CS 810: Complexity Theory 2/12/2007

Lecture 10: Parallelism

Instructor: Dieter van Melkebeek Scribe: Matt Elder

The goal of parallelism is to speed up computation by dividing it among many processors. In this lecture, we discuss models for parallelism and complexity classes that capture efficiently parallel-computable problems.

1 Conceptual Model

We want our model of parallelism to be of roughly the same power as large collections of Turing machines, all acting together through some means of communication.

Our model must capture this means of communication. What form the interconnections between processors? In our model, we will blithely assume that connections are free. This is not realistic; in real parallel computers the interconnection network is one of the most confusing pieces. However, there are several network configurations, like butterfly nets and hypercubes, that grow at reasonable rates and yield communication between any two processors in $\log p$ time, where p is the number of processors.

Our model must also impose some limits on the number of processors, and here our model must diverge somewhat from physically realizable computers. If we allow only a constant number of processors, then we can give only constant speedup over a standard Turing machine for any problem. Thus, we must allow the number of processors to grow with the input size. This will entail issues of uniformity.

Our criteria for efficiency change when we move from standard Turing machines to vastly parallel computers; we will now aim for polylog time instead of polynomial time. To achieve this, we will permit a number of processors polynomial in the size of the input.

2 Concrete Model

We model parallel computation with *Uniform NC-Circuits*. The class NC is very similar to AC and is defined as follows:

Definition 1. NC^k *is the set of all languages recognizable by circuits with bounded fanin, polynomial size, and* $O(\log^k n)$ *depth. NC is the union of all classes* NC^k*, that is,* $\mathrm{NC} = \cup_{k\geq 0}\mathrm{NC}^k$.

An NC Circuit is uniform *if the circuit can be computed from the size of its input in logarithmic space. The uniformity condition is considered standard: unless otherwise specified, "NC" typically means "Uniform NC."*

Thesis 1 L has an efficient parallel algorithm iff L is in Uniform NC.

We leave it to the reader to verify that the complexity class Uniform NC^k is closed under composition and log–space mapping reductions.

3 Complexity of NC

Next we place the class L among the classes in the NC^k hierarchy.

Theorem 1. *Uniform* $NC^1 \subseteq \mathrm{L} \subseteq$ *Uniform* NC^2.

Proof. First, we show that Uniform $\mathrm{NC}^1 \subseteq L$. Suppose that a uniform family F of NC^1-circuits decides language A. Then, given an input x, we can simulate F in logarithmic space as follows:

1. Compute the circuit C appropriate for $|x|$. More precisely, compute each bit of the description of C as it is needed. We do not have enough space to store the entire description of the circuit, but we can compute each part as we need it in logarithmic space because F is uniform.

2. From the output node of C, compute the values of each gate in C recursively, without memoization. This is painfully slow, but we wish to optimize for space. Since the depth of C is in $O(\log |x|)$, we can do this computation in logarithmic space.

3. If the output of C is 1, accept. Else, reject.

The fact that $L \subseteq$ Uniform NC^2 is left as an exercise. □

We can also relate every class in the NC hierarchy to classes in the closely related AC hierarchy, as follows:

Theorem 2. $\mathrm{NC}^k \subseteq \mathrm{AC}^k \subseteq \mathrm{NC}^{k+1}$.

Proof. Since NC^k is a restriction of AC^k, the first inclusion is clear. The second inclusion is implied by the fact that polynomially-bounded fanin can be simulated by a simple logarithmic-depth circuit of bounded fanin. □

4 Languages in Various NC^k

To give a feel for the NC hierarchy, we see how efficiently some basic tasks can be accomplished in parallel.

4.1 NC^0

The class NC^0 contains, by definition, only those languages decidable by constant-depth, constant-fanin circuits. This implies, among other things, that these problems must be decidable by checking only a constant number of bits of the input.

4.2 NC^1

By Theorem 2, this class contains AC^0, and thus contains, for example, binary addition. This class also contains iterated addition.

Theorem 3. *Iterated addition is in* NC^1.

```
   10010101
   01111011
 + 00111101
 ----------
   11010011   mod 2
+ 001111010   carries
```

Figure 1: Finding two numbers with the same sum as three different numbers.

Proof. Given three binary numbers, we can output two numbers with the same sum using a constant depth circuit. Each triple of input bits contains 0 to 3 ones; thus we produce one binary number containing the value of these additions modulo 2, and another binary number containing the carries. For example, see Figure 1.

This operation is possible with constant-depth circuits. So, we group all of our inputs into groups of 3, apply this operation, and repeat until there remain only two inputs. This operation reduces the number of remaining numbers to add by $1/3$, so some logarithmic number of layers of these circuits reduces the problem to binary addition, which is in NC^1, as we have already seen. $\square$

Because iterated addition is in NC^1, binary multiplication is also in NC^1. We can also perform matrix multiplication in NC^1: we do every useful element-wise multiply in one binary multiplication layer, and follow it by a layer of iterated addition. Both subproblems are in NC^1, so their concatenation is in NC^1.

A symmetric function is a function whose value does not change when its input bits are permuted; thus, its value is dependent only on the size and the number of ones in the input. Both of these can be determined by iterated addition, so all symmetric functions are NC^1–computable, though not necessarily in Uniform NC^1.

It is known that iterated multiplication is also Uniform NC^1–computable, though this proof is more complex.

4.3 NC^2

By a divide-and-conquer algorithm using circuits in NC^1, we can show that iterated matrix multiplication is in NC^2. This implies that matrix inversion, linear systems, and most of the rest of linear algebra is NC^2–computable.

4.4 Upper Bounds

For the classes NC^k with $k > 0$, known upper bounds on computation power are quite weak. For example, the truth of the following statements are all open questions:

- $P \subseteq$ Uniform NC^1?
- $P \subseteq$ Uniform NC?
- $NP \subseteq$ Uniform NC^1?

CVP is the Circuit Value Problem: Given a circuit C and an input x, return what C would output given x. Because CVP is P-complete under log-space mapping reductions, we know that P is in Uniform NC iff CVP is NC-computable.

5 Connection Between NC^1 and BP

The following theorem states that NC^1 circuits and bounded-width branching programs of polynomial size are equally powerful. The proof uses formulas, a restricted version of circuits.

Definition 2. *A* formula *is a circuit in which all gates have a maximum fanout of 1.*

Since every gate in a formula has a maximum fanout of 1, the number of gates in a formula matches our notion of the size of a boolean expression. A standard circuit may have the shape of any directed acyclic graph, but a formula must look like a rooted tree, except at the inputs. Thus, a circuit might be much smaller than an equivalent formula by reducing duplication and sharing outputs.

Theorem 4. *The following are equivalent in power:*

1. NC^1 *circuits*
2. *Polynomial–size formulas*
3. *Log–depth formulas*
4. *Bounded-width branching programs of polynomial size*

Proof. We will show that each of the theorem's elements can simulate its predecessor in the above theorem; so, poly-size formulas capture NC^1 circuits, log–depth formulas capture poly-size formulas, and so on.

Proposition 1. NC^1 *circuits can be simulated by poly-size formulas.*

Proof. A circuit forms a rooted directed acyclic graph, with its root at the topmost operator. Given an NC^1–circuit, we can transform it into a formula by recursively replacing subgraphs. For each node with fanout k, with $k > 1$, we replace that node (and its child subgraph) with k copies of the node (and its child subgraph) so that each node has fanout 1, and each node is the child of one of the old parents. For example, the black node in Figure 2 gets transformed in this way.

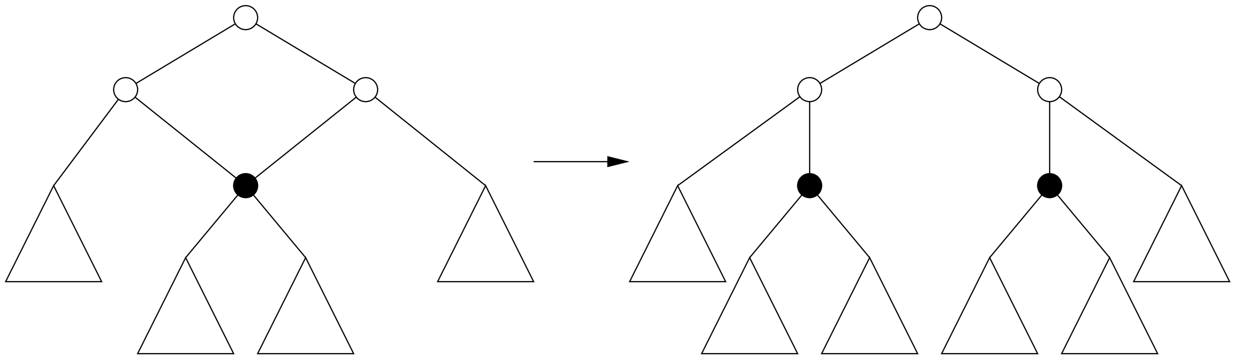

Figure 2: One step of the recursive transformation from NC^1–circuit to poly–size formula.

The circuit has only one root for output. So, suppose we repeat this procedure at every node from the root down on a circuit of maximum fanin f and depth d. The number of nodes at depth

$t+1$ is no more than f times the number of nodes at depth t, so there are at most f^t nodes at layer t. The size of the bottom layer dominates the size of the formula, so this process yields a formula of size $O(f^d)$. Because all NC^1 circuits have depth $O(\log n)$, the size of the formula is $f^{O(\log n)}$, which is polynomial in n. At each step, the function computed by the generated circuit remains the same, so this process creates a poly-size formula equivalent to an NC^1 circuit. □

Proposition 2. *Polynomial–size formulas can be simulated by log–depth formulas.*

Proof. Given a formula with binary fanin, we can find an edge of the formula so that the sub-formulas on either side of that edge are at least 1/3 the size of the whole formula. Let f be the sub-formula at the low side of the cut edge, and let g_x be the sub-formula on the high side of the cut edge with the constant literal x placed where f was. Figure 3 illustrates these two trees.

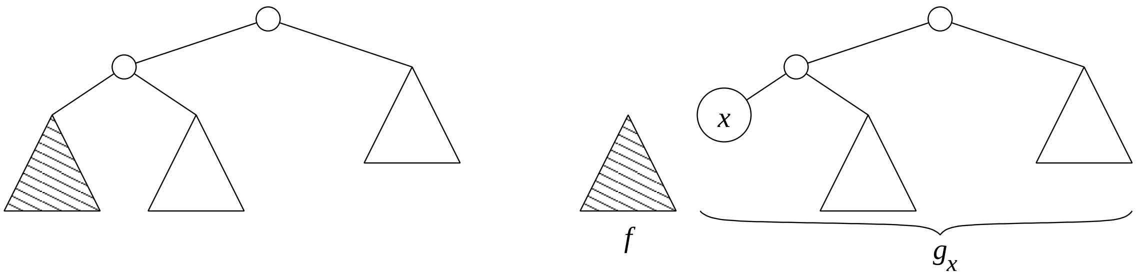

Figure 3: First step in the transformation from polynomial size formulas to log-depth formulas. A cut at a well-chosen edge of a formula yields two sub-formulas, f and g.

From f and g, we create a formula with the same function as the original, but with decreased depth. This formula is $(f \wedge g_1) \vee (\neg f \wedge g_0)$, and is shown in Figure 4. To see that this formula computes the same function as the original, consider the value of f. If f is 1 on its inputs, then the original function would have had the value of g_1. Likewise, if f is 0 on its inputs, then the original function would have had the value of g_0. So, the new function combines both cases.

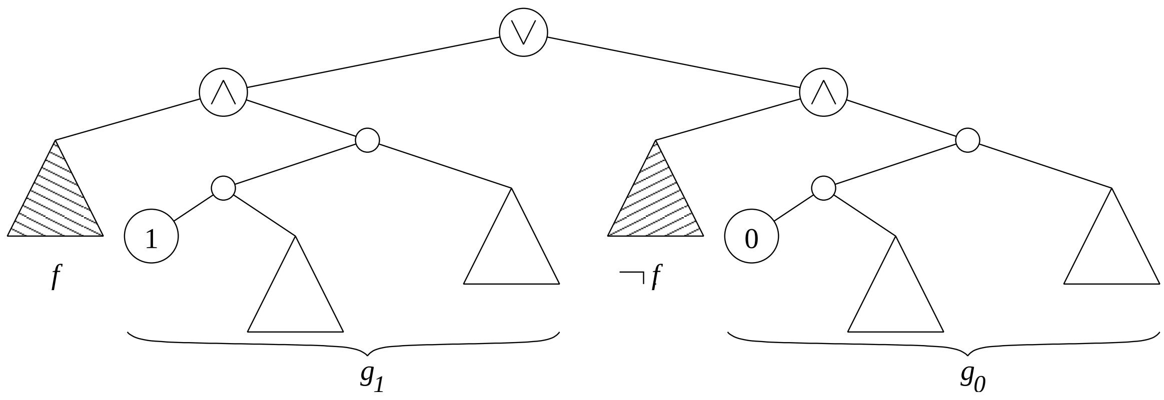

Figure 4: Second step in the transformation from polynomial size formulas to log-depth formulas. How to combine the two sub-formulas.

We then recurse the procedure on the sub-trees of f, g_0, and g_1. We continue recursing until we are considering constant-size formulas. Let s be the size of the original formula. Notice that f, g_0, and g_1 are each of size at most $2s/3$. After applying the next step, the sub-formulas being considered are further reduced by a factor of $2/3$. Also notice that each level of recursion places a depth two circuit at the top of the sub-formula being worked on. Then, the depth d of the final formula generated satisfies the inequality $2 \cdot (2/3)^d s \geq 1$, in other words $d = O(\log s)$. □

Proposition 3. *Bounded-width branching programs of polynomial size can be simulated by* NC^1 *circuits.*

Proof. Suppose B is a branching program of width w, containing a polynomial number of layers p. We construct an NC^1 circuit to simulate B with the following divide–and–conquer strategy:

1. Place an OR gate, with fanin w. We will ensure that this OR gate is true if the input induces a path from the start state in the first layer of B to the accepting state in the last layer (layer p) of B.
2. At the i^{th} input to the OR gate, place an AND gate with fanin two. This AND outputs true if the input induces a path from the start state of the first layer, to the the i^{th} state of layer $p/2$, and to the accepting state of layer p. One input to this AND is true iff the sub-path from layer 1 to layer $p/2$ is induced, and the other input is true iff the sub-path from layer $p/2$ to layer p is induced.
3. Recurse on the inputs of the ANDs until reaching the base case of checking adjacent layers.

Because p is polynomial, this divide–and–conquer strategy recurses only $O(\log(n))$ times, giving the constructed circuit a logarithmic depth. To analyze the size of the circuit, we rely on the fact that we are generating a circuit and not a formula: once a sub-problem is computed once in the circuit, we do not need to compute it again if it is needed again. There are roughly $2p$ intervals considered in subproblems, and w^2 subproblems of the form "Can state a in layer A be reached from state b in layer B?" are asked for each interval. Thus, the number of individual "questions" that our circuit computes is only $2pw^2$, which is polynomial in the size of the input. So, the circuit we have constructed uses a polynomial number of gates; since it also has logarithmic depth, the circuit is in NC^1. □

Proposition 4. *Log-depth formulas can be simulated by bounded–width branching programs of polynomial size.*

Proof. The BP M we construct has the following properties:

- The width of M is 5.
- The label of each node depends only on its layer, that is, M is an *oblivious BP*.
- Between any two levels, all of the 0–branches have distinct end states and all of the 1–branches have distinct end states. Since the width of M is constant, this makes M a *permutation BP*.
- Overall, the effect of M is the identity permutation e if its input is rejected, or a single cycle π if its input is accepted. We say that such a BP is π*–accepting*.

The following claims and corollaries give use the components we'll need to construct M recursively, building structures equivalent to pieces of the log–depth formula we are given.

Claim 1. *If there exists a π–accepting BP of size s, for some cyclic π, then there exists a σ–accepting BP of size s for* any *cycle $\sigma \neq e$.*

Proof. Any cyclic permutation σ is conjugate to π; that is, there exists a permutation γ such that $\sigma = \gamma^{-1}\pi\gamma$. So, to construct a σ–acceptor from a given π–acceptor, we need only to permute the machine's top-layer nodes by γ^{-1} and the bottom-layer nodes by γ. □

Corollary 1. *If we can decide the language A using a π–acceptor of size s, then we can decide the language $\overline{A}$ using a π–acceptor of size s.*

Proof. Build a π^{-1}–acceptor that decides A. Apply π to the last nodes, and you have a π–acceptor that decides $\overline{A}$. □

Claim 2. *If there exists a π–acceptor of size s_A that decides language A and there exists a σ–acceptor of size s_B that decides language B, and $\tau = \pi^{-1}\sigma^{-1}\pi\sigma \neq e$, then there exists a τ–acceptor of size $2(s_A + s_B)$ that decides $A \cap B$.*

Proof. Suppose that M_A is the π–acceptor deciding language A and M_B is the σ–acceptor deciding language B. Let $M_{A\cap B}$ be the machine formed by concatenating M_A^{-1}, M_B^{-1}, M_A, and M_B, in that order, start-to-finish. (M_A^{-1} is a π^{-1}–acceptor deciding A, and M_B^{-1} is analogous. The previous claim shows how to build these.)

Consider the fate of input x fed to $M_{A\cap B}$. If $x \in A \cap B$, then x undergoes the permutation π^{-1} from M_A^{-1}, followed by σ^{-1} from M_B^{-1}, followed by π from M_A, followed by σ from M_B. Because $\tau = \pi^{-1}\sigma^{-1}\pi\sigma$, the net permutation that x undergoes is τ.

If, instead, x is in $A \setminus B$, then M_B^{-1} and M_B perform the identity permutation on x. Therefore, the net permutation experienced by x is $\pi^{-1}\pi$, or just e, and x is rejected. The case where x is in $B \setminus A$ is similar, and the case where x is in neither A nor B is even clearer. So, the machine $M_{A\cap B}$ accepts precisely the language $A \cap B$, and the size of the machine $2(s_A + s_B)$. □

There exist permutations π and σ so that $\tau \neq e$, for example, let $\pi = (12345), \sigma = (13542)$, and $\tau = (12534)$. Thus, we will be able to combine branching programs to simulate AND gates; and since we've already seen how to complement these branching programs, we can simulate OR gates by De Morgan's law.

We need to encode individual input variables as well, but these are trivial within this model: the 0–branches form the permutation e and the 1–branches form the permutation π, and the label of every node in the layer is the relevant variable.

So, consider the standard post-order formula traversal that these constructions suggest. At each level, our branching program may increase by no more than a factor of two. If the given formula has depth d, then the branching program we construct has size $O(2^d)$. Since d is $O(\log n)$, the size of the branching program is polynomial in n. □

So, we have proved that each of the four computational models in question can be transformed into another of the four. Since we can chain these transformations as we please, any machine in any these models can be transformed into an equivalent machine in any of the other models. All four models are therefore equivalent in power. □