Parallel Programming

Parallel algorithms 2

Morgan Ericsson

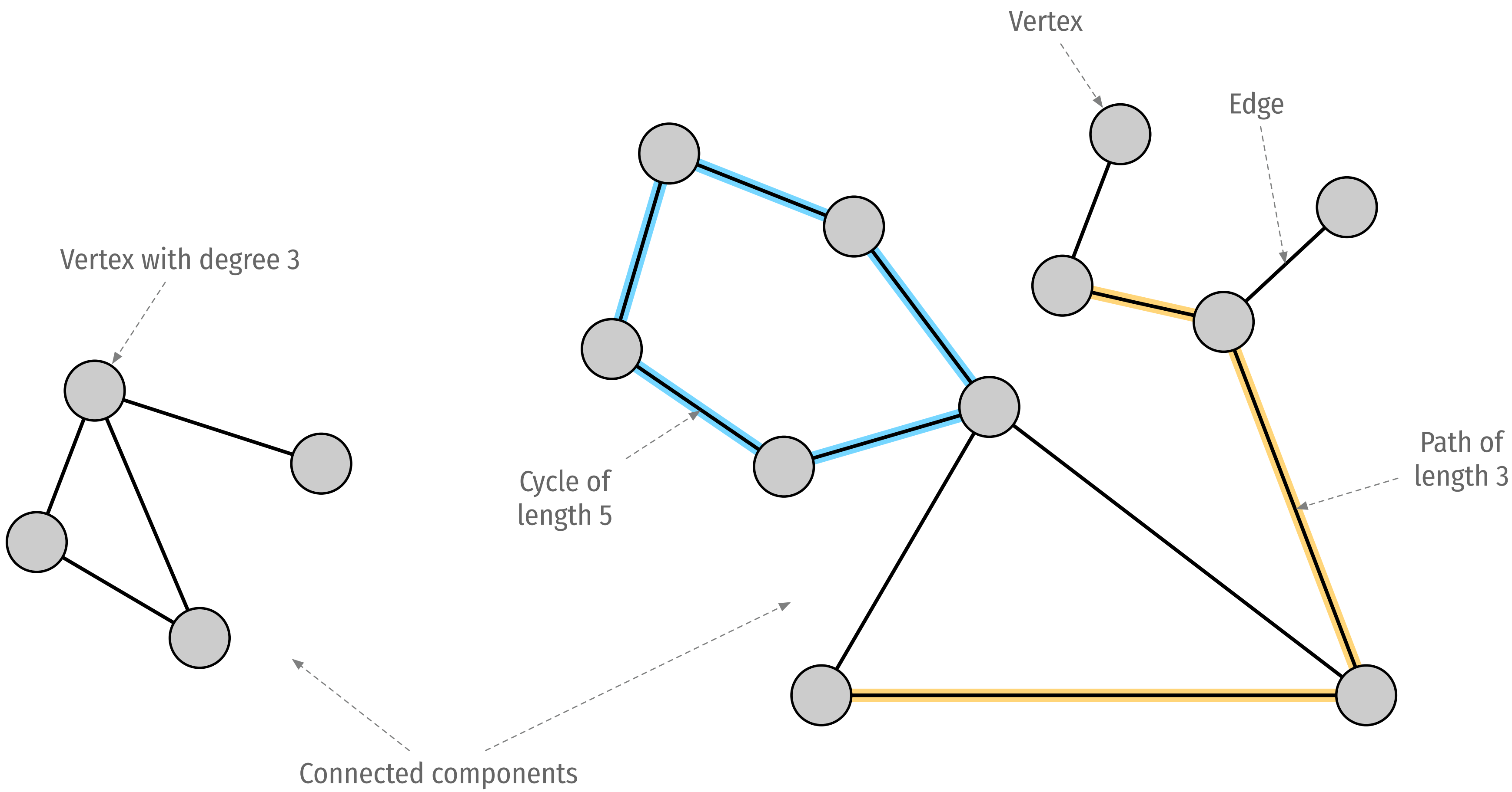
Today

» Graphs

» Searching

Graphs

Remember graphs ...



… and DFS

```python
class DFSPaths:
  def __init__(self, g:Graph, s:int) -> None:
    self.G = g
    self.s = s
    self.marked = np.zeros(self.G.V, dtype=bool)
    self.edge_to = np.zeros(self.G.V, dtype=int)
    self._dfs(s)
```

... and DFS

```python
def _dfs(self:DFSPaths, v:int) -> None:
  self.marked[v] = True
  for w in self.G.adj(v):
    if not self.marked[w]:
      self._dfs(w)
      self.edge_to[w] = v
```

Concurrent DFS?

- We use a recursive algorithm
- The graph is shared, but read-only
- `marked` and `edge_to` are shared and read-write

First, iterative version!

```
def _dfs(self) -> None:
  while not self.Q.empty():
    v = self.Q.get()
    if self.marked[v]:
      continue
    self.marked[v] = True
    for w in self.G.adj(v):
      if not self.marked[w]:
        self.Q.put(w)
```

First, iterative version!

```python
class DFSPaths:
  def __init__(self, g:Graph, s:int) -> None:
    self.G = g
    self.s = s
    self.marked = np.zeros(self.G.V, dtype=bool)
    self.Q = LifoQueue()
    self.Q.put(self.s)
    self._dfs()
```

Making dfs concurrent

```
def _dfs(self) -> None:
  while not self.Q.empty():
    v = self.Q.get()
    lock.acquire()
    if self.marked[v]:
      continue
    lock.release()
    lock.acquire()
    self.marked[v] = True
    lock.release()
    for w in self.G.adj(v):
      lock.acquire()
      if not self.marked[w]:
        self.Q.put(w)
      lock.release()
```

Fails

- The queue can be empty at times, causing threads to exit prematurely
- if ... continue inside a lock is a really bad idea
- We can get rid of one of the if marked/if not marked checks

Second attempt

```
def _dfs(self) -> None:
  while True:
    v = self.Q.get()
    lock.acquire()
    mark = self.marked[v]
    lock.release()
    if mark:
      continue
    lock.acquire()
    self.marked[v] = True
    lock.release()
    for w in self.G.adj(v):
      self.Q.put(w)
```

Ugly

```python
lock = Lock()

lock.aquire()
try:
    # do something critical
    pass
finally:
    lock.release()
```

We can use a context manager

```python
lock = Lock()

with lock:
    # do something critical
    pass
```

Third attempt

```python
def _dfs(self) -> None:
  while True:
    v = self.Q.get()
    with lock:
      mark = self.marked[v]
      if not mark:
        self.marked[v] = True
    if mark:
      continue
    for w in self.G.adj(v):
      self.Q.put(w)
```

Still fails

» We have not addressed the queue

» So it never exits

» We can use the same trick we used for quicksort

Fourth attempt

```python
def _dfs(self) -> None:
  while true:
    self.qsem.aquire()
    v = self.Q.get()
    with lock:
      mark = self.marked[v]
      if not mark:
        self.marked[v] = True
    if mark:
      continue
    for w in self.G.adj(v):
      self.Q.put(w)
      self.qsem.release()
```

Still fails

- Will not exist, stuck on the queue semaphore
- We can check if all node are processed

Fifth attempt

```
def _dfs(self) -> None:
  while true:
    self.qsem.aquire()
    if self.cnt == self.G.V:
      break
    v = self.Q.get()
    with lock:
      mark = self.marked[v]
      if not mark:
        self.marked[v] = True
        self.cnt += 1
    if not mark:
      for w in self.G.adj(v):
        self.Q.put(w)
        self.qsem.release()
        if self.cnt == self.G.V:
          self.tsig.set()
```

Spawning threads

```python
def __init__(self, g:Graph, nt:int) -> None:
  self.G = g
  self.marked = np.zeros(self.G.V, dtype=bool)
  self.Q = LifoQueue()
  self.lock = Lock()
  self.tsig = Event()
  self.cnt = 0

  for v in range(self.G.V): self.Q.put(v)
  self.qsem = Semaphore(self.G.V)
  self.ts = [Thread(target=self._dfs) for _ in range(nt)]
  for t in self.ts: t.start()
  self.tsig.wait()
  for t in self.ts: t.join()
```

One problem remains

```
while true:
  self.qsem.aquire()
  if self.cnt == self.G.V:
    break
```

- We previously introduced a timeout and a loop
- How should we set the timeout?
  - Unnecessary wake-ups or long delays
- We can use another trick!

Spawning threads

```
def __init__(self, g:Graph, nt:int) -> None:
  self.G = g
  self.marked = np.zeros(self.G.V, dtype=bool)
  self.Q = LifoQueue()
  self.lock = Lock()
  self.tsig = Event()
  self.cnt = 0

  for v in range(self.G.V): self.Q.put(v)
  self.qsem = Semaphore(self.G.V)
  self.ts = [Thread(target=self._dfs) for _ in range(nt)]
  for t in self.ts: t.start()
  self.tsig.wait()
  self.qsem.release(nt)
  for t in self.ts: t.join()
```

Can we improve?

```python
with lock:
  mark = self.marked[v]
    if not mark:
      self.marked[v] = True
      self.cnt += 1
```

Few locks

- We use a single lock for the marked array
- It is safe to mark two different vertices in parallel
- This can be a problem for large graphs
- Reduced concurrency, parallelism, performance

Fix?

Setup

```
lock = Semaphore(10)
```

Threads

```
with lock:
  mark = self.marked[v]
    if not mark:
      self.marked[v] = True
      self.cnt += 1
```

No!

- The semaphore allows 10 threads to enter the critical section
- Which can all modify the same vertex
- The locks must be connected to the vertices

Fix?

Setup

```
locks = [Lock() for _ in range(self.G.V)]
```

Threads

```
with locks[v]:
  mark = self.marked[v]
    if not mark:
      self.marked[v] = True
      self.cnt += 1
```

Better, not good

- Many locks in large graphs
- Waste of resources
- We will most likely not access all vertices at once
- But, an idea we can work with

Fix?

Setup

```
locks = [Lock() for _ in range(NUM_LOCKS)]
```

Threads

```
with locks[v % NUM_LOCKS]:
  mark = self.marked[v]
    if not mark:
      self.marked[v] = True
      self.cnt += 1
```

Back to Go

```go
type DFSPaths struct {
	g        Graph
	s, nc    int
	marked   []bool
	edge_to  []int
	lock     sync.Mutex
	ch       chan int
}
```

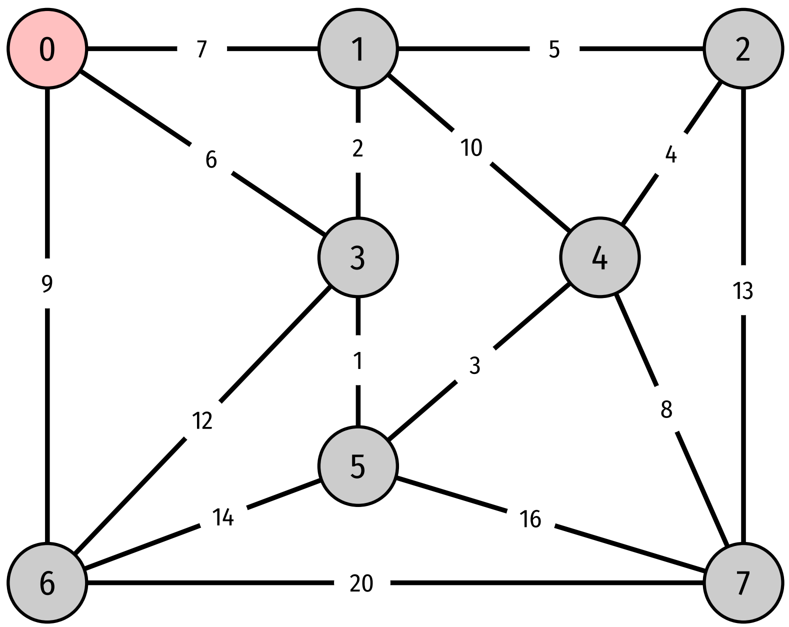
Back to Go

```go
func (dfp *DFSPaths) dfs() {
    var mark bool
    for v := range dfp.ch {
        dfp.lock.Lock()
        mark = dfp.marked[v]
        dfp.lock.Unlock()

        if mark { continue }

        dfp.lock.Lock()
        dfp.marked[v] = true
        dfp.nc += 1
        dfp.lock.Unlock()

    // ...
```

Back to Go

```
    for _, w := range dfp.g.Adj(v) {
            dfp.lock.Lock()
            mark = dfp.marked[w]
            dfp.lock.Unlock()
            if !mark {
                dfp.ch <- w
                dfp.lock.Lock()
                dfp.edge_to[w] = v
                dfp.lock.Unlock()
            }
        }
        dfp.lock.Lock()
        if dfp.nc == dfp.g.V() {
            close(dfp.ch)
        }
        dfp.lock.Unlock()
    }
}
```

Prim's algoritm (MST)




```
3-5, 1
1-3, 2
4-5, 3
2-4, 4
1-2, 5
0-3, 6
0-1, 7
4-7, 8
0-6, 9
1-4, 10
3-6, 12
2-7, 13
5-6, 14
5-7, 16
6-7, 20
```

Prim's algoritm (MST)

- Prim's algorithm is sequential
  - We add one edge per iteration
  - Which edges are available to add depends on the previous iteration
- No need for exhaustive search, we know which edge to add
- But, we can improve how we find that edge

Remember parallel reductions

- We can replace the heap with a parallel reduction
- Find the minimum weight across a large list of edges

All-pairs shortest path (APSP)

- Dijkstra and Bellman-Ford computed single source shortest path
- APSP is equivalent to running, e.g., Dijkstra on all possible sources
  - Which can be done in parallel
  - But unnecessary overhead
- Floyd-Warshall

Remember `EWDiGraph`

- Directed graph with edge weights
- We can iterate over all edges
  - Get source, destination, and weight

Distance matrix

```python
def gendistmatrix(g):
  dm = np.full((g.V, g.V), np.inf)
  np.fill_diagonal(dm, 0)

  for e in g.edges
    dm[e.src, e.dst] = e.weight

  return dm
```

Serial Floyd-Warshall

```
def FloydW(dm, V):
  for k in range(V):
    for i in range(V):
      for j in range(V):
        dm[i, j] = min(dm[i, j], dm[i, k] + dm[k, j])
```

Concurrency

- We cannot do the outer loop
- We can do the *i* or *j* loops
  - Which one?
- Outmost is better, since that leaves more work per thread
- Can mix in this case, just interested in all combinations of i and j.
  - No dependencies between them, only to k

Benefits?

- Floyd-Warshall is $O(N^3)$ operations
- Finding min is $O(N)$
- Parallel require slighly more operations, but executed in parallel
  - Assume P processors
  - Floyd-Warshall takes N^3 / P
  - Finding min takes $N / P + p$
- Speedup approaches P if $N \gg P$

Benefits?

- We do not improve the number of operations required
  - Often the opposite, we may have to do more!
  - To achieve concurrency
- We improve the number of operations we can do at once
  - Parallel execution
- Synchronization increases the number of operations and reduces the “at once”-part

Searching

Unsorted lists

- Searching in unsorted lists is easy
  - Just a reduction
- E.g., find min, but looking for specific value instead

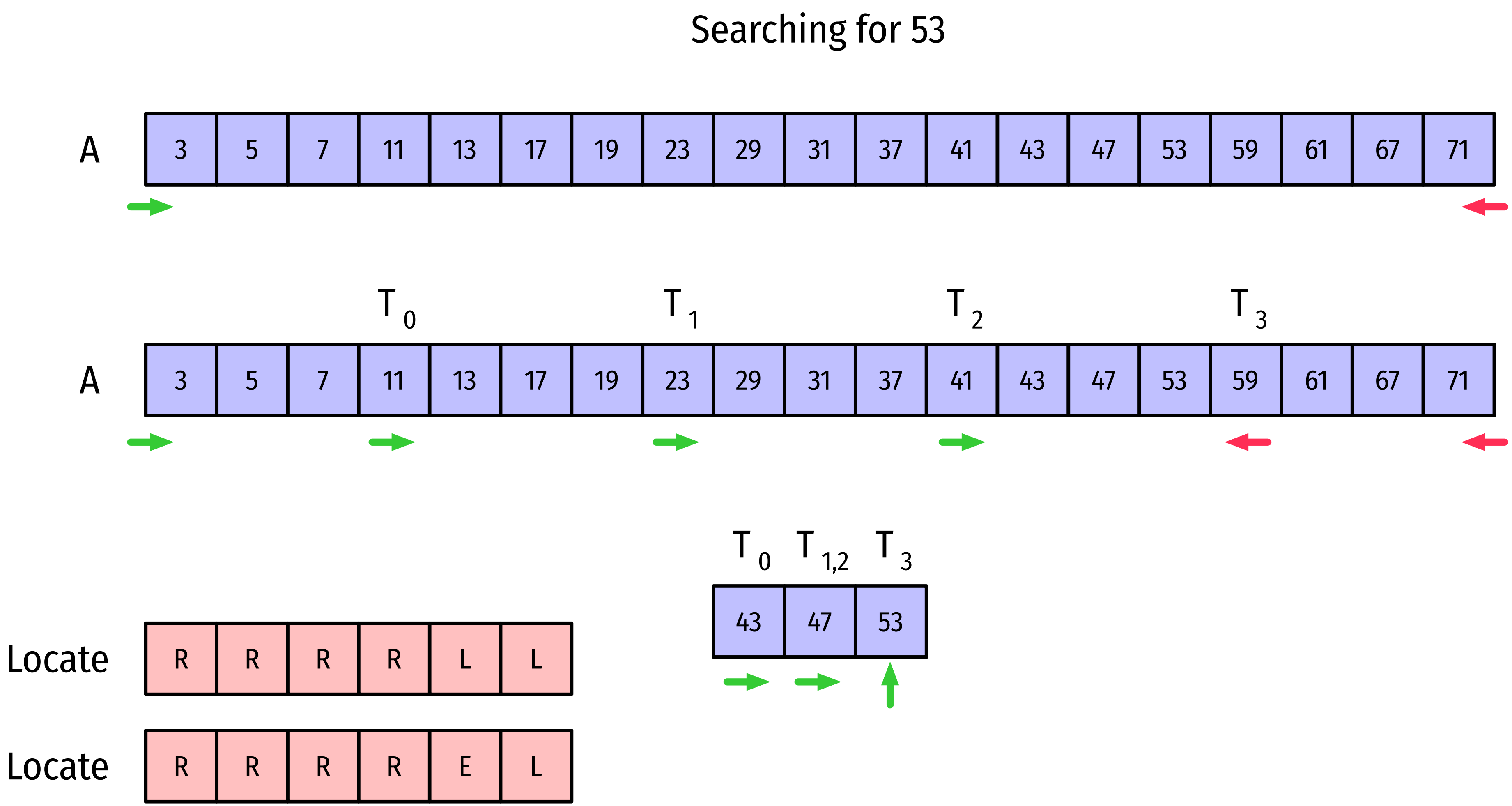
Sorted lists

```python
def binsearch(l:list[int], x:int) -> int|None:
  lo, hi = 0, len(l) - 1

  while lo <= hi:
    mid = (lo + hi) // 2
    if l[mid] == x:
      return mid
    elif l[mid] < x:
      lo = mid + 1
    else:
      hi = mid - 1

  return None
```

N-aray search

- Assume *N* threads
- Checks *N* well-spaced values, one per thread
- Marks the values based on the result of comparisons

Example


Searching for 53
A
3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71
T 0
T 1
T 2
T 3
A
3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71
T 0
T 1,2
T 3
43 47 53
Locate
R R R R L L
Locate
R R R R E L

N-aray search (serial)

```
def nary(A, lo, hi, key, intv):
  mid, locate = [None] * (intv + 1), [None] * (intv + 2)
  pos = None

  locate[0], locate[intv + 1] = 'R', 'L'
  while lo <= hi and pos is None:
    mid[0] = lo - 1
    step = (hi - lo + 1) // (intv + 1)

    mark_loc()

    for i in range(1, intv + 1):
      if locate[i] != locate[i - 1]:
        lo = mid[i - 1] + 1
        hi = mid[i] - 1

    if locate[intv] != locate[intv + 1]:
      lo = mid[intv] + 1

  return pos
```

N-aray search (serial)

```
def mark_loc():
  for i in range(1, intv + 1):
    offs = step * i + (i -1)
    mid[i], lmid = lo + offs, lo + offs

    if lmid <= hi:
      if A[lmid] > key:
        locate[i] = 'L'
      elif A[lmid] < key:
        locate[i] = 'R'
      else:
        locate[i] = 'E'
        pos = lmid
    else:
      mid[i] = hi + 1
      locate[i] = 'L'
```

Concurrent version (sketch)

- The while loop will be the starting point of the threads
- The variables before should be done synchronized by a single thread
- `mark_loc` will be the main concurrent work
- The for-loop and if-statement needs to be run by a single thread

Concurrent version (sketch)

1. Spawn threads that start the while
2. Barrier
3. Do updates
4. Barrier
5. Run mark-lock (consider shared/local data)
6. barrier
7. for and if
8. barrier