COLLECTIVE COMMUNICATIONS - CS624

1. POINT-POINT MESSAGES ARE SUFFICIENT

CSP (see C.A.R. Hoare - Communicating Sequential Processes) communicates entirely through “rendezvous” between processes - that is, fully synchronous point to point communications. CSP is a fully general concurrent language, it is in fact used in software engineering to specify concurrent systems. The fact that we can program anything in CSP tells us we don’t need any communications mechanism beyond message passing.

It is relatively simple to prove this claim. Consider a total state $S_{\Gamma,J} = \{\sigma_j^p, M_j^p\}$ of a concurrent execution. The set $\{M_j^p\}$ is the collection of memory contents at all processes in $\Gamma$. Now assume that $\{\sigma_j^p\}$, the basic blocks being executed, contains message code that simply copies $M_j^p$ from each process to some specific process, for example $p = 0$. Then, process 0 can perform any computable operation on $\{M_j^p\}$, since it has all the information, and transmit results back to all other processes via messages. Therefore anything computable by the full set of processes $\Gamma$ can be done using only message passing.

The above is a proof outline, but it can be formalized if needed. Also, note that we are not saying anything about efficiency - there may be methods that are more efficient than message passing, particularly if there is hardware support for them.

2. COLLECTIVE COMMUNICATION AND COMPLEXITY

Even though point to point messages are sufficient, how we arrange our communications impacts how our programs run.

When we analyze sequential programs, it is important to estimate the amount of work required by an algorithm (time complexity, big O), and the space required (space complexity). There is frequently a trade-off between time and space complexity - i.e. we can save time by using more space. In fact, concurrent programming uses this tradeoff - we speed up our computations by using space and resources of many computers rather than just one. Similar considerations apply to communications, in particular when there is data to be transferred to and from multiple processes (as in the proof outline in section 1).

We consider two specific modes of collective communication (both used in the proof):

1. Broadcast: one process sends information to all other processes.
2. Reduction: one process needs to receive information from all other process - this is frequently combined with broadcast in what MPI calls an allreduce; information is gathered at one process, computation is performed to produce a result at that one process, the result is then broadcast.

2.1. Broadcast. Designate some process $p$ as root process, send data from $p$ to all other processes. MPI_Bcast has $p$, variable that holds data, communicator (e.g. MPI_COMM_WORLD) as parameters. All processes in the communicator
execute the same MPI_Bcast statement, the process identified as root sends the
data, all others receive it. How is this done? (By the way, in Planguges broadcast
is also a single statement - it looks like \( x = y^@p \), where \( p \) is the root process - this
means copy the value of \( y \) at \( p \) and put it into \( x \) everywhere).

Hardware support is an issue - if we have a data bus (for example an ethernet
segment where many nodes are wired together without passing through a switch or
router), or a wireless network, the root sends and all others receive. This does not
scale well to large numbers of processes - we used switched networks. Therefore we
need to send multiple messages.

Let root process be numbered 0, we have \( N \) processes numbered 0 to \( N - 1 \)ty.
Space complexity is the number of messages required. Time complexity is the
number of steps required, where there may be multiple messages sent concurrently
in one step. We assume that each process can send or receive one message per
step - changing this assumption to any fixed number can speed things up, but only
changes the constant not the complexity (as in big O complexity estimates).

Simple solution: root process sends in turn to every other process. We need
\( N - 1 \) messages (space complexity); since each message is sent from 0 and it only
sends one message per step, there are \( N - 1 \) steps (time complexity). Space and
time communication complexity are both \( O(N) \).

Ring: ring networks (as in baton passing systems) are simple to build and ana-
lyze, we can set up a logical ring if processes send to \((id + 1)mod(N)\) and receive
from \((id - 1)mod(N)\). Under this scheme, process 0 sends only one message, to
1; all others pass the message on until it reaches process \( N - 1 \). Space and time
complexity are still \( O(N) \) - note that this is also the complexity of baton passing.

Tree: arrange processes in a tree (for example, a binary tree - each process has a
right child \((2*id + 1)\) and a left child \((2*id + 2)\). We still need \( N - 1 \) total messages
(number of edges in the tree) but now we get concurrent messages - it takes 2 steps
for 0 to send first to 1 and then to 2, but 1 and 2 can then send concurrently to 3, 4, 5
and 6 (and in fact 1 can start immediately while 0 is still sending to 2). Therefore
each level of the tree can send to the next level in 2 steps (because each parent node
performs 2 sends, but all nodes in a level can send concurrently). Therefore the
number of steps (time complexity) is \( 2*(height) \); the height of the tree is \( log(N) \)
(base 2 \( \log \) for binary tree), so communication time complexity is \( O(log(N)) \).

We can do more efficient trees; for example in a binary tree the root node does
nothing while lower layers are active - we may want to give nodes higher in the
tree more work to increase concurrency. Calculating a tree from a hypercube (see
notes on networks) does this - nodes on each level of the tree have one less child
than nodes on the next highest level. However, this only changes the constant;
complexity is still \( O(log(N)) \) because that is tree height.

Note that we are not limited to a root node of 0- one easy way of calculating the
tree for any root is to put node numbers into an array, placing the node we want
as root in index 0 - then we calculate a tree with root 0, but use the numbers in
the tree as indices to the actual process numbers.

Trees appear to be the best we can do for broadcast, or for that matter for any
collective communication based on point-point messages.

2.2. Reduction. A reduction is just a broadcast in reverse - the entire argument
in the broadcast section is repeated, just reversing the message direction. In a tree,
for example, we start in the leaves and work up to the root.
A reduction generally involves computing some binary function - that is, there is some function \( y = f(a, b) \) and we want to compute a total \( y \) for all processes. For instance, \( f \) could be the sum of \( a \) and \( b \) or the maximum. \( f \) should not depend on order of its parameters: \( f(a, b) = f(b, a) \) for reduction to make sense, although we can compute reduction for any binary function (Consider, what, if anything, does the difference of all \( x \) values at all processes mean?). (Technically we want \( f \) to be Abelian - don’t worry about this terminology, but you may come across it in more mathematical papers).

Syntax of reductions can be confusing - even though we have a function of 2 parameters, MPI reduce or allreduce only takes one variable; in Planguage we would write \( y = f\{a\} \) for a reduction using function \( y = (a, b) \). The idea is, the second parameter is the same variable at some other process.

In practice, reductions execute code something like:

- if(i am not leaf node)
- loop(each child node) my-value=f(my-value,value-from-child)
- if(i am not root) send(my-value to parent)

That is, each node gets values from its children, combines them with its own value and passes it up the tree. The root gets the final values and computes the final total. In an allreduce, the above code is followed by a broadcast.

3. SEMANTICS, AND WHY WE CARE ABOUT IT

When we argued, in section 1, that \( \{\sigma^p_i\} \) (set of basic blocks being executed) contains point-point message code that copies all of memory to a single process we were playing fast and loose with our own theory, because we have previously said that we are allowing only a single communication statement per basic block, and we know that it takes multiple steps (therefore multiple blocks) to get all the data to a single process. We could argue that, with multiple steps, the memory could change at each step and that therefore the root process does not actually get a true copy of the memory everywhere else.

That is one reason why I said we were doing a proof outline; we really need to have multiple steps that only communicate, without computation (as in BSP). We can argue about whether multiple steps make a difference or not, the proof is obviously cleaner if we can communicate in a single step. (In a proof that allows multiple steps, we also need to say that whatever computation occurs during each step can increase the speed, but not actually calculate anything not otherwise computable - this is what happens in our pseudocode for reduction in section 2.1).

Now, though, we have explained how to implement broadcast and reduction operations with point-point messages, and we have MPI and Planguage instructions that implement both in a single statement - so we can actually write the whole proof algorithm from section 1 in a single block (using an allreduce statement), or at most 2 blocks (a reduction followed by computation at the root in the first block, a broadcast in the second). We show that the computation during reduction only increases speed, but if we insist on only computing at the root, we could write a function that passes an array indexed on process number, and just has each process put its local data into the array so the root gets an array with data from each process indexed by process number and can then compute using all data).
So the question becomes - is it legitimate to treat collective communication statements like allreduce as single communication statements, even though we know that they really represent a complicated distributed pattern of messages?

We have argued previously that the control flow graph gives us a structure that is consistent at all language levels because the compiler preserves the meaning. Now we are forced to acknowledge that this is not always the case - when a reduction - which is a single communication in a single block - gets translated to low level message code, it expands to multiple statements and multiple blocks surrounded by control logic (things like - “if i am an interior node then receive from child nodes and send to parent”). This may happen at compile time or in run time code, but in either case what actually executes is the complex set of statements not the high level simple code.

Semantics comes to our rescue (always assuming we trust the compiler and run-time libraries). The concept here is, it does not really matter what the actual executing code looks like, as long as it is a faithful translation that preserves the meaning (and function) of the high level code. We are justified in performing our analysis on the high level code as long as we can show that it is possible to translate into low level code without changing the meaning (which we have done here for broadcast and reduction).

It is important to really analyze and if possible prove the semantics. If the actual code does not mean what is claimed for it, or if its meaning can change at execution (as is the case for the MPI standard send and receive code, see discussion of message semantics), then the program will behave in unexpected and possibly incorrect ways. In particular, we need to consider the theoretical implications of the code we write so that we can predict and understand how it will behave.