CSE 6220/CX 4220
Introduction to HPC

Lecture 5: Cache-Oblivious Algorithms

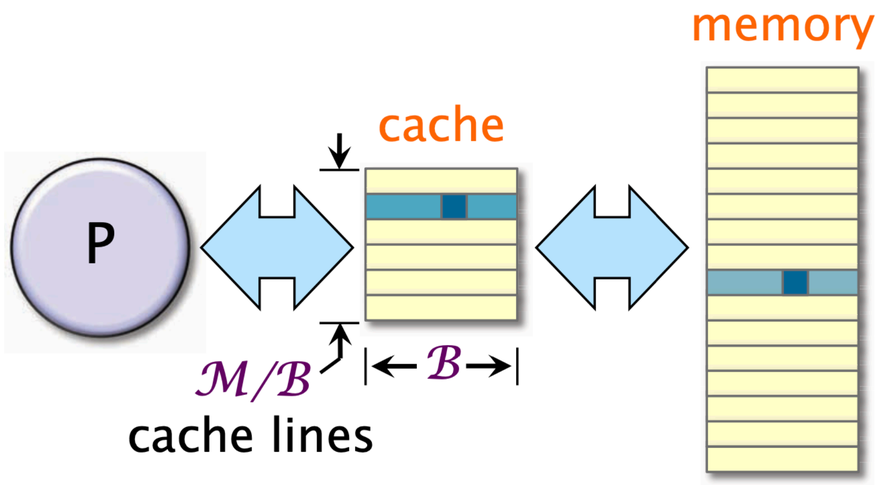
Helen Xu
hxu615@gatech.edu

Slides from MIT 6.172, UC Davis EEC 289Q

Recall: Ideal-Cache Model

Parameters

- Two-level hierarchy
- Cache size of $\mathscr{M}$ bytes
- Cache-line length of $\mathscr{B}$ bytes
- Fully associative
- Optimal, omniscient replacement.



Performance Measures

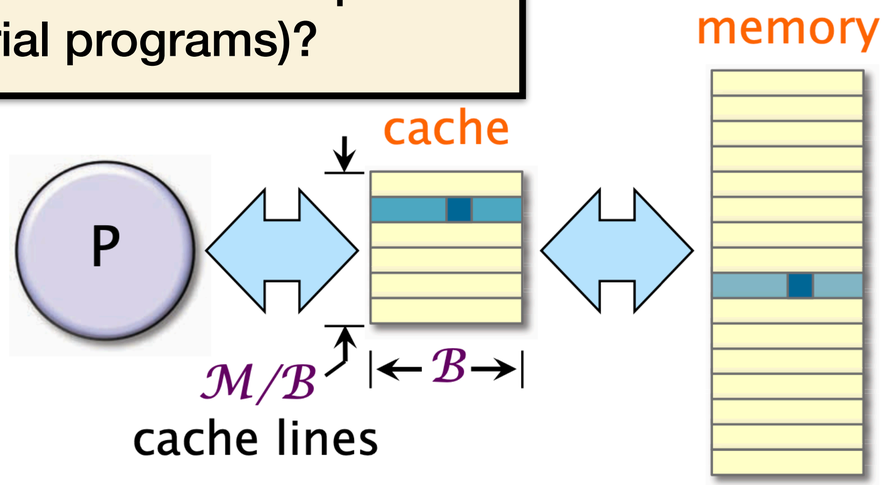
- **Work** W (ordinary running time)
- **Cache misses** Q (number of cache lines that need to be transferred between cache and memory)

Recall: Ideal-Cache Model

Quiz: What type of cache misses are not captured by this model (for serial programs)?

Parameters

- Two-level hierarchy
- Cache size of $\mathscr{M}$ bytes
- Cache-line length of $\mathscr{B}$ bytes
- Fully associative
- Optimal, omniscient replacement.



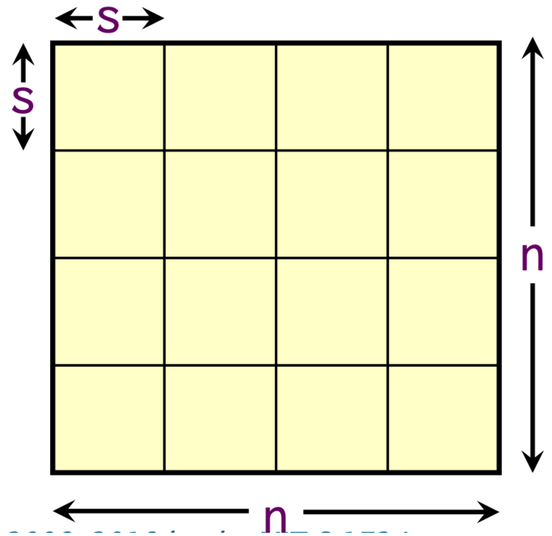
Performance Measures

- **Work** W (ordinary running time)
- **Cache misses** Q (number of cache lines that need to be transferred between cache and memory)

Recap: Tiled (Blocked) matrix multiply

```
void Tiled_Mult(double *C, double *A, double *B, int64_t n) {
  for (int64_t i1=0; i1<n; i1+=s)
    for (int64_t j1=0; j1<n; j1+=s)
      for (int64_t k1=0; k1<n; k1+=s)
        for (int64_t i=i1;                    i++)
          for (int64_t j=j                    n; j++)
            for (int64_t k                    k<n; k++)
              C[i*n+j] +=                     n+j];
}
```

How?



Analysis of cache misses

Tune s so that the submatrices just fit into cache:

$$s = \Theta(\mathscr{M}^{1/2}).$$

Submatrix Caching Lemma implies $\Theta(s^2/\mathscr{B})$ misses per submatrix.

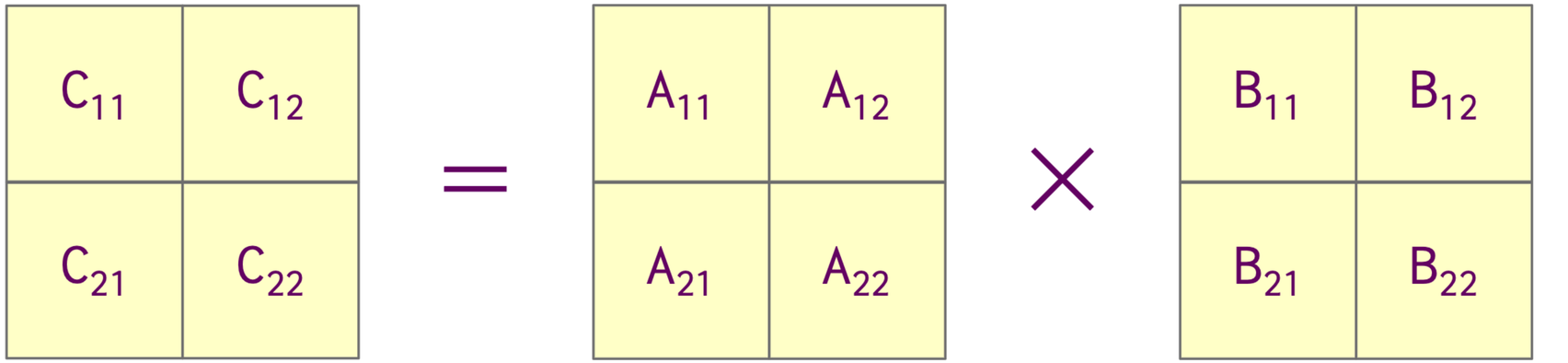
Optimal [HK81]

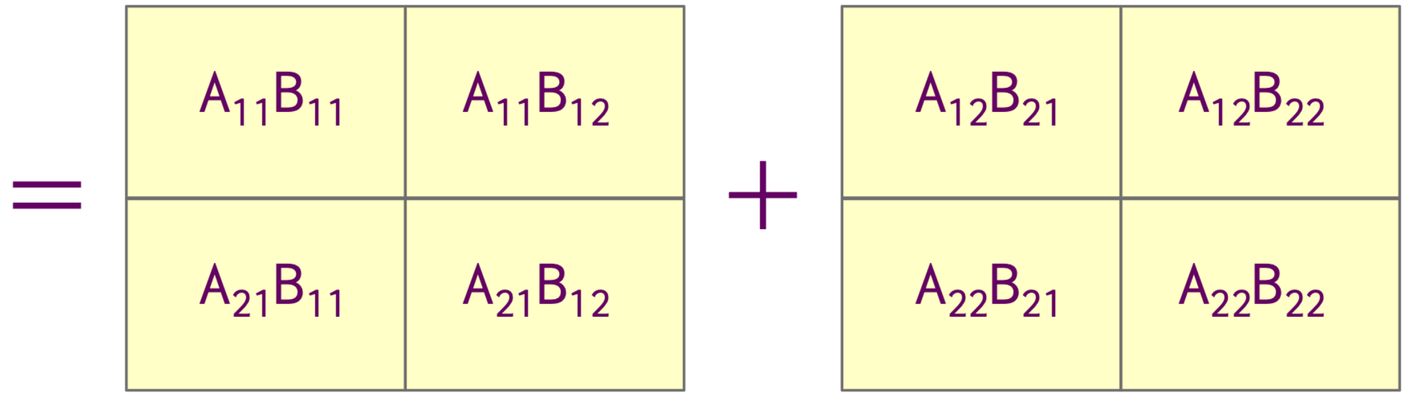
$$Q(n) = \Theta((n/s)^3(s^2/\mathscr{B})) = \Theta(n^3/(\mathscr{B}\mathscr{M}^{1/2})).$$

Recursive Matrix Multiplication

Recursive matrix multiplication

Divide-and-conquer on nxn matrices:





8 multiply-adds of (n/2)x(n/2) matrices.

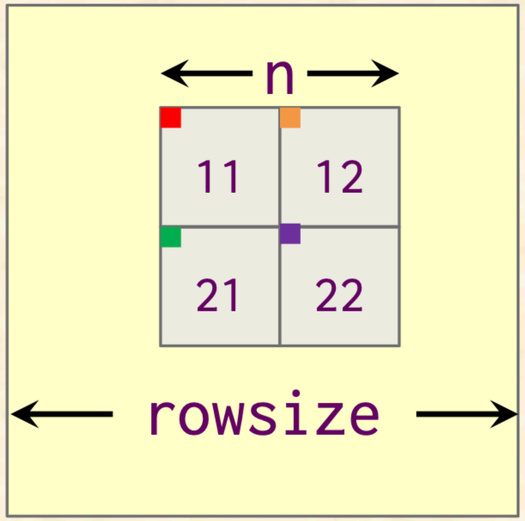
Recursive code

```
// Assume that n is an exact power of 2.
void Rec_Mult(double *C, double *A, double *B,
              int64_t n, int64_t rowsize) {
  if (n == 1)
    C[0] += A[0] * B[0];
  else {
    int64_t d11 = 0;
    int64_t d12 = n/2;
    int64_t d21 = (n/2) * rowsize;
    int64_t d22 = (n/2) * (rowsize+1);

    Rec_Mult(C+d11, A+d11, B+d11, n/2, rowsize);
    Rec_Mult(C+d11, A+d12, B+d21, n/2, rowsize);
    Rec_Mult(C+d12, A+d11, B+d12, n/2, rowsize);
    Rec_Mult(C+d12, A+d12, B+d22, n/2, rowsize);
    Rec_Mult(C+d21, A+d21, B+d11, n/2, rowsize);
    Rec_Mult(C+d21, A+d22, B+d21, n/2, rowsize);
    Rec_Mult(C+d22, A+d21, B+d12, n/2, rowsize);
    Rec_Mult(C+d22, A+d22, B+d22, n/2, rowsize);
} }
```

Coarsen base case to overcome function-call overheads.

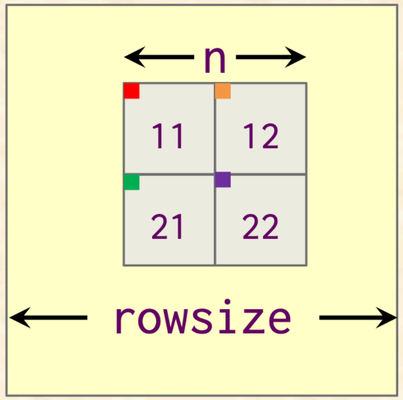
Recursive code

```
// Assume that n is an exact power of 2.
void Rec_Mult(double *C, double *A, double *B,
              int64_t n, int64_t rowsize) {
  if (n == 1)
    C[0] += A[0] * B[0];
  else {
    int64_t d11 = 0;
    int64_t d12 = n/2;
    int64_t d21 = (n/2) * rowsize;
    int64_t d22 = (n/2) * (rowsize+1);

    Rec_Mult(C+d11, A+d11, B+d11, n/2, rowsize);
    Rec_Mult(C+d11, A+d12, B+d21, n/2, rowsize);
    Rec_Mult(C+d12, A+d11, B+d12, n/2, rowsize);
    Rec_Mult(C+d12, A+d12, B+d22, n/2, rowsize);
    Rec_Mult(C+d21, A+d21, B+d11, n/2, rowsize);
    Rec_Mult(C+d21, A+d22, B+d21, n/2, rowsize);
    Rec_Mult(C+d22, A+d21, B+d12, n/2, rowsize);
    Rec_Mult(C+d22, A+d22, B+d22, n/2, rowsize);
} }
```



Analysis of work

```
// Assume that n is an exact power of 2.
void Rec_Mult(double *C, double *A, double *B,
              int64_t n, int64_t rowsize) {
  if (n == 1)
    C[0] += A[0] * B[0];
  else {
    int64_t d11 = 0;
    int64_t d12 = n/2;
    int64_t d21 = (n/2) * rowsize;
    int64_t d22 = (n/2) * (rowsize+1);

    Rec_Mult(C+d11, A+d11, B+d11, n/2, rowsize);
    Rec_Mult(C+d11, A+d12, B+d21, n/2, rowsize);
    Rec_Mult(C+d12, A+d11, B+d12, n/2, rowsize);
    Rec_Mult(C+d12, A+d12, B+d22, n/2, rowsize);
    Rec_Mult(C+d21, A+d21, B+d11, n/2, rowsize);
    Rec_Mult(C+d21, A+d22, B+d21, n/2, rowsize);
    Rec_Mult(C+d22, A+d21, B+d12, n/2, rowsize);
    Rec_Mult(C+d22, A+d22, B+d22, n/2, rowsize);
} }
```



$$W(n) = 8W(n/2) + \Theta(1)$$
$$= \Theta(n^3)$$

Analysis of work

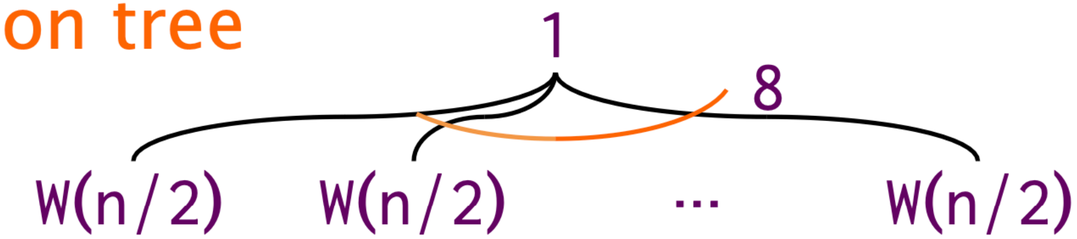
$$W(n) = 8W(n/2) + \Theta(1)$$

recursion tree W(n)

Analysis of work

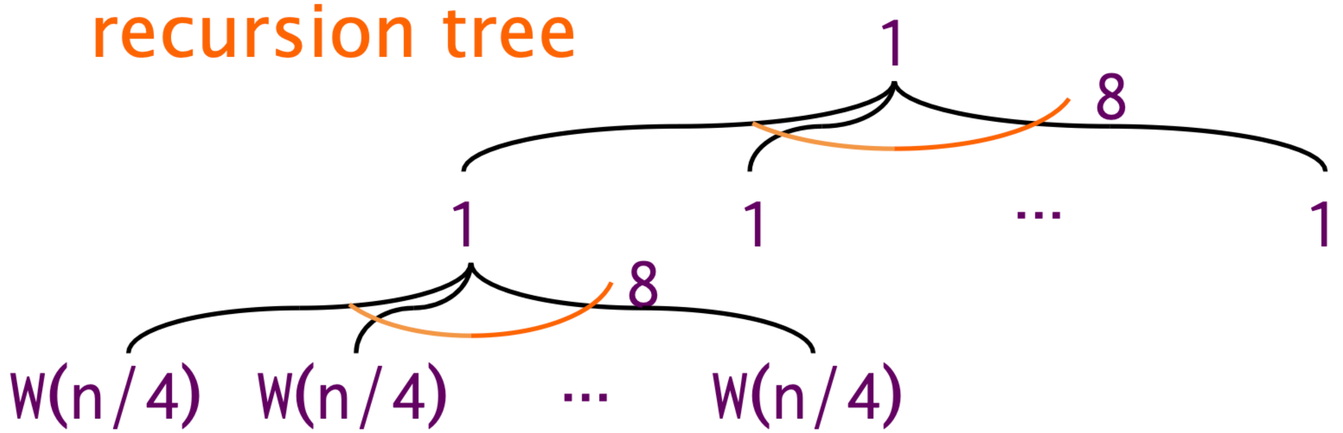
$$W(n) = 8W(n/2) + \Theta(1)$$

recursion tree



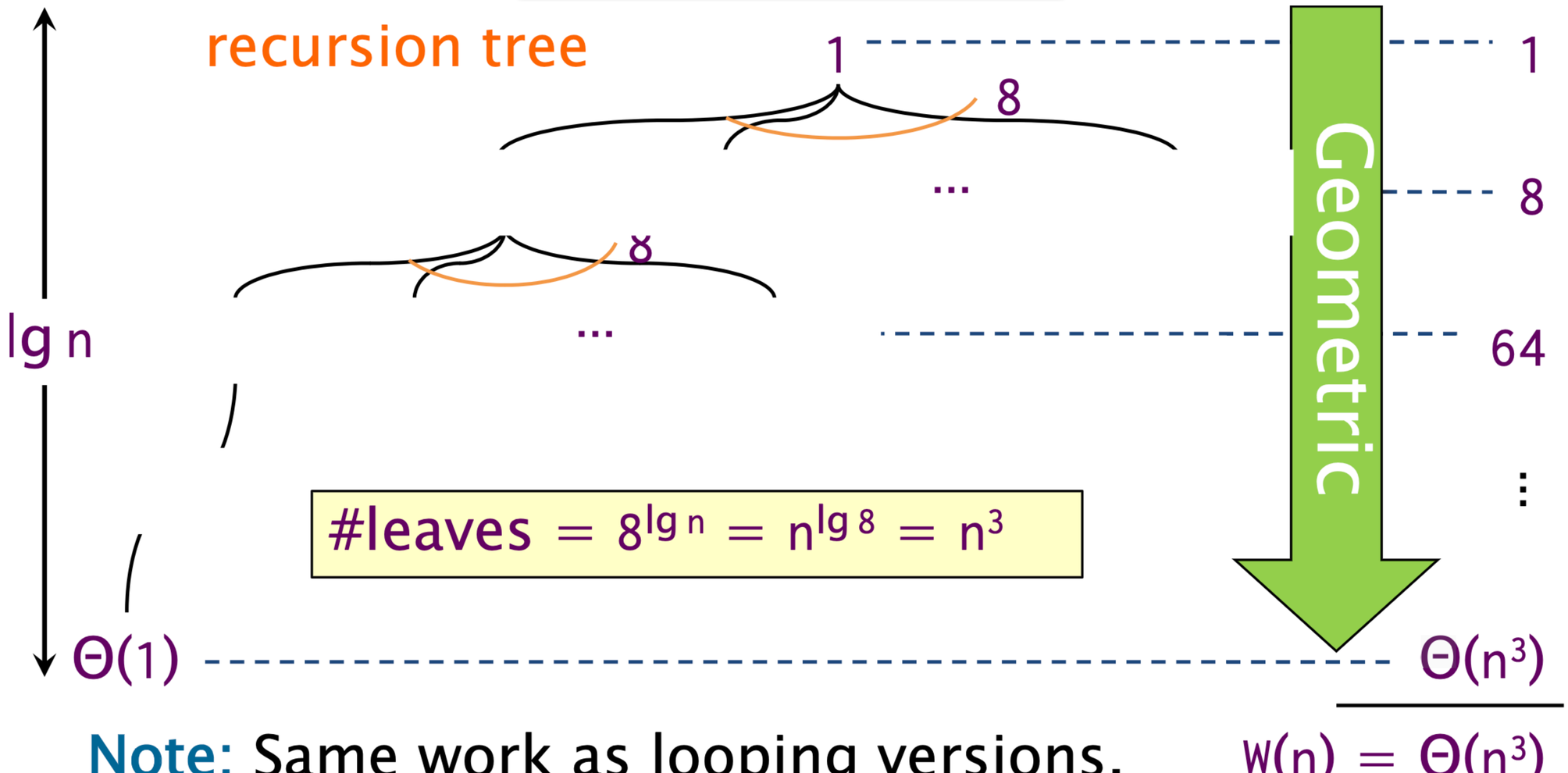
Analysis of work

$$W(n) = 8W(n/2) + \Theta(1)$$



Analysis of work

$$W(n) = 8W(n/2) + \Theta(1)$$



Note: Same work as looping versions.

$W(n) = \Theta(n^3)$

Analysis of cache misses

```
// Assume that n is an exact power of 2.
void Rec_Mult(double *C, double *A, double *B,
              int64_t n, int64_t rowsize) {
  if (n == 1)
    C[0] += A[0] * B[0];
  else {
    int64_t d11 = 0;
    int64_t d12 = n/2;
    int64_t d21 = (n/2) * rowsize;
    int64_t d22 = (n/2) * (rowsize+1);

    Rec_Mult(C+d11, A+d11, B+d11, n/2, rowsize);
    Rec_Mult(C+d11, A+d12, B+d21, n/2, rowsize);
    Rec_Mult(C+d12, A+d11, B+d12, n/2, rowsize);
    Rec_Mult(C+d12, A+d12, B+d22, n/2, rowsize);
    Rec_Mult(C+d21, A+d21, B+d11, n/2, rowsize);
    Rec_Mult(C+d21, A+d22, B+d21, n/2, rowsize);
    Rec_Mult(C+d22, A+d21, B+d12, n/2, rowsize);
    Rec_Mult(C+d22, A+d22, B+d22, n/2, rowsize);
} }
```

Submatrix Caching Lemma

$$Q(n) = \begin{cases} \Theta(n^2/\mathcal{B}) \text{ if } n^2 < c\mathcal{M} \text{ for suff. small const } c \le 1, \\ 8Q(n/2) + \Theta(1) \text{ otherwise.} \end{cases}$$

Analysis of cache misses

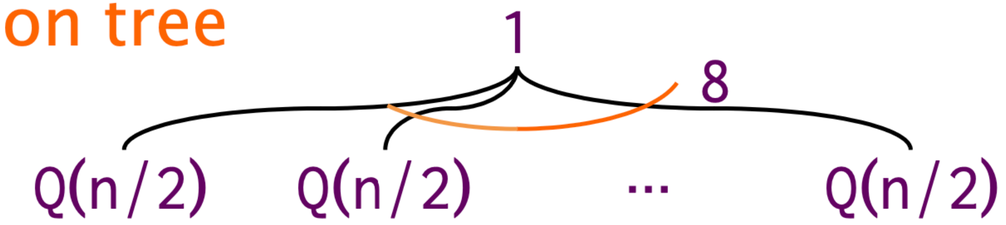
$$Q(n) = \begin{cases} \Theta(n^2/\mathcal{B}) \text{ if } n^2 < c\mathcal{M} \text{ for suff. small const } c \le 1, \\ 8Q(n/2) + \Theta(1) \text{ otherwise.} \end{cases}$$

recursion tree $Q(n)$

Analysis of cache misses

$$Q(n) = \begin{cases} \Theta(n^2/\mathcal{B}) \text{ if } n^2 < c\mathcal{M} \text{ for suff. small const } c \le 1, \\ 8Q(n/2) + \Theta(1) \text{ otherwise.} \end{cases}$$

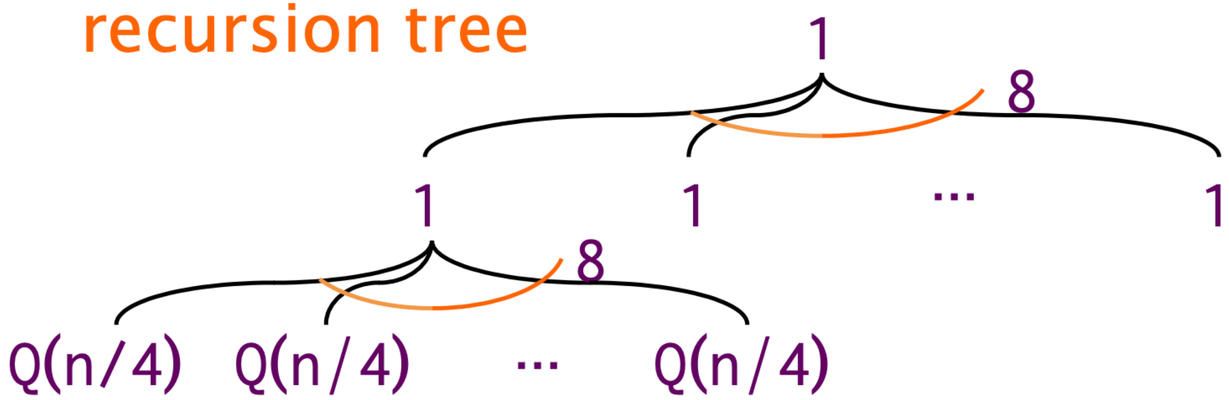
recursion tree



Analysis of cache misses

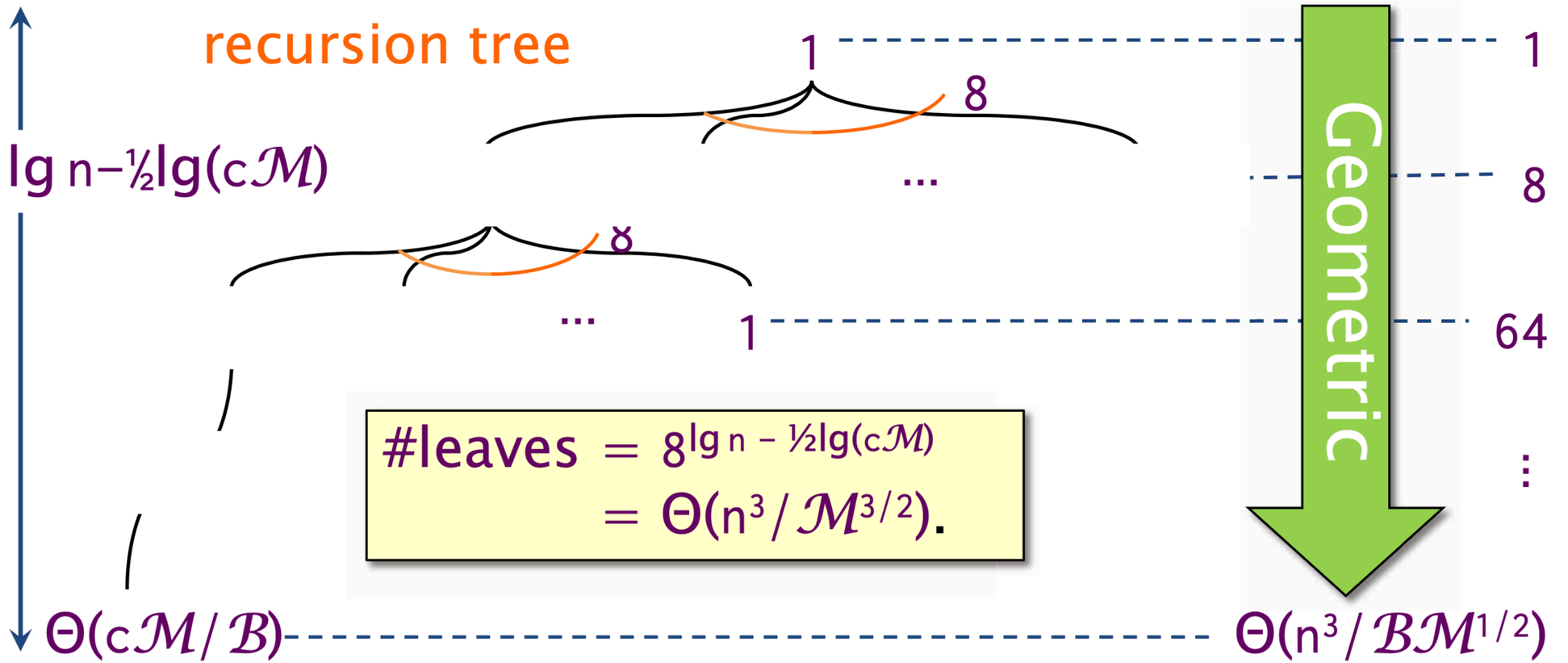
$$Q(n) = \begin{cases} \Theta(n^2/\mathcal{B}) \text{ if } n^2 < c\mathcal{M} \text{ for suff. small const } c \le 1, \\ 8Q(n/2) + \Theta(1) \text{ otherwise.} \end{cases}$$

recursion tree



Analysis of cache misses

$$Q(n) = \begin{cases} \Theta(n^2/\mathcal{B}) \text{ if } n^2 < c\mathcal{M} \text{ for suff. small const } c \le 1, \\ 8Q(n/2) + \Theta(1) \text{ otherwise.} \end{cases}$$



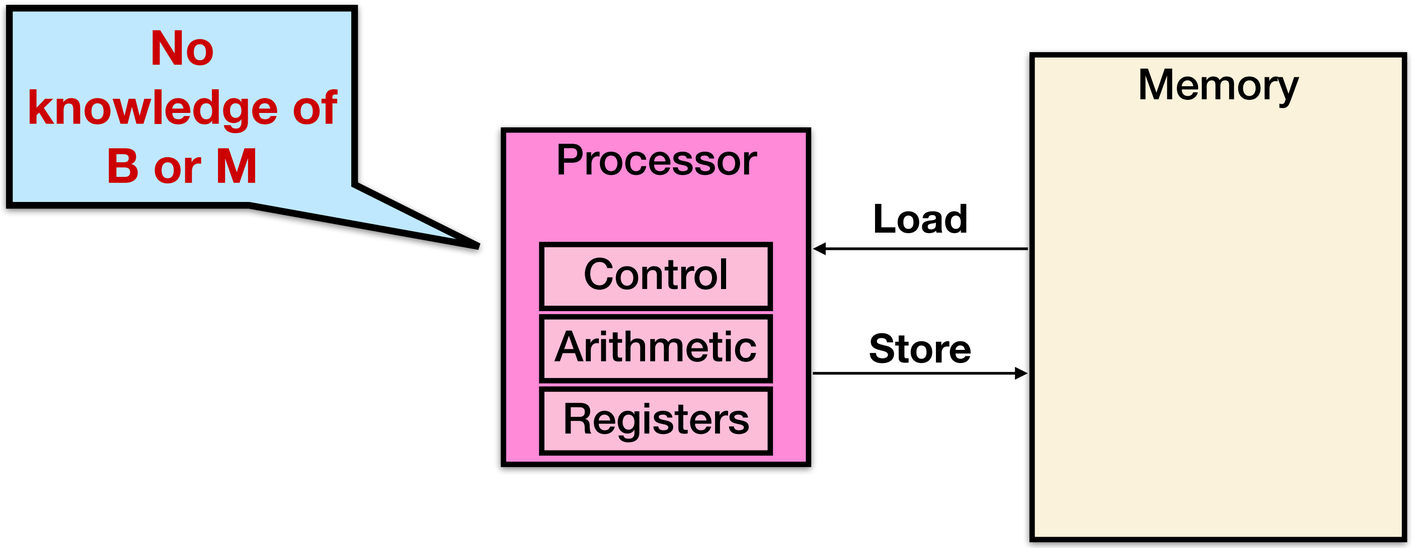
#leaves $= 8^{\lg n - \frac{1}{2}\lg(c\mathcal{M})}$
$= \Theta(n^3/\mathcal{M}^{3/2})$.

$$Q(n) = \Theta(n^3/\mathcal{B}\mathcal{M}^{1/2})$$

Same cache misses as with tiling!

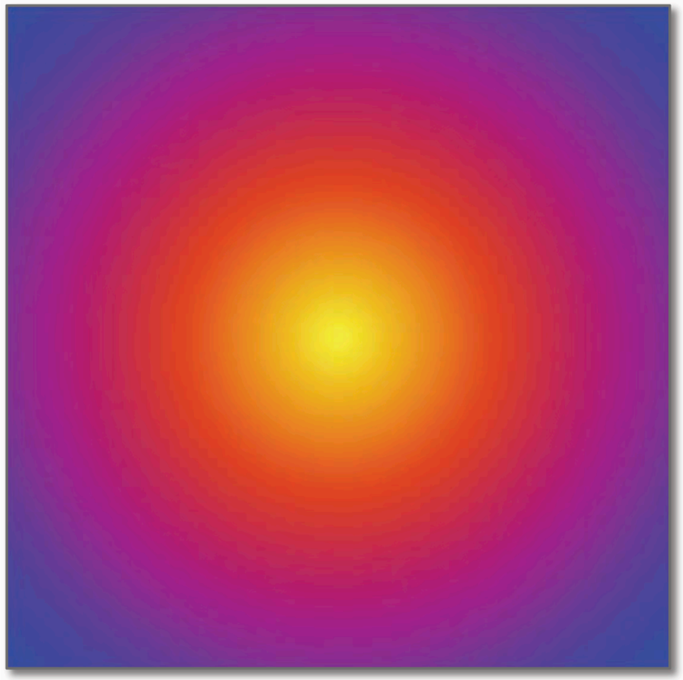
Efficient cache-oblivious algorithms

- No tuning parameters.
- No explicit knowledge of cache sizes.
- Handle multilevel caches automatically (asymptotically optimally).
- Good in multiprogrammed environments (see: cache-adaptive algorithms).



Simulation of Heat Diffusion

Heat Diffusion



2D heat equation

Let `u(t, x, y)` = temperature at time `t` of point `(x, y)`.

$$\frac{\partial u}{\partial t} = \alpha \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right)$$

α is the thermal diffusivity (constant).

1D Heat Equation

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}$$

Finite-Difference Approximation

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}$$

$$\frac{\partial}{\partial t} u(t,x) \approx \frac{u(t+\Delta t, x) - u(t,x)}{\Delta t},$$

$$\frac{\partial}{\partial x} u(t,x) \approx \frac{u(t, x+\Delta x/2) - u(t, x-\Delta x/2)}{\Delta x},$$

$$\frac{\partial^2}{\partial x^2} u(t,x) \approx \frac{\frac{\partial}{\partial x} u(t, x+\Delta x/2) - \frac{\partial}{\partial x} u(t, x-\Delta x/2)}{\Delta x},$$

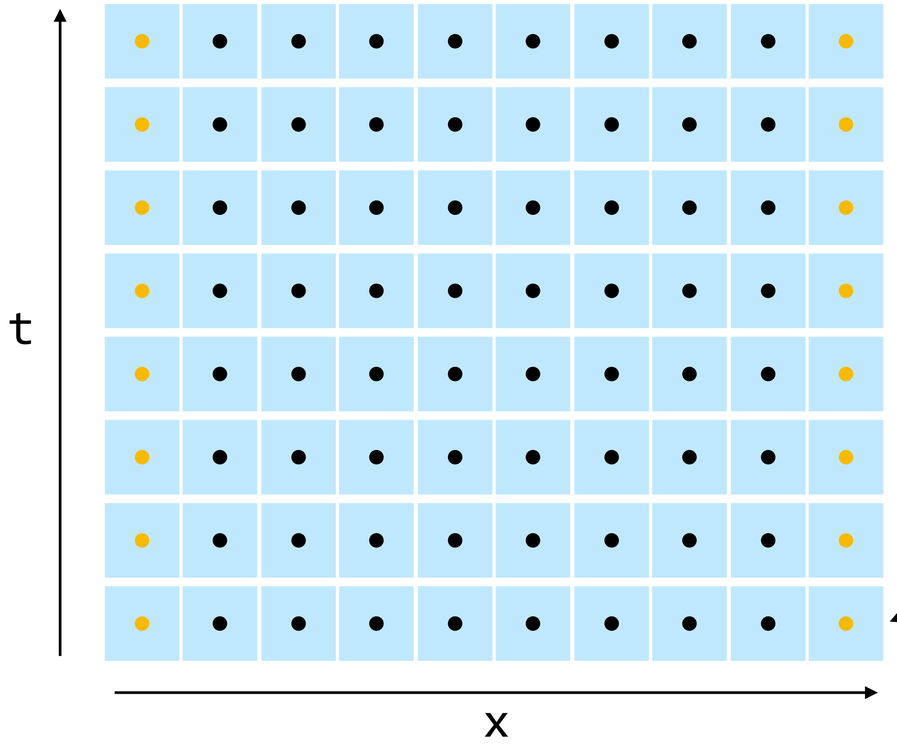
$$\frac{\partial^2}{\partial x^2} u(t,x) \approx \frac{u(t, x+\Delta x) - 2u(t,x) + u(t, x-\Delta x)}{(\Delta x)^2}.$$

So the 1D heat equation reduces to:

$$\frac{u(t+\Delta t, x) - u(t,x)}{\Delta t} \approx \alpha \left(\frac{u(t, x+\Delta x) - 2u(t,x) + u(t, x-\Delta x)}{(\Delta x)^2} \right).$$

3-Point Stencil

$$\frac{u(t+\Delta t,x)-u(t,x)}{\Delta t} \approx \alpha\left(\frac{u(t,x+\Delta x)-2u(t,x)+u(t,x-\Delta x)}{(\Delta x)^2}\right)$$



A **stencil computation** updates each point in an array by a fixed pattern, called a stencil.

iteration space boundary

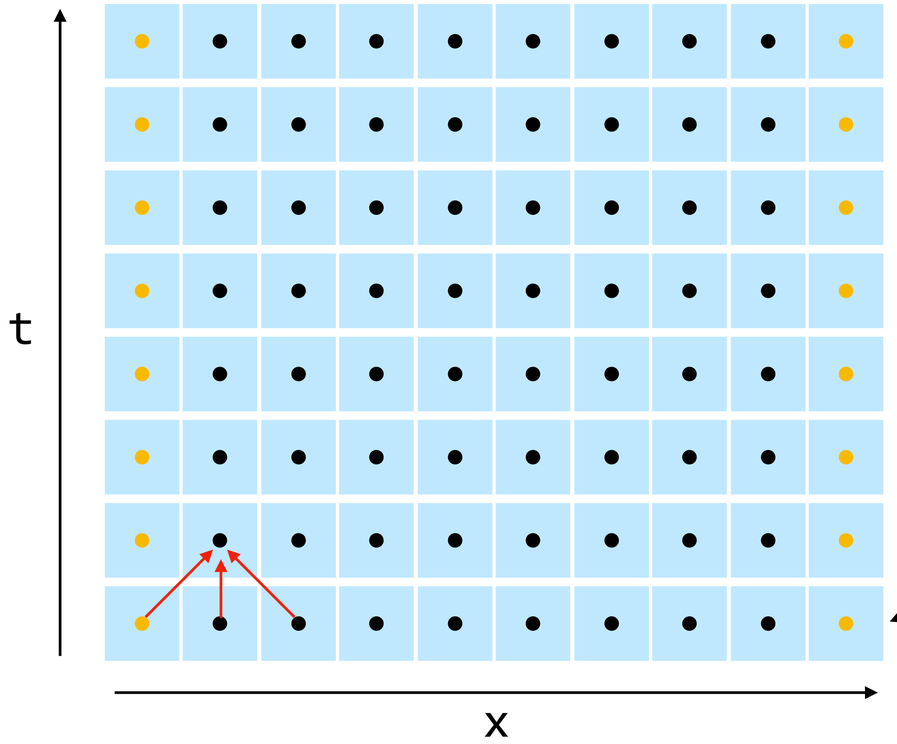
Update rule

$\Delta t = 1, \Delta x = 1$

```
u[t+1][x] = u[t][x] + ALPHA * (u[t][x+1] - 2*u[t][x] + u[t][x-1])
```

3-Point Stencil

$$\frac{u(t+\Delta t,x)-u(t,x)}{\Delta t} \approx \alpha\left(\frac{u(t,x+\Delta x)-2u(t,x)+u(t,x-\Delta x)}{(\Delta x)^2}\right)$$



A **stencil computation** updates each point in an array by a fixed pattern, called a stencil.

iteration space boundary

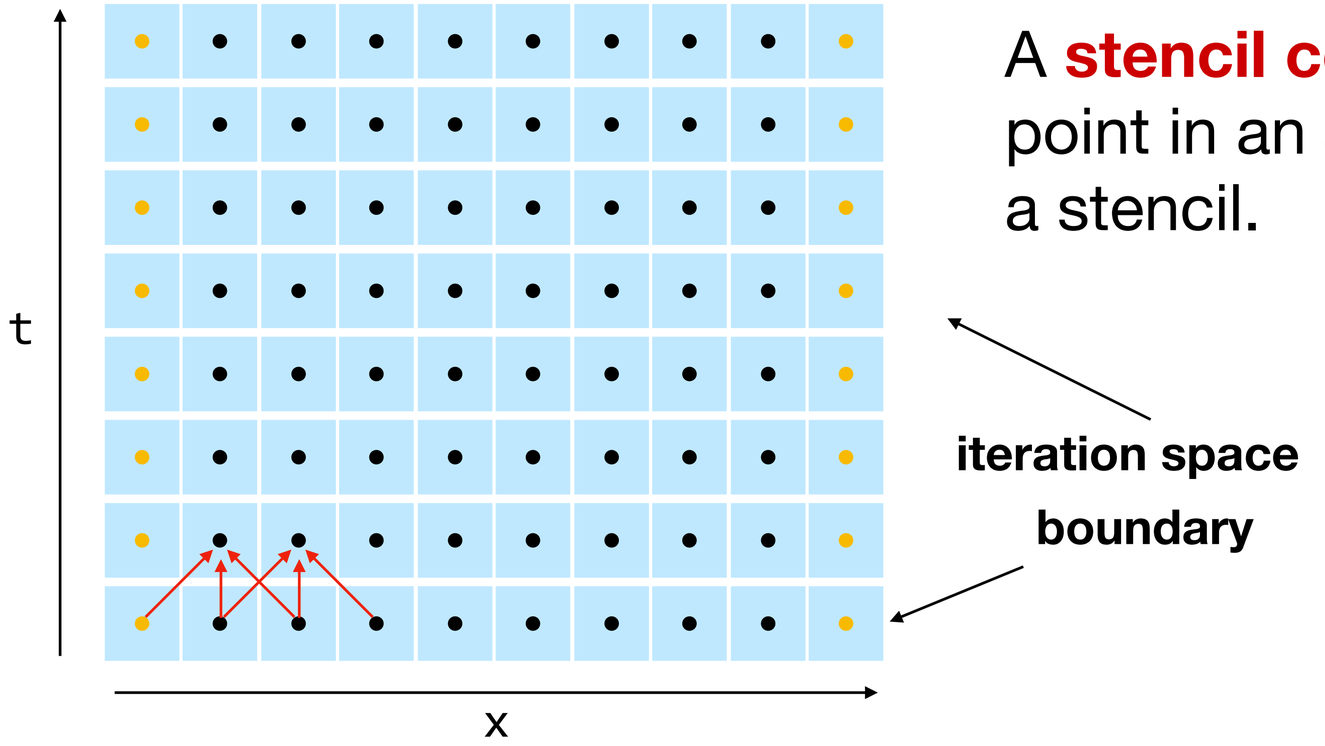
Update rule

$\Delta t = 1, \Delta x = 1$

```
u[t+1][x] = u[t][x] + ALPHA * (u[t][x+1] - 2*u[t][x] + u[t][x-1])
```

3-Point Stencil

$$\frac{u(t+\Delta t,x)-u(t,x)}{\Delta t} \approx \alpha\left(\frac{u(t,x+\Delta x)-2u(t,x)+u(t,x-\Delta x)}{(\Delta x)^2}\right)$$



A **stencil computation** updates each point in an array by a fixed pattern, called a stencil.

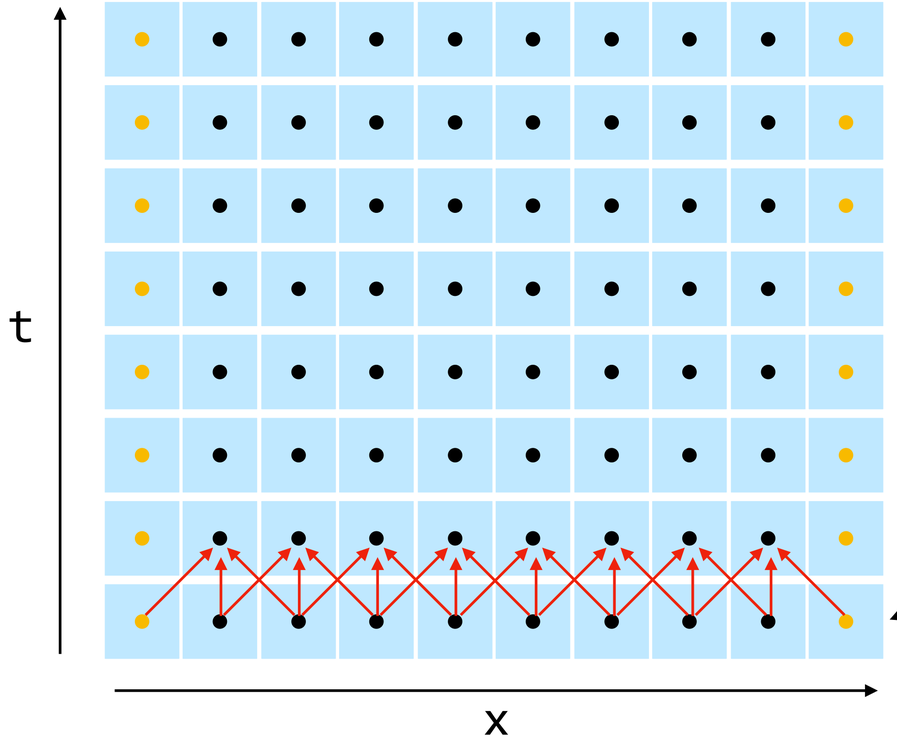
Update rule

$\Delta t = 1, \Delta x = 1$

```
u[t+1][x] = u[t][x] + ALPHA * (u[t][x+1] - 2*u[t][x] + u[t][x-1])
```

3-Point Stencil

$$\frac{u(t+\Delta t,x)-u(t,x)}{\Delta t} \approx \alpha\left(\frac{u(t,x+\Delta x)-2u(t,x)+u(t,x-\Delta x)}{(\Delta x)^2}\right)$$



A **stencil computation** updates each point in an array by a fixed pattern, called a stencil.

iteration space boundary

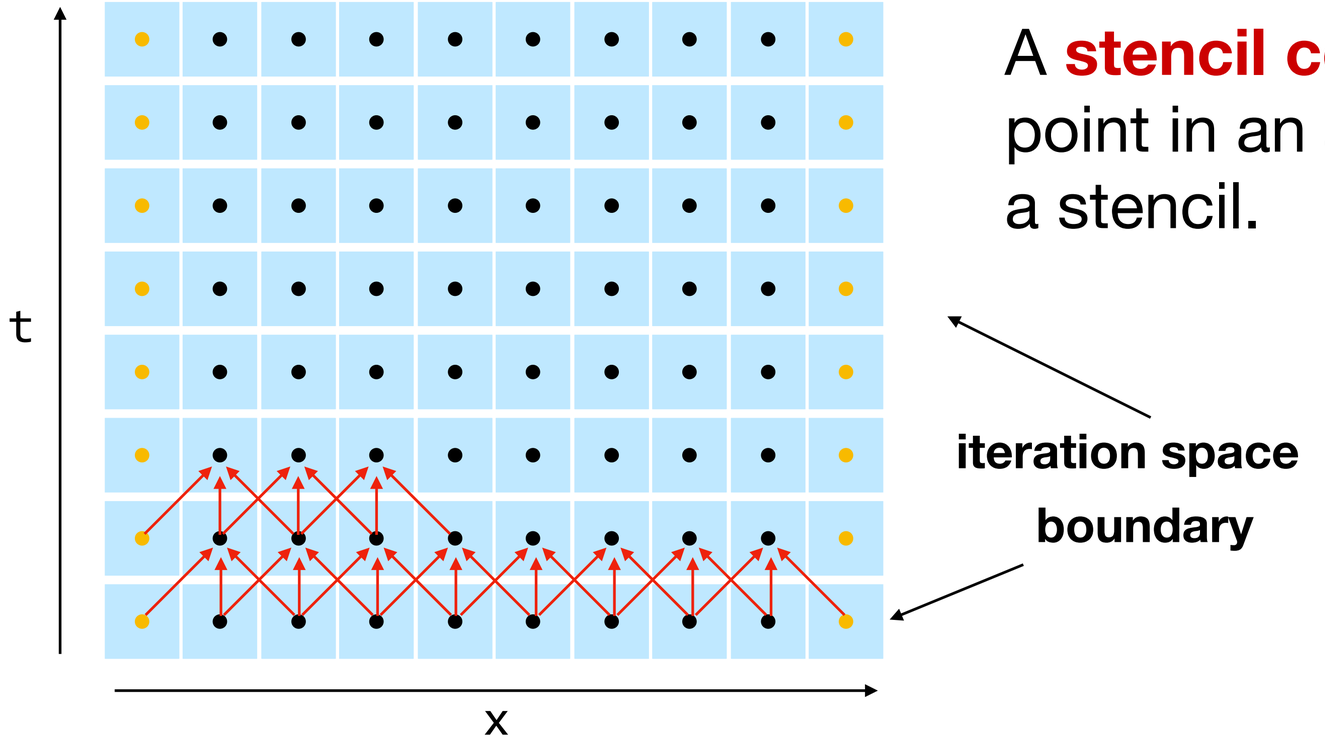
Update rule

$\Delta t = 1, \Delta x = 1$

```
u[t+1][x] = u[t][x] + ALPHA * (u[t][x+1] - 2*u[t][x] + u[t][x-1])
```

3-Point Stencil

$$\frac{u(t+\Delta t,x)-u(t,x)}{\Delta t} \approx \alpha\left(\frac{u(t,x+\Delta x)-2u(t,x)+u(t,x-\Delta x)}{(\Delta x)^2}\right)$$



A **stencil computation** updates each point in an array by a fixed pattern, called a stencil.

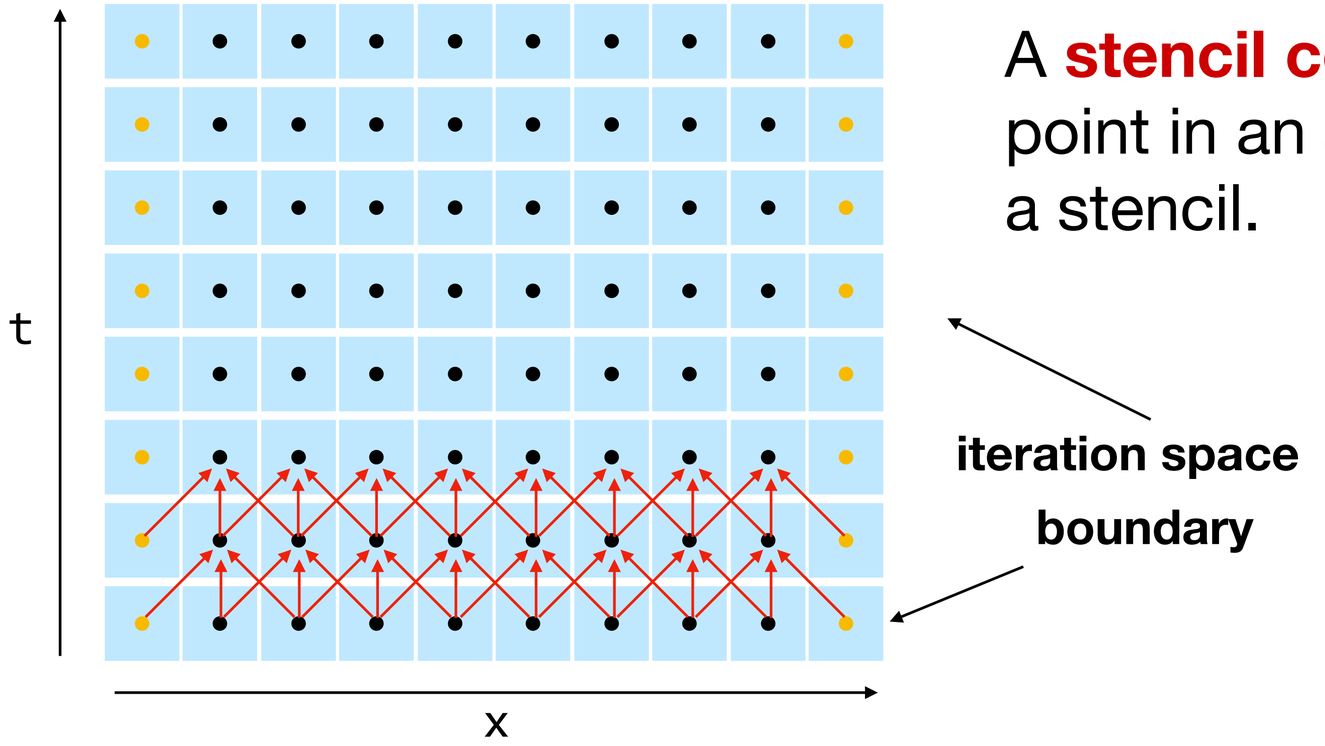
Update rule

$\Delta t = 1, \Delta x = 1$

```
u[t+1][x] = u[t][x] + ALPHA * (u[t][x+1] - 2*u[t][x] + u[t][x-1])
```

3-Point Stencil

$$\frac{u(t+\Delta t,x)-u(t,x)}{\Delta t} \approx \alpha\left(\frac{u(t,x+\Delta x)-2u(t,x)+u(t,x-\Delta x)}{(\Delta x)^2}\right)$$



A **stencil computation** updates each point in an array by a fixed pattern, called a stencil.

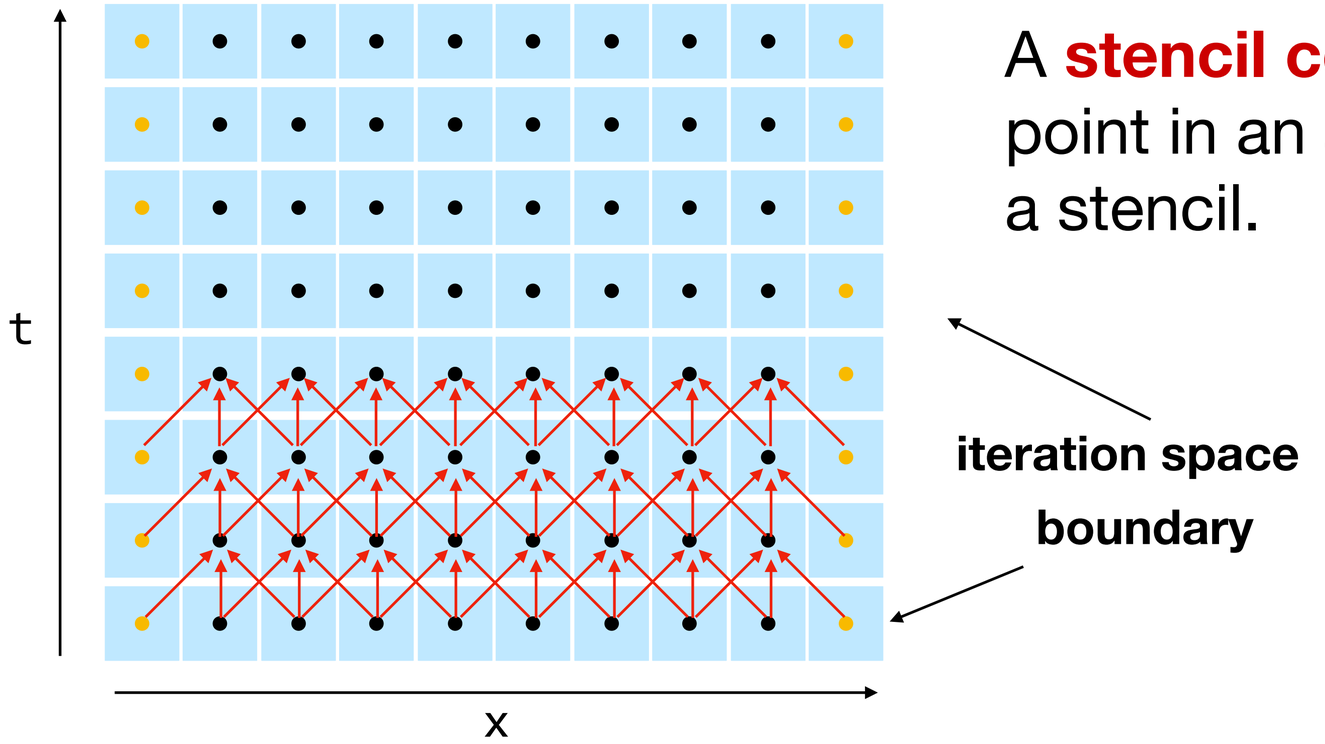
Update rule

$\Delta t = 1, \Delta x = 1$

```
u[t+1][x] = u[t][x] + ALPHA * (u[t][x+1] - 2*u[t][x] + u[t][x-1])
```

3-Point Stencil

$$\frac{u(t+\Delta t,x)-u(t,x)}{\Delta t} \approx \alpha\left(\frac{u(t,x+\Delta x)-2u(t,x)+u(t,x-\Delta x)}{(\Delta x)^2}\right)$$



A **stencil computation** updates each point in an array by a fixed pattern, called a stencil.

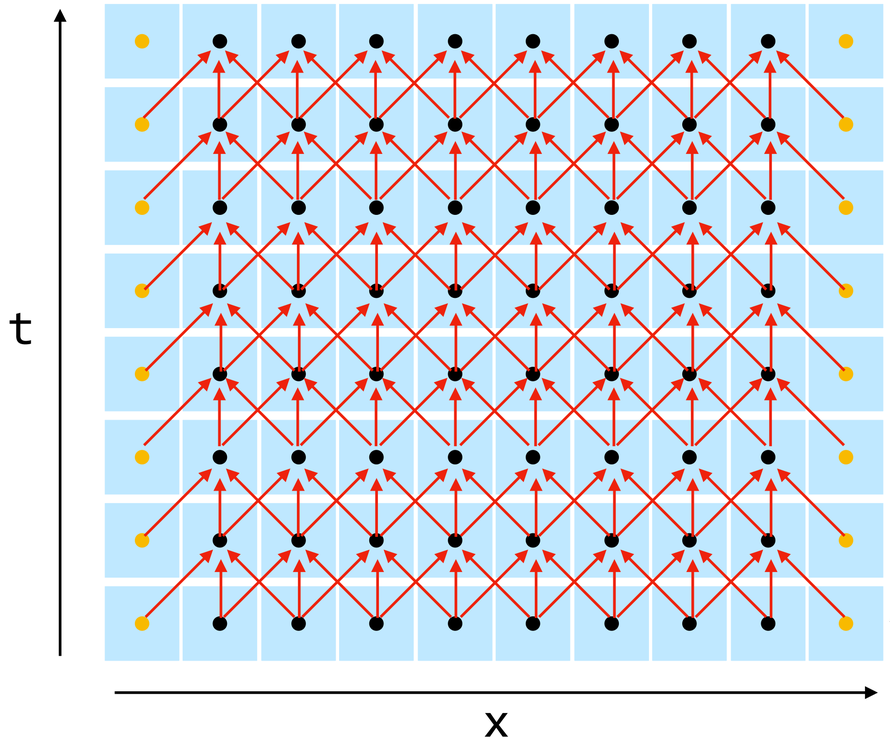
Update rule

$\Delta t = 1, \Delta x = 1$

```
u[t+1][x] = u[t][x] + ALPHA * (u[t][x+1] - 2*u[t][x] + u[t][x-1])
```

3-Point Stencil

$$\frac{u(t+\Delta t,x)-u(t,x)}{\Delta t} \approx \alpha\left(\frac{u(t,x+\Delta x)-2u(t,x)+u(t,x-\Delta x)}{(\Delta x)^2}\right)$$



A **stencil computation** updates each point in an array by a fixed pattern, called a stencil.

iteration space boundary

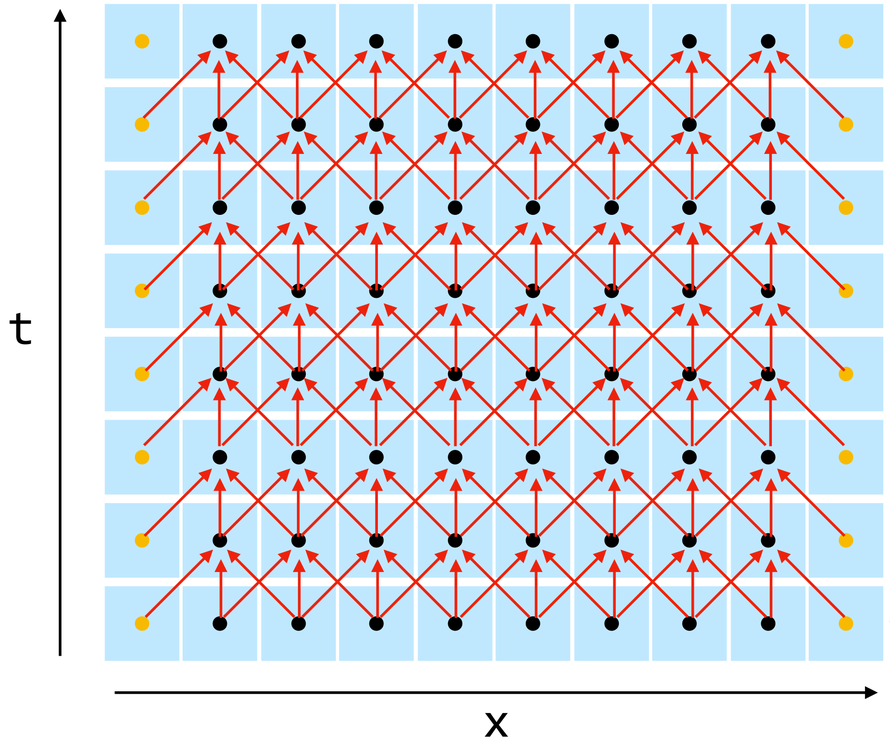
Update rule

$\Delta t = 1, \Delta x = 1$

```
u[t+1][x] = u[t][x] + ALPHA * (u[t][x+1] - 2*u[t][x] + u[t][x-1])
```

3-Point Stencil

$$\frac{u(t+\Delta t,x)-u(t,x)}{\Delta t} \approx \alpha\left(\frac{u(t,x+\Delta x)-2u(t,x)+u(t,x-\Delta x)}{(\Delta x)^2}\right)$$



A **stencil computation** updates each point in an array by a fixed pattern, called a stencil.

iteration space boundary

How would this code perform with respect to caching?

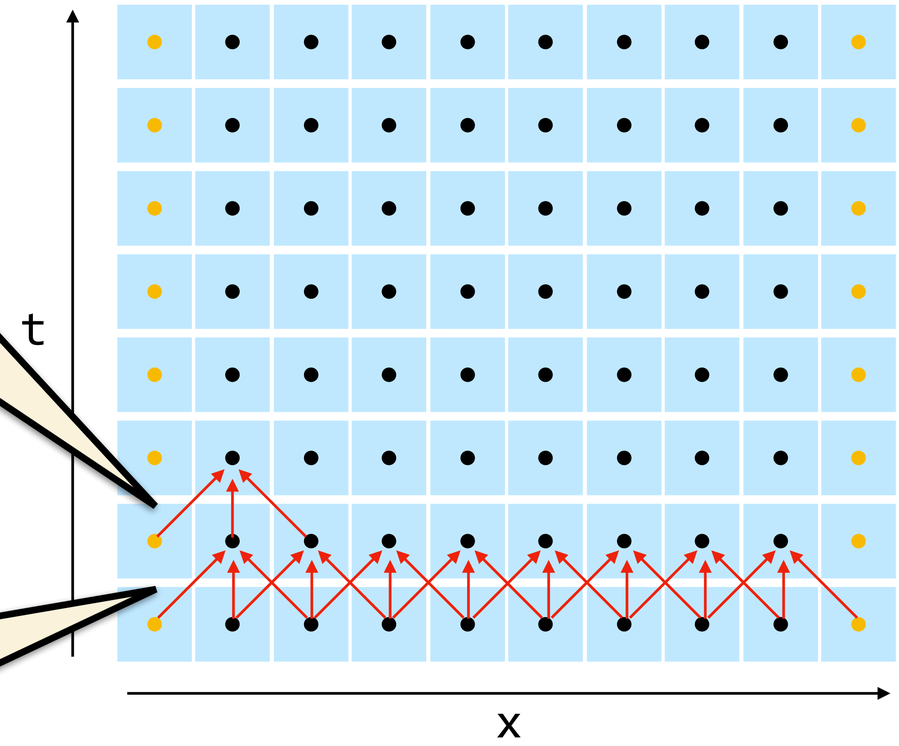
Update rule

$\Delta t = 1, \Delta x = 1$

```
u[t+1][x] = u[t][x] + ALPHA * (u[t][x+1] - 2*u[t][x] + u[t][x-1])
```

Cache behavior of looping stencil

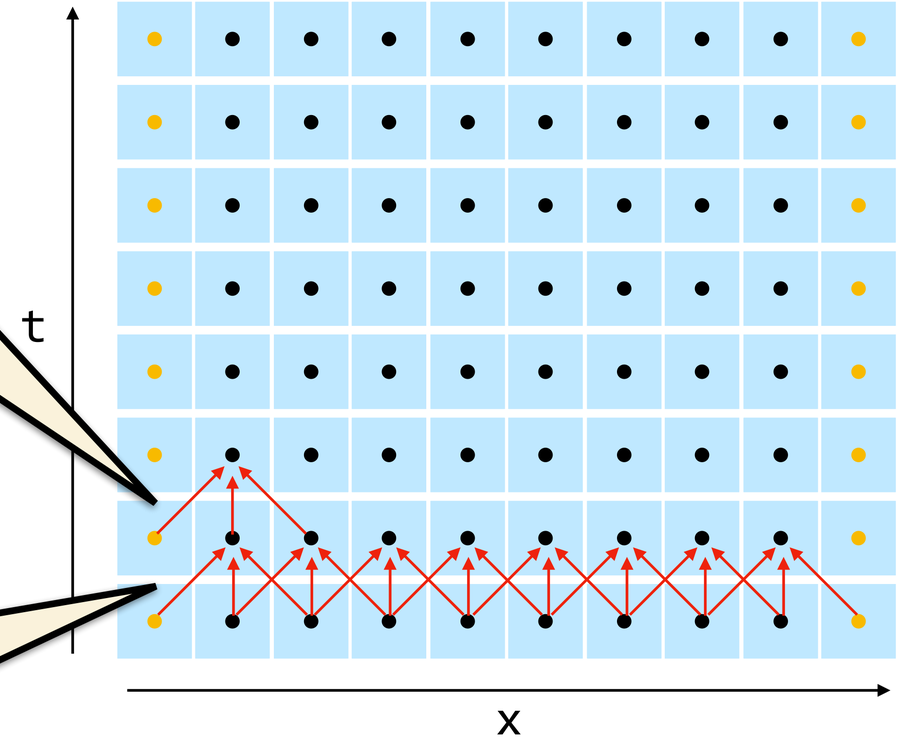
If x is larger than the cache size, by the time you start the next row, the relevant entries are evicted



Potential for reuse by keeping around the input points

Cache behavior of looping stencil

If x is larger than the cache size, by the time you start the next row, the relevant entries are evicted

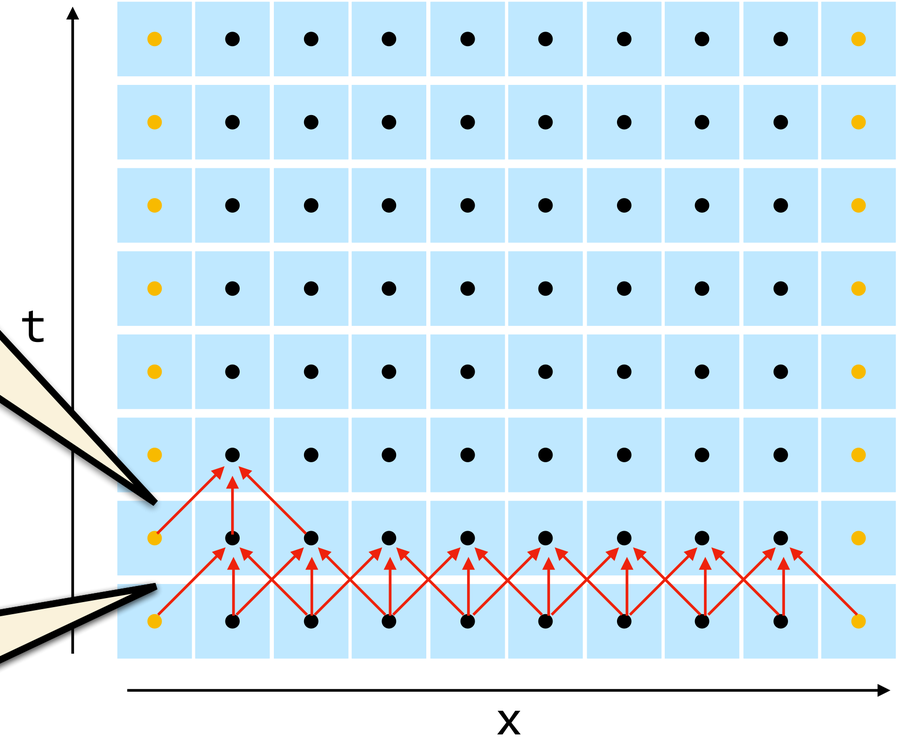


Potential for reuse by keeping around the input points

If we only care about the values of x at the most recent time step, do we need to keep around the whole 2D matrix? How many rows do we need?

Cache behavior of looping stencil

If x is larger than the cache size, by the time you start the next row, the relevant entries are evicted



Potential for reuse by keeping around the input points

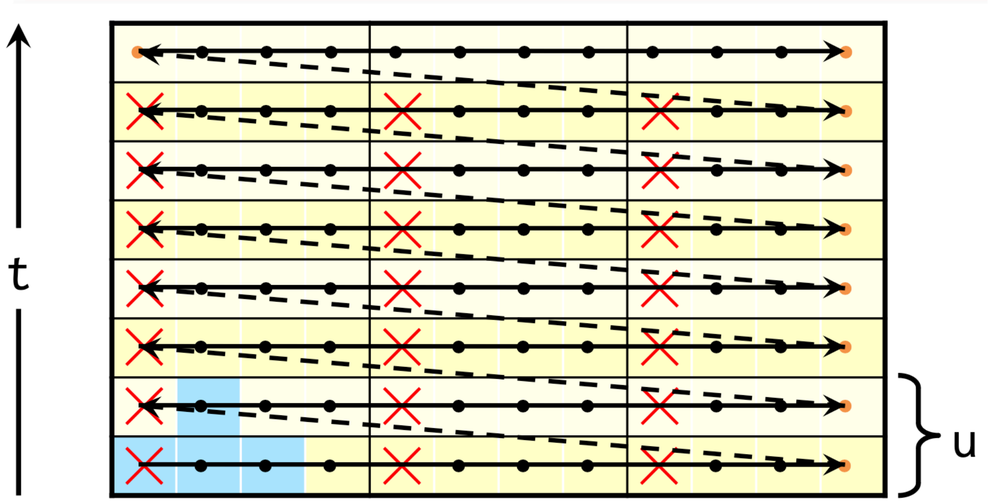
If we only care about the values of x at the most recent time step, do we need to keep around the whole 2D matrix? How many rows do we need?

Can keep 2 rows - one for previous and one for current time

Cache-Oblivious Stencil Computations

Cache behavior of looping

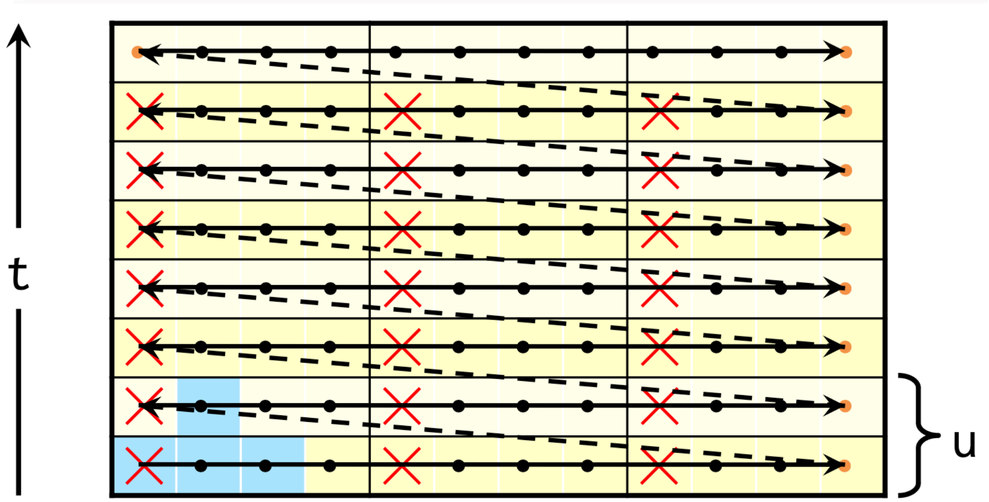
```
double u[2][N]; // even-odd trick

static inline double kernel(double * w) {
  return w[0] + ALPHA * (w[-1] – 2*w[0] + w[1]);
}

for (size_t t = 1; t < T-1; ++t) { // time loop
  for(size_t x = 1; x < N-1; ++x) // space loop
    u[(t+1)%2][x] = kernel( &u[t%2][x] );
```



Assuming LRU, how many cache misses does the looping code incur (as a function of N and T)?

Cache behavior of looping

```
double u[2][N]; // even-odd trick

static inline double kernel(double * w) {
  return w[0] + ALPHA * (w[-1] - 2*w[0] + w[1]);
}

for (size_t t = 1; t < T-1; ++t) { // time loop
  for(size_t x = 1; x < N-1; ++x) // space loop
    u[(t+1)%2][x] = kernel( &u[t%2][x] );
```



Assuming LRU, if $N > \mathscr{M}$, then

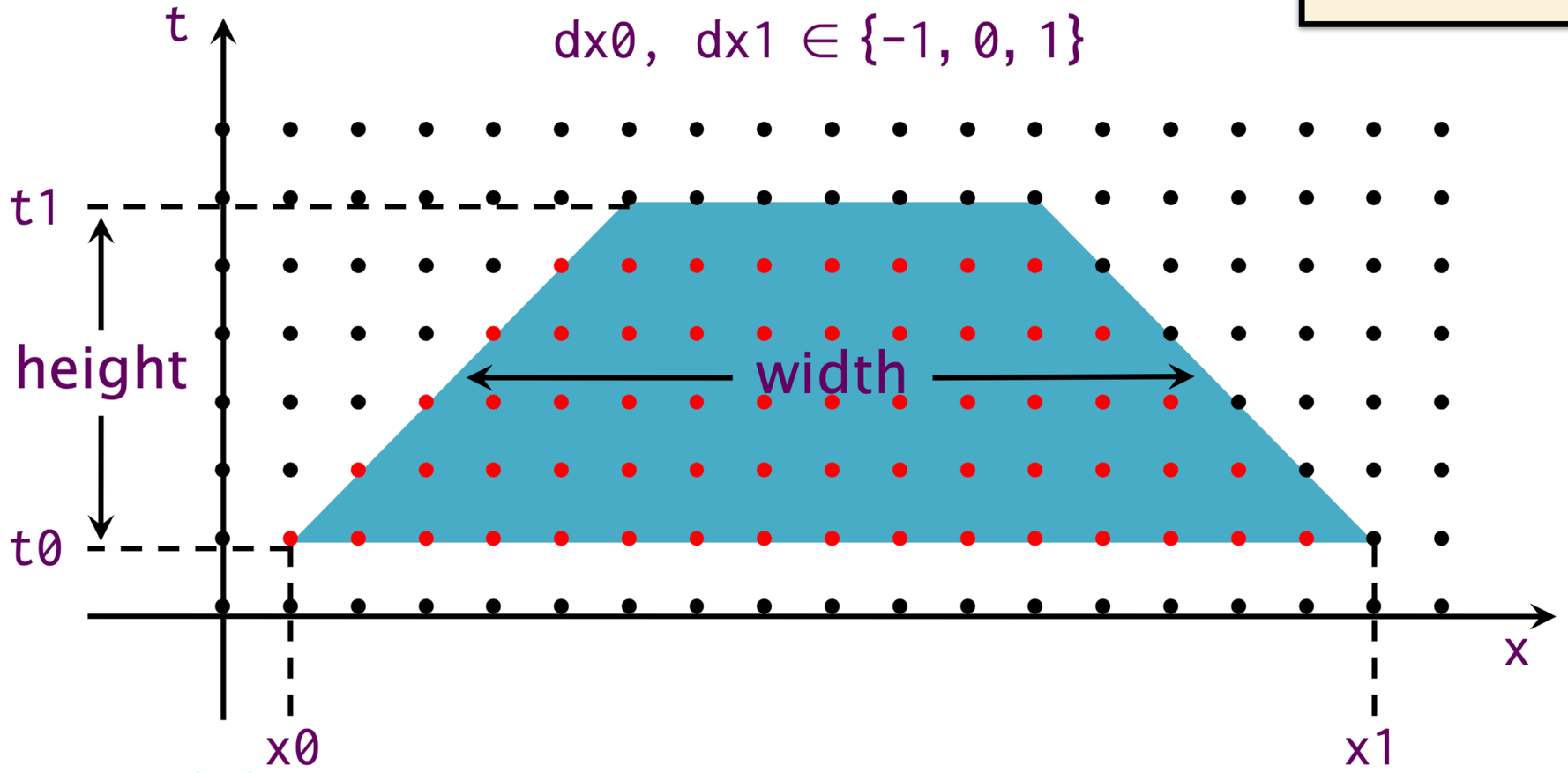
$$Q = \Theta(NT/\mathscr{B})$$

Cache-Oblivious 3-Point Stencil

Recursively traverse trapezoidal regions of space-time points (t,x) such that

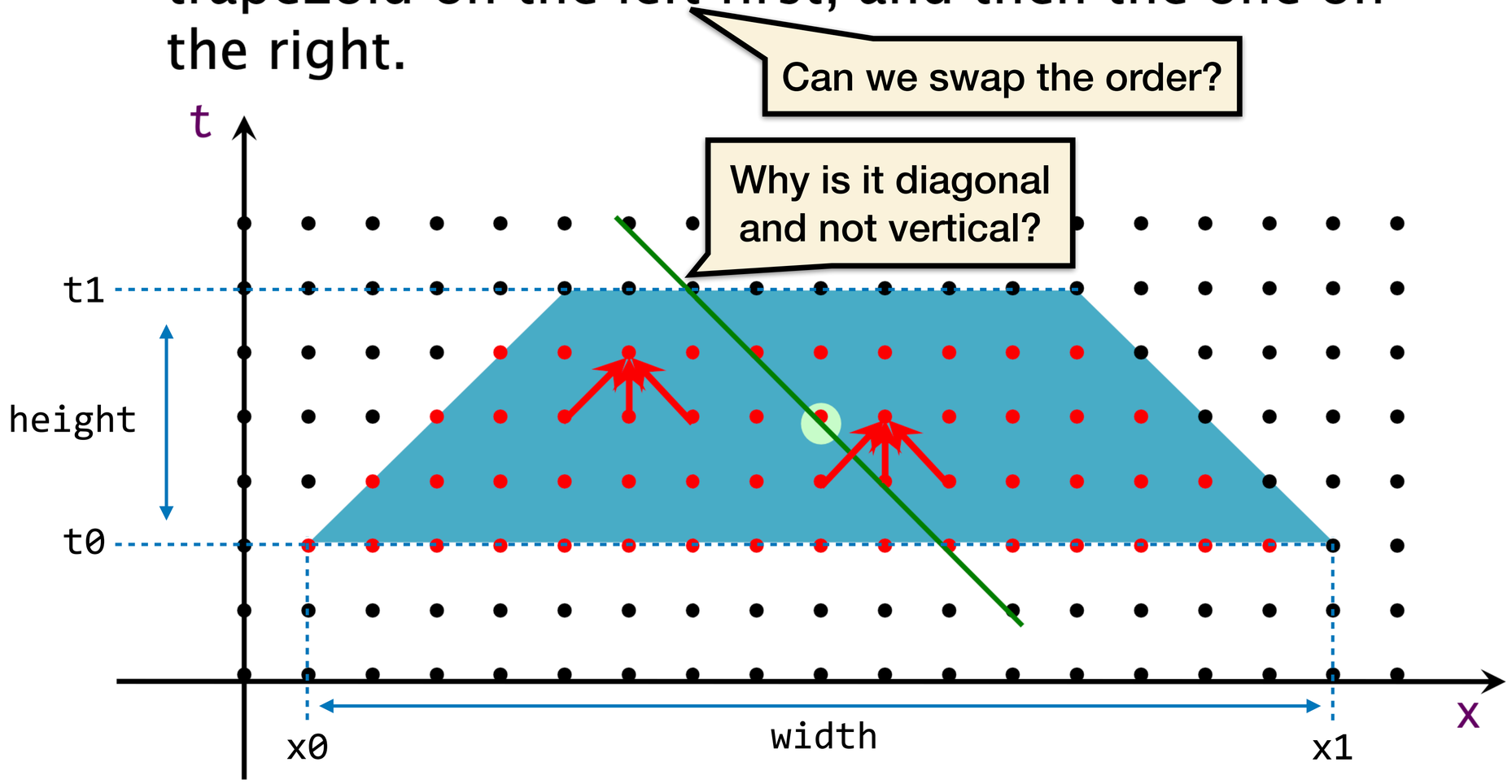
$$t0 \le t < t1$$
$$x0+dx0(t-t0) \le x < x1+dx1(t-t0)$$
$$dx0,\ dx1 \in \{-1, 0, 1\}$$

Why not quadrants like we did before in tableau construction?



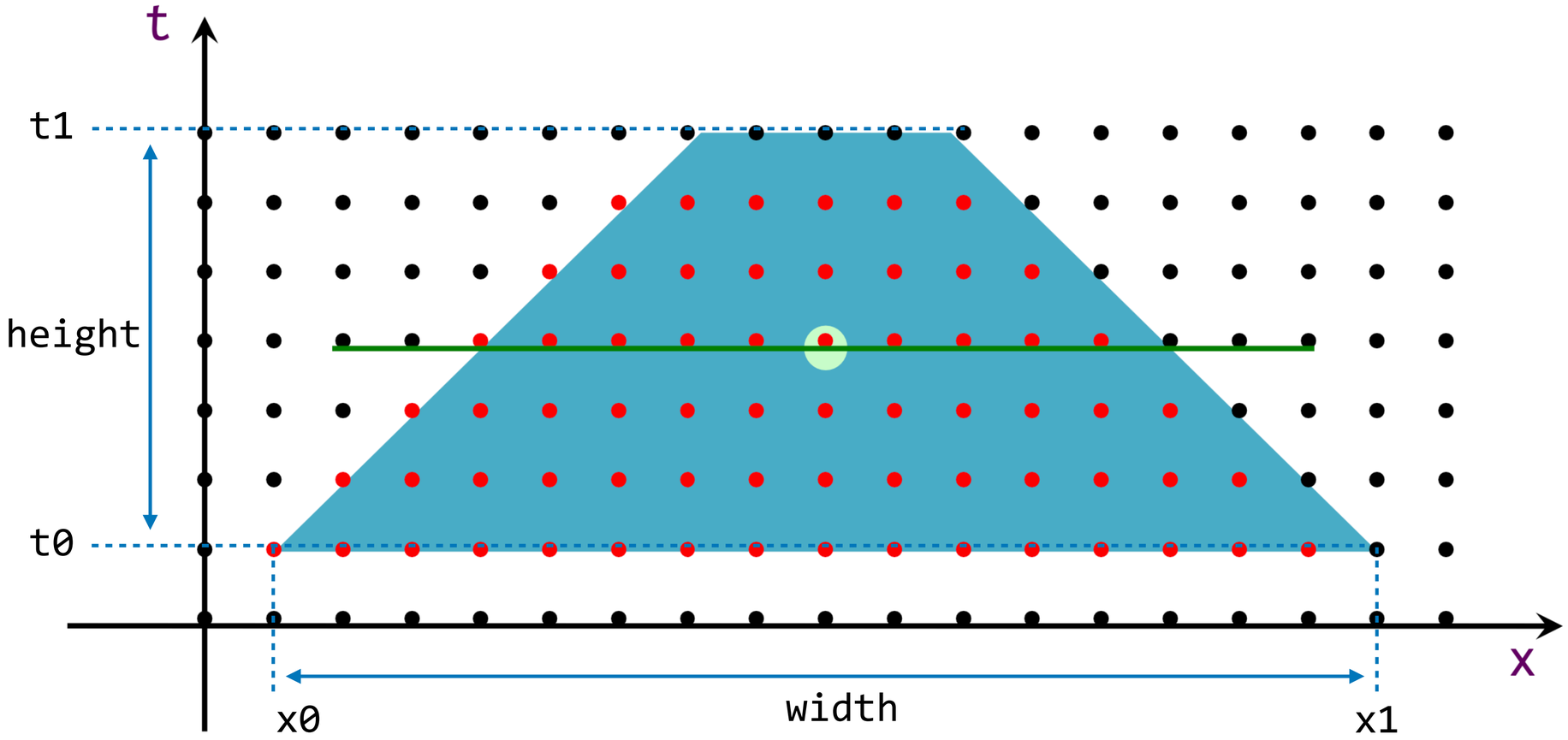
Squat Trapezoid: Space Cut

If width ≥ 2·height, cut the trapezoid with a line of slope −1 through the center. Traverse the trapezoid on the left first, and then the one on the right.



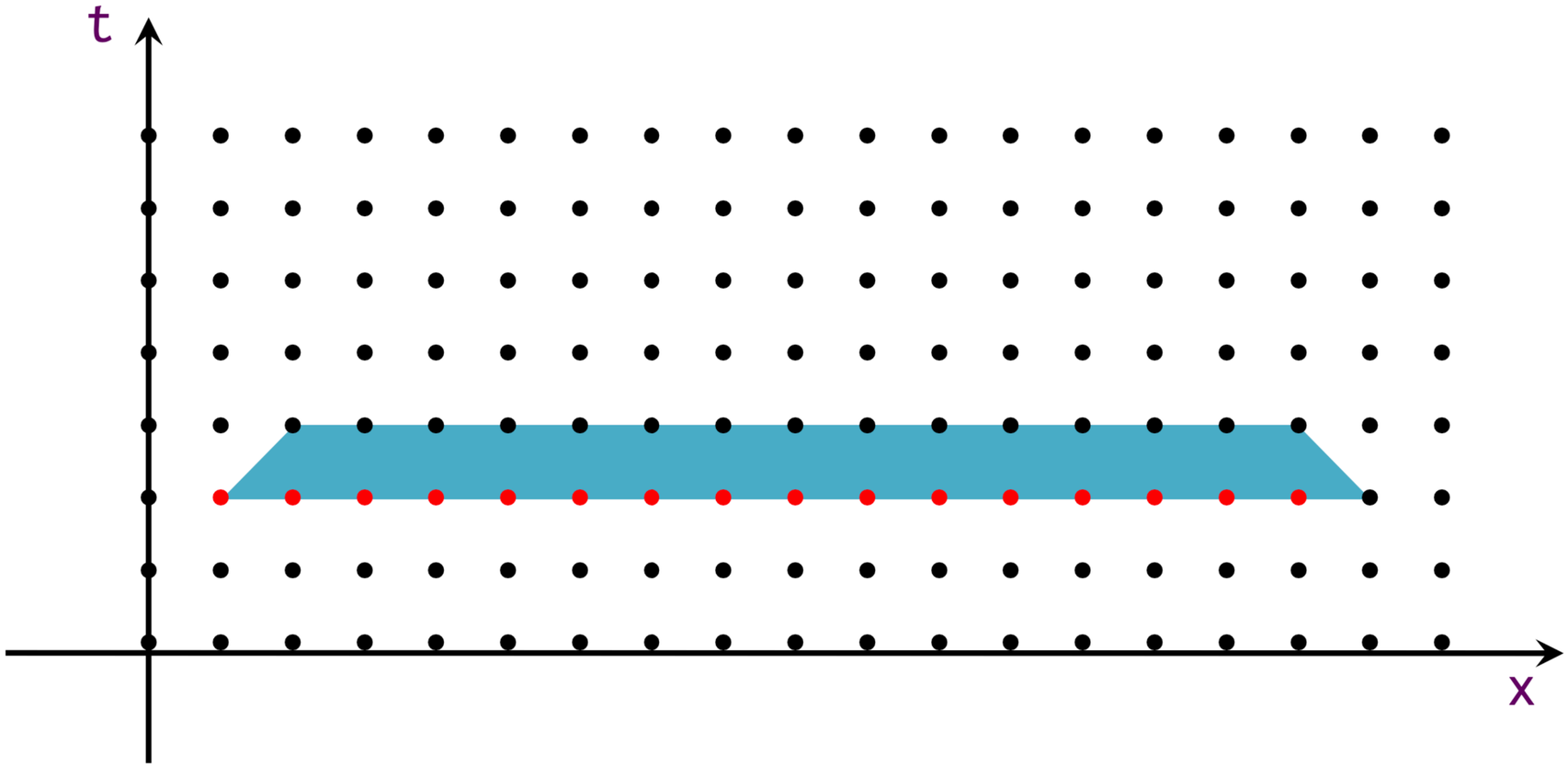
Tall Trapezoid: Time Cut

If width < 2·height, cut the trapezoid with a horizontal line through the center. Traverse the bottom trapezoid first, and then the top one.



Base Case

If height = 1, compute all space-time points in the trapezoid. Any order of computation is valid, since no point depends on another.

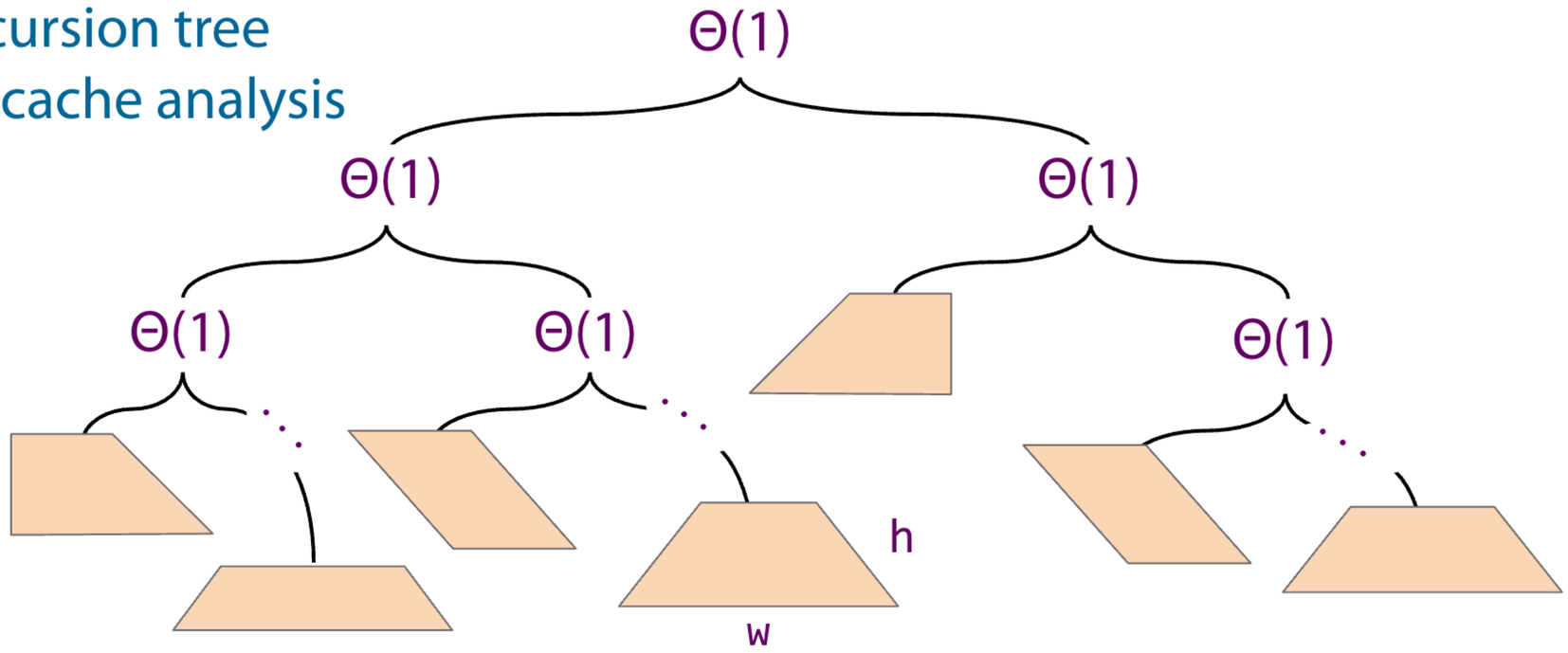


C Implementation

```
void trapezoid(int64_t t0, int64_t t1, int64_t x0, int64_t dx0,
               int64_t x1, int64_t dx1)
{
  int64_t lt = t1 - t0;
  if (lt == 1) { //base case
      for (int64_t x = x0; x < x1; x++)
        u[t1%2][x] = kernel( &u[t0%2][x] );
  } else if (lt > 1) {
    if (2 * (x1 - x0) + (dx1 - dx0) * lt >= 4 * lt) { //space cut
      int64_t xm = (2 * (x0 + x1) + (2 + dx0 + dx1) * lt) / 4;
      trapezoid(t0, t1, x0, dx0, xm, -1);
      trapezoid(t0, t1, xm, -1, x1, dx1);
    } else { //time cut
      int64_t halflt = lt / 2;
      trapezoid(t0, t0 + halflt, x0, dx0, x1, dx1);
      trapezoid(t0 + halflt, t1, x0 + dx0 * halflt, dx0,
                x1 + dx1 * halflt, dx1);
    }
  }
}
```

Work and Cache Analysis

Recursion tree
for cache analysis

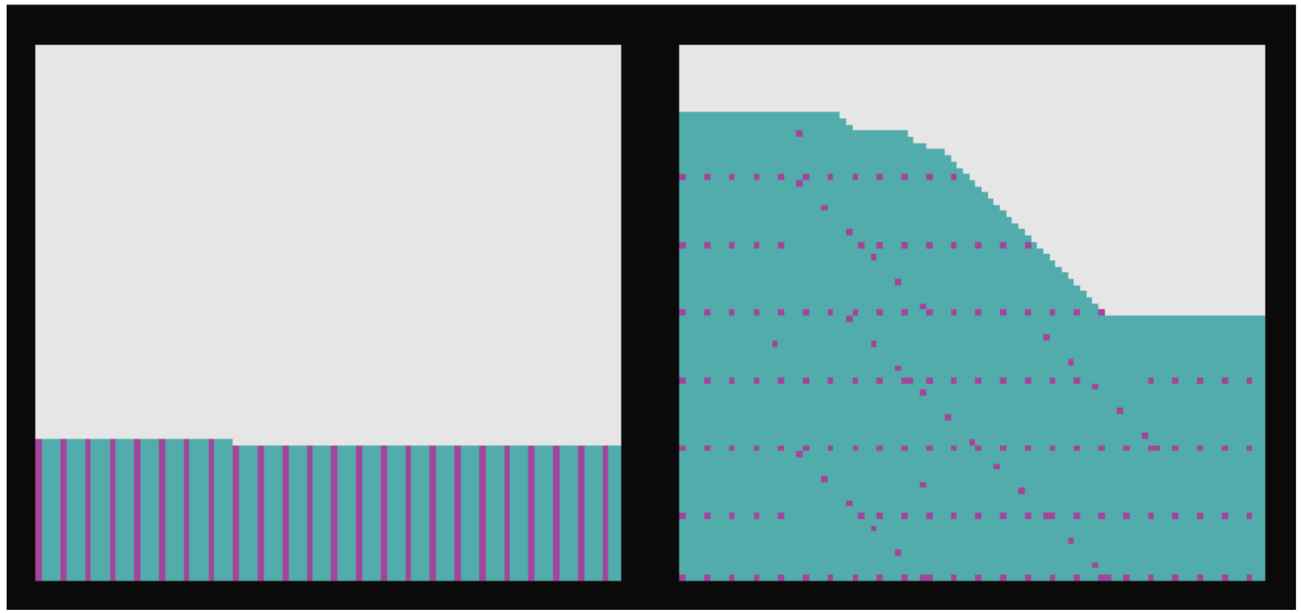


- The bottom of a leaf trapezoid just fits in the cache, so $w = \Theta(\mathcal{M})$.
- A leaf trapezoid contains $\Theta(hw) = \Theta(w^2)$ points and $\Theta(w^2)$ work.
- Since $w \le \mathcal{M}$, a leaf incurs $\Theta(w/\mathcal{B})$ cache misses.
- There are $\Theta(NT/hw) = \Theta(NT/w^2)$ leaves and internal nodes.
- The internal nodes contribute little to both work and cache misses.
- Work $= \Theta(NT/w^2) \cdot \Theta(w^2) = \Theta(NT)$.
- Cache misses $= \Theta(NT/w^2) \cdot \Theta(w/\mathcal{B}) = \Theta(NT/\mathcal{B}w) = \Theta(NT/\mathcal{B}\mathcal{M})$ (*vs.* $\Theta(NT/\mathcal{B})$.)

For d dimensions,
$\Theta(NT/\mathcal{B}\mathcal{M}^{1/d})$

Simulation: 3-Point Stencil

- Rectangular region
  - $N = 95$
  - $T = 87$



Looping Trapezoid

- Fully associative LRU cache
  - $\mathcal{B} = 4$ points
  - $\mathcal{M} = 32$ points
- Cache-hit latency = 1 cycle
- Cache-miss latency = 10 cycles

Empirical Stencil Performance

For an example 2D heat stencil, we see the following performance:

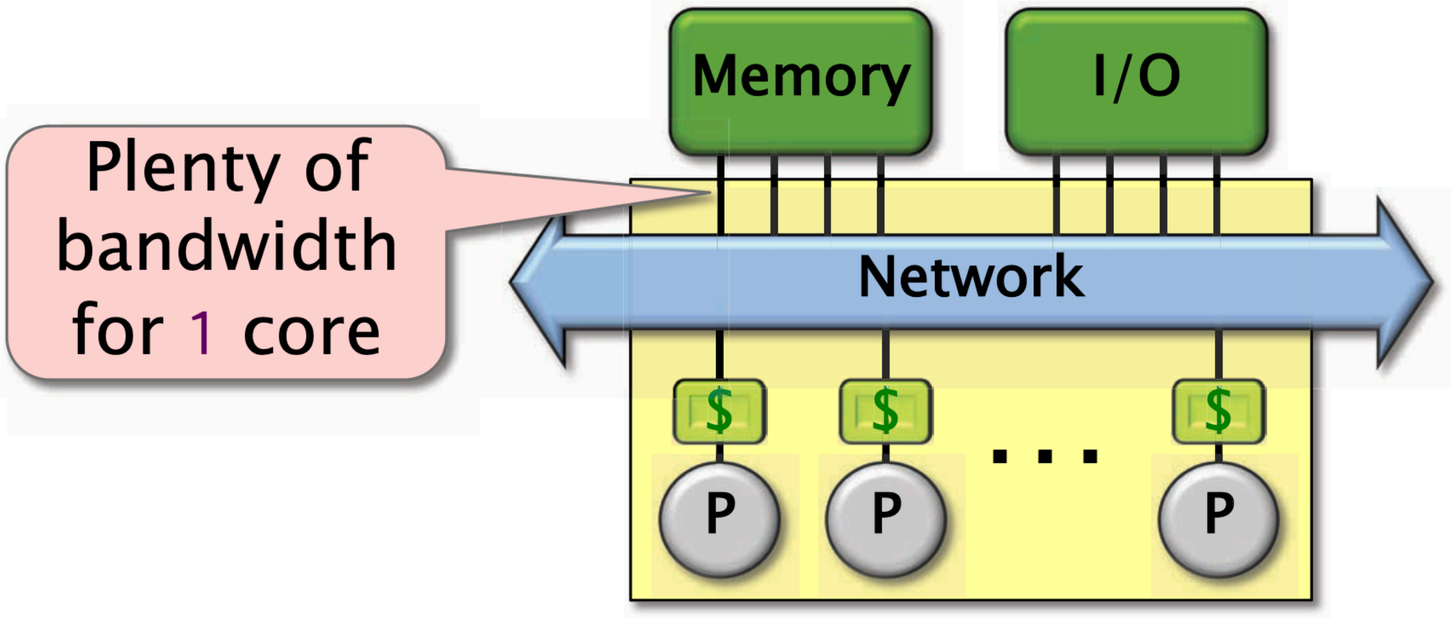
- Looping ~1560 iterations / s
- Trapezoid ~ 1830 iterations / s

What are some reasons that they might be so close together?

Impact on Performance

Q. How can the cache-oblivious trapezoidal decomposition have so many fewer cache misses, but the advantage gained over the looping version be so marginal?

A. Prefetching and a good memory architecture. One core cannot saturate the memory bandwidth.



Caching and Parallelism

Rule of Thumb for Caching and Parallelism

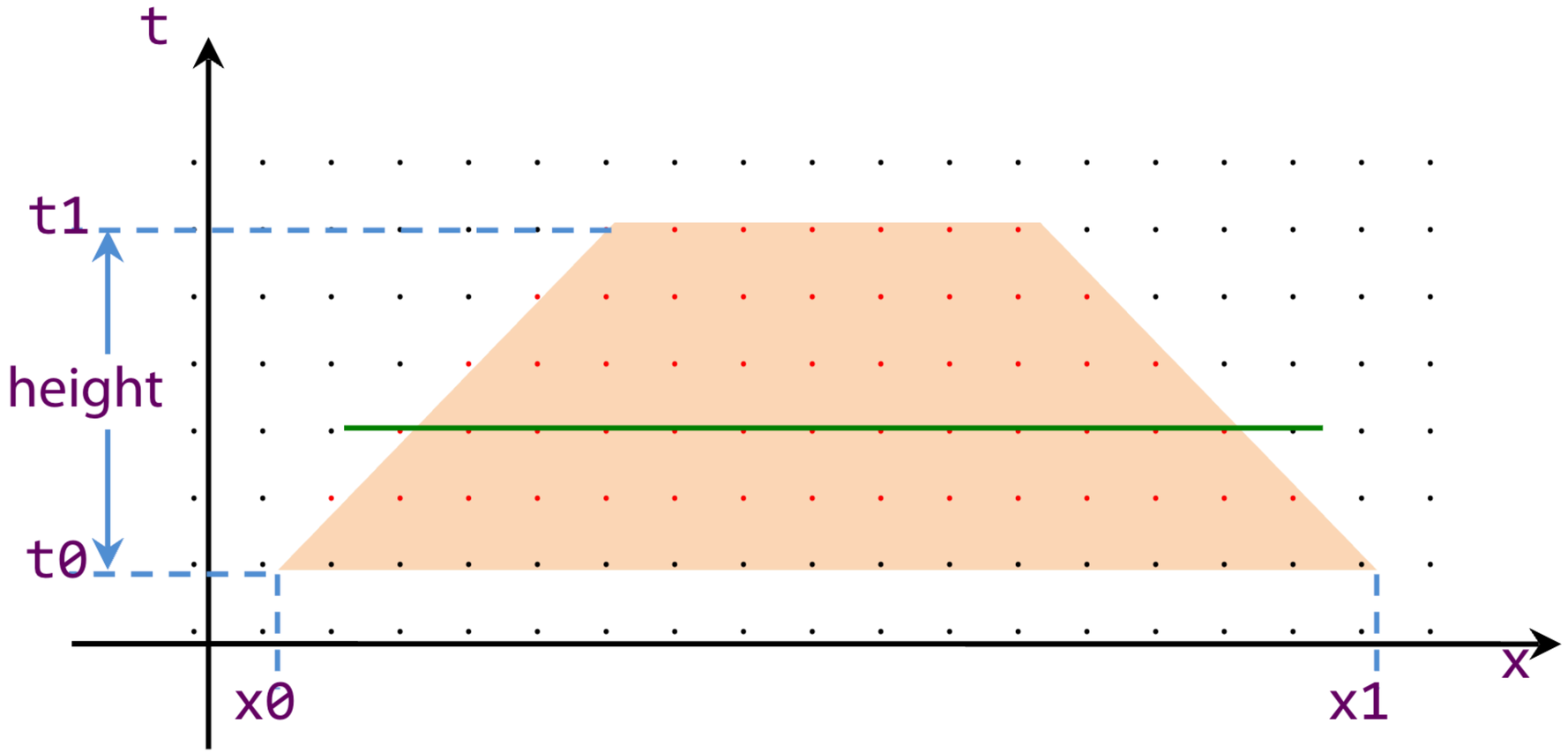
At a high level, minimizing cache misses in the **serial projection** minimizes them in parallel executions.

Serial projection is just the program minus the parallel constructs (spawn, sync, pfor)

cache misses in parallel $\geq$ cache misses in serial

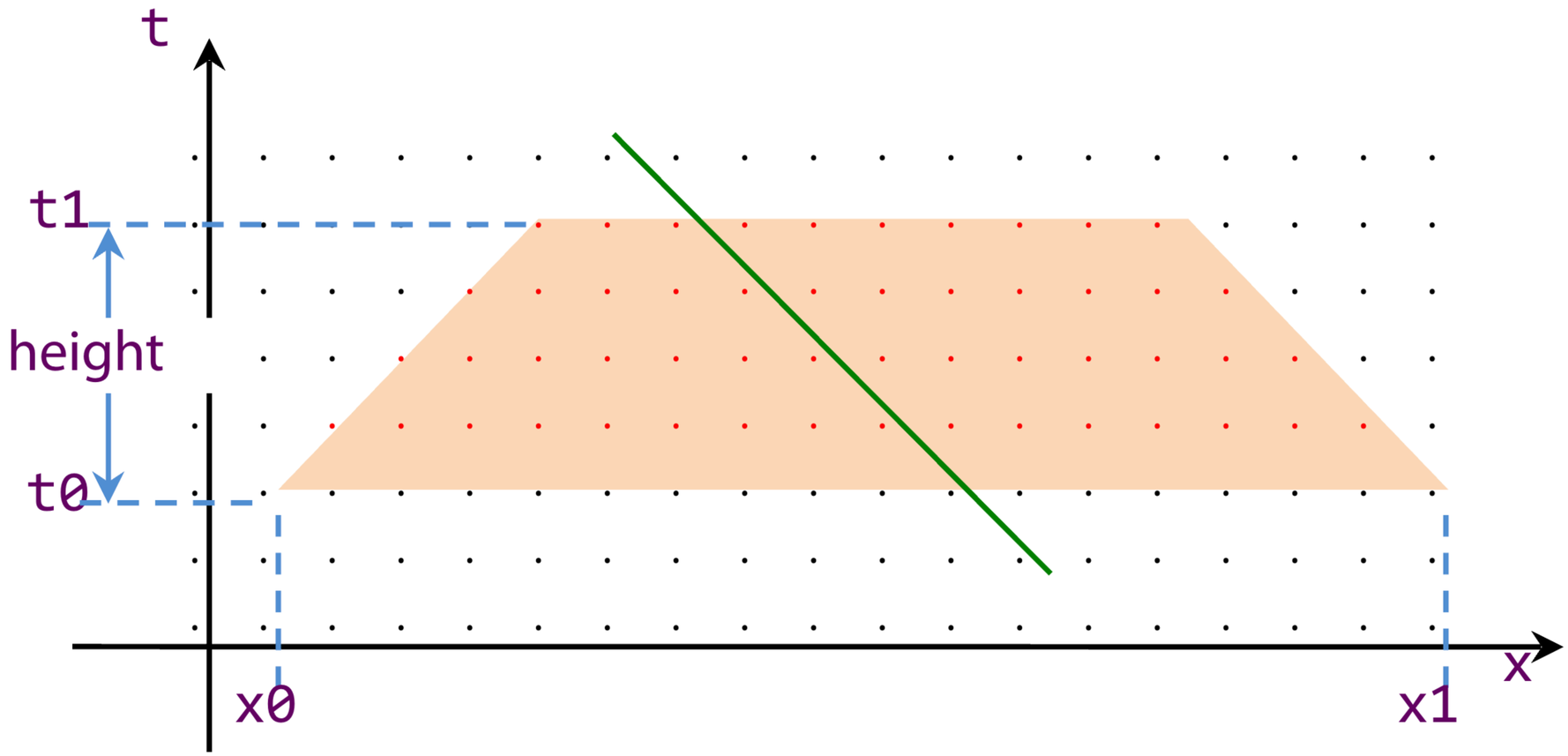
Time Cuts Don’t Parallelize

There’s no way to parallelize a time cut. The bottom trapezoid must be traversed first, and then the top one.

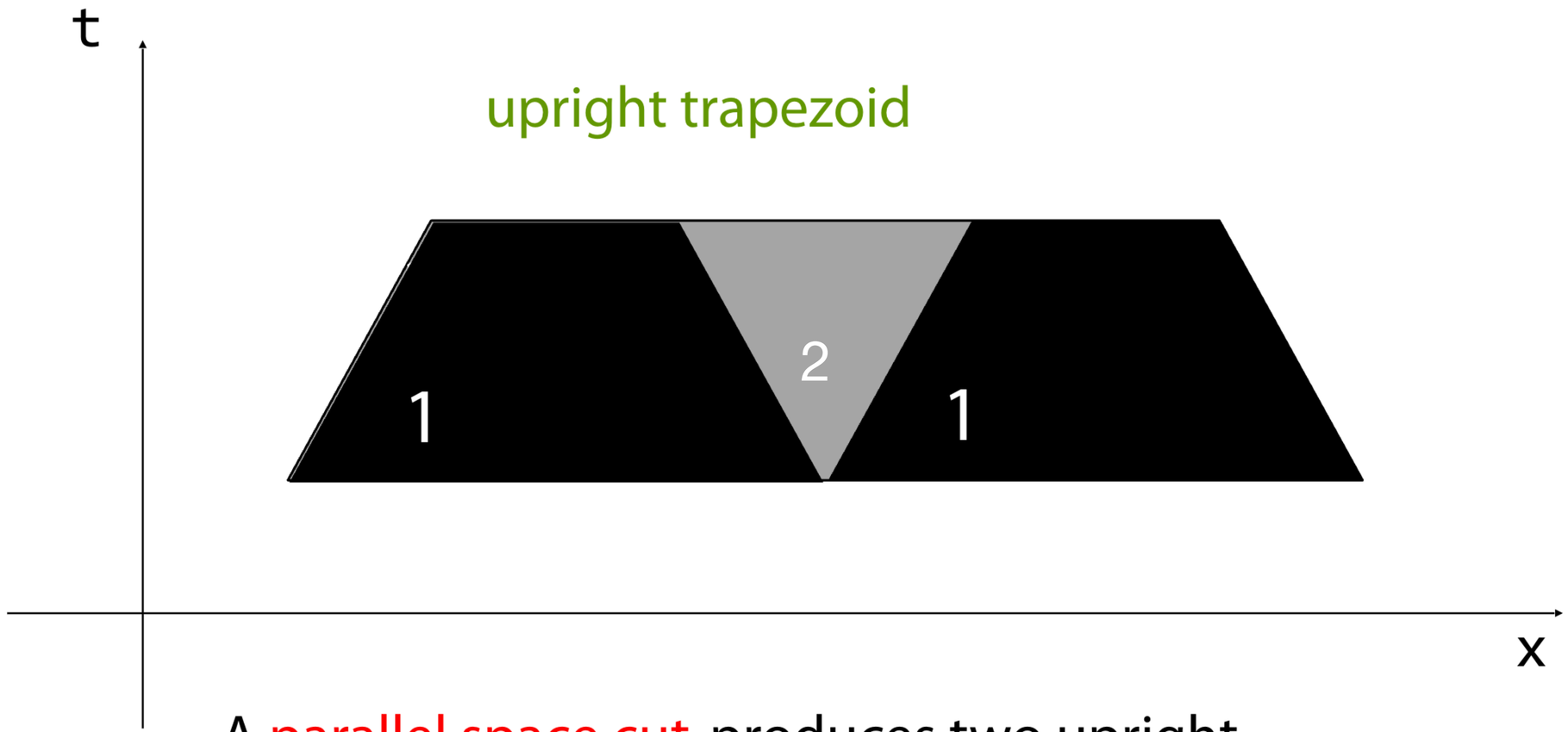


Space Cuts Don't Parallelize, or Do They?

A space cut poses a similar problem. You must traverse the trapezoid on the left before you can traverse the one on the right.

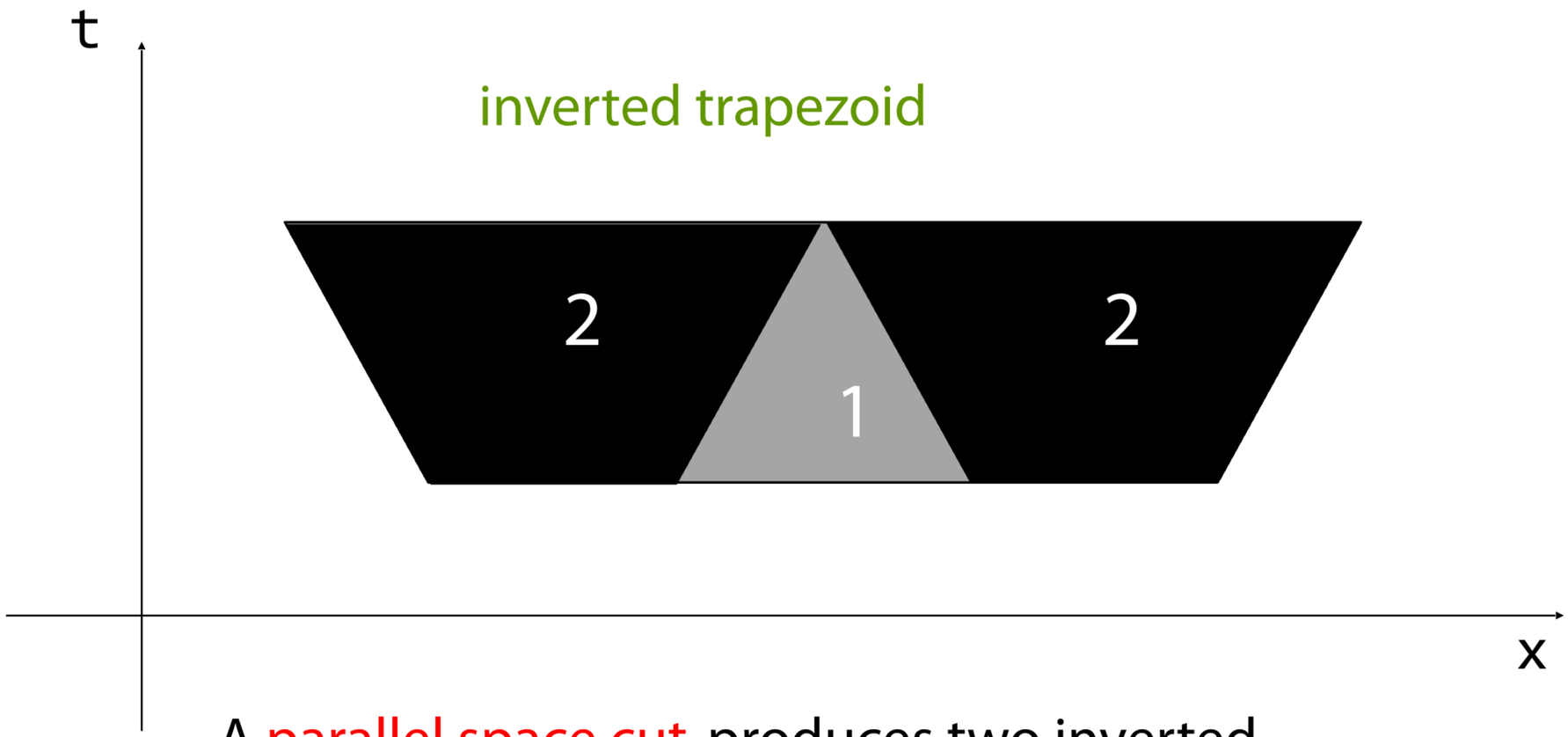


Parallel Space Cuts



A parallel space cut produces two upright trapezoids (black) that can be executed in parallel and a third "inverted" trapezoid (gray) that must execute in series after the two upright trapezoids.

Parallel Space Cuts



A parallel space cut produces two inverted trapezoids (black) that can be executed in parallel and a third upright trapezoid (gray) that must execute in series before the inverted trapezoids.

Performance Comparison

Heat equation on a 3000×3000 grid for 1000 time steps (4 processor cores with 8MB LLC)

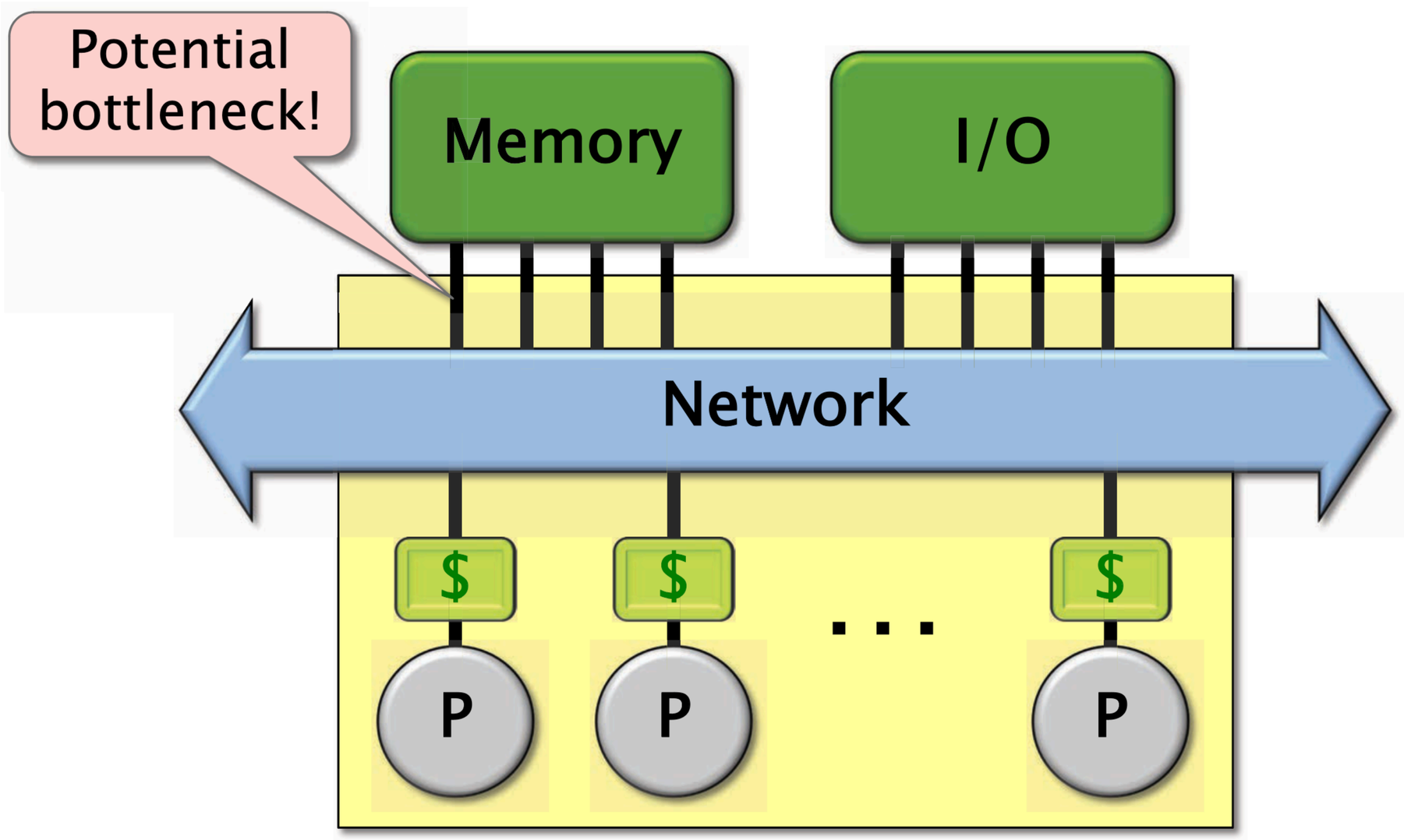
Code	Time
Serial looping	128.95s
Parallel looping	66.97s
Serial trapezoidal	66.76s
Parallel trapezoidal	16.86s

} 1.93x

} 3.96x

The parallel looping code achieves less than half the potential speedup, even though it has far more parallelism.

Memory Bandwidth



Impediments to Speedup

- Insufficient parallelism

Can get parallelism by computing work and span

- Scheduling overhead

Usually ok if your work chunks are big enough / have low span

- Lack of memory bandwidth

Can run **P identical copies** of the serial code in parallel - if you have enough memory

- Contention (locking and true/false sharing)

Tools exist to detect lock contention in an execution, but not the potential for lock contention. Potential for true/false sharing is even harder to detect.

Cache-Oblivious Sorting

Merging Two Sorted Arrays

```
void merge(int *C, int *A, int na, int *B, int nb) {
  while (na > 0 && nb > 0) {
    if (*A <= *B) {
      *C++ = *A++, na--;
    } else {
      *C++ = *B++, nb–;
    }
  }
  while (na > 0) {
    *C++ = *A++, na--;
  }
  while (nb > 0) {
    *C++ = *B++, nb–;
  }
}
```

Work to merge n elements = $\Theta(n)$

Number of cache misses = $\Theta(n/\mathscr{B})$

Merge Sort

```
void merge_sort(int *B, int *A, int n) {
  if (n == 1) {
    B[0] = A[0];
  } else {
    int C[n];
    parallel_spawn merge_sort(C, A, n/2);
    merge_sort(C+n/2, A+n/2, n-n/2);
    parallel_sync;
    merge(B, C, n/2, C+n/2, n-n/2);
  }
}
```

Work: $W(n) = 2W(n/2) + \Theta(n) = \Theta(n \lg n)$

Solve

$$W(n) = 2W(n/2) + \Theta(n)$$

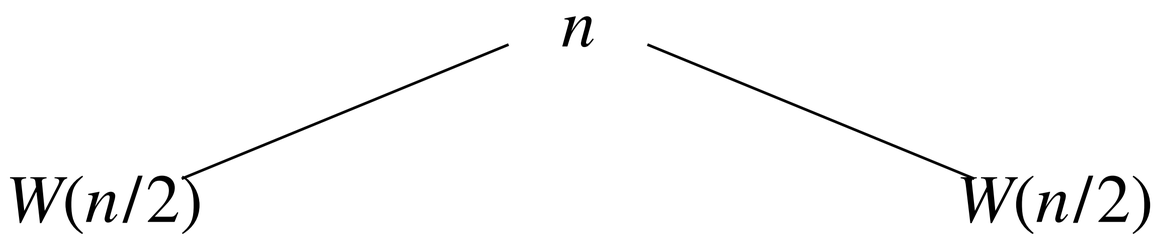
Recursion Tree

$W(n)$

Solve

$$W(n) = 2W(n/2) + \Theta(n)$$

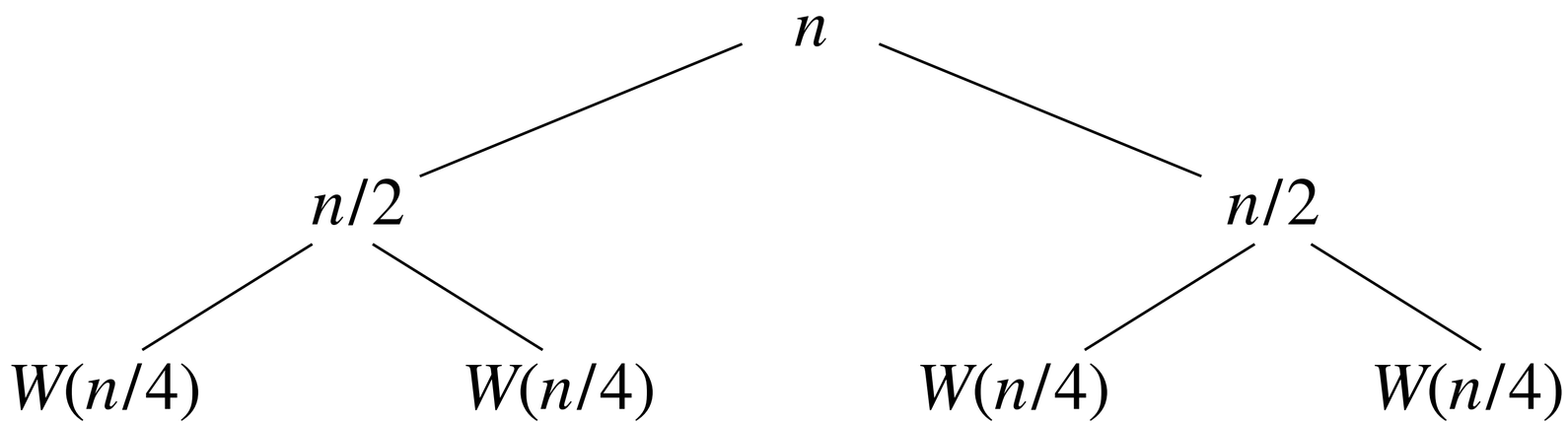
Recursion Tree



Solve

$$W(n) = 2W(n/2) + \Theta(n)$$

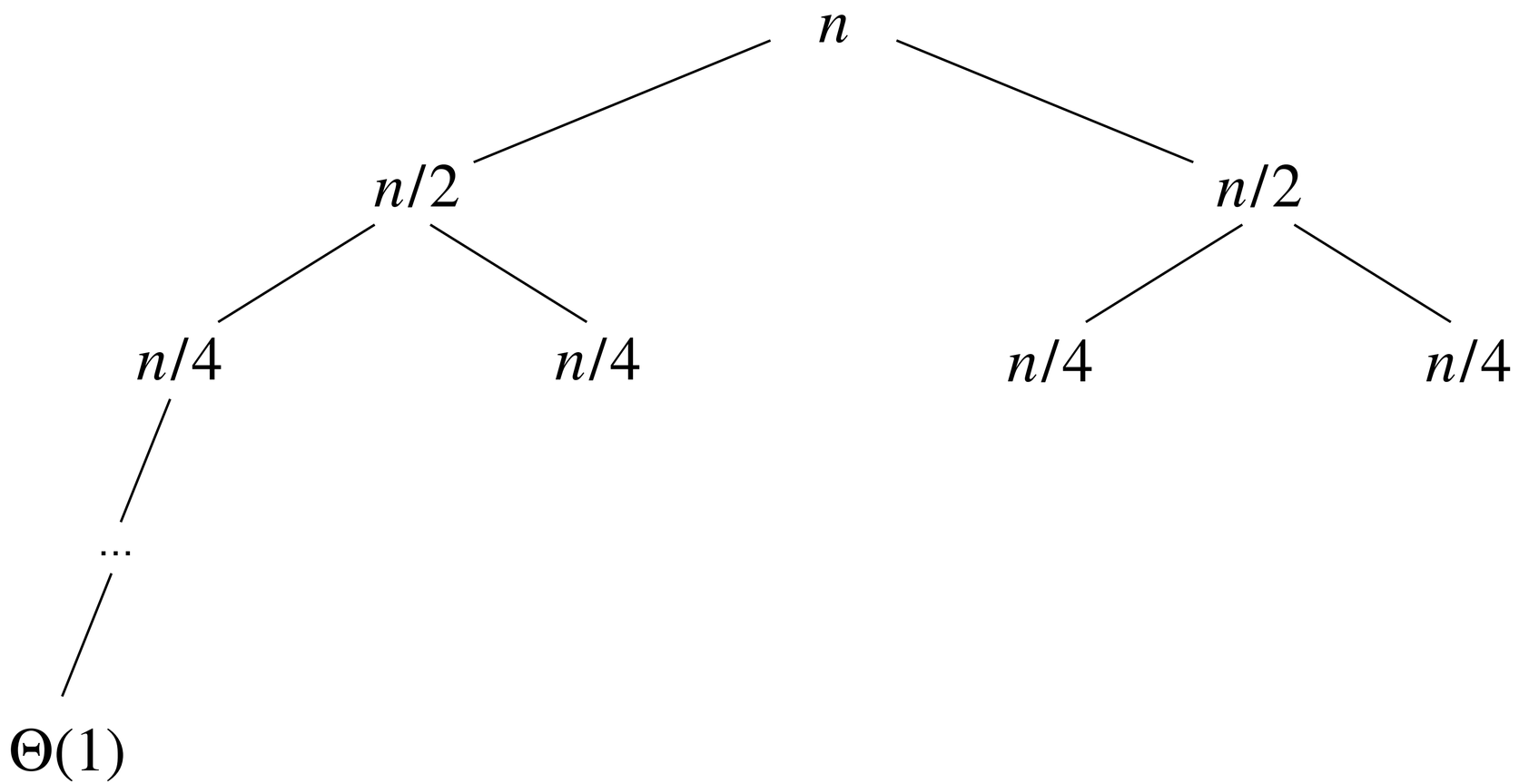
Recursion Tree



Solve

$$W(n) = 2W(n/2) + \Theta(n)$$

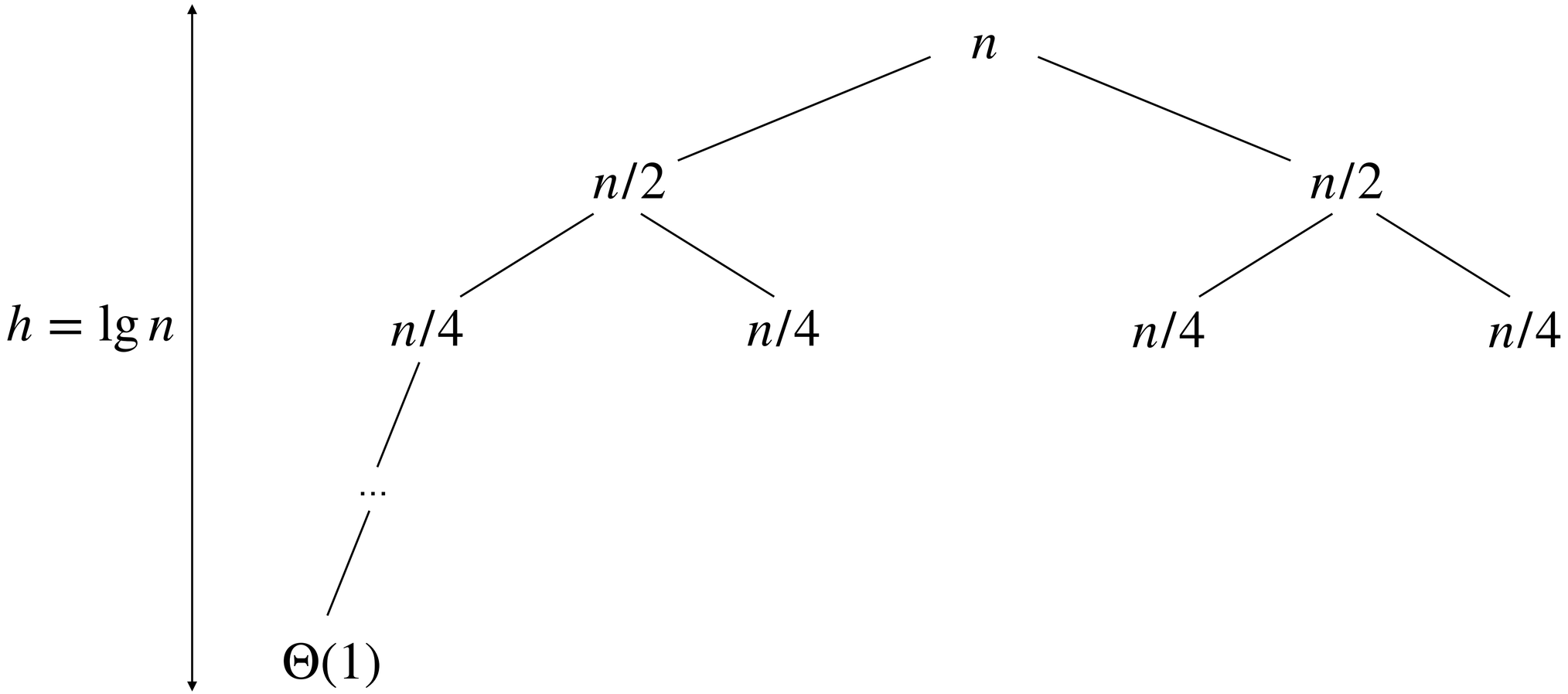
Recursion Tree



Solve

$$W(n) = 2W(n/2) + \Theta(n)$$

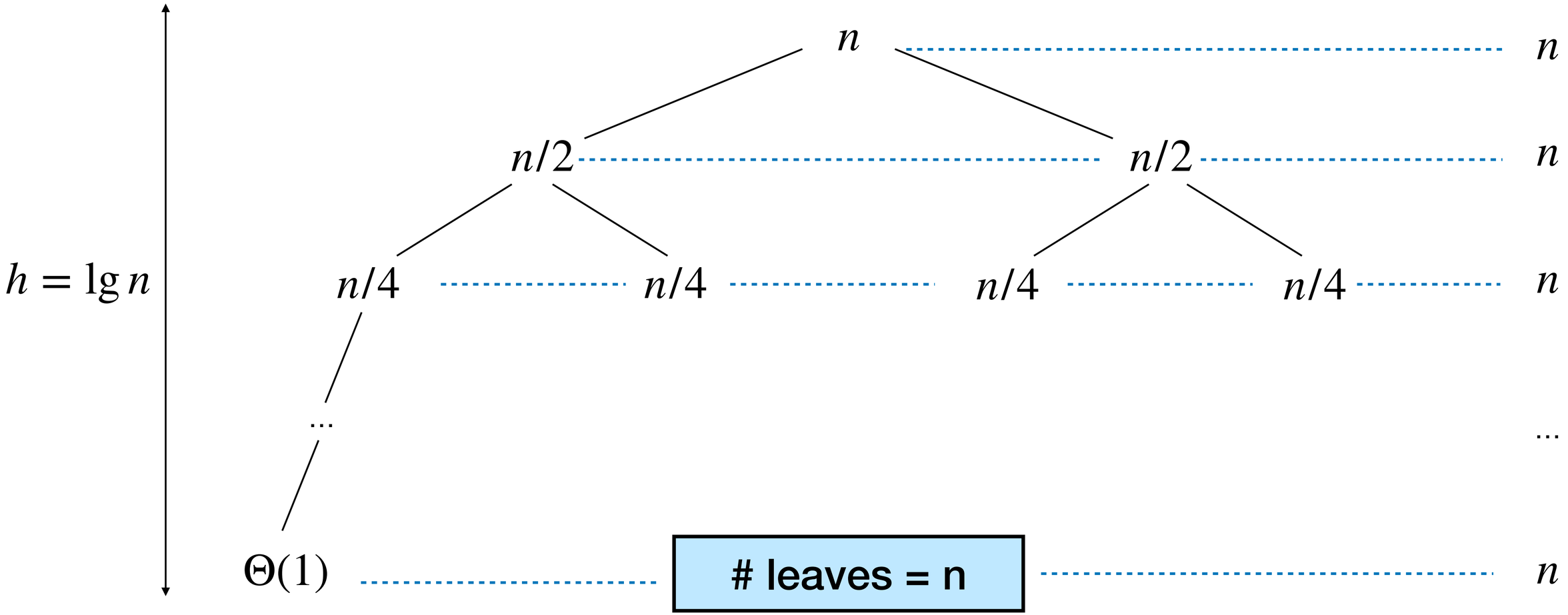
Recursion Tree



Solve

$$W(n) = 2W(n/2) + \Theta(n)$$

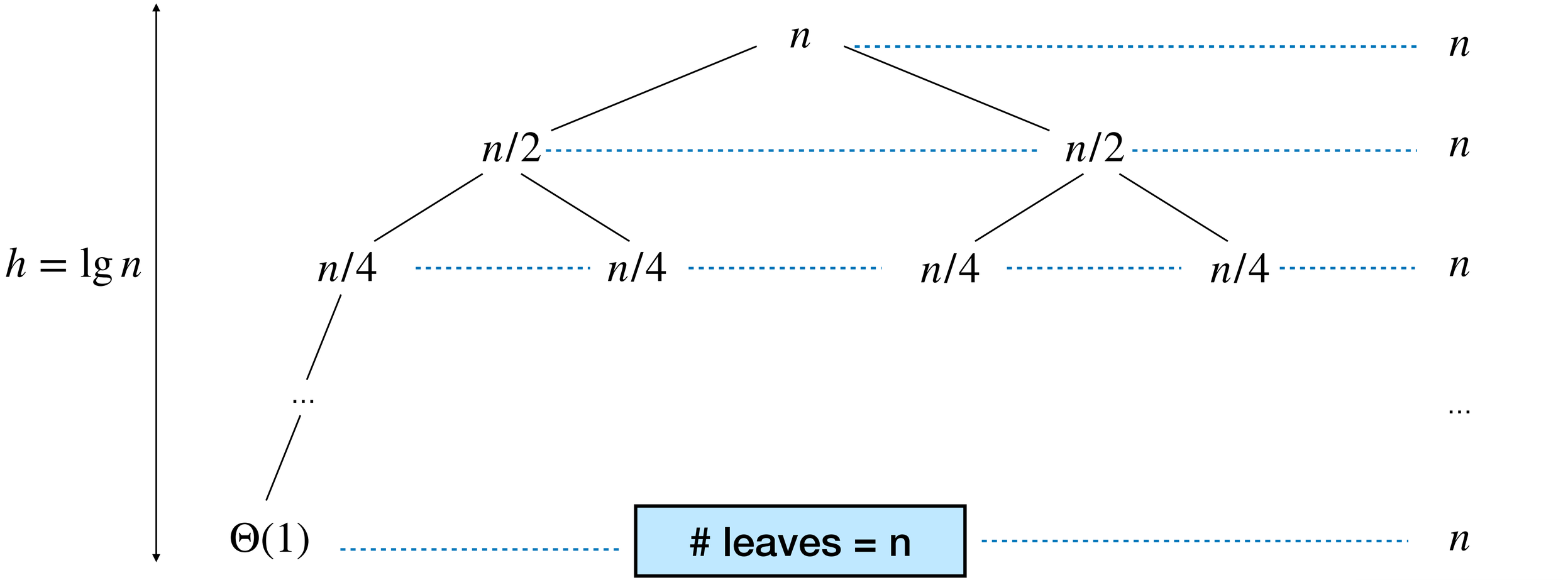
Recursion Tree



Solve

$$W(n) = 2W(n/2) + \Theta(n)$$

Recursion Tree



$$W(n) = \Theta(n \lg n)$$

Now with Caching

Merge subroutine

$$Q(n) = \Theta(n/\mathscr{B})$$

Merge sort

$$Q(n) = \begin{cases} \Theta(n/\mathscr{B}) & \text{if } n \leq c\mathscr{M}\text{, constant } c \leq 1 \\ 2Q(n/2) + \Theta(n/B) & \text{otherwise} \end{cases}$$

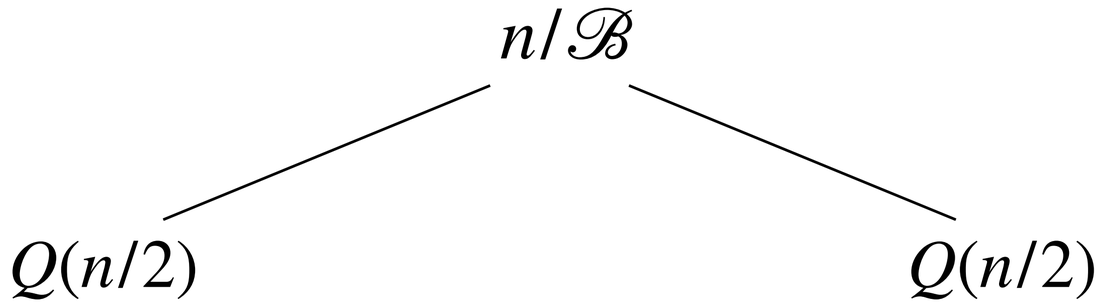
Cache Analysis of Merge Sort

$$Q(n) = \begin{cases} \Theta(n/\mathscr{B}) & \text{if } n \leq c\mathscr{M}\text{, constant } c \leq 1 \\ 2Q(n/2) + \Theta(n/B) & \text{otherwise} \end{cases}$$

$$Q(n)$$

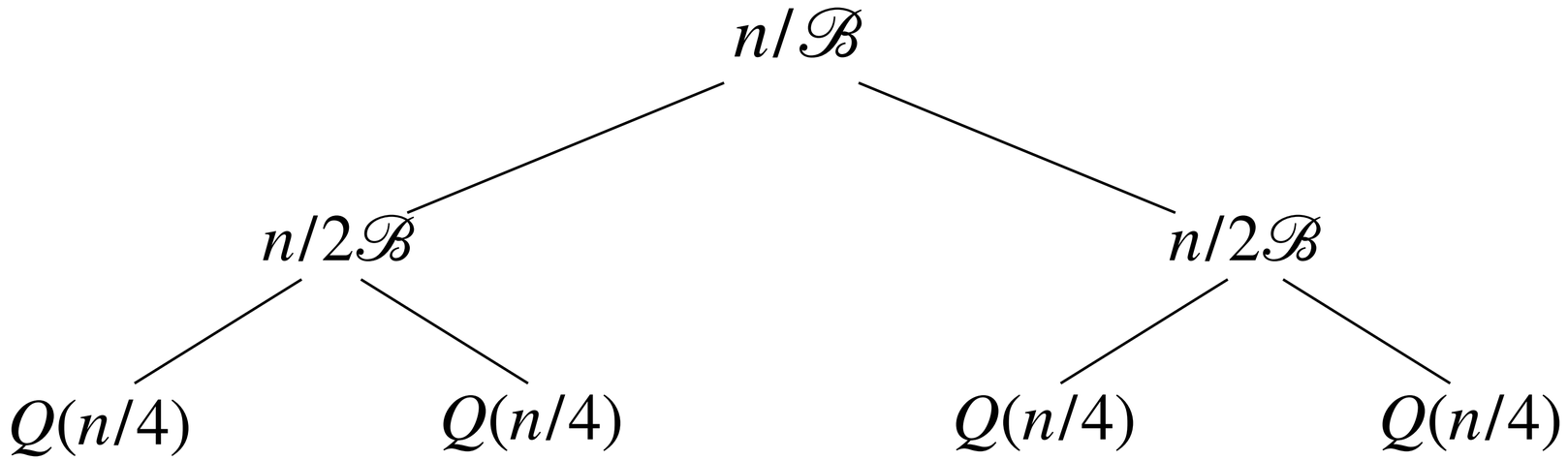
Cache Analysis of Merge Sort

$$Q(n) = \begin{cases} \Theta(n/\mathscr{B}) & \text{if } n \leq c\mathscr{M}\text{, constant } c \leq 1 \\ 2Q(n/2) + \Theta(n/B) & \text{otherwise} \end{cases}$$



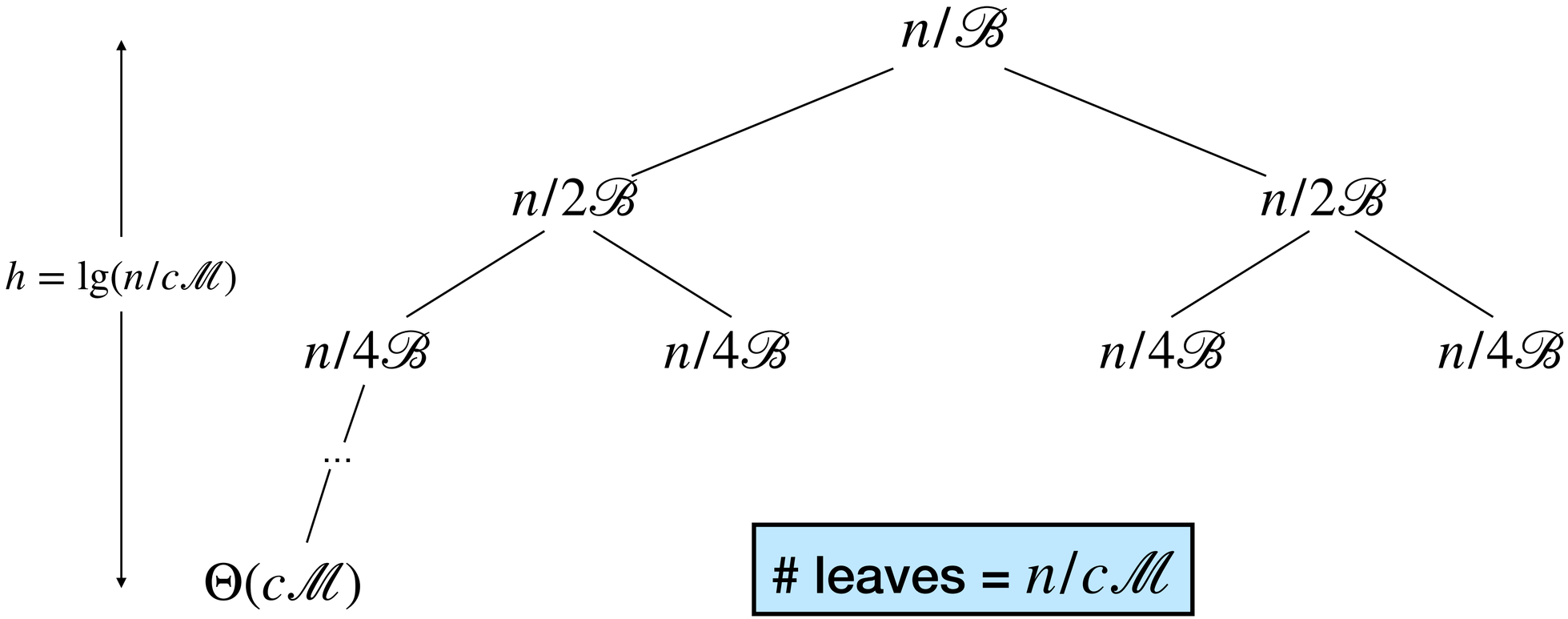
Cache Analysis of Merge Sort

$$Q(n) = \begin{cases} \Theta(n/\mathscr{B}) & \text{if } n \leq c\mathscr{M}, \text{ constant } c \leq 1 \\ 2Q(n/2) + \Theta(n/B) & \text{otherwise} \end{cases}$$



Cache Analysis of Merge Sort

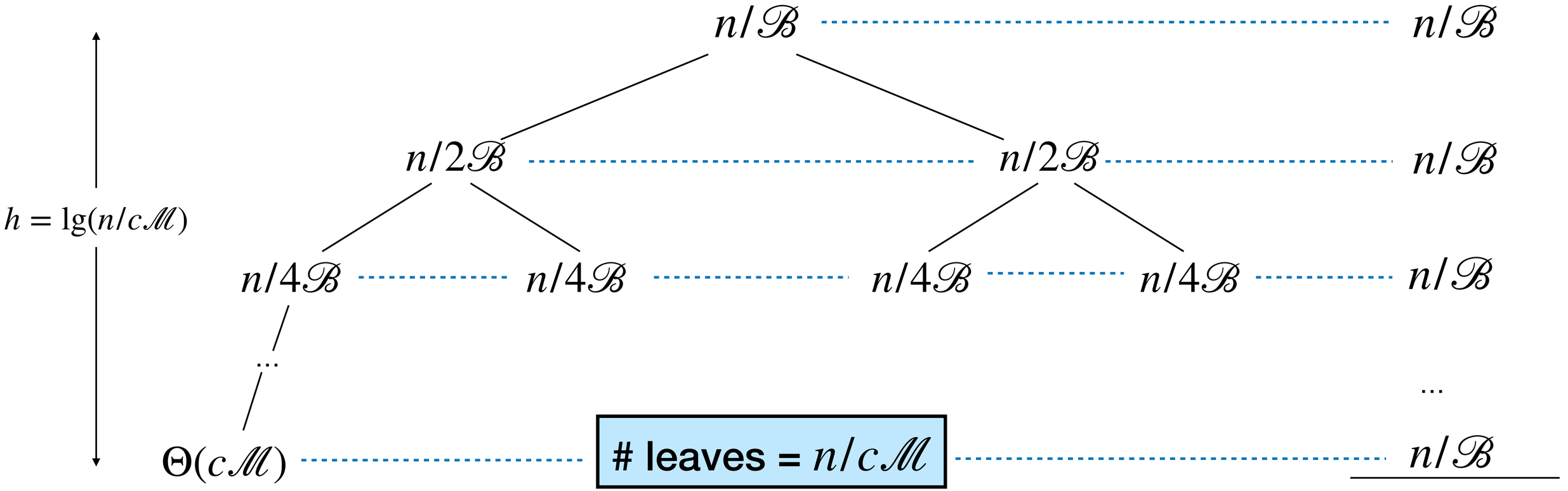
$$Q(n) = \begin{cases} \Theta(n/\mathscr{B}) & \text{if } n \le c\mathscr{M}, \text{ constant } c \le 1 \\ 2Q(n/2) + \Theta(n/B) & \text{otherwise} \end{cases}$$



 # leaves = $n/c\mathscr{M}$

Cache Analysis of Merge Sort

$$Q(n) = \begin{cases} \Theta(n/\mathscr{B}) & \text{if } n \le c\mathscr{M}, \text{ constant } c \le 1 \\ 2Q(n/2) + \Theta(n/B) & \text{otherwise} \end{cases}$$



$$Q(n) = \Theta((n/\mathscr{B})\lg(n/\mathscr{M}))$$

Bottom line for merge sort

$$Q(n) = \begin{cases} \Theta(n/\mathscr{B}) & \text{if } n \le c\mathscr{M}, \text{ constant } c \le 1 \\ 2Q(n/2) + \Theta(n/B) & \text{otherwise} \end{cases}$$

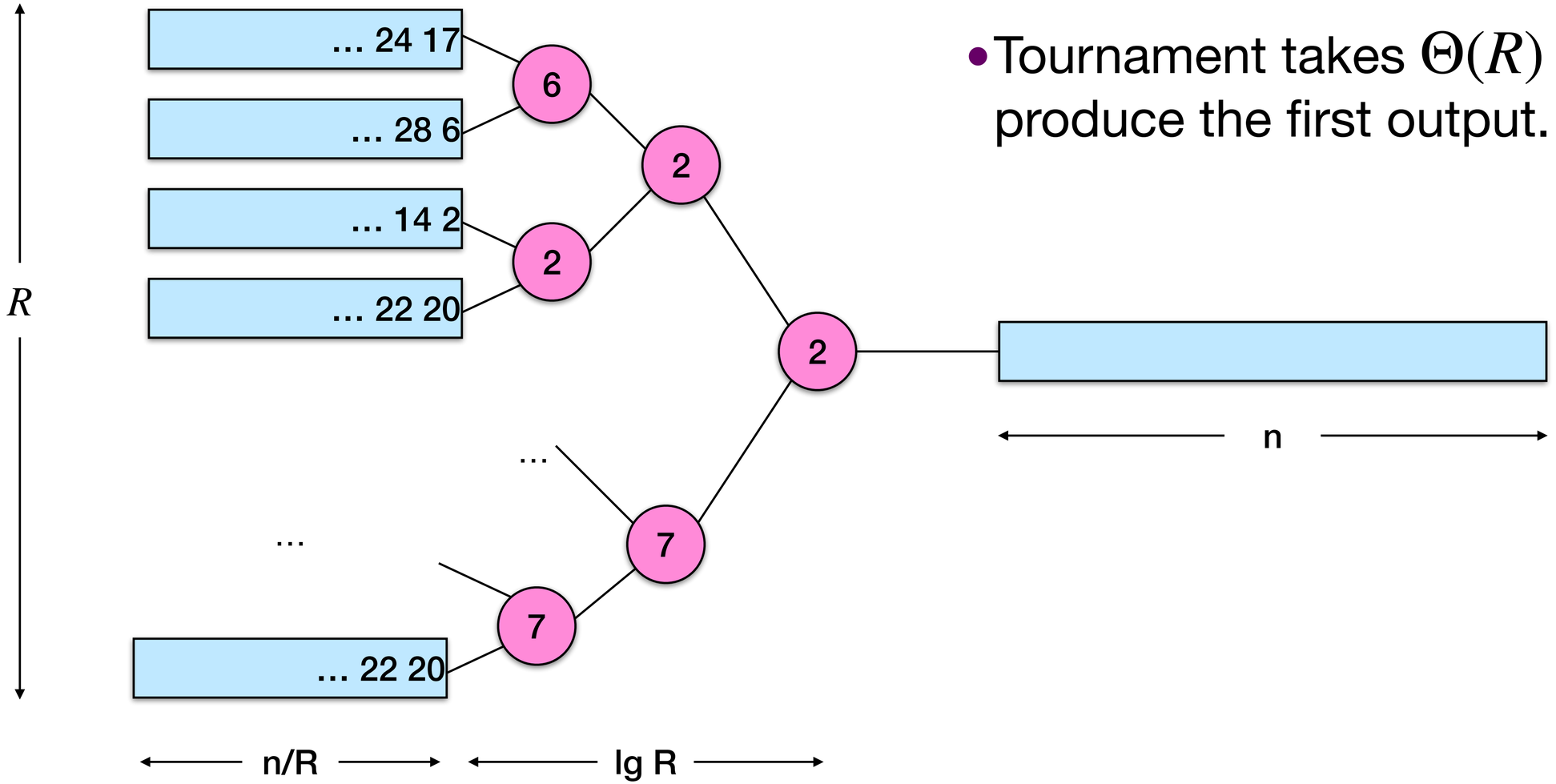
$$Q(n) = \Theta((n/\mathscr{B})\lg(n/\mathscr{M}))$$

- For $n \gg \mathscr{M}$, we have $\lg(n/\mathscr{M}) \approx \lg n$, so $W(n)/Q(n) \approx \Theta(\mathscr{B})$.
- For $n \approx \mathscr{M}$, we have $\lg(n/\mathscr{M}) \approx \Theta(1)$, so $W(n)/Q(n) \approx \Theta(\mathscr{B} \lg n)$.

Can we do better?

Multiway merging

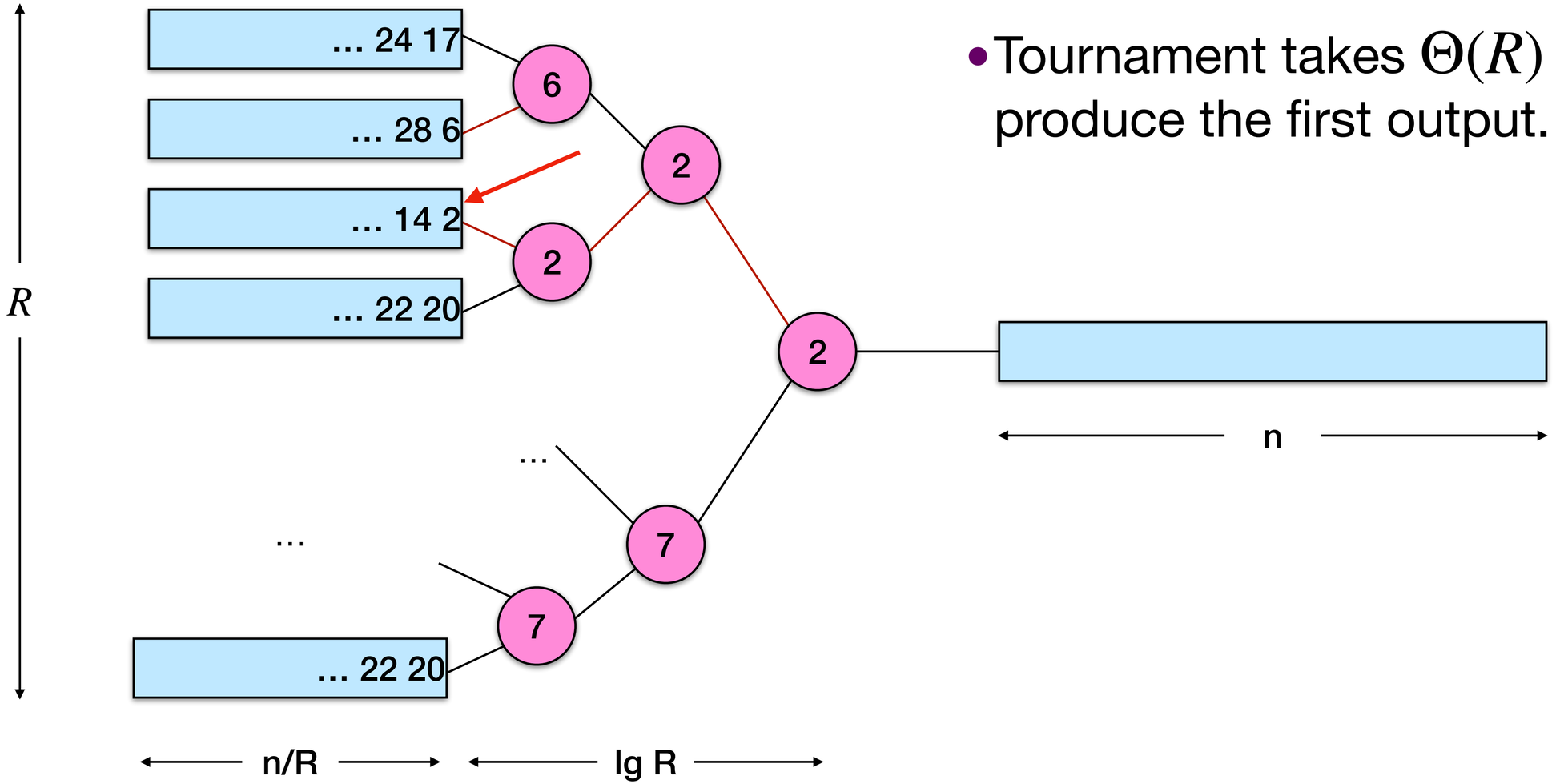
Idea: Merge R < n subarrays with a tournament.



- Tournament takes $\Theta(R)$ work to produce the first output.

Multiway merging

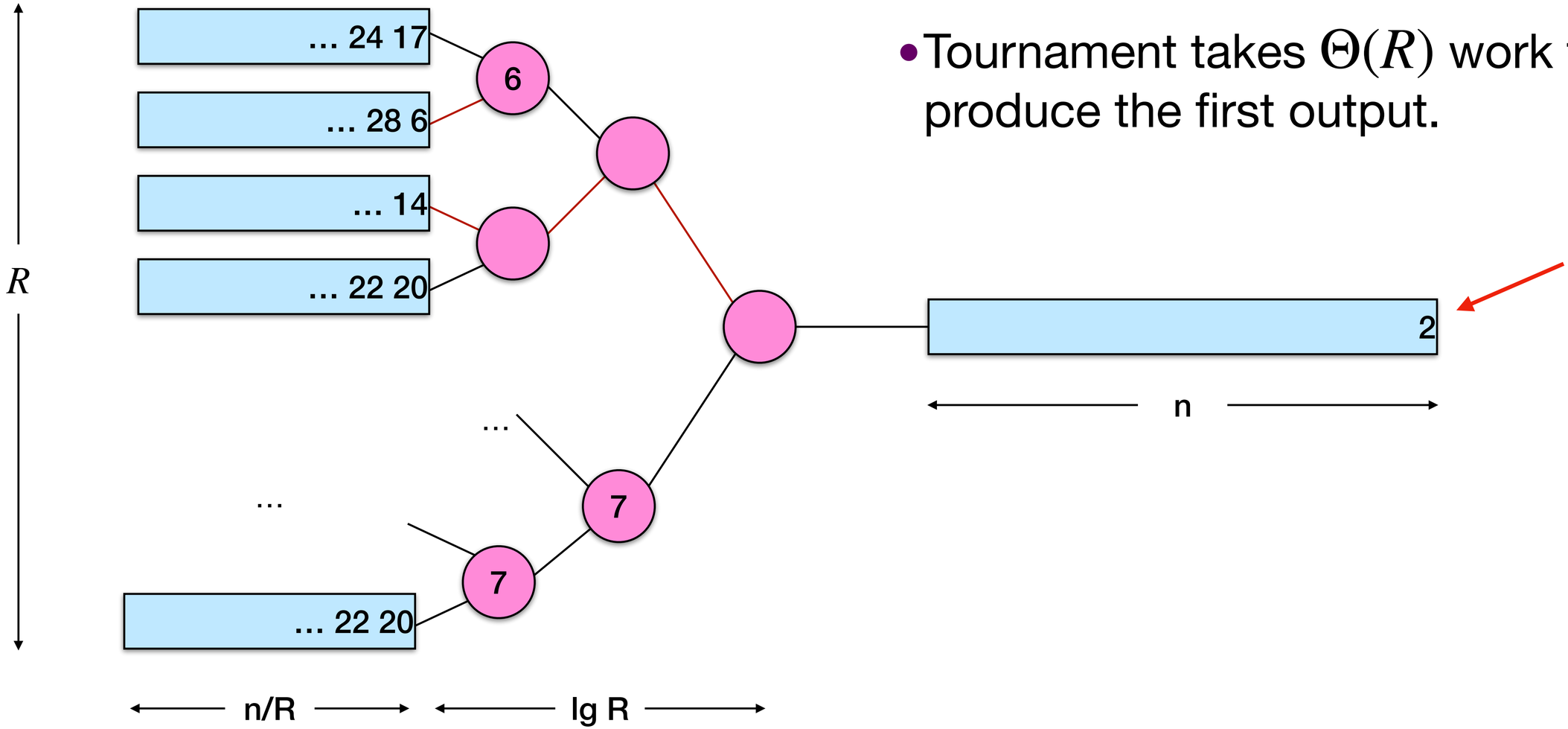
Idea: Merge R < n subarrays with a tournament.



- Tournament takes $\Theta(R)$ work to produce the first output.

Multiway merging

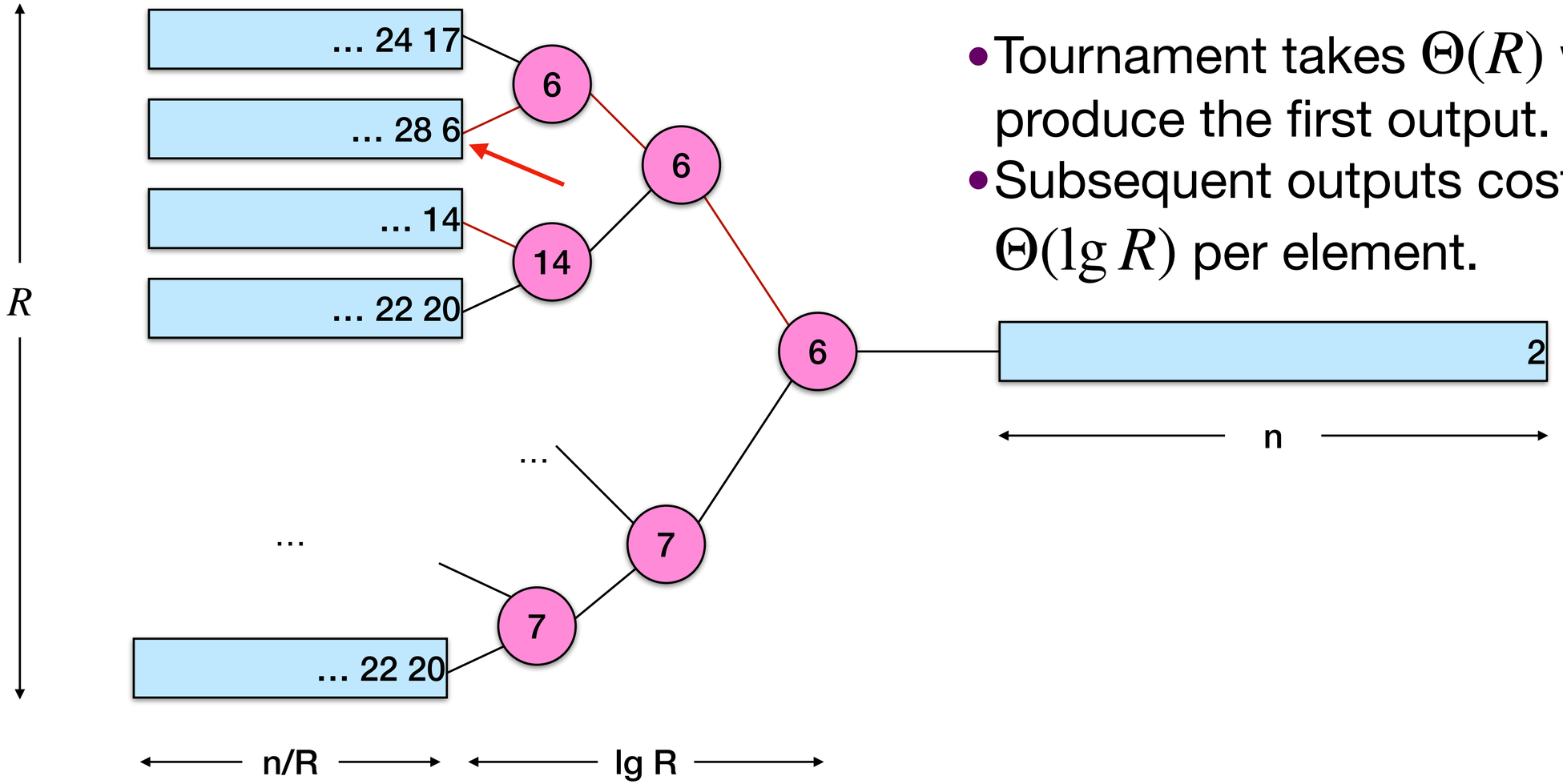
Idea: Merge R < n subarrays with a tournament.



- Tournament takes $\Theta(R)$ work to produce the first output.

Multiway merging

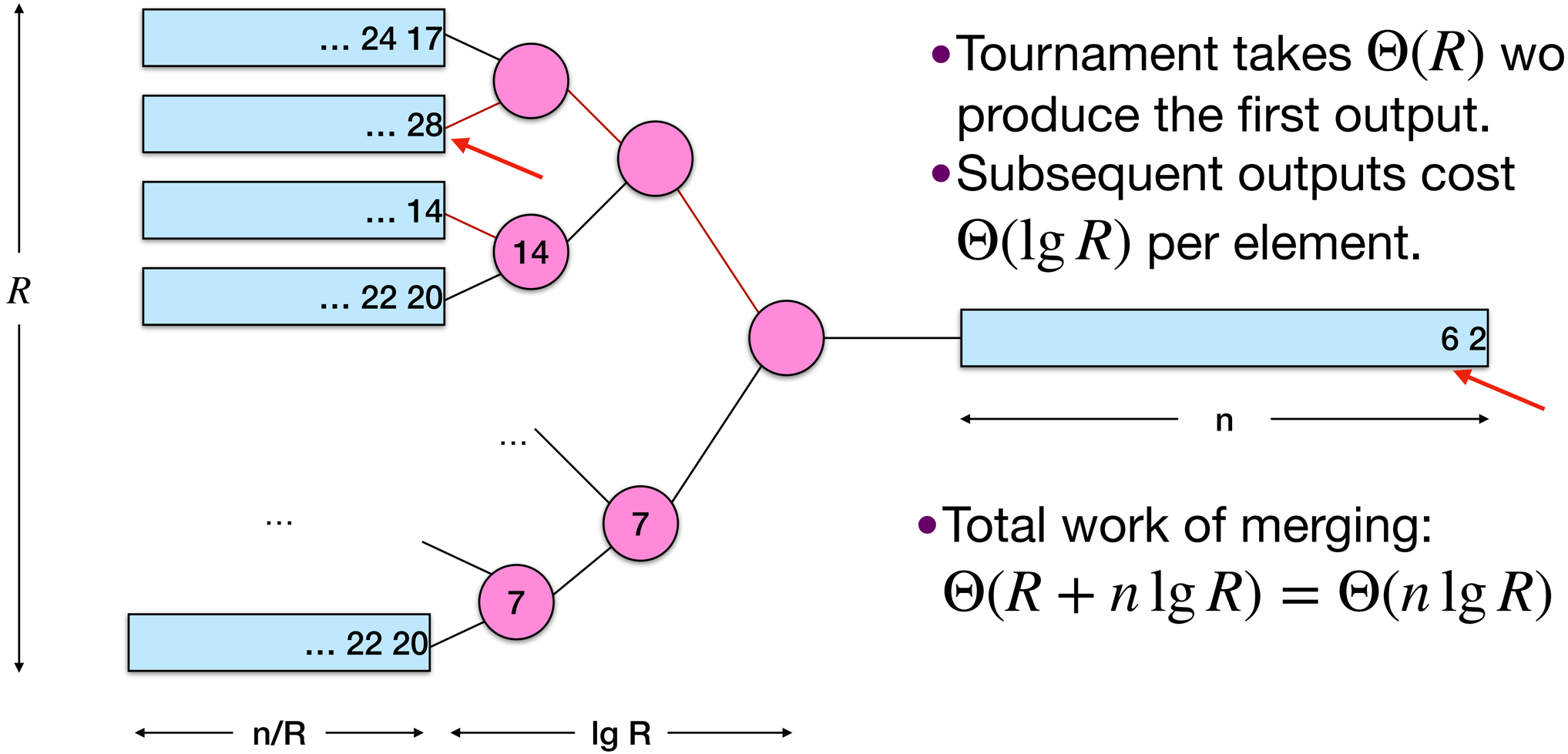
Idea: Merge R < n subarrays with a tournament.



- Tournament takes $\Theta(R)$ work to produce the first output.
- Subsequent outputs cost $\Theta(\lg R)$ per element.

Multiway merging

Idea: Merge R < n subarrays with a tournament.



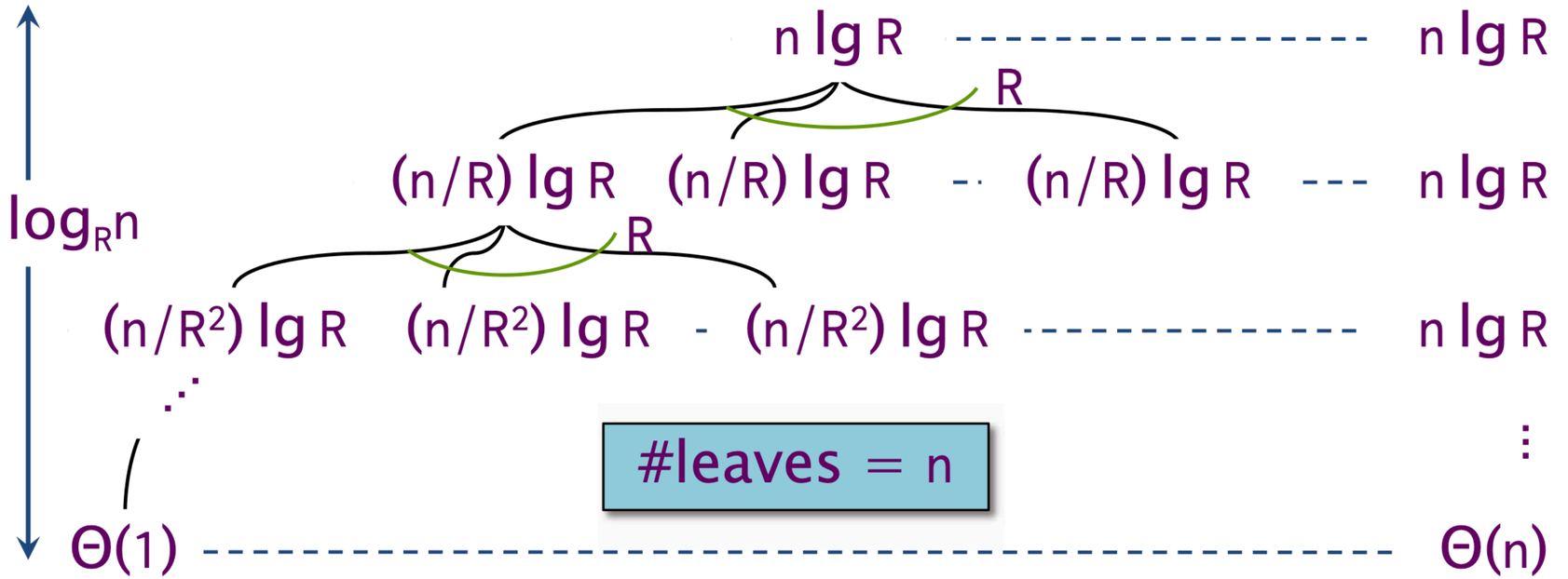
- Tournament takes $\Theta(R)$ work to produce the first output.
- Subsequent outputs cost $\Theta(\lg R)$ per element.

- Total work of merging: $\Theta(R + n \lg R) = \Theta(n \lg R)$

Work of Multiway Merge Sort

$$W(n) = \begin{cases} \Theta(1) & \text{if } n = 1; \\ R \cdot W(n/R) + \Theta(n \lg R) & \text{otherwise.} \end{cases}$$

Recursion tree



$$\begin{aligned} W(n) &= \Theta((n \lg R) \log_R n + n) \\ &= \Theta((n \lg R)(\lg n)/\lg R + n) \\ &= \Theta(n \lg n) \end{aligned}$$

Same as binary merge sort.

Caching Recurrence

Assume that we have $R < c\mathcal{M}/\mathcal{B}$ for a sufficiently small constant $c \leq 1$.

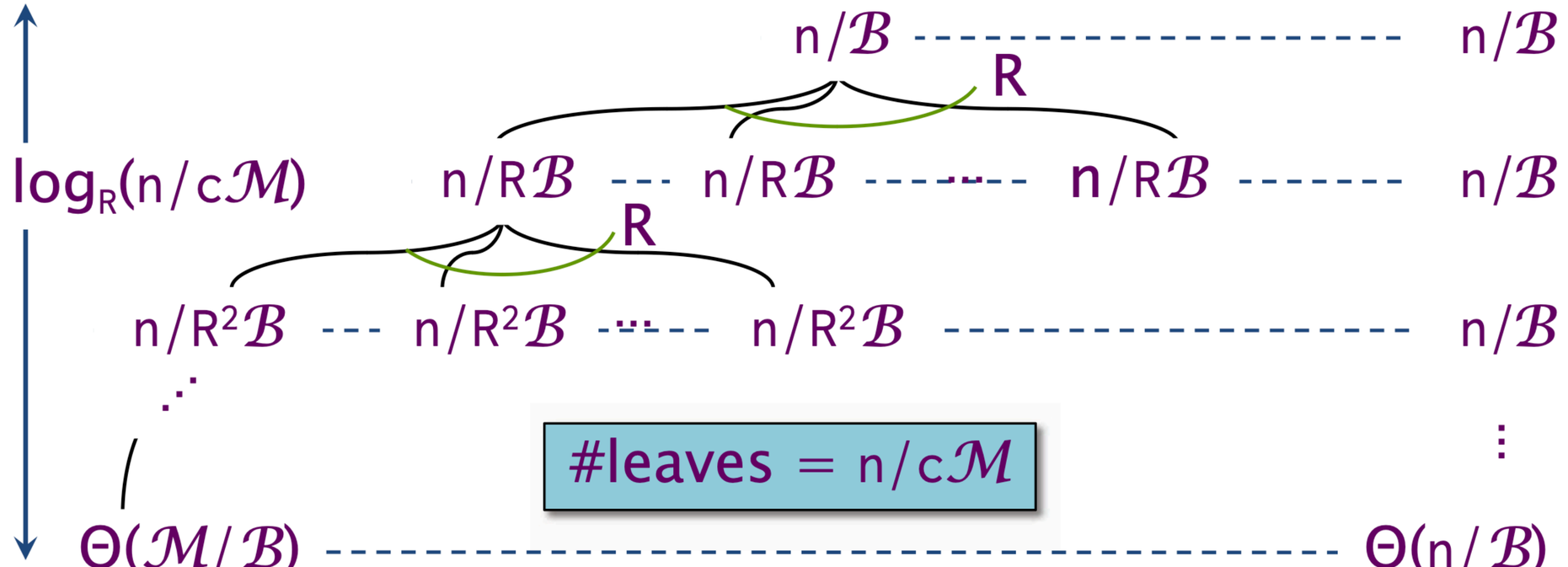
Consider the R-way merging of contiguous arrays of total size n. If $R < c\mathcal{M}/\mathcal{B}$, the entire tournament plus 1 block from each array can fit in cache.
$\Rightarrow Q(n) \leq \Theta(n/\mathcal{B})$ for merging.

R-way merge sort

$$Q(n) \leq \begin{cases} \Theta(n/\mathcal{B}) & \text{if } n < c\mathcal{M}; \\ R \cdot Q(n/R) + \Theta(n/\mathcal{B}) & \text{otherwise.} \end{cases}$$

Cache Analysis

$$Q(n) \le \begin{cases} \Theta(n/\mathcal{B}) & \text{if } n < c\mathcal{M}; \\ R \cdot Q(n/R) + \Theta(n/\mathcal{B}) & \text{otherwise.} \end{cases}$$



$$Q(n) = \Theta((n/\mathcal{B}) \log_R(n/\mathcal{M}))$$

Tuning the Parameter

We have

$$Q(n) = \Theta((n/\mathcal{B}) \log_R(n/\mathcal{M})) ,$$

which decreases as $R \leq c\mathcal{M}/\mathcal{B}$ increases.
Choosing R as big as possible yields

$$R = \Theta(\mathcal{M}/\mathcal{B}) .$$

By the tall-cache assumption and the fact that $\log_{\mathcal{M}}(n/\mathcal{M}) = \Theta((\lg n)/\lg \mathcal{M})$, we have

$$\begin{aligned} Q(n) &= \Theta((n/\mathcal{B}) \log_{\mathcal{M}/\mathcal{B}}(n/\mathcal{M})) \\ &= \Theta((n/\mathcal{B}) \log_{\mathcal{M}}(n/\mathcal{M})) \\ &= \Theta((n \lg n)/\mathcal{B} \lg \mathcal{M}) . \end{aligned}$$

Hence, we have $W(n)/Q(n) \approx \Theta(\mathcal{B} \lg \mathcal{M})$.

Multiway vs Binary Merge Sort

We have

$$Q_{\text{multiway}}(n) = \Theta((n \lg n)/\mathcal{B} \lg \mathcal{M})$$

versus

$$\begin{aligned} Q_{\text{binary}}(n) &= \Theta((n/\mathcal{B}) \lg(n/\mathcal{M})) \\ &= \Theta((n \lg n)/\mathcal{B}) \ , \end{aligned}$$

as long as $n \gg \mathcal{M}$, because then $\lg(n/\mathcal{M}) \approx \lg n$. Thus, multiway merge sort saves a factor of $\Theta(\lg \mathcal{M})$ in cache misses.

Example (ignoring constants)

- L1-cache: $\mathcal{M} = 2^{15} \Rightarrow 15\times$ savings.
- L2-cache: $\mathcal{M} = 2^{18} \Rightarrow 18\times$ savings.
- L3-cache: $\mathcal{M} = 2^{23} \Rightarrow 23\times$ savings.

Optimal Cache-Oblivious Sorting

Funnelsort [FLPR99]

1. Recursively sort $n^{1/3}$ groups of $n^{2/3}$ items.
2. Merge the sorted groups with an $n^{1/3}$–funnel.

A k–funnel merges k^3 items in k sorted lists, incurring at most

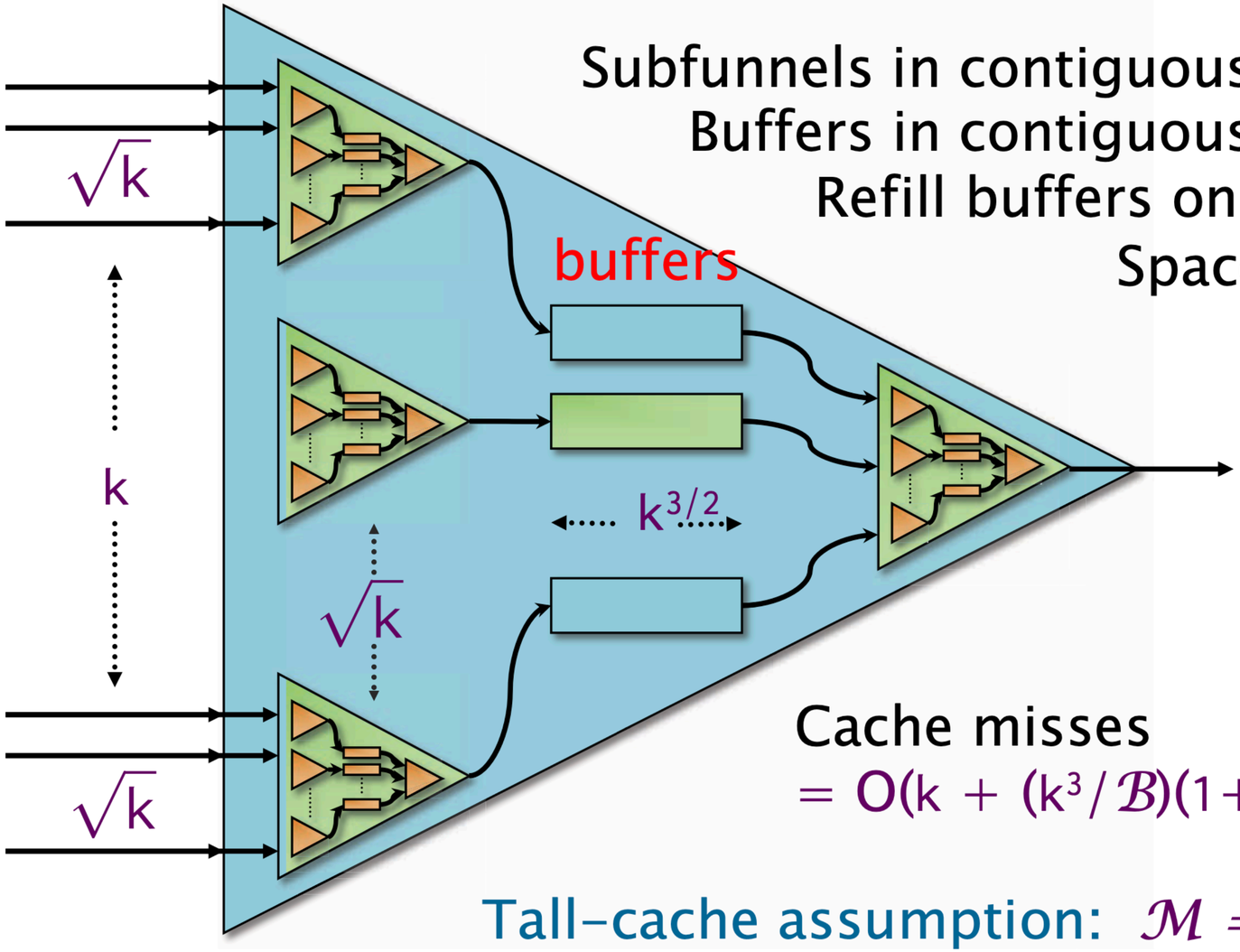
$$\Theta(k + (k^3/\mathcal{B})(1 + \log_{\mathcal{M}} k))$$

cache misses. Thus, funnelsort incurs

$$\begin{aligned} Q(n) &\leq n^{1/3} Q(n^{2/3}) + \Theta(n^{1/3} + (n/b)(1 + \log_{\mathcal{M}} n)) \\ &= \Theta(1 + (n/\mathcal{B})(1 + \log_{\mathcal{M}} n)) \,, \end{aligned}$$

cache misses, which is asymptotically optimal [AV88].

Construction of a k-funnel



Subfunnels in contiguous storage.
Buffers in contiguous storage.
Refill buffers on demand.
Space = $O(k^2)$.

k^3 items

Cache misses
$= O(k + (k^3/\mathcal{B})(1+\log_{\mathcal{M}} k))$.

Tall-cache assumption: $\mathcal{M} = \Omega(\mathcal{B}^2)$.

Other Cache-Oblivious Algorithms

Matrix Transposition/Addition $\Theta(1+mn/\mathcal{B})$

Straightforward recursive algorithm.

Strassen's Algorithm $\Theta(n + n^2/\mathcal{B} + n^{\lg 7}/\mathcal{B}\mathcal{M}^{(\lg 7)/2 - 1})$

Straightforward recursive algorithm.

Fast Fourier Transform $\Theta(1 + (n/\mathcal{B})(1 + \log_{\mathcal{M}} n))$

Variant of Cooley–Tukey [CT65] using cache-oblivious matrix transpose.

LUP-Decomposition $\Theta(1 + n^2/\mathcal{B} + n^3/\mathcal{B}\mathcal{M}^{1/2})$

Recursive algorithm due to Sivan Toledo [T97].

Cache-Oblivious Data Structures

Ordered-File Maintenance $O(1 + (\lg^2 n)/\mathcal{B})$

INSERT/DELETE or delete anywhere in file while maintaining $O(1)$-sized gaps. Amortized bound [BDFC00], later improved in [BCDFC02].

B-Trees

INSERT/DELETE:	$O(1+\log_{\mathcal{B}+1} n+(\lg^2 n)/\mathcal{B}$
SEARCH:	$O(1+\log_{\mathcal{B}+1} n)$
TRAVERSE:	$O(1+k/\mathcal{B})$

Solution [BDFC00] with later simplifications [BDIW02], [BFJ02].

Priority Queues $O(1+(1/\mathcal{B})\log_{\mathcal{M}/\mathcal{B}}(n/\mathcal{B}))$

Funnel-based solution [BF02]. General scheme based on buffer trees [ABDHMM02] supports INSERT/DELETE.