1 Introduction

We continue from last time with explanation of the BG simulation technique. This technique helps to obtain both possibility and impossibility results. The basic idea is to formalize the notion of solving problem $A$ (which has $n_1$ inputs) if we have a way to solve problem $B$ (which has $n_2$ inputs). We will say that $p_1, \ldots, p_{n_1}$ are the simulating processors, and they are simulating $q_1, \ldots, q_{n_2}$ simulated processes or programs.

2 Simulation overview

What is the structure of the transformation from inputs of $A$ to inputs of $B$? If $n_1 < n_2$, we could use the simpler simulation technique from the previous lecture, letting each $p_i$ simulate some number of the $q_j$’s, so that no program is simulated by more than one simulating processor. In this more powerful technique, $n_1$ and $n_2$ can be arbitrary numbers. So we need coordination.

Our plan is to let $n_1$ real processors each simulate $n_2$ programs, so that the $n_1$ sets of simulations are consistent with one another and also consistent with a real execution of the $n_2$ programs. We will assume we are dealing with an Atomic Snapshot model, so there are two basic operations: update and scan. The update operation is a simple step: if $p_i$’s state information about $q_1$ before the update is the same as $p_j$’s state of information about $q_1$, the update will be the same. It is the scan we have to argue about, because different processors will be doing a scan at different times. We want a valid value of scans to show up. So we run a pseudo-agreement algorithm at that point. Then the scan value for all simulating processors will be the same.

Is the scan value also consistent with a particular execution of the problem $B$? Answer: yes, because the winner of the scan step got the scan information from taking execution steps, and we look at those steps to show there’s a legal run of $B$ that achieves those scanned values.

Each $p_i$ simulates its own shared memory for all the $q_j$’s. Each processor is simulating an entire global parallel execution of $B$.

3 Safe agreement module

This idea of a safe agreement module is formalized using input/output automata. For each processor, there must be a propose. But it’s not always going to be true that processors can get out of the protocol, especially in event of failures. We will show that any particular processor
Algorithm 1 Protocol for processor $p_i$

$\rightarrow$ propose($v_i$)

Initially, level$_i = 0$, val$_i = \perp$

1. val$_i := v$
2. level$_i := 1$
3. read $(\text{val}_j, \text{level}_j)$ of all $p_j$
4. if $\exists j \text{ level}_j = 2$
5. then level$_i := 0$
6. else level$_i := 2$
7. repeat read($\text{val}_j$, level$_j$) of all $p_j$ until $\forall j \text{ level}_j \neq 1$
8. choose $j = \min\{k | \text{level}_k = 2\}$
9. return val$_j$

$\leftarrow$ agree($v_i$)

can block only one safe agreement module. So $f$ processors can block up to $f$ modules. Then there can be at most $f$ programs whose progress is blocked.

The naive dovetailing idea of a processor running one step of a program, and then moving onto one step of the next program, is not sufficient, because a process could be in the unsafe portion of two programs at the same time.

Formally, the safe agreement module has two inputs for each $i$: propose($v_i$) and stop$_i$. The stop is an abstract input to the module saying, “This processor is dead.” Of course, the module does not really have to know that a certain processor dies. The formalism gives us a way to argue: “Suppose for no processor, stop$_i$ happens. Then agree$_i$ happens.”

The safe agreement module has two outputs: agree($v_i$) and safe$_i$. The output safe is an abstract output that tells us the unsafe portion of the algorithm (to wit, Lines 1-5) is done.

Any safe agreement module must have the following Safe Properties:

Well-formedness: the module requires at most one propose$_i$ for each $i$.

Agreement: all agree($v_i$) values are identical.

Validity: If agree($v_i$) then $v = w$ for some propose($w$)$_j$.

Any safe agreement module must also have the following Liveness Properties:

Wait-free Progress: In any fair execution for any $i$, if propose$_i$ event occurs and no stop$_i$ event occurs, then the safe$_i$ event eventually occurs.

Safe Termination: In any fair execution, if there is no $j$ such that propose$_j$ occurs and safe$_j$ does not occur, then for any $i$ if propose$_i$ event occurs and no stop$_i$ event occurs then agree$_i$ occurs. (Phrased differently, if $\{\text{propose}_j \text{ occurs then safe}_j \text{ occurs}\}$ then for any $i$ if no stop$_i$ occurs, then agree$_i$ occurs.)
There is a safe agreement module for each scan in the program. All the processors access the same safe agreement module for the scan of that program. So the modules are global.

It is clear that the algorithm satisfies Well-formedness and Validity. Satisfaction of Agreement requires proof.

**Lemma 1** Algorithm 1 satisfies the Agreement condition.

**Proof:** Suppose $p_i$ is the first to perform a successful Step 7. That causes it to decide on the value of processor $p_k$. Let $\pi$ be that successful step. Then at $\pi$, $p_i$ sees that level $\neq 1$ for all $j$, and sees level$_k = 2$, where $k$ is the smallest index such that level$_k = 2$. We claim there is no subsequent step $\phi$ such that some process $j$ sets level$_j := 2$. Suppose for contradiction that such a step $\phi$ exists. Then process $j$ sets level$_j := 2$ in step $\phi$. Since level$_j \neq 1$ at $\pi$, it must be that $p_j$ sets level$_j = 1$ in Line 2 and does a scan at Line 3, where execution of Lines 2 and 3 happen after $\pi$ and before $\phi$.

However, at step $\pi$, so level$_j$ must equal zero, as we are assuming level$_j$ is set to 2 in step $\phi$. However, when $j$ runs Line 3 after $\pi$ but before $\pi$, it discovers that level$_k = 2$, so it sets level$_j = 0$. Hence it is impossible that it would set level$_j = 2$ at step $\phi$. Contradiction.

It is obvious that the algorithm satisfies the Wait-free Progress condition, because of the steps of the unsafe portion of the algorithm. Processors do not have to wait.

To see that the algorithm satisfies Safe Termination, assume there is no $j$ such that propose$_i$ occurs but safe$_j$ does not occur. Then Step 5 or 6 has to happen, and no process will stay at level$_= 1$ forever. Assuming $p_i$ is nonfaulty, $p_i$ will get an agree$_i$.

That proves the algorithm has the properties of a safe agreement module.

### 4 Conclusion

We have not discussed how to ensure processors agree on what inputs to use to start the simulation of the run of $B$. In other words, we need an analogue of the function $f$ in the sequential many-one reducibility from $A$ to $B$ where $x \in A$ iff $f(x) \in B$. One way to think about what we have done so far is that it is as though each process has its own input, and the processors, along with knowing their programs, also know all the inputs.