Parallel Algorithms and Sequential Languages

Guy Blelloch and John Greiner

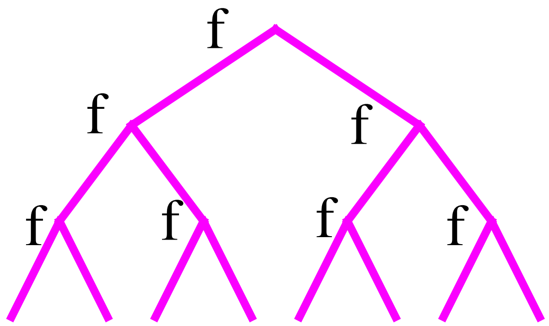
Carnegie Mellon University

Inherent Parallelism in Functional Languages

Referential transparency allows parallel execution of arguments in functional languages.

e.g. the following can be evaluated on a tree

$f(f(f(x_1, x_2), f(x_3, x_4)), f(f(x_5, x_6), f(x_7, x_8)))$



Various Implementation Techniques

- Parallel graph reduction
- Dataflow
- Compiler techniques

How Parallel are Functional Languages?

Some questions to ask:

- How fast can we sort as a function of the input size?
- What kind of sort would we use and how should we represent the data?
- How do bounds relate to other models, such as PRAM?
- Does it matter whether the language is eager or lazy?

To answer these we need:

- A formal model of what "time" is
- To relate the model to other models (simulation bounds)
- To use parallel algorithms

This talk is not about:

- Adding parallel constructs
- Program transformations to make a program more parallel

For now assume:

- ML-like language (eager), but with no side effects
- Arguments can be evaluated in parallel

What About Sorting?

How fast can we sort a list?

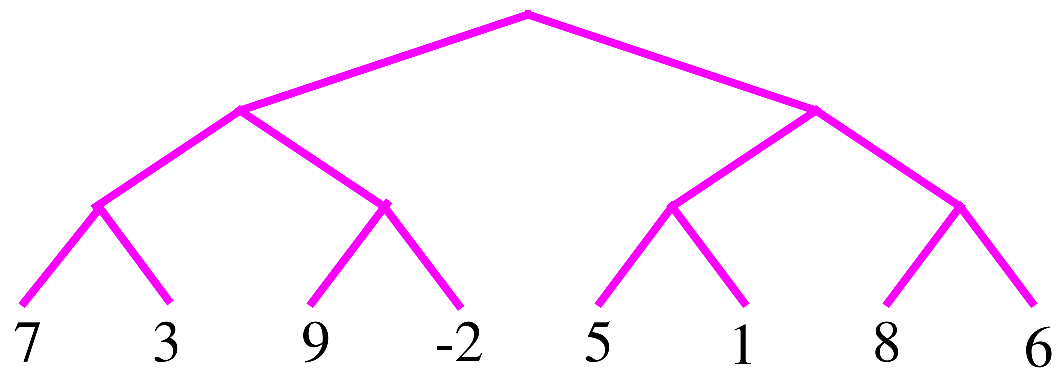
Just to go to the end of the list requires $\Omega(n)$ time

Not hard to show that mergesort takes O(n) time

Only get O(log n) speedup over sequential code

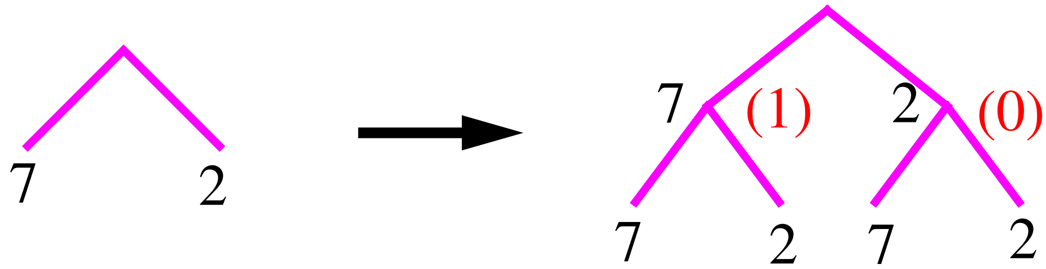
What about sorting data on a tree?

Store the values at the leafs of a balance binary tree

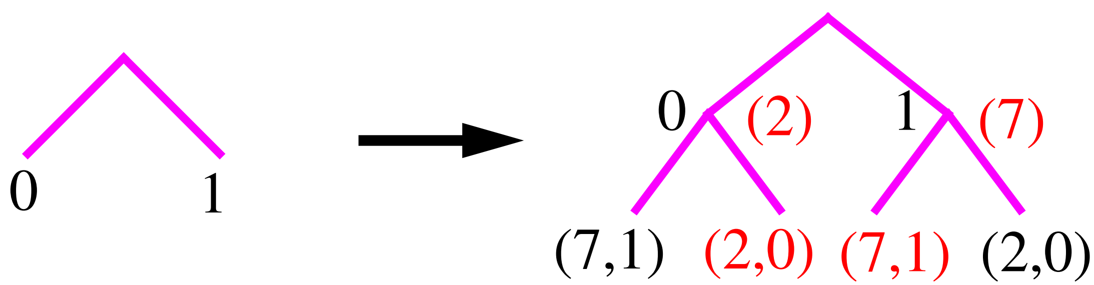


An O(log n) step parallel sort

1. Every key counts number of keys less than it (or equal but to the left). This gives every key its final location.



2. Every of the n locations picks out its key .



What's Wrong?

Requires a total of $O(n^2)$ comparisons.

Not cost effective!!!!

Should keep track of two measures:

- Work -- total number of operations
- Steps -- parallel depth of the computation

These are the complexities used in NESL

Very important when relating to other models.

Definition: Work efficient

When work is equal to best serial running time.

Fast and Work Efficient Sorting

Consider parallel variants of the following sorts:

- Heapsort --- very hard to parallelize
- Quicksort -- can parallelize, but gives expected bounds
- Mergesort -- need a parallel merge

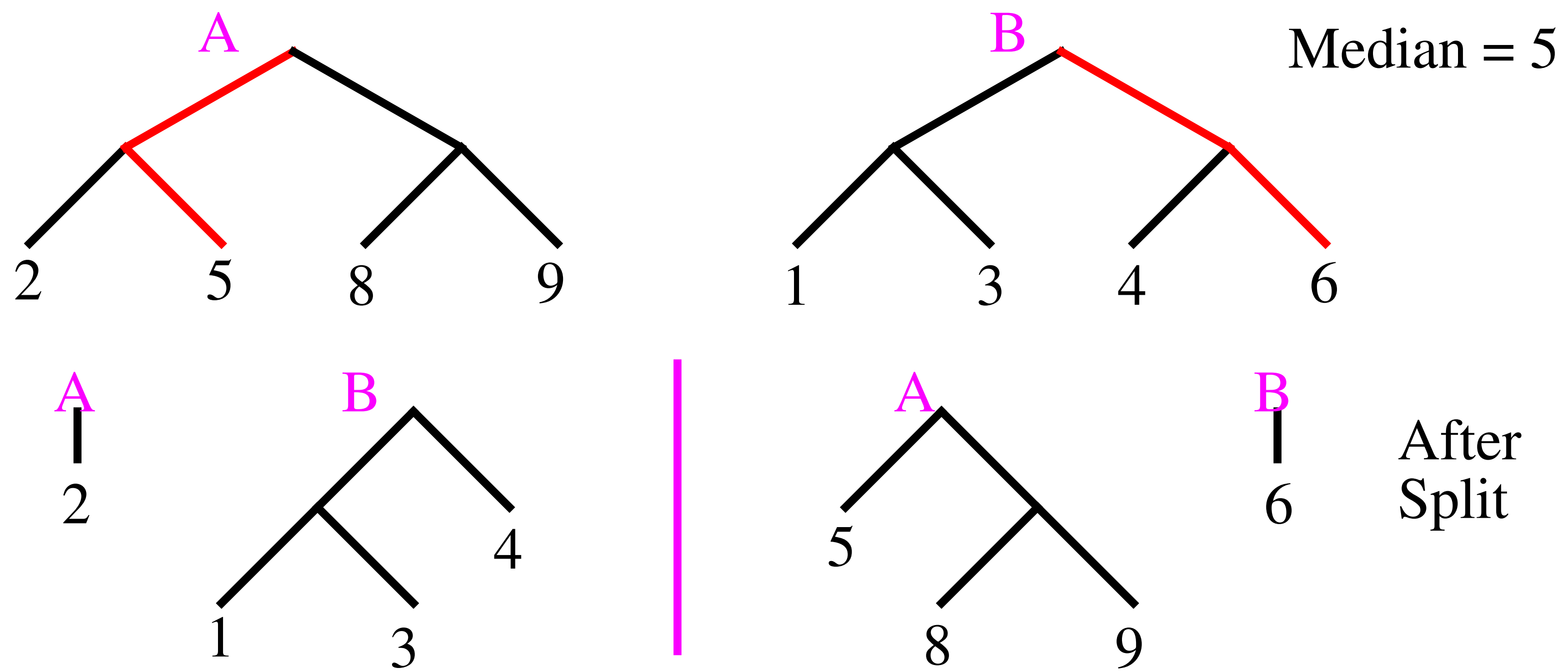
Parallel Merging:

Sequential merge clearly won't work --- requires O(n) steps

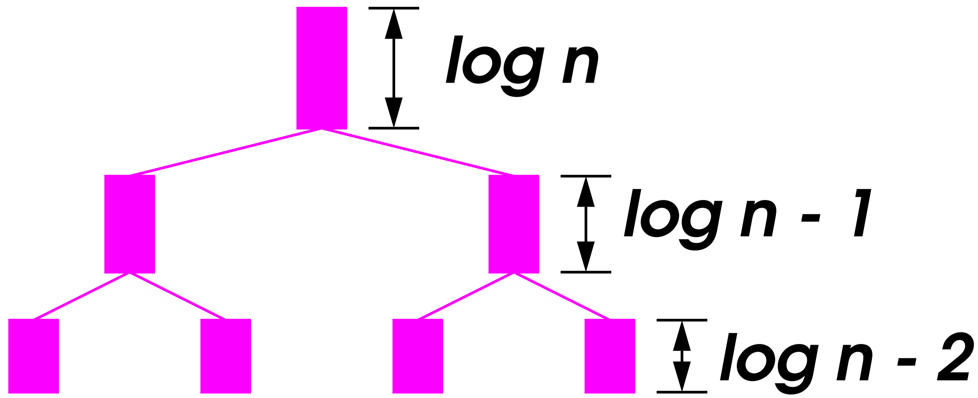
Need some kind of divide-and-conquer algorithm?

Parallel Merging in $O(\log^2 n)$ Steps

- Find median of all elements
  This uses a dual binary search
- Split the two trees based on the median
- Recurse on the two parts of both trees
- Link up the results



Analysis



$$W(n) = 2W\left(\frac{n}{2}\right) + O(\log n) = O(n)$$

$$S(n) = S\left(\frac{n}{2}\right) + O(\log n) = O(\log^2 n)$$

It is work efficient -- can we improve the steps?

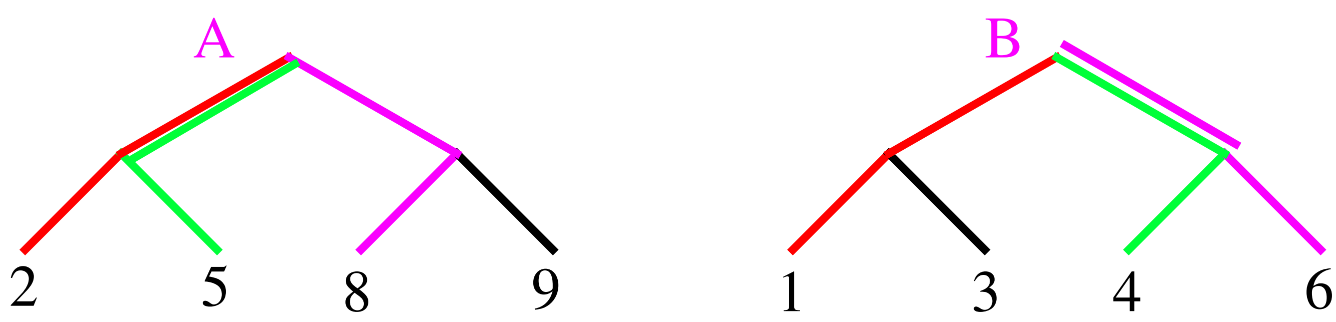
Parallel Merging in O(log n) Steps

For k = [0, log n, 2 log n, 3 log n, ..., n]

- Find kth smallest value in combined trees
- Extract regions from both trees
- Merge Sequentially

Combine results

k = [0, 3, 6] $\qquad$ N_k = [1, 4, 8]



Mergesort: $\quad S = O(\log^2 n) \quad W = O(n \log n)$

Formalizing the model

How do we formalize work and steps?

(1) Use λ-calculus with a "profiling" operational semantics and applicative order reduction

- Most "basic" model
- Normal-order reduction is not naturally parallel
- Easy to use when proving simulation bounds

(2) Prove that can map higher -level constructs onto the λ-calculus with only constant overhead in work and steps

- let, letrec (Y -combinator), datatypes, conditionals

(3) Need to include arithmetic constants otherwise will suffer more than constant overheads.

The Model

Parallel Applicative Lambda-Calculus (PAL)

Arithmetic-PAL (A-PAL)

Relating to other Models

The PAL as a complexity model

Would like to relate it to other standard parallel models

- PRAM
- Butterfly and Hypercube
- LOGP

The class NC are P   AL algorithms with polylogarithmic steps and polynomial size ----- not useful statement in practice.

Is the PRAM or PAL a better model when mapped onto "realistic" machines?

Theorem : If *[ ]* e $v;s,w$, and each of the constant functions can be calculated on one processor in constant time, then v can be calculated from e on a CREW PRAM or a hypercube with P processors within

$$k v_e \left(\frac{w}{p} + s \log P \right)$$

time for some constant k. □

For Example : Our merging algorithm would take time

$$O\left(\frac{n}{p} + \log n \log p \right)$$

The P-ECD machine

State transitions on a state (Q,M) where:

$$Q = [(E_1, C_1, D_1), (E_2, C_2, D_2), \ldots, (E_n, C_n, D_n)]$$

and

$$M = [V_1, V_2, \ldots, V_m]$$

Q is used to keep a set of states, which will be evaluated in parallel. The size will vary from one step to the next since some states will fork into two and others will terminate.

M is used to store partial results when one branch finishes and is waiting for the other .

Mapping the P-ECD machine onto a PRAM

- The states Q and results M are stored in arrays.
- Each step can be executed in time

$$T_i = O\left(\frac{|Q|}{P} + \log P\right)$$

the logP is needed on the PRAM to allocate states to processors.

- Total number of steps can be shown to be S
- $W = \sum_{i=1}^{S} |Q|_i$
- $T = \sum_{i=1}^{S} T_i < k \sum_{i=1}^{S} \frac{|Q|_i}{P} + \log P = k\left(\frac{W}{P} + S \log P\right)$

Merging and Sorting

When mapping onto a hypercube or butterfly network

Merging:

- A-PAL: $\theta(n/p + \log n \log p)$
- EREW PRAM: $\theta(n/p + \log n \log p)$

Sorting:

- A-PAL: $O((n \log n)/p + \log^2 n \log p)$
- EREW PRAM: $\theta((n \log n)/p + \log n \log p)$

Open questions:

- Possible to sort work efficiently in $o(\log^2 n)$?
- Can simulations be improved?
- Can simulations be made more memory efficient?
- Does "call-by-speculation" help for any problems

Problem:

- Lack of random access requires log n extra work for certain problems, such as random permutations.

Related W ork

- Cost augmented semantics (Rosendahl, Sands, ...)
- Call-by-speculation time semantics (Roe)
- NESL -- work and steps with arrays
- PRAM cost semantics (Zimmermann)
- Pomset models of parallelism for functional languages (Hudack)
- Mapping from language to machine models (Jones)
- Sequential Pointer Machines (Ben Amram, T arjan)
- Birds-Merten cost semantics (Skillicorn)

Summary

- Presented a formal parallel cost model for the call-by-value λ-calculus. Related it to ML-like language (recursion, datatypes, conditionals, ...)
- Related the model to standard parallel models ---PRAM, hypercube, butterfly
- Showed algorithms for merging and sorting in the model

Conclusion:

Functional languages with cost model could serve as a reasonable complexity model for parallel algorithms.