at the same time that node 4 wants to send a message to node 7. How much time will this take? I want you to type your answer in this box. You can use little a for alpha, little b for beta, and n.

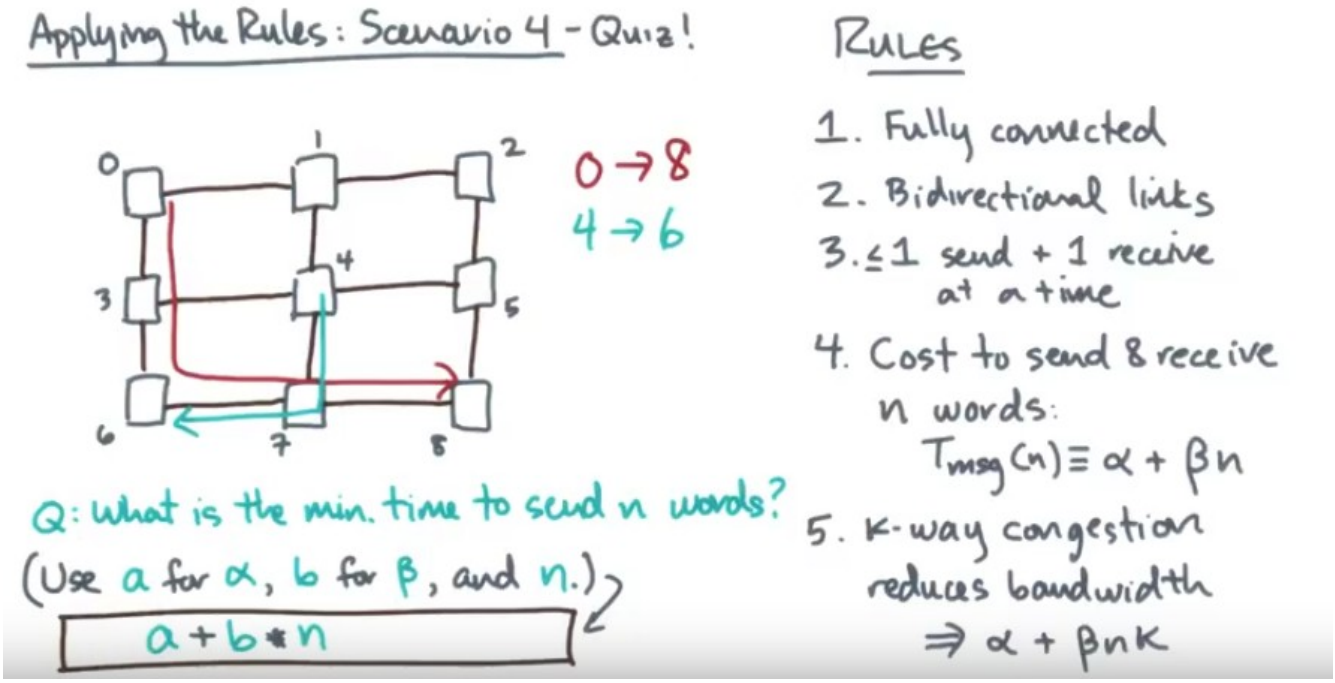
13. Let's start with an observation. Here are the two paths that these two messages are trying to take. Notice that the paths intersect and they're going in the same direction. That means the messages will need to serialize in their bandwidth term. And since there are two such messages, we'll take bandwidth term and multiply by 2. This is essentially an application of rule 5, k-way congestion.



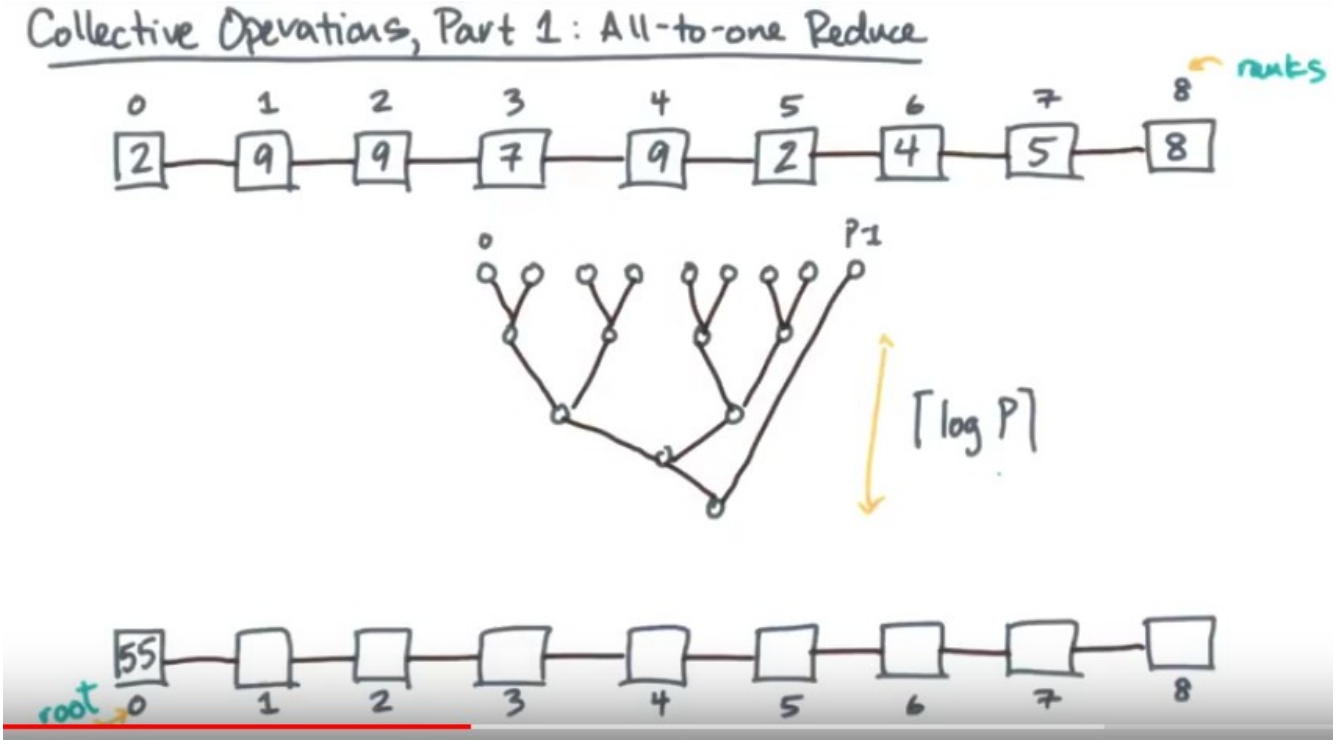
14. Recall our five rules of message passing. Now, I want you to think about them in the following scenario. Consider this two dimensional mesh network of nine nodes arranged in a three by three grid. Suppose node zero wants to send a message to node eight. And suppose at the same time node four wants to send a message to node six. Here's my question. What is the minimum time to send n words? Type your answer here symbolically. You can use little a for alpha, little b for beta. And of course, n for itself.



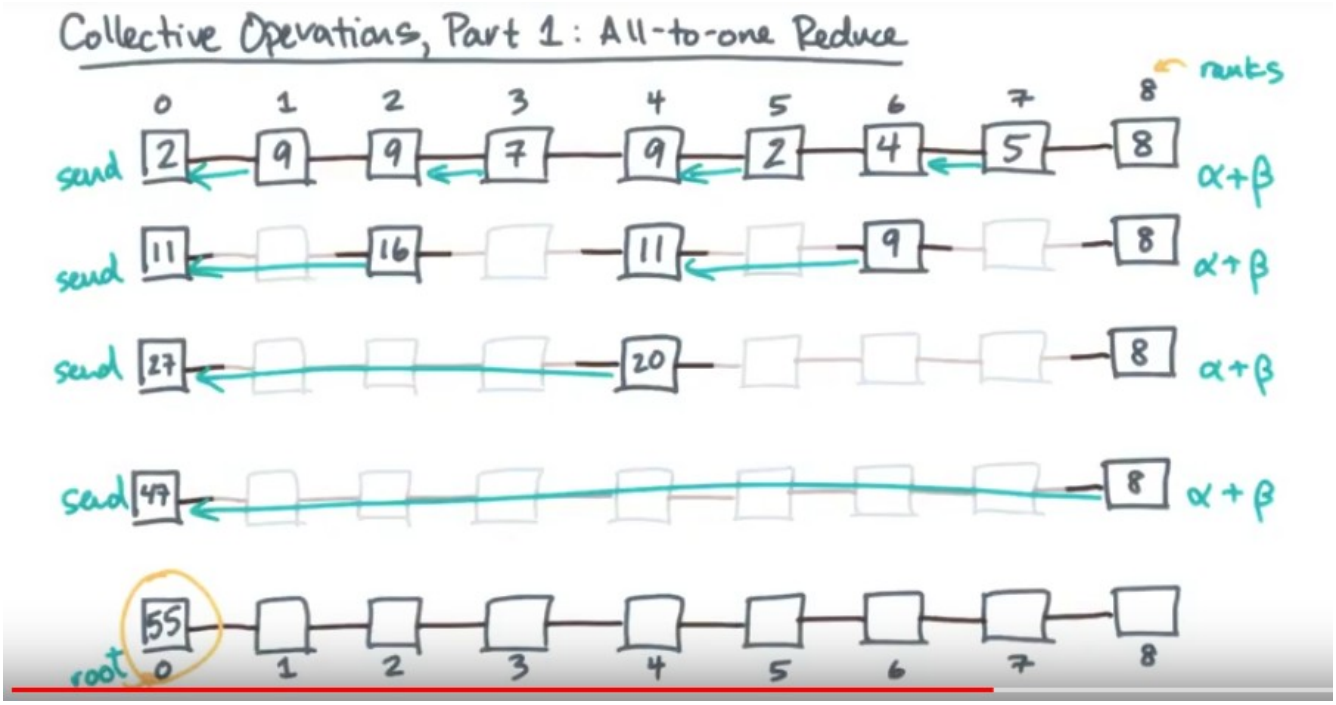
15. The answer is it depends. There's a deliberate ambiguity here. For example, suppose node zero sends to node eight via this path. If the second message takes this route, then the two paths intersect going in the same direction and there will be congestion at this link.



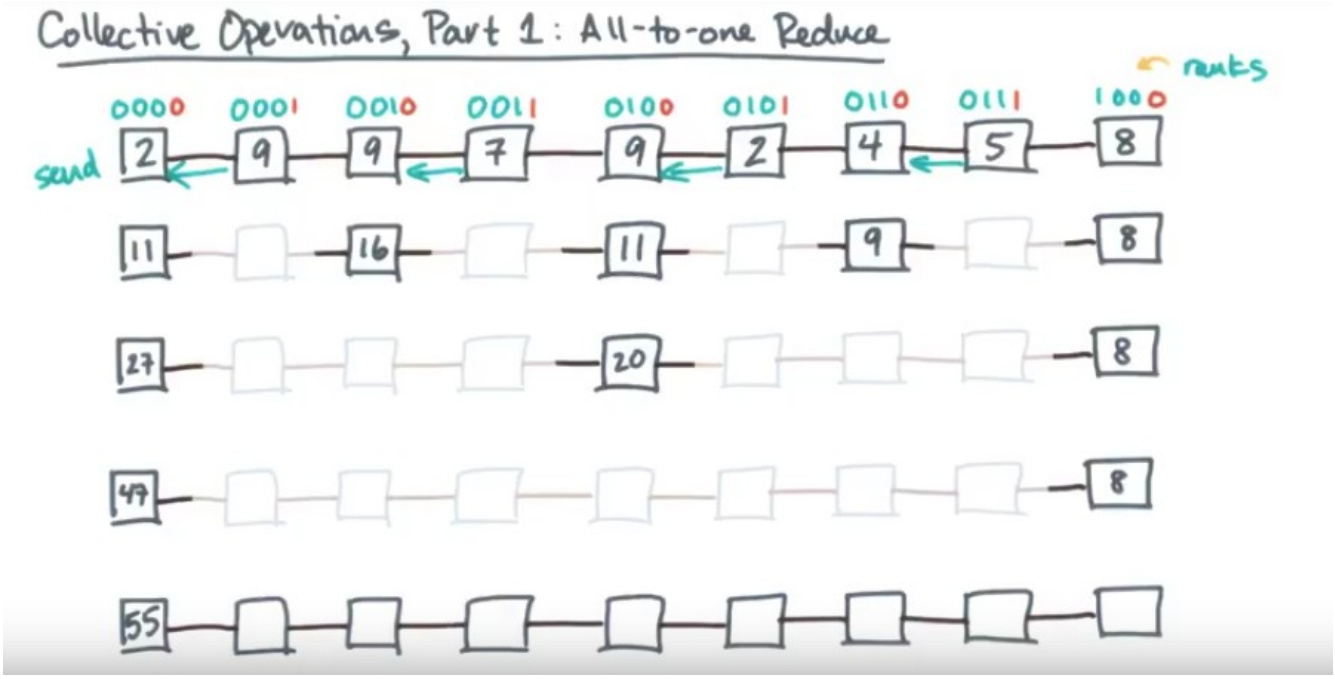
If instead the second message takes this route, then there's no contention. So, in fact, if these are the only two pending messages, you can overlap them by choosing good routes. In genera, for some analysis, you're going to need to reason about whether or not congestion or contention are occurring. We'll discuss some techniques to do so in a different lesson. Now the question asks for the minimum time, so if you assume optimistically the non-colliding paths, then you would get just a + b * n.



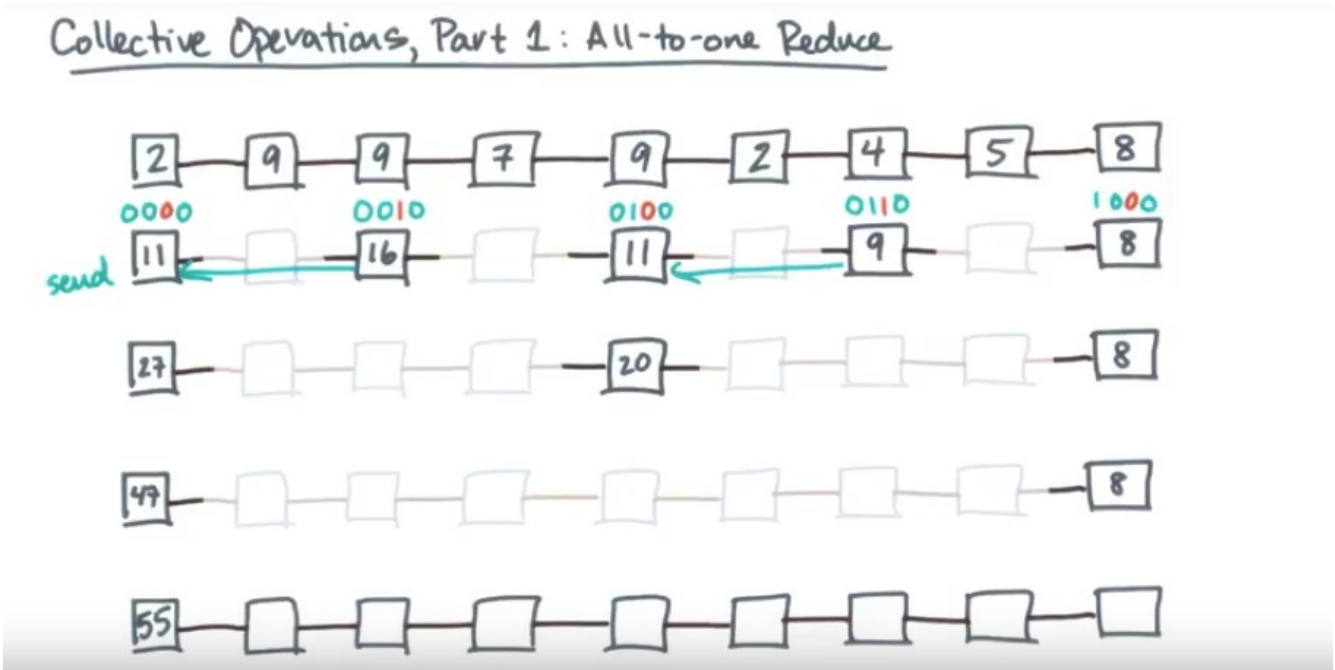
16. An important class of distributed memory algorithmic primitives are the so-called collective operations. No, not that kind of collective. One example of a collective operation is a reduction, which in the distributed memory case, I will call all-to-one reduce. The phrase all-to-one makes it clear that all nodes (這裡的 node 還是電腦的意思) participate to produce a final result on one node. Let's take an example. Consider this linear network which I've numbered from 0 to 8. These ID (above the node squares), numbers I'll sometimes call ranks. So, for example, the rank of this node is 4. Now suppose each node has a single value in its private memory, let's say these values (inside the node squares). In an all-to-one reduction, you want to produce a final result which is the reduction, in this case sum of all the values on a root node. So the sum of these values it turns out is 55, and let's say we want that result to appear on node 0. Now intuitively, you expect a tree base scheme like this one to work well. It's span is logarithmic in the size of the input. In this case, there are P values. Now, this observation suggests that you'll need to perform at least log P communication steps.



Now, initially, the data is distributed across all nodes, so no node can do any actual work. Therefore, you'll need to do some communication. Okay, well that's easy. Let's have all the odd number nodes send their data to their leftmost neighbor. So 1 will send to 0, 3 will send to 2, and so on. None of these sends conflict so all the sends can happen simultaneously. So this round of communication will cost you alpha + beta time since you're only sending one word. After the communication, the even ranks add the incoming value. The new state of the system then looks like this. Now in this next step, you'll notice that only even numbered ranks have values. So notice the even ranks have partial sums. So we can repeat this scheme, except this time with odd partners of the remaining set sending to the closest even partner. So we might do these sends, for example. Again, there are no conflicts in the paths so this costs us alpha + beta time. And you'll just repeat this process until a final result remains.

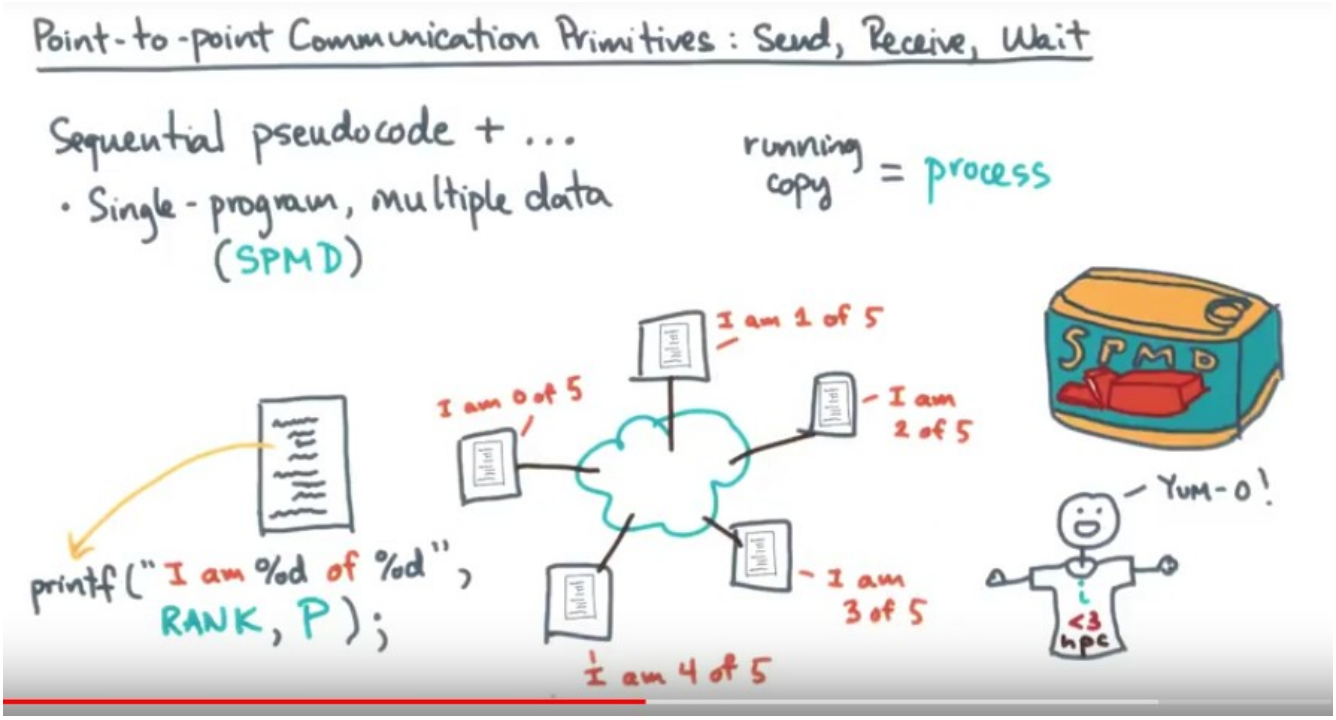


Now, to implement this scheme, I want to point out a useful fact. Suppose we replace the ranks by their equivalent binary strings. Recall that in the first round, the odds send to the evens. Notice the odds all have a 1 in their least significant bit, and the evens all have 0.

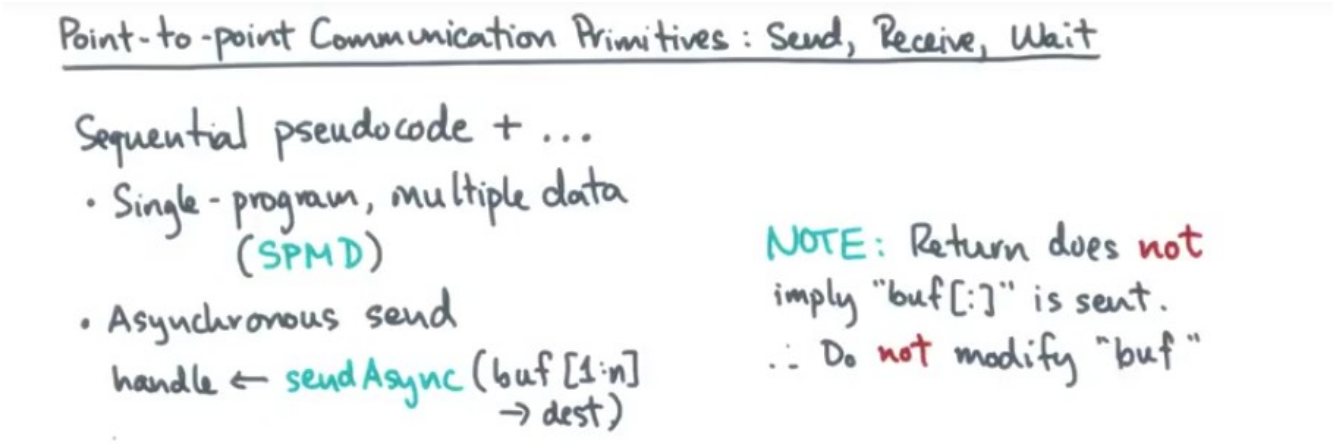


In the next round, the only participating processes have a 0 in their least significant bit. And notice what happens. Nodes with a 1 in their second bit send to nodes who have a 0 in their second bit. So again, it's basically odds to evens where you drop off the last bit. Neat. If you've heard of things like hypercubes, binary reflected Gray codes, or Hamming distances, then this bit manipulation or bitview of the world will seem very familiar. And if you aren't, we'll give you some pointers in the instructor's

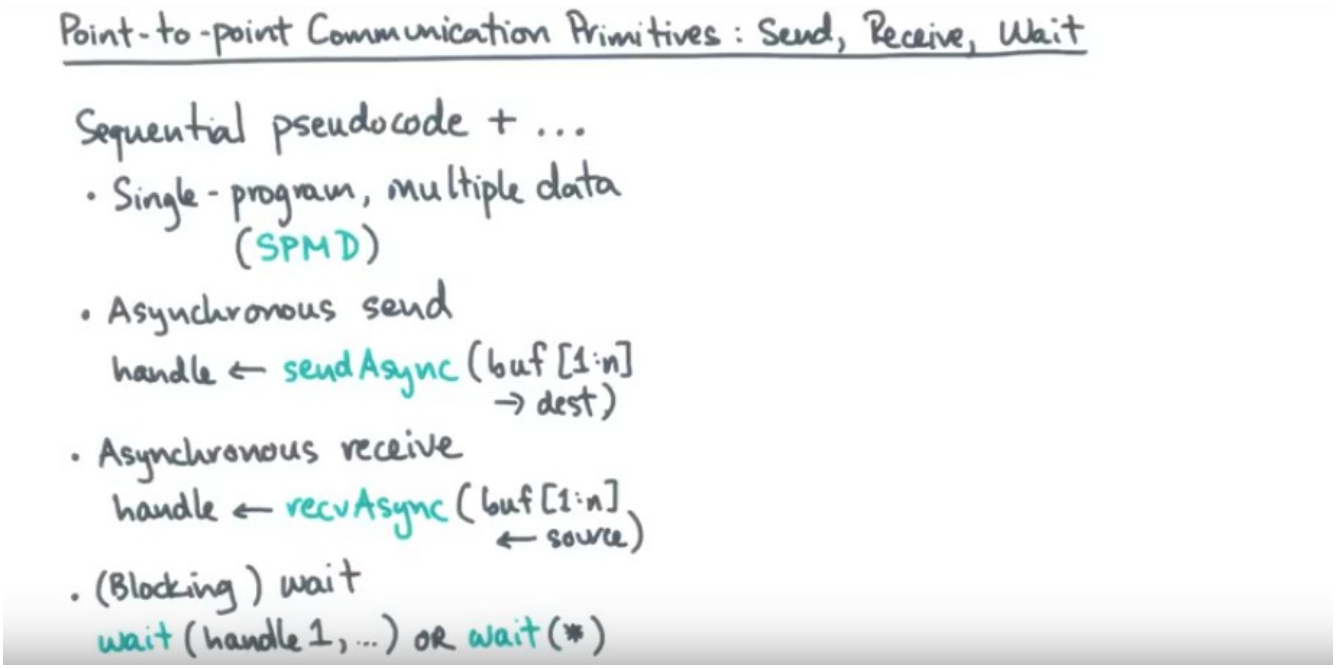
notes. Shave and a haircut, two bits. I don't know what that shave and a haircut two bits thing is, but I got it from somewhere.



17. Beyond the model, you need a pseudocode notation for formalizing algorithms. So let's start with our traditional sequential pseudocode notation and let's make the following changes. First we'll write our algorithms assuming a single-program, multiple data style. HPC nerds refer to this as S-P-M-D or SPMD style. The way SPMD works is as follows. First, you'll write some pseudocode algorithm. Then you'll imagine running that algorithm on some cluster, and you'll assume that that pseudocode is replicated on all the nodes. We'll call every running copy a process. Each process runs independently and asynchronously from the others in the absence of barriers or any other kind of explicit synchronization. Now, to distinguish the copies from one another, we'll assume that the pseudocode algorithm has access to two global variables. One called RANK and one called P. Since the memories are private, these variables are private to each running process. RANK will be the ID of the running process and it'll be unique. P will be the number of processes. Now for the moment, the concept of a **process** and the concept of a **node** are basically interchangeable. But in practice, a process virtualizes the concept of a node. That means you might have more than one process assigned to a node if, for example, you're running on a multi-socket, multi-core system. Now, to see how this works, suppose that this pseudocode algorithm contains a line of code which reads as follows. So when all the processes run, they'll all print statements of the following form. I am 0 out of 5, I'm 1 out of 5, and so on. They each print their own rank and the total.

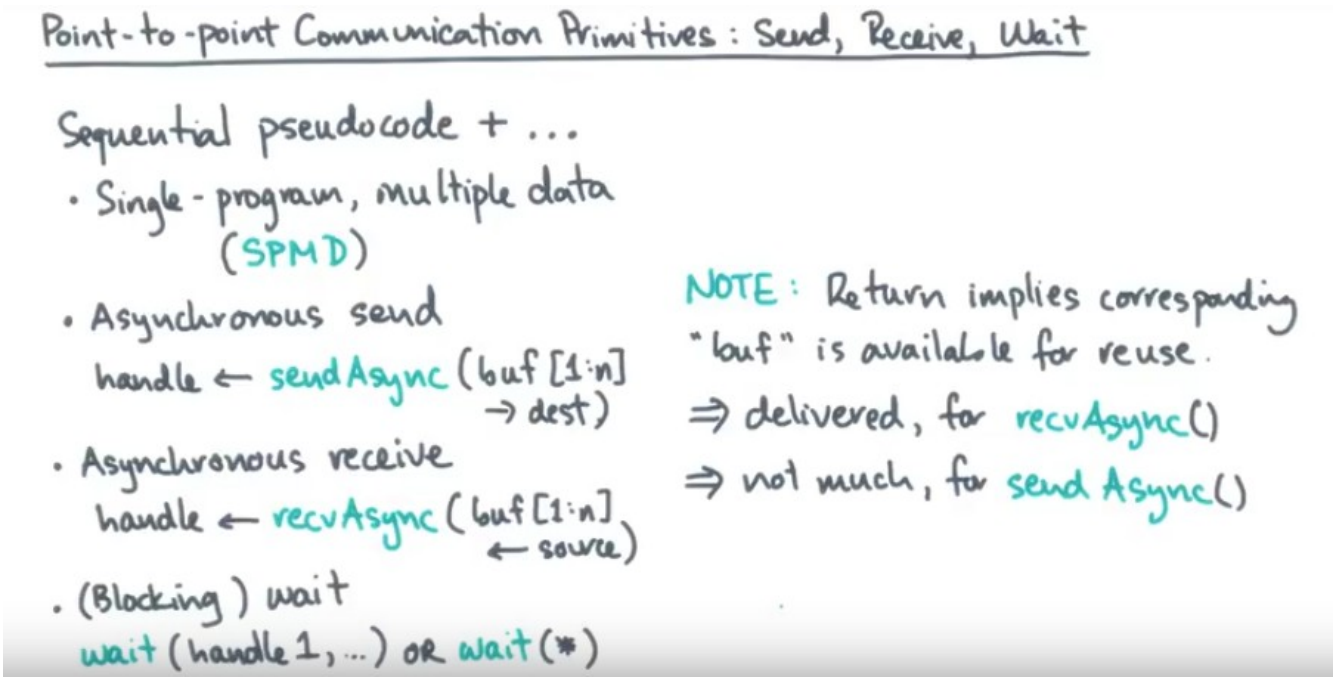


Here's a second change to a sequential pseudocode. I will give you a primitive which performs an asynchronous send. Think of it as an API call that looks like this. Now it has two arguments, one is a buffer of size n. The second is a destination rank, basically the rank of the process that's supposed to receive this message. Now, an important and subtle point about sendAsync is what does it mean when it returns? When it returns, it does not mean that the buffer has been sent. It just means that a send is registered with the system. So in particular, until you know what's happened you should not modify buf. To find out what happened, sendAsync will return a handle. You can do some testing on the handle.



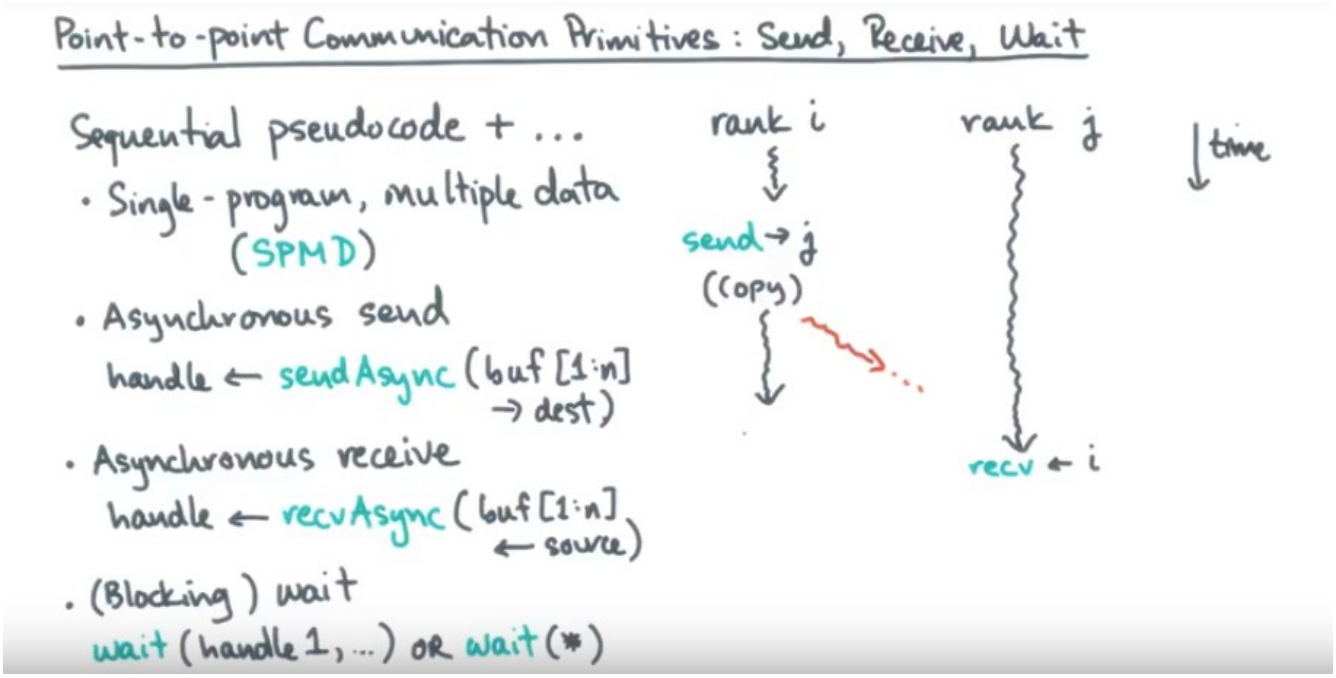
Now, for this send to eventually complete, the destination rank has to post an asynchronous receive. The signature looks the same. It names buffer of some size, and it names a source rank, source being the sender. Just like sendAsync, when recvAsync returns, it does not mean the data is available. Rather, recvAsync will return a handle and you should do some testing on the handle. This business about handles brings us to the final primitive which is called wait. In particular, **wait** is a blocking operation that takes one or more handles as arguments. Now, wait will pause until the corresponding operations complete. Now there's also a special form of wait called a wait all, abbreviated here by wait with an asterisk. This is a shorthand that says wait for all outstanding sends and receives so that we

don't have to track and check all the handles all the time.



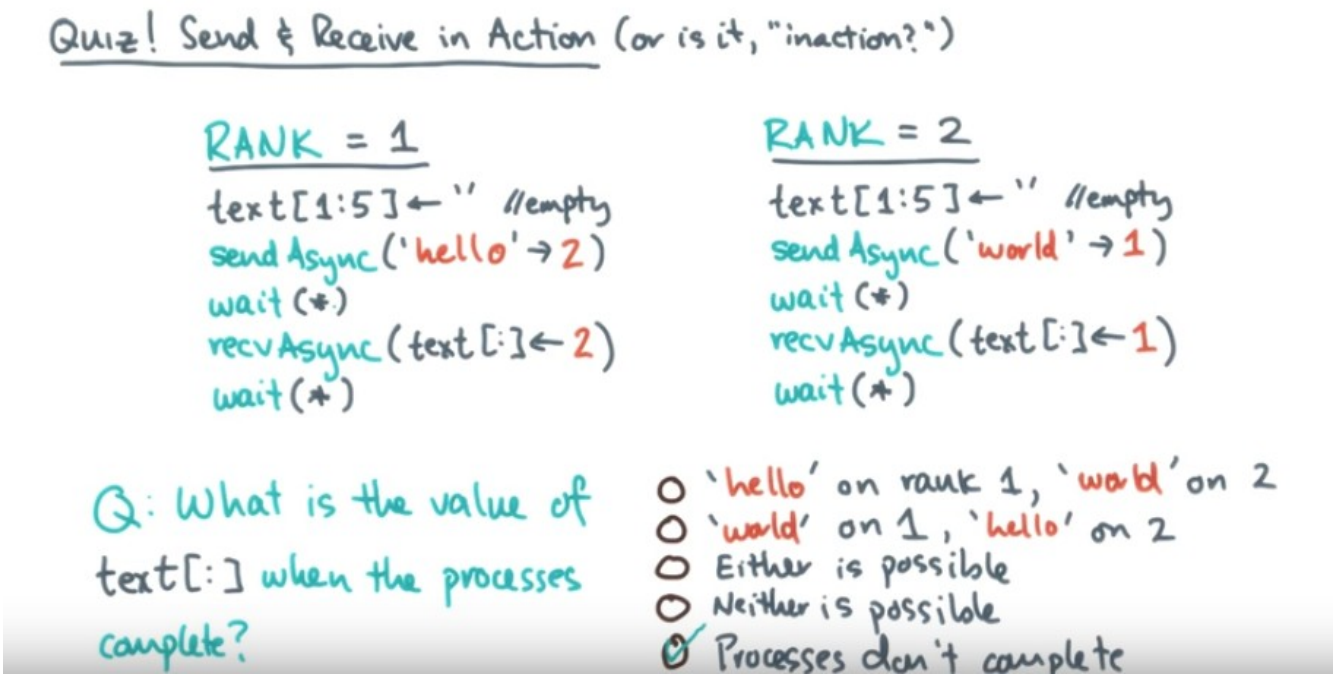
由後面知, sendAsync(buf → dest)和 recvAsync(buf ← dest)中的 buf 不是同一個數組, 它們分別是不同電腦上的 local array.

18. Now here's a very important and subtle point about the completion of a wait. When wait returns, the only thing that you know is that **buf** is safe to reuse. If the handle is for a receive, then it must mean that the message was delivered. But what about send? For a send, buffer being available for use doesn't really tell you anything. For instance it could mean the message was delivered since the wave might actually wait until the message is delivered, but it doesn't have to mean that. In fact it might not even be the case that the message is even in flight yet. The precise action is implementation dependent. Implementation dependent.



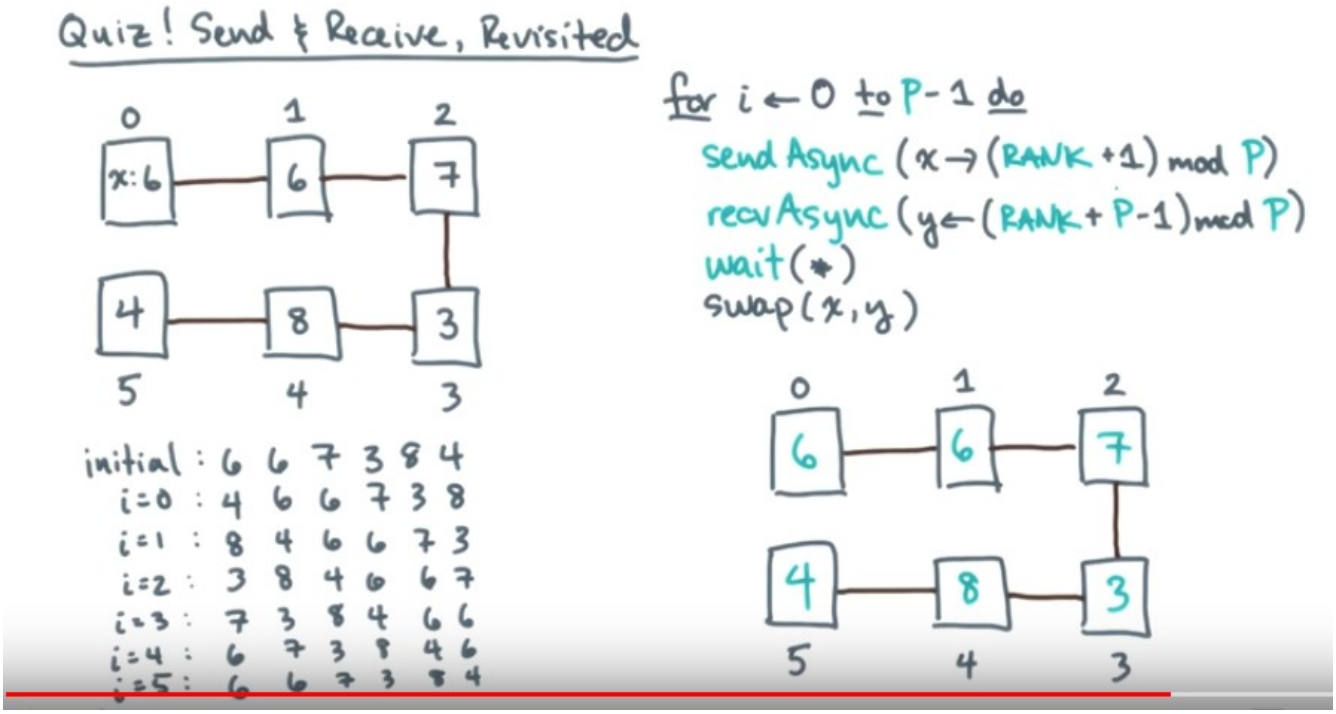
But why? The answer has to do with buffering implied by these primitives. Remember that the processes are executing asynchronously. Also remember that a send completes when the caller may reuse the message buffer. So let's imagine what might happen. Here are two ranks, i and j. Rank i is computing, and then decides to send a message to j. In the meantime, j is running off asynchronously for a long, long time before it decides, oh, I'm ready to receive a message from i. So the send could arrive way before the receive is actually ready. So should the send stop and wait until the receive has been posted before it starts delivering? In this scenario, rank i would have to wait a long, long time. What rank i might do instead is make a copy and then later on decide to start the delivery. That way rank i can kind of keep going. So by leaving the meaning of the completion of a send a little bit loose, you give more freedom to the implementation to make a good trade off.

Okay, so what's the punchline of all this? Well, it means that when you're trying to determine if your algorithm is correct, meaning that a sent message is received, you need to prove that for every posted send, there is a matching receive. This protocol that sends, match, receives is called two-sided messaging. Two-sided suggests there might be a one sided, and in fact that's kind of what shared memory communication is. I'll let you think about that.



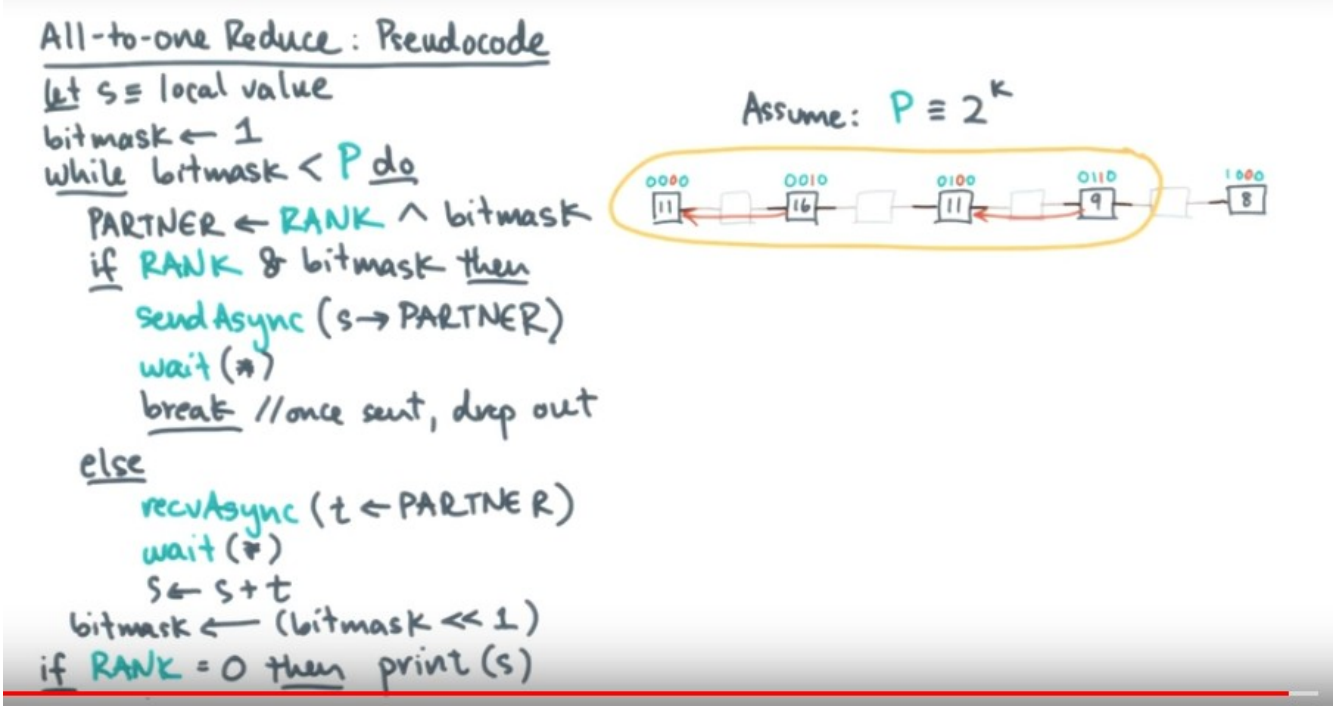
19. Okay, time for a quiz. Suppose there are two processes, each of which executes the following operations. So, they each try to send messages and they each try to receive messages. The buffer for receiving is called text. Here's my question. What is the value of text when the processes complete? Does rank one's text have the value hello, while rank two has the value world? Or is it just the reverse? Or maybe it's either, or maybe it's neither. Or maybe the processes don't even complete. You decide.

20. The answer is that the behavior is undefined. The processes don't complete. They both initiate sends and then wait for the send to complete. But remember, there's an ambiguity about what complete means in the case of a send. It might be that there's some copying going on, but it might be that the message is very large and the implementation has decided to wait for delivery. In the latter case, that's a problem because it means that neither process can ever execute its receive. The result is deadlock, each process is waiting for the other to do something, but no one does it.

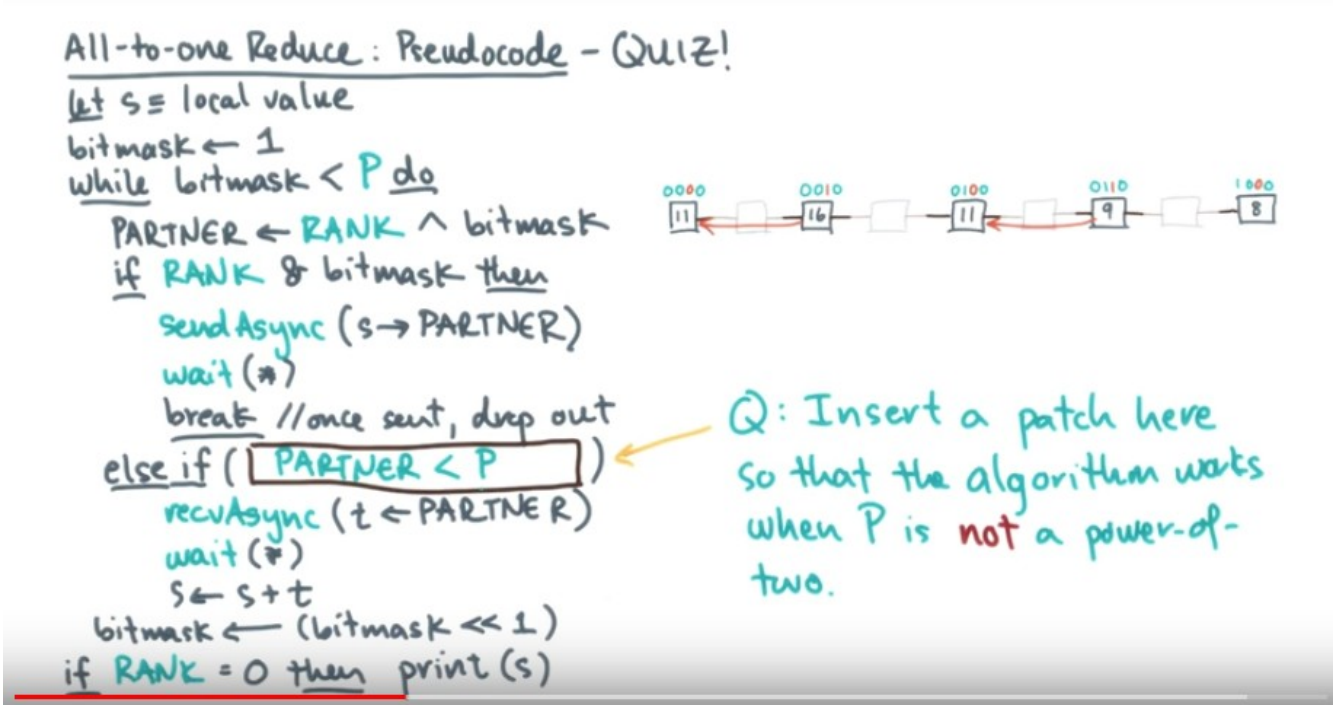


21. Consider six processes. Each of which has a buffer named x. X is large enough to hold a single value. The initial values of x are as shown. Suppose we run this pseudocode program. If this is the initial state, what is the final state of x on all the nodes? I want you to fill in these blanks with the final values. And if for some reason, you think a value is undefined, put in a question mark.

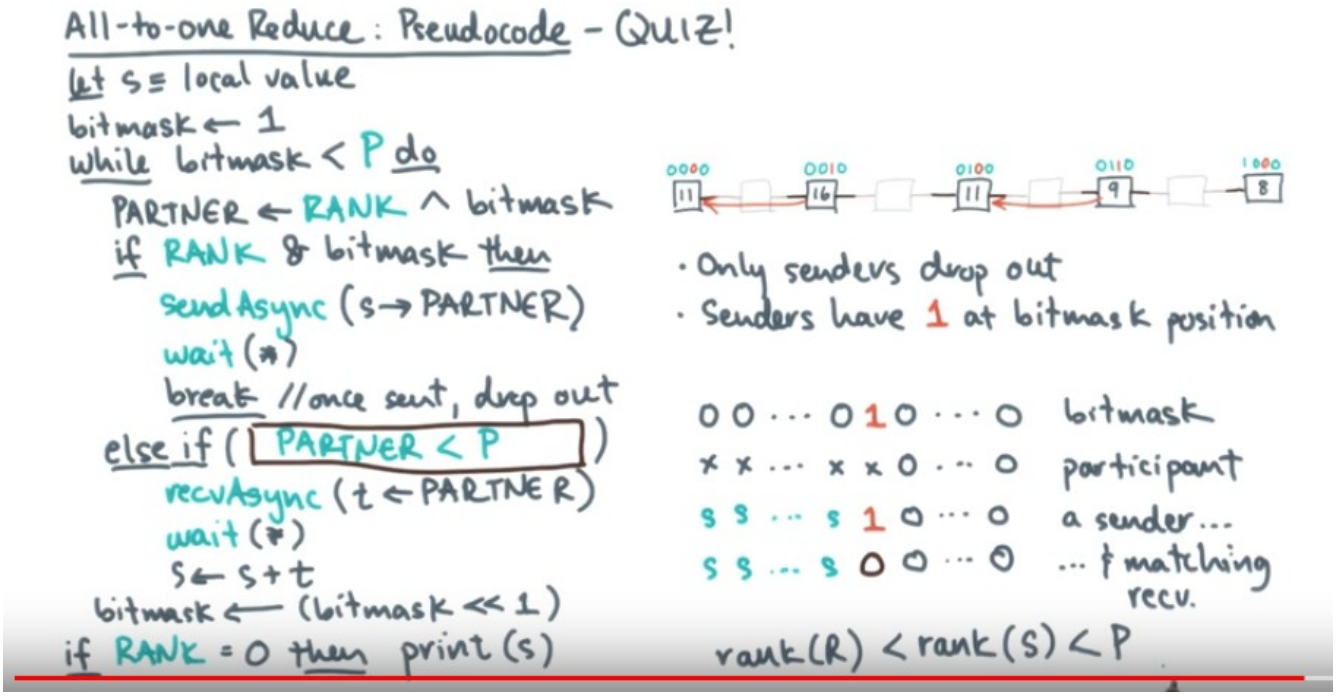
22. So here's my solution, which is basically exactly the same values. To help you see that I've included a little trace, here on the left. This row is the set of initial values. This row is what happens after i = 0 completes. Basically all the values shift by one (如 initial 的 667384, 4 跑到最左邊去). because shifting by one is essentially what this funny indexing corresponds to.



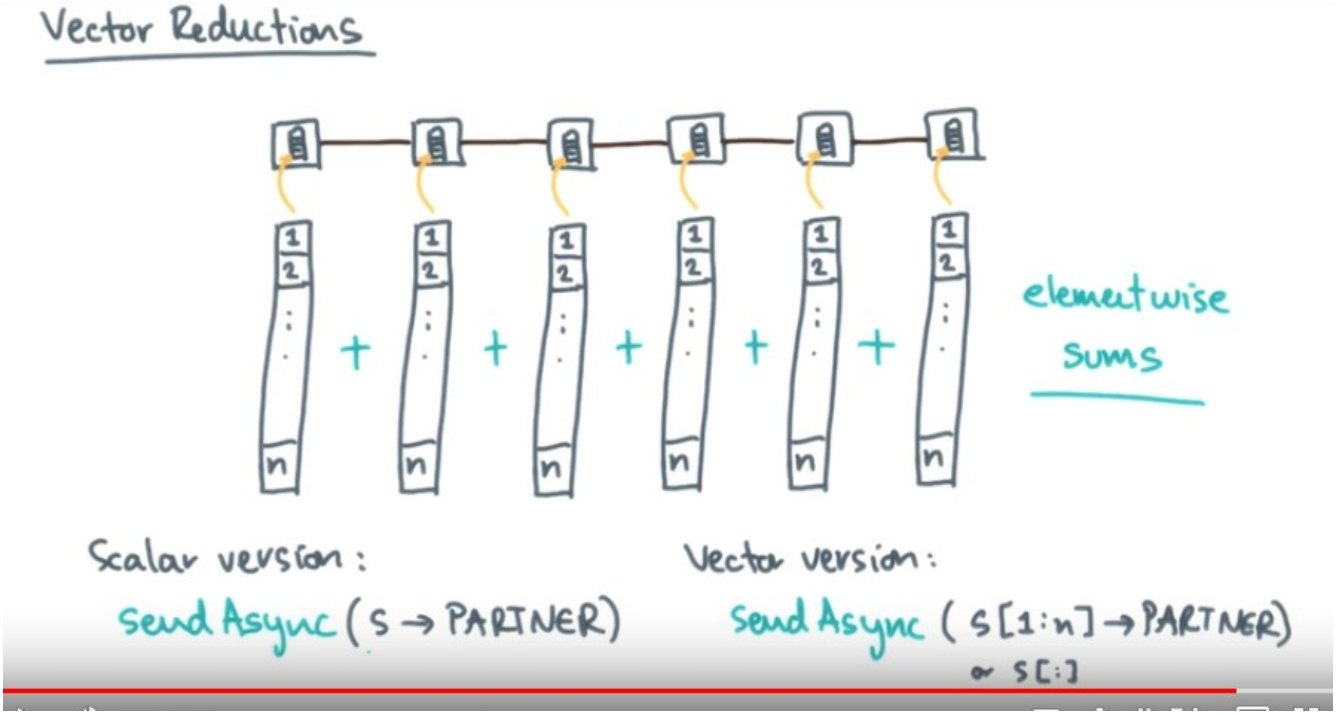
23. If you understand the SPMD model and the asynchronous communication primitives, then you're ready to write the pseudocode for an all-to-one reduction. So recall the problem and our algorithmic idea. There are p processes. Each process has a value, and we want to compute the global sum of these values, leaving the result on some root process, let's say process zero. Our algorithmic idea was to use a tree-based reduction. The algorithm proceeds in about log P rounds of communication. The rounds iterate over bit positions. In each round there's a current bit position. Processes with a one send to processes with a zero in that bit position. Okay so let's sketch some pseudocode. For simplicity, let's assume that P has a power of two, so we'll ignore this oddball. Let's have an outer loop that maintains a bitmask and we'll shift the bitmask after every round. The next part of the algorithm is to have pairs of processes communicate. The partner of any given rank differs only in the bitmask position, so this is an exclusive or. Remember that senders had a one in the current bit position, so if you're a sender, you need to send a message. So if S is our local value, then the send can just send that value to a partner and then wait. Once the send is complete, the sender drops out. Now if we're not a sender, then we might be a receiver. And if P is a power of two, it will turn out we must be a receiver. That case, we'll receive the value from our partner and then accumulate it. Now in this scheme, the final result will end up on rank zero. So for rank zero we can print the final result.



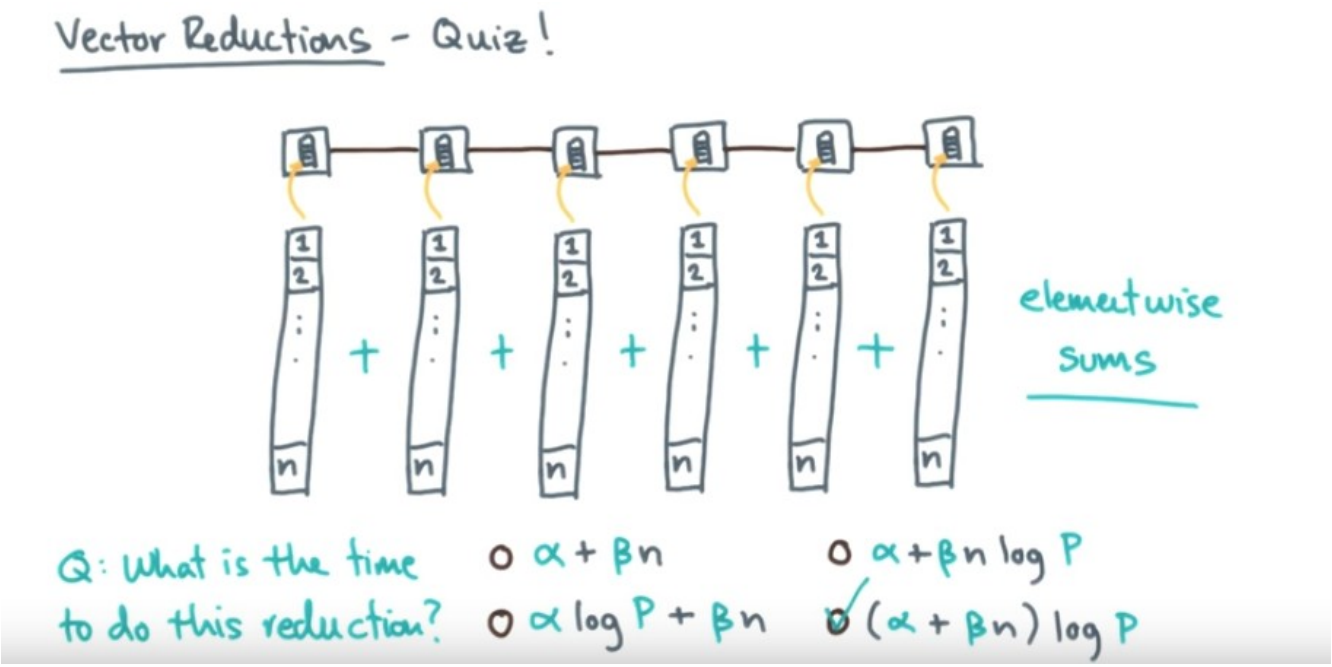
24. This algorithm performs an all to one reduce leaving the final result on rank zero. And it works provided P is a power of two. That seems like an annoying restriction let's see if we can get rid of it. When P is not a power of two I claim you can patch the code by inserting something here. So what I want you to do is come up with a patch and stick it in there.



25. Okay, here's my solution. I suggest that you check whether a partner is less than P, and you only post the receive if that's true. So if partner is less than P, you'll post the receive, and you'll continue on. If partner is not less than P, then you just won't do anything, but you'll keep executing the while loop. Now if you go back and you look at the animation, you'll see that in fact there was an oddball processor who kind of stuck around until the very end. And I claim this patch will give you that behavior. Let me put it a little differently. Start by considering the following two facts. First, only senders drop out. Second, senders always have a 1 at the current bitmask position. Let's think about that. Suppose the current bitmask looks like this. So it's 0 everywhere except 1 in the current position. The two facts imply the following. Any process that's still participating, meaning it hasn't dropped out yet, must have a bit string that looks like this. In particular, All the least significant bits will be 0. P xs are don't care bits. Here's another handy fact. A sender must have a 1 in the bitmask position, so its bitstring is going to look like this. Its partner will have a flipped bit, so the matching receive bitstring will look like this. This means that the sender's rank is always greater than the receiver's rank. And, of course, to be a valid sender, the rank has to be less than P. So a sender's rank is less than P. And a receiver's rank is less than the senders rank, which in turn will be less than P. So a candidate receiver who's partner has a rank greater than P can't be valid.

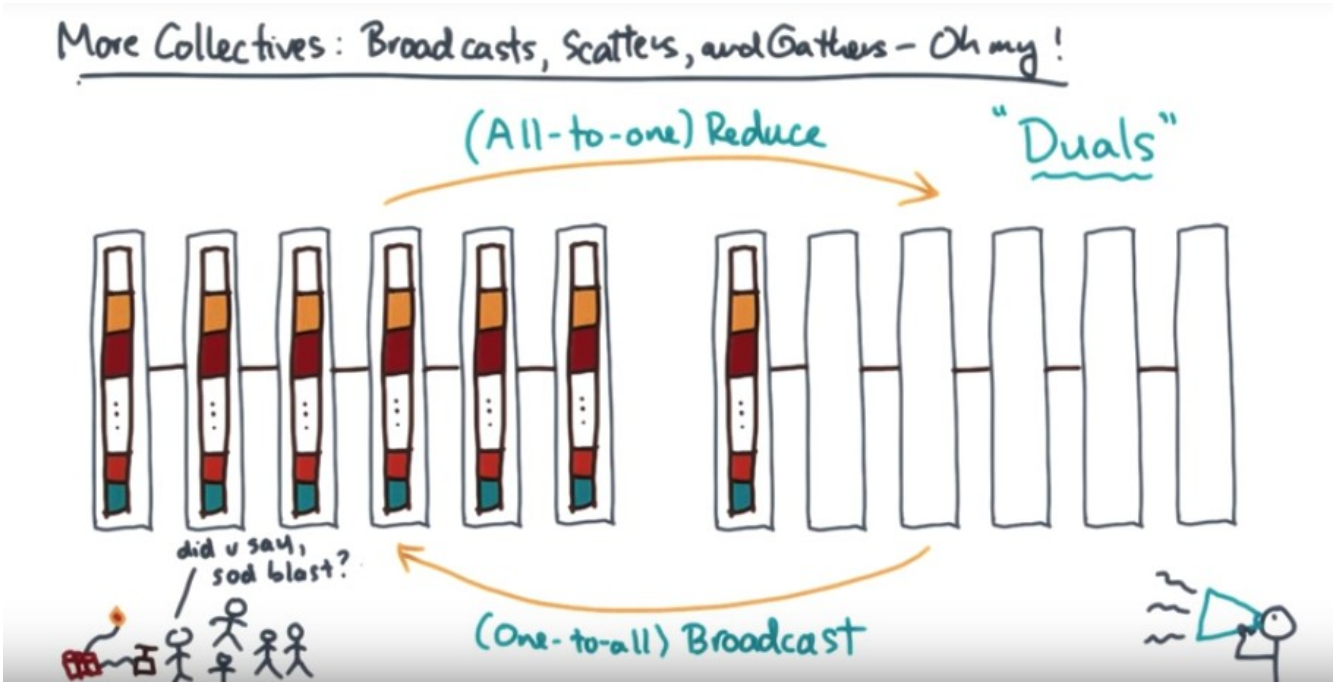


26. Let's consider a slight generalization of a reduction called a vector reduction. In it, we'll assume that every node has not a single word of data, but a vector of data. A vector reduction will be the elementwise reduction of these vectors. So if we're doing a sum reduction, then we'll take vectors, and we'll sum them elementwise. The result will be a single vector of length n, whose elements are the sums of the corresponding elements from the other nodes. Now the corresponding algorithm would change only very slightly. In our scalar pseudocode, anywhere we sent a scalar, we could instead send the vector.



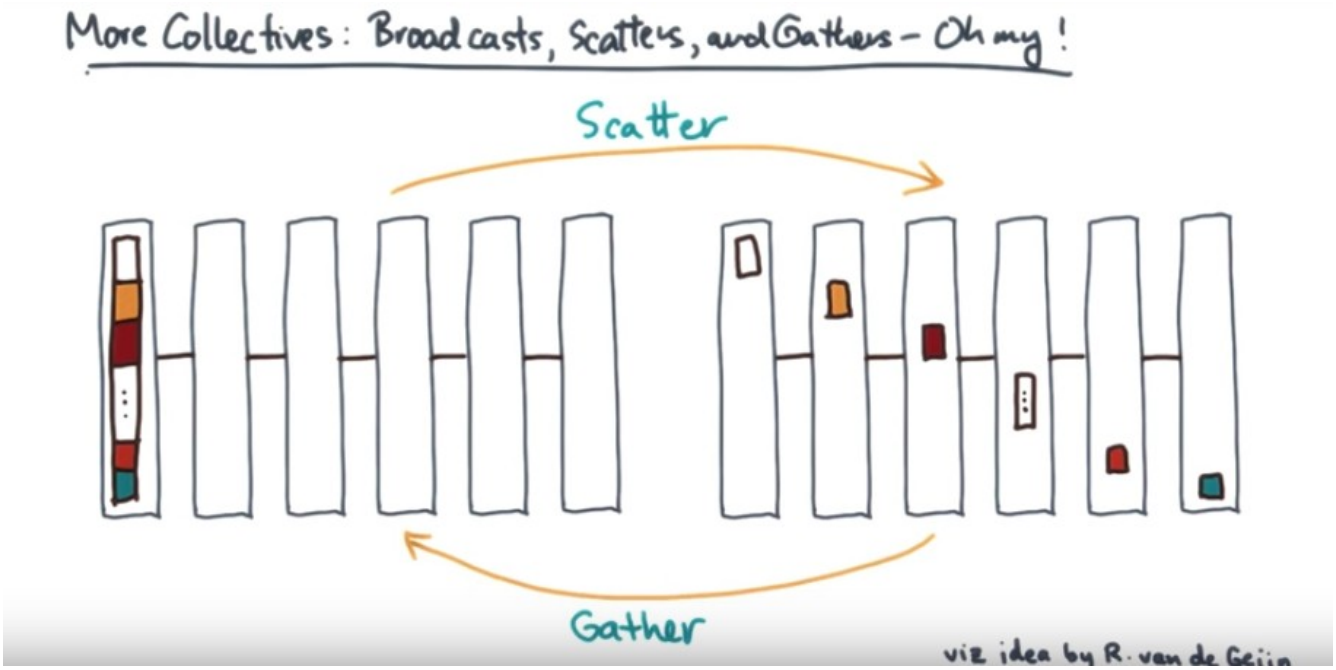
27. Here's another question for you. What is the total communication cost of doing a vector reduction? Assume a linear network and our usual tree-based algorithm. You can also assume that the number of nodes is a power of two. I'll make this multiple choice with the following options, pick one.

28. This is the answer. We're sending full vectors at every round, and we've got log P rounds of communication.

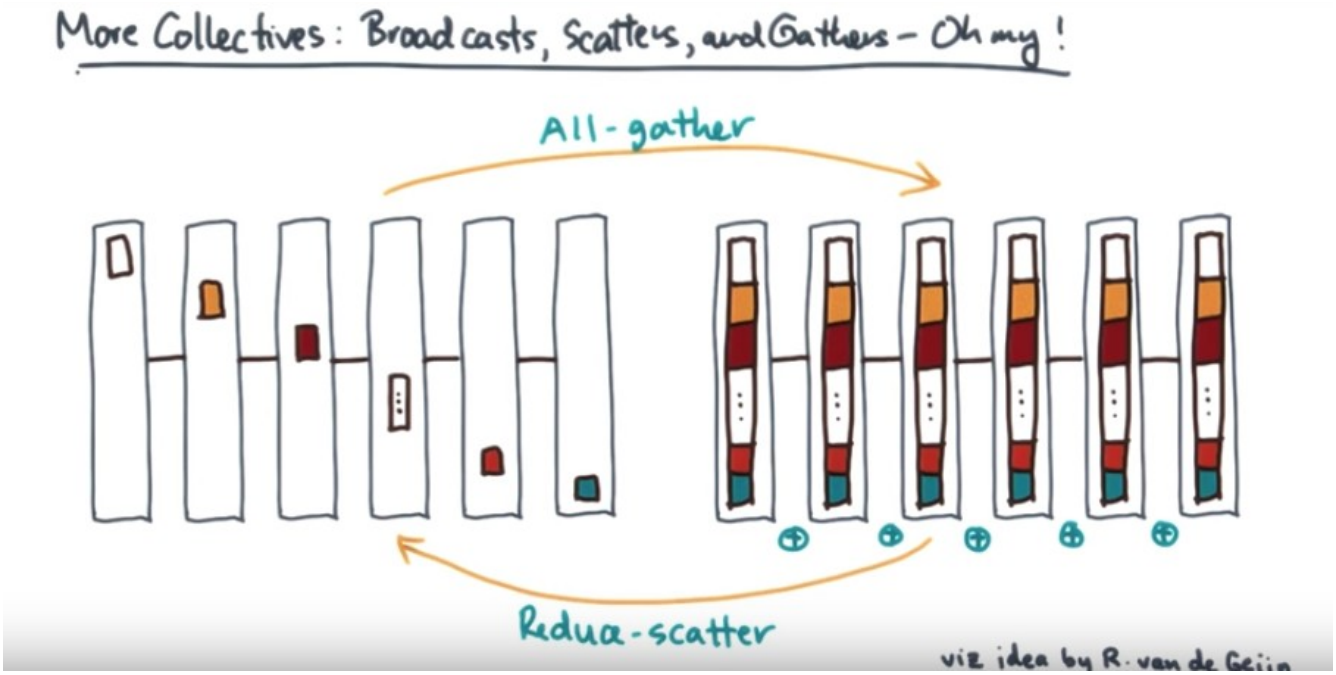


這些 collectives (reduce, broadcast, scatter, gather, all-gather, reduce-scatter) 的意思稍後在它們的 function signature 中會詳細講, 此處若不理解, 沒事.

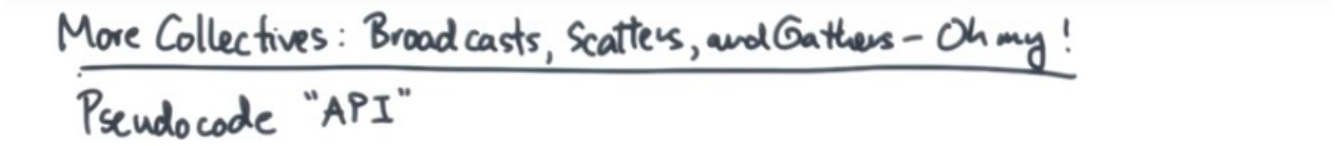
29. So far, we've talked about all to one reduce. Start with a set of nodes on a linear network. Initially, each node has a vector of data. Performing a reduce produces the combined data on one of the nodes. This node that stores the result is called the root. Now a reduce has a natural partner (意思是有一個跟 reduce 相反的操作). It's called a one-to-all broadcast, or just broadcast. In a broadcast, one processor has all the data initially. It wants to send a copy of this data to all other processes. So it's basically one process shouting in a room to all others. So in principal if you know an algorithm for reduction, then you also know an algorithm for broadcast. You basically run the reduction in reverse. Because of this relationship between Reduce and Broadcast, I'll refer to them as Duals of one another.



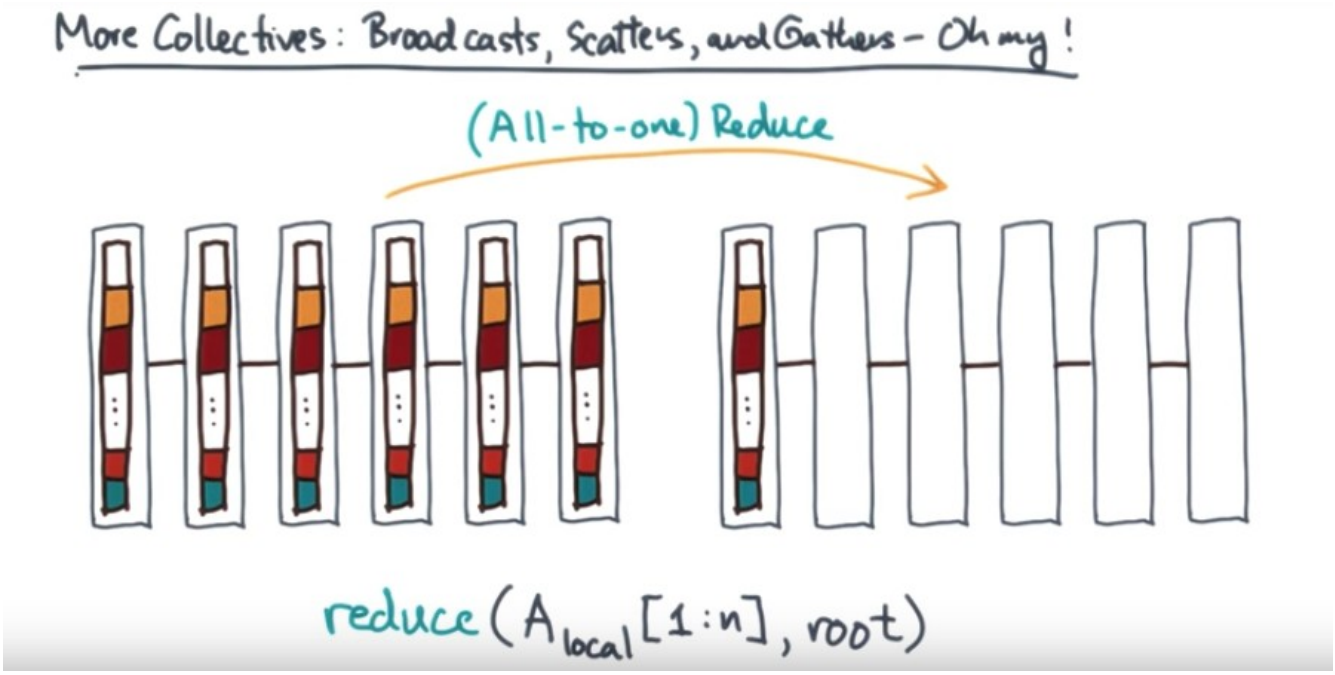
Here's another useful collective operation. It's called a Scatter. In a Scatter, one process starts with a data. It then sends a piece of its data to each of the other processes. A scatter also has a natural dual. The dual is called a Gather. In a Gather, everyone has a piece of data, and one node collects it. How do you like my primitive cheese people?



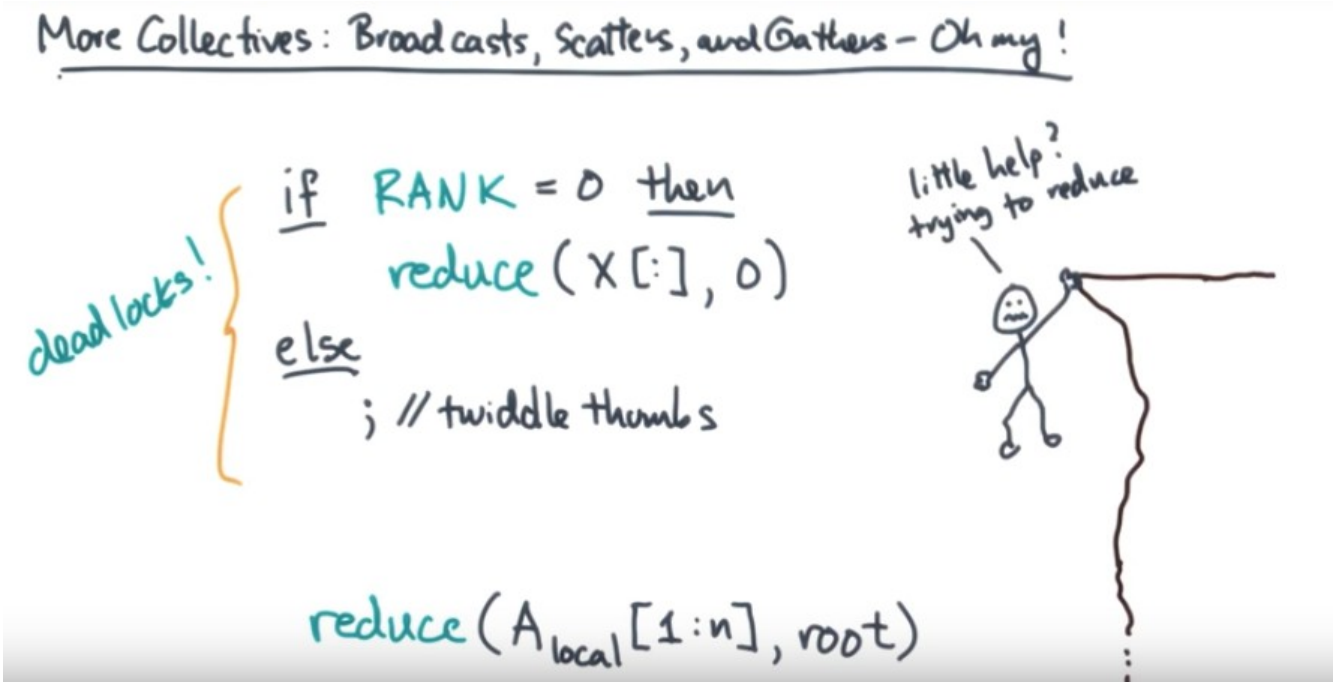
Another useful primitive is an All-gather. Initially each process has a little bit of data. After the All-gather, all processes have a copy of all of the data. And All-gather also has a natural dual called a Reduce-scatter. In a reduce-scatter, initially all processes contain a vector of data. They then globally reduce this data using some kind of combining operator like a vector addition or vector element-wise multiplication. The result is distributed to all processes. By contrast, recall that a normal reduction, an all to one reduction, leaves all of the results on just one process. I'll show you some techniques for implementing all of these collective operations efficiently. I learned these techniques from a fellow by the name of Robert van de Geijn. Once you've learned those methods, you can then simply invoke them as collective operations in your algorithms whenever you need them.



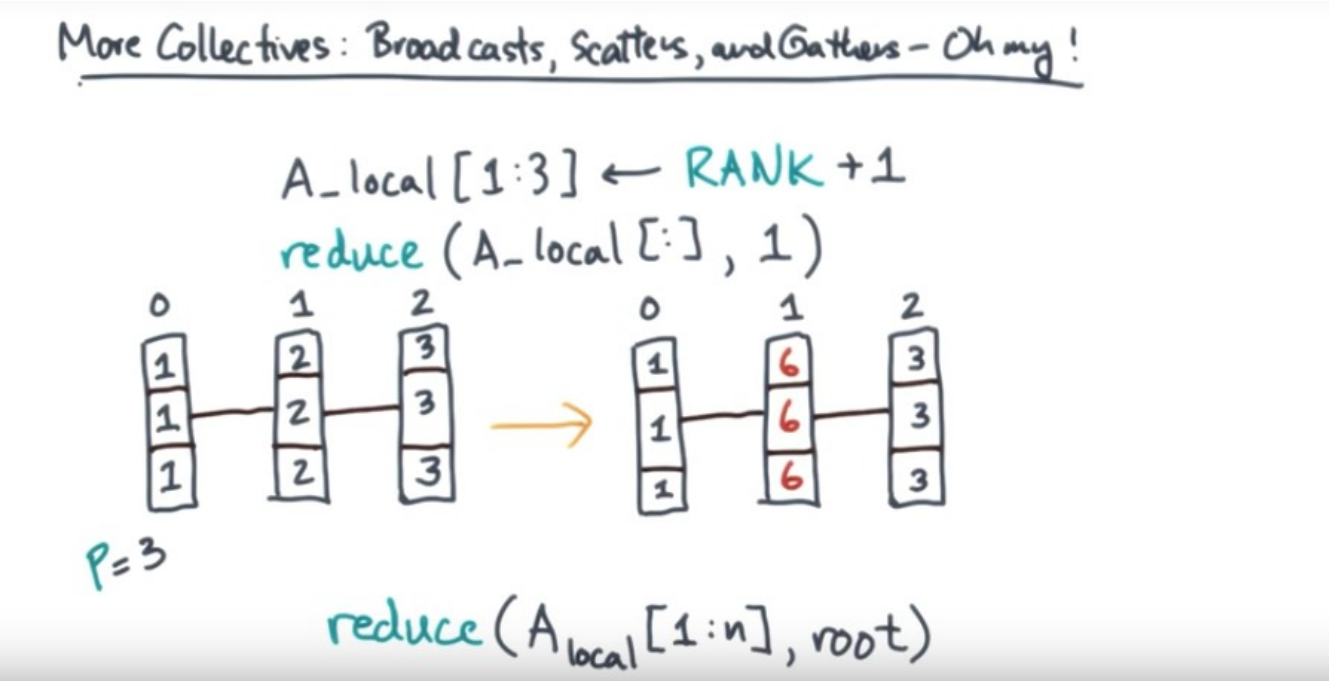
30. In fact, to make it easy to use these as building blocks for your algorithms, let's define some pseudocode interfaces.



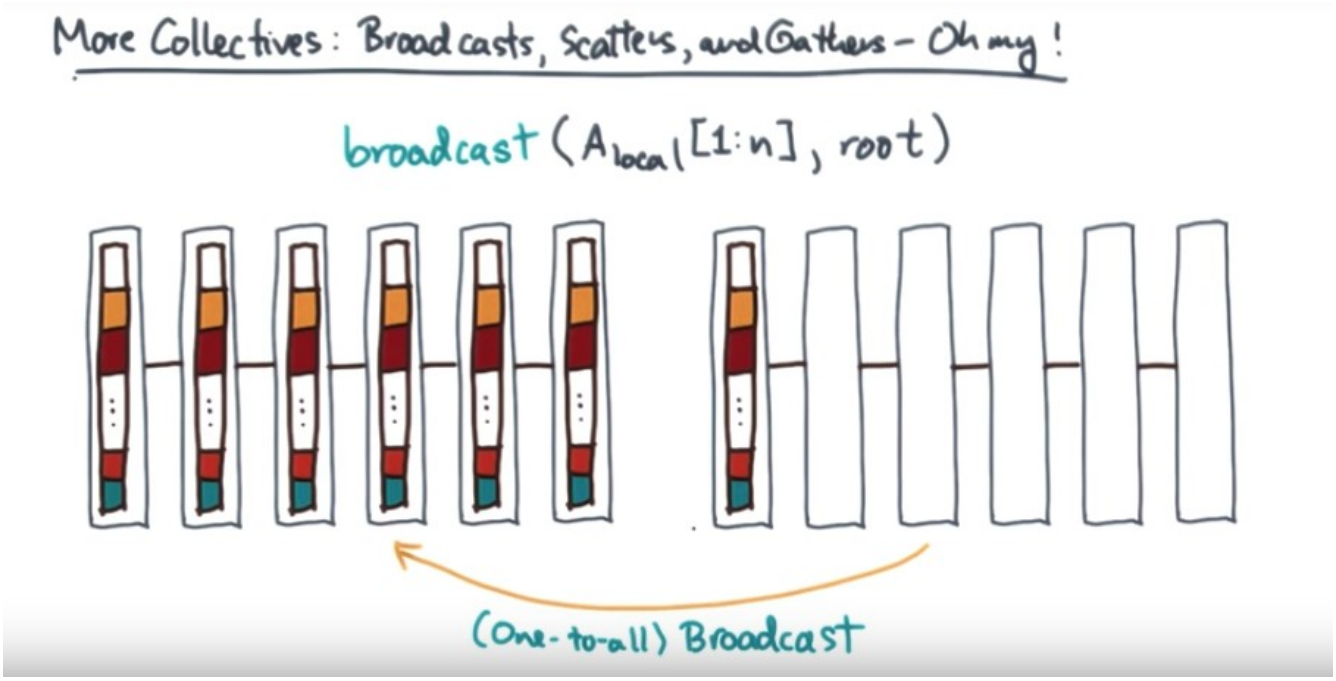
Suppose every process has a private array of size n. To perform a vector reduce on all of those arrays you would just write this. Every process needs to call this reduce, supplying as input its private array, along with the ID of the root process. This operation will perform the reduction. It will leave the final output on the process whose rank is root. And that result will be in root's private array. Now, this operation is called a collective, meaning that your algorithm or program must be structured in such a way that all processes execute it.



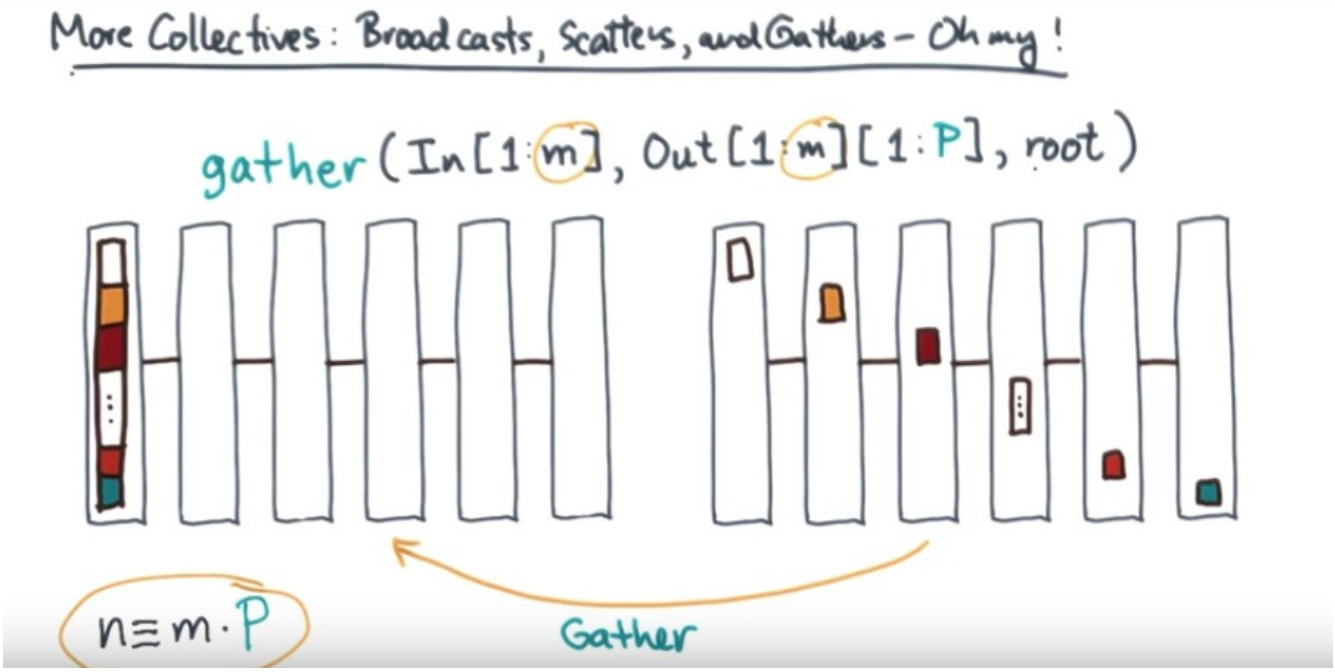
Let me show you what I mean. Here's an example of a program. Remember, all processes are executing this program concurrently. Notice that only the process whose rank is 0 will actually call the reduce. None of the other processes call reduce. So this program is actually invalid and its behavior would be essentially to deadlock. Put another way, this program hangs because rank zero starts to reduce but not other process does. So that leaves rank zero hanging.



Here's a different example program. What is its output if there are say, P = 3 processes? Say initially we have 3 processes. Each one initializes its local private array with its RANK+1. Then all processes enter the reduce. Notice that the root is the process whose rank is 1. So the final result should appear on rank 1. So after the reduce, the result should look like this. A_local is unmodified on ranks 0 and ranks 2. The final output appears on the root.

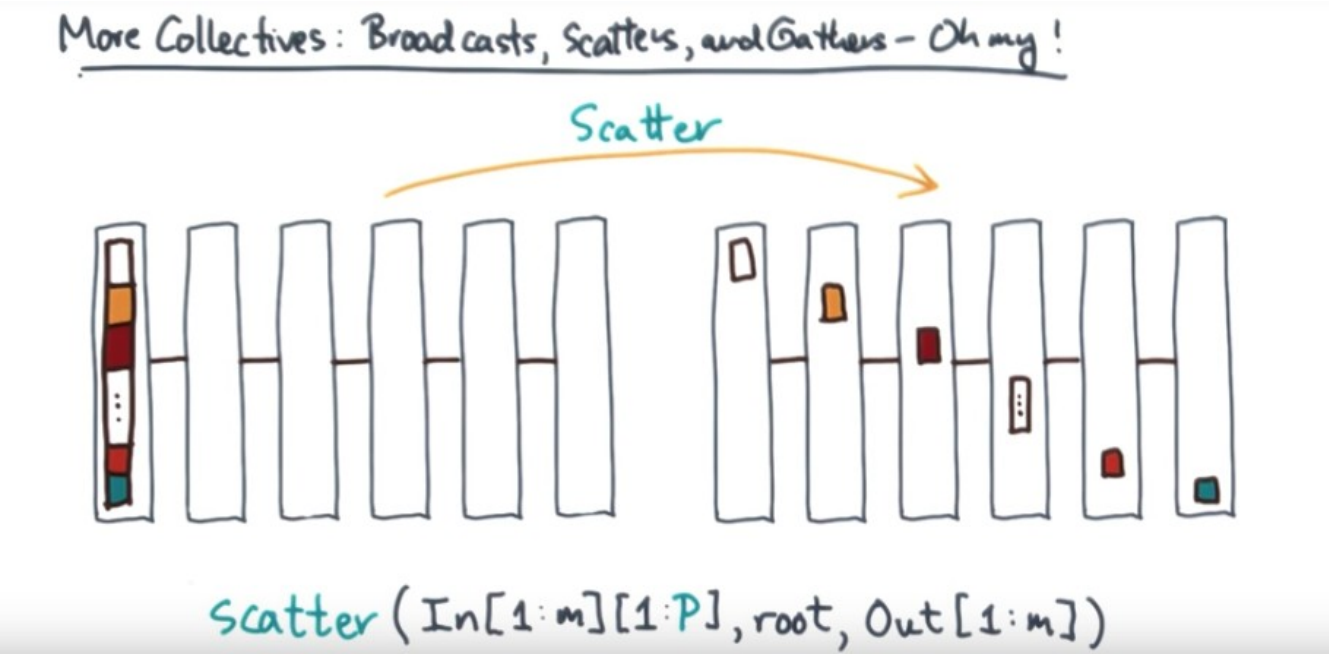


Okay, let's move on. For a broadcast, the dual of reduce, I want you to assume this signature. The process whose rank is root will have the initial data in its local buffer. When the broadcast completes, every process will have the same data in its local buffer.

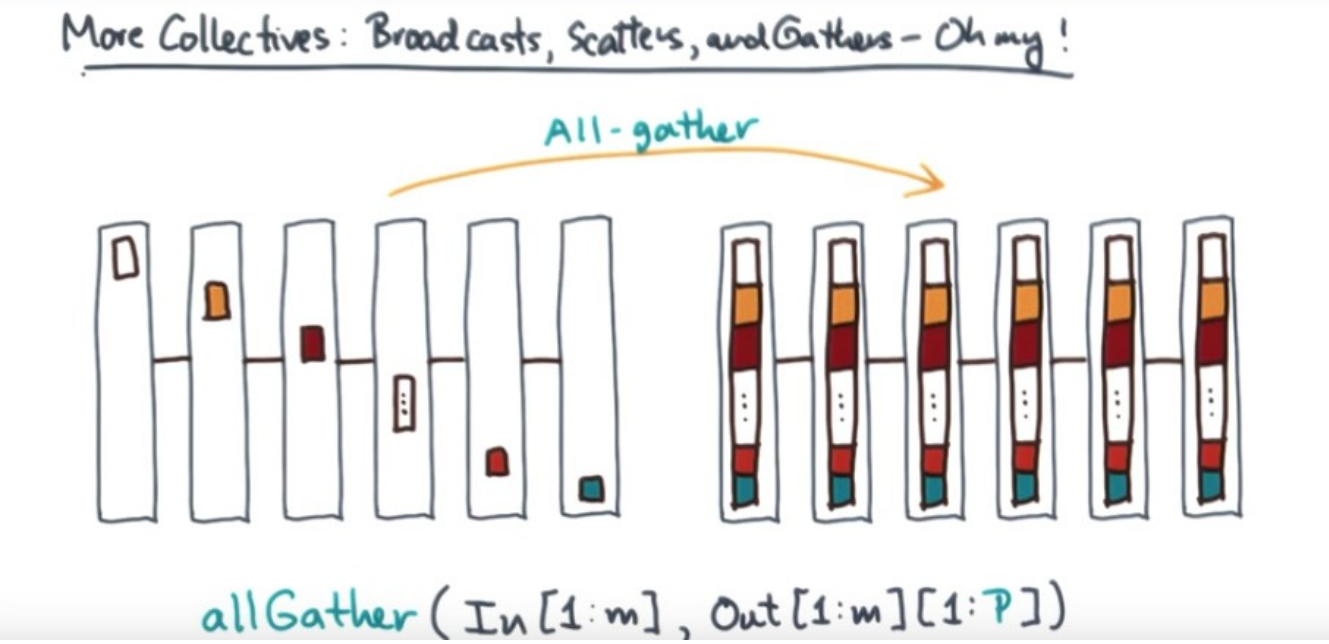


So for a gather, I want you to assume this signature. Remember that in a gather, everybody has a little piece of data and you want to collect all the data on one process. So every process has an input buffer of size little m. The output is all m buffers from all processes collected on the root. Notice that the output buffer is two-dimensional. Now the output buffer need only be defined on the root process. So what does that mean? That means in your algorithm, when everybody calls gather, output is only valid on root. In fact, you can assume that the other processes have no output buffer, unless the algorithm

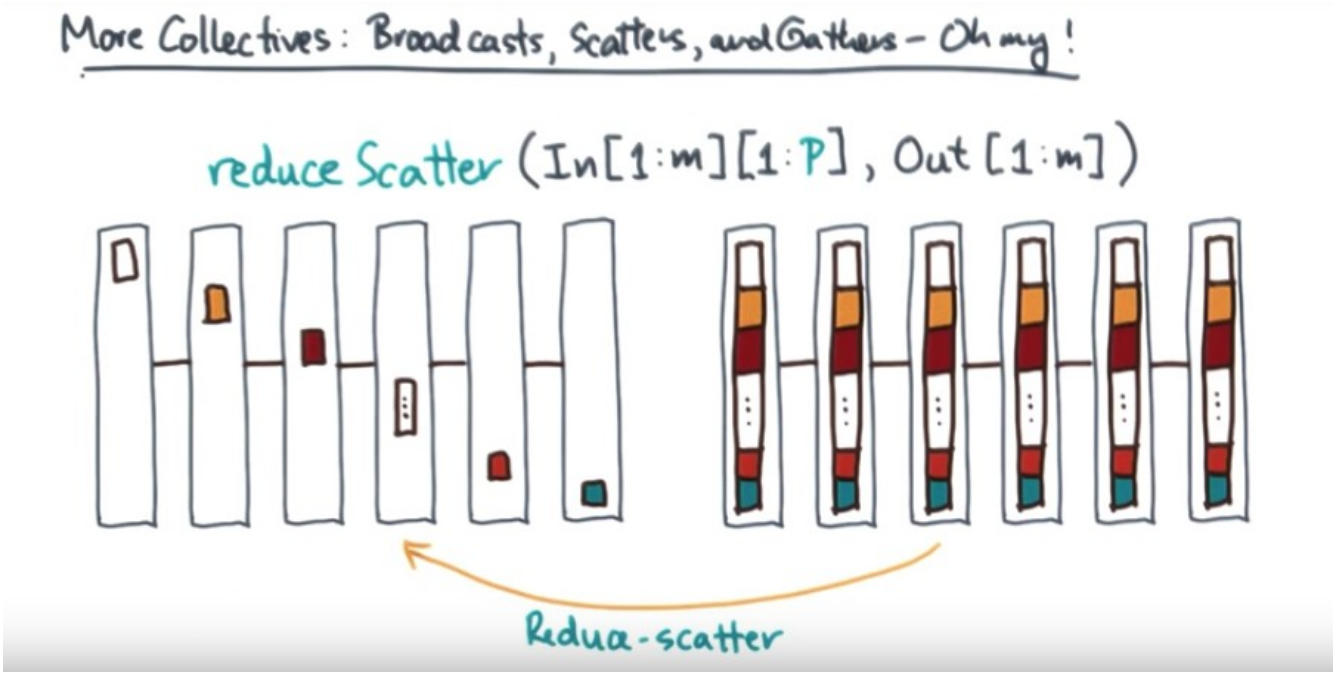
needs it for some reason. Now notice I've used the symbol little m instead of little n. The reason is so that when you do analysis, I want you to assume that our usual problem-sized variable, n, is defined to be m times P. That (n) is the size of the combined output. This is basically just saying that we always want to assume that P divides n. Of course, in a real message passing library there will be ways to send variable amounts of data per process.



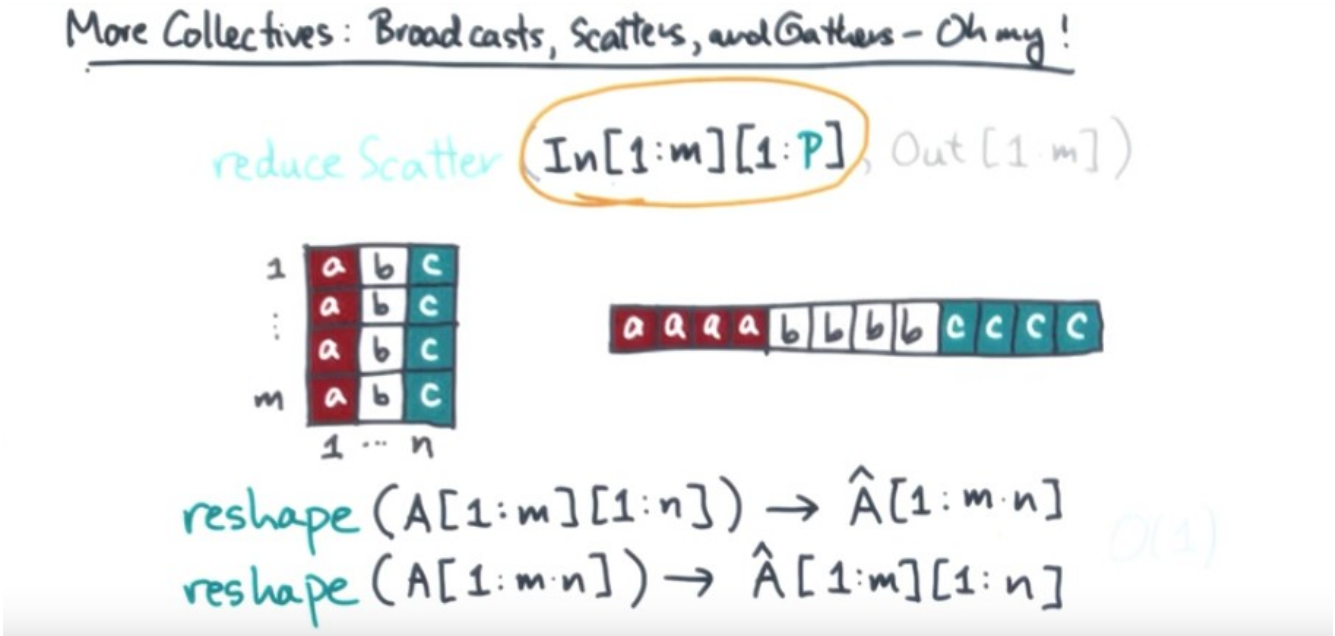
Now the dual of the gather, or a scatter, looks essentially the same. The difference is that the input is of size m times P defined on the root.



Finally, we come to allGather and reduce scatter. Here's allGather. Note that there's no concept of a root rank. The allGather essentially replicates the input buffer everywhere.

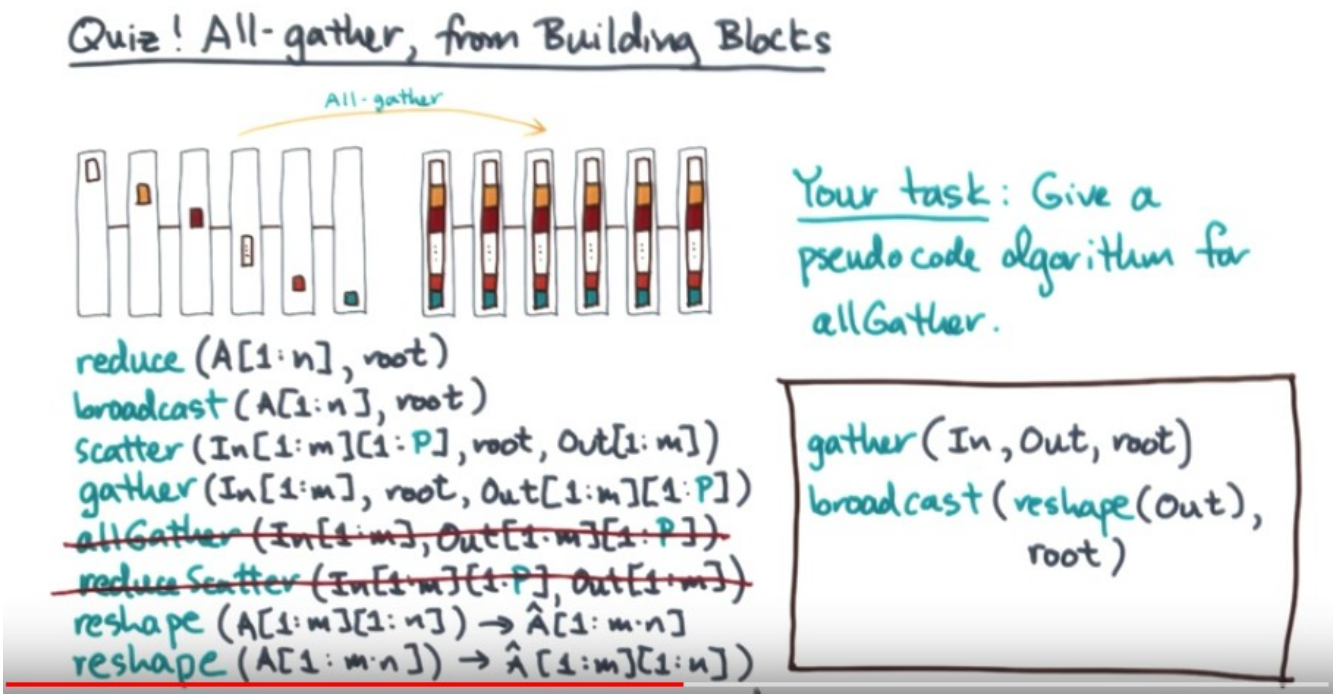


Now reduceScatter is the same as an allGather just going in the other direction. So in particular, the dimensions on the input and output buffers are reversed relative to allGather.



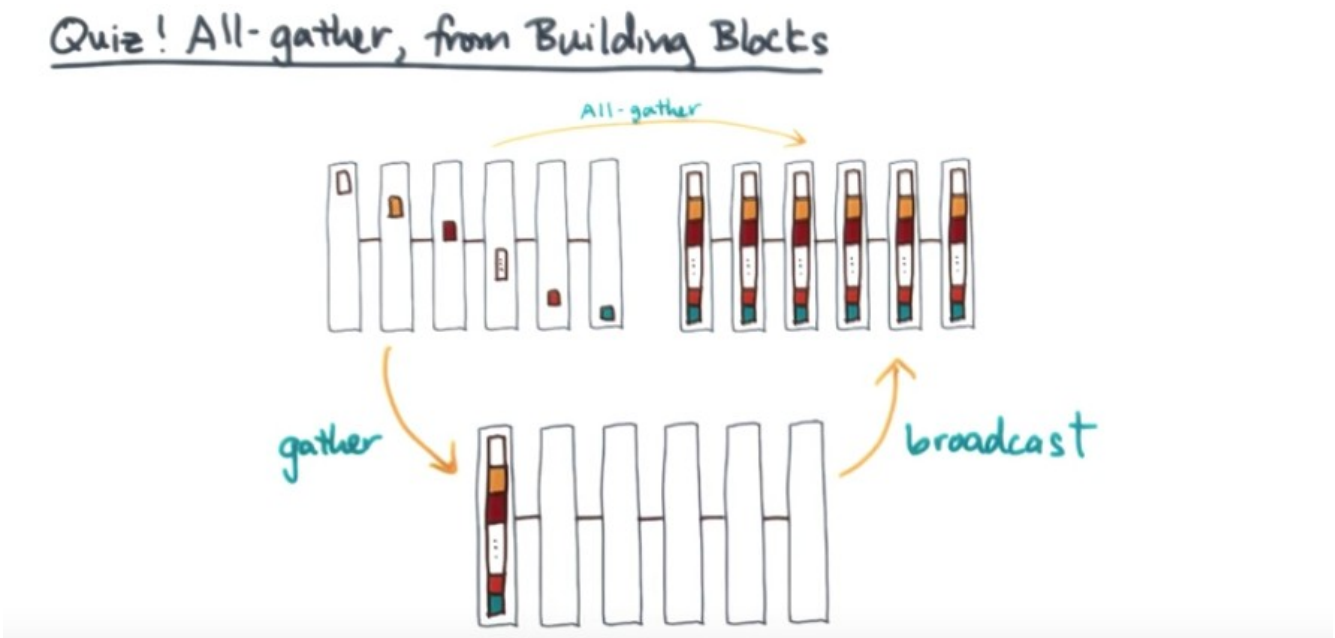
Now we've been using these two-dimensional objects, so I want to introduce one more useful primitive. It's called a reshape. It has two forms. One takes a two-dimensional input and produces a one-dimensional output. The other goes from 1-D to 2-D. This primitive is a purely logical operation, rather than a physical one. By that, I mean it doesn't actually copy a 2-D object to a 1-D array. It just says that it changes the interface of the object so that you can start viewing it and operating on it as if it were 1-D. In fact, if you're familiar with multi-dimensional arrays in languages like C and C++, you know a dense multi-dimensional array always maps to a linear address base anyway. As such you can assume it has constant cost. Just to be concrete, here's a picture of going from a logical 2-D

representation to a logical 1-D representation. The convention I'll use linearizes the 2-D array column by column. This is sometimes called column major storage.

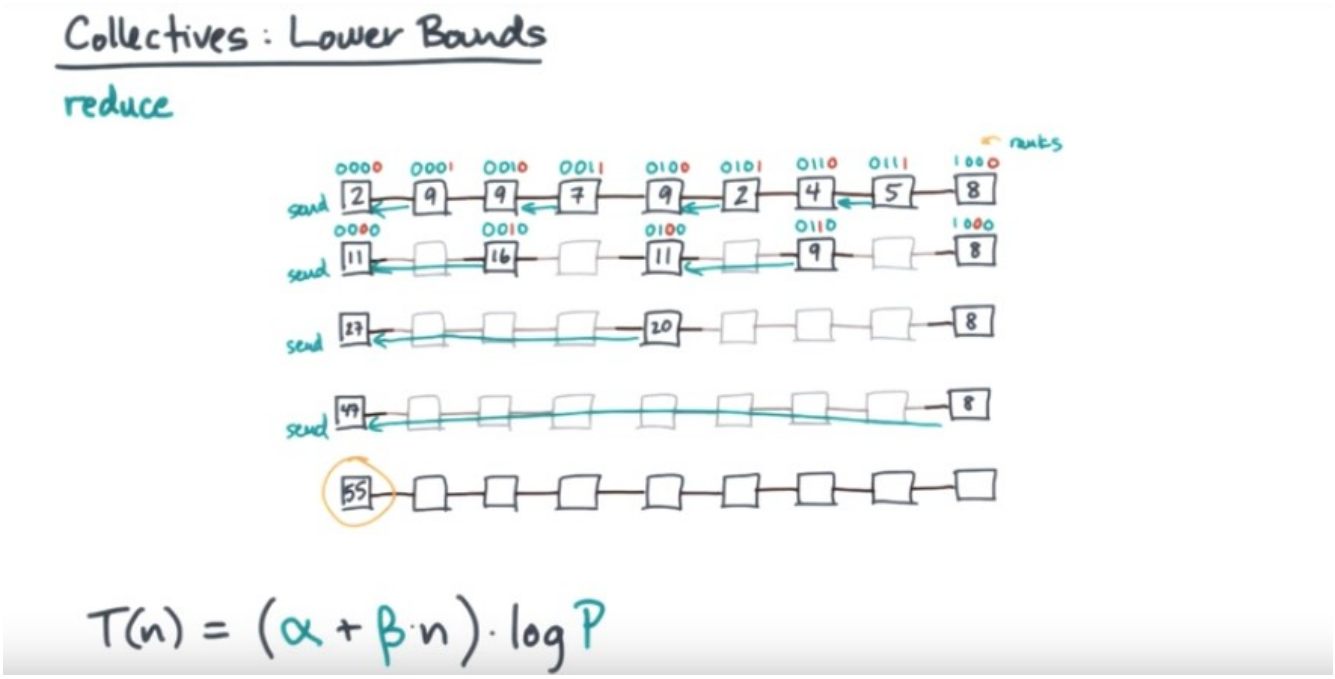


31. Let's say you want to implement an allGather, and let's say you can only do so using these four primitives or reshapes. What I want you to do is give me a pseudo code algorithm for an all gather. Type your answer here.

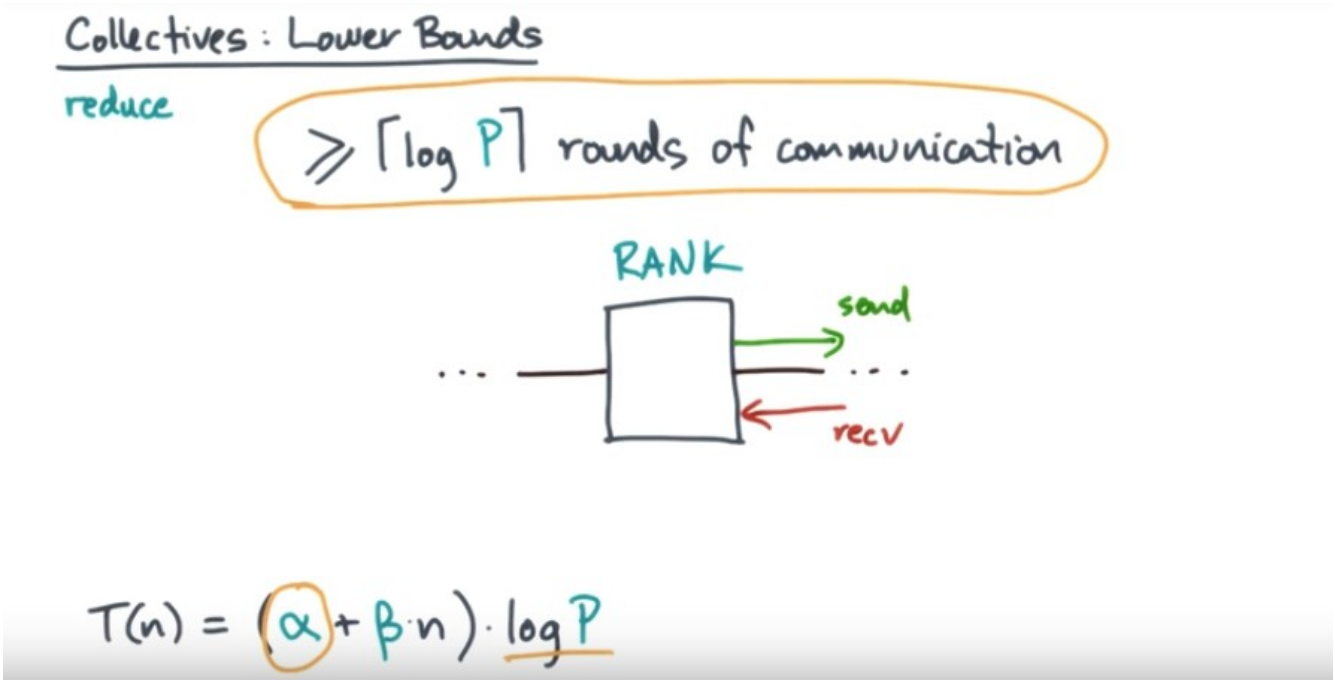
32. Here's one solution. It does a gather followed by a broadcast. And it uses reshape to make sure that the inputs can form to the right format. So, output is 2D as we can see here, but broadcast requires a 1D input. So, you can see why I wanted you to have some reshapes.



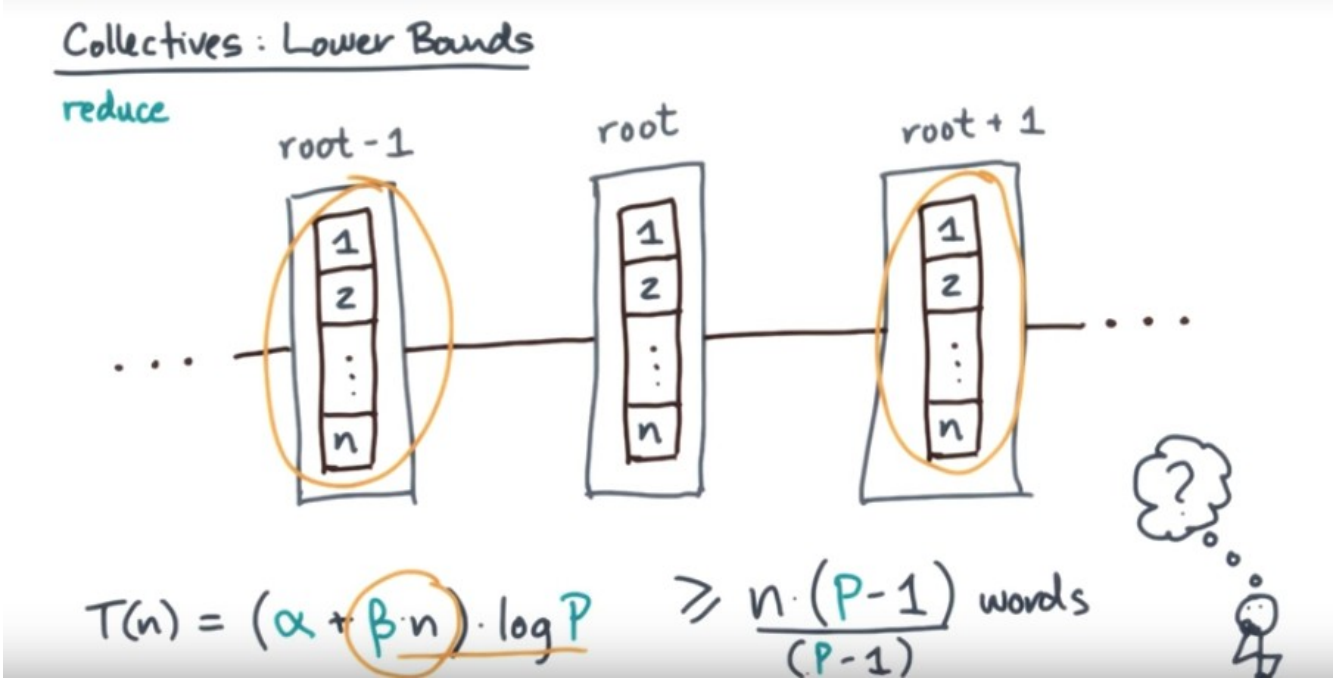
Here's a picture of what the algorithm does. It first does a gather. It follows that with a broadcast. Now a good follow up question is, is this a good way to implement an allGather? Hm, so many things to ponder, and so little time.



33. Before we chat about ways to implement collectives, I want you to think about what our cost goals should be. For example, let's take reduce. We described a tree-based scheme. Is this good or bad? You showed that the cost was essentially this. Basically, the time to send a message of size n times log P. So put another way, you're sending log P messages and you're sending a copy of the array log P times. So is that good or bad? Let's first think about the fact that we're paying for log P messages. That's alpha times log P. Can we do better than that? At least on a linear network the answer is probably not.

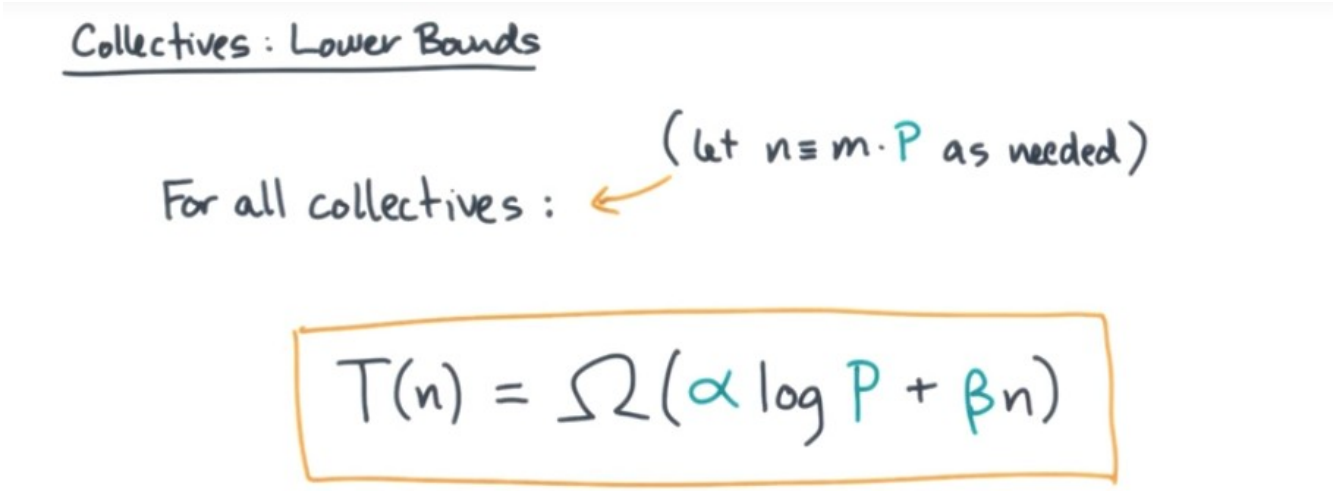


Let's see why. Remember that any process can only perform one send and one simultaneous receive during any round of communication. So this process can send and receive simultaneously. If this process wanted to perform a second send, it's not allowed to do that. The second send has to wait until the next round. So in the very best case, you can pair sends and receives, no matter how you choose to arrange the computation. And if you're pairing you'll need at least log P rounds of communication. Since the alpha terms essentially measures the number of rounds, then this says that log P rounds is good.



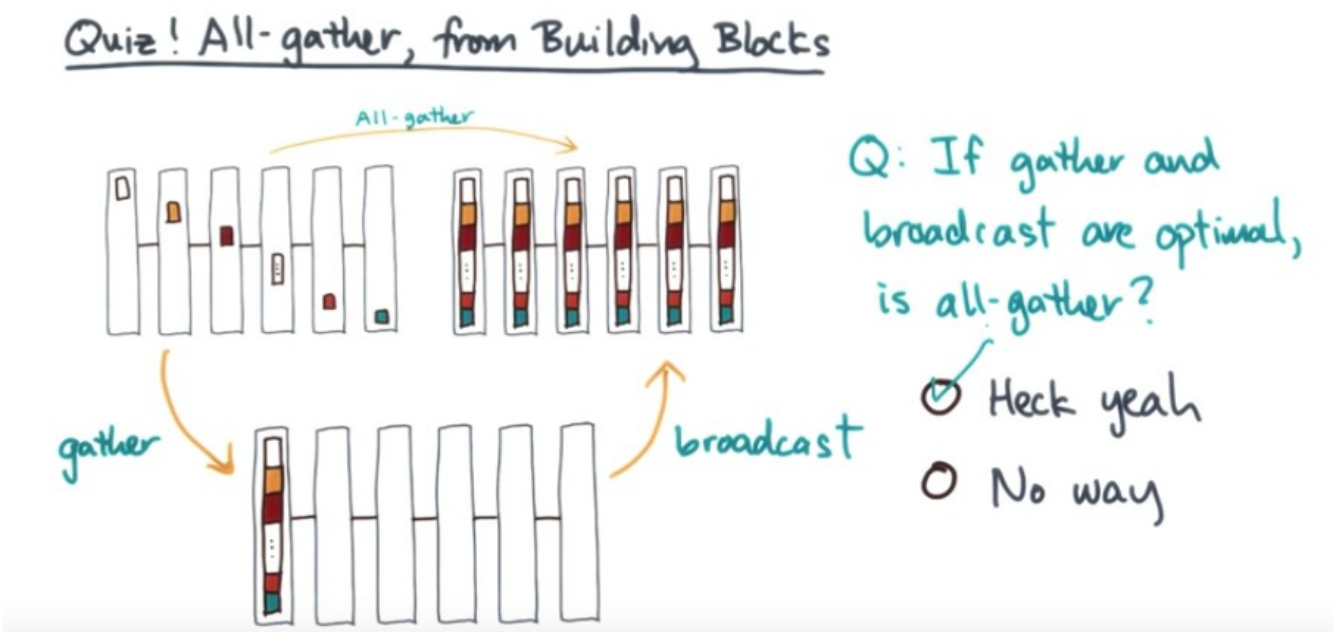
So what about the beta term? Is beta times n times log P good or bad? Let's start with an intuitive and somewhat trivial lower bound. Each process has n words of data. So, at some point in time, every process except the root is going to have to send its n words somewhere. So, in the very best case, you'd need to send a total volume of n (P-1) words. Since there are (P-1) of these processes. If all (P-1) processes could send their data simultaneously, wouldn't that be great? The time we would pay, would be proportional to, the total volume divided by (P-1). So we need to pay to send this many words.

And the speed of communication is one over beta. Therefore a lower bound on time with respect to beta would be just n times beta. Looks like our tree-based scheme might be sending too much data by a factor of log P.



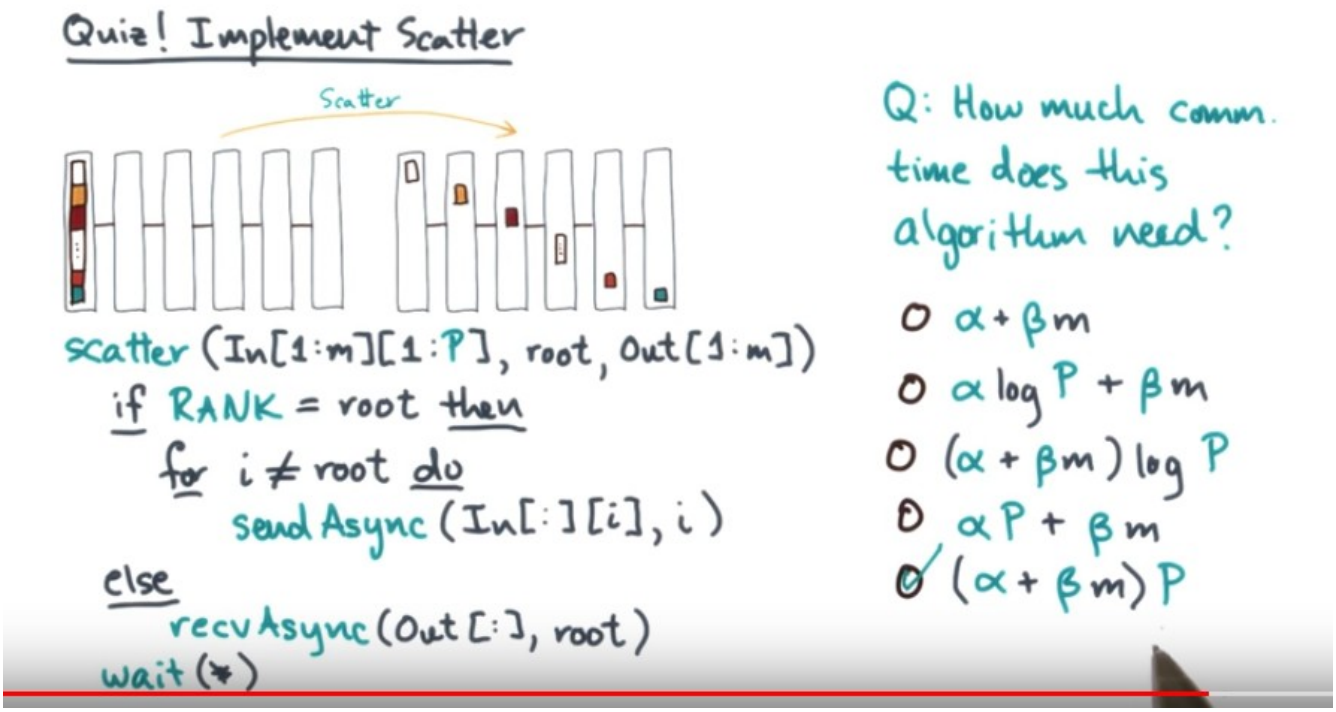
以上的 T(n)是所有 collective 的 implementation 之目標. 記住它, 後面會以它為目標.

Now you can apply a similar line of reasoning essentially to all the other collectives. So for all the collectives we've discussed, the lower bound on communication is this. For collectives like scatter and gather remember that we use the symbol m, m denoted the local problem size. So in this lower bound, this size n is really the combined size.



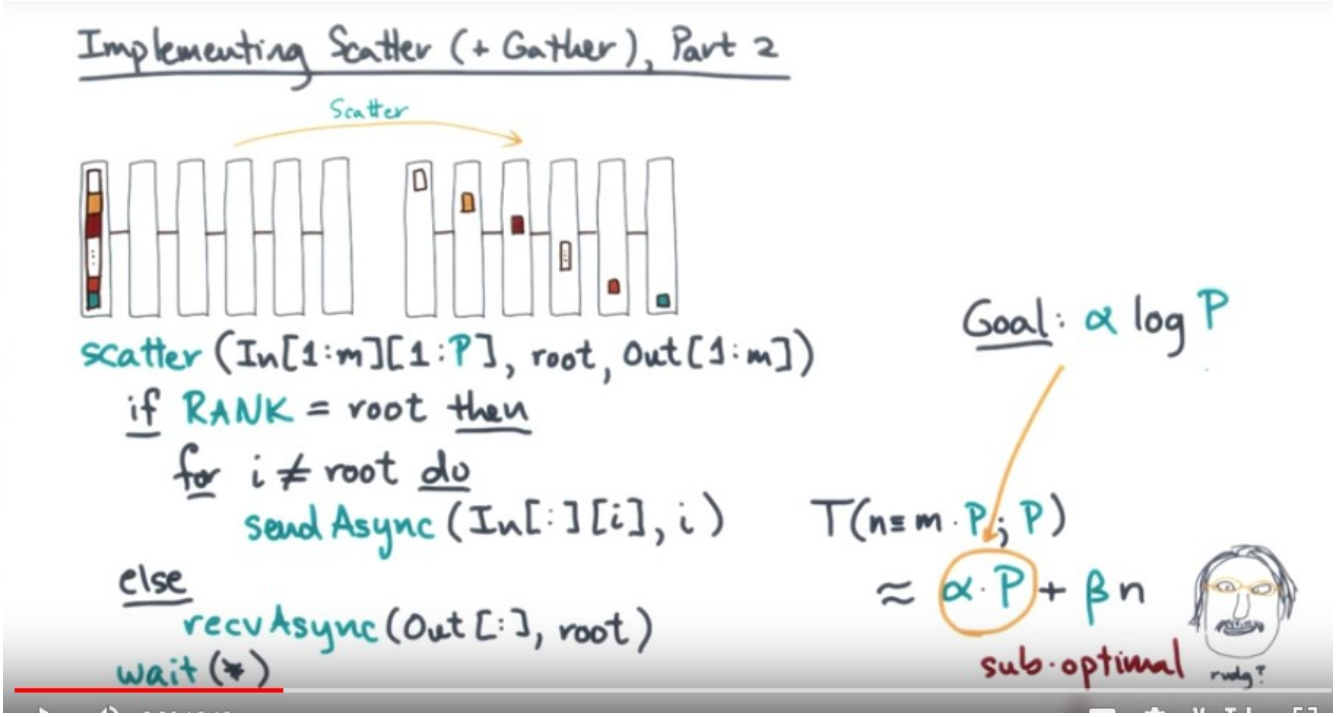
34. If we implement all-gather using the building block approach of a gather step followed by a broadcast step, is that good or bad? Well let's suppose that both gather and broadcast somehow attain the lower bounds. Will the combined primitive also attain the lower bounds? I want you to tell me. Yes or no.

35. If we're being optimal in an asymptotic sense, the answer is yes. That is, a constant number of optimal operations is still optimal. Certainly simplifies our algorithm design and software engineering, if we can use some collectives to build the other collectives. Nice.

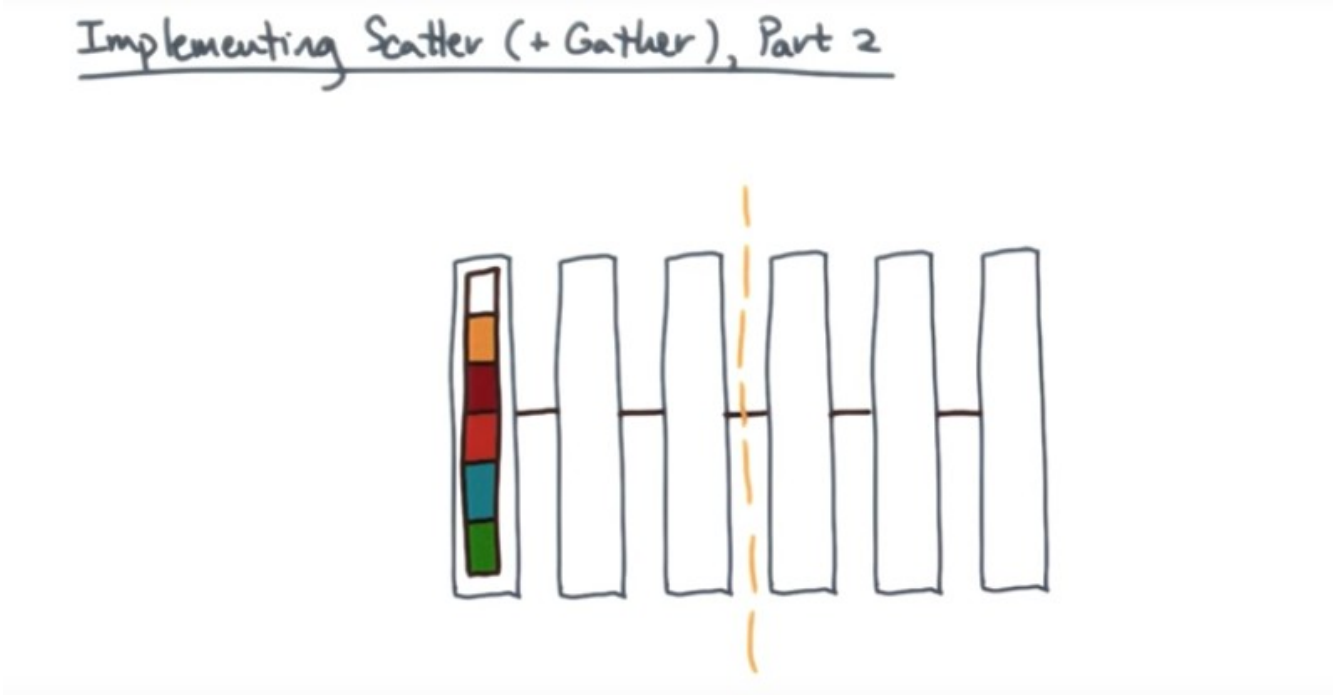


36. Remember how Scatter works. A root has all the data, and it distributes that data to all the other processes. Suppose I give you the following scheme. The roots sends each piece of data, and all other p - 1 processes receive. My question is, how much time does this take? Here are a bunch of options. I want you to choose the best one. That is, even if an option doesn't seem exact, just choose the asymptotically closest.

37. Remember that a process can only do, at most, one simultaneous send and receive. In this case, the root is doing all of the sends. So the sends have to proceed one send at a time. That means you'll have to pay for at least P - 1 sends. So then the best answer is this one, which is linear and P.

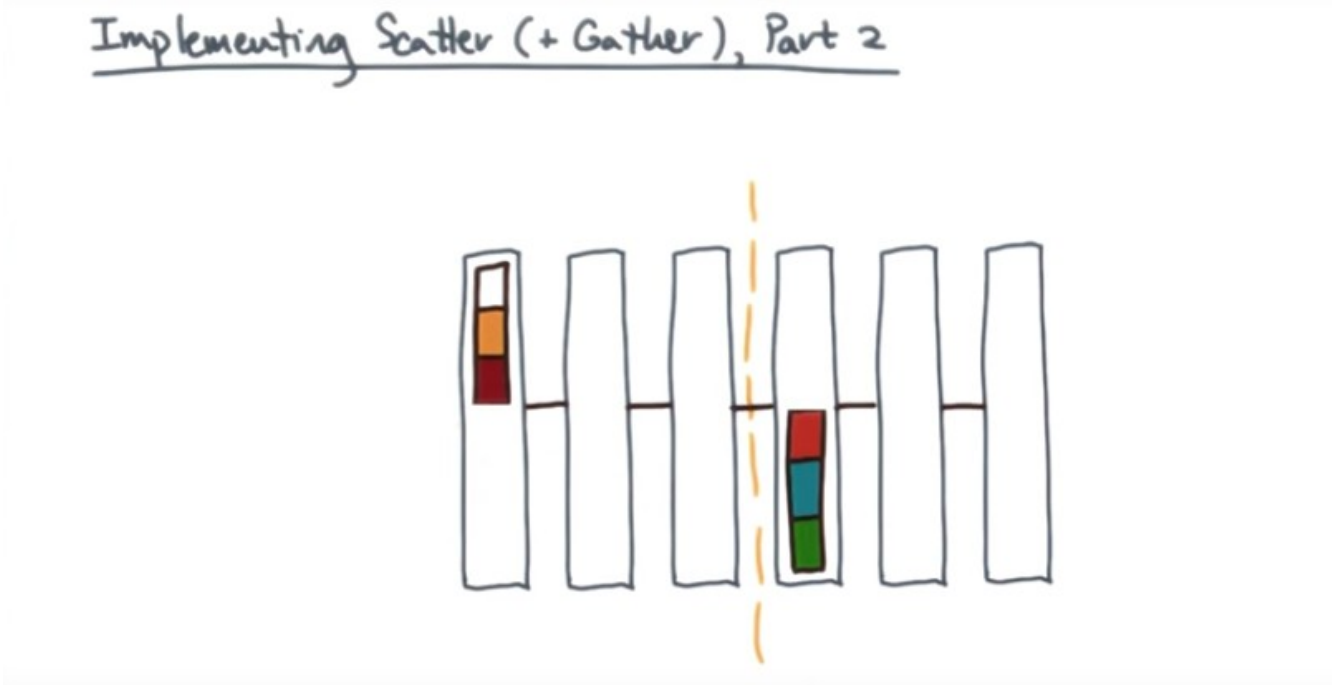


38. I'm about to walk you through an efficient way to implement scatter as well as other collectives. I learned these techniques from a terrific colleague of mine, Robert Van Deguine. I can't do his presentation justice, but I'll do my best. So, here was my first implementation of scatter. It's fairly naive because the root does all the sending. That made the communication time linear in P. And that's sub-optimal. Remember that our goal was to find an alpha log p algorithm.

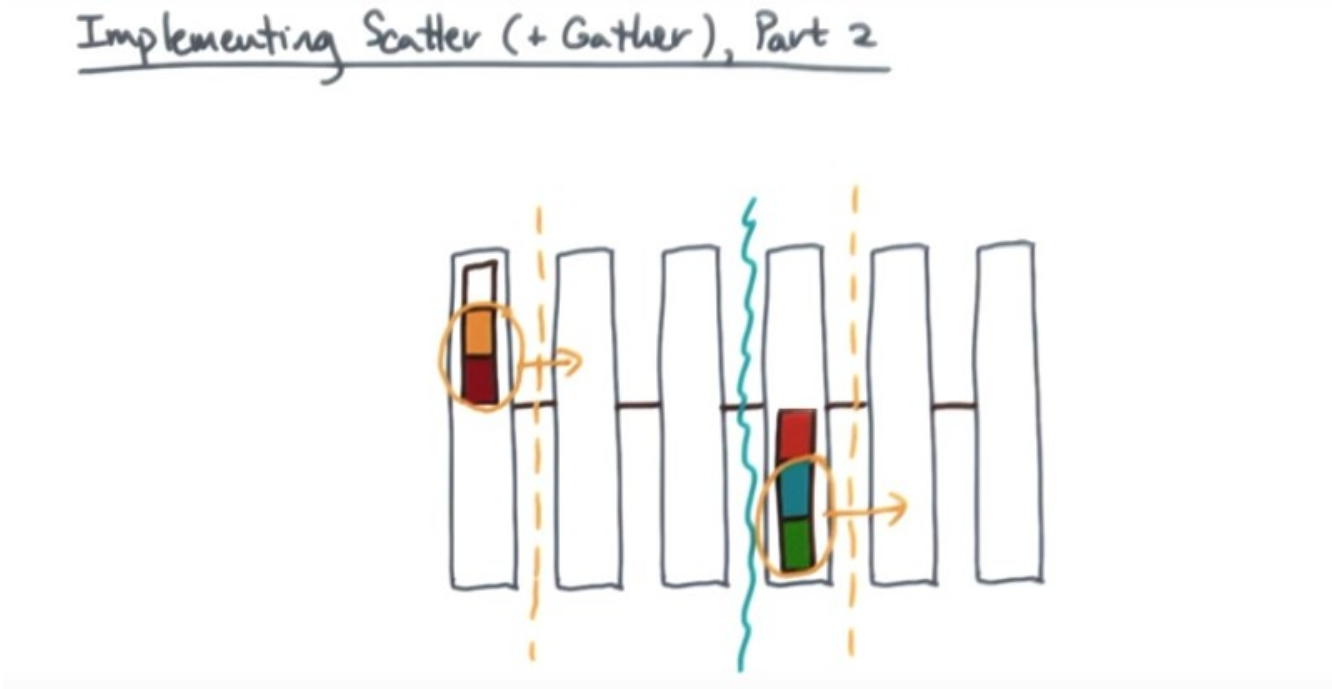


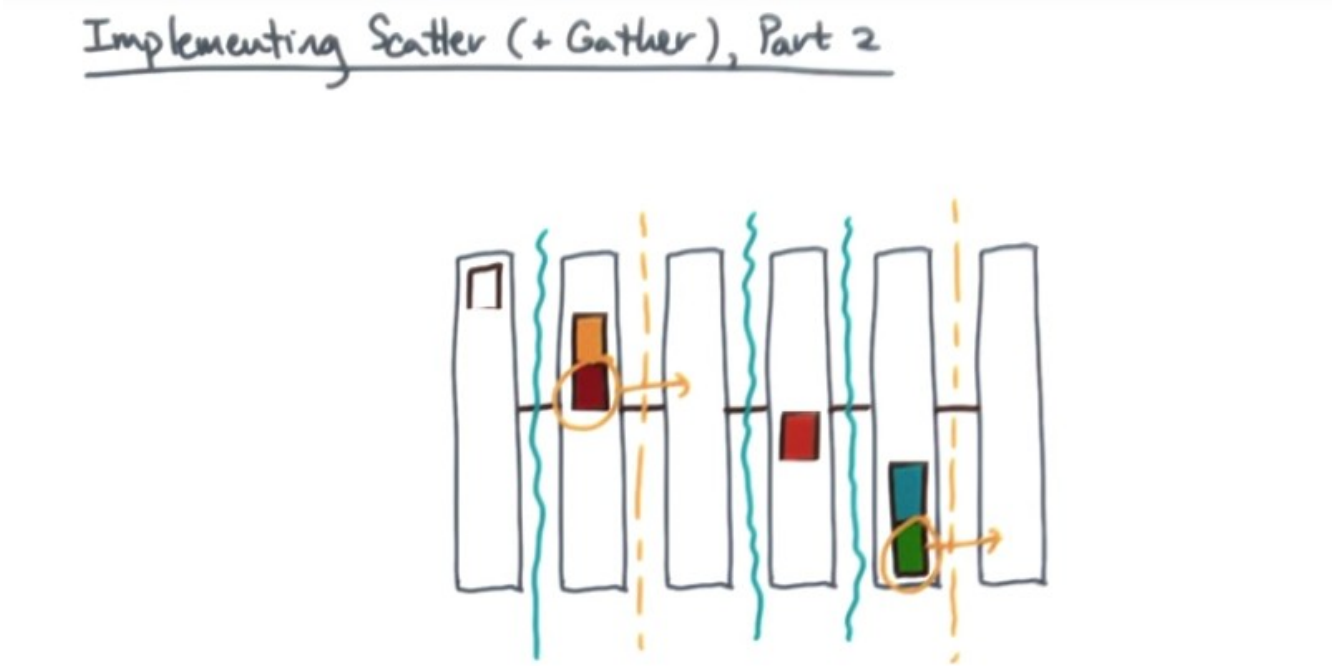
Here's a different approach. Suppose this is the root. Let's start by logically dividing the network in

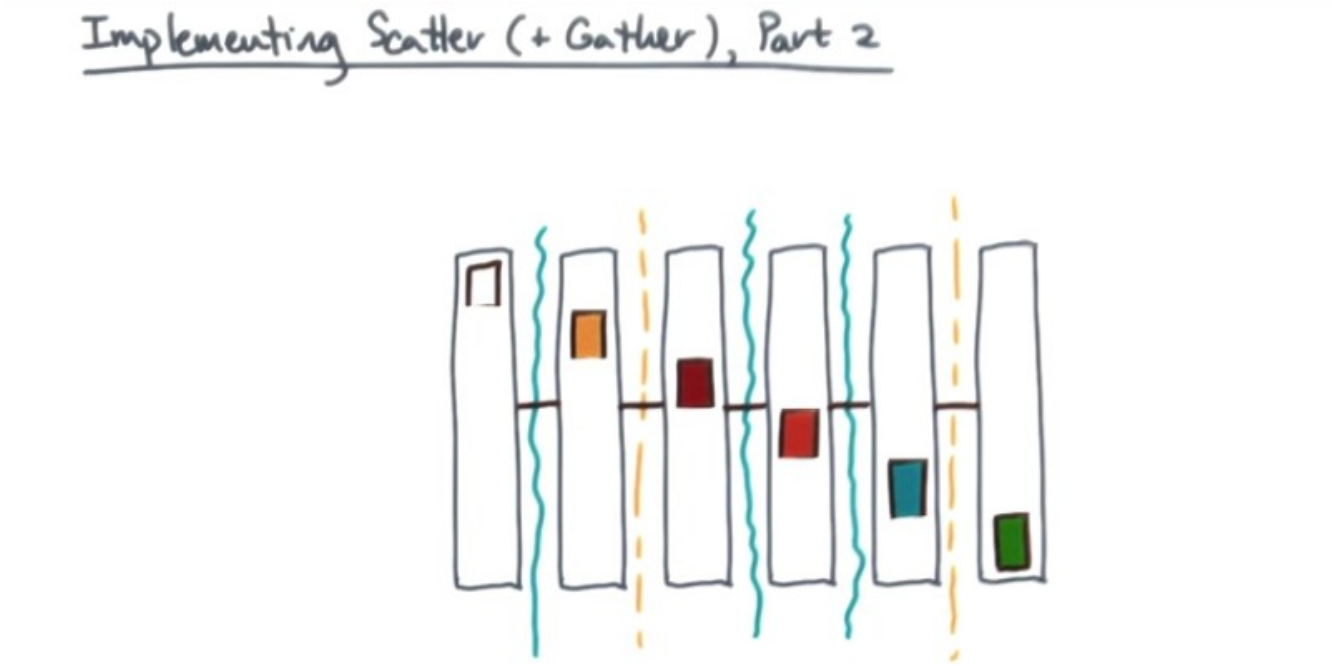
two pieces.



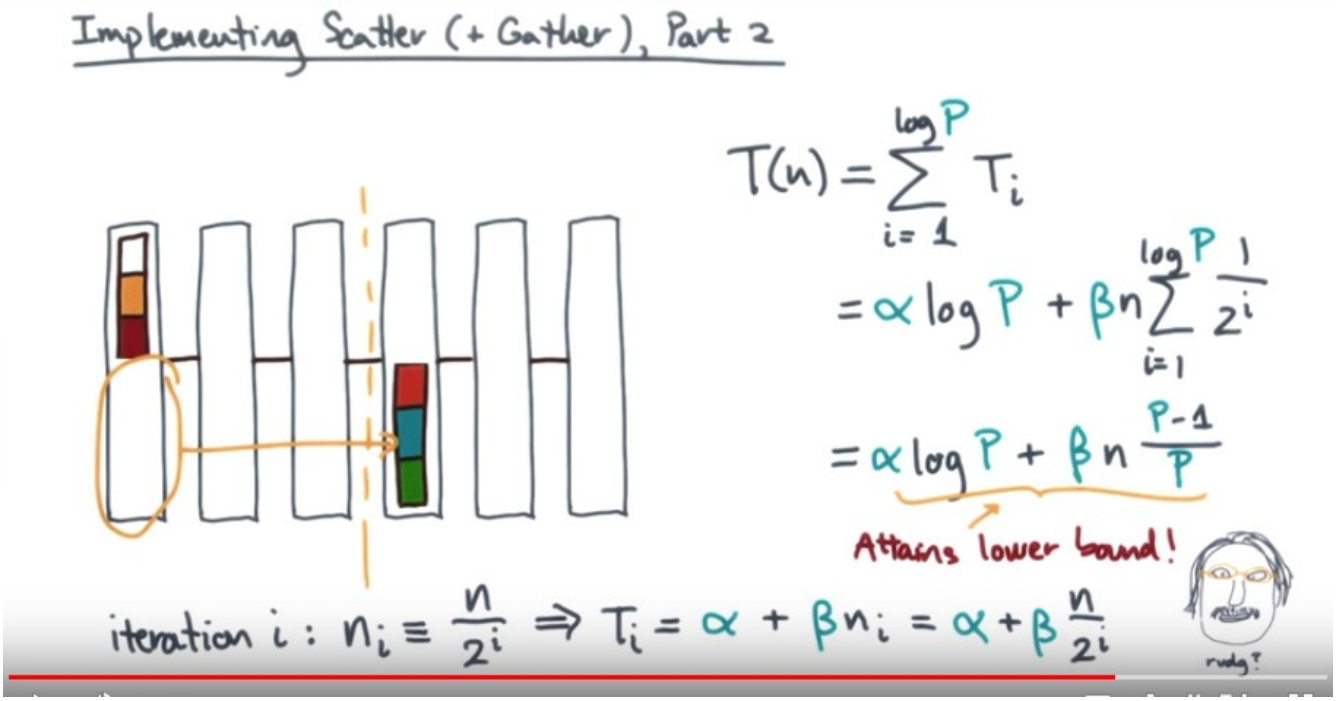
Then let's send the lower half of the data, and we'll just pick some process over here to the destination, let's say this one. So we'll pack up this half, and then we'll send it. Surprise, surprise, you are dividing and conquering. Did someone say conquering?



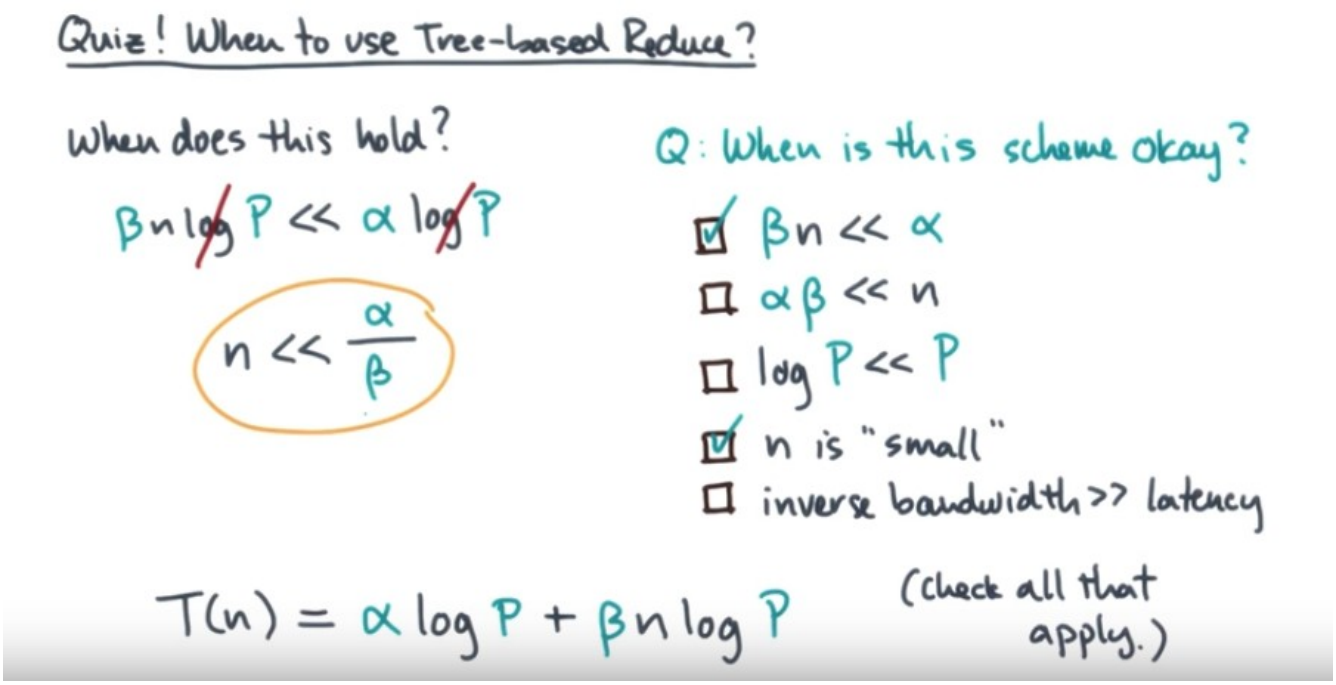




I think you can imagine that you would continue this process. Voila.

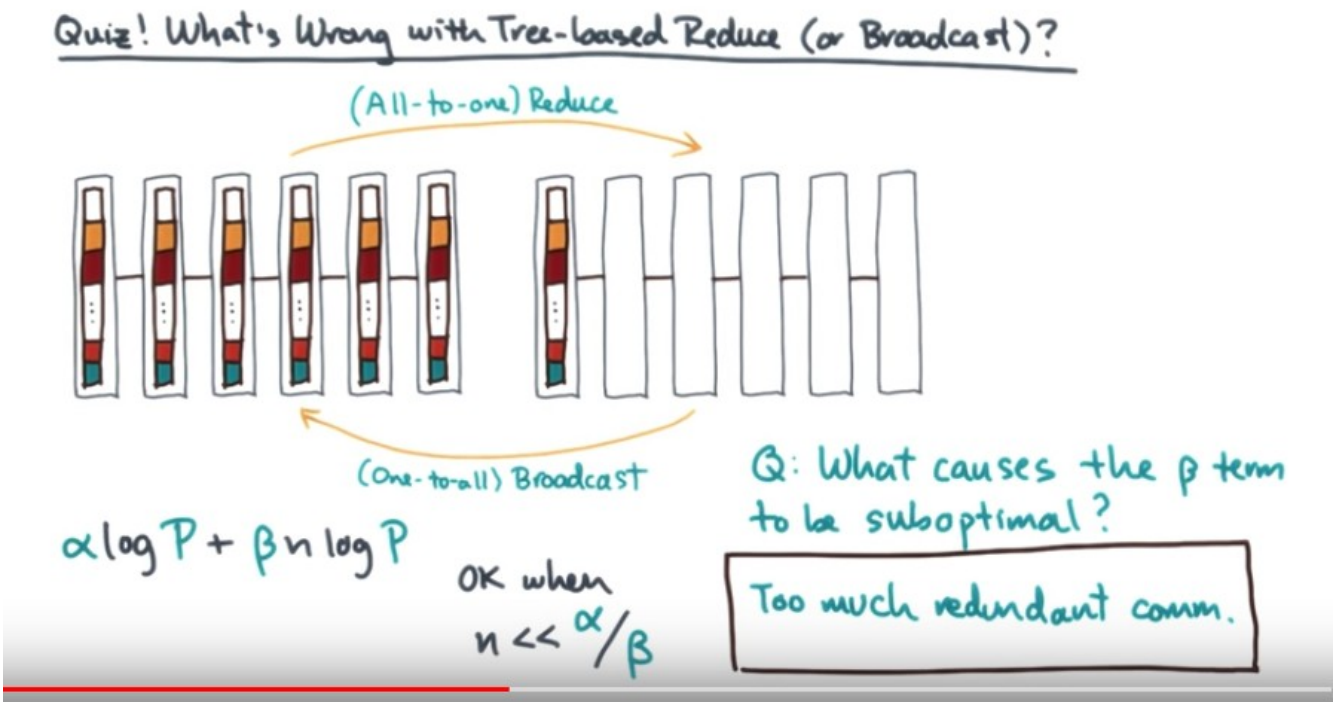


So what is the cost of this scheme? Pick any round of communication, little i. The rounds go from 1 to log p assuming p is a power of 2. Now in round i, this scheme sends a message of size n over 2 to the i. So the time for round i is alpha plus beta times n sub i. So we just need to substitute and then sum over all the rounds. Plug some stuff in. Simplify. So if you work it out, you should get something that looks like this. Notice how awesome you are. You've attained the lower bound, both with respect to latency and with respect to bandwidth. Nice job, you. In fact, at this point, I can let you teach the class while I retire to a beach community, sipping pina coladas and rolling naked in my wads of OMSCS cash. Now, since gather is a dual if scatter, running the same technique backwards, at least conceptually, gives you another good algorithm.



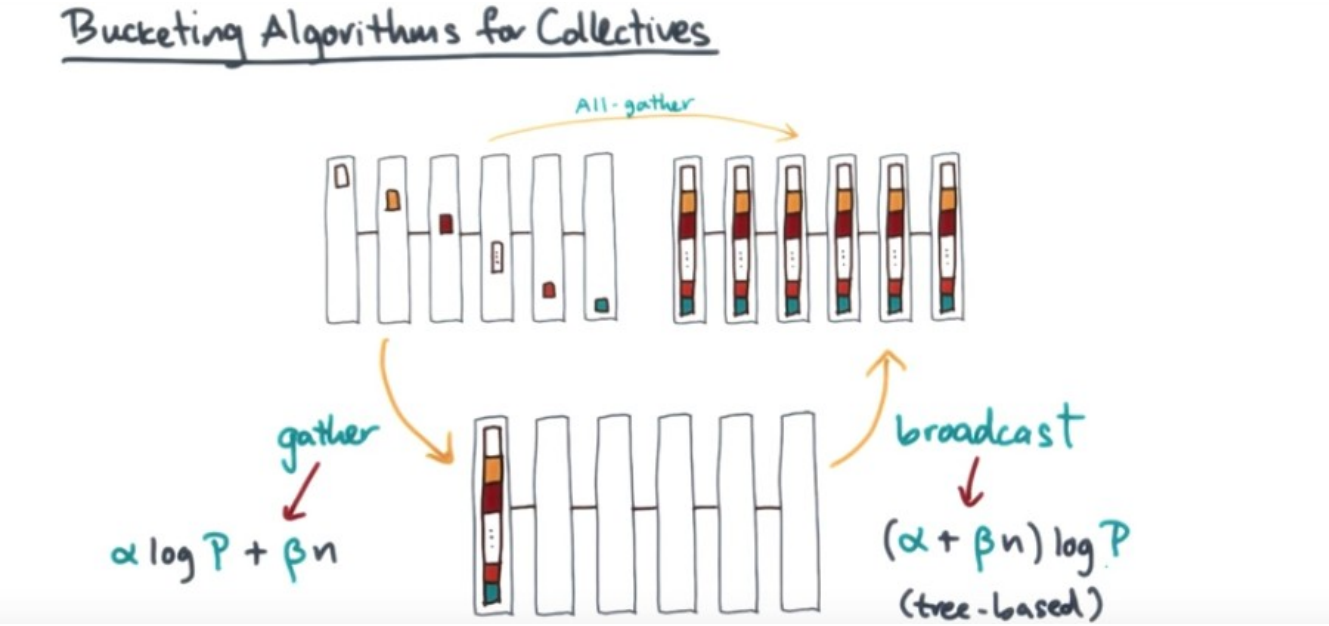
39. Recall the tree based reduction where its dual or broadcast. Now, even though, it doesn't attain the lower bound on end, it doesn't mean it's no good. So, even though, this term depends on log p, it doesn't mean that it's a terrible algorithm. The question is, under what condition, is it still a reasonable algorithm? So I've listed a bunch of choices here, and I want you to check all that apply.

40. Here is what I would say. Let me explain myself. One way to think about the question is to ask an analytical question. Mainly, when does the alpha term dominate the beta term? When that happens, then we might not care that the beta term depends on log p. Algebraically, this term dominating this term is the same as saying that it's much, much greater. You'll notice the log p's cancel. That leaves us with something like this, which basically says n is much less than the ratio of alpha to beta. So let's compare that to our options. The first option is algebraically equivalent. The fourth option, even though it's qualitative, kind of captures what this says. So this says that we want n to be small compared to something. So to me, this is kind of an acceptable intuitive answer. What about the other options? Option two has the relationship algebraically wrong. Option three is just kind of irrelevant, there's no dependence on p here. The last option is also a bit of a red herring. I don't know what that's supposed to mean. In fact, even if you assume n is constant, it doesn't satisfy this relationship. You also remember, it's far from reality. Remember, I said at some point you expect, in real networks, for alpha to be much greater than beta.

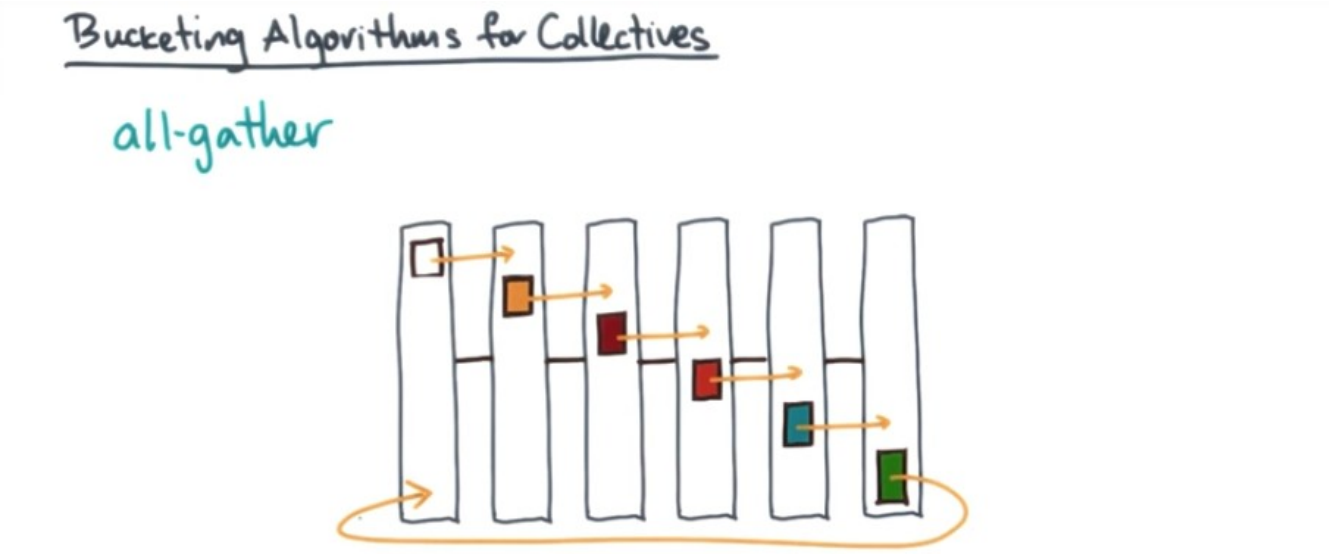


41. The algorithm you've analyzed for reductions and broadcasts is a tree-based scheme. You showed that it was sub-optimal in the beta term. What that really means is that it's a perfectly fine algorithm, as long as the messages are small (即 n 小). But what if you care about the case when n is very large? After all, if beta is much smaller than alpha, then you can utilize a network bandwidth better by sending larger messages. Let me turn this question around on you. Why is the beta term so large? That is, can you explain to me intuitively in English, what is the problem with the algorithm(即這個 tree-based algorithm) that leads to this extra factor of log P communication? Type your explanation here. Note that the question is somewhat open ended so I'm not necessarily looking for a fixed answer. Well actually, I am looking for a fixed answer, but I want to see what you come up with.

42. Here's one answer I'm hoping to see. The tree-based scheme talks too much. In particular, at every round it sends the full message over and over again. Somehow it seems redundant. Let's see if we can fix this excess communication.



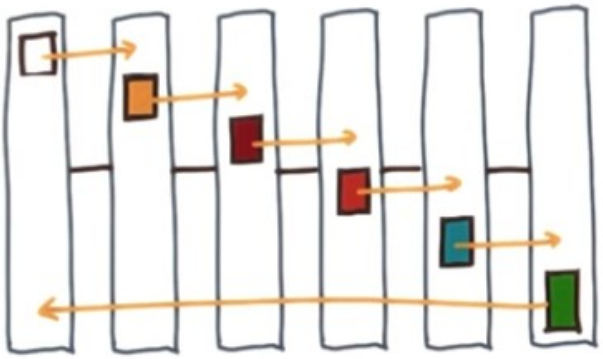
43. Here's a neat algorithmic trick. You can use it to improve the bandwidth term on some of the collectives. It's called bucketing. Let me illustrate the idea using all gather. Remember that an all gather starts with all of the data distributed across all the nodes and ends with the data replicated on all the nodes. One scheme might be to first do a gather followed by a broadcast. So the gather part is fine. It attains the lower bound. What about broadcast? If you only have the tree based scheme, then you'll be off on the beta term by a factor of log P.



I think you need a fresh perspective on all-gather. Let's go back to the initial state. Intuitively, the bandwidth or beta term is fundamentally about using as many of the links as possible. One way to do that is to have every process perform a send to a neighbor in say a ring like fashion. So, for example, we'll have zero send to one, while one sends to two, while two sends to three and so on. Now we also one need P - 1 to send back to zero. You might be thinking to yourself, wait a minute, there's no link between P minus one and zero. But in fact there is. Remember that the links are bidirectional so in fact you can send from P - 1 to 0 by just going in the reverse direction.

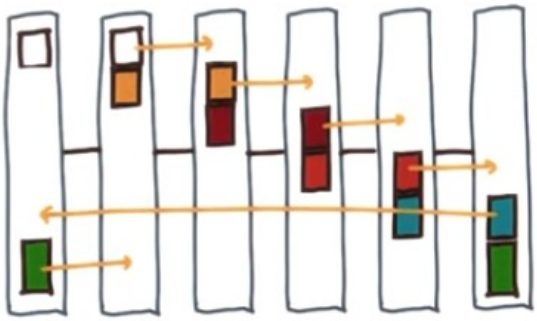
Bucketing Algorithms for Collectives

all-gather



Bucketing Algorithms for Collectives

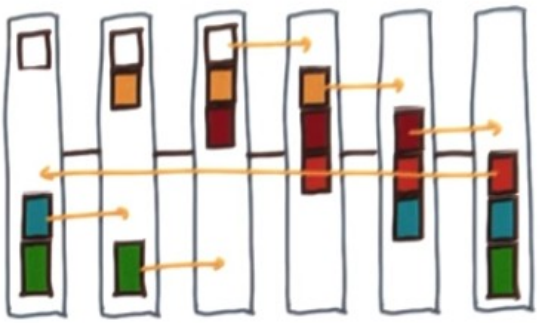
all-gather



So if we try this scheme and we execute one round of communication, let's see what we get.

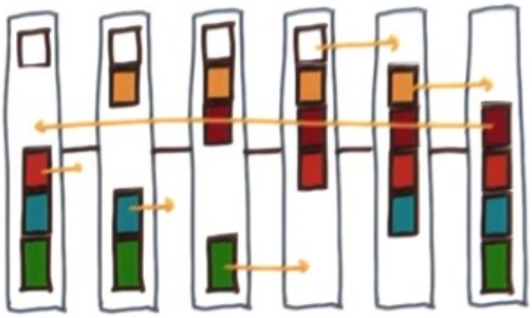
Bucketing Algorithms for Collectives

all-gather



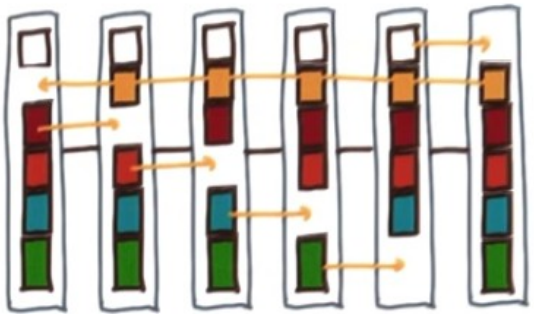
Bucketing Algorithms for Collectives

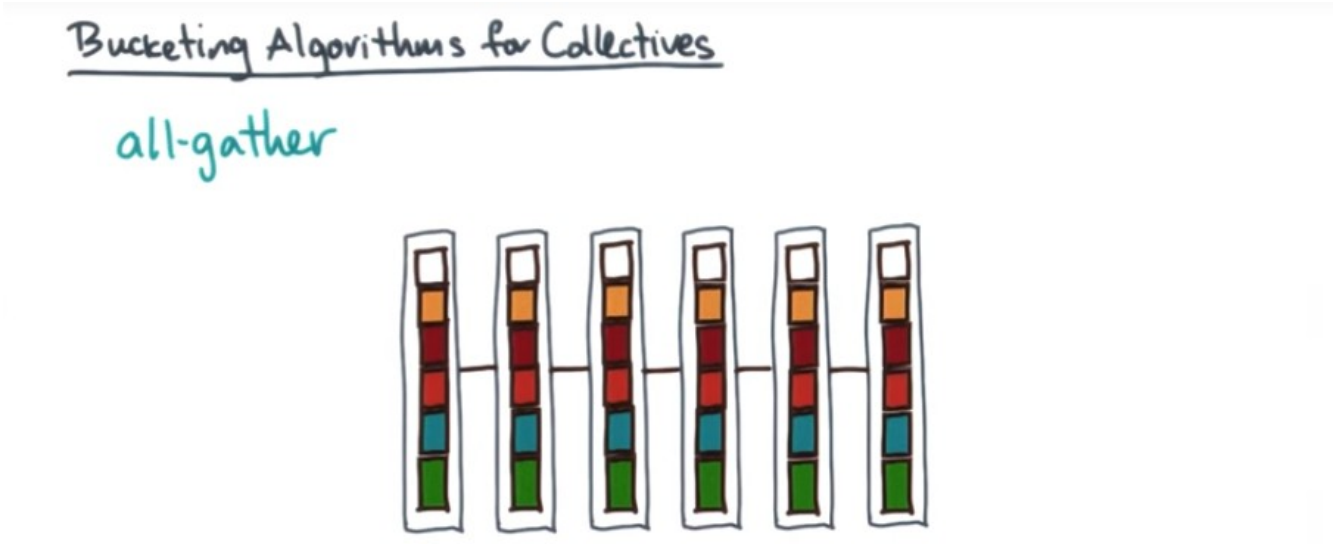
all-gather



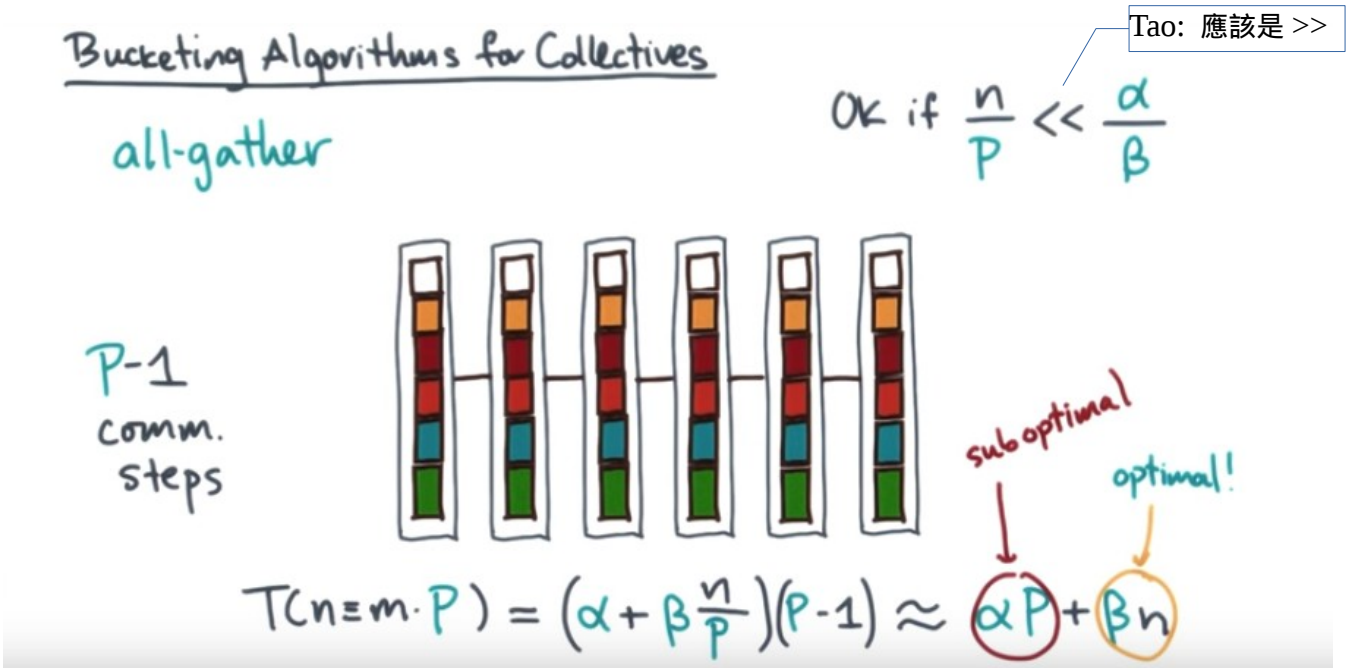
Bucketing Algorithms for Collectives

all-gather

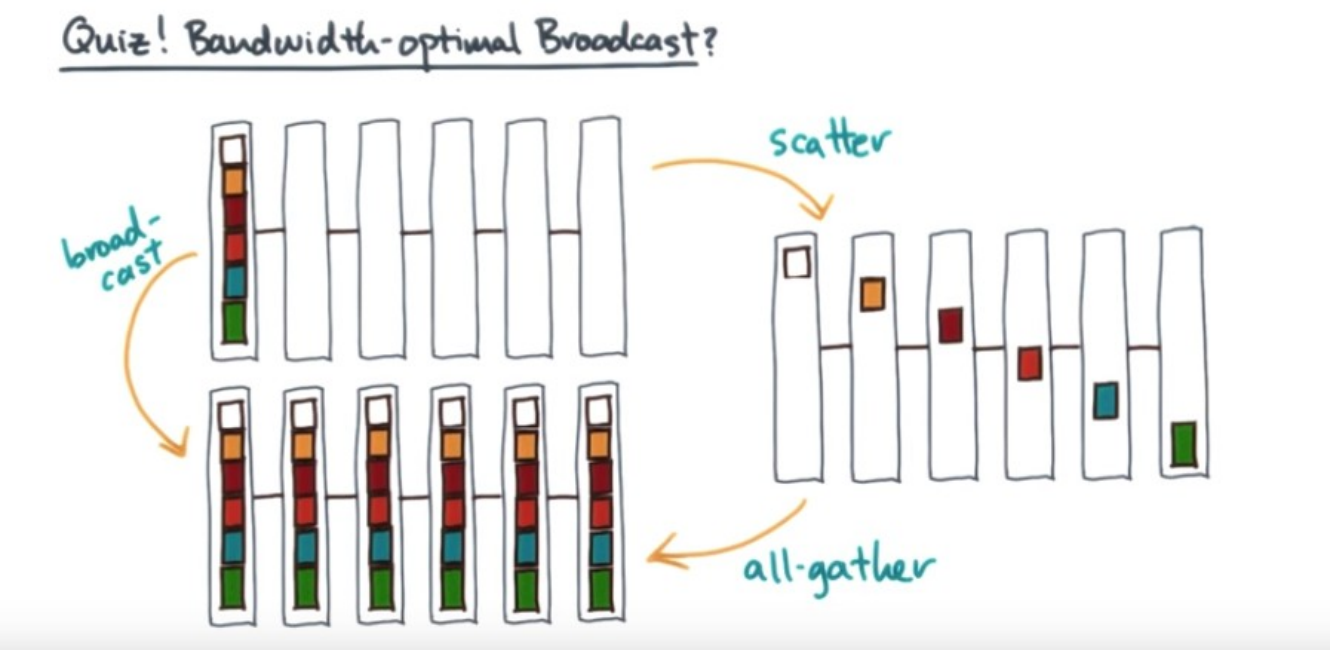




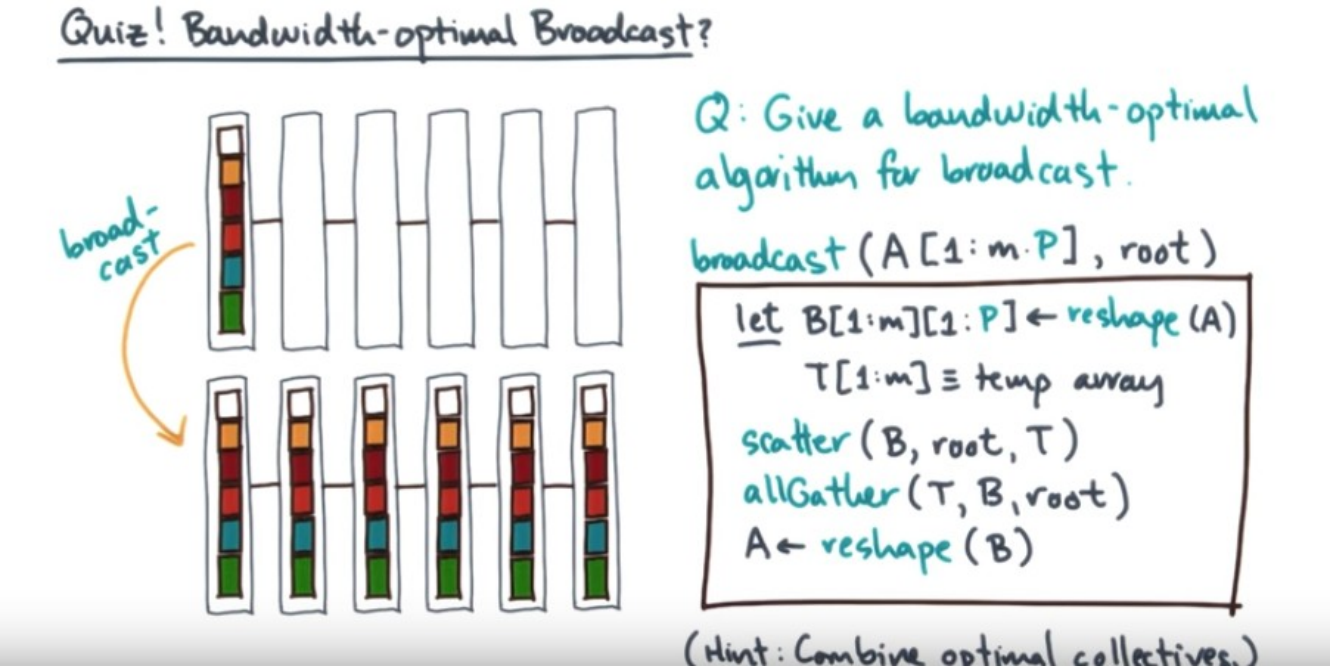
Let's keep going, in particular, let's have every process take the data it just received and pass it along and continue on and on.



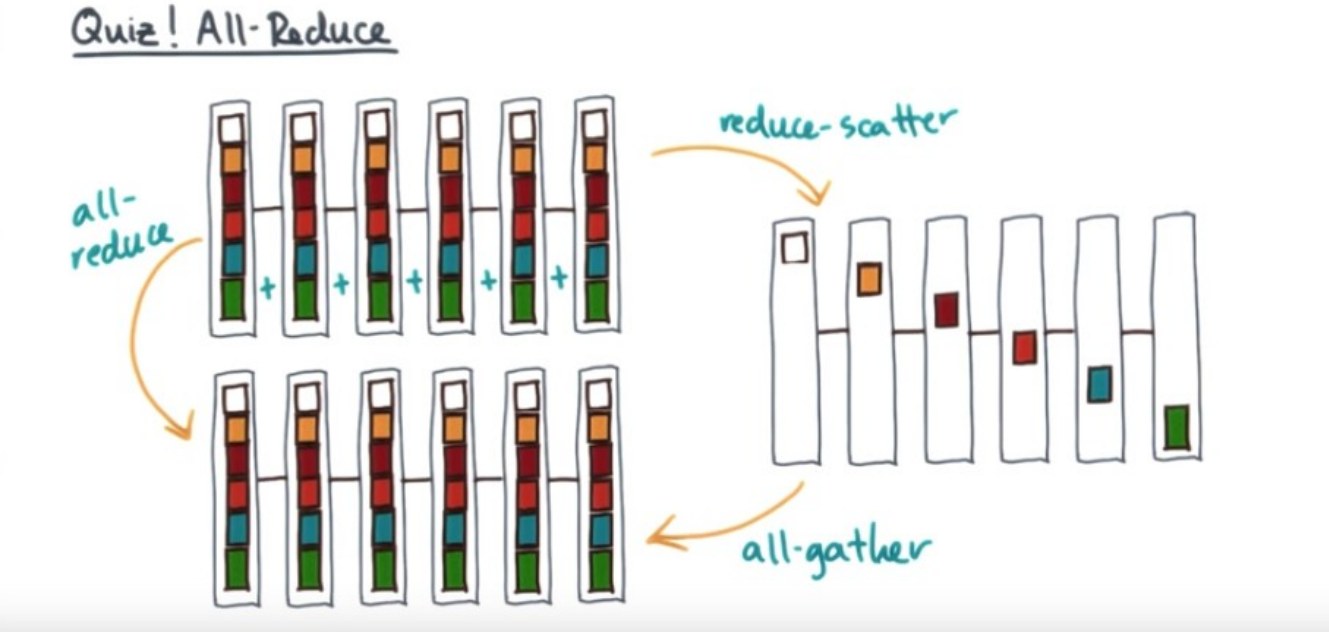
All told, that will be P-1 communication steps. So what's the total cost? Well, each process and each step sends, M equals N divided by P words. But it does so in parallel. So, writing down the cost, each message is alpha, plus beta, times N over P, and there are P minus 1 rounds, and that gives you something that's roughly alpha times P, plus beta. And with respect to the beta term, this is optimal. Woot. But you can also see that there's a catch. It's sub optimal with respect to the alpha term. But that's okay. If the message is large so that this term (beta * n, from video) dominates this term (alpha * P, from video), then the following relationship has to hold. The per process message size has to be much less than the ratio of an alpha to beta. So, provided that's true, we'll say that this algorithm is good. So, now you have a bandwidth optimal all-gather. Since reduce-scatter is the dual, you also have a bandwidth optimal reduce-scatter.



44. Given what you now know, I want you to give me a new broadcast algorithm that is asymptotically optimal in bandwidth. That is, I want you to define the implementation of broadcast. As a hint, think about combining bandwidth optimal collectives.



45. Following the hint, one way to do it is to combine scatter and all-gather. So visually, that would look like this. First, scatter the input, then do an all-gather. The key assumption is that all-gather uses bucketing. Now there are many possible ways to write the pseudo code, which our auto grader probably doesn't even really fully check. But hopefully you came up with something that looks like this. Remember that these reshapes are not actually physical copies, they're just different perspectives on their inputs. This code uses a temporary array to hold the result of the scatter, and then sends that temporary array.



46. Here's a new collective, called an All-Reduce. It's the same as a reduction, except that instead of having the final result on one process, it has a copy on all the processes. So in other words we add all the distributed vectors, and then we make sure that the result is available everywhere. So suppose we want a bandwidth optimal implementation. Which pair of the following collectives do you need to combine? I want you to choose two of them.

47. If it were up to me, I would use reduce-scatter followed by all-gather. Let's see why that works. Reduce-scatter will perform the reduction, leaving a piece of the result everywhere. All-gather will collect the results everywhere. All-gathers and reduce-scatters are duals of one another. And since we have a bandwidth optimal version of all-gather we have a bandwidth optimal version of reduce-scatter. Their combination is therefore also bandwidth optimal.

48. The alpha beta model is messy, especially if you compare it to the dynamic multi-threading model, which you might use on a shared memory multi course setting. But, the basic argument for message passing is that you need to get your hands dirty if you want to be efficient at large scales. And that's the big idea of this lesson. How to think about efficiency, specifically in terms of computation versus communication. Now, the message passing model also has at least one major weakness. It forces you to think about a lot of thinks like who and how many processes there are, when and how those processes communicate, as well as how are the processes are connected. So an interesting and largely open research question is, are there efficient algorithms that are network oblivious(不在意的)? → 此句話之意思: That is, is there a framework for writing and analyzing algorithms that will be efficient independent of the network? I wonder. [SOUND]