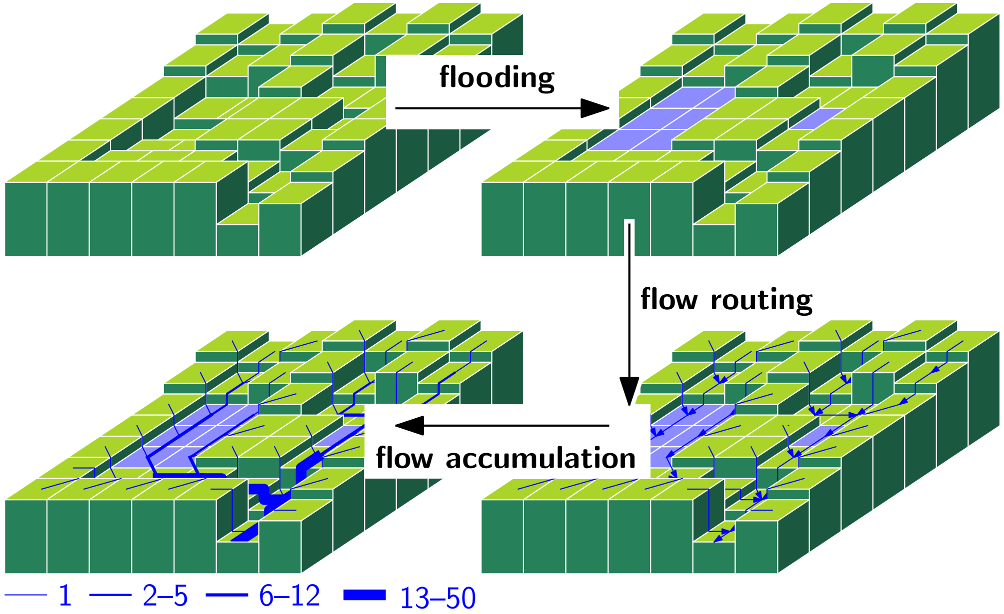
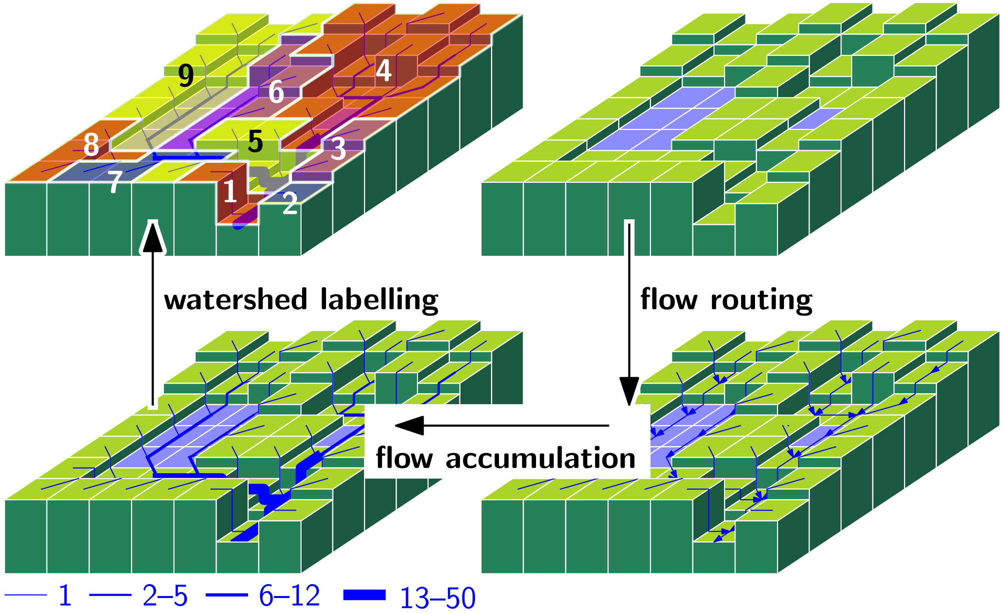
Drainage network analysis

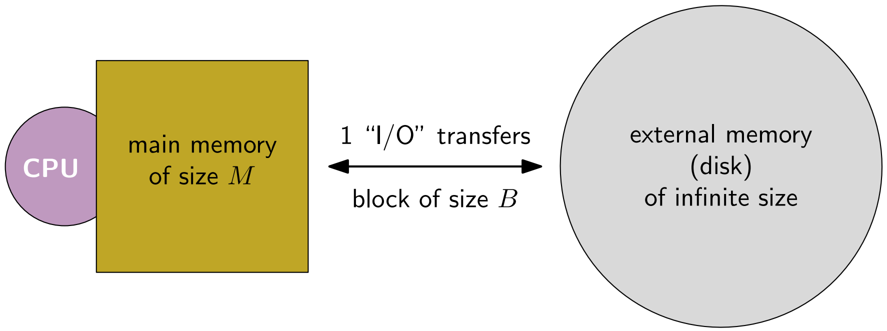


compute for each cell c, from how many cells water passes through c

Drainage network analysis



Analysing I/O-efficiency

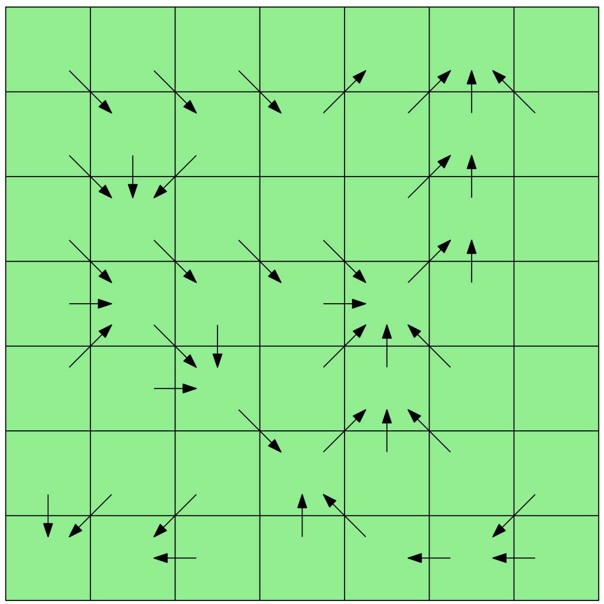


CPU only operates on data in main memory (for free)

I/O-efficiency = number of I/O's as function of M, B, and grid size N
(sometimes assume $M \geq c \cdot B^2$)

Flow accumulation: naïve algorithm

Goal: compute flow accumulation for each cell c
=
#cells from which water passes through c
=
size of tree rooted at c



Flow accumulation: naïve algorithm

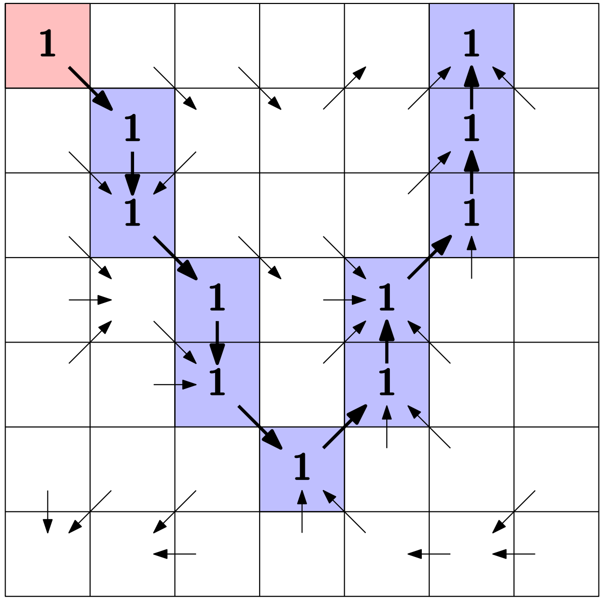
Goal: compute flow accumulation for each cell c

=

#cells from which water passes through c

=

size of tree rooted at c



Flow accumulation: naïve algorithm

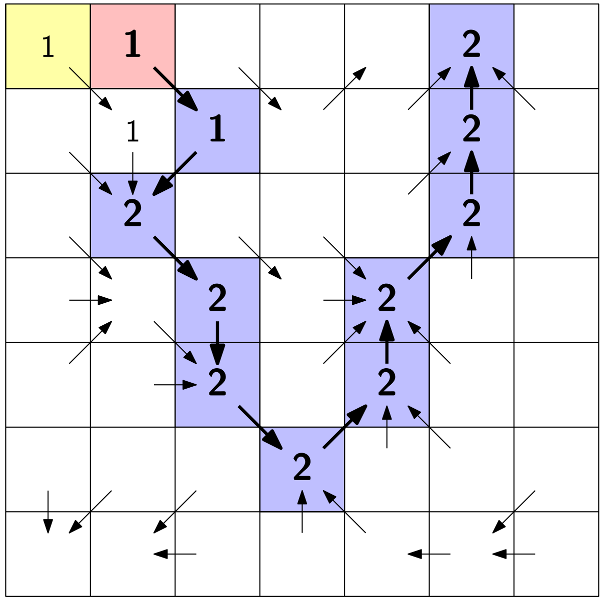
Goal: compute flow accumulation for each cell c

=

#cells from which water passes through c

=

size of tree rooted at c



Flow accumulation: naïve algorithm

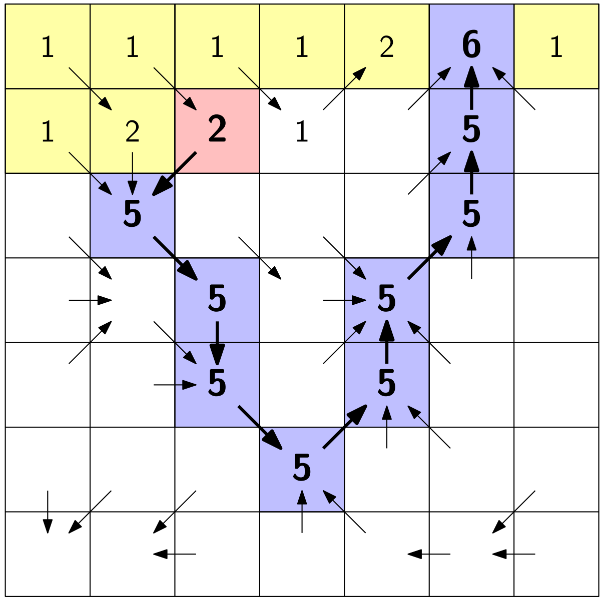
Goal: compute flow accumulation for each cell c

=

#cells from which water passes through c

=

size of tree rooted at c



Row-by-row scan

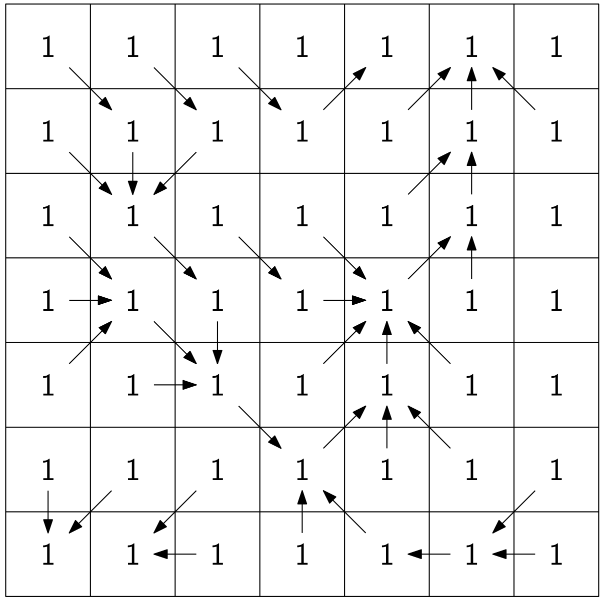
Goal: compute flow accumulation for each cell c

=

#cells from which water passes through c

=

size of tree rooted at c



Row-by-row scan

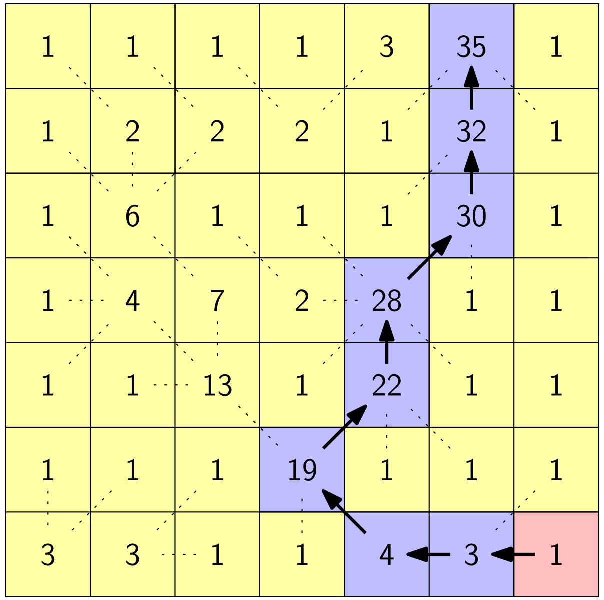
Goal: compute flow accumulation for each cell c

$=$

#cells from which water passes through c

$=$

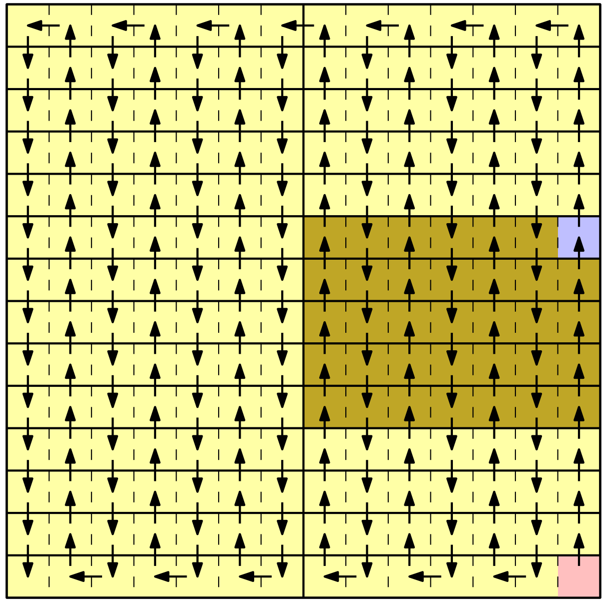
size of tree rooted at c



Running time: $\Theta(N)$

N = #cells in grid

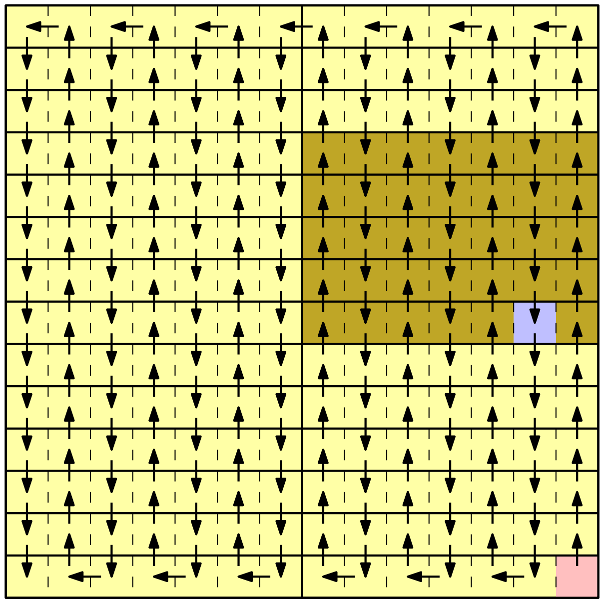
Row-by-row scan



Row-by-row scan

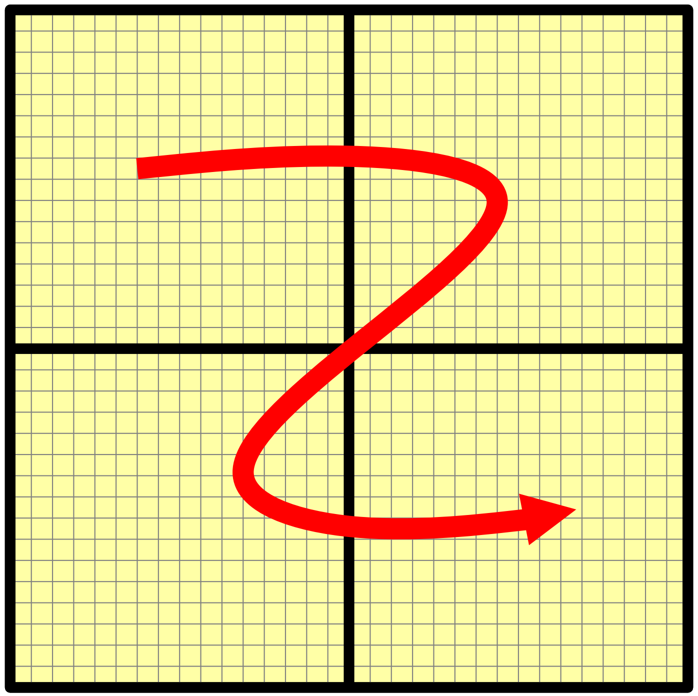
$\Theta(N)$ I/O's in the worst case

$\approx$ 1 year for 28 GB grid

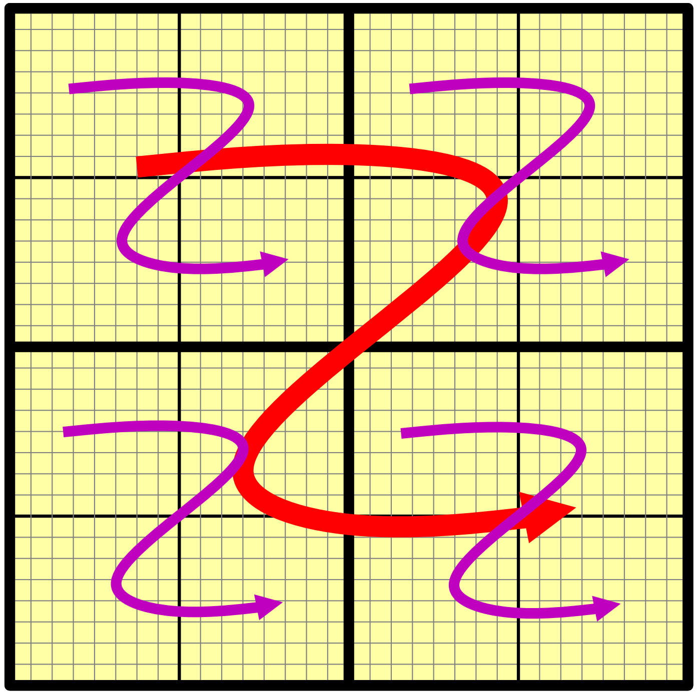


N = #cells in grid

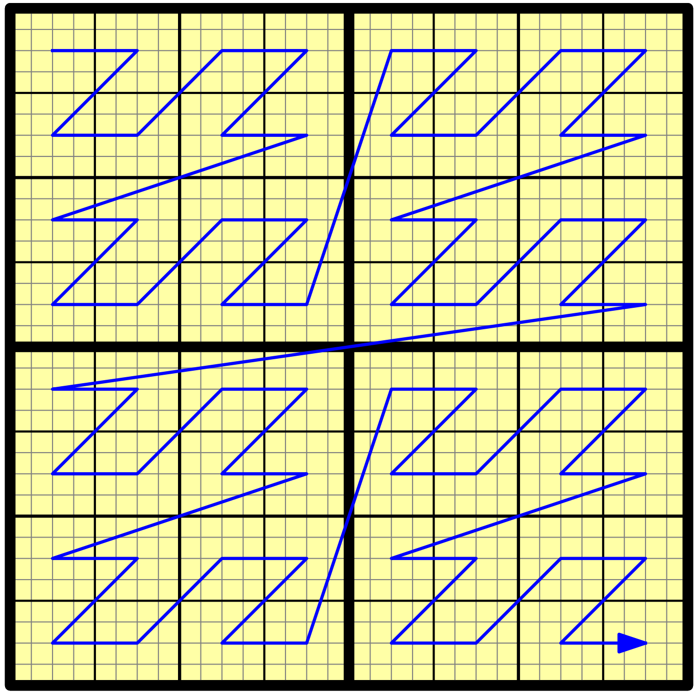
Z-order scan



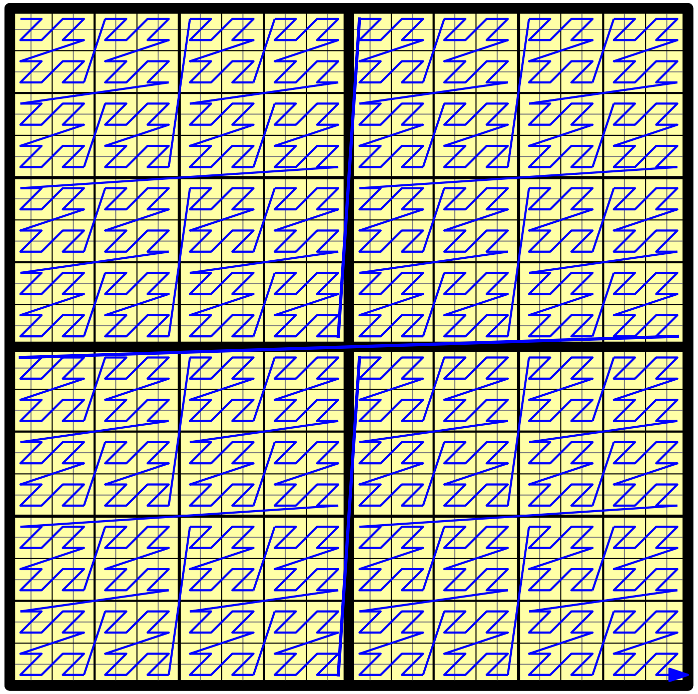
Z-order scan



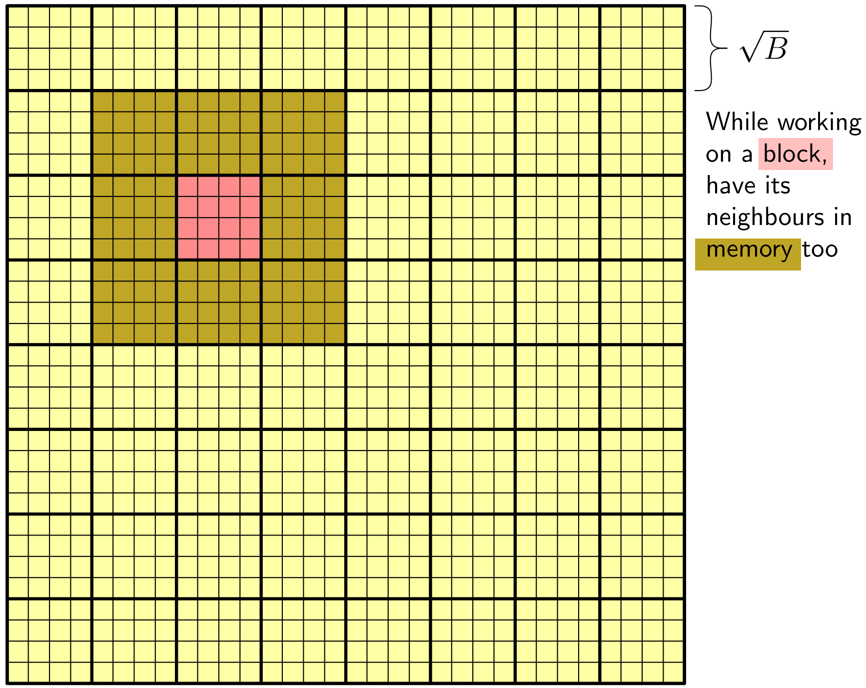
Z-order scan



Z-order scan

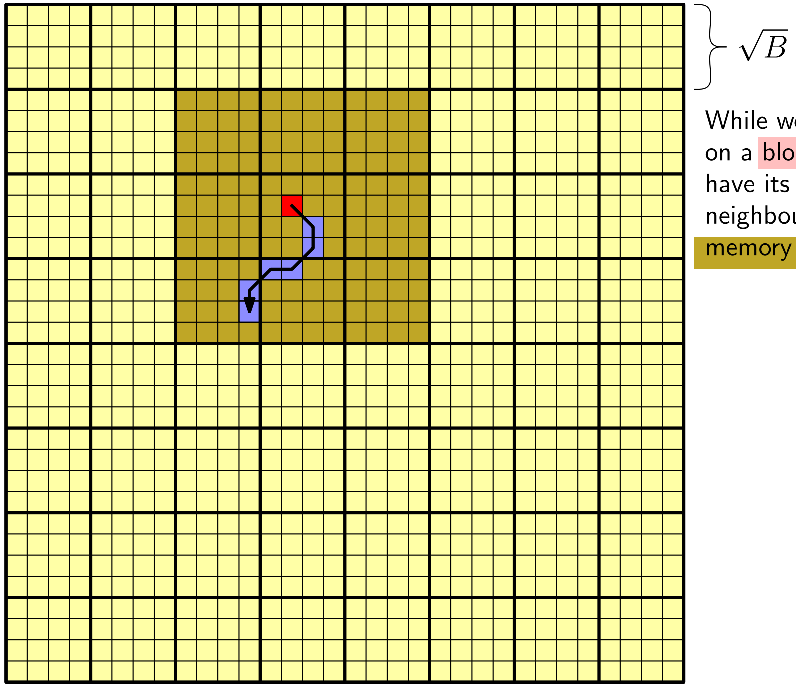


Z-order scan on Z-order file



B = #bytes in one I/O

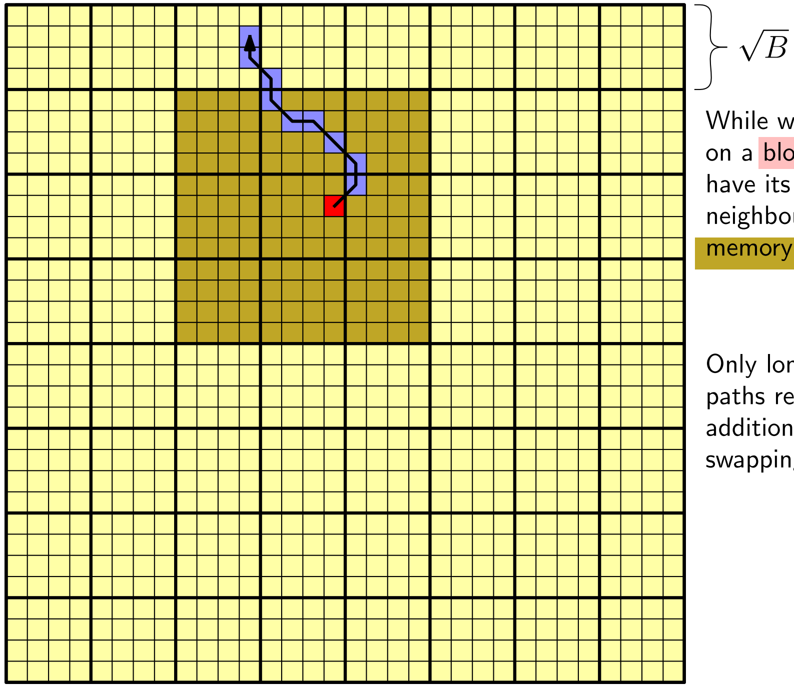
Z-order scan on Z-order file



While working on a block, have its neighbours in memory too

B = #bytes in one I/O

Z-order scan on Z-order file

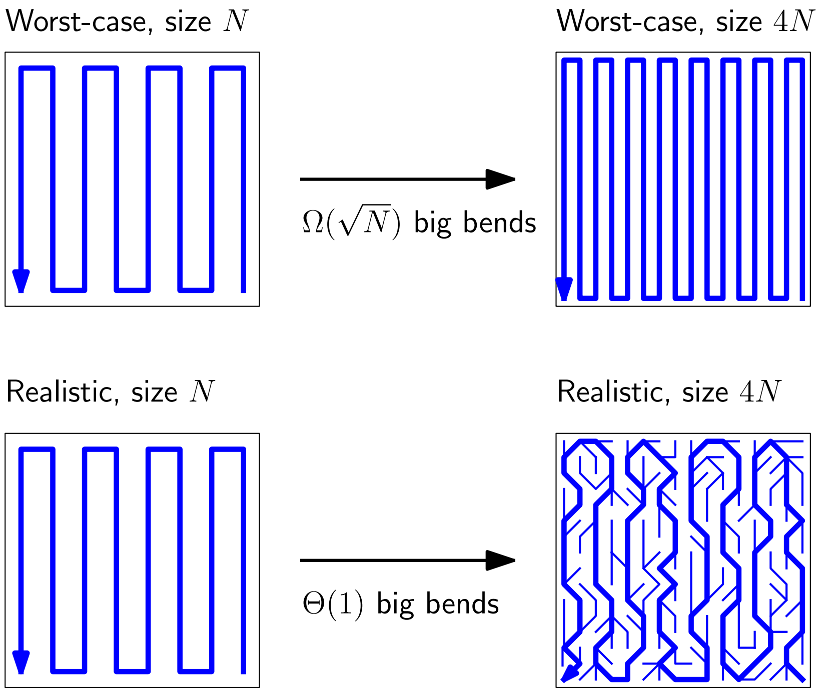


While working on a block, have its neighbours in memory too

Only long paths require additional swapping

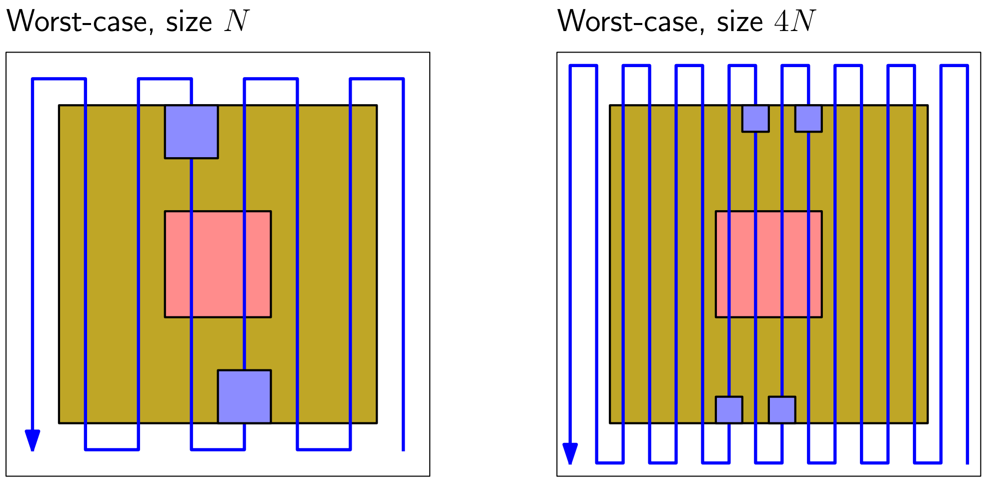
B = #bytes in one I/O

Worst-case terrains vs. real terrains



N = #cells in grid

Worst-case terrains vs. real terrains



Q' = Q scaled by factor 3.

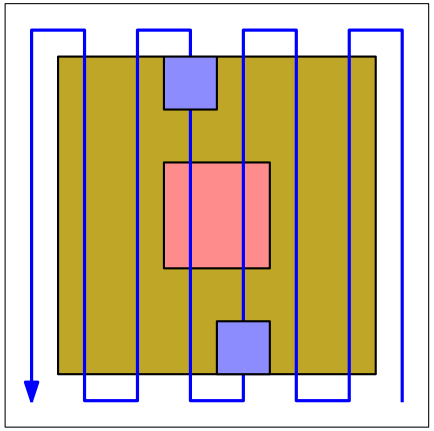
Far cells of Q: cells on boundary of Q' where water from Q collects.

In the worst case, maximum number of far cells grows with resolution.

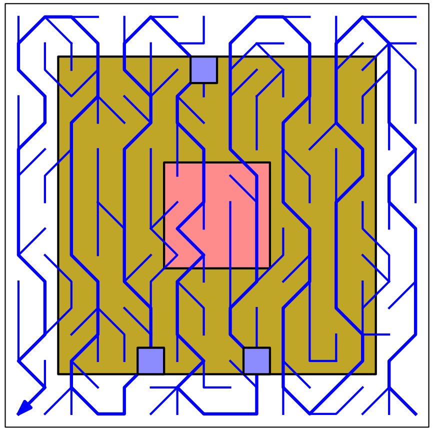
N = #cells in grid

Worst-case terrains vs. real terrains

Realistic, size N



Realistic, size $4N$



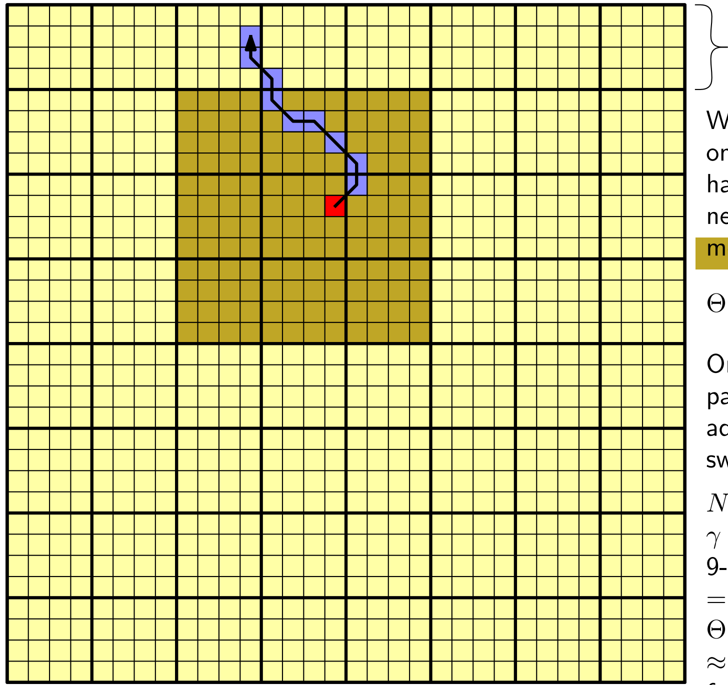
Q' = Q scaled by factor 3.

Far cells of Q: cells on boundary of Q' where water from Q collects.

Confluence assumption: number of far cells for any square $Q \leq$ constant γ

N = #cells in grid

Z-order scan on Z-order file



$\sqrt{B}$

While working on a block, have its neighbours in memory too

$\Theta(N/B)$ I/Os

Only long paths require additional swapping

N/B blocks $\times$ γ swaps $\times$ 9-block window $=$ $\Theta(N/B)$ I/Os $\approx$ few hours

N = #cells in grid B = #bytes in one I/O

Flow accumulation by scanning in practice

algorithm	file order	*theoretical analysis* worst case	‘realistic’	bytes per cell	*experiments* time (mins)
row-by-row scan	row by row	$O(N)$	$O(N/\sqrt{B})$	tenthousands	111
Z-order scan	row by row	$O(N)$	$O(N/B)^*$	thousands	
Z-order scan	Z-order	$O(N/\sqrt{B})$	$O(N/B)$	hundreds	41

bytes of disk I/O per cell calculated based on: $N = 2^{32}$, $M = 1$ GB, $B = 16$ to 64KB

time: 3 GHz Pentium, one disk for data + scratch, $N = 3.5 \cdot 10^9$ (Neuse), $M = 1$ GB

*) needs tall cache: $M \geq cB^2$

Easy implementation:

- needs efficient conversion (*row nr.*, *column nr.*) $\leftrightarrow$ index in Z-order
- Z-order scan $\rightarrow$ good caching by OS, no need to tune to hardware / implement I/O-control

Z-order-traversal has many other applications, e.g.:

- I/O-efficient matrix operations
- I/O-efficient algorithms and data structures for geographic maps

N = #cells in grid B = #bytes in one I/O

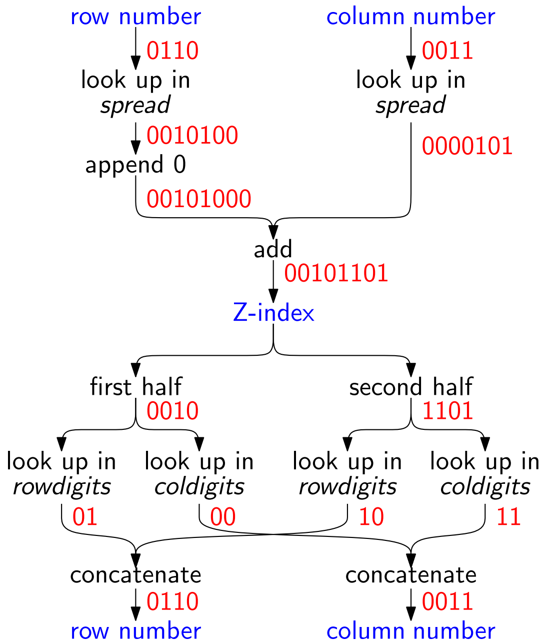
(row, column) $\leftrightarrow$ Z-index

	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
0 0 0	000000 (0)	000001 (1)	000100 (4)	000101 (5)	010000 (16)	010001 (17)	010100 (20)	010101 (21)
0 0 1	000010 (2)	000011 (3)	000110 (6)	000111 (7)	010010 (18)	010011 (19)	010110 (22)	010111 (23)
0 1 0	001000 (8)	001001 (9)	001100 (12)	001101 (13)	011000 (24)	011001 (25)	011100 (28)	011101 (29)
0 1 1	001010 (10)	001011 (11)	001110 (14)	001111 (15)	011010 (26)	011011 (27)	011110 (30)	011111 (31)
1 0 0	100000 (32)	100001 (33)	100100 (36)	100101 (37)	110000 (48)	110001 (49)	110100 (52)	110101 (53)
1 0 1	100010 (34)	100011 (35)	100110 (38)	100111 (39)	110010 (50)	110011 (51)	110110 (54)	110111 (55)
1 1 0	101000 (40)	101001 (41)	101100 (44)	101101 (45)	111000 (56)	111001 (57)	111100 (60)	111101 (61)
1 1 1	101010 (42)	101011 (43)	101110 (46)	101111 (47)	111010 (58)	111011 (59)	111110 (62)	111111 (63)

(row, column) ↔ Z-index

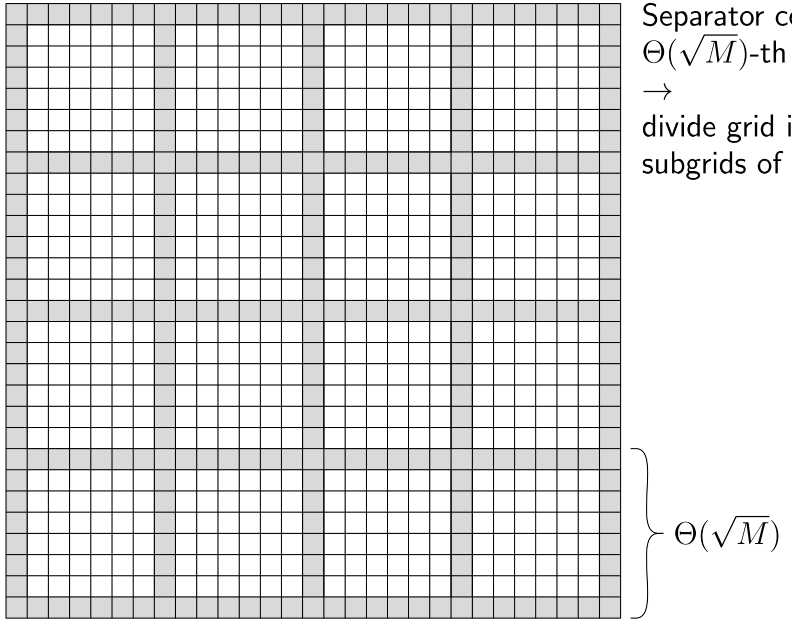
Quick conversion through look-ups in tables of size $\sqrt{N}$ (example: $\sqrt{N} = 16$)

	spread	*rowdigits*	*coldigits*
0000	0000000	00	00
0001	0000001	00	01
0010	0000100	01	00
0011	0000101	01	01
0100	0010000	00	10
0101	0010001	00	11
0110	0010100	01	10
0111	0010101	01	11
1000	1000000	10	00
1001	1000001	10	01
1010	1000100	11	00
1011	1000101	11	01
1100	1010000	10	10
1101	1010001	10	11
1110	1010100	11	10
1111	1010101	11	11



can be adapted to non-square grids

Flow accumulation: separator-based algorithm



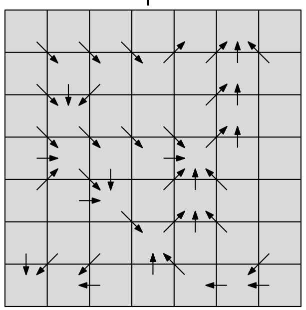
Separator cells: each $\Theta(\sqrt{M})$-th row/column
→
divide grid into $\Theta(N/M)$ subgrids of size $\Theta(M)$

N = #cells in grid M = main memory size

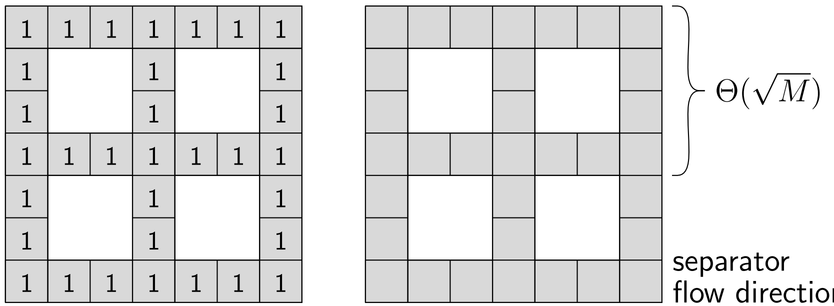
Flow accumulation: separator-based algorithm

1. move flow from interior of subgrids to separators

input



separator flow accumul.



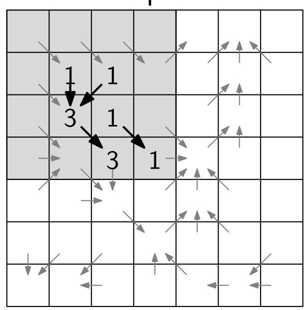
separator flow directions

N = #cells in grid M = main memory size

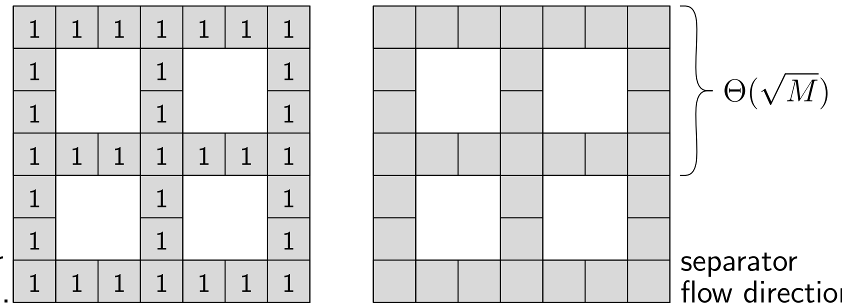
Flow accumulation: separator-based algorithm

1. move flow from interior of subgrids to separators

input



separator flow accumul.

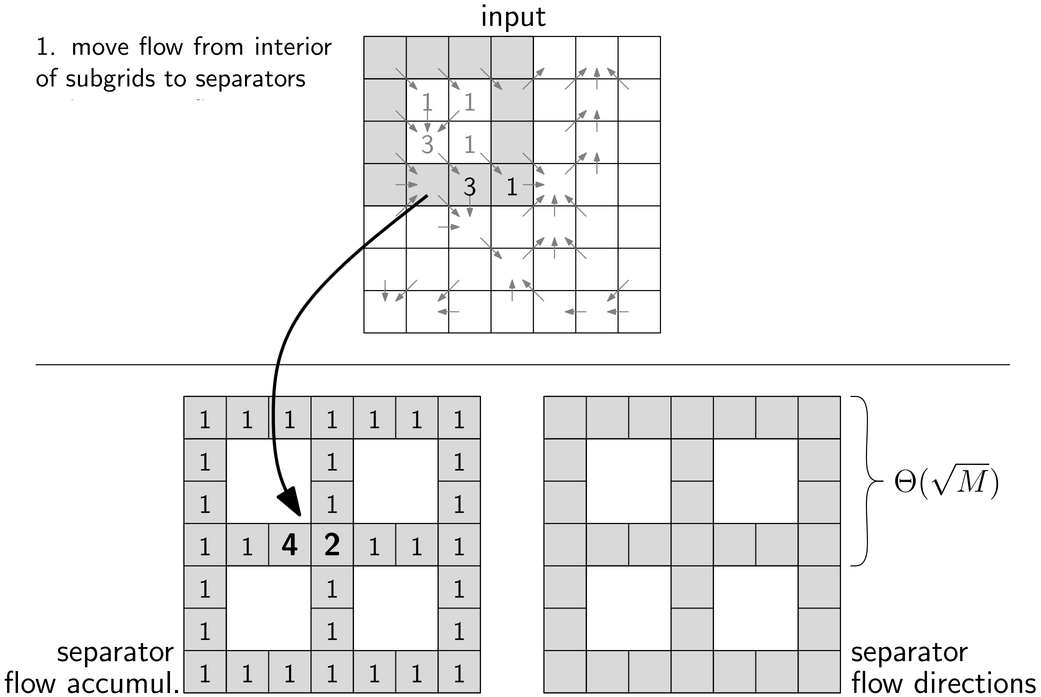


separator flow directions

N = #cells in grid M = main memory size

Flow accumulation: separator-based algorithm

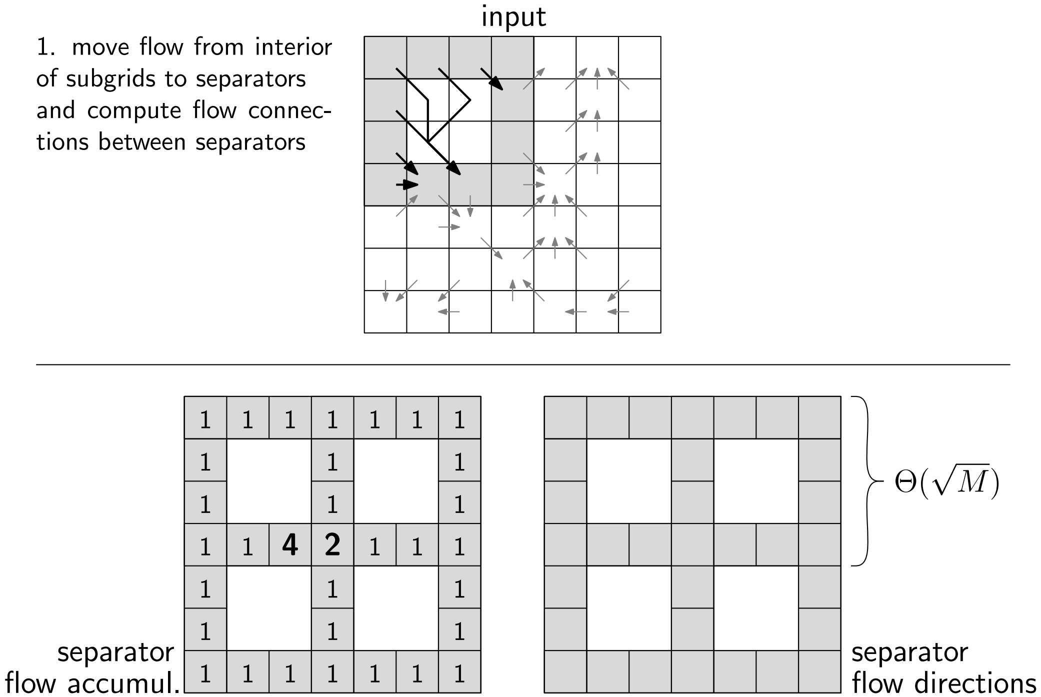
1. move flow from interior of subgrids to separators



N = #cells in grid M = main memory size

Flow accumulation: separator-based algorithm

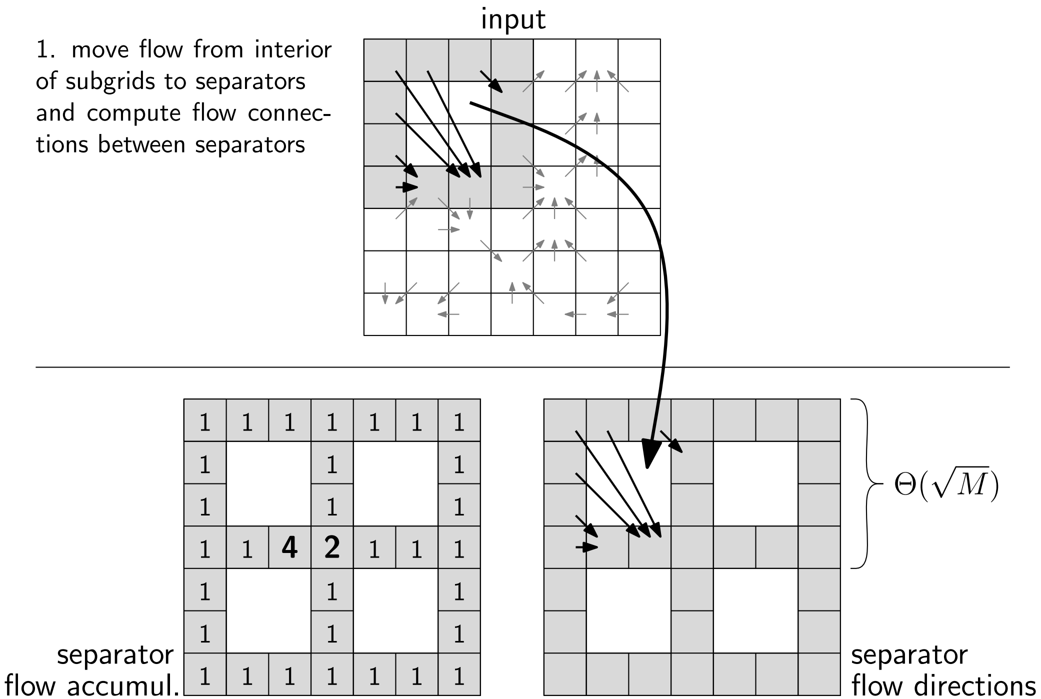
1. move flow from interior of subgrids to separators and compute flow connections between separators



N = #cells in grid M = main memory size

Flow accumulation: separator-based algorithm

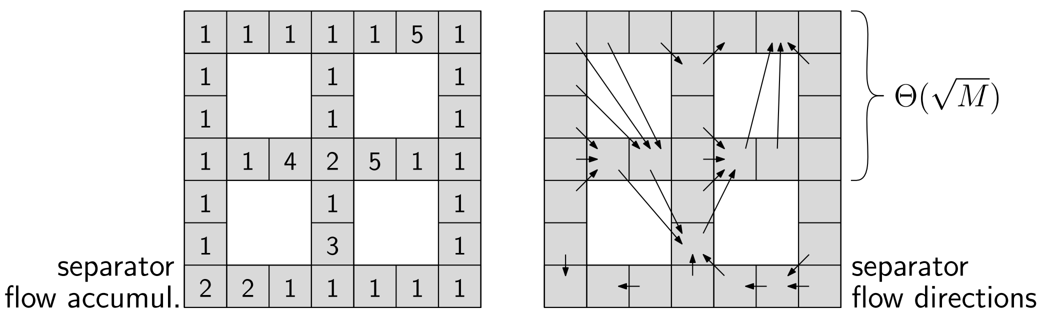
1. move flow from interior of subgrids to separators and compute flow connections between separators



N = #cells in grid M = main memory size

Flow accumulation: separator-based algorithm

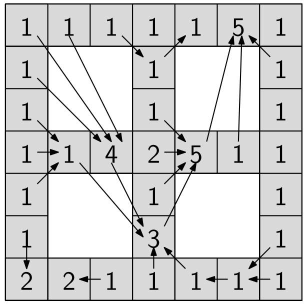
1. move flow from interior of subgrids to separators and compute flow connections between separators



N = #cells in grid M = main memory size

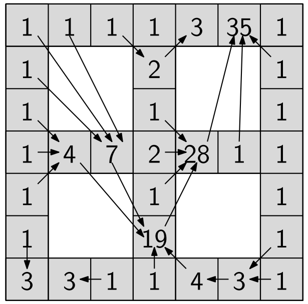
Flow accumulation: separator-based algorithm

1. move flow from interior of subgrids to separators and compute flow connections between separators
2. compute flow accumulation of separators



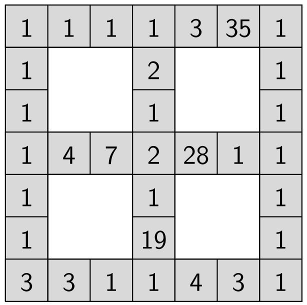
Flow accumulation: separator-based algorithm

1. move flow from interior of subgrids to separators and compute flow connections between separators
2. compute flow accumulation of separators



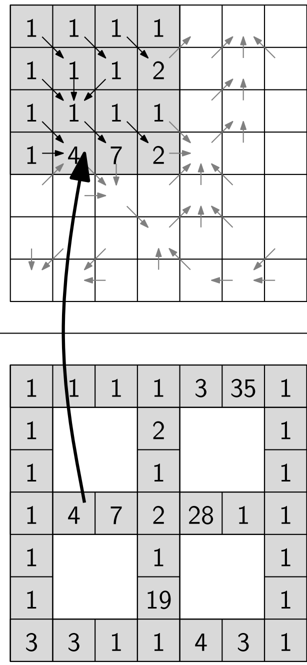
Flow accumulation: separator-based algorithm

1. move flow from interior of subgrids to separators and compute flow connections between separators
2. compute flow accumulation of separators



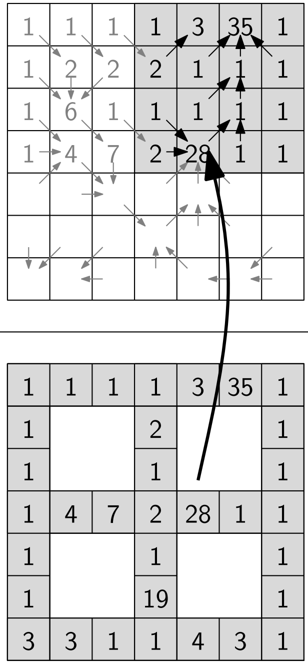
Flow accumulation: separator-based algorithm

1. move flow from interior of subgrids to separators and compute flow connections between separators
2. compute flow accumulation of separators
3. move flow from separators into subgrids



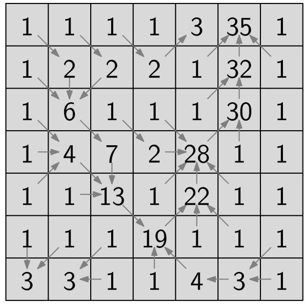
Flow accumulation: separator-based algorithm

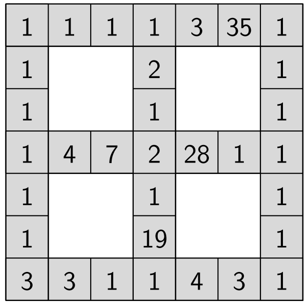
1. move flow from interior of subgrids to separators and compute flow connections between separators
2. compute flow accumulation of separators
3. move flow from separators into subgrids



Flow accumulation: separator-based algorithm

1. move flow from interior of subgrids to separators and compute flow connections between separators
2. compute flow accumulation of separators
3. move flow from separators into subgrids





Flow accumulation: separator-based algorithm

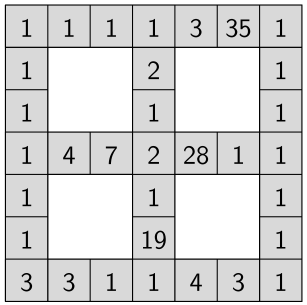
1. move flow from interior of subgrids to separators and compute flow connections between separators
2. compute flow accumulation of separators
3. move flow from separators into subgrids

1. $\Theta(N/M)$ subgrids $\times$ $\Theta(M/B + \sqrt{M}) =$ $\Theta(M/B) =$ $\Theta(N/B)$ I/O's
2. linear-time algo, input $\Theta(N/\sqrt{M}) = O(N/B)$
3. $\Theta(N/B)$ I/O's (like phase 1)

1. 1 byte of I/O per cell
2. no I/O in practice
3. 9 bytes of I/O per cell

Total: 10 bytes per cell if grid stored in Z-order

Total: 20 to 60 bytes if grid stored row by row (for $1/4 \leq M/B^2 \leq 4$)



N = #cells in grid M = main memory size B = #bytes in one I/O $M \geq cB^2$

Flow accumulation: Z-order scan versus separators

algorithm	file order	*theoretical analysis* worst case	'realistic'	bytes per cell	*experiments* time (mins)
row-by-row scan	row by row	$O(N)$	$O(N/\sqrt{B})$	tenthousands	111
Z-order scan	row by row	$O(N)$	$O(N/B)^*$	thousands	
Z-order scan	Z-order	$O(N/\sqrt{B})$	$O(N/B)$	hundreds	41
Easy implementation: no need to tune to hardware / implement I/O-control					
separator-based	row by row	$O(N/B)^*$		20 to 60	39
separator-based	Z-order	$O(N/B)$		10	should try!
Implementation must explicitly adapt subgrid size to available memory M					

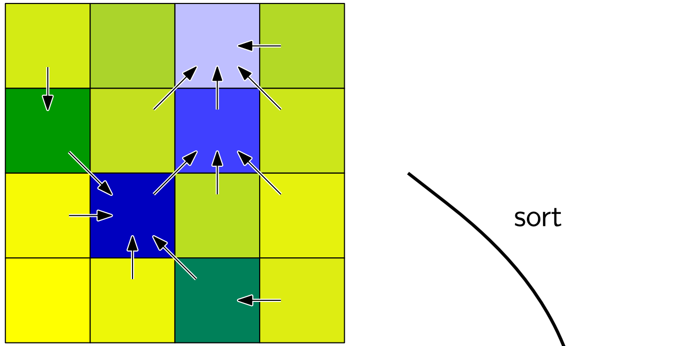
bytes of disk I/O per cell calculated based on: $N = 2^{32}$, $M = 1$ GB, $B = 16$ to 64KB
time: 3 GHz Pentium, one disk for data + scratch, $N = 3.5 \cdot 10^9$ (Neuse), $M = 1$ GB
*) needs tall cache: $M \geq cB^2$

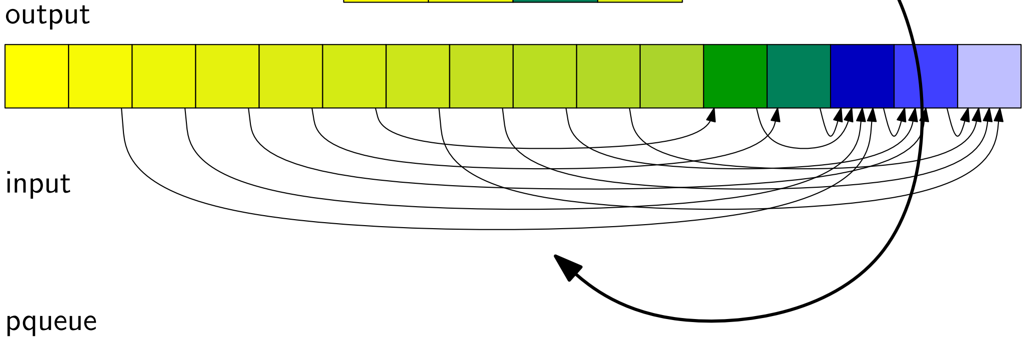
Other applications of separators:
minimum spanning trees & flooding; BFS & flow routing; single-source shortest paths.

N = #cells in grid M = main memory size B = #bytes in one I/O

Time-forward processing (Arge et al.)

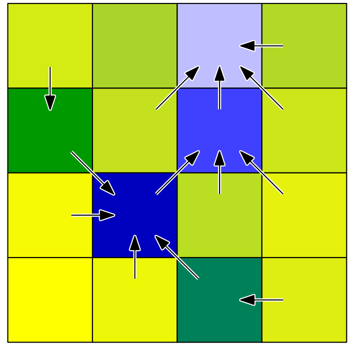
Goal: compute flow accumulation for each cell c
=
#cells from which water passes through c
=
size of tree rooted at c

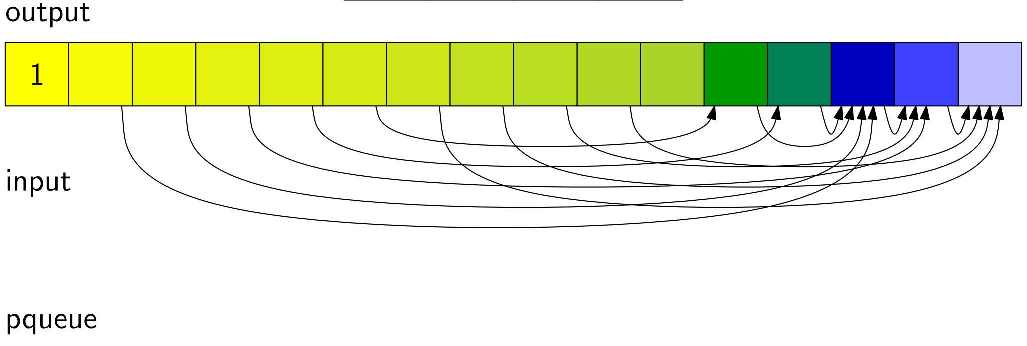




Time-forward processing (Arge et al.)

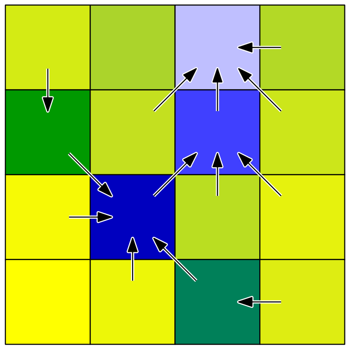
Goal: compute flow accumulation for each cell c
=
#cells from which water passes through c
=
size of tree rooted at c

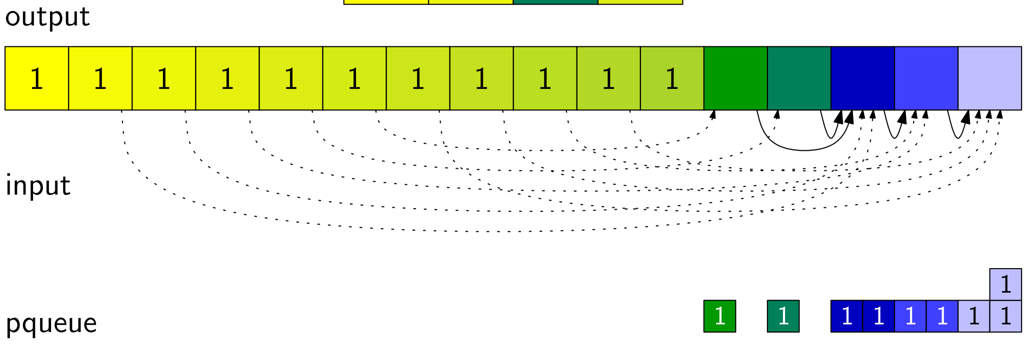




Time-forward processing (Arge et al.)

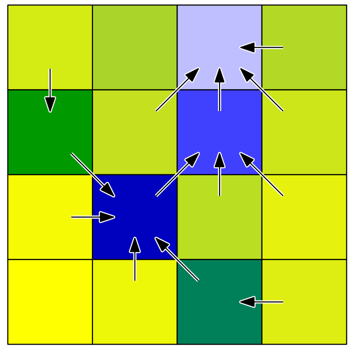
Goal: compute flow accumulation for each cell c
=
#cells from which water passes through c
=
size of tree rooted at c

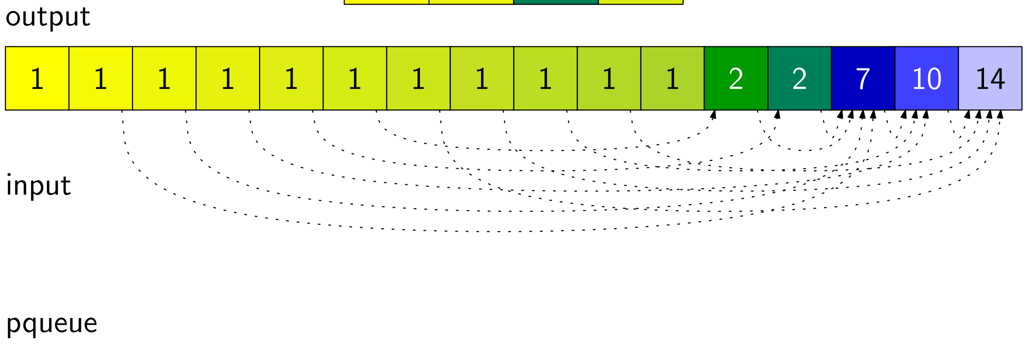




Time-forward processing (Arge et al.)

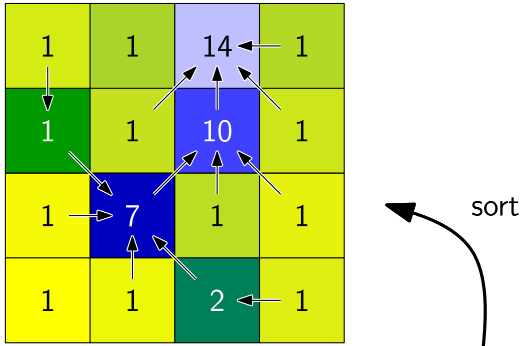
Goal: compute flow accumulation for each cell c
=
#cells from which water passes through c
=
size of tree rooted at c



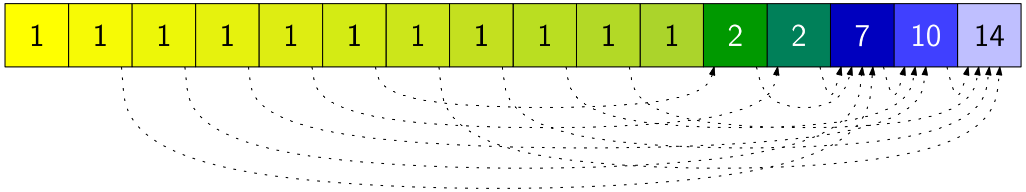


Time-forward processing (Arge et al.)

Goal: compute flow accumulation for each cell c
=
#cells from which water passes through c
=
size of tree rooted at c



output



Time-forward processing

Worst-case I/O's: $\Theta(\frac{N}{B}\log_{M/B}\frac{N}{B})$ (Arge et al.)

mergesort, recursion depth = 2; each level: read once, write once; 24 bytes per element

mergesort recursion depth = 2; priority queue fits in memory

I/O-volume per grid cell (optimistic):

Sorting grid into list of (xy-location, topological nr., out-neighbour top. nr.)	$2 \times 2 \times 24 = 96$ bytes
Flow accumulation, input:	24 bytes
Flow accumulation, output: (xy-location, flow)	16 bytes
Sorting output into grid	$2 \times 2 \times 16 = 64$ bytes
Total:	**200 bytes**

topological number = sorting key ≈ elevation

mergesort recursion depth = 3; priority queue does not fit

mergesort, recursion depth = 3

I/O-volume per grid cell (pessimistic):

Sorting grid into list of (xy-location, topological nr., out-neighbour top. nr.)	$3 \times 2 \times 24 = 144$ bytes
Flow accumulation, input:	24 bytes
Flow accumulation, priority queue:	$2 \times 16 = 32$ bytes
Flow accumulation, output: (xy-location, flow)	16 bytes
Sorting output into grid	$3 \times 2 \times 16 = 96$ bytes
Total:	**312 bytes**

assume each element written once, read once

16 bytes per element (key + amount)

N = #cells in grid M = main memory size B = #bytes in one I/O

Results on flow accumulation

algorithm	file order	theoretical analysis: worst case	'realistic'	bytes per cell	*experiments* time (mins)
row-by-row scan	row by row	$O(N)$	$O(N/\sqrt{B})$	tenthousands	111
Z-order scan	row by row	$O(N)$	$O(N/B)^*$	thousands	
Z-order scan	Z-order	$O(N/\sqrt{B})$	$O(N/B)$	hundreds	41
Easy implementation: no need to tune to hardware / implement I/O-control					
separator-based	row by row	$O(N/B)^*$		20 to 60	39
separator-based	Z-order	$O(N/B)$		10	should try!
Implementation must explicitly adapt subgrid size to available memory M					
time-fwd proc.	any	$O(\frac{N}{B}\log_{M/B}\frac{N}{B})$		70 to 300	sev. hundred
Flexible; requires I/O-efficient sorting and priority queue					

bytes of disk I/O per cell calculated based on: $N = 2^{32}$, $M = 1$ GB, $B = 16$ to 64KB
time: 3 GHz Pentium, one disk for data + scratch, $N = 3.5 \cdot 10^9$ (Neuse), $M = 1$ GB
*) needs tall cache: $M \geq cB^2$

N = #cells in grid M = main memory size B = #bytes in one I/O

Extensions / other applications

Z-order traversal (easy to implement, no tuning to hardware):
- flow accumulation (single-directional flow)
- visibility maps
- matrix operations
- spatial data structures

Separator-based technique (tuned to available memory size):
- flooding local minima (minimum spanning trees)
- flow accumulation (single-directional flow)
- (?) single-source shortest paths

Time-forward processing (using library for I/O-efficient priority queue):
- flow accumulation (multi-directional flow, also irregular network models)

http://haverkort.net
→ *research*
→ *algorithms for geographic elevation models* and *I/O-efficient graph algorithms*