Efficient Divide-and-Conquer Simulations Of Symmetric FSAs

David Pritchard,
University of Waterloo
June 23, 2008

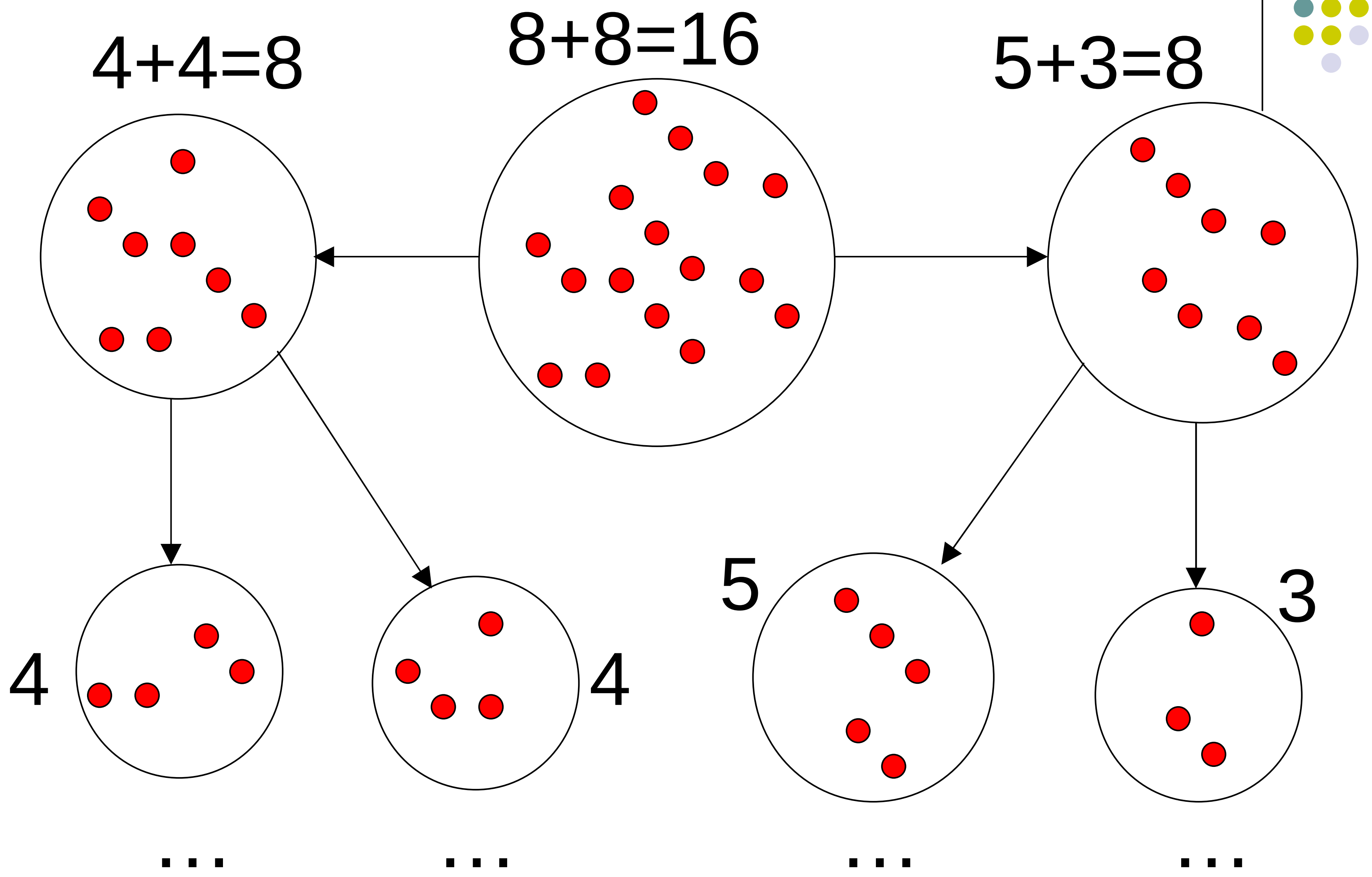
Jist of Talk

- A “computational process” takes a stream of input symbols and outputs a single result
- Divide-and-conquer is a computing paradigm needing less central coordination
- Can we transform any* computational process into an equivalent divide-and-conquer one? If so, can we make the divide-and-conquer one *efficient*?
  - *we’ll look at Finite State Automata (FSA), the simplest kind of computational process

Overview

- An example of divide-and-conquer
- Definition of FSA's (transformation monoids)
- Basic divide-and-conquer FSA simulation [Ladner-Fischer 1977]
- Positive result: improved efficiency for *symmetric* FSA's
- Negative result: no improvement possible for *asymmetric* FSA's

Divide-and-Conquer Example



Divide-and-Conquer, Generally

- A divide-and-conquer computational process should have
  - temporary results of intermediate computations (e.g. counts of subpiles)
  - a rule for combining temporary results (e.g. +)
  - a rule for computing “base case” results when there is only one input (e.g. “1”)
  - a rule for interpreting the final result as output
- such that the same final answer is obtained no matter how the inputs were divided.

Another Example of Divide-and-Conquer

- Given a picture book, tell me the maximum number of consecutive pages w/ monkeys
- How can we do it via divide-and-conquer?
  - Each subproblem is a contiguous bunch of pages
  - Need each "intermediate result" to contain 3 numbers: maximum number of consecutive monkey pages, # of initial monkey pages, # of final monkey pages. Then we can do it!
- Next: let's get formal

Finite-State Automata (1/3)

- From now on, consider only "finite-state" computational processes
  - Explicitly, there needs to be a finite set such that all possible intermediate results are drawn from this set
- A *finite state automaton* keeps track of a single intermediate result (its "state") at each moment of time; reads one input symbol at a time; and for each pair of (current state, input symbol) has a rule telling which state is next
  - Some motivation: FSA's are fundamental in theory of computation

(Definition of) Finite-State Automata (2/3)

- Input alphabet X, output alphabet O, state space Q, all finite. Initial state q_0 in Q.
- Transition function $f_\sigma : Q \to Q$ for each σ in Σ.
- On input string $\alpha\beta\gamma\ldots\omega$ start in state q_0, then apply f_α to current state, then f_β, etc.
  - i.e., (Q, Σ, f) is a transformation monoid
- FSA also has post-processing f^n $\Pi : Q \to O$
  - Output value is $\Pi(q)$ where q is final state.
  - E.g. could have O = {accept, reject}

Finite-State Automata (3/3)

- Explicitly, output of FSA (Q, Σ, f, O, Π) on input $\alpha\beta\gamma\ldots\omega$ is

$$\Pi(f_\omega(\ldots f_\gamma(f_\beta(f_\alpha(q_0)))\ldots))$$

- Suppose we build an FSA to read a string of jellybean colours (of which there are finitely many possible, Σ). We can compute, e.g.:
  - How many red jellybeans are there (mod 10)?
  - Are there at least 20 blue jellybeans?
  - Was there a subsequence (red, blue, green, red)?

Equivalence of FSA's

- Each FSA yields a function that
  - takes an arbitrary string *w* over Σ as input,
  - yields an element of O as output
- We identify each FSA with its function and we say that two FSA are *equivalent* if they compute the same function $\Sigma^* \rightarrow O$
  - Or, that the FSA's *simulate* each other
- Next: put divide-and-conquer in this framework

Divide-and-Conquer Analogue of FSA's (informal definition)

- Terminology: "intermediate results" ⇔ "states"
- Computational process using finite state space Q:
  - Input string is partitioned into two parts (left, right substring)
  - Have a base case B if string has only one character
  - Recursively obtain an intermediate result q from each part
  - Use a deterministic rule C to combine left & right results
  - Post-processing function Π maps to output alphabet.
  - Overall, computes a function $\Sigma^* \rightarrow O$ just like an FSA.
- Definition: If output is independent of how the division was performed, (Q, Σ, B, C, O, Π) is a *Divide-and-Conquer Automaton* (*DCA*).

[Ladner & Fischer '77] Functional Composition Idea

- Th^m: Can simulate any FSA (Q, Σ, f, O, Π) with a DCA.
- Proof: Note, output of FSA on input $\alpha\beta\gamma\ldots\omega$ is

$$\Pi(f_\omega(\ldots f_\gamma(f_\beta(f_\alpha(q_o)))\ldots))$$
$$= \Pi(f_\omega \circ \cdots \circ f_\gamma \circ f_\beta \circ f_\alpha(q_o))$$

- Key observation: composition ($\circ$) is associative and there are finitely many functions from $Q \to Q$
- "base case" for character σ is f_σ
- "intermediate result" for substring $\kappa\lambda\ldots\pi$ is $f_\pi \circ \cdots \circ f_\lambda \circ f_\kappa$
  - combining rule is (f_{left}, f_{right}) |-> $f_{right} \circ f_{left}$
- In post-processing, $f_\omega \circ \cdots \circ f_\alpha \mapsto \Pi(f_\omega \circ \cdots \circ f_\alpha(q_o))$.

Remark

- Corollary of Ladner-Fischer: {class of all functions computed by FSA's} = {class of all functions computed by DCA's}
  - Proof: Ladner-Fischer showed ≤. To see that ≥ holds, observe that every FSA can be rewritten as a DCA that "conquers" one input at a time. ∎

Part 2: Efficiency

Definition of Symmetric FSA's

- $f\colon \Sigma^* \rightarrow O$ is symmetric if, for every string *w* and every permutation *w'* of *w*, $f(w)=f(w')$
- An FSA is symmetric if the function it computes is symmetric. Similarly for DCA's.
- Some motivation from P.-Vempala '06
  - FSSGA distributed computing model = graph w/ same symmetric FSA at each node
    - Symmetric computation => fault-tolerance, empirically
  - Showed {class of all symmetric functions computed by FSA's} = {"mod-thresh formulae"}

FSSGA Applet

FSSGA Update via Divide-and-Conquer

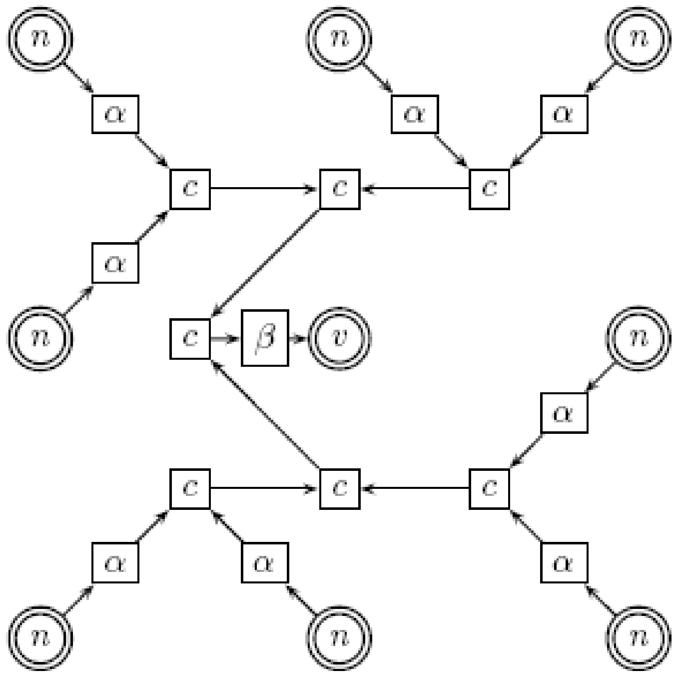


FIGURE 1

An FSA in a network updates its state via divide-and-conquer. The node v is activating and its neighbours are labeled n. The lines carry values from tail to head, and the boxes apply functions, like in a circuit diagram. Each neighbour supplies an input symbol and the divide-and-conquer process produces an output symbol which is used by v to update its state.

Main Results

- The Ladner-Fischer FSA->DCA conversion entails an exponential increase in the state space size (i.e., from $|Q|$ to $|Q|^{|Q|}$)
- Main result: a way to convert a **symmetric** FSA to a DCA **without any increase** in size of state space.
- Can also show that if we don't assume symmetry, Ladner-Fischer result is optimal

2 Applications of Main Result

- Can *efficiently* implement the “read all neighbours and update” step in FSSGA model via the divide-and-conquer circuit
- Divide-and-conquer lets us simulate FSA’s in the model of parallel processing; for symmetric FSA’s our conversion makes these parallel programs use less memory

Main Lemma (1/3)

- For string $S = \alpha\beta\ldots\omega$ define $f_S = f_\omega \circ \cdots \circ f_\beta \circ f_\alpha$
- State q of FSA inaccessible if no string S has

$$f_S(q_o) = q.$$

- States q, q' are indistinguishable if for all S,

$$\Pi(f_S(q)) = \Pi(f_S(q')).$$

- If an FSA has no inaccessible states and no indistinguishable pairs, it is irredundant.
- Claim: we can make any FSA irredundant without changing the function it computes.
- Aside: FSA is irredundant iff it is "minimal" (smallest FSA to compute its function) [Myhill-Nerode '58]

Main Lemma (2/3)

- Statement of main lemma: if an FSA is irredundant and symmetric, then its transition functions $\{f_\sigma | \sigma \text{ in } \Sigma\}$ commute.
  - Symmetry is a black-box property; add to it the innocent-looking "white-box" property of irredundancy and we get a "white-box" result (commuting transition functions).
- We then obtain a simple D&C construction with a reasonably short proof of correctness.

Main Lemma (3/3)

In a symmetric irredundant FSA, f_σ's commute.

- Say input symbols σ,σ' have $f_\sigma(f_{\sigma'}(q)) \neq f_{\sigma'}(f_\sigma(q))$
- By distinguishability some string S has

$$\Pi(f_S(f_\sigma(f_{\sigma'}(q)))) \neq \Pi(f_S(f_{\sigma'}(f_\sigma(q)))).$$

- By accessibility some string T has $q = f_T(q_0)$.
- ⁂ $\Pi(f_S(f_\sigma(f_{\sigma'}(f_T(q_0))))) \neq \Pi(f_S(f_{\sigma'}(f_\sigma(f_T(q_0))))).$
- But this says that outputs on inputs $T\sigma'\sigma S$ and $T\sigma\sigma' S$ differ, contradicting symmetry. ∎

Intermission

- We will show shortly how the Main Lemma is used to obtain the efficient simulation
- Meanwhile, notice that the content of the Main Lemma is that we really care about finite abelian transformation monoids
- Finite abelian groups are very well-understood. Is there a known structure for finite abelian monoids?

Construction, Proof Idea (1/2)

- Given: symmetric irredundant FSA.
- Wanted: equivalent DCA with few states.
- Construction: For each state q fix a ***representative string*** $r[q]$ that brings FSA to state q from q_o,

$$f_{r[q]}(q_o)=q$$

- Easy claim: for every string S, where $f_S(q_o)=q$,

$$f_S = f_{r[q]}$$

  i.e., we can swap S for $r[q]$ wherever it appears in the input. (This is trivially true at start of input.)
- => Key observation: for intermediate result q, we may assume $r[q]$ was the substring to generate it

Construction, Proof Idea (2/2)

Definition of the DCA to simulate the FSA

- DCA intermediate state space = FSA state space; its size could only have decreased when redundancy was removed.
- Base case: map input character σ to $f_\sigma(q_0)$.
- Combining: map pair (q, q') to $f_{r[q']}(q)$.
- Post-processing: use same Π as FSA did
- Proof of correctness is straightforward, using claim and observation from previous slide

The Negative Result (sketch)

- For any $n \geq 1$, there is an n-state FSA on a three-symbol alphabet Σ so that any equivalent DCA has at least n^n states.
- Idea: set $Q=\{1,\ldots,n\}$. Want the groupoid generated by $\{f_\sigma | \sigma \text{ in } \Sigma\}$ to be the set Q^Q of all transformations.
  - [Dénes '68]: such a generating set of size 3 exists
- Then argue that every function in Q^Q needs its own intermediate result in the DCA.

Related/Future Work

- [Feldman et al. '08]: independent work with analogous notions to FSA's and DCA's but on Turing machines
  - One consequence: for probabilistic symmetric automata, efficient simulation is not possible
- Is there an analogue of these results for nondeterministic FSAs?
- If *f* is given implicitly, as a Turing machine, is efficient FSA->DCA conversion possible?
  - Partial answers are known

Thanks for listening!