Utrecht University

B3CC: Concurrency

06: Delta-stepping

Ivo Gabe de Wolff

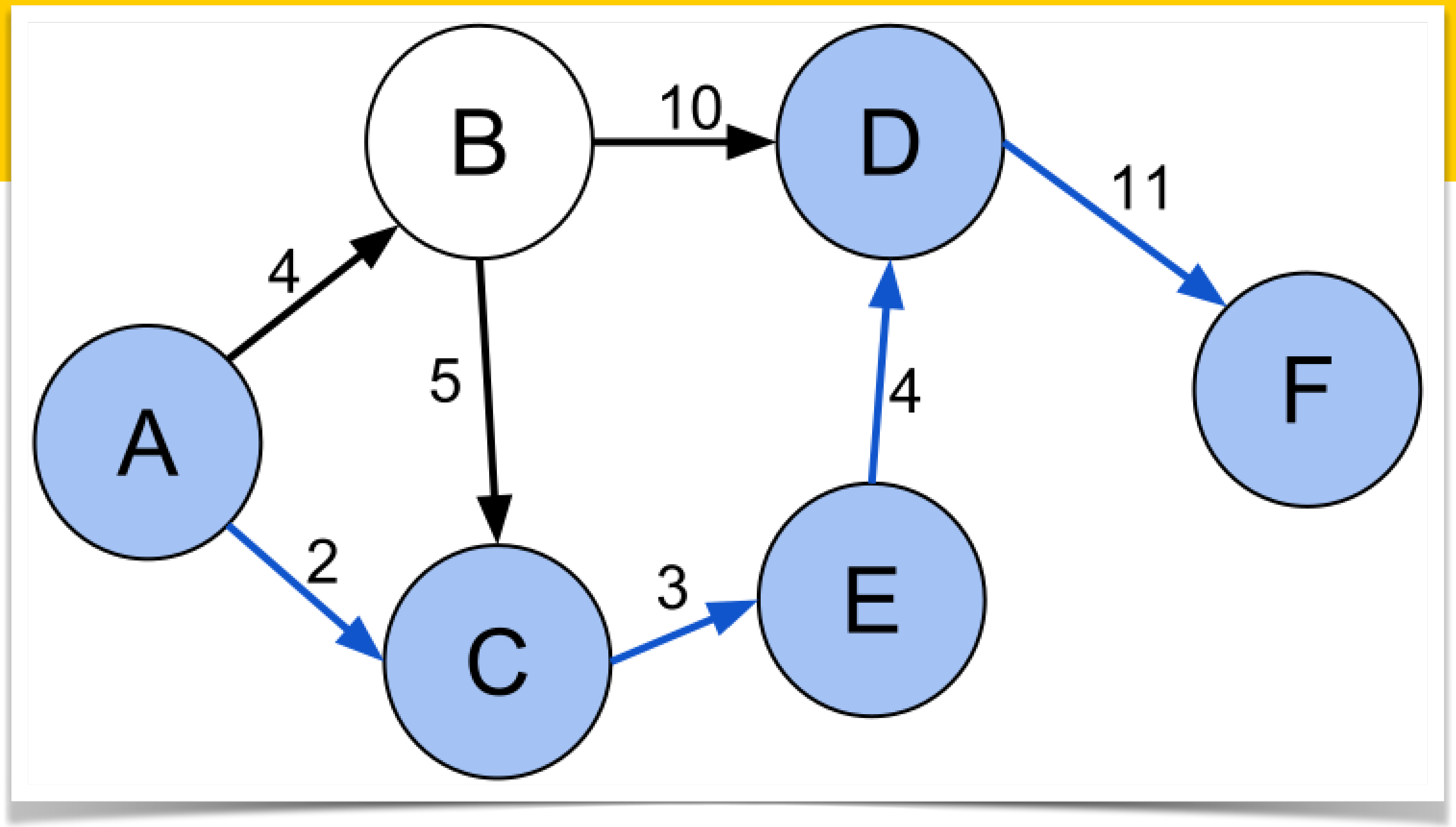
Announcement

- The second practical assignment has been released
  - https://ics.uu.nl/docs/vakken/b3cc/assessment.html
  - You may work in pairs, if you wish
  - Deadline: 22-12-2023 @ 23:59

Parallel algorithms

- You have seen many sequential algorithms
  - In “datastructuren”
- Can we convert a sequential algorithm to a parallel algorithm?
  - No automatic approach
  - Sequential code typically has a long, sequential, critical path
- Today: Converting Dijkstra’s shortest-path algorithm to Delta-stepping

SSSP



- A central problem in algorithmic graph theory is the *single-source shortest path* problem
  - e.g. starting at vertex A, what is the shortest path to reach vertex F?
  - Many practical and theoretical applications
  - One of the benchmarks used in the Graph500 supercomputer ranking
    - The smallest problem size uses 2^{26} vertices, requiring 17 GB RAM

https://en.wikipedia.org/wiki/Shortest_path_problem

SSSP

- Given…
  - A directed graph $G(V, E)$ with $n = |V|$ nodes (or vertices) and $m = |E|$ edges
  - A distinguished node in the graph s: the "source"
  - A function c that returns the (non-negative) weight of a given edge in G
- Objective:
  - For each node v reachable from s,
    compute the weight of the minimum-weight (i.e. shortest) path from s to v
  - The *weight of the path* is the sum of the weights of its edges, denoted $\text{dist}(s,v)$ or $\text{dist}(v)$
  - If v is not reachable from s, then $\text{dist}(s,v) \coloneqq \infty$

Dijkstra's algorithm

- Keep track of *tentative distance* per vertex
  - The distance of the shortest known path
- Repeatedly,
  - Take the unvisited node v with the shortest tentative distance
  - Its tentative distance is now fixed
  - Look at its neighbors: we may have a shorter path to them, via v
  - Update tentative distances of neighbors accordingly

Dijkstra’s algorithm

- Keep track of *tentative distance* per vertex
  - The distance of the shortest known path
- Repeatedly,
  - Take the unvisited node v with the shortest tentative distance
  - Its tentative distance is now fixed
  - Look at its neighbors: we may have a shorter path to them, via v
  - Update tentative distances of neighbors accordingly

O(n) iterations

O(log n) with
proper data structure
(min heap)

Towards parallel shortest path

- Keep track of *tentative distance* per vertex
  - The distance of the shortest known path
- Repeatedly,
  - Take some set of nodes
  - ~~Their tentative distances are now fixed~~
  - Look at their neighbors: we may have a shorter path to them
  - Update tentative distances of neighbors accordingly

Towards parallel shortest path

- Keep track of *tentative distance* per vertex
  - The distance of the shortest known path
- Repeatedly,
  - Take some set of nodes
  - ~~Their tentative distances are now fixed~~
  - Look at their neighbors: we may have a shorter path to them
  - Update tentative distances of neighbors accordingly

Work may be redundant:
Later iterations may need to look at this node again.

Trade-off between parallelization and work overhead

Towards parallel shortest path

- Keep track of *tentative distance* per vertex
  - The distance of the shortest known path
- Repeatedly,
  - Take some set of nodes
  - ~~Their tentative distances are now fixed~~
  - Look at their neighbors: we may have a shorter path to them
  - Update tentative distances of neighbors accordingly

What set?

Only the first unvisited node:
Dijkstra's algorithm

All nodes:
Bellman-Ford

Δ-stepping

- *Delta-stepping* is a parallelisable single-source shortest path algorithm
  - Algorithm stores an array of buckets B
  - Nodes are grouped by tentative distance in *buckets*
  - The range of distances in a bucket is parameter Δ (Greek letter Delta)
  - Bucket $B[i]$ stores the set of unsettled nodes v with

    $$i \cdot \Delta \leq \mathrm{tent}(v) < (i+1) \cdot \Delta$$

https://www.sciencedirect.com/science/article/pii/S0196677403000762

Towards parallel shortest path

- Keep track of *tentative distance* per vertex
  - The distance of the shortest known path
- Repeatedly,
  - Take all nodes from the <u>first non-empty bucket</u>
  - ~~Their tentative distances are now fixed~~
  - *Find requests:* In parallel, look at their neighbors: we may have a shorter path to them
  - *Relax requests:* In parallel, update tentative distances of neighbors accordingly

Towards parallel shortest path

- Keep track of *tentative distance* per vertex
  - The distance of the shortest known path
- Repeatedly,
  - Take all nodes from the first non-empty bucket
  - ~~Their tentative distances are now fixed~~
  - *Find requests:* In parallel, look at their neighbors: we may have a shorter path to them
  - *Relax requests:* In parallel, update tentative distances of neighbors accordingly

This could add some of the same nodes back into the same bucket, or new nodes into this bucket.

Light and heavy edges

- A node may repeatedly be in the same bucket.
- This gives redundant work: we repeatedly look at its neighbors.
- To reduce this, we handle *light* and *heavy* edges separately.
  - Light edges ($c(e) \leq \Delta$) may cause that nodes are added back to the same bucket.
  - Heavy edges ($c(e) > \Delta$) can only affect later buckets.

Towards parallel shortest path

- Repeatedly,
  - Find index i of the first non-empty bucket.
  - Repeatedly handle all outgoing light edges from nodes $B[i]$:
    - Remove all nodes from $B[i]$
    - Find requests of light edges
    - Relax requests
    - Keep track of all nodes that have been in this bucket
  - When the bucket remains empty, handle all outgoing heavy edges of nodes that have been in $B[i]$:
    - Find requests of light edges
    - Relax requests

Towards parallel shortest path

- Repeatedly,
  - Find index i of the first non-empty bucket.
  - Repeatedly handle all outgoing light edges from nodes $B[i]$:
    - Remove all nodes from $B[i]$
    - Find requests of light edges
    - Relax requests
    - Keep track of all nodes that have been in this bucket
  - When the bucket remains empty, handle all outgoing heavy edges of nodes that have been in $B[i]$:
    - Find requests of light edges
    - Relax requests

This could add some of the same nodes back into the same bucket, or new nodes into this bucket.

Initialisation

- All vertices have infinite tentative distance, except s which has distance zero
- All buckets are empty, except $B[0]$ which contains s

- How many buckets do we need?
  - Depends on the longest path
  - Or, on the longest edge: many buckets will be empty.
    - with a cyclic buffer we can reuse the buckets we no longer need.

Basic algorithm

```
foreach v ∈ V do tent(v) := ∞
relax(s, 0);                                   (* Insert source node with distance 0     *)
while ¬isEmpty(B) do                           (* A phase: Some queued nodes left (a) *)
    i := min{j ⩾ 0: B[j] ≠ ∅}                  (* Smallest nonempty bucket (b) *)
    R := ∅                                     (* No nodes deleted for bucket B[i] yet     *)
    while B[i] ≠ ∅ do                          (* New phase (c) *)
        Req := findRequests(B[i], light)       (* Create requests for light edges (d) *)
        R := R ∪ B[i]                          (* Remember deleted nodes (e) *)
        B[i] := ∅                              (* Current bucket empty     *)
        relaxRequests(Req)                     (* Do relaxations, nodes may (re)enter B[i] (f) *)
    Req := findRequests(R, heavy)              (* Create requests for heavy edges (g) *)
    relaxRequests(Req)                         (* Relaxations will not refill B[i] (h) *)

Function findRequests(V′, kind : {light, heavy}) : set of Request
    return {(w, tent(v) + c(v, w)): v ∈ V′ ∧ (v, w) ∈ E_kind)}

Procedure relaxRequests(Req)
    foreach (w, x) ∈ Req do relax(w, x)

Procedure relax(w, x)                          (* Insert or move w in B if x < tent(w) *)
    if x < tent(w) then
        B[⌊tent(w)/Δ⌋] := B[⌊tent(w)/Δ⌋] \ {w}  (* If in, remove from old bucket *)
        B[⌊x      /Δ⌋] := B[⌊x      /Δ⌋] ∪ {w}  (* Insert into new bucket *)
        tent(w) := x
```

Example

On blackboard

Utilities

fgl

- Inductive representation for graph structures
  - Data.Graph.Inductive.Graph contains functions for querying the given graph
    - Number of nodes / vertices in the graph
    - Return the in / out edges for the given node

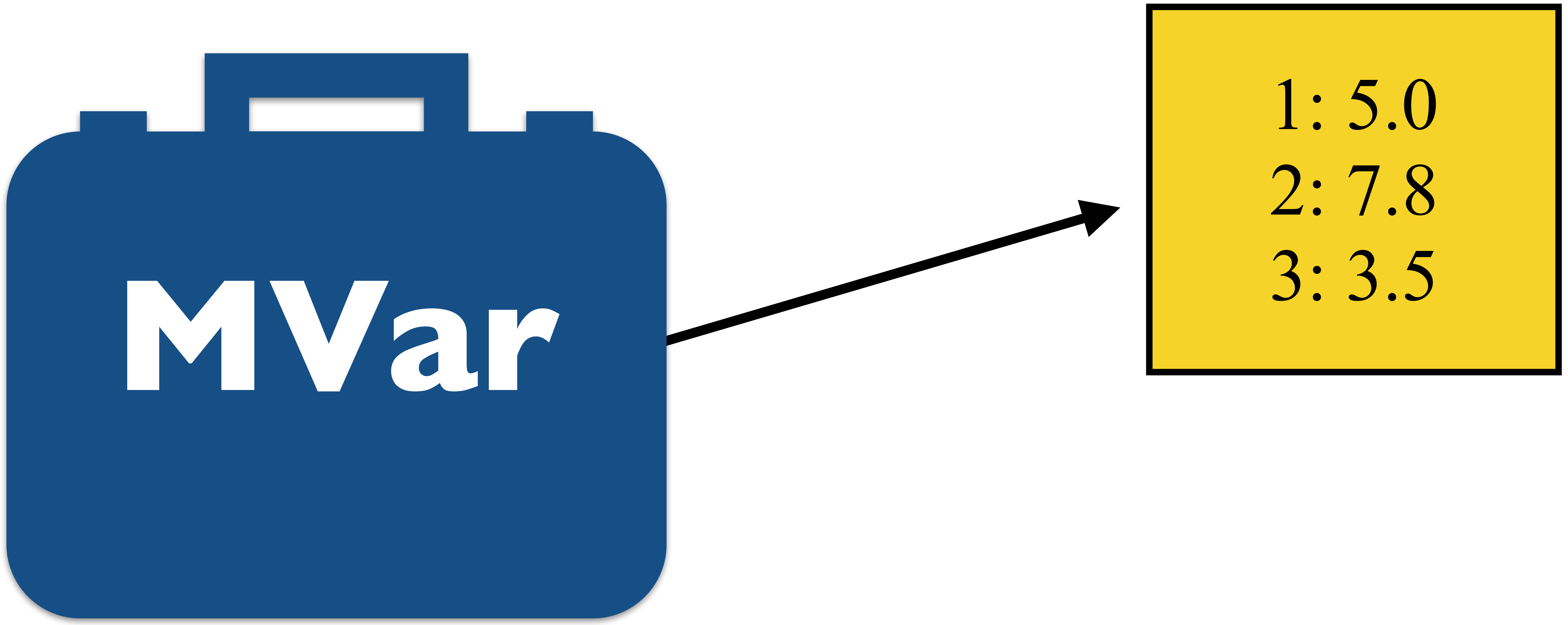
https://hackage.haskell.org/package/fgl

containers

- Assorted immutable data structures
  - [Int]Map (dictionary), [Int]Set, etc.
  - Put in an `IORef`/`MVar`/etc. to create a simple (non-concurrent) mutable container

https://hackage.haskell.org/package/containers

MVar (Map Int Float)



MVar (Map Int Float)

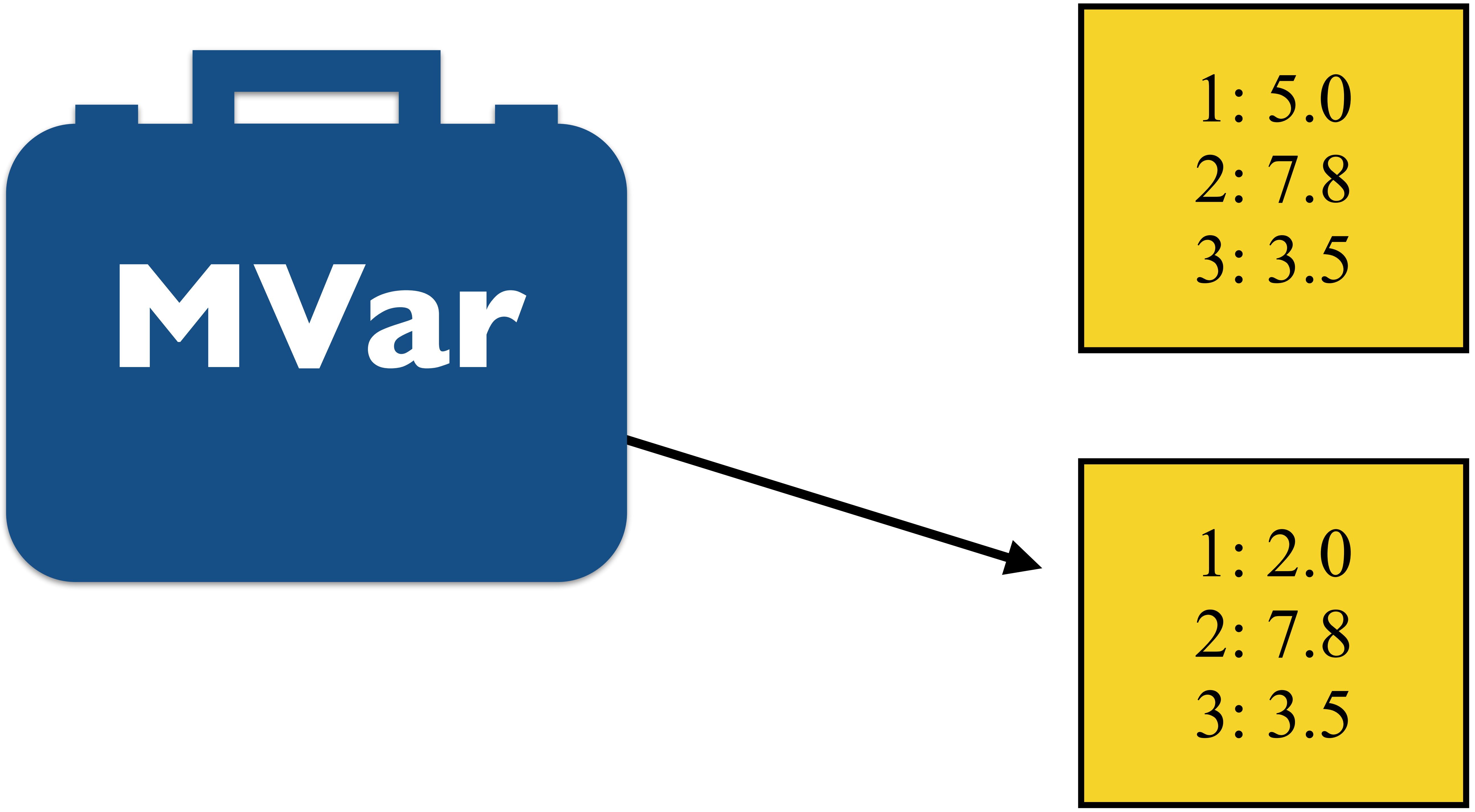
1: 5.0
2: 7.8
3: 3.5

MVar (Map Int Float)

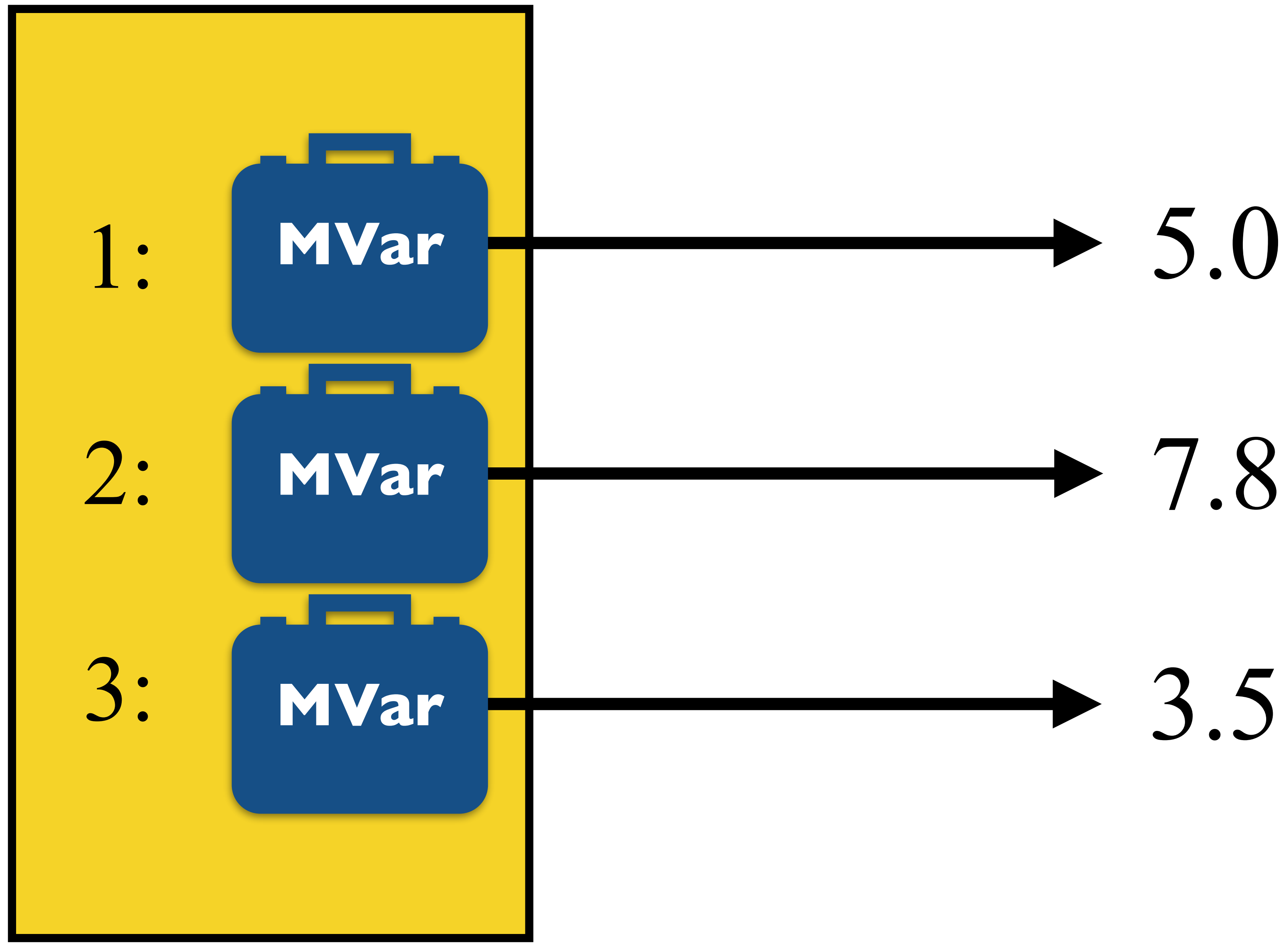
1: 5.0
2: 7.8
3: 3.5

1: 2.0
2: 7.8
3: 3.5

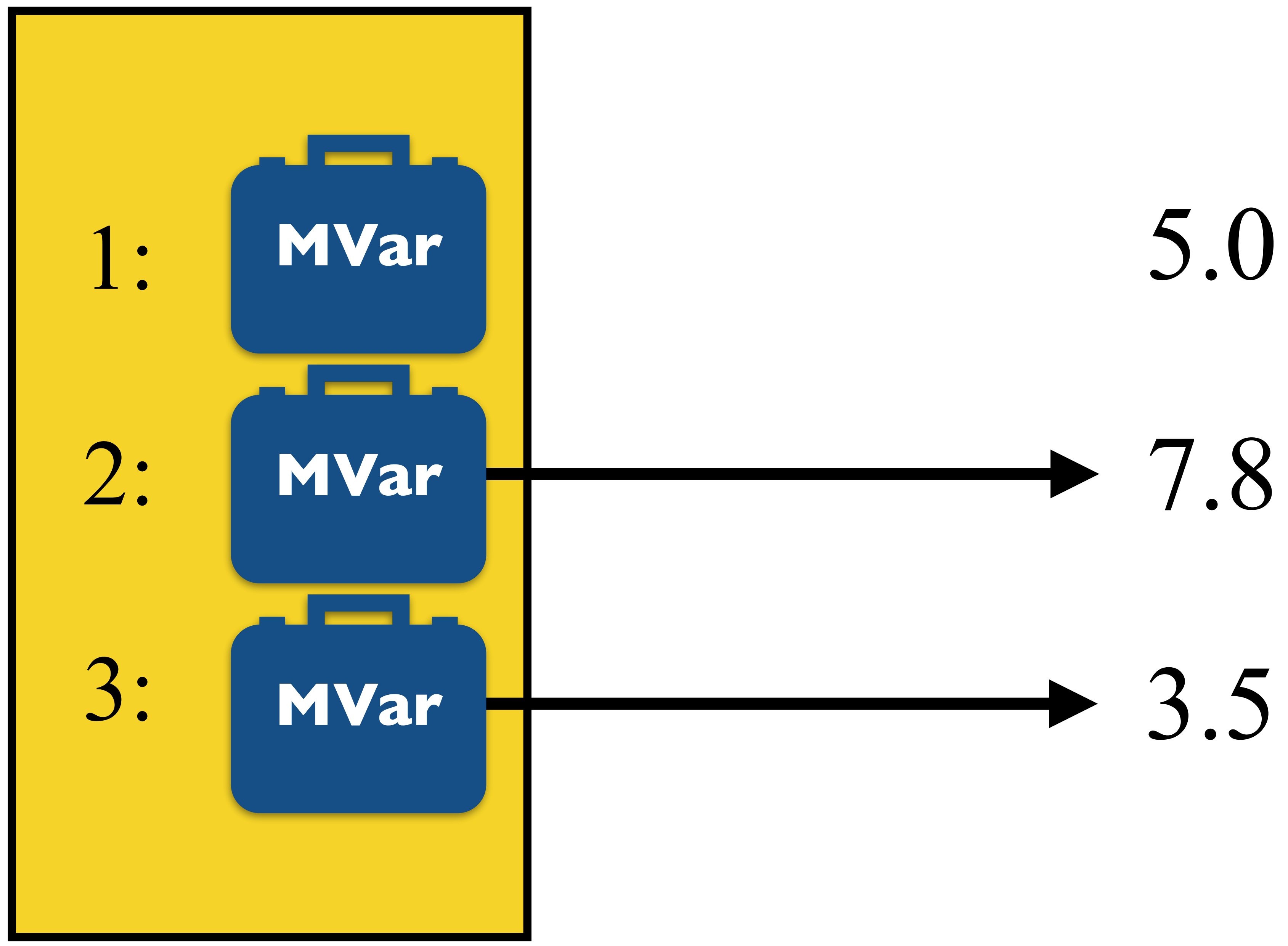
MVar (Map Int Float)



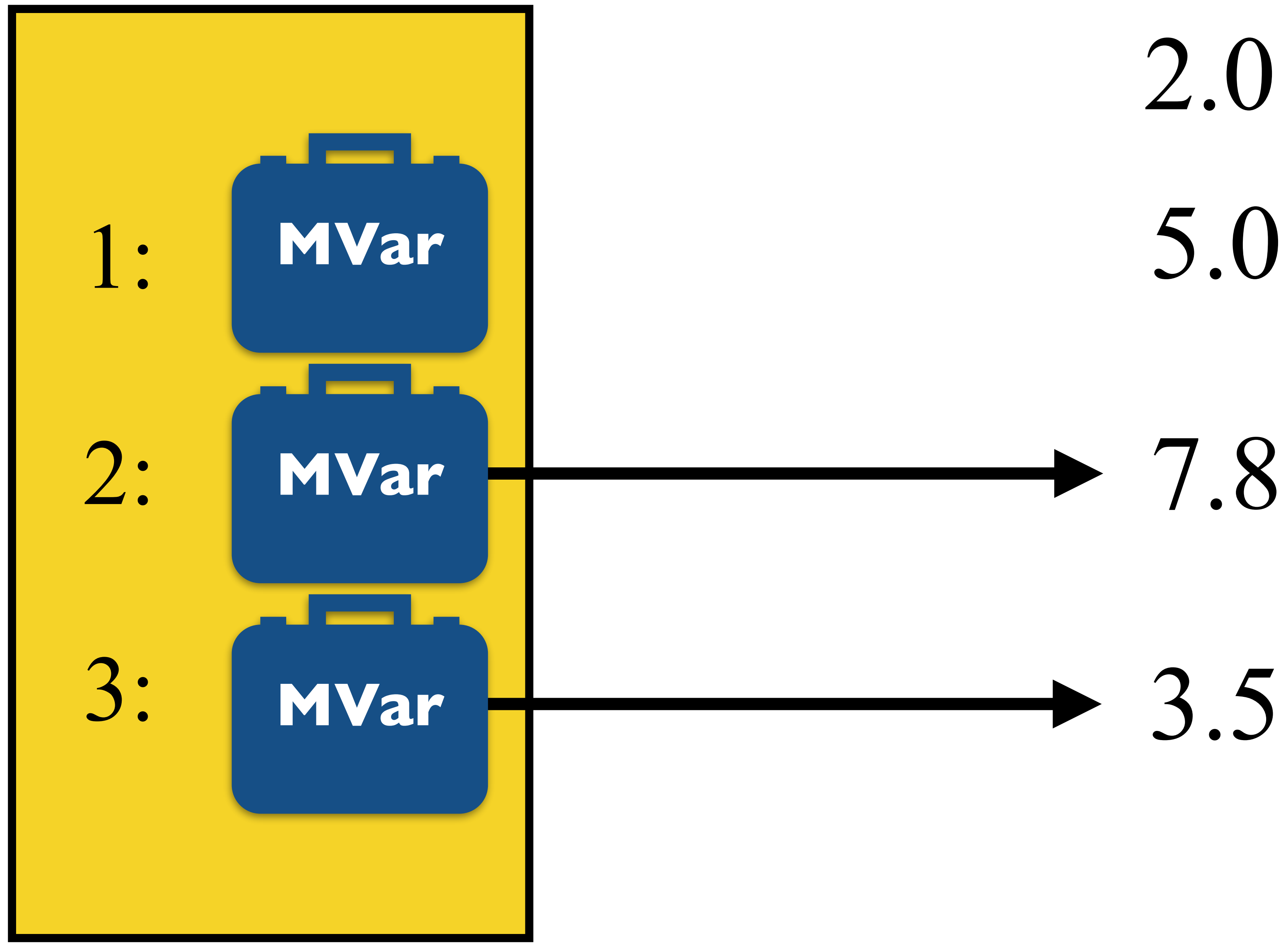
Map Int (MVar Float)



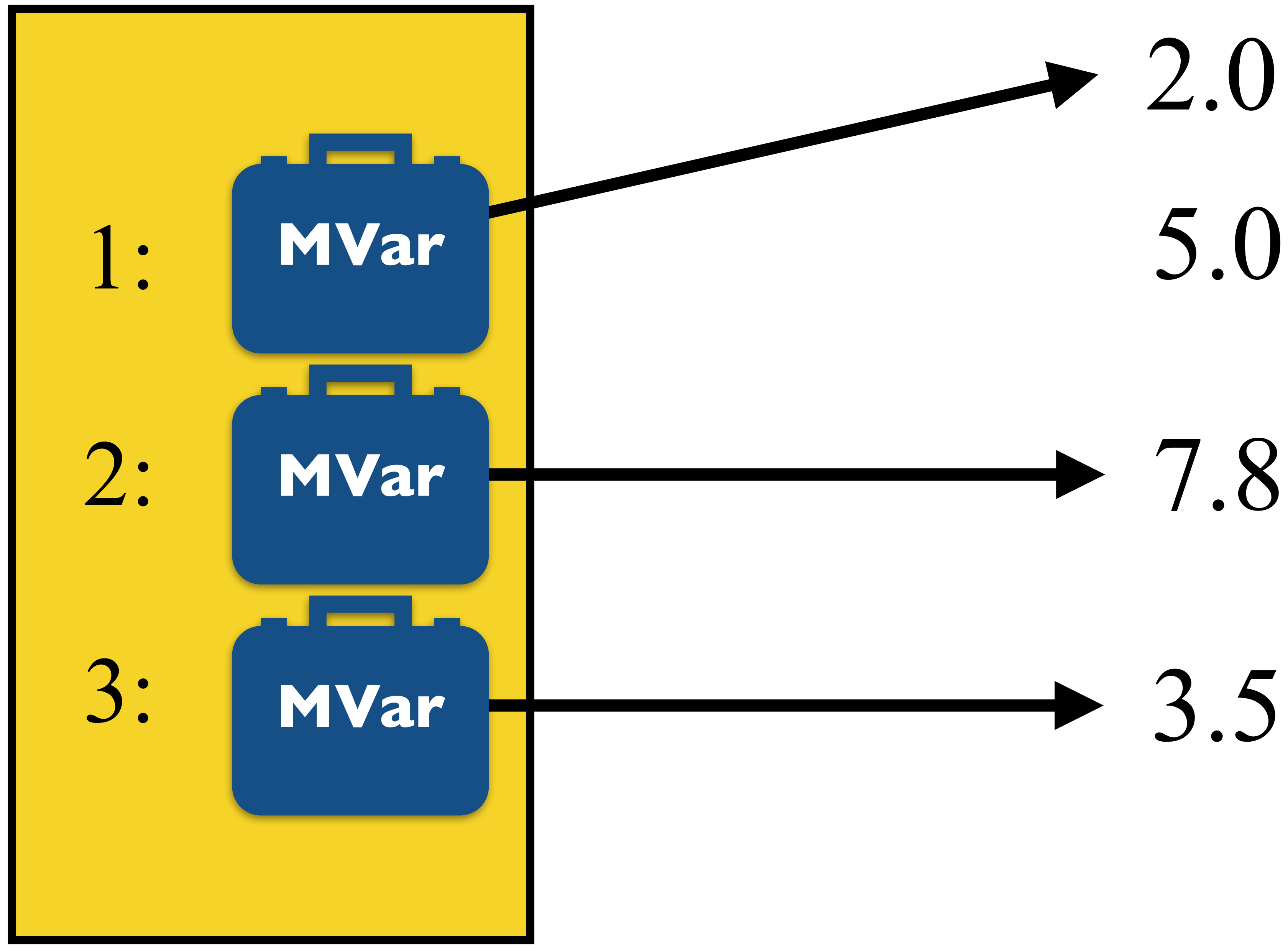
Map Int (MVar Float)



Map Int (MVar Float)

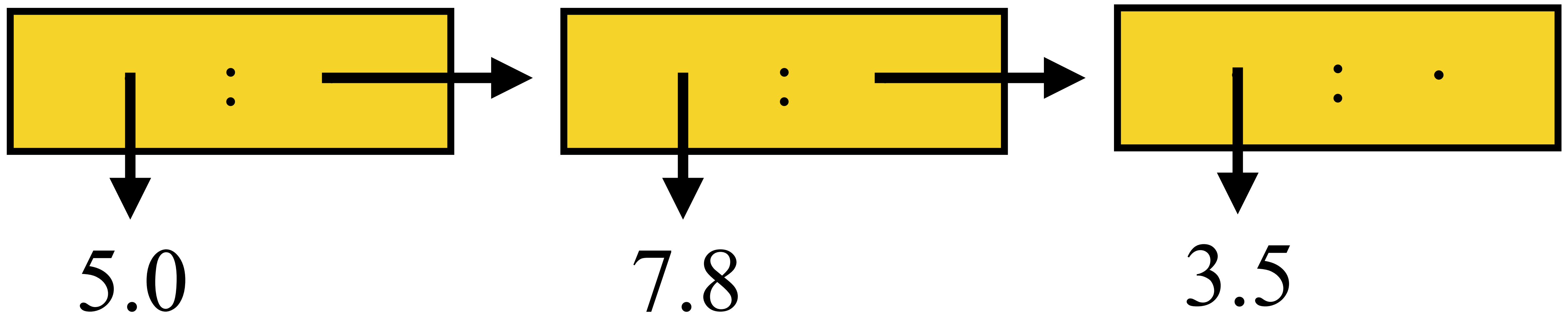


Map Int (MVar Float)

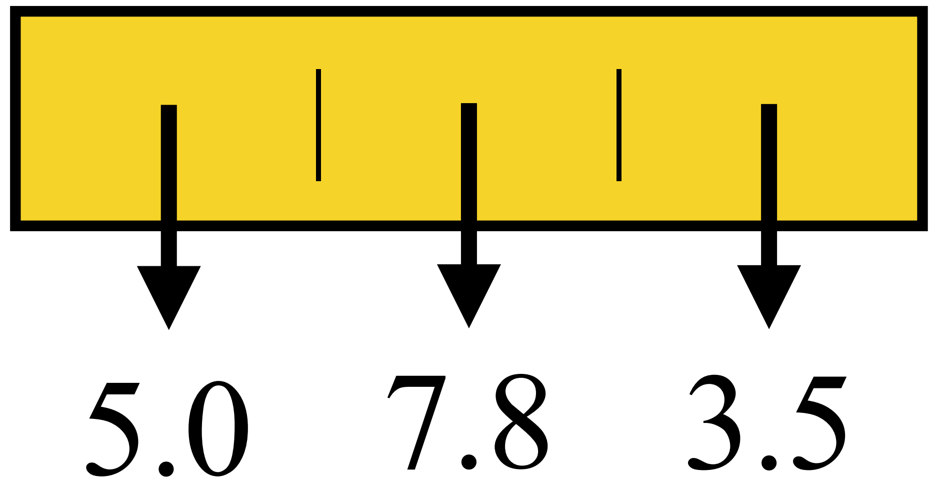


Lists vs Vectors

- Lists in Haskell are linked-lists:
  - (:) and tail are O(1),
  - Indexing is O(n)



- Vectors are stored as arrays, with pointers to the values:



- Unboxed vectors store values instead of pointers in arrays:

vector

- (Un)boxed (im)mutable int-indexed arrays
  - Provides arrays in several flavours (i.e. underlying representation), but all with the same API
  - Data.Vector[.Mutable]
    - Boxed vectors (i.e. array of pointers) that can hold any structure
  - Data.Vector.Storable[.Mutable]
    - Unboxed vectors (i.e. array of values) that can hold only Storable (i.e. primitive) values
    - You can get a pointer directly to the array elements: useful for low-level atomic instructions
  - You can convert between different representations

https://hackage.haskell.org/package/vector

Conclusion

- The long, sequential, critical path was a problem.
- Trade-off between work overhead and parallelism:
  - We do some redundant work,
  - but when done properly, we will get a faster algorithm!
- Separation in light and heavy edges reduces work overhead.
- It is up to you to determine where the parallelism in the algorithm is (easy) and how to exploit this (hard)
- You are free to use IORefs, MVars, STM, mutable vectors, …

Note about P1

- If you didn’t add your name and student number to the repository, then post a comment with them in the feedback pull request.

Photo by James Barker