Lecture 1

Vertex Coloring

1.1 The problem

Nowadays multi-core computers get more and more processors, and the question is how to handle all this parallelism well. So, here's a basic problem: Consider a doubly linked list that is shared by many processors. It supports insertions and deletions, and there are simple operations like summing the size of the entries that should be done very fast. We decide to organize the data structure as an array of dynamic length, where each array index may or may not hold an entry. Each entry consists of the array indices of the next entry and the previous entry in the list, some basic information about the entry (e.g. its size), and a pointer to the lion's share of the data, which can be anywhere in the memory.

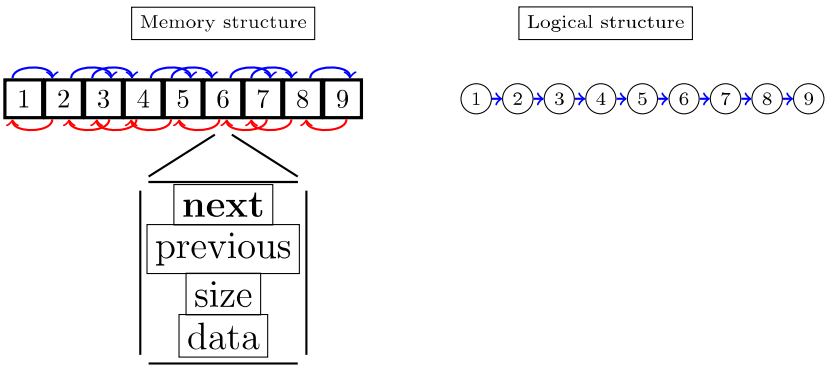

Figure 1.1: Linked listed after initialization. Blue links are forward pointers, red links backward pointers (these will be omitted from now on).

On the other hand, *any* set of concurrent modifications that does *not* involve neighbors is fine: the result will be neat doubly linked list. Clearly, we want to be able to manipulate arbitrary list entries. This can be rephrased as an infamous graph problem.

Problem 1.1 (Vertex Coloring)**.** *Given an undirected graph $G = (V, E)$, assign a color c_u to each vertex $u \in V$ such that the following holds: $e = (v, w) \in E \Rightarrow c_v \neq c_w$.*

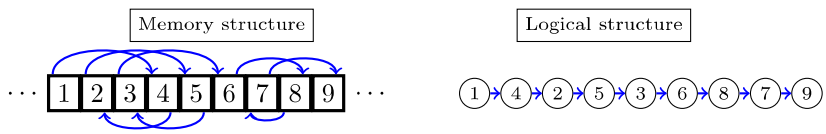

Figure 1.2: State after many insertion and deletion operations. There may also be "dead" cells which are currently not part of the list. These are not shown; we assume that these are taken care of every now and then to avoid wasting too much memory.

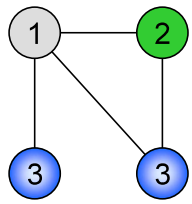

Figure 1.3: 3-colorable graph with a valid coloring.

We then can "cycle" through the colors and perform concurrent operations on all "nodes" (a.k.a. list entries) of the same color without worrying. Once we're done with all colors, we color the new list, and so on. We now have a challenging task:

- We want to use very few colors, so cycling through them is completed quickly. Coloring with a minimal number of colors is in general very hard, but fortunately we're dealing with a very simple graph.
- The coloring itself needs to be done fast, too. Otherwise we'll be waiting for the new coloring to be ready all the time.
- That means we want to use all our processors. It's easy to split up responsibility for the list entries by splitting up the array. The downside of this is that the processors receive only fragmented parts of the list (see Figure 1.4).
- Trying to get consecutive pieces under the control of a single processor requires to *break symmetry:* list fragments get longer only if more nodes are added than removed. If the list is fragmented into single nodes, this roughly means that we want to find a *maximal independent set*, i.e., a set containing no neighbors to which we cannot add a node without destroying this property. This turns out to be essentially the same problem as coloring, as we will see in the exercises.

As the list is fragmented among the processors anyway, it's useful to pretend that we have as many processors as we want. That means each of the nodes can have its "own" processor! If we can deal with this case efficiently, it will certainly work out with fewer processors! Oh, and one more thing: we have some additional information we can glean from the setup. Each node has a *unique identifier* associated with it, namely its array index. Note that this means nodes

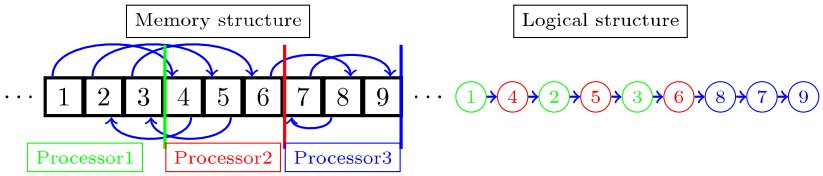

Figure 1.4: List split. We may get lucky in some places of the list (as for the blue processor), but in wide parts the list will be fragmented between processes.

already "look different" initially, which is crucial for coloring deterministically without starting from the endpoints only.

Remarks:

- In distributed computing, we often take the point of view that the system is a graph whose nodes are processors and whose edges are both representing relations with respect to the problem at hand *and* communication links. This will come in handy here as an abstraction, but in many systems it is literally true.

- The assumption of unique identifiers is standard, the reason being that deterministic distributed algorithms can't even do basic things without them (for instance coloring a list quickly). On the other hand, using randomization it is trivial to generate unique identifiers with overwhelming probability. Nonetheless, it is also studied how important such identifiers actually are; more about that in another lecture!

- The linked list here is a toy example, but an entire branch of distributed computing is occupied with finding efficient data structures for *shared memory* systems like the one informally described above. We'll have another look at such systems further into the course!

1.2 2-coloring the list

Clearly, we can color the list with two colors, simply by passing through the list and alternating. Since this is sequential, i.e., only one process is actually working, it takes $\Theta(n)$ steps in a list of n nodes. We can parallelize this strategy, however. For $i \in [n] := \{0, \ldots, n-1\}$, denote by v_i the array index of the ith node in the list. As always in this course, log will denote the base-2 logarithm. First, we add "shortcuts" to our linked list.

Here we assume that processes share a common clock to coordinate the execution of the outer loop, or somehow simulate this behavior. We'll examine this issue more closely in the next lecture; let's assume for now that we can handle this and call each iteration of the outer loop a *round*.

Let's have a closer look at what this algorithm does.

Lemma 1.2 (Shortcuts from pointer jumping)**.** *After r rounds of Algorithm 1, at each array index v_i with $i \geq 2^r$ the index v_{i-2^r} is stored. Likewise, at each index v_i with $i < n - 2^r$, v_{i+2^r} is stored.*

Algorithm 1 Parallel pointer jumping

```
for j = 0, ..., ⌈log n⌉ − 1 do
    for each node v_i in parallel do
        if i − 2^j ≥ 0 and i + 2^j < n then
            add v_{i−2^j} to the entry at array element with index i + 2^j
            add v_{i+2^j} to the entry at array element with index i − 2^j
        end if
    end for
end for
```

Proof. We show the claim by induction. The base case is $r = 0$, i.e., the initial state. As $2^0 = 1$, the statement is just another way of saying that we have a doubly linked list, so we're in the clear. Now assume that the claim is true for some $0 \leq r < \lceil \log n \rceil$. In the rth round, we have $j = r - 1$. For any $i \geq 2^r$, (the process responsible for) node $v_{i-2^{r-1}}$ will add v_{i-2^r} to the entry at array index v_i. It can do this, because by induction hypothesis v_{i-2^r} and v_i can be looked up at the array index $v_{i-2^{r-1}}$. Note that processes dealing with a v_i with $i < 2^{r-1}$ will not get confused: they will know that $i < 2^{r-1}$ because $v_{i-2^{r-1}}$ was not added to the entry corresponding to v_i. Similarly, for each $i < n - 2^r$, $v_{i+2^{r-1}}$ will add v_{i+2^r} to the entry at index v_i. □

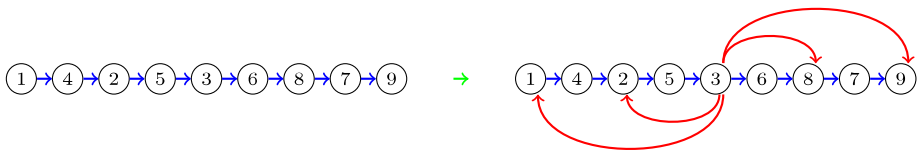

Figure 1.5: Parallel pointer jumping. Depicted are the additional pointers/links of node 3 only.

With these shortcuts, we can color the list quickly.

Algorithm 2 2-coloring the list

```
execute Algorithm 1
color the list head by 0 (i.e., index v_0)
for j = ⌈log n⌉ − 1, ..., 1 do
    for each node v_i colored 0 in parallel do
        if i + 2^j < n then
            color index v_{i+2^j} by 0
        end if
    end for
end for
color all remaining nodes by 1
```

Theorem 1.3 (Correctness of Algorithm 2)**.** *Algorithm 2 colors the doubly linked list with 2 colors.*

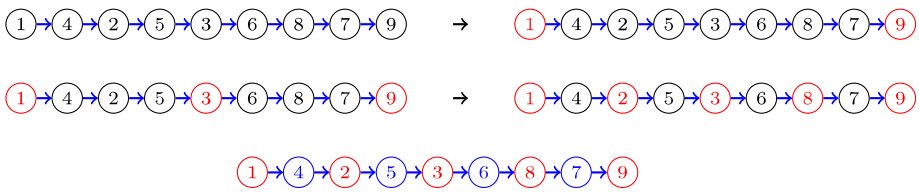

Figure 1.6: Execution of the 2-coloring algorithm. Each step uses a different "level" of pointers constructed with the pointer jumping algorithm; the final steps just uses the neighbor pointers.

Proof. From Lemma 1.2, we know that array elements will store the necessary information to execute the for-loops. By induction, we see that after r rounds of the loop, all nodes v_i with $i \bmod 2^{\lceil \log n \rceil - r} = 0$ are colored 0. The loop runs for $\lceil \log n \rceil - 1$ rounds, i.e., until $j = 1$. Thus, all nodes in even distance from the list head are colored 0, while the remaining nodes get colored 1. □

Remarks:

- In the above algorithms, we referred to n. However, n is unknown due to parallel insertions and deletions (maintaining a shared counter is another fundamental problem!). This can be resolved by letting v_i *terminate* when it knows that its work is done, which is the case when not both v_{i-2^j} and v_{i+2^j} are written to v_i in round j. The processors then just need to notify each other once all their associated nodes are terminated.
- For 2-coloring, the $\mathcal{O}(\log n)$ rounds of this algorithm are the best we can get: another straightforward induction shows that following pointers, it takes $\lceil \log h \rceil$ rounds to "see" something that is h "hops" in the list away, and unless individual processors read large chunks of memory, this has to be done.
- The issue is that 2-coloring is too rigid. Once we color a single node, all other nodes' colors are determined. The problem is not *local*.
- This is also bad for another reason: if we have only small changes in the list, we would like to avoid having to recolor it from scratch. It would be nice to have an algorithm where the output depends only on a small number of hops around each node. This would most likely also yield a fast and efficient algorithm!
- We can use the pointer jumping technique to speed up algorithms that are more local in this sense: if in round r nodes write everything they know to the array entries of nodes in distance 2^{r-1}, it takes only $\lceil \log h \rceil$ steps until the output of an algorithm depending on nodes in distance at most h can be determined. However, this is only practical if h is small, as otherwise a lot of work is done!

1.3 Using 3 colors

What good does it do to get down to two colors, but at a large overhead? None, as we have to do it again after each change of the list. Let's be a bit more relaxed

and permit $c > 2$ colors. This means that, no matter what the neighbors' colors are, there's always a free one to pick! Given that we start with a valid coloring – the array indices – we can use this to reduce the number of colors to 3. For simplicity, let's assume in the following that $v_{-1} = v_{n-1}$ and $v_n = v_0$ (i.e., head and tail of the list also have pointers to each other), since this will simplify describing algorithms.

Algorithm 3 color reduction

```
for each node v_i in parallel do
    c_{v_i} := v_i
end for
while ∃v_i : c_{v_i} > 2 do
    for each node v_i with c_{v_i} > max{c_{v_{i-1}}, c_{v_{i+1}}, 2} in parallel do
        c_{v_i} := min([2] \ {c_{v_{i-1}}, c_{v_{i+1}}})
    end for
end while
```

Lemma 1.4. *Algorithm 3 computes a 3-coloring. It terminates in c rounds, where c is the number of different colors in the initial coloring (here n, because the array indices are unique).*

Proof. No two neighbors can change their color in the same round, as this would require that each of their colors is larger than the other. Thus, the coloring is valid after each round (given that it was valid initially). A node with the current maximum color will change its color (because no neighbor can have this color, too). Note also that no colors other than 0, 1, or 2 are ever picked by a node. It follows that the algorithm completes after at most c rounds and the result is a valid 3-coloring. □

Remarks:

- The time complexity of n rounds is attained if we still have a nice, well-ordered list, i.e., $v_i = i$ for all i. In other words, if we're unlucky, we color the list sequentially.
- The algorithm is however good to reduce the number of colors to 3 if we only have a few colors to begin with.
- If we have an arbitrary graph of *maximum degree* Δ (i.e., no node has more than Δ neighbors), the same approach can be used to find a $(\Delta+1)$-coloring (see Figure 1.7).
- It's not hard to show that if the initial coloring is random, the algorithm will finish in $\Theta(\log n / \log\log n)$ rounds with a very large probability. Can you prove it?
- One can construct a such an initial coloring by picking colors randomly at each node from a sufficiently large range.
- Combining with pointer jumping, we get a running time of $\Theta(\log\log n)$ for 3-coloring, exponentially faster than for 2-coloring!

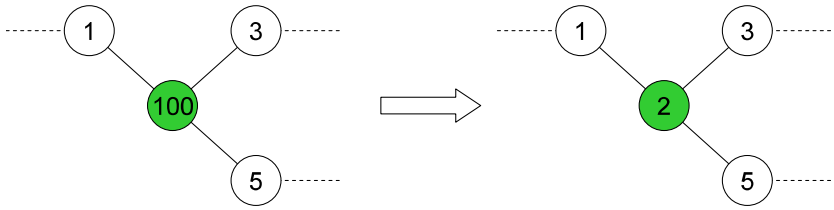

Figure 1.7: Vertex 100 receives the lowest possible color.

1.4 Cole-Vishkin

The previous algorithm reduced the number of colors in each step, starting from a valid coloring. Obviously, we can now ask the question whether we can do this more quickly/efficiently? The answer turns out to be yes, as shown by the following algorithm, which is based on a simple, but ingenious idea.

Algorithm 4 Cole-Vishkin color reduction

```
for each node v_i in parallel do
    c_{v_i} := v_i
end for
while ∃v_i : c_{v_i} > 5 for all nodes in parallel do
    interpret c_{v_i} and c_{v_{i-1}} as (infinite) little-endian bit-strings, i.e., starting
    with the least significant bit
    let j be the smallest index where they differ
    concatenate j (as bitstring) and the differing bit itself, yielding color c
    c_{v_i} := c
end while
```

Example:

Part of an execution of Algorithm 4:

v_{i-2}	0010110000	$\rightarrow$	...	$\rightarrow$	...
v_{i-1}	1010010000	$\rightarrow$	01010	$\rightarrow$	...
v_i	0110010000	$\rightarrow$	10001	$\rightarrow$	001

The trick is that either the first or the second part of (the bit string of) the new color saves the day.

Lemma 1.5 (Correctness of Cole-Vishkin)**.** *Algorithm 4 computes a valid coloring.*

Proof. Since the initial coloring is valid, we need to show that a valid coloring enables to compute the new colors and the new coloring is valid. The first part readily follow from the fact that two different colors must have differing bit strings, so the index j can be computed. Now consider two neighbors v_i and v_{i-1}. If the determine different indices j for which the current colors differ from v_{i-1} and v_{i-2} respectively, the front part of the new colors is different. Otherwise, the second part of their new colors consists precisely of the least significant *differing* bit! $\square$

This algorithm terminates in (almost) $\log^* n$ time. Log-Star is the *number* of times one needs to take the logarithm (to the base 2) to get to at most 2, starting with n:

Definition 1.6 (Log-Star)**.**
$\forall x \leq 2: \ \log^* x := 1 \quad \forall x > 2: \ \log^* x := 1 + \log^*(\log x)$

Theorem 1.7. *Algorithm 4 computes a valid* 6*-coloring in* $\log^* n + \mathcal{O}(1)$ *rounds.*

Proof. Correctness is shown in Lemma 1.5. The time complexity follows from the fact that if the original color had b bits, the new color has at most $\lceil b \rceil + 1$ bits: the number of bits to encode a index in a b-bit string plus the appended bit. The $\mathcal{O}(1)$ term addresses the fact that we don't actually apply the base-2 logarithm in each step. (The non-exciting computations showing that this makes only a minor difference are omitted.) The reason why we end up with 6 colors is simple: encoding an index yields 00, 01, or 10 as leading parts; appending a bit yields 6 possibilities. It's simple to check that for any larger number of initial colors, fewer possibilities will remain. □

Remarks:

- Log-star is an amazingly slowly growing function. Log-star of all the atoms in the observable universe (estimated to be 10^{80}) is 5. Hence, for all practical purposes, it's constant.
- One can use Algorithm 3 to reduce the number of colors to 3 in 3 rounds.
- As stated, the algorithm has a termination condition that cannot be *checked* efficiently based on local information. Fortunately, we can just get rid of this condition and run the algorithm for the right number of rounds given by Theorem 1.7.
- This does not work if n is unknown. This issue has two different solutions: a practical one and a theoretical one. Can you figure out both?
- For a change, the $\mathcal{O}(1)$ term is actually hiding only a *small* constant. It has been narrowed down quite exactly how fast one can make this approach [Ryb11].
- Another detail here is that instead of n, the argument of the $\log^*$ is, in fact, the initial *range* of colors. In our case, this is the current size of the array, which may be larger than n, typically by some constant factor. However, even if it would be exponentially larger, this would mean we need to do just one or two more rounds of the algorithm to handle this.
- A simple modification results in running time $1/2 \cdot \log^* n + \mathcal{O}(1)$ (see exercises).
- Using pointer jumping, the running time can be reduced to $\log(\log^* n) + \mathcal{O}(1)$. Shockingly, this is *not* the most ridiculously slow-growing function I've encountered in a statement that is not deliberately about slow-growing functions.
- The technique is not limited to lists. It can be used to color oriented trees and constant-degree graphs in $\mathcal{O}(\log^* n)$ rounds, too (see exercises).

1.5 Linial's lower bound

If we can color the list *that* fast, can't we find an algorithm that does it in truly constant time? The answer is no, and we're going to see now why. We'll focus on the case where interactions are solely with neighbors, in which one requires $\Omega(\log^* n)$ rounds. Such algorithms are called *message-passing algorithms*, for reasons that will be discussed in the next lecture. With shared memory, the variant of Cole-Vishkin with pointer jumping is asymptotically optimal [FR90]. We also restrict to deterministic algorithms.

Before we do the proof, let's simplify the situation a bit. First, observe that all information the output of v_i can be influenced by in a T-round message passing algorithm is the information that's initially available at nodes $v_{i-T}, v_{i-T+1}, \ldots, v_{i+T}$. In the worst case, every content stored is identical,[1] so the only real difference are the actual array indices (and memory addresses). Note also that the order is relevant: We have forward and backward pointers, i.e., we can distinguish directions, and obviously it's possible to count the number of "hops" traversed. Consequently, even if we don't know anything about a coloring algorithm except that it is a deterministic T-round algorithm $\mathcal{A}$ (with neighbor-neighbor interactions only), we can conclude that there is a function

$$f : (x_0, \ldots, x_{2T}) \to [c]$$

so that $c_{v_i} = f(v_{i-T}, v_{i-T+1}, \ldots, v_{i+T})$ when executing $\mathcal{A}$. Here, c is the number of colors used by $\mathcal{A}$ and we assume *without loss of generality* (w.l.o.g.) that $[c]$ is the set of colors produced by the algorithm.[2]

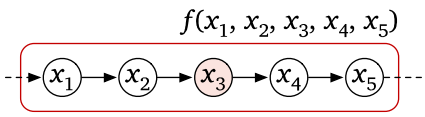

Figure 1.8: Interpreting a 2-round coloring algorithm as a coloring function f mapping 5-tuples to colors.

If $\mathcal{A}$ produces a valid coloring, we also know that

$$f(x_0, \ldots, x_{2T}) \neq f(x_1, \ldots, x_{2T+1})$$

provided that $x_i \neq x_j$ for $i \neq j$, $i, j \in [2T+2]$: the two arguments could be the views of adjacent nodes in the list, and they must not compute the same color.

Now comes the clever bit making our lives much easier: we restrict the problem without actually taking away what makes it hard. This will simplify our key argument, as it has an algorithmic component – and it would be more challenging to come up with an algorithm for the more general setting. For $c, k \in \mathbb{N}$, we say that g is a *k-ary c-coloring function* if

$$\forall 0 \leq x_1 < x_2 < \ldots < x_k < n : \; g(x_1, x_2, \ldots, x_k) \in [c]$$

[1] Even if that wasn't true, the same argument applies taking this content into account.
[2] As opposed to, e.g., {pink, elephant, turtle}.

and

$$\forall 0 \leq x_1 < x_2 < \ldots < x_{k+1} < n : \; g(x_1, x_2, \ldots, x_k) \neq g(x_2, x_3, \ldots, x_{k+1}).$$

For $k = 2T+1$, these are the exact same requirements as to f, however, only for ascending addresses $x_1 < \ldots < x_{k+1} < n$. Note that by restricting the domain of f to such inputs, we see that the existence of a T-round algorithm $\mathcal{A}$ using c colors implies the existence of a $(2T+1)$-ary c-coloring function f.

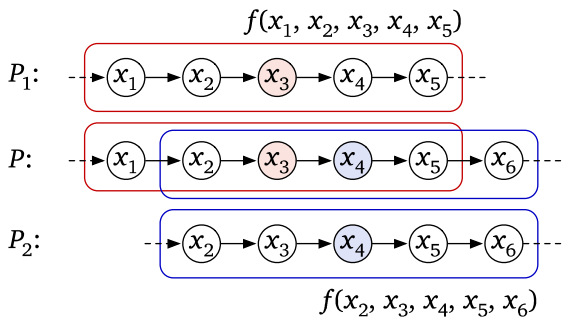

Figure 1.9: 5-tuples that correspond to possible views of adjacent nodes must result in different colors.

Using this connection, we can now move on to the proof of the lower bound, which consists of showing that if c is small, then T cannot be arbitrarily small, too.

Lemma 1.8 (1-ary functions require many colors)**.** *If f is a 1-ary c-coloring function, then $c \geq n$.*

Proof. By definition, $f(x_1) \neq f(x_2)$ for all $0 \leq x_1 < x_2 < n$, i.e.,

$$\forall x_1 \neq x_2 \in [n] : \; x_1 \neq x_2 \Leftrightarrow f(x_1) \neq f(x_2).$$

In other words, f is an injection, which is only possible if $c \geq n$. □

The main step of the proof is to show that we can construct $(k-1)$-ary 2^c-coloring functions out of k-ary c-coloring functions. That is, we can "pay" for saving time by using more colors.

Lemma 1.9 (k-ary c-coloring enables $(k-1)$-ary 2^c-coloring)**.** *If f is a k-ary c-coloring function for some $k > 0$, then a $(k-1)$-ary 2^c-coloring function g exists.*

Proof. First, let h be a bijection from the subsets of $[c]$ to $[2^c]$. Concretely, we may choose for $S \subseteq [c]$ as $h(S)$ the string of c bits in which the ith bit is 1 if and only if $i - 1 \in S$ (but any other bijection would do, too).

Next, define

$$g'(x_1, \ldots, x_{k-1}) := \{f(x_1, \ldots, x_k) \,|\, x_{k-1} < x_k < n\},$$

i.e., g' is the *set* of all colors that can possibly be assigned by f when all but the last argument of f are specified. These are the colors that might cause trouble

when g assigns a color to $x_1, \ldots, x_{k-1}$ without considering x_k. Using h, we can interpret this set as a new color:[3]

$$g(x_1, \ldots, x_{k-1}) := h \circ g'(x_1, \ldots, x_{k-1}) = h(g'(x_1, \ldots, x_{k-1})).$$

It's straightforward to check that this is a well-defined function with range $[2^c]$: $g'(x_1, \ldots, x_{k-1}) \subseteq [c]$ and h maps such sets to a color from $[2^c]$. In order to verify that g is indeed a $(k-1)$-ary 2^c-coloring function, we thus must show that

$$\forall 0 \leq x_1 < x_2 < \ldots, x_k < n : \; g(x_1, \ldots, x_{k-1}) \neq g(x_2, \ldots, x_k).$$

Let $0 \leq x_1 < x_2 < \ldots < x_k < n$. Clearly, $f(x_1, \ldots, x_k) \in g'(x_1, \ldots, x_{k-1})$. On the other hand, we have that $f(x_1, \ldots, x_k) \neq f(x_2, \ldots, x_{k+1})$ for any $x_k < x_{k+1} < n$, because f is a coloring function. This is equivalent to saying that $f(x_1, \ldots, x_k) \notin g'(x_2, \ldots, x_k)$. We conclude that $g'(x_1, \ldots, x_{k-1}) \neq g'(x_2, \ldots, x_k)$. Since h is a bijection, this is equivalent to

$$g(x_1, \ldots, x_{k-1}) = h(g'(x_1, \ldots, x_{k-1})) \neq h(g'(x_2, \ldots, x_k)) = g(x_2, \ldots, x_k).$$

□

With these lemmas, it's a piece of cake to obtain the lower bound.

Theorem 1.10 (Linial's lower bound)**.** *Coloring a list with a message passing algorithm that uses (at most) 4 colors requires at least* $1/2 \cdot \log^* n - 1$ *rounds.*

Proof. Assume that $\mathcal{A}$ is a T-round coloring algorithm using 4 colors. Thus, a $(2T+1)$-ary 4-coloring function exists. We apply Lemma 1.9 for $2T$ times, to see that then a 1-ary $\left({}^{2T}2\right)^4$-coloring function exists. Here, ${}^a2$ denotes the tetration or "power tower", the a-fold iterated exponentiation by 2. From Lemma 1.8, we know that

$${}^{2T+2}2 = \left({}^{2T}2\right)^4 \geq n,$$

yielding

$$2T + 2 \geq \log^* n$$

and finally

$$T \geq \frac{\log^* n}{2} - 1.$$

□

Remarks:

- More colors don't help a lot. If we consider c colors in the above proof, we get that it requires at least $1/2 \cdot (\log^* n - \log^* c)$ rounds to color with c colors.
- Randomization doesn't help either. Naor extended the lower bound to randomized algorithms [Nao91].

[3] Note that h doesn't really do anything but "rename" the sets such that they are easy to count. That's why, rather than turtles or sets, we like our colors to be numbers!

- I've been a bit sloppy, as I haven't defined the model precisely. This can easily lead to mistakes, so I will make amends in the next lecture. The given proof works in the so-called *message passing* model, which we get to know in more detail in the next lecture.

- If one permits non-neighbor interactions, the lower bound weakens to $\lceil \log(1/2 \cdot (\log^* n - \log^* c)) \rceil$ [FR90], just like we could speed up the Cole-Vishkin algorithm using pointer jumping.

What to take home

- Exploiting parallelism, distributed algorithms can be extremely fast.

- *Symmetry breaking* is a fundamental challenge in distributed computing, and colorings are a basic structure that breaks symmetry between neighbors.

- The key to understanding parallelism is to understand what is possible based on limited (in particular local) information.

- What can and can't be done is quite sensitive to the model. When considering running time bounds, impossibility results, etc. it is thus important to keep in mind that changing an aspect of the model may have a dramatic impact. Try always to understand what aspects of a model cause a certain result, and wonder whether changing them would change the game!

- On the other hand, we can frequently prove *unconditional* lower bounds in distributed computing, such as Theorem 1.7. If we *do* figure out what the suitable model of computation is for a given system, we may be able to understand precisely how fast things can be done. Contrast this with lower bounds on sorting (which restrict the feasible operations) or impossibilities in the sequential world that rest on conjectures like $\mathrm{P} \neq \mathrm{NP}$ or the unique games conjecture!

- Math is going to be our friend in this lecture. If your reflex is to disagree, try to imagine figuring out how fast the list can be colored by concurrent processes without the tools we used. Moreover, coming up with a proof requires us to rethink our assummptions and crystallize ideas; that's difficult, but very useful when dealing with more complex problems later on!

Bibliographic Notes

The basic technique of the log-star algorithm is by Cole and Vishkin [CV86]. The technique can be generalized and extended, e.g., to a ring topology or to graphs with constant degree [GP87, GPS88, KMW05]. Using it as a subroutine, one can solve many problems in log-star time.

The lower bound of Theorem 1.7 is due to Linial [Lin92]. Linial's paper also contains a number of other results on coloring, e.g., that any message passing algorithm for coloring d-regular trees of radius r that runs in time at most

$2r/3$ requires at least $\Omega(\sqrt{d})$ colors. The presentation here is based on a more streamlined version by Laurinharju and Suomela [LS14].

Figures 1.8 and 1.9 are courtesy of Jukka Suomela and under a creative commons license.[4] Figures 1.3 and 1.7 are courtesy of Roger Wattenhofer; substantial parts of today's lecture are based on material from his course at ETH Zurich.

Wide parts of today's lecture are covered by books ([CLR90],[Pel00]).

Bibliography

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.

[CV86] R. Cole and U. Vishkin. Deterministic coin tossing and accelerating cascades: micro and macro techniques for designing parallel algorithms. In *18th annual ACM Symposium on Theory of Computing (STOC)*, 1986.

[FR90] Faith E. Fich and Vijaya Ramachandran. Lower bounds for parallel computation on linked structures. In *Proc. 2nd Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA 1990)*, pages 109–116, 1990.

[GP87] Andrew V. Goldberg and Serge A. Plotkin. Parallel (Δ+1)-coloring of constant-degree graphs. *Inf. Process. Lett.*, 25(4):241–245, June 1987.

[GPS88] Andrew V. Goldberg, Serge A. Plotkin, and Gregory E. Shannon. Parallel Symmetry-Breaking in Sparse Graphs. *SIAM J. Discrete Math.*, 1(4):434–446, 1988.

[KMW05] Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. On the Locality of Bounded Growth. In *24th ACM Symposium on the Principles of Distributed Computing (PODC), Las Vegas, Nevada, USA*, July 2005.

[Lin92] N. Linial. Locality in Distributed Graph Algorithms. *SIAM Journal on Computing*, 21(1)(1):193–201, February 1992.

[LS14] Juhana Laurinharju and Jukka Suomela. Brief Announcement: Linial's Lower Bound Made Easy. In *Symposium on Principles of Distributed Computing (PODC)*, pages 377–378, 2014.

[Nao91] Moni Naor. A Lower Bound on Probabilistic Algorithms for Distributive Ring Coloring. *SIAM J. Discrete Math.*, 4(3):409–412, 1991.

[Pel00] David Peleg. *Distributed computing: a locality-sensitive approach*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[Ryb11] Joel Rybicki. Exact bounds for distributed graph colouring. Master's thesis, Department of Computer Science, University of Helsinki, May 2011.

[4] CC BY-SA 3.0, see https://creativecommons.org/licenses/by-sa/3.0/.