15-150 Fall 2024
Lecture 18

Parallelism

Cost Semantics and Sequences

today

- 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*

- 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

what can programmers do?

- Lists bake in sequential evaluation. Trees don't.
- Today, we introduce sequences that have a linear structure like lists but offer parallelism of trees.
- Reason about time complexity using work and span

Cost semantics

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 tool to deal with work and span
- Cost graphs also allow us to talk about *schedules...*
- *...* and the potential for *speed-up*

Cost graphs

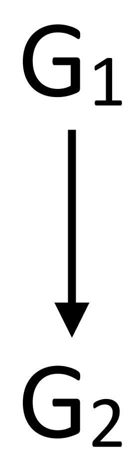
A *cost graph* is a ***series-parallel graph***

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

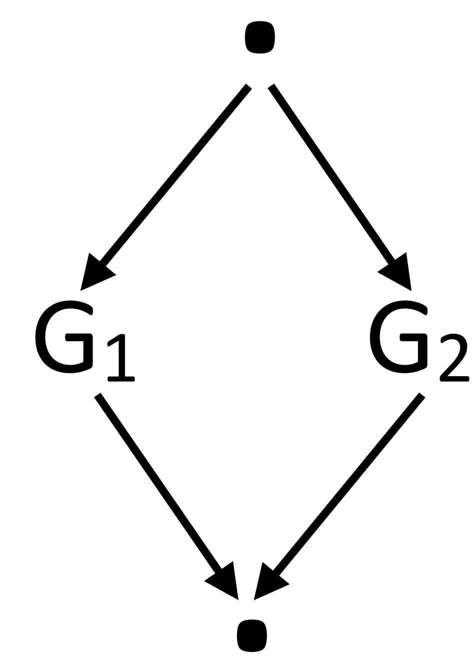
series-parallel graphs

•

a single node



sequential
composition

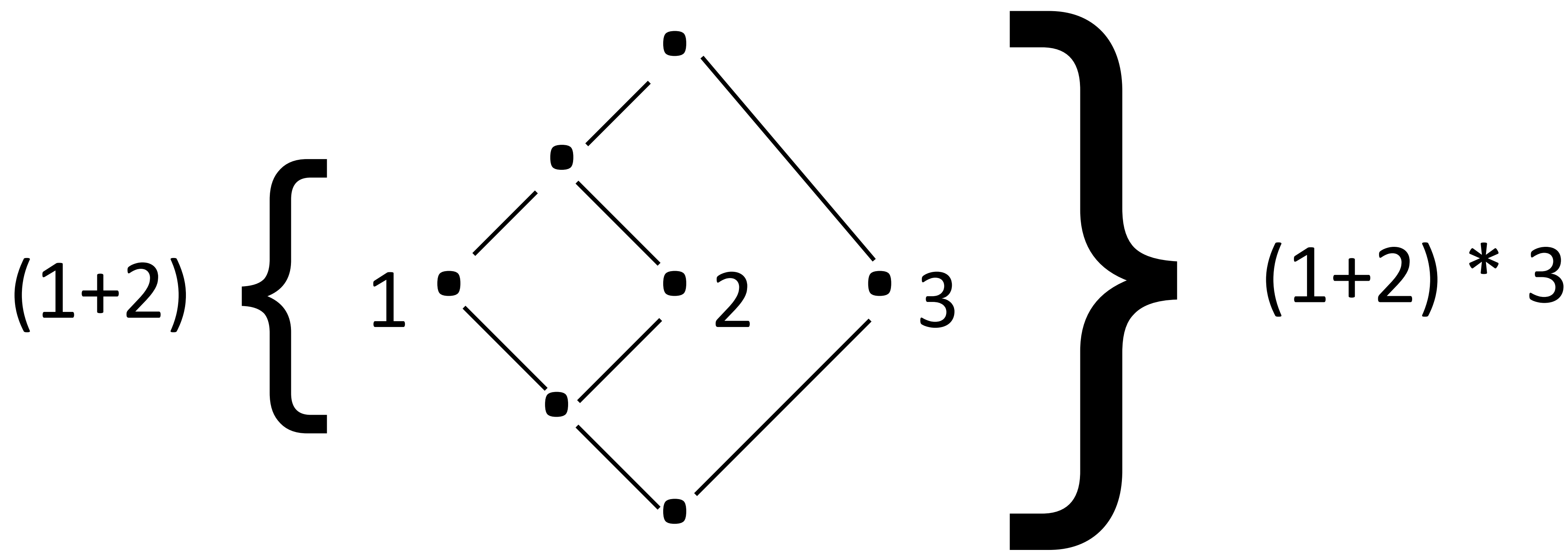


parallel
composition

(n-ary parallelism allowed)

example

(1+2) * 3



(Edges are implicitly directed downward)

work and span

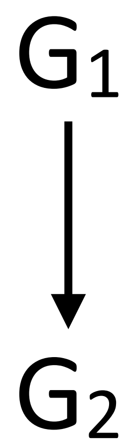
of a cost graph

- The ***work*** is the *number of nodes*
- The ***span*** is the *length of the longest path* from *source* to *sink*

span(G) ≤ ***work***(G)

span

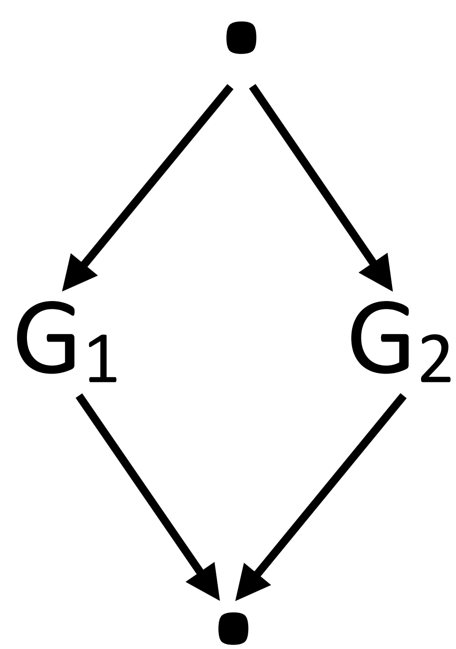
span



$= \text{span } G_1 + \text{span } G_2 + c$

sequential code … add the span

span



parallel
composition

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

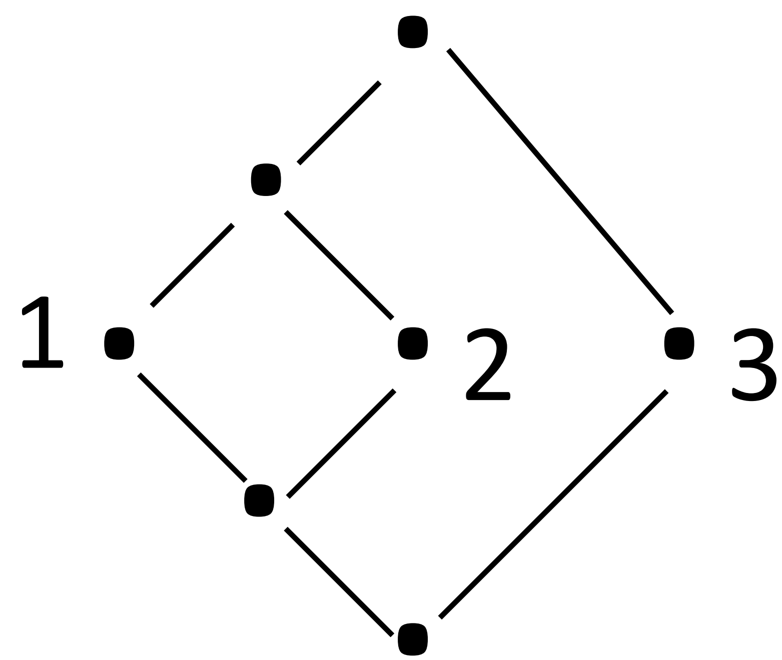
parallel code … max the span

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*

example

(1+2) * 3



work = 7

span = 5

Brent’s Theorem

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

Optimal schedule using **p** processors:

Do (up to) **p** units of work each round

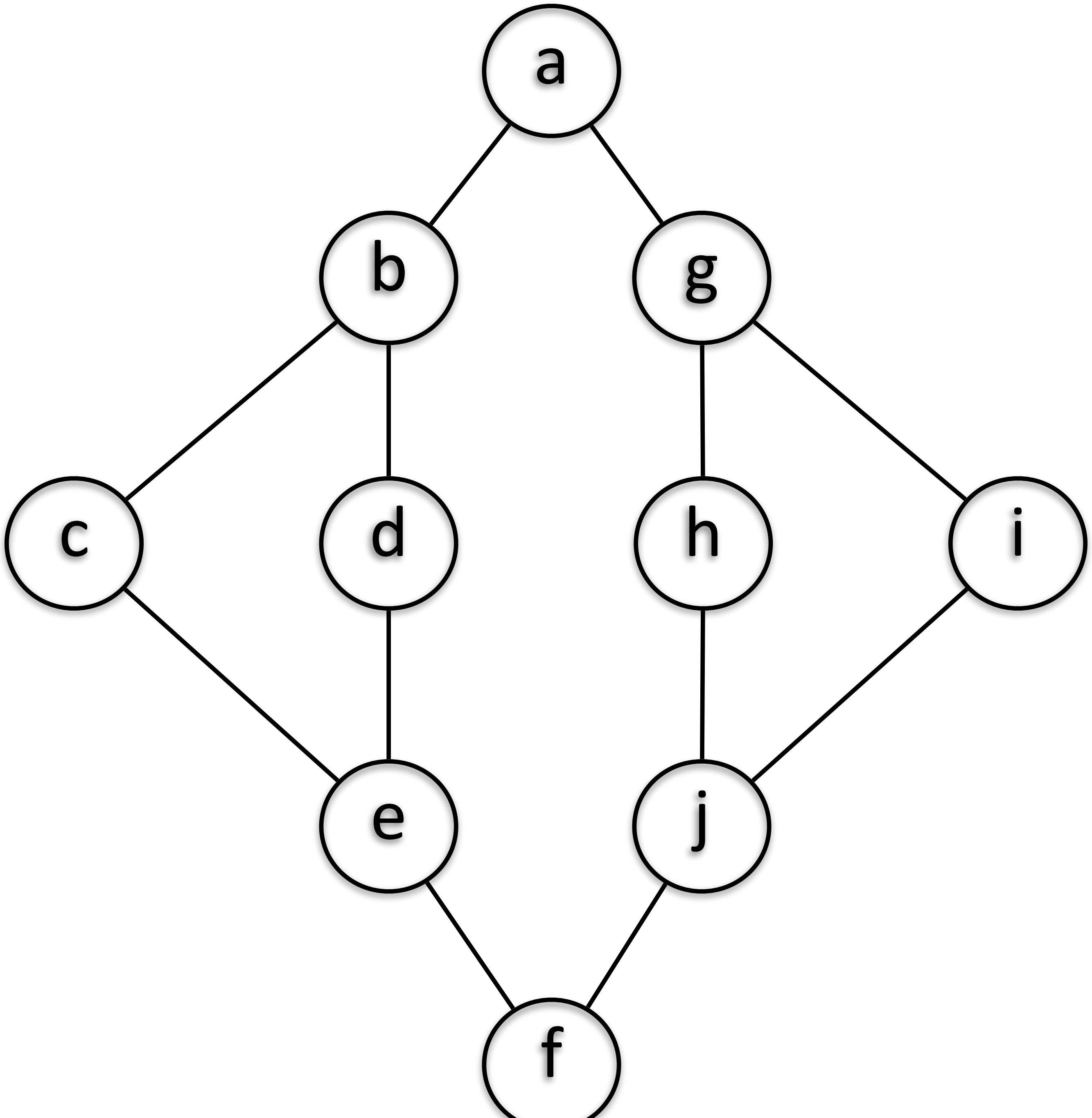
Total work to do is **w**

Needs at least **s** steps

scheduling

- p pebbles, with p the number of processors
- Start with one pebble on cost graph G's source
- Putting a pebble on a node visits the node
- At each time step, pick up all pebbles and put at most p on the graph, no more than one per node. Can only put a pebble on an unvisited model all of whose ancestors have been visited.

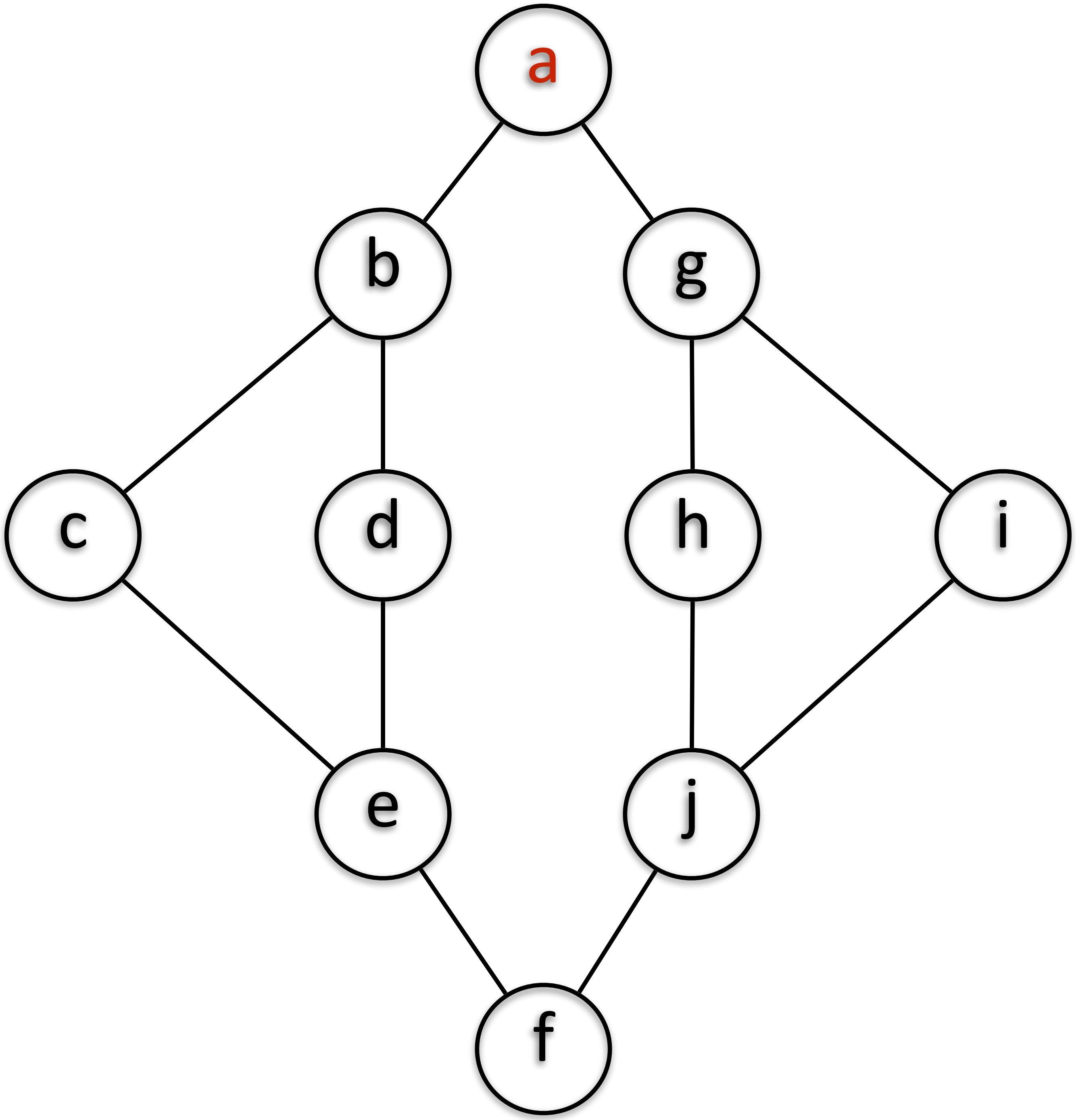
This could be a cost graph for (1+2) * (3+4)



processors

time	1	2
1		
2		
3		
4		
5		
6		

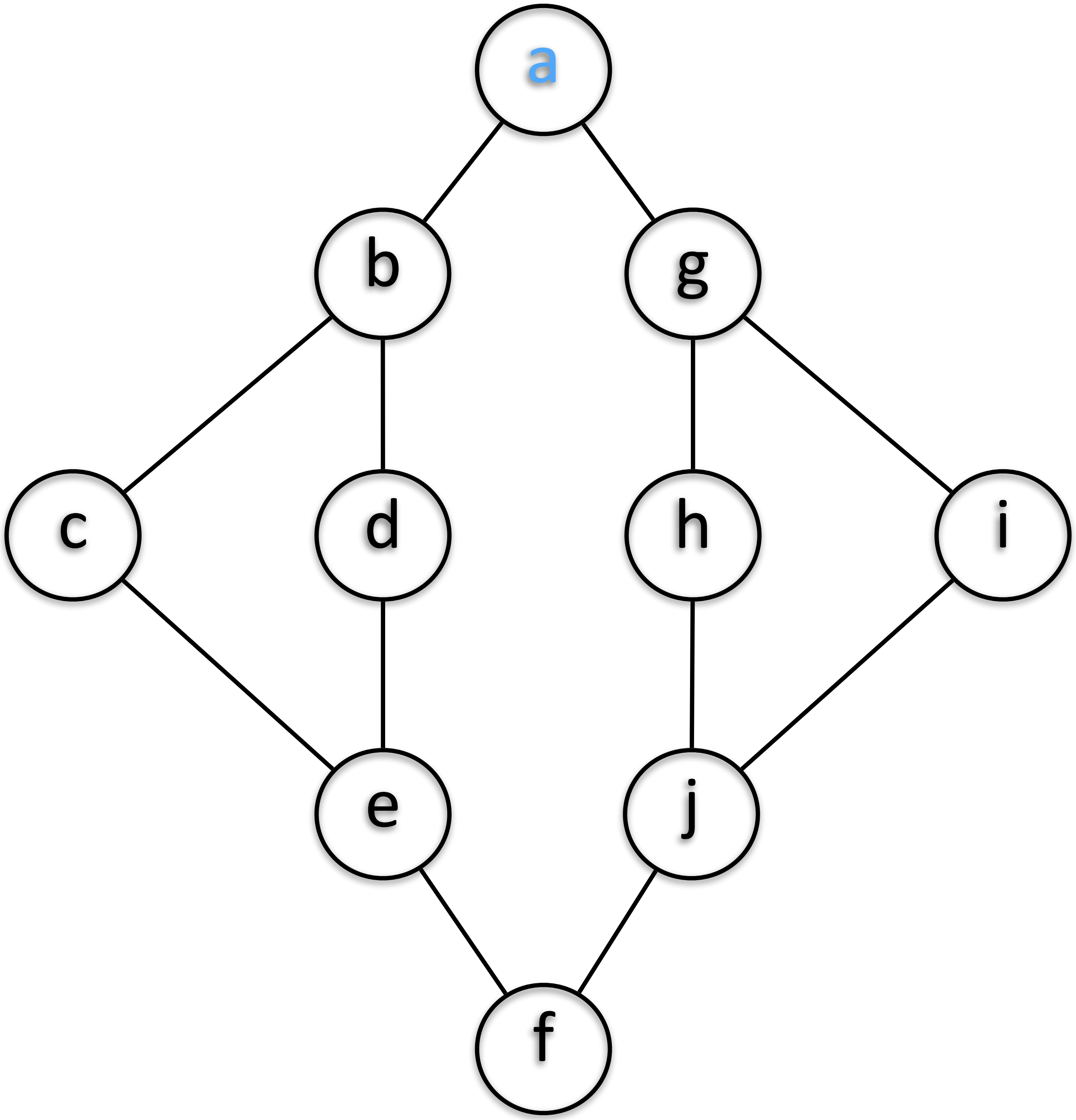
This could be a cost graph for (1+2) * (3+4)



processors

time	1	2
1	a	(idle)
2		
3		
4		
5		
6		

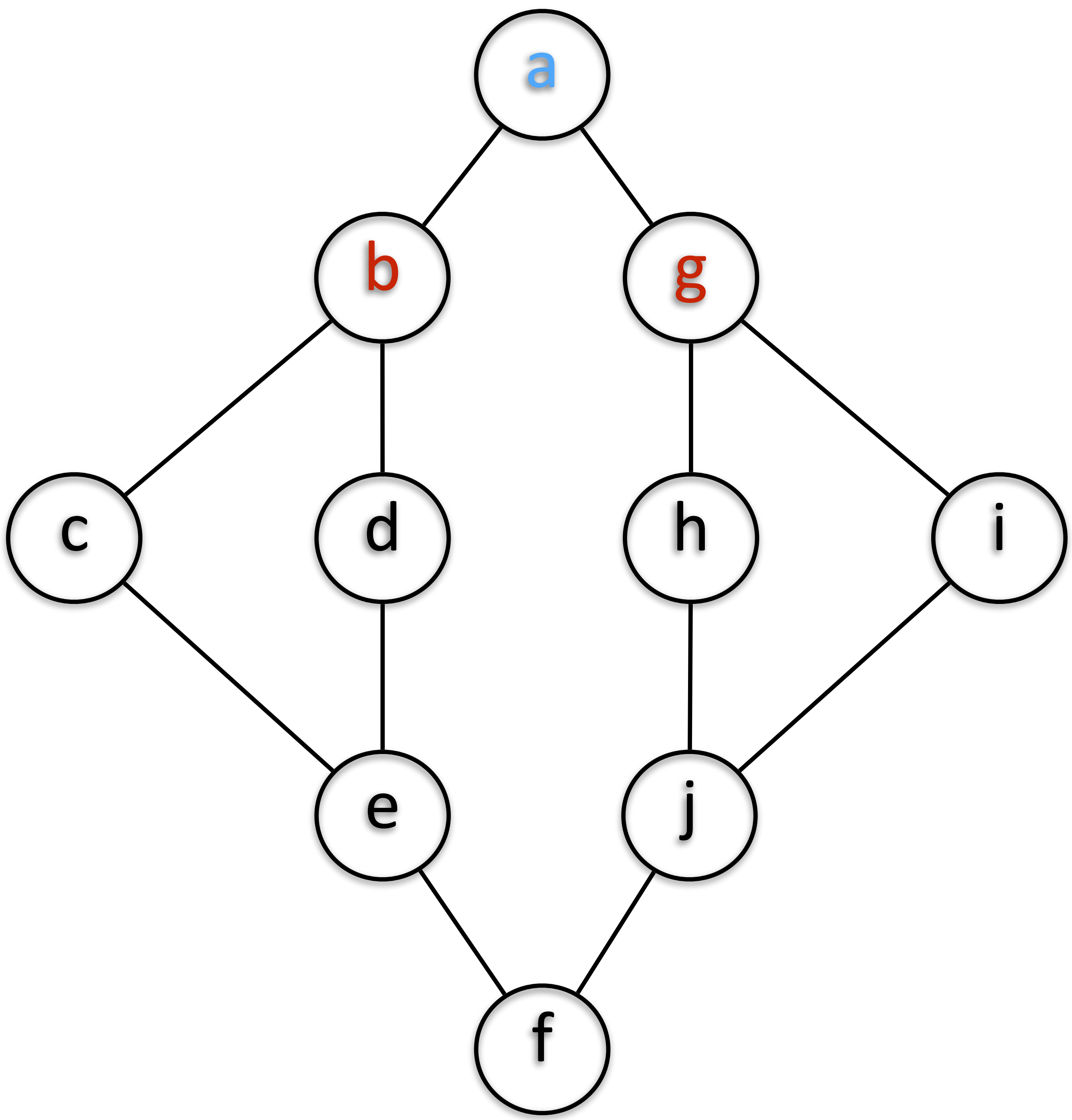
This could be a cost graph for (1+2) * (3+4)



processors

time	1	2
1	a	(idle)
2		
3		
4		
5		
6		

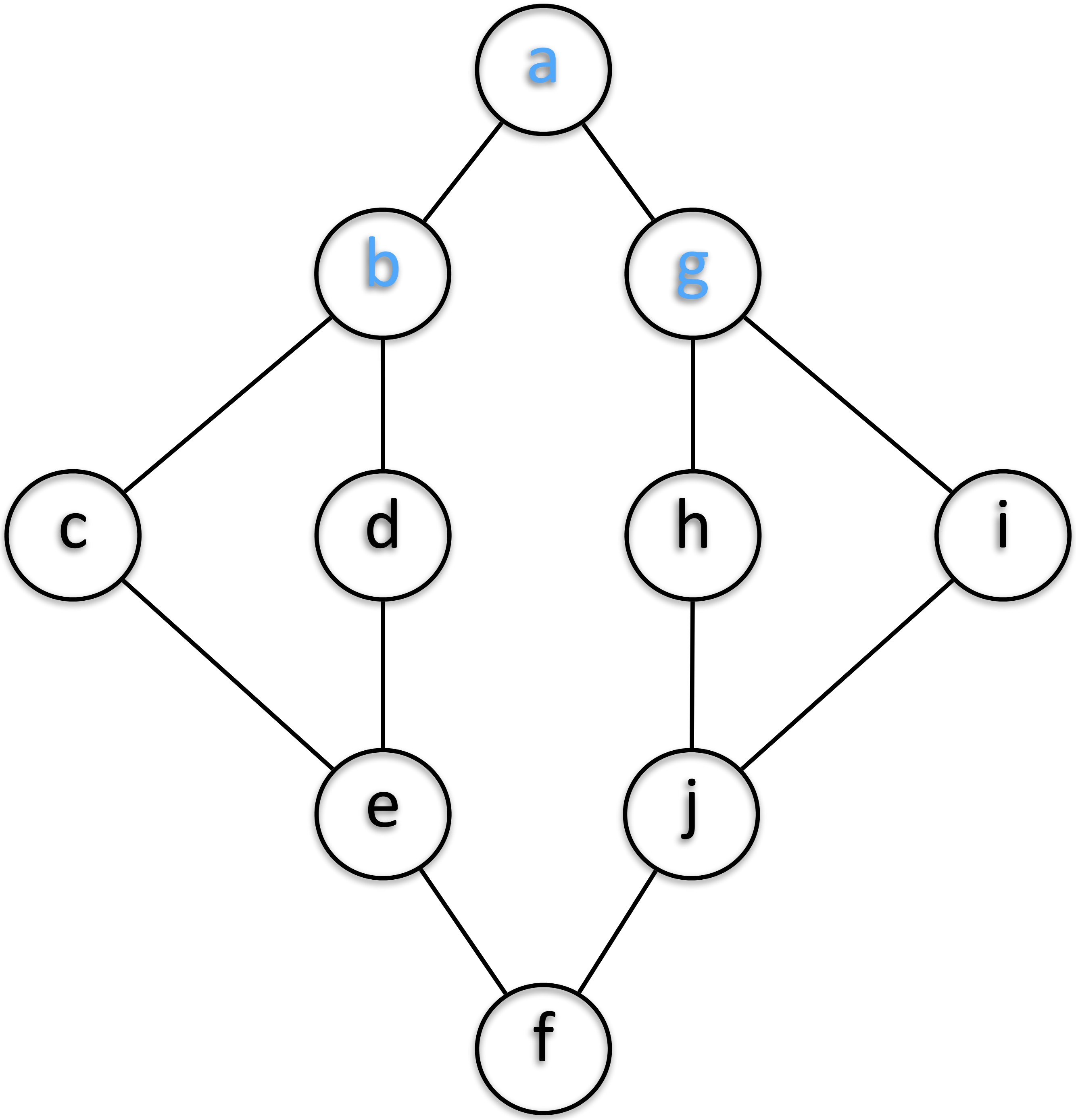
This could be a cost graph for (1+2) * (3+4)



processors

time	1	2
1	a	(idle)
2	b	g
3		
4		
5		
6		

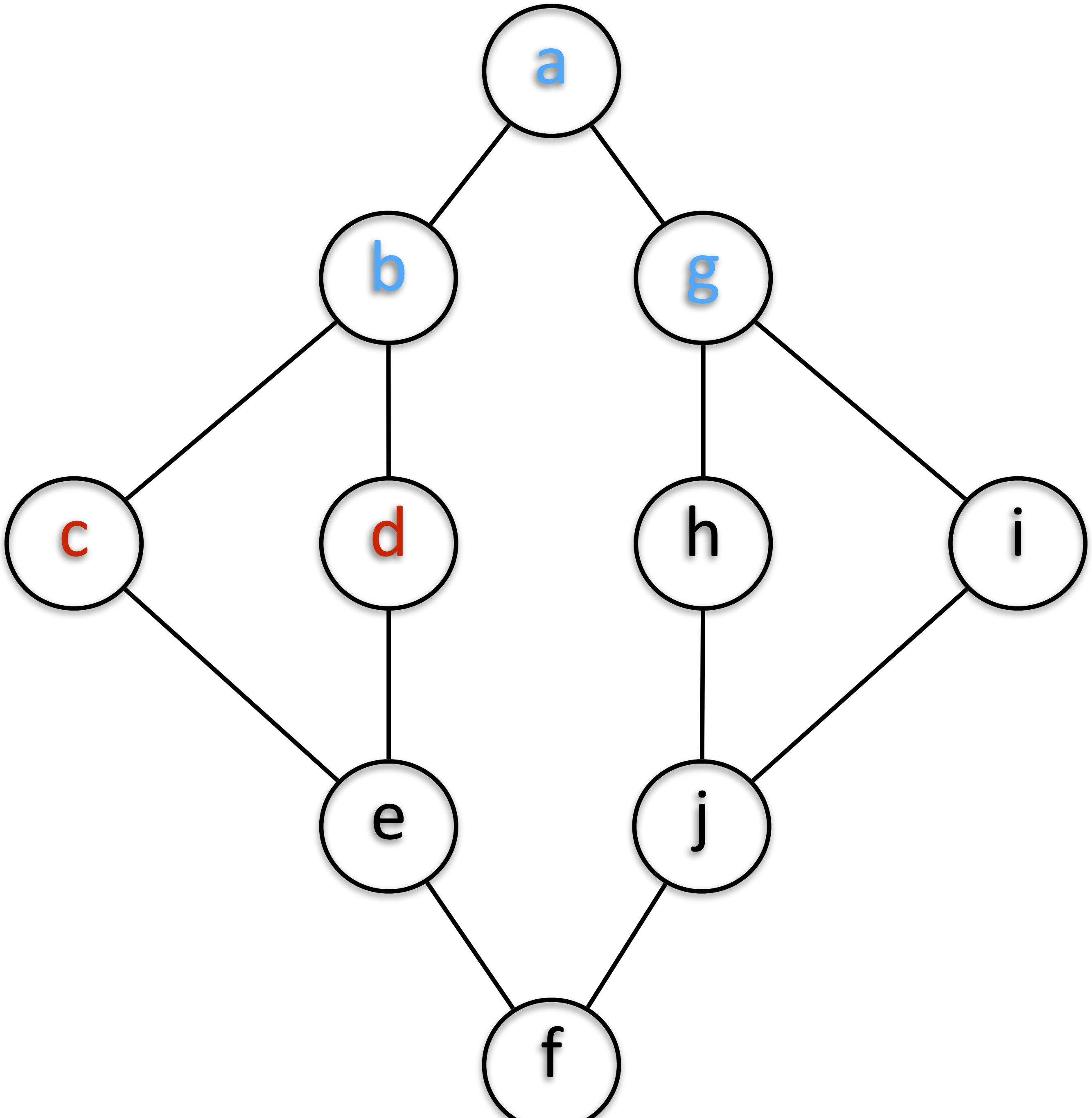
This could be a cost graph for (1+2) * (3+4)



processors

time	1	2
1	a	(idle)
2	b	g
3		
4		
5		
6		

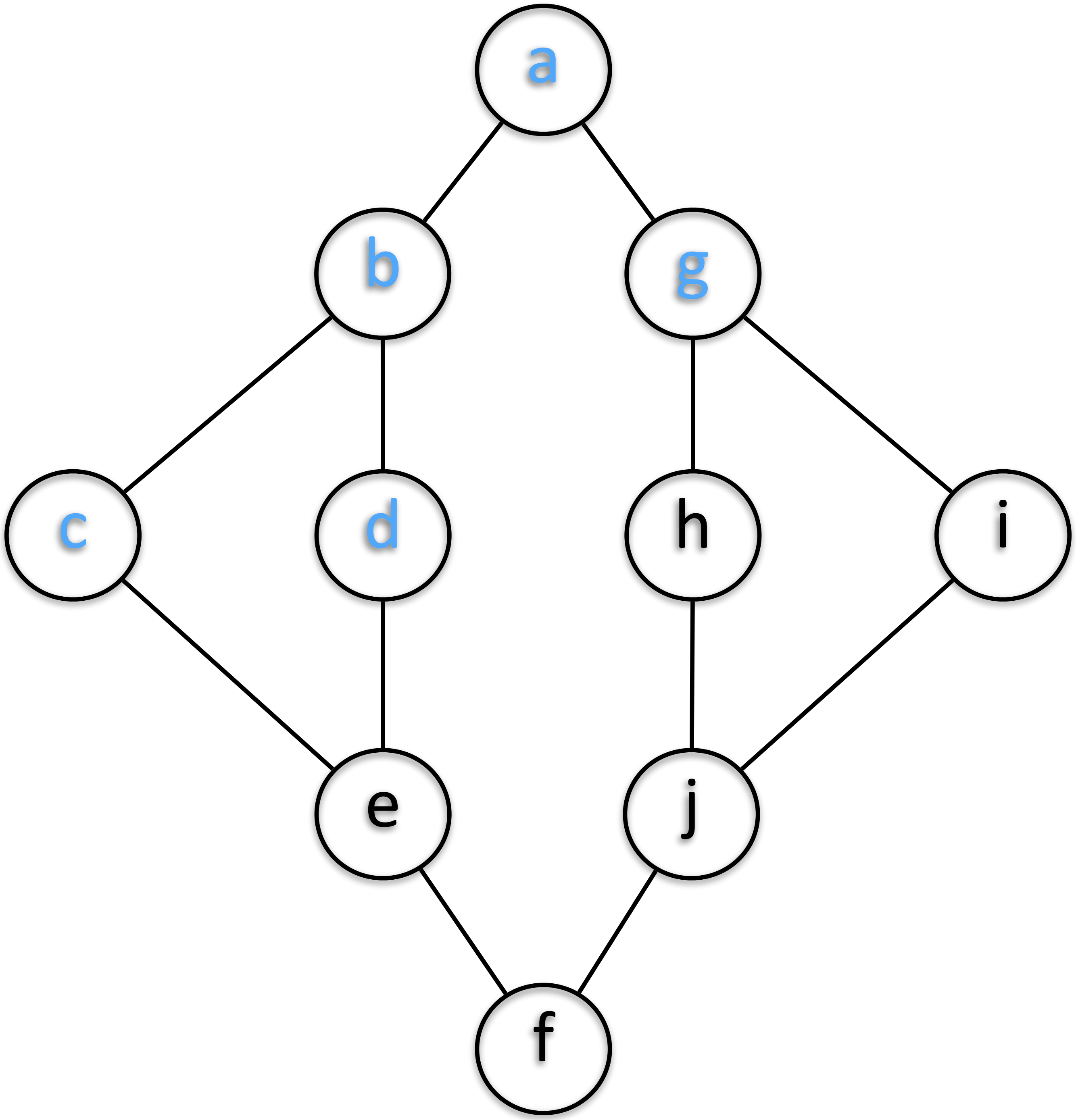
This could be a cost graph for (1+2) * (3+4)



processors

time	1	2
1	a	(idle)
2	b	g
3	c	d
4		
5		
6		

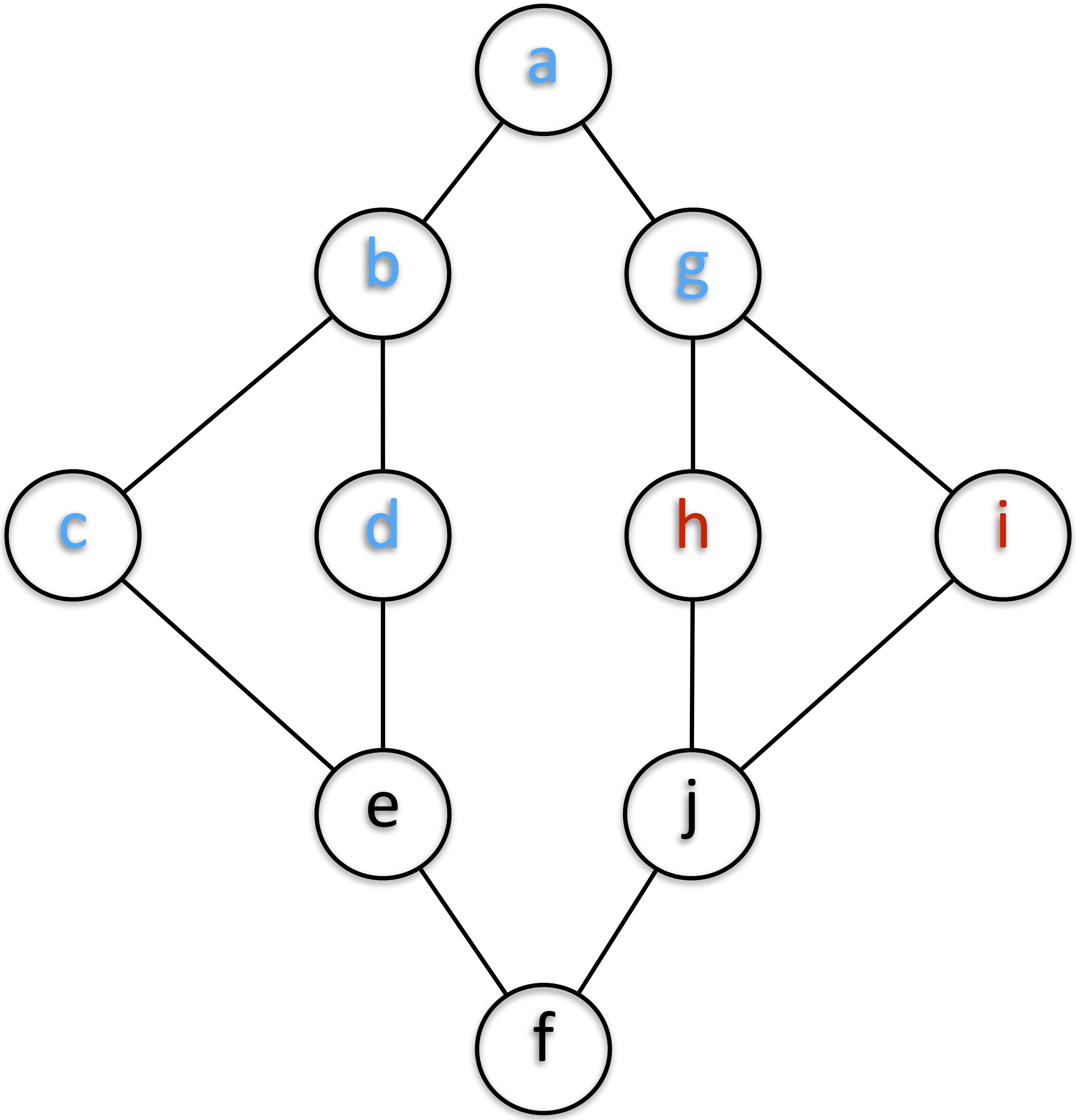
This could be a cost graph for (1+2) * (3+4)



processors

time	1	2
1	a	(idle)
2	b	g
3	c	d
4		
5		
6		

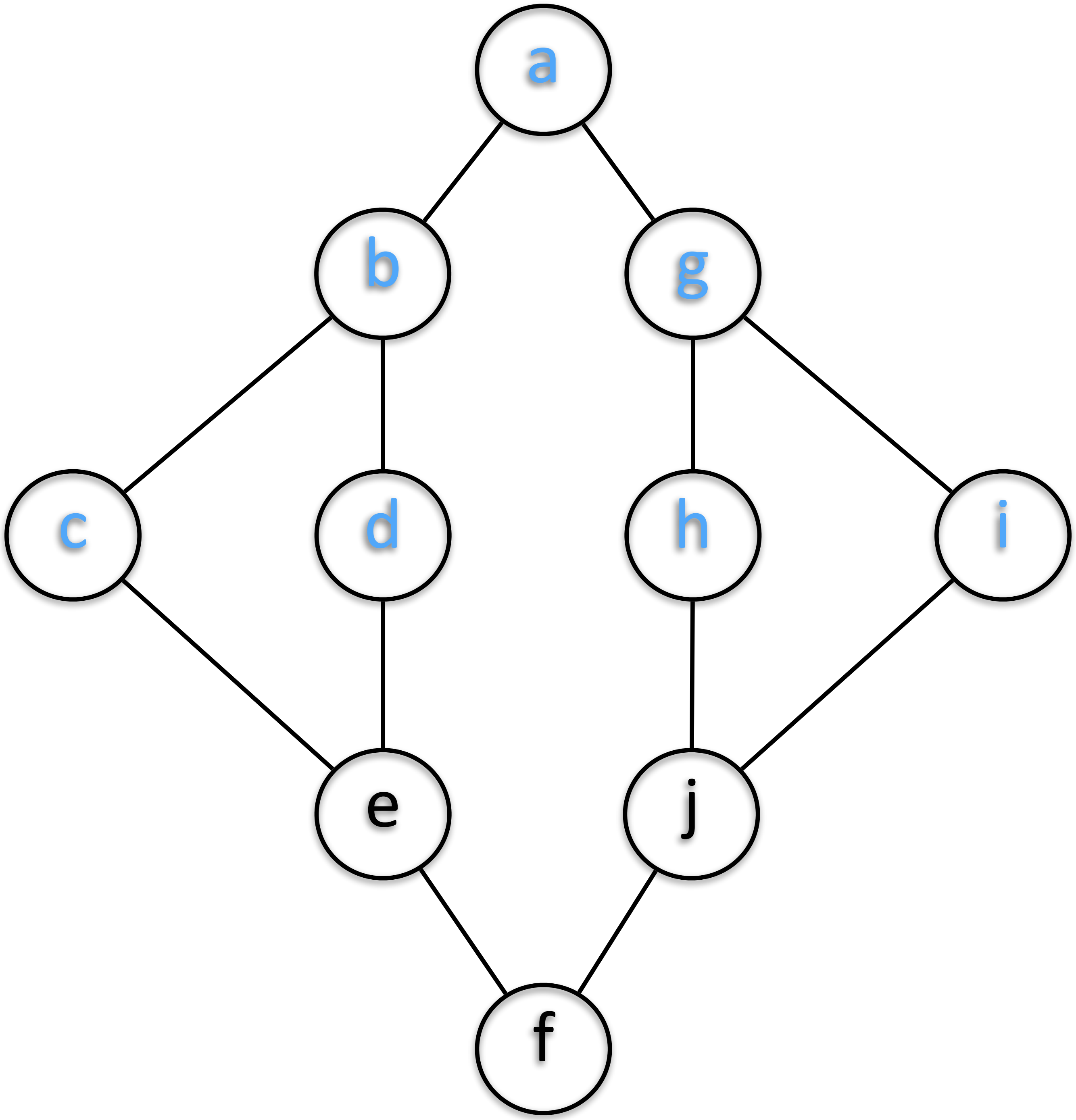
This could be a cost graph for (1+2) * (3+4)



processors

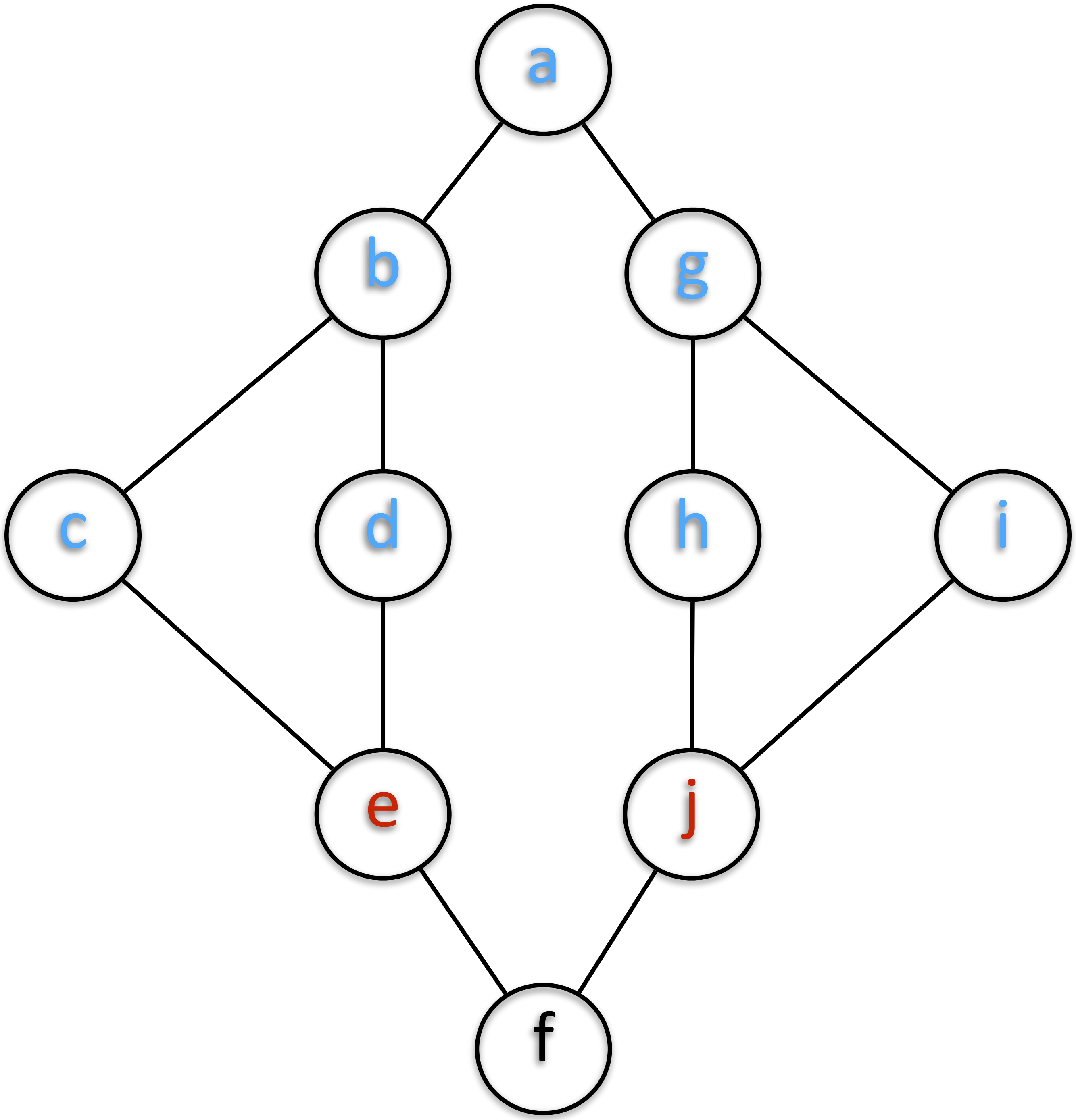
time	1	2
1	a	(idle)
2	b	g
3	c	d
4	h	i
5		
6		

This could be a cost graph for (1+2) * (3+4)



time	processors 1	processors 2
1	a	(idle)
2	b	g
3	c	d
4	h	i
5		
6		

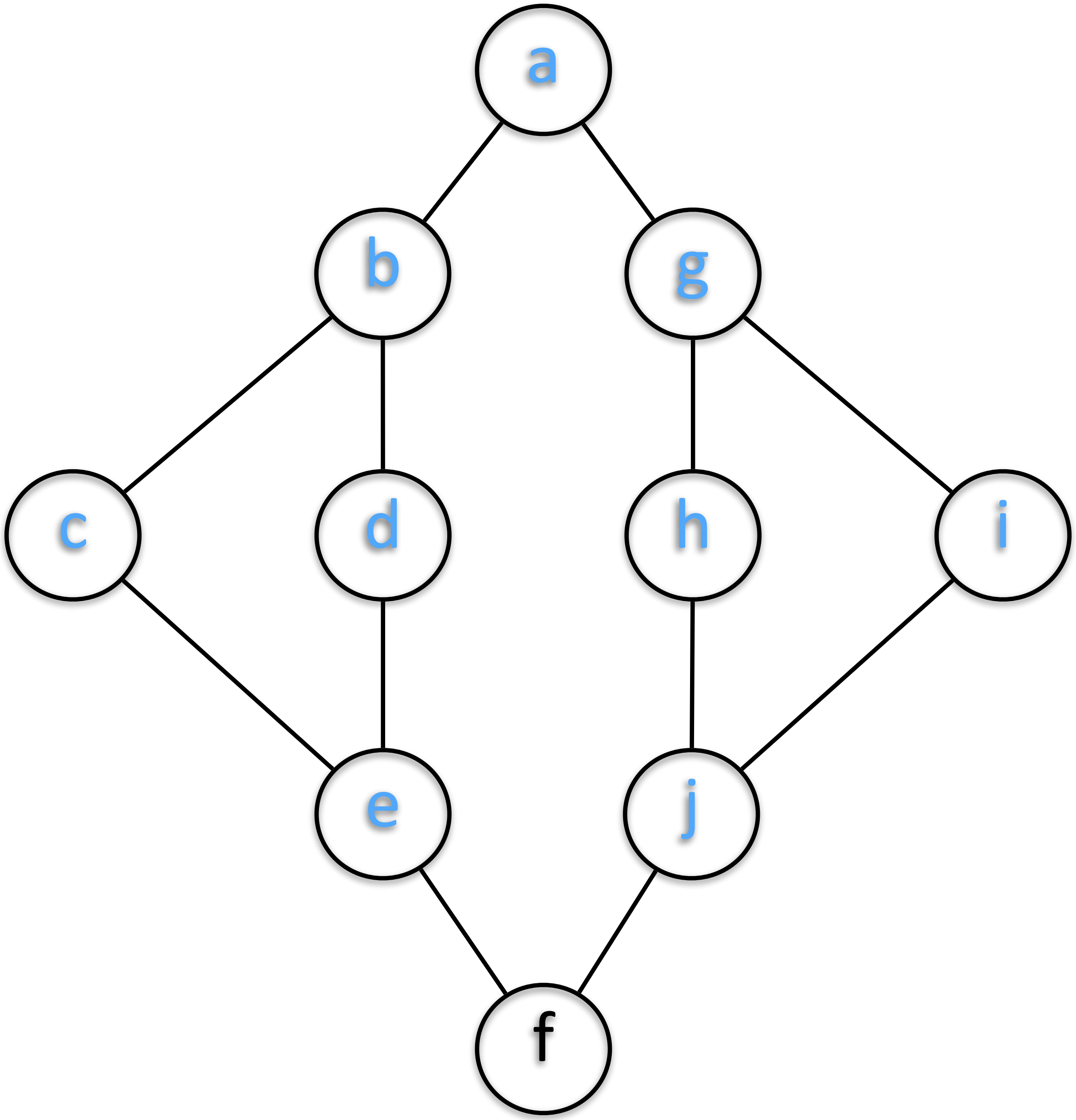
This could be a cost graph for (1+2) * (3+4)



processors

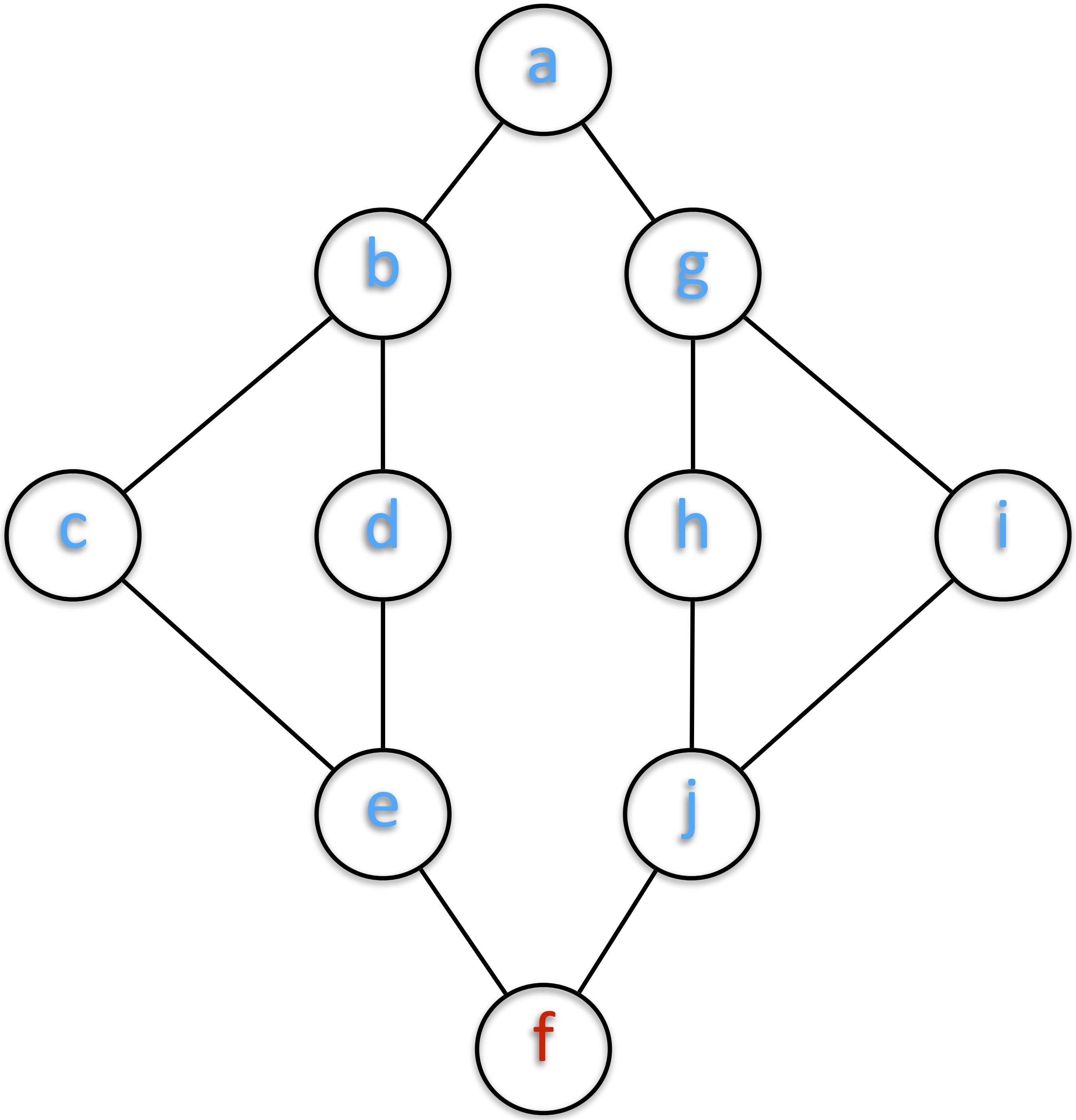
time	1	2
1	a	(idle)
2	b	g
3	c	d
4	h	i
5	e	J
6		

This could be a cost graph for (1+2) * (3+4)



time	processors 1	processors 2
1	a	(idle)
2	b	g
3	c	d
4	h	i
5	e	j
6		

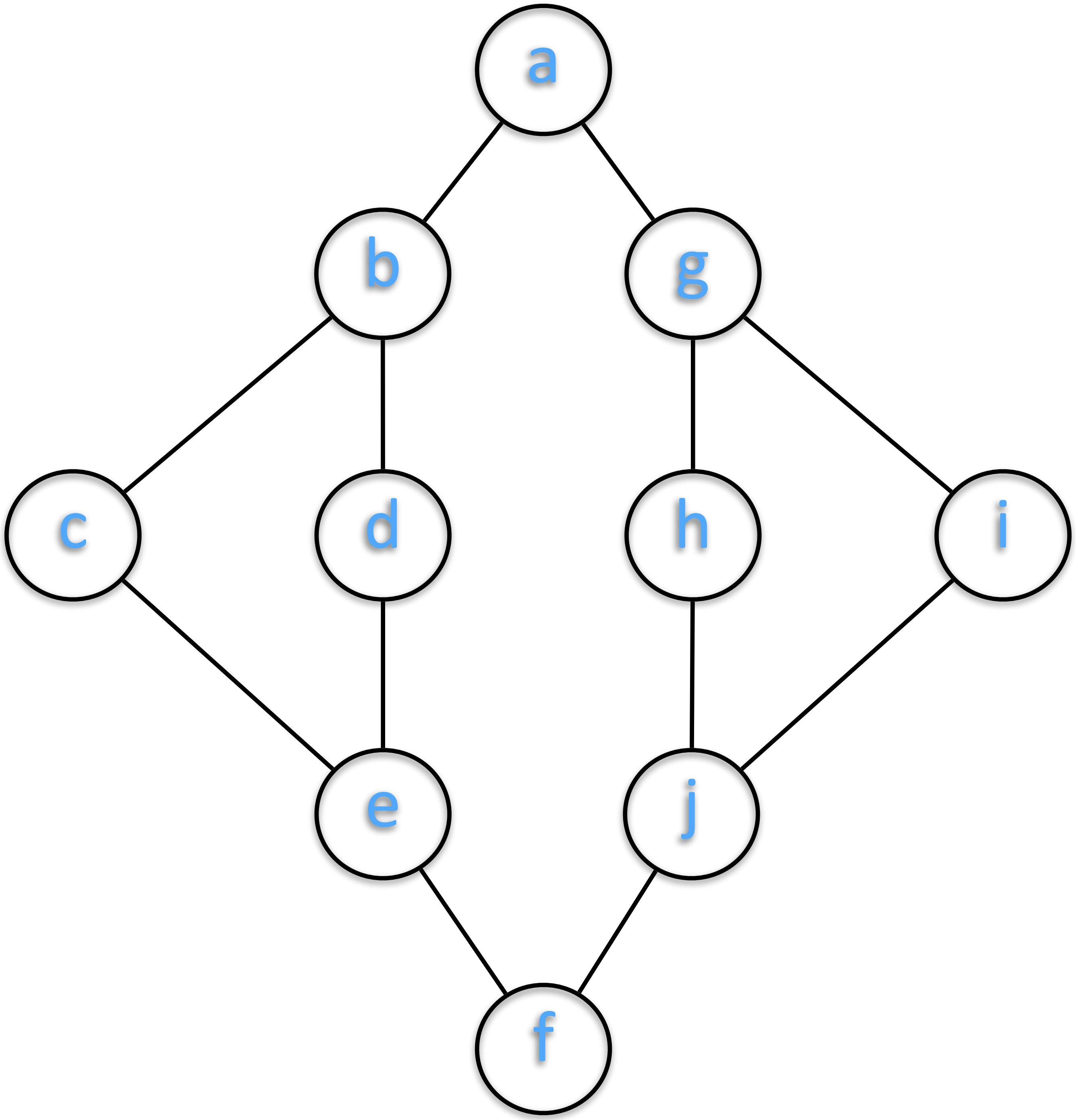
This could be a cost graph for (1+2) * (3+4)



time	processors 1	processors 2
1	a	(idle)
2	b	g
3	c	d
4	h	i
5	e	j
6	(idle)	f

work = 10

span = 5



processors

time	1	2
1	a	(idle)
2	b	g
3	c	d
4	h	i
5	e	j
6	(idle)	f

next

- Exploiting parallelism in ML
- A signature for *parallel collections*
- *Cost analysis* of implementations
- *Cost benefits* of parallel algorithm design

sequences

```
signature SEQ =
sig
  type 'a seq  (* abstract *)
  exception Range of string
  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 seq

is a sequence of values of type t

- We use math notation like

$\langle v_0, \ldots, v_{n-1} \rangle$

$\langle\ \rangle$

for sequence values

$\langle 1, 2, 4, 8 \rangle$ is a value of type int seq

Reminder:
A client would
write t Seq.seq

equivalence

- Two sequence values are *extensionally equivalent* iff they have the same length and have extensionally equivalent items at all positions

$$\langle v_0, \ldots, v_{n-1} \rangle \cong \langle u_0, \ldots, u_{m-1} \rangle$$

if and only if

$n = m$ and for all i, $v_i \cong u_i$

operations

For our given structure Seq : SEQ, we specify

- the (extensional) *behavior*
- the *cost semantics*

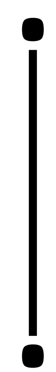
of each operation

Other implementations of SEQ may achieve *different* work and span profiles

Learn to choose wisely!

empty () returns ⟨⟩

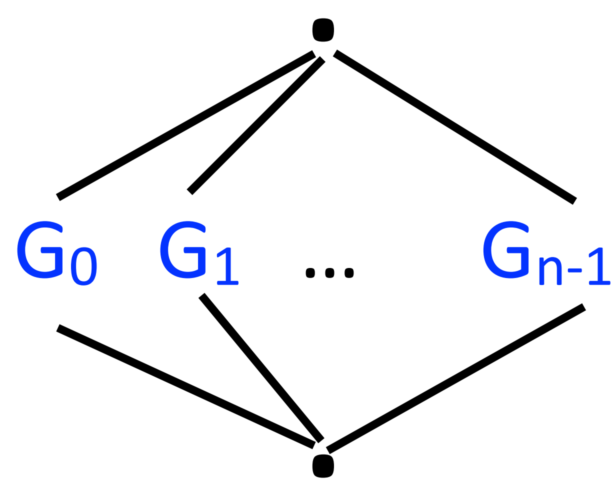
- Type can be t seq for any type t
- Cost graph



work and span O(1)

$$\text{tabulate f n} \cong \langle f\ 0, \ldots, f(n\text{-}1)\rangle$$

- If G_i is cost graph for f(i), the cost graph for tabulate f n is



If f is O(1), the work for tabulate f n is O(n)

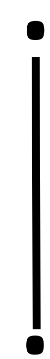
If f is O(1), the span for tabulate f n is O(1) tabulate f n ≅ ⟨f 0, ..., f(n-1)⟩

examples

- tabulate (**fn** x:int => x) 6 ⟨0, 1, 2, 3, 4, 5⟩
- tabulate (**fn** x:int => x*x) 6 ⟨0, 1, 4, 9, 16, 25⟩

$$\text{nth } \langle v_0, \ldots, v_{n-1} \rangle \ i \cong v_i \qquad \text{if } 0 \le i < n$$
$$\cong \textbf{raise} \text{ Range} \qquad \text{otherwise}$$

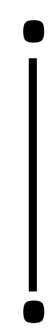
- Work is O(1)
- Span is O(1)
- Cost graph is



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

length $\langle v_0, \ldots, v_{n-1} \rangle \cong n$

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

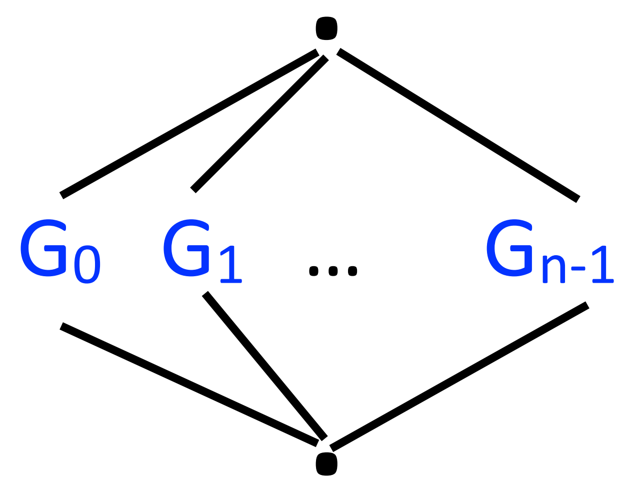


Contrast: List.length $[v_0,\ldots,v_{n-1}] \cong n$

work, span O(n)

$$\text{map } f \ \langle v_0, \ldots, v_{n-1} \rangle \cong \langle f\ v_0, \ldots, f\ v_{n-1} \rangle$$

$\text{map } f \ \langle v_0, \ldots, v_{n-1} \rangle$ has cost graph



where each G_i is cost graph for $f\ v_i$

- If f is constant time, $\text{map } f \ \langle v_0, \ldots, v_{n-1} \rangle$ has work O(n), span O(1)

(contrast with List.map)

reduce

reduce is used to *combine* a *sequence*

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

Compare it with

reduce

('a * 'a -> 'a) -> 'a ->'a seq -> 'a

$\text{reduce } g\ z\ \langle v_0, \ldots, v_{n-1}\rangle \cong v_0 \odot v_1 \ldots \odot v_{n-1} \odot z$

where g is an *associative* function with a base value z

where we represent g with the infix operator $\odot$

- g : t * t -> t is ***associative*** iff for all x_1,x_2,x_3:t

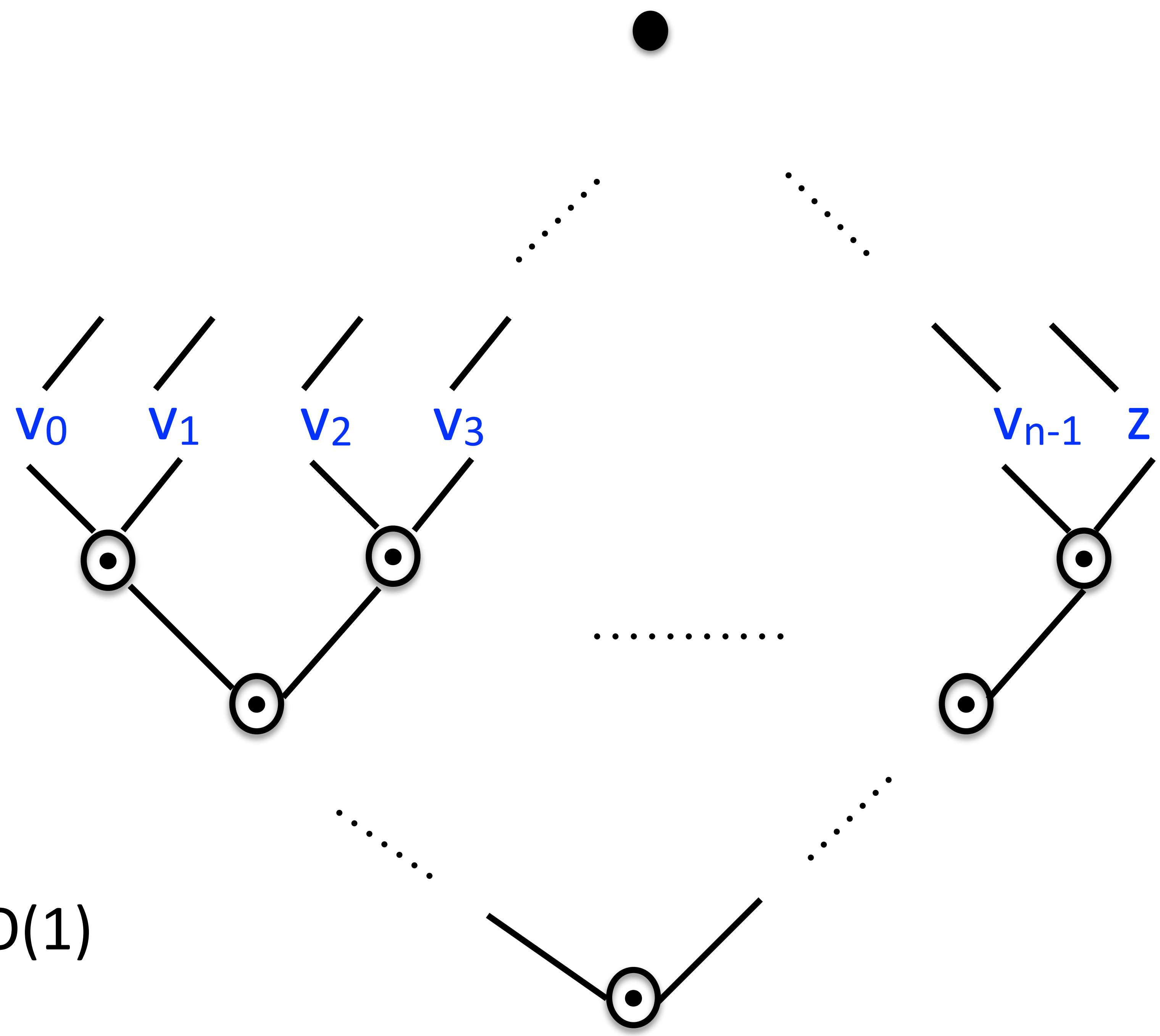
  $$g(x_1, g(x_2, x_3)) = g(g(x_1, x_2), x_3)$$

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

$\text{reduce } g\ z\ \langle v_0, \ldots, v_{n-1}\rangle \cong v_0 \odot v_1 \ldots \odot v_{n-1}$

$\text{reduce } g\ z\ \langle\rangle \cong z$

reduce g z $\langle v_0, \ldots, v_{n-1}\rangle \cong v_0 \odot v_1 \ldots \odot v_{n-1} \odot z$



assuming g is O(1)

work is O(n)

span is O(log n)

$$\text{mapreduce } f\ z\ g\ \langle v_0, \ldots, v_n\rangle \cong (f\ v_0) \odot \cdots \odot (f\ v_{n-1}) \odot z$$

assuming f and g are O(1)

has *work* O(n)

and *span* O(log n)

filter p s ≅ s'

with s' a sequence consisting of all x_i in s such that $p(x_i)$ ≅ true. The order of retained elements in s' is the same as in s

Assuming p is O(1), has work O(n)

and span O(log n)

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

```
val singleton : 'a -> 'a seq   (* gives a single element
                                  sequence *)
val append : 'a seq  * 'a seq -> 'a seq

fun filter (p: 'a -> bool) : 'a seq -> 'a seq =
      let val nothing  = empty ()
          fun keep x = if p (x) then singleton x
                                else nothing
      in
        mapreduce keep nothing append
      end
```

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

```
val singleton : 'a -> 'a seq   (* gives a single element
                                   sequence *)
val append : 'a seq  * 'a seq -> 'a seq

fun filter (p: 'a -> bool) : 'a seq -> 'a seq =
      let val nothing  = empty ()
          fun keep x = if p (x) then singleton x
                                else nothing
      in
         mapreduce keep nothing append
      end
```

S(n) = O(log n), W(n) = O(n log n) assuming append has span O(1)

Example: count

fun sum (s : int Seq.seq) : int = ____________________

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

fun count (class: room) : int = sum ____________________

Example: count

using map

```
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 = sum  ____________________
```

Example: count

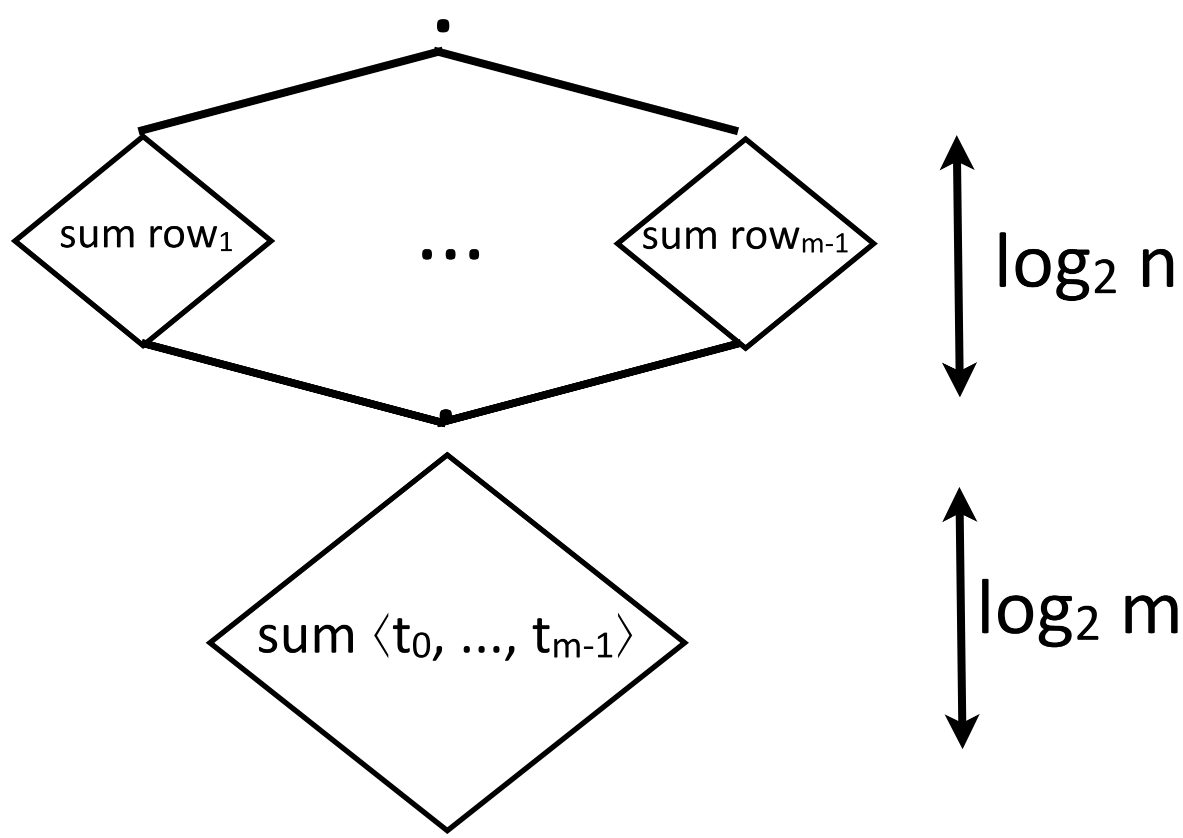
```
fun sum (s : int Seq.seq) : int =  Seq.reduce (op +) 0 s

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

fun count (class: room) : int = sum  (Seq.map sum class)
```

analysis

Let t_i = sum row_i

m rows of length n each

count s = sum $\langle t_0, \ldots, t_{m-1} \rangle$



cost graph of sum (map sum s)

work is O(mn)

span is O(log n+ log m)

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

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
```