Lecture 19

Parallelism

Cost Semantics and Sequences
today

parallel programming

• parallelism and functional style
• cost semantics
• Brent’s Theorem and speed-ups
• sequences: an abstract type with efficient parallel operations
parallelism

exploiting *multiple* processors
evaluating *independent* code *simultaneously*

concurrency

scheduling *multiple* computations *at once*
parallelism

exploiting *multiple processors*

evaluating *independent code simultaneously*

- low-level implementation
  - *scheduling* work onto processors, tell each processor to do at each time step

- high-level planning
  - designing code *abstractly*
  - without *baking in a schedule*
our approach

Deal with scheduling implicitly
• Programmer specifies what to do
• Compiler determines how to schedule the work

Our thesis: this approach to parallelism will prevail..

(and 15-210 builds on these ideas...)
functional benefits

• No side effects, so…
  evaluation order doesn’t affect correctness

• Can build abstract types that support efficient parallel-friendly operations

• Can use work and span to predict potential for parallel speed-up
  • Work and span are independent of scheduling details
caveat

• In practice, it’s hard to achieve speed-up
• Current language implementations don’t make it easy
• Problems include:
  • scheduling overhead
  • locality of data (cache problems)
  • runtime sensitive to scheduling choices
We already introduced work and span

- *Work* estimates the *sequential* running time on a *single* processor

- *Span* takes account of data dependency, estimates the *parallel* running time with *unlimited* processors
cost semantics

• We showed how to calculate work and span for recursive functions with recurrence relations
• Now we introduce cost graphs, another way to deal with work and span
• Cost graphs also allow us to talk about schedules...
• ... and the potential for speed-up
A cost graph is a *series-parallel graph*

- a *directed* graph, with source and sink
- nodes represent *units of work* (constant time)
- edges represent *data dependencies*
- branching indicates *potential parallelism*
series-parallel graphs

a single node

- sequential composition

- parallel composition
work and span
of a cost graph

• The work is the number of nodes

• The span is the length of the longest path from source to sink

\[ \text{span}(G) \leq \text{work}(G) \]
\[ \text{work} = \text{work } G_1 + \text{work } G_2 + c \]

**Sequential Code**: ... add the work

**Independent Code**: ... add the work
\[ \text{span} \]

\[ \text{span} = \text{span } G_1 + \text{span } G_2 + c \]

\textit{sequential code ... add the span}

\[ \text{span} = \max(\text{span } G_1, \text{span } G_2) + c \]

\textit{parallel code ... max the span}
sources and sinks

• Sometimes we omit them from pictures

• No loss of generality
  • easy to put them in

• No difference, asymptotically
  • a single node represents an additive constant amount of work and span

• Allows easier explanation of execution
using cost graphs

• Every expression can be given a cost graph
• Can calculate work and span using the graph
  • These are asymptotically the same as the work and span derived from recurrence relations

work and span provide asymptotic estimates of actual running time, under certain assumptions

work: single processor
span: unlimited processors

basic ops take constant time
scheduling

• Work: number of nodes
• Span: length of critical path

w = 11
s = 4

assign units of work to processors respecting data dependency

uses 5 processors

(i) 1 2 6 3 4
(ii) 7 5
(iii) 9 8
(iv) 10
(v) 11

an optimal parallel schedule
(5 rounds, or 4 steps)
What if there are only 2 processors?

2 processors cannot do the job as fast as 5 (!)
Brent’s Theorem

An expression with work $w$ and span $s$ can be evaluated on a $p$-processor machine in time $O(\max(w/p, s))$.

Optimal schedule using $p$ processors:
- Do (up to) $p$ units of work each round
- Total work to do is $w$
- Needs at least $s$ steps

Richard Brent is an illustrious Australian mathematician and computer scientist. He is known for **Brent’s Theorem**, which shows that a parallel algorithm can always be adapted to run on fewer processors with only the obvious time penalty—a beautiful example of an “obvious” but non-trivial theorem.

Find me the smallest $p$ such that $w/p \leq s$

Using more than this many processors won’t yield any speed-up
A best schedule for 3 processors

(i) 1 3 4
(ii) 2 6 5
(iii) 7 8
(iv) 9 10
(v) 11

3 processors can do the work as fast as 5(!)
next

• Exploiting parallelism in ML
• A signature for \textit{parallel collections}
• \textit{Cost analysis} of implementations
• \textit{Cost benefits} of parallel algorithm design
sequences

signature SEQ =

sig

  type 'a seq

  exception Range

  val empty : unit -> 'a seq

  val tabulate : (int -> 'a) -> int -> 'a seq

  val length : 'a seq -> int

  val nth : 'a seq -> int -> 'a

  val map : ('a -> 'b) -> 'a seq -> 'b seq

  val reduce : ('a * 'a -> 'a) -> 'a -> 'a seq -> 'a

  val mapreduce : ('a -> 'b) -> 'b -> ('b * 'b -> 'b) -> 'a seq -> 'b

  val filter: ('a -> bool) -> 'a seq -> 'a seq

end
implementations

- Many ways to implement the signature
  - lists, balanced trees, arrays, ...
- For each one, can give a cost analysis
- There may be implementation trade-offs
  - arrays: item access is $O(1)$
  - trees: item access is $O(\log n)$
Seq : SEQ

• An abstract parameterized type of sequences
• Think of a sequence as a parallel collection
• With parallel-friendly operations
  • constant-time access to items
  • efficient map and reduce
sequence values

A value of type \( t \text{ seq} \)
is a sequence of values of type \( t \)

- We use math notation like

  \[
  \langle v_1, \ldots, v_n \rangle \\
  \langle v_0, \ldots, v_{n-1} \rangle \\
  \langle \rangle 
  \]

  for sequence values

\[\langle 1, 2, 4, 8 \rangle\] is a value of type \( \text{int seq} \)
Equality

- Two sequence values are (extensionally) equal iff they have the same length and have equal items at all positions.

\[ \langle v_1, ..., v_n \rangle = \langle u_1, ..., u_m \rangle \]

if and only if

\[ n = m \text{ and for all } i, v_i = u_i \]
operations

For our given structure Seq : SEQ, we specify

• the (extensional) behavior

• the cost semantics

of each operation

Other implementations of SEQ may achieve different work and span profiles

Learn to choose wisely!
**tabulate**

\[
\text{tabulate } f(n) = \langle f(0), \ldots, f(n-1) \rangle
\]

- If \( G_i \) is cost graph for \( f(i) \),
  the cost graph for \( \text{tabulate } f(n) \) is

![Diagram showing a series of cost graphs \( G_0, \ldots, G_{n-1} \).]

If \( f \) is \( O(1) \), the work for \( \text{tabulate } f(n) \) is \( O(n) \)

If \( f \) is \( O(1) \), the span for \( \text{tabulate } f(n) \) is \( O(1) \)
examples

- tabulate (\textbf{fn} \text{\textup{x}:int} \Rightarrow \text{\textup{x}}) 6 \quad \langle 0, 1, 2, 3, 4, 5 \rangle
- tabulate (\textbf{fn} \text{\textup{x}:int} \Rightarrow \text{\textup{x}}^*\text{\textup{x}}) 6 \quad \langle 0, 1, 4, 9, 16, 25 \rangle
- tabulate (\textbf{fn} _ \Rightarrow \texttt{raise} \text{\textup{Range}}) 0 \quad \langle \rangle
nth

\[ \text{nth} \langle v_0, ..., v_{n-1} \rangle \quad i = v_i \quad \text{if } 0 \leq i < n \]
\[ = \text{raise Range} \quad \text{otherwise} \]

- Work is \( O(1) \)
- Span is \( O(1) \)
- Cost graph is

Contrast: List.nth work, span \( O(n) \)
length

\[ \text{length} \langle v_1, ..., v_n \rangle = n \]

- Work is \( O(1) \)
- Span is \( O(1) \)
- Cost graph is

Contrast: \( \text{List.length} [v_1, ..., v_n] = n \) work, span \( O(n) \)
map

\[
\text{map } f \langle v_1, \ldots, v_n \rangle = \langle f v_1, \ldots, f v_n \rangle
\]

map \( f \langle v_1, \ldots, v_n \rangle \) has cost graph

\[
\begin{array}{c}
G_1 \\
\vdots \\
G_n
\end{array}
\]

where each \( G_i \) is cost graph for \( f v_i \)

- If \( f \) is constant time, \( \text{map } f \langle v_1, \ldots, v_n \rangle \) has work \( O(n) \), span \( O(1) \)

(contrast with List.map)
reduce should be used to combine a sequence using an associative function \( g \) with a base value \( z \)

- \( g : t \times t \rightarrow t \) is associative iff for all \( x_1, x_2, x_3 : t \)
  \[
  g(x_1, g(x_2, x_3)) = g(g(x_1, x_2), x_3)
  \]

- Sometimes we will assume that \( z \) is an identity element for \( g \), i.e. for all \( x : t \), \( g(x, z) = x \)

We write \( v_1 \ g \ v_2 \ g \ldots g \ v_n \ g \ z \)

for the result of combining \( v_1, \ldots, v_n, z \)

\[
\text{reduce } g \ z \ \langle v_1, \ldots, v_n \rangle = v_1 \ g \ v_2 \ g \ldots g \ v_n \ g \ z
\]
reduce cost

reduce g z \langle v_1, ..., v_n \rangle

n/2 nodes at this level

\begin{itemize}
\item v_1 \ g \ v_2
\item v_3 \ g \ v_4
\item v_n \ g \ z
\end{itemize}

combine with g

about log n levels

W(n) is O(n)
S(n) is O(\log n)
mapreduce

- When g is associative and z is an identity,

  \[ \text{mapreduce } f \ z \ g \langle v_1, ..., v_n \rangle = (f \ v_1) \ g \ldots \ g \ (f \ v_n) \ g \ z \]

- When f, g are constant time,

  \[ \text{mapreduce } f \ z \ g \langle v_1, ..., v_n \rangle \]

  has work \( \mathcal{O}(n) \)

  and \( \text{span } \mathcal{O}(\log n) \)
Example: count

using map

fun sum (s : int Seq.seq) : int =  Seq.reduce (op +) 0 s

type row = int Seq.seq

fun count (class: room) : int =

    sum (Seq.map sum class)
Alternatively

using mapreduce

fun sum (s : int Seq.seq) : int = Seq.reduce (op +) 0 s

type row = int Seq.seq
type room = row Seq.seq

fun count (class: room) : int = Seq.mapreduce sum 0 (op +) class
Let class = \langle rrow_1, ..., rrow_n \rangle, \text{row}_i = \langle x_{i1}, ..., x_{in} \rangle

For each i, \( \text{sum s}_i = \text{reduce}(\text{op +}) 0 \langle x_{i1}, ..., x_{in} \rangle \)

\text{work is } O(n) \quad \text{span is } O(\log n)
Let class = \langle \text{row}_1, ..., \text{row}_n \rangle, \text{row}_i = \langle x_{i1}, ..., x_{in} \rangle

For each i, \text{sum } s_i = \text{reduce}(\text{op } +) 0 \langle x_{i1}, ..., x_{in} \rangle

\text{work is } O(n) \quad \text{span is } O(\log n)

\text{cost graph of } \text{sum row}_i

\text{map sum class} = \langle \text{sum row}_1, ..., \text{sum row}_n \rangle

\text{work is } O(n^2) \quad \text{span is } O(\log n)

\text{cost graph of } \text{map sum class}
Let $t_i = \text{sum row}_i$

$$\text{count } s = \text{sum } \langle t_1, \ldots, t_n \rangle$$

work is $O(n^2)$

span is $O(\log n)$