Boruvka's Algorithm

Boruvka's algorithm for the MST is this:

```
assumption:  all edge lengths are different
C is a set of connected components, initially {{1 },{2},...,{|V|}}
X is the set of tree edges, initially empty
repeat
for each component c in C, find the shortest edge e leaving c, and add it to X
find the connected components of X
until |C| = 1 (there is only one component)
```

It takes linear time to do each iteration. Finding the connected components requires of course linear time, but how do you find all shortest edges out of all components in linear time?

Here is how: You process all edges one by one, and look up the component of each endpoint of the edge. If they are the same, nothing to do. If they are different, you update a number you keep for each connected component which is the length of the shortest edge out of the component that you have seen so far.

Importantly, there are at most $\log |V|$ iterations: At each iteration the number of components $|C|$ is divided by two at least (can you see why?). So, this is an $O(|E| \log |V|$ algorithm, very much like the others.

But among the three MST algorithms we have seen, this is the only one that we can hope to parallelize. All others are *very* sequential (imagine a large graph whose MST is a looooong path!).

Parallel Algorithms

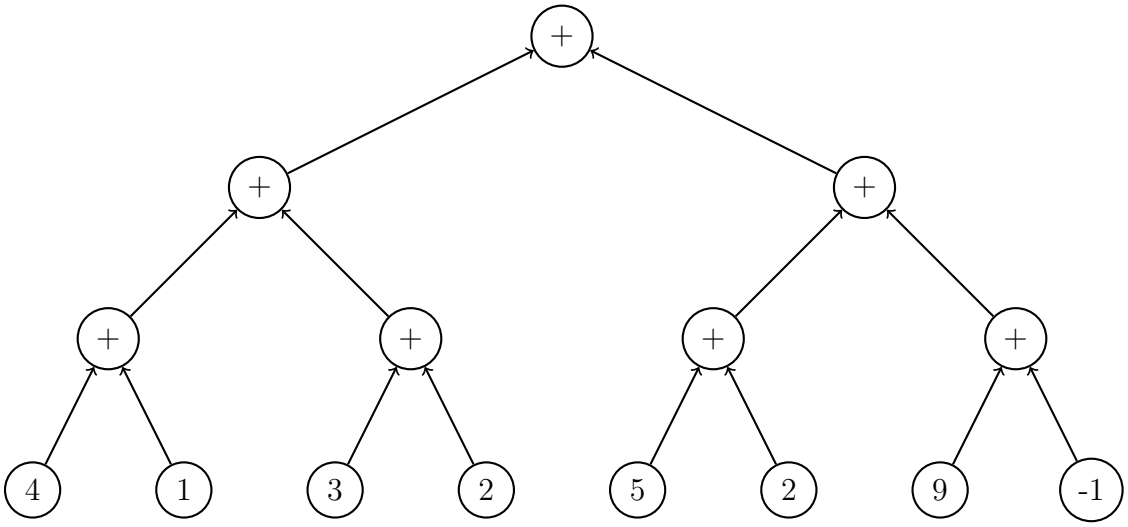
Designing parallel algorithms requires a very different mindset than designing sequential algorithms (as in the rest of CS170). In this lecture we'll get a taste of this mindset.

Let us start with the simplest problem: Add n numbers together. Sequentially, this is trivial:

```
sum = 0, for i = 0 to n-1 do sum = sum + a[i]
```

It takes $O(n)$ time, and obviously it cannot be done better.

But how about in parallel? The best way to add n numbers in parallel, say the eight numbers $4, 1, 3, 2, 5, 2, 9$,and -1, is to build a *circuit* of adders, shaped as a binary tree:



But now we can view this as a parallel algorithm! The input numbers initially reside in n processors (bottom layer), call them processor 0, 1, 2, etc. Then $n/2$ of these processors (the even-numbered ones) take the initiative and add their number to the one in the processor following them. And so on. The output is in processor 0.

We are assuming here a very simple model of parallel computation. We have many processors operating *synchronously*, that is, in complete unison. The processors have local memory, but they can also access each other's memory in one step.

Some sticky questions: Can two or more processors read the same memory cell at the same step? How about write it? And what happens if the written values are different? There are many possible models for this. Here we will simplify matters by assuming that the answers are "Yes," "Yes," and "Any one of the values prevails." This is called the CRACOW model of parallel computation (for concurrent read, arbitrary concurrent write).

But is this a reasonable model? The answer is an unqualified "no." Many parallel machines are asynchronous (and synchronization of threads is tedious). Our model ignores communication between processors: the processors in real parallel machines or systems are connected through some interconnection network, and communication is costly. Also, simulating the CRACOW model on a real machine is not easy, and may take many steps for each write operation, because most computer memory does not work this way.

Still, developing algorithms for this model is useful. By understanding how to solve problems in parallel in this idealized model reveals the intricacies of parallel algorithms, and creates algorithmic ideas which can then be mapped to actual parallel machines or systems.

We can now write pseudocode for the addition algorithm.

```
n processors named i = 0, 1, ...  , n-1, holding the inputs in the field value
for each i pardo:  (this means, do in parallel with tight synchrony)
offset = 1
repeat
   j = i + offset
```

```
   if j <= n value = value + j.value (j.value means value of remote processor
j)
   offset = offset + offset (double the offset to go up in the tree)
   *if offset does not divide i retire (i will have no work to do from now on)
until offset >= n
parend
output is 0.value
```

"Retire" means that we don't use the processor any more. Notice how the *-ed instruction ensures that the tree shrinks in size as time proceeds (as we go up), and so the total "work" done by the algorithm is kept linear, as in the binary tree of adders.

In a sequential algorithm we ask "How much time does it take as a function of n?" In parallel algorithms there are three questions to ask:

- How many processors does it use?
- How much parallel steps does it make?
- How much *work* does it do?

The work of a parallel algorithm is the total number of steps executed by all processors at all times. In a circuit, the number of processors is the *width* of the circuit (the size of its largest "row"); the number of parallel steps (or parallel time) is the *depth* of the circuit (the number of rows, or the longest upward path), and the work is the total number of gates (call it the "size" or "volume") of the circuit.

In our addition example, there are n processors, $\log n$ parallel steps, and work n (omitting $O(\cdot)$'s). We may write $P = n, T = \log n, W = n$. If we had omitted the * line of the pseudocode, we would still have a correct algorithm with the same P and T, except that W would be $n \log n$.

Can it be done better? Work cannot be improved, obviously, but how about parallel time? Can we add n numbers in parallel faster than $\log n$?

This is impossible, and it's not hard to see why. Consider one of the processors. In the beginning, at time zero, it is only aware of its own existence, for all it knows there is no other processor in the universe. Then it takes a step. t becomes 1. During this step it may access the memory of another processor, and so it becomes aware of the existence of that processor. In the next step, it may access yet another, and now this other processor may know about itself *and* somebody else. Or another processor may write a piece of information on its memory (even if many processors try, by CR**A**COW only one will succeed), and so our processor now may know what *that* processor knew. The point is that the number of processors, including itself, of which our processor is aware —"has heard from" — at most doubles at each step. After t steps, it is at most 2^t. If $t < \log n$, then no processor has not heard from all inputs, and thus no processor cannot possibly know and output the sum. The same argument establishes that $\log n$ parallel time is required for most tasks — just as time $O(n)$ is required for all reasonable sequential tasks.

For another application of the same idea, if instead of ``value = value + j.value'' we write ``value = min (value, j.value),'' then the head of the list ends up holding the smallest value. This is how you find min and max of n numbers in $\log n$ parallel steps.

Question: How about the FFT of n numbers, how can it be implemented in parallel?

Well, the FFT *is* a circuit (remember Figure 2.10). By inspection, we see that $P = n, T = \log n, W = n \log n$ (this is the width, depth, and volume of the circuit, respectively).

And how about matrix multiplication? It can be done in parallel time $\log n$ and work n^3 on n^3 processors.

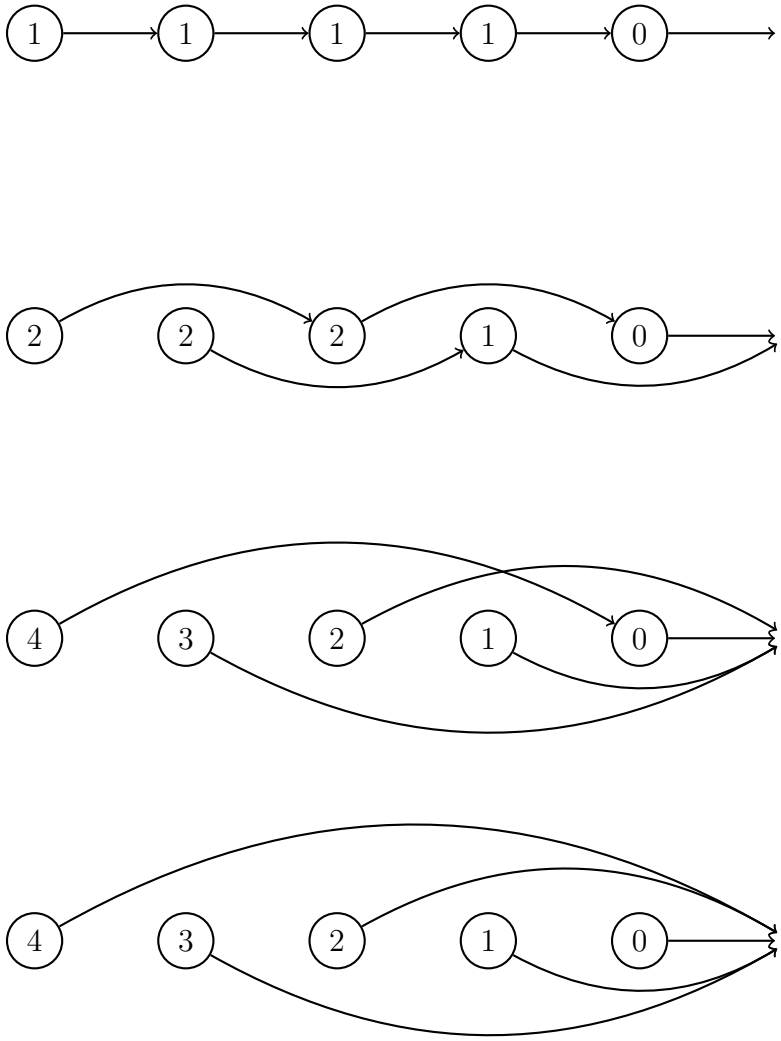
List ranking

Suppose that you have a linked list, and you want to compute the *rank* of each cell, the length of the path from it to nil. This is easy to do, of course, except the obvious algorithms are sequential. In fact, this problem feels "inherently sequential." How on earth can you telescope the rank of a node in a long list?

Here is a clever parallel algorithm that does this. The algorithm assumes that every cell sits on a different processor, with local variables x and next. The local memory of other processors can be accessed through pointers.

```
for each cell pardo:  (this means, do in parallel with tight synchrony)
if next = nil x = 0 else x = 1
repeat
   x = x + next.x
   next = next.next
until next = nil
parend
```

Example: Here are the steps for a list of five cells.



Notice how, by the method of "pointer-jumping" and "path-doubling", this algorithm defeats any long list in only $\log n$ steps.

But the same idea does much more: Suppose your cells hold an initial value, call it x, and you want to compute, for each cell, the sum of the values over the whole path to nil. This is done by the same algorithm! Just omit the first line, the one that initializes x. (Check it.) So, this algorithm is another $n \log n$ work way of adding numbers.

List ranking is one of the most useful parallel algorithms, often compared to depth-first search and breadth-first search combined...

Brent's Rule

In sequential algorithms, we account for the time required to solve an instance of size n, as a function of n. In a parallel algorithm, we gauge the parallel time, the work and the number of processors employed, again as a function of n.

Number of processors as a function of n? This seems very strange. In real life, we are going to have a fixed parallel machine, or farm of machines, whose number of parallel

processors is known, and cannot be adjusted for each instance size.

There is a simple observation in this regard, known as *Brent's Rule* [1] which tells us this: Any parallel algorithm that solves a problem utilizing many processors, can be scaled back easily to fewer processors.

Brent's Rule: If a problem can be solved in parallel time T and work W with P processors, then it can also be solved with $p < P$ processors in the same work and time $T + \frac{W}{p}$.

The proof proceeds with simulating every time step t of the processor-hogging algorithm, in which w_t processors were active, with $\lceil \frac{w_t}{p} \rceil$ steps using p processors, and then noticing that $\lceil \frac{w_t}{p} \rceil \leq \frac{w_t}{p} + 1$ and adding over the T steps (obviously, $\sum_t w_t = W$).

This means that in parallel algorithms we should try to saturate the problem with processors, even if we know that we will never have so many.

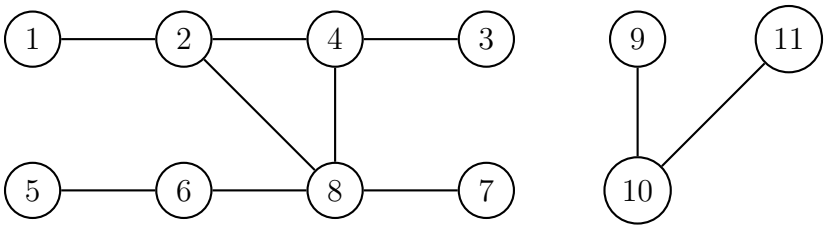
Connected components in parallel

Some problems are very easy to solve fast in parallel, because their sequential algorithm is eminently parallelizable. We have seen a few so far. There are other problems which, even though they can be solved very fast sequentially, we know of no algorithms that utilize parallelism in any substantial way; such problems are called *inherently sequential.*

There is an intermediate class of problems, which can be solved very fast in parallel; however, in order to do so we must forget all we know about how to solve these problems sequentially. The sequential algorithms for these problems do not parallelize easily.

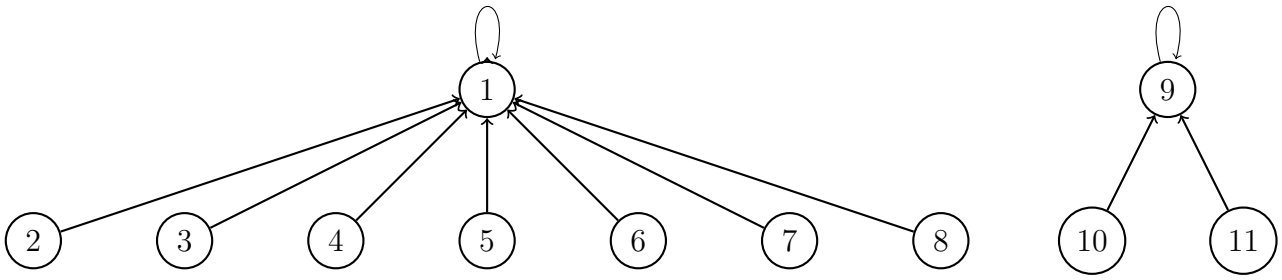
One of these problems is the problem of finding the connected components of a graph. It is very easy to solve sequentially through depth-first search. However, a little reflection will reveal that DFS is a *very* sequential algorithm (just recall the pre and post numbers assigned to the nodes). To find the connected components of a graph in parallel, we must develop a completely new algorithm, which is dramatically different. It is also instructive about the difficulties in writing parallel algorithms.

We are given an undirected graph, say



Our goal is to find the connected components of this graph. That is, we want to compute the graph into the following two "star" structures, in which every node points to the name of its connected component. A *star* is a tree of depth one (see the figure).

[1] Discovered in 1973 by the Australian computer scientist Richard Brent — no connection to http://articles.latimes.com/1997-04-16/entertainment/ca-49080_1_time-limits .



The algorithm uses $|V|+|E|$ processors, one for each vertex i and one for each undirected edge $[i, j]$ of the graph. The basic idea is this: Each vertex has a pointer P, initially pointing to itself. We use the pointers to join the vertices in trees, then join these trees if they belong to the same component, and "shorten" the trees by pointer jumping to make them stars. This has to be done with *extreme* care so there are no cycles (on a cycle you can pointer-jump forever...). We assume the vertices are numbers, and we use the numbers to avoid cycles: We try to always join larger number to smaller numbers (note that the representative of each component in the figure above is the smallest-numbered vertex of the component). But more tricks are needed. For example, if we want to join vertices i and j we don't simply make P[i] = j. Instead we "hook" them together:

```
Hook(i,j):  P[P[i]] = P[j]
```

This way, vertex i does not lose its connection to its current parent (typically, i will be a leaf node in a star, and so the whole star will be joined). And furthermore, P[i] is joined not to j but to the parent of j, thus decreasing the depth of the new tree formed.

Crucially, we will need to know when node i belongs to a star. This is done by all vertex processors executing in parallel this procedure, called `test-for-star`:

```
star[i] = true
if P[P[i]] ≠ P[i] then star[i] = false, star[P[P[i]]] = false
if not star[P[i]] star[i] = false
```

Here is the algorithm:

```
1.  For each vertex i pardo P[i] = i parend
for each edge (i,j) pardo[2]
   2.  if i > j Hook(i,j) (here Hook is used for uniformity, P[i]=j would be enough)
   3.  if i is a singleton (that is, a node all by itself --- exercise:  how do
we test this?)  Hook(i,j)
parend
(at this point all vertices are in a tree with at least two vertices, assuming
no isolated nodes.)
repeat
   for all edges i-j pardo
      test-for-star
      4.  if star[i] and P(i) > P(j) Hook(i,j)
```

[2]If there is an edge between u and v, then two processors will consider this edge: one will have $i = u$ and $j = v$ and the other will have $i = v$ and $j = u$.

```
      test-for-star
      5.  if star[i] and P(i) ≠ P(j) Hook(i,j)
   parend
   6.  for all vertices i pardo
      P[i] = P[P[i]] (pointer-jumping)
   parend
until pointer-jumping does nothing in the whole graph
```

The figure below shows the first execution of the steps 1 – 5 of the algorithm. Notice that, for example, in step 2 many edge processors may try to hook a node, but only one will succeed (thanks to CRACOW). The attempts are indicated in the figure.

Here is why this works:

- After the steps 1 – 3, it is clear that the pointers form a forest (many trees, no cycle except for the self-loops on the roots), and there are no singletons (they were forced to Hook in Step 3).

- From then on we execute the repeat loop. We need to show that no cycles are ever introduced (this is our nightmare in this problem). Stars play an important role in the argument. Stars are formed by pointer-jumping in non-star trees in Step 6 (pointer-jumping has no effect in stars). There is no other way to form a new star, because the Hook operations in Steps 4 and 5 join stars to other trees, and the new tree that is formed has depth at least two.

- Step 5 never Hooks i to another star. This is because if j is in a star, that star must have existed before Step 4 by the previous paragraph, and so it would have been Hooked with i at that Step. In addition, Step 4 may Hook two stars, but in a hierarchical manner (from larger roots to smaller roots). It follows that no Hooking in Steps 4 and 5 of an execution of the repeat loop can create a cycle, because Hookings happen from large roots to small roots of stars, and then possibly to a non-stars (Step 5), from which no other Hookings could have happened, and so the cycle cannot close.

- Notice also that if a star survives Steps 4 and 5, then it must be a whole component of the graph (otherwise it would be connected to something else and would have been Hooked). From now on it will be inactive, it has been discovered and waits for the other components to finish.

- Notice also that every active tree has its depth reduced by at least 2/3 at each execution of Step 6. This is because pointer-jumping reduces a tree's height from h to $\lceil h/2 \rceil$, which is at most $2h/3$ if $h > 1$ (which it is, because we are only considering active trees). So, the sum of the heights of all active trees is multiplied by 2/3 each execution of the while loop. So, the algorithm will eventually terminate after $\log n$ iterations.

- So, this algorithm correctly finds the connected components of any graph, and requires $|V| + |E|$ processors, parallel time $\log |V|$, and $(|E| + |V|) \log |V|$ work.

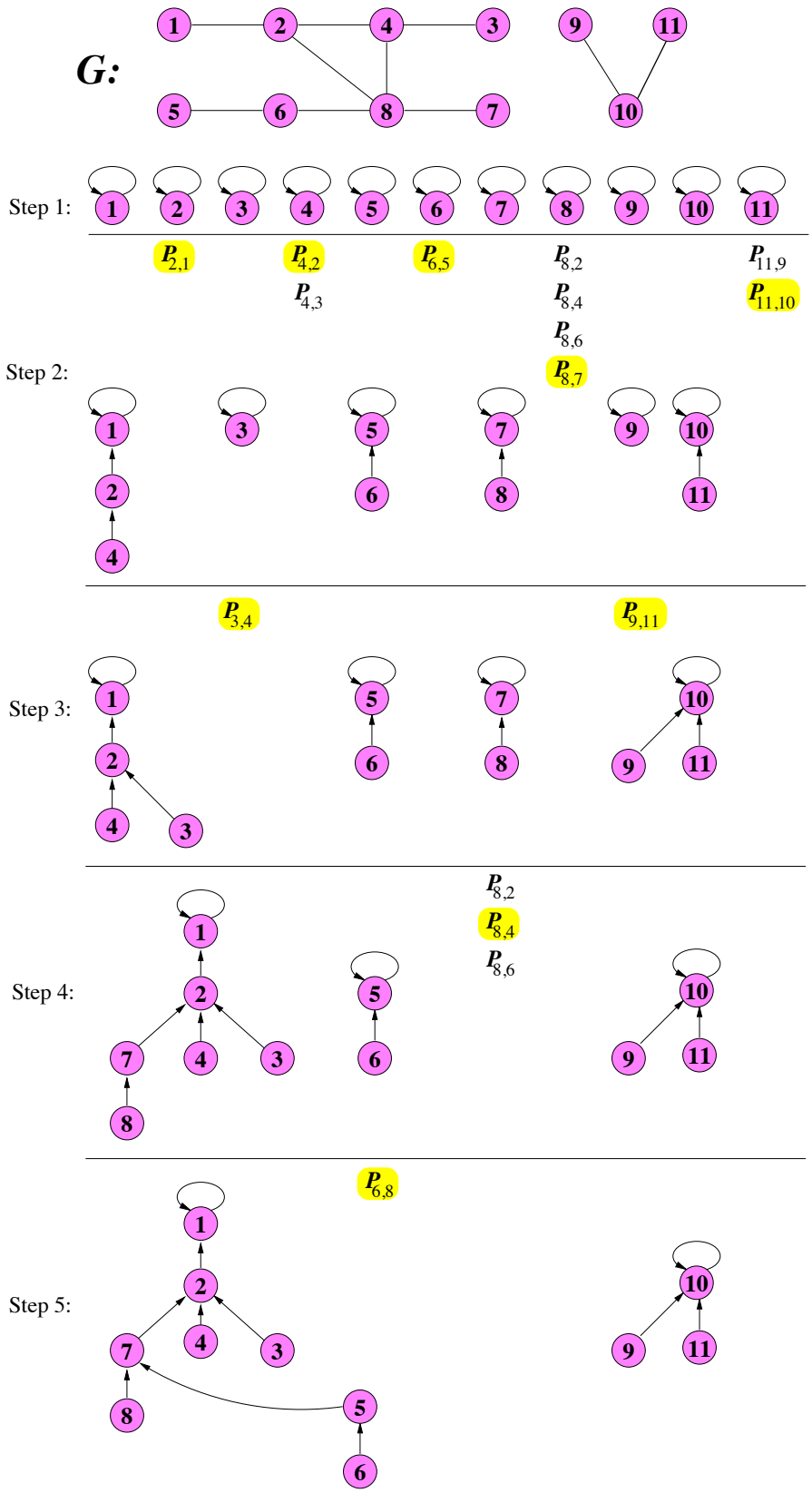


Figure 27.4. The first 5 steps of algorithm **CC**.