6.172
Performance
Engineering
of Software
Systems

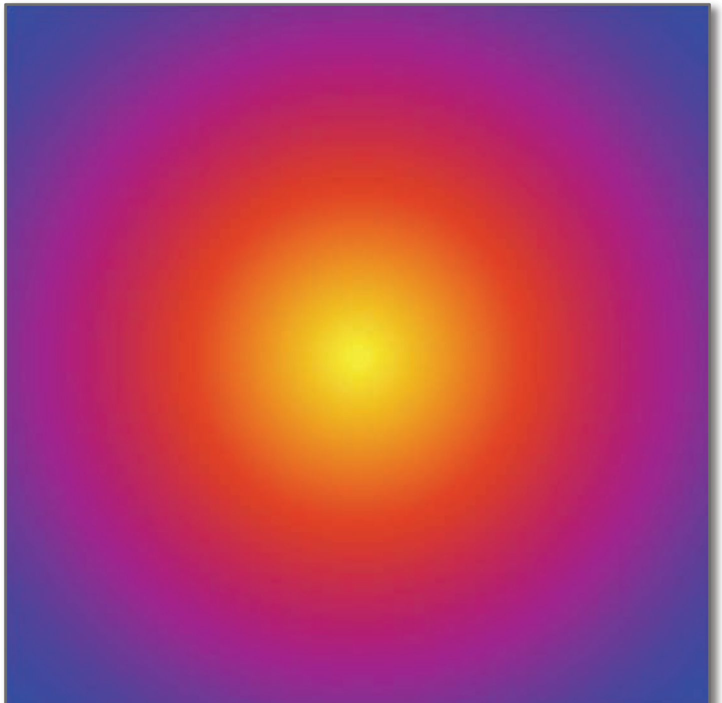
LECTURE 15

Cache-Oblivious Algorithms

Julian Shun

Simulation of Heat Diffusion

Heat Diffusion



2D heat equation

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

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

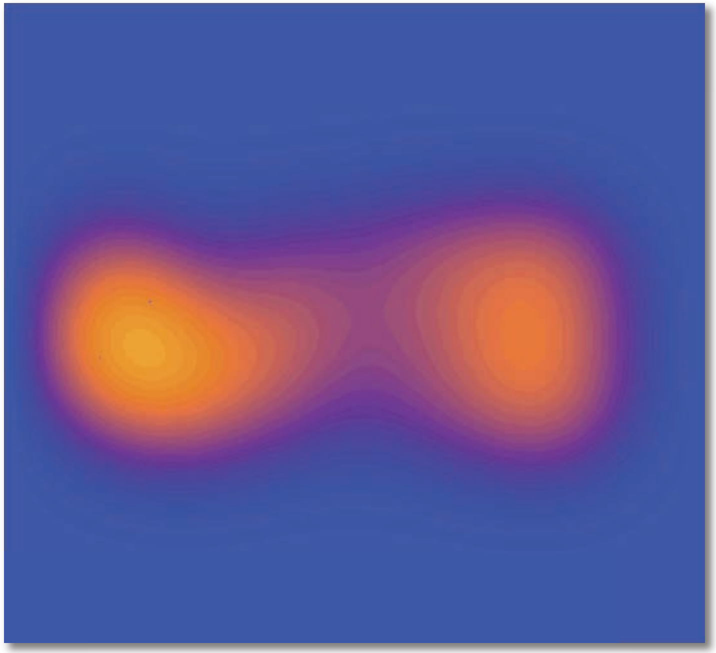
α is the *thermal diffusivity*.

Acknowledgment

Some of the slides in this presentation were inspired by originals due to Matteo Frigo.

2D Heat-Diffusion Simulation

Before



After

1D Heat Equation

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

Finite-Difference Approximation

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

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

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

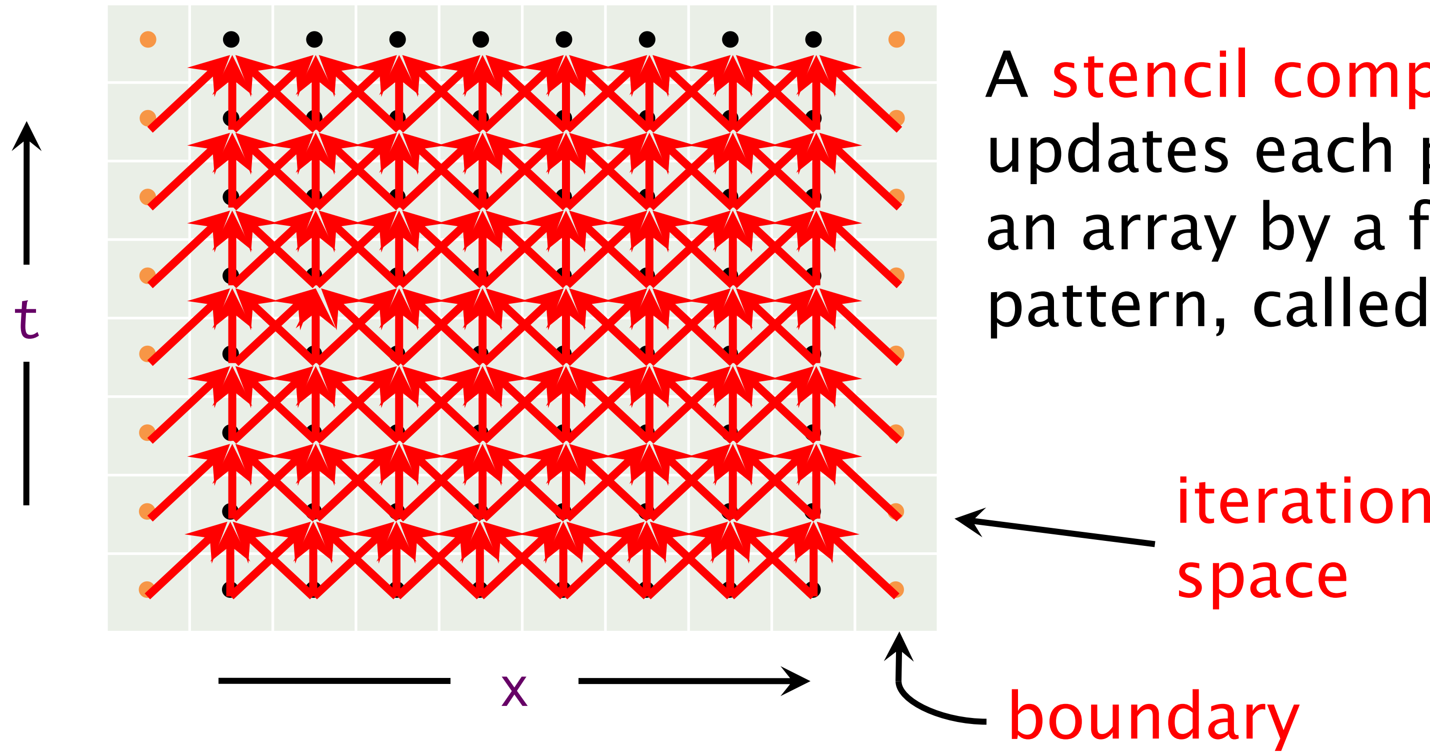
$$\frac{\partial^2}{\partial x^2} u\ \ t\ \ x \quad \approx \quad \frac{\frac{\partial}{\partial x} u\ \ t\ \ x\ +\Delta x/2\ ) - \frac{\partial}{\partial x} u\ \ t\ \ x - \Delta x/2}{\Delta x}$$

The 1D heat equation thus reduces to

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

3-Point Stencil

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



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

Update rule

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

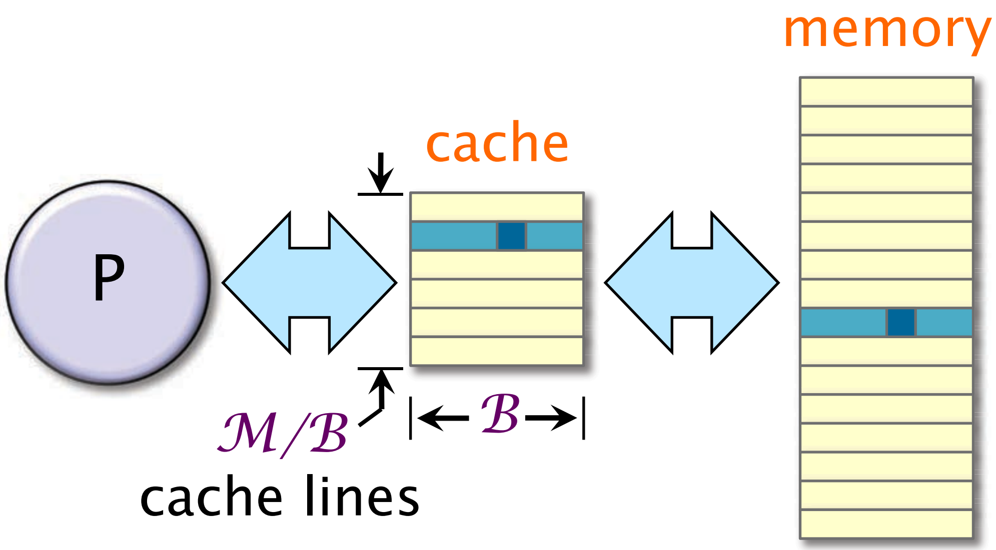
```
u[t+1][x] = u[t][x] + ALPHA
                * (u[t][x+1] - 2*u[t][x] + u[t][x-1]);
```

Cache-Oblivious Stencil Computations

Recall: Ideal-Cache Model

Parameters

- Two-level hierarchy.
- Cache size of $\mathcal{M}$ bytes.
- Cache-line length (block size) of $\mathcal{B}$ bytes.
- Fully associative.
- Optimal omniscient replacement, or LRU.



Performance Measures

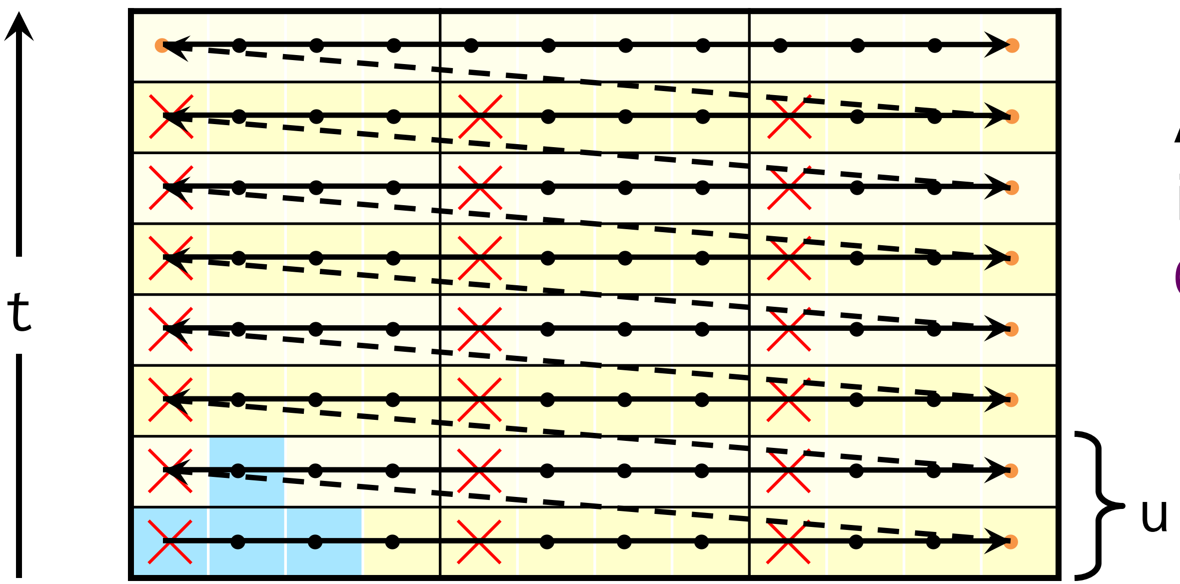
- work W (ordinary running time)
- cache misses Q

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] );
```

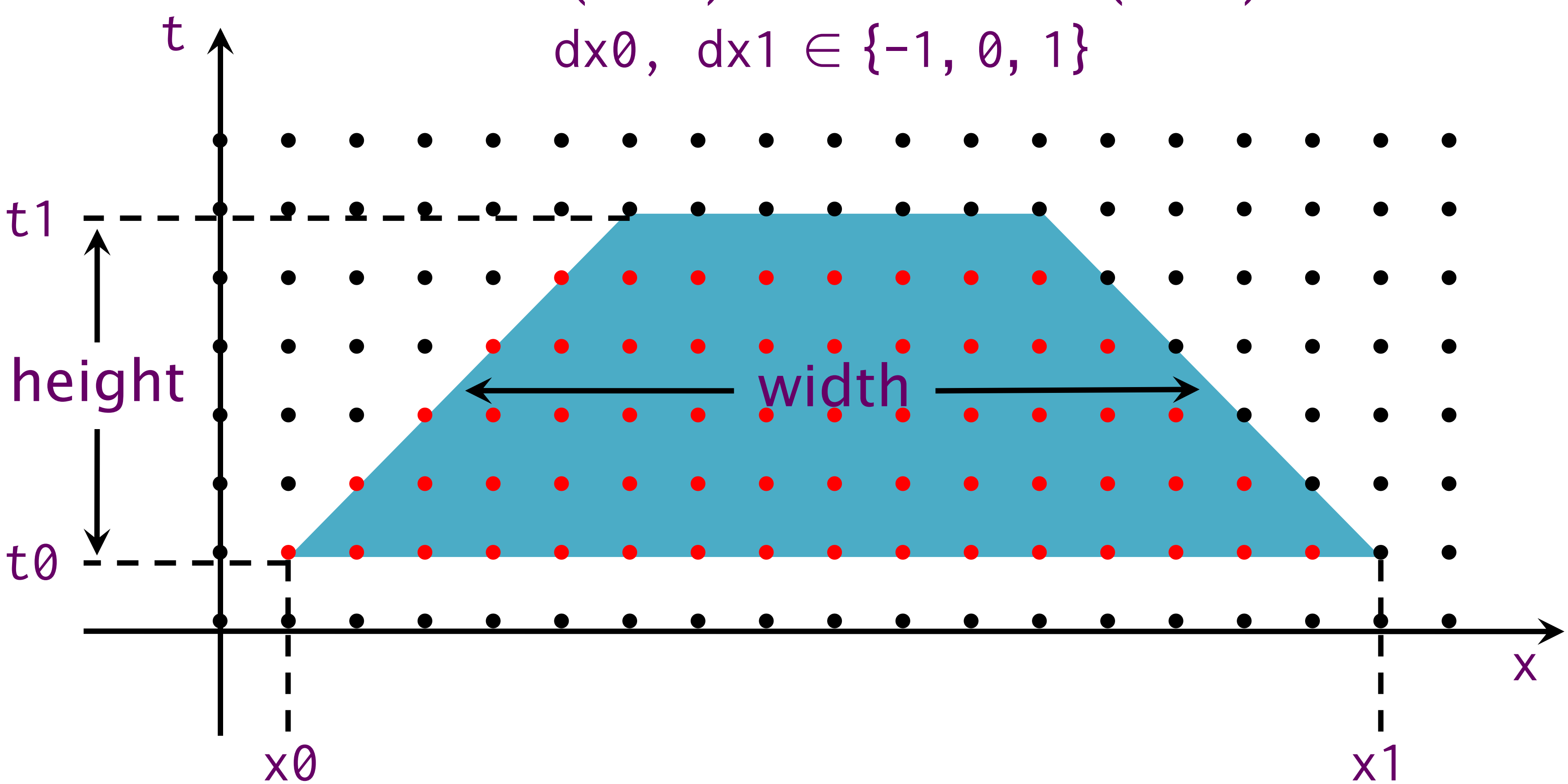


Assuming LRU, if $N > \mathcal{M}$, then $Q = \Theta(NT/\mathcal{B})$.

Cache-Oblivious 3-Point Stencil

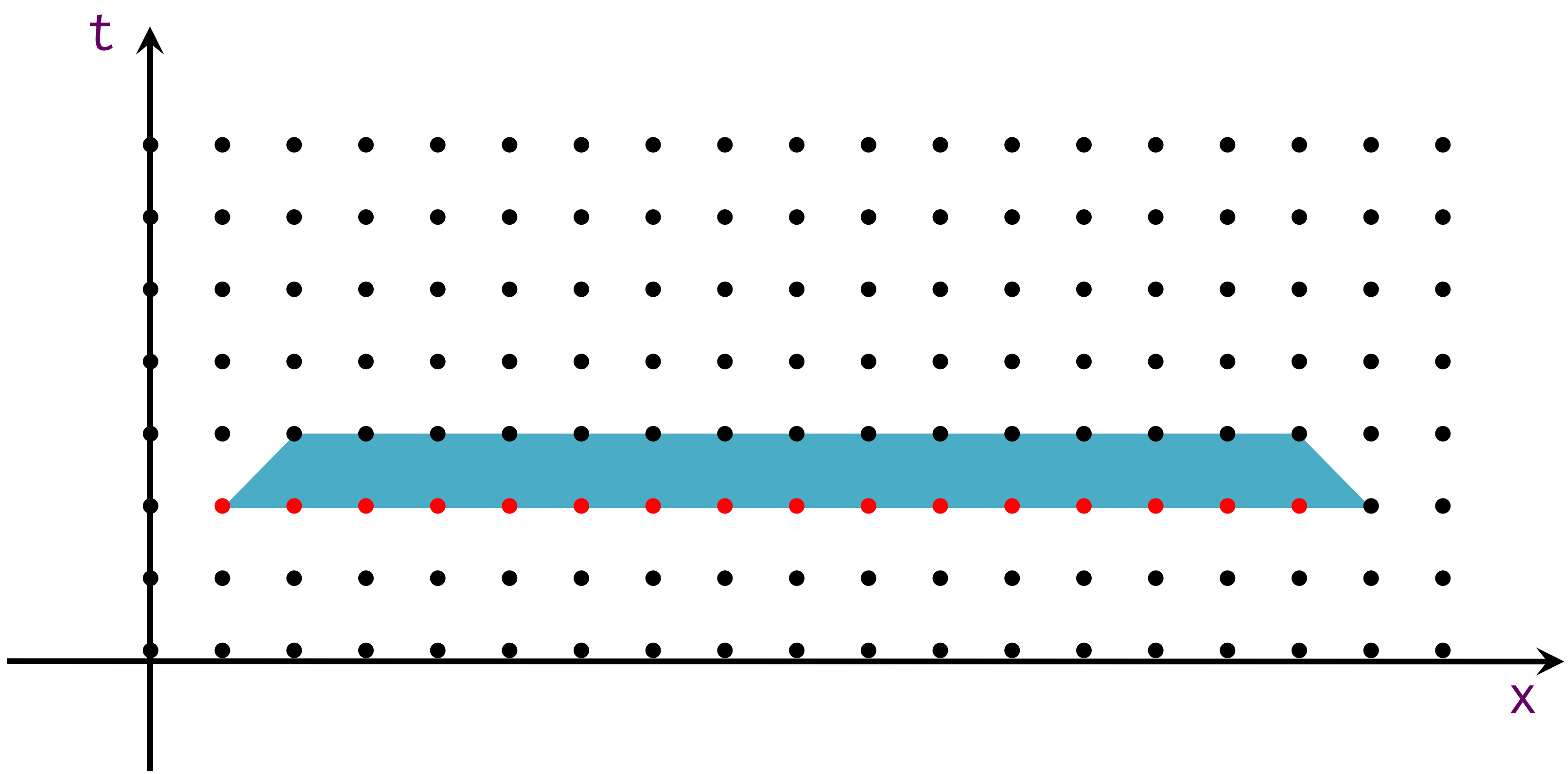
Recursively traverse trapezoidal regions of space-time points (t,x) such that

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



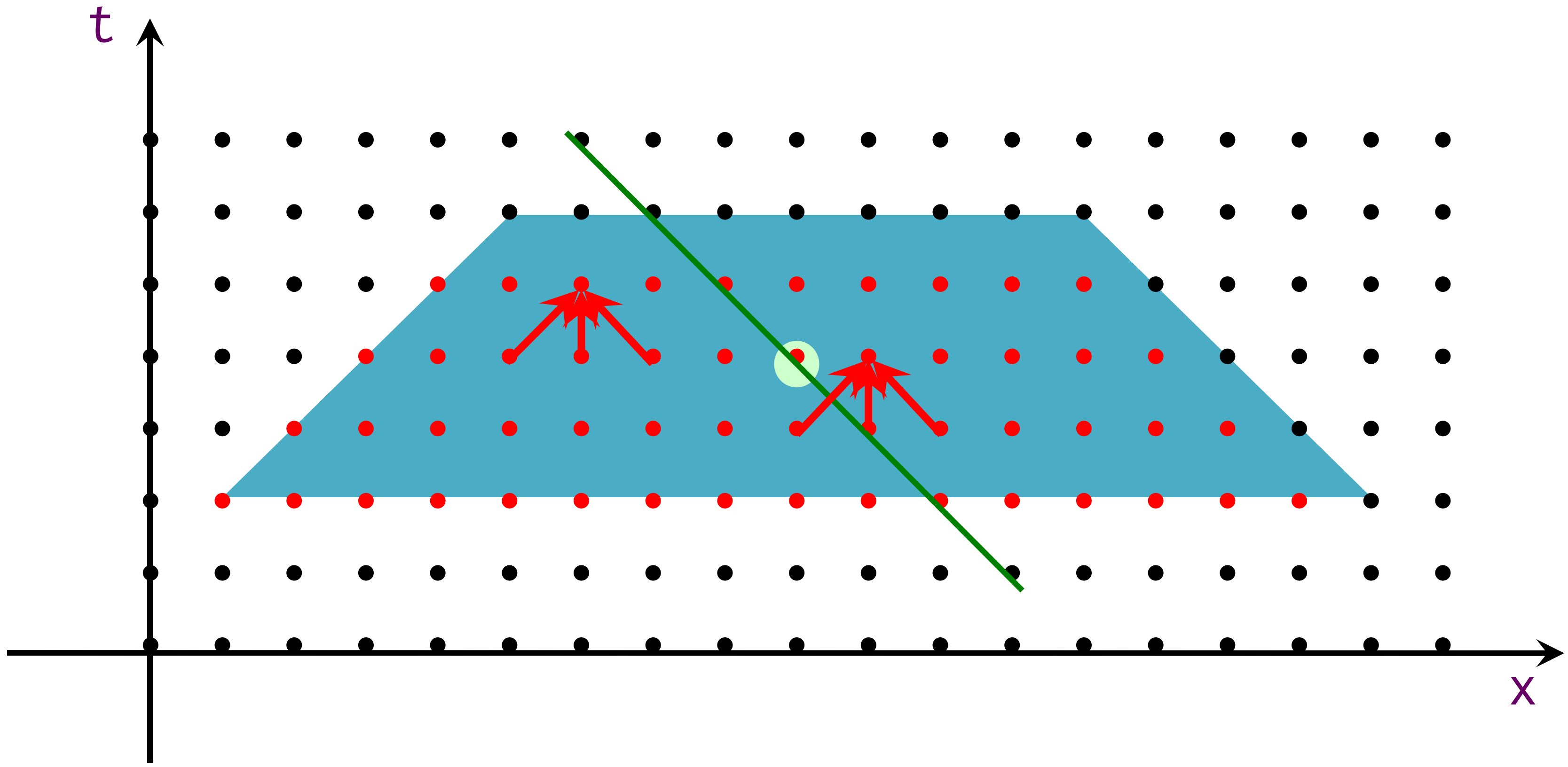
Base Case

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



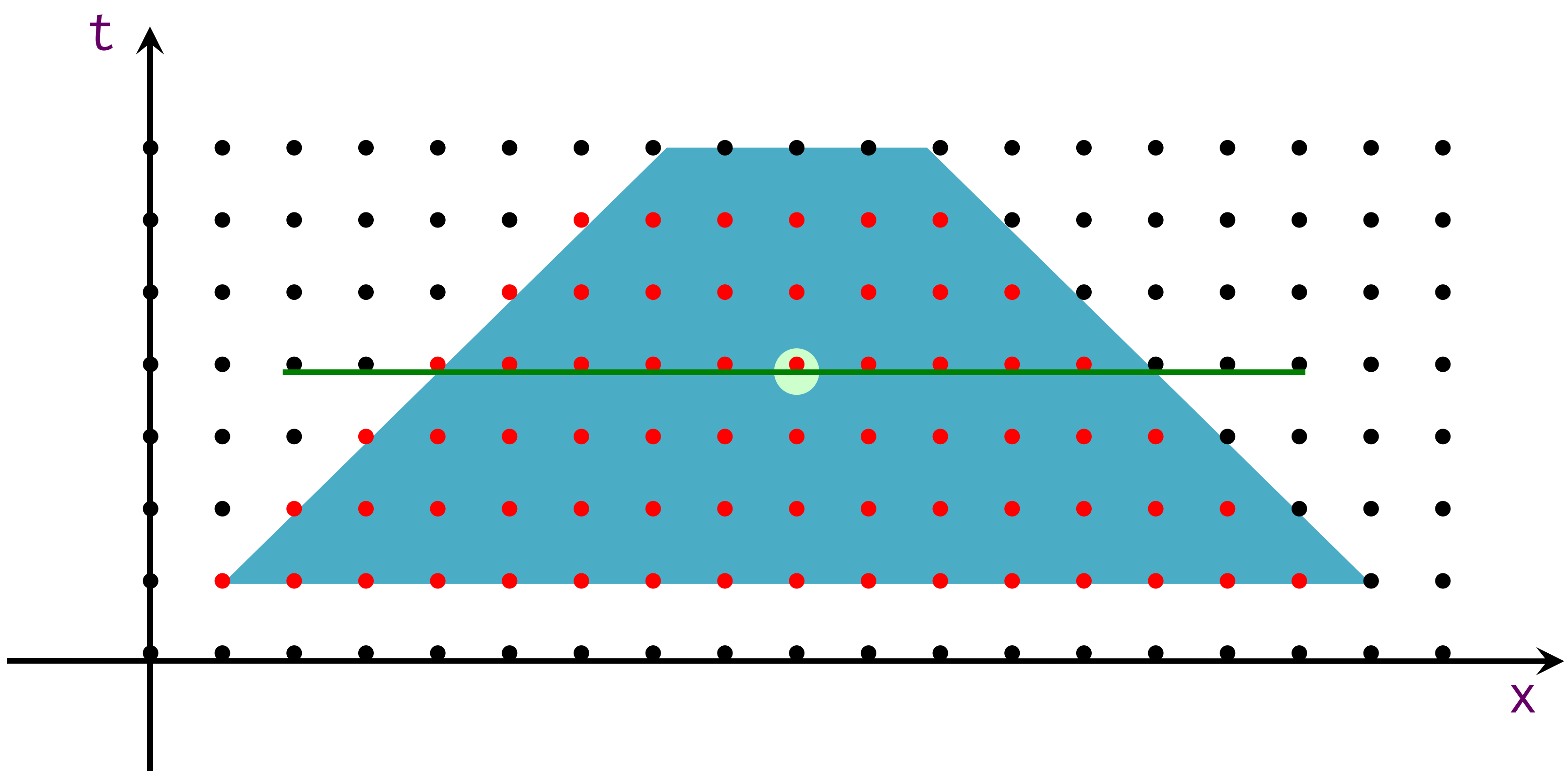
Space Cut

If width $\geq$ 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.



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.

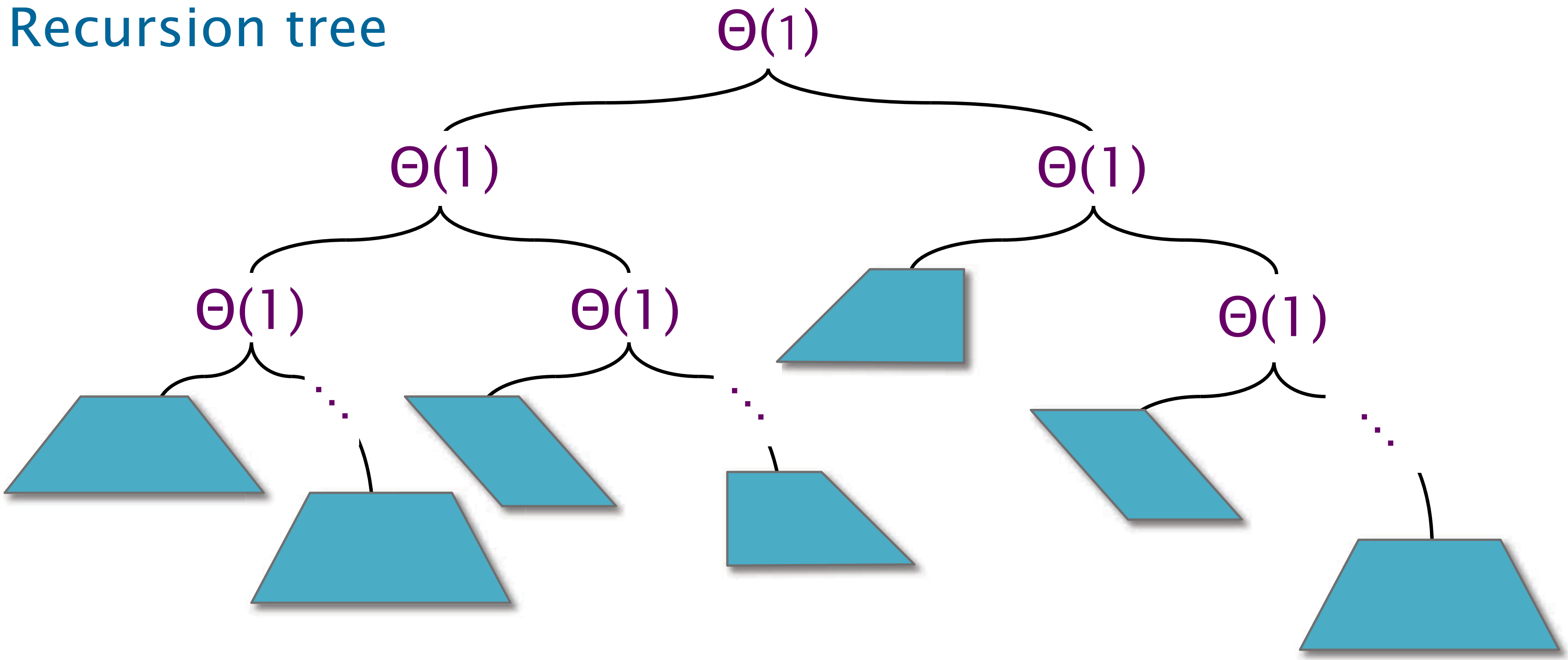


C Implementation

```
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);
    }
  }
}
```

Cache Analysis

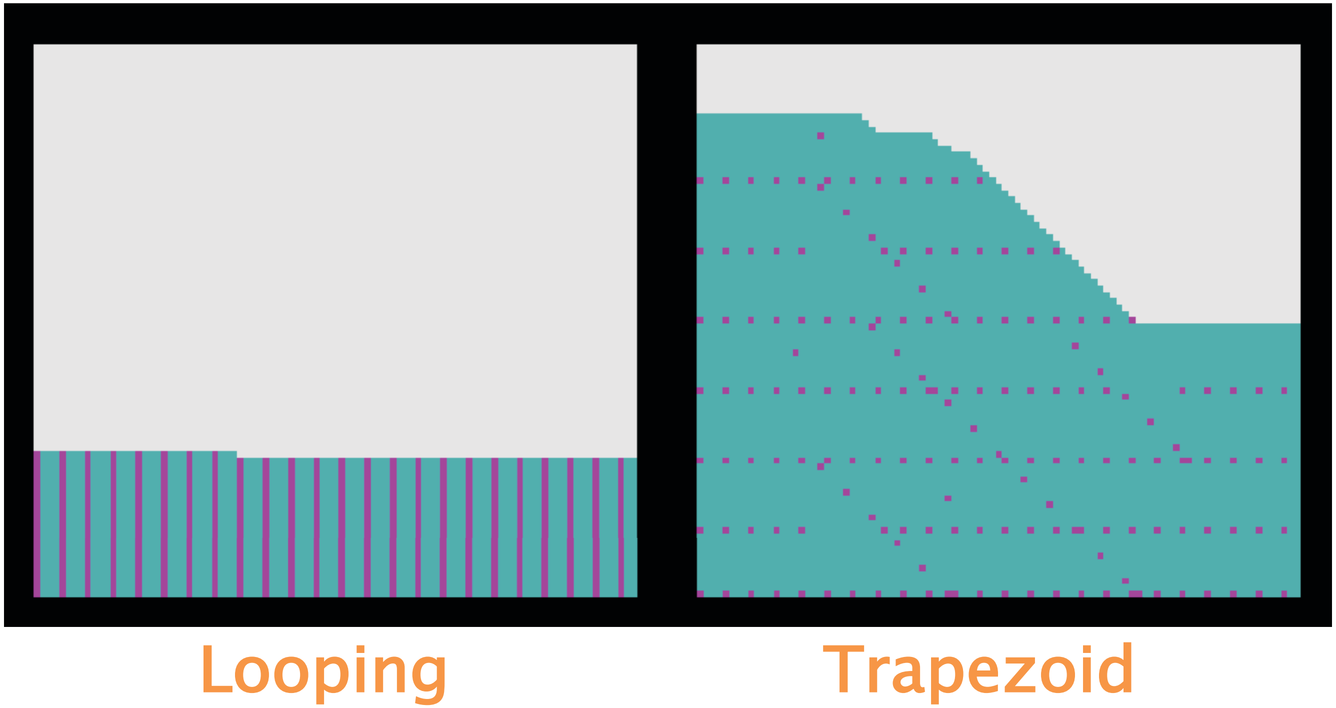
Recursion tree



- Each leaf represents $\Theta(hw)$ points, where $h = \Theta(w)$.
- Each leaf incurs $\Theta(w/\mathcal{B})$ misses, where $w = \Theta(\mathcal{M})$.
- $\Theta(NT/hw)$ leaves.
- #internal nodes = #leaves − 1 do not contribute substantially to Q.
- $Q = \Theta(NT/hw) \cdot \Theta(w/\mathcal{B}) = \Theta(NT/\mathcal{M}^2) \cdot \Theta(\mathcal{M}/\mathcal{B}) = \Theta(NT/\mathcal{M}\mathcal{B})$.
- For d dimensions, $Q = \Theta(NT/\mathcal{M}^{1/d}\mathcal{B})$

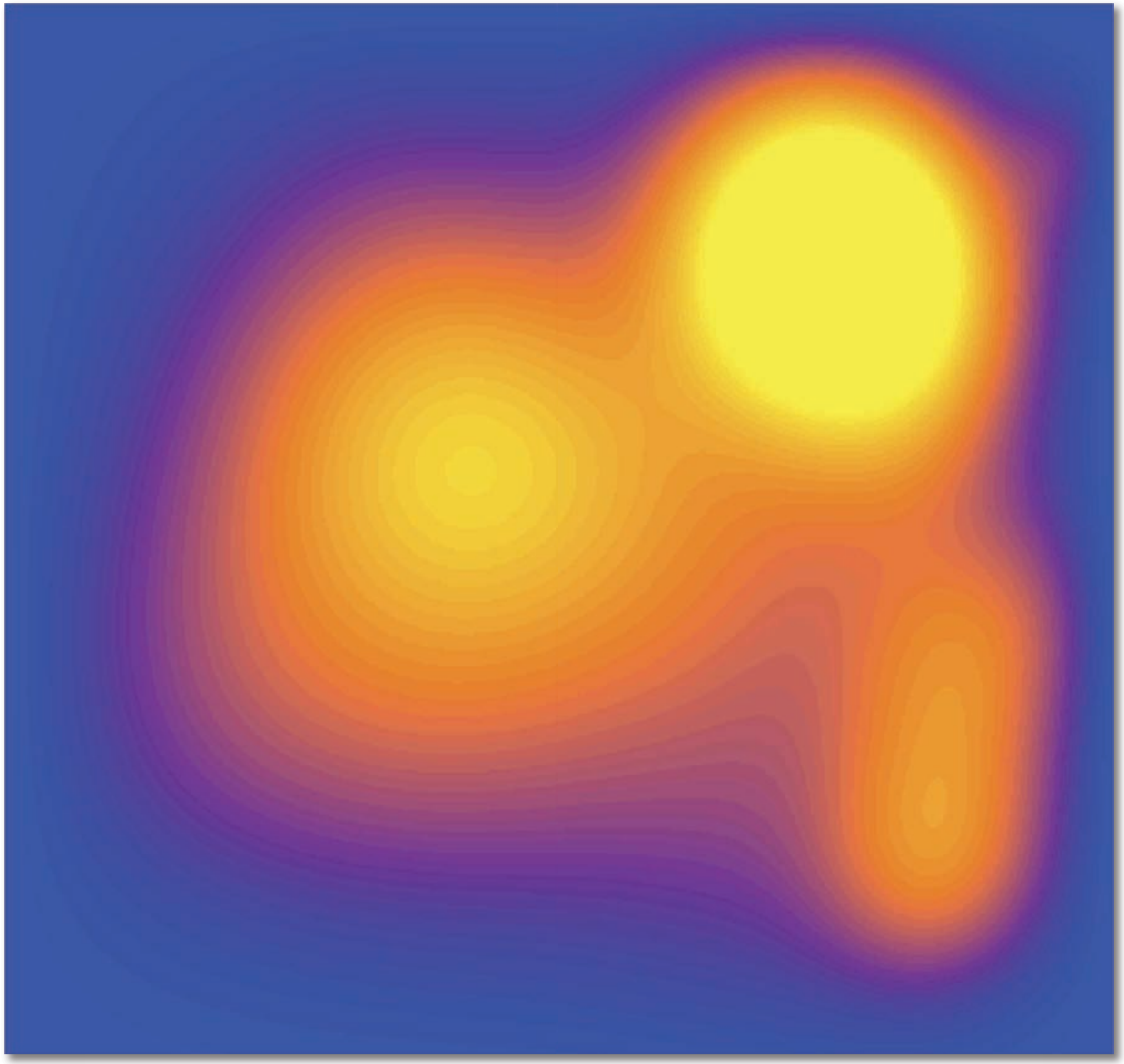
Simulation: 3-Point Stencil

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



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

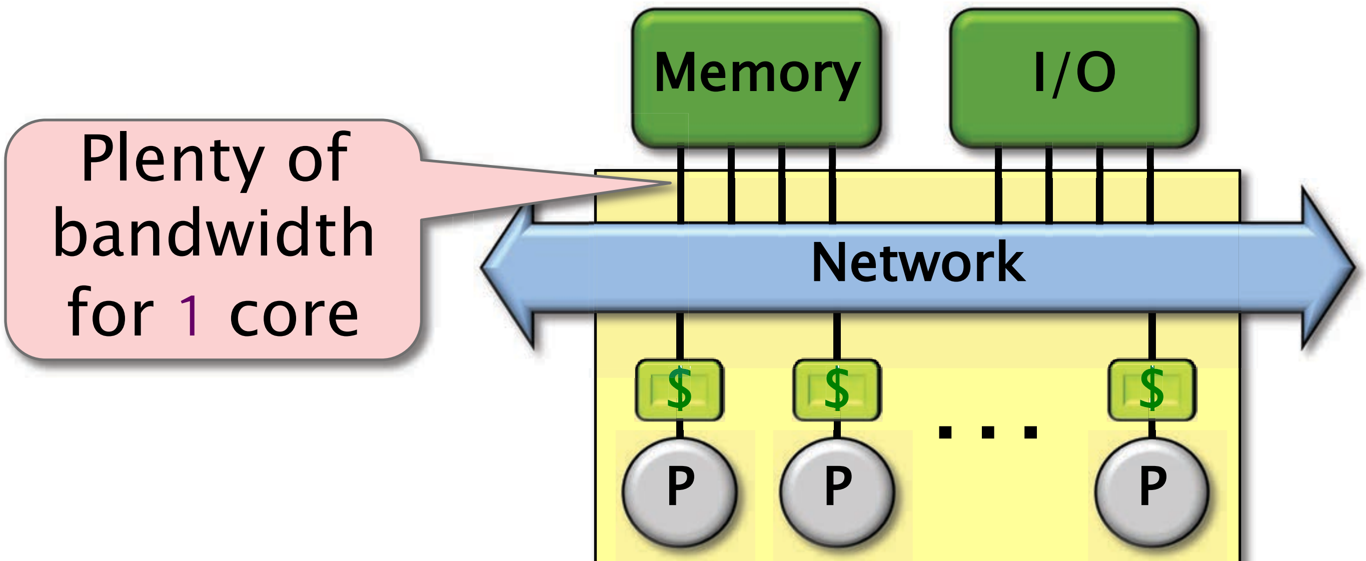
Looping v. Trapezoid on Heat



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

Cilk and Caching

Theorem. Let Q_P be the number of cache misses in a deterministic Cilk computation when run on P processors, each with a private cache, and let S_P be the number of successful steals during the computation. In the ideal-cache model, we have

$$Q_P = Q_1 + O(S_P \mathcal{M}/\mathcal{B}) ,$$

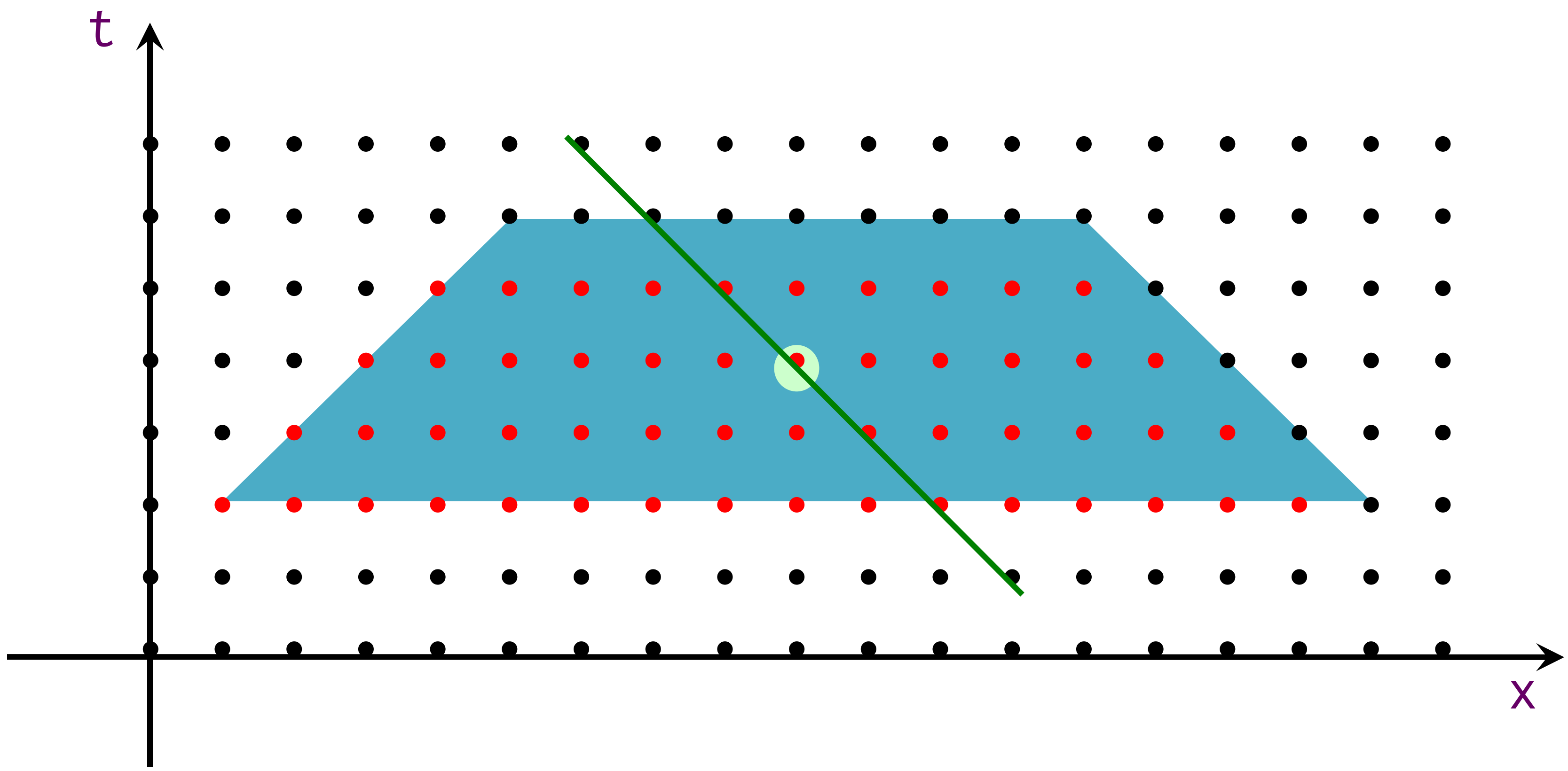
where $\mathcal{M}$ is the cache size and $\mathcal{B}$ is the size of a cache block.

Proof. After a worker steals a continuation, its cache is completely cold in the worst case. But after $\mathcal{M}/\mathcal{B}$ (cold) cache misses, its cache is identical to that in the serial execution. The same is true when a worker resumes a stolen subcomputation after a `cilk_sync`. The number of times these two situations can occur is at most $2S_P$. ■

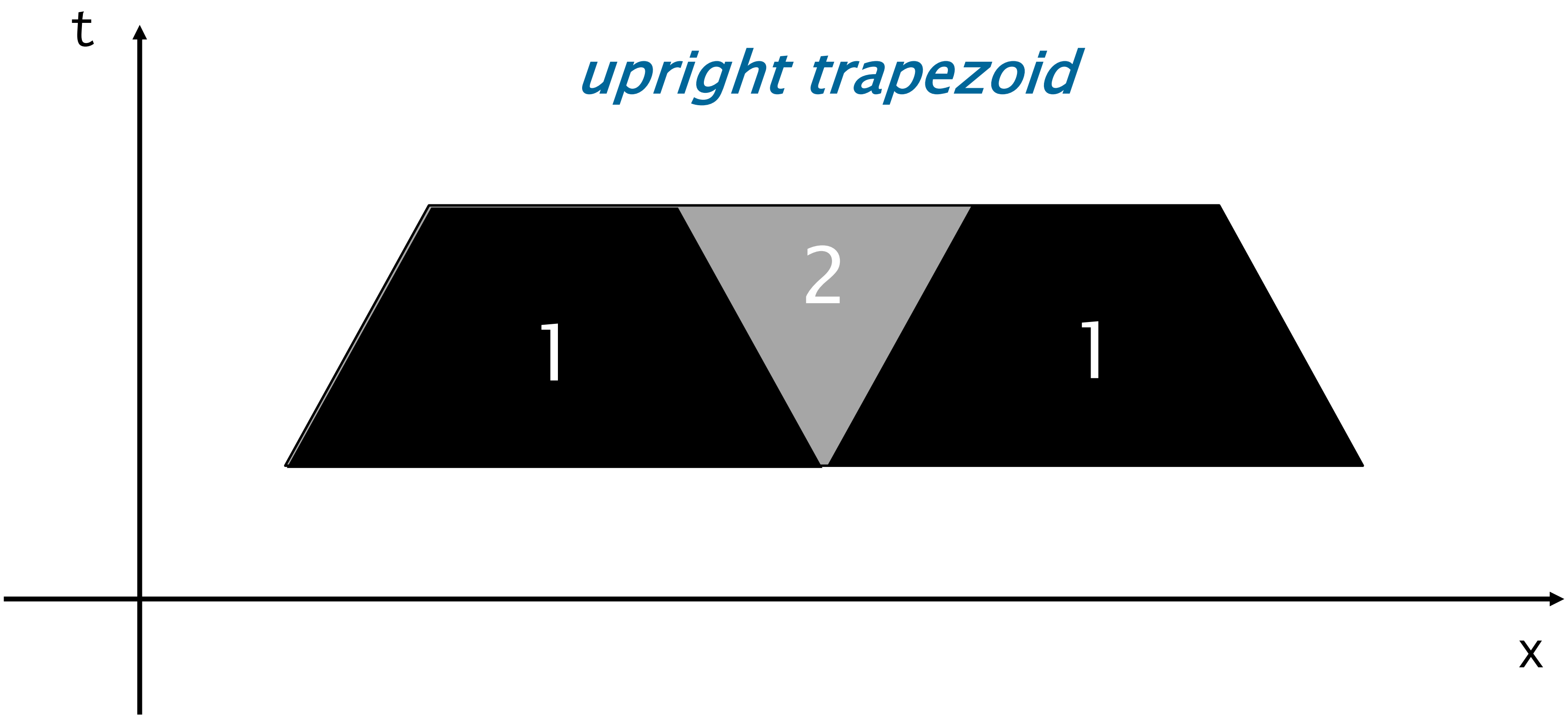
MORAL: Minimizing cache misses in the serial elision essentially minimizes them in parallel executions.

Does this work in parallel?

Space cut: If width $\geq$ 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.

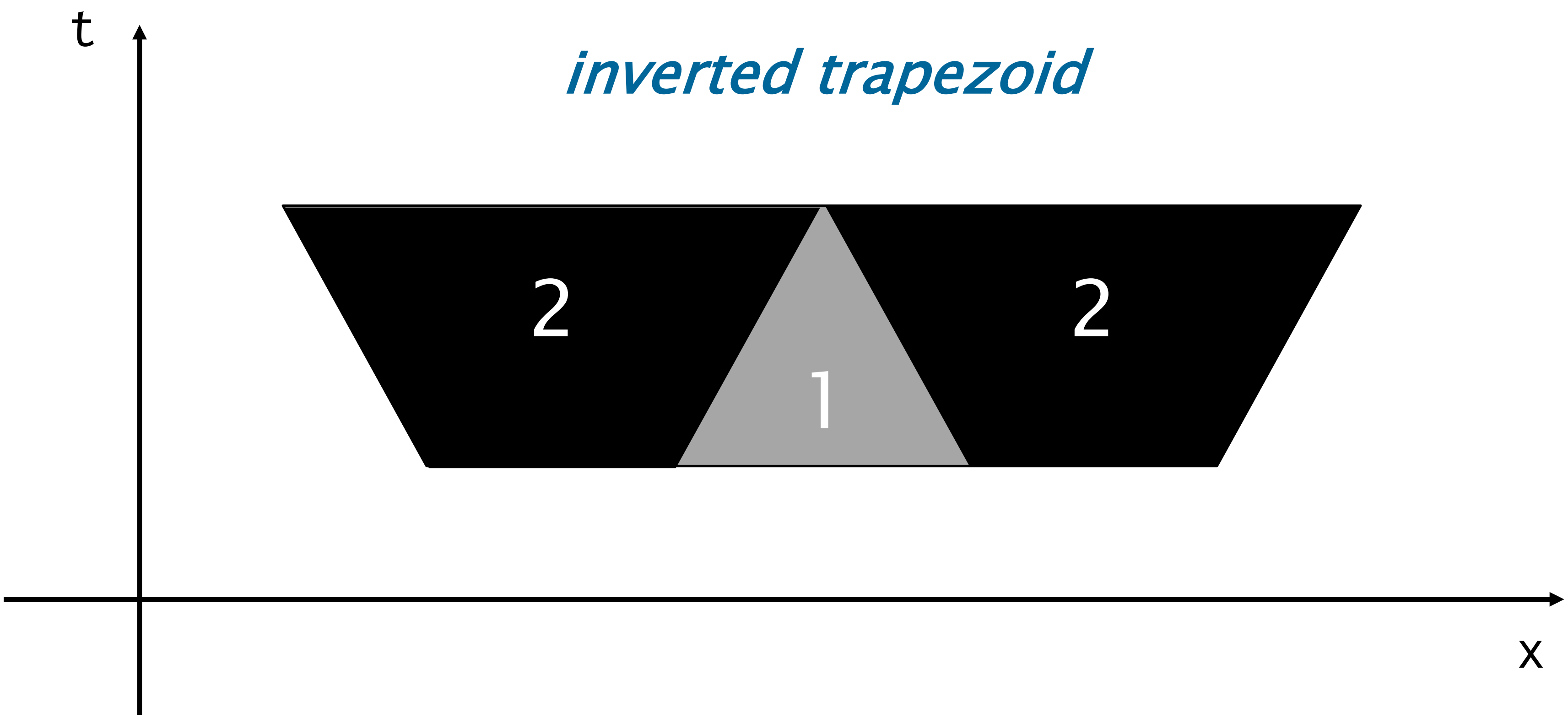


Parallel Space Cuts



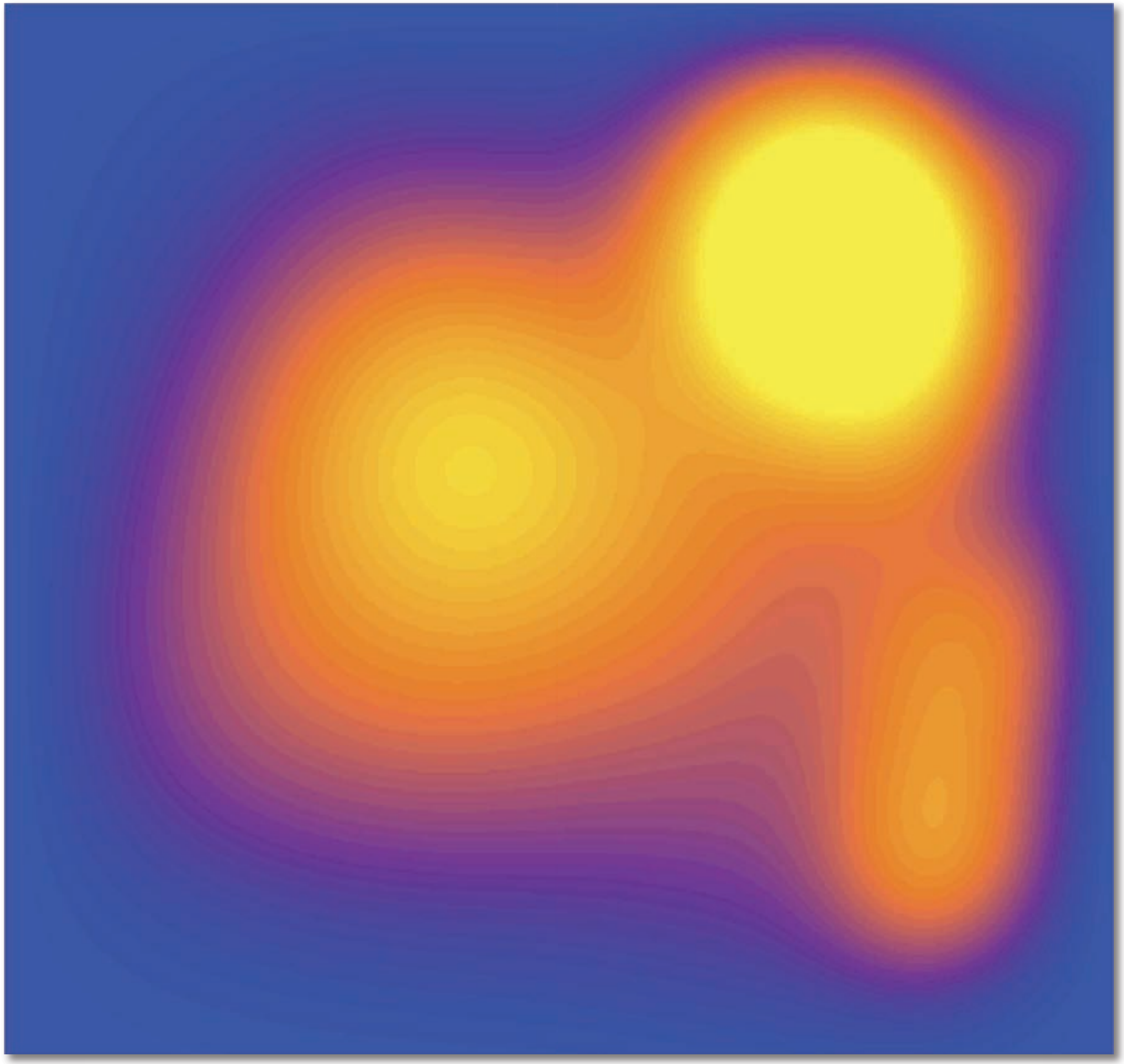
A *parallel space cut* produces two black trapezoids that can be executed in parallel and a third gray trapezoid that executes in series with the black trapezoids.

Parallel Space Cuts



A ***parallel space cut*** produces two black trapezoids that can be executed in parallel and a third gray trapezoid that executes in series with the black trapezoids.

Parallel Looping v. Parallel Trap.



Performance Comparison

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

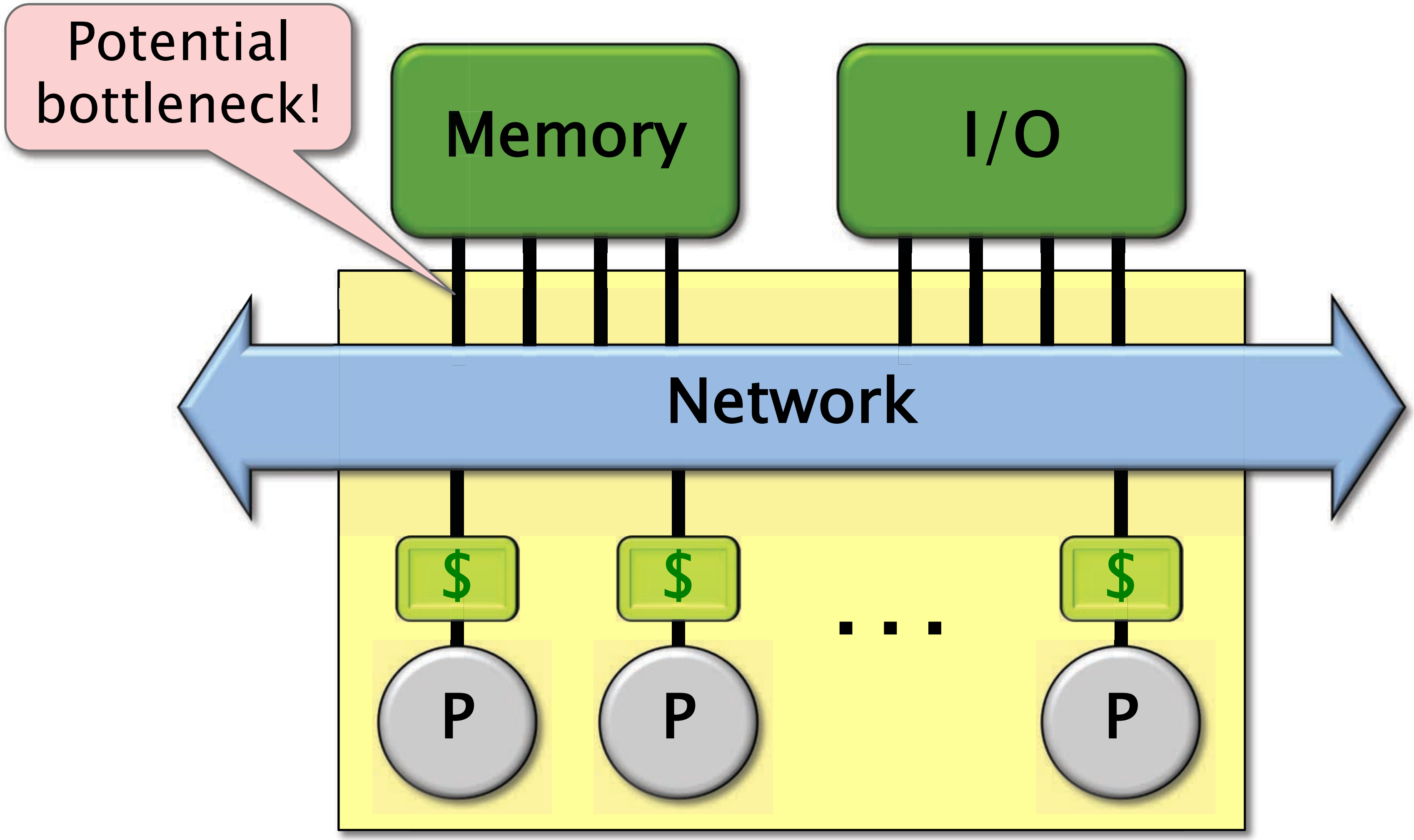
Code	Time
Serial looping	128.95s
Parallel looping	66.97s
Serial trapezoidal	66.76s
Parallel trapezoidal	16.86s

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
- ☑ Scheduling overhead
- ☑ Lack of memory bandwidth
- ☑ Contention (locking and true/false sharing)

Cilkscale can diagnose the first two problems.

Q. How can we diagnose the third?

A. Run P identical copies of the serial code in parallel — if you have enough memory.

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

Cache-Oblivious Sorting

OUTLINE

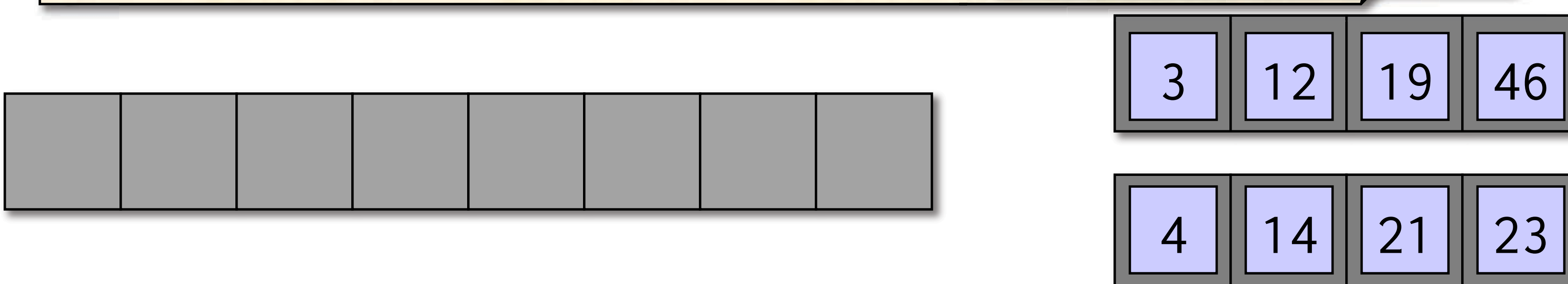
- Simulation of Heat Diffusion
- Cache-Oblivious Stencil Computations
- Caching and Parallelism
- Cache-Oblivious Sorting

Merging Two Sorted Arrays

```
void merge(int64_t *C, int64_t *A, int64_t na,
           int64_t *B, int64_t 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--;
  }
}
```

Time to merge n elements $= \Theta(n)$.

Number of cache misses $= \Theta(n/B)$.



Merge Sort

```
void merge_sort(int64_t *B, int64_t *A, int64_t n) {
  if (n==1) {
    B[0] = A[0];
  } else {
    int64_t C[n];
    cilk_spawn merge_sort(C, A, n/2);
               merge_sort(C+n/2, A+n/2, n-n/2);
    cilk_sync;
    merge(B, C, n/2, C+n/2, n-n/2);
  }
}
```

19	3	12	46	33	4	21	14

Merge Sort

```
void merge_sort(int64_t *B, int64_t *A, int64_t n) {
  if (n==1) {
    B[0] = A[0];
  } else {
    int64_t C[n];
    cilk_spawn merge_sort(C, A, n/2);
               merge_sort(C+n/2, A+n/2, n-n/2);
    cilk_sync;
    merge(B, C, n/2, C+n/2, n-n/2);
  }
}
```

19	3	12	46

33	4	21	14

Merge Sort

```
void merge_sort(int64_t *B, int64_t *A, int64_t n) {
  if (n==1) {
    B[0] = A[0];
  } else {
    int64_t C[n];
    cilk_spawn merge_sort(C, A, n/2);
               merge_sort(C+n/2, A+n/2, n-n/2);
    cilk_sync;
    merge(B, C, n/2, C+n/2, n-n/2);
  }
}
```

19 3	12 46	33 4	21 14

Merge Sort

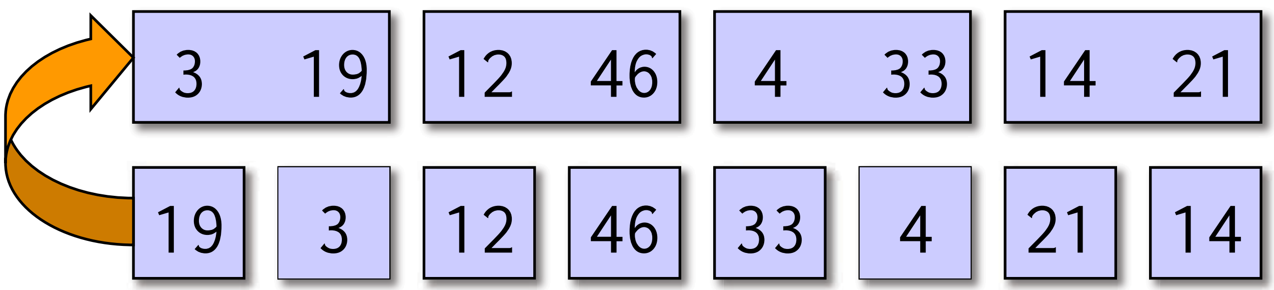
```
void merge_sort(int64_t *B, int64_t *A, int64_t n) {
  if (n==1) {
    B[0] = A[0];
  } else {
    int64_t C[n];
    cilk_spawn merge_sort(C, A, n/2);
               merge_sort(C+n/2, A+n/2, n-n/2);
    cilk_sync;
    merge(B, C, n/2, C+n/2, n-n/2);
  }
}
```

19	3	12	46	33	4	21	14

Merge Sort

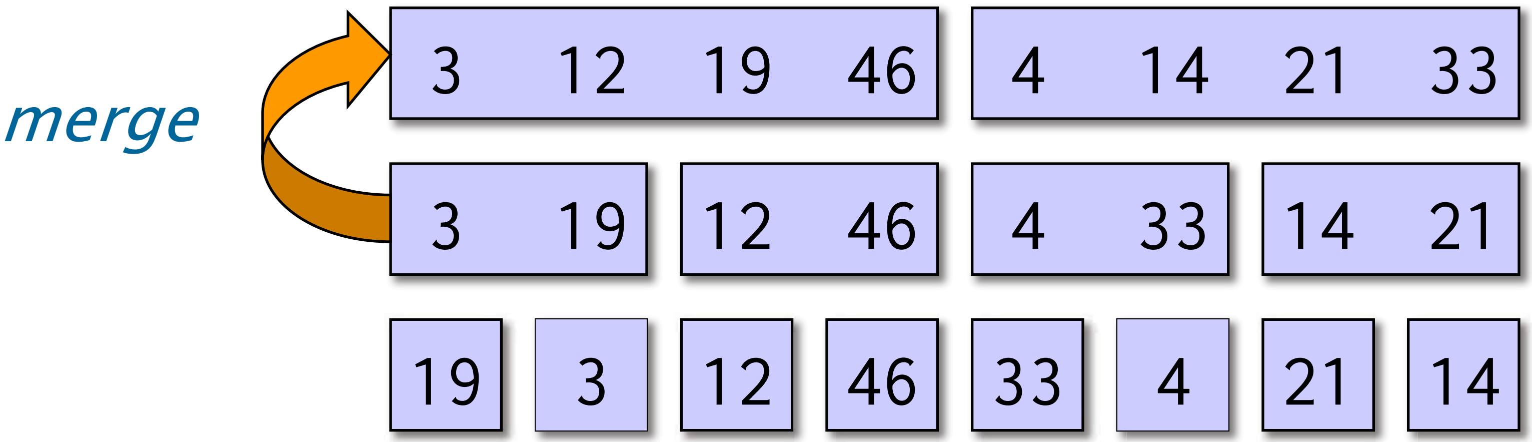
```
void merge_sort(int64_t *B, int64_t *A, int64_t n) {
  if (n==1) {
    B[0] = A[0];
  } else {
    int64_t C[n];
    cilk_spawn merge_sort(C, A, n/2);
               merge_sort(C+n/2, A+n/2, n-n/2);
    cilk_sync;
    merge(B, C, n/2, C+n/2, n-n/2);
  }
}
```

merge



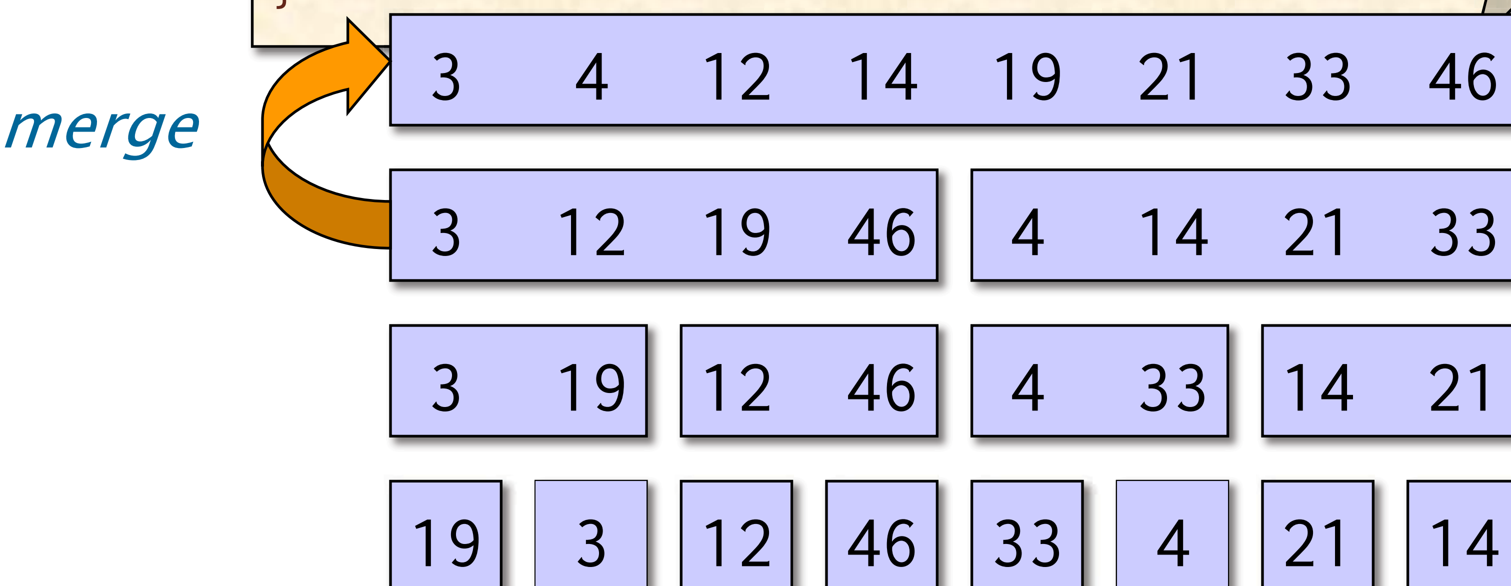
Merge Sort

```
void merge_sort(int64_t *B, int64_t *A, int64_t n) {
  if (n==1) {
    B[0] = A[0];
  } else {
    int64_t C[n];
    cilk_spawn merge_sort(C, A, n/2);
               merge_sort(C+n/2, A+n/2, n-n/2);
    cilk_sync;
    merge(B, C, n/2, C+n/2, n-n/2);
  }
}
```



Merge Sort

```
void merge_sort(int64_t *B, int64_t *A, int64_t n) {
  if (n==1) {
    B[0] = A[0];
  } else {
    int64_t C[n];
    cilk_spawn merge_sort(C, A, n/2);
               merge_sort(C+n/2, A+n/2, n-n/2);
    cilk_sync;
    merge(B, C, n/2, C+n/2, n-n/2);
  }
}
```



Work of Merge Sort

```
void merge_sort(int64_t *B, int64_t *A, int64_t n) {
  if (n==1) {
    B[0] = A[0];
  } else {
    int64_t C[n];
    merge_sort(C, A, n/2);
    merge_sort(C+n/2, A+n/2, n-n/2);
    merge(B, C, n/2, C+n/2, n-n/2);
  }
}
```

CASE 2

$n^{\log_b a} = n^{\log_2 2} = n$

$f(n) = \Theta(n^{\log_b a} \lg^0 n)$

Work:

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

Recursion Tree

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

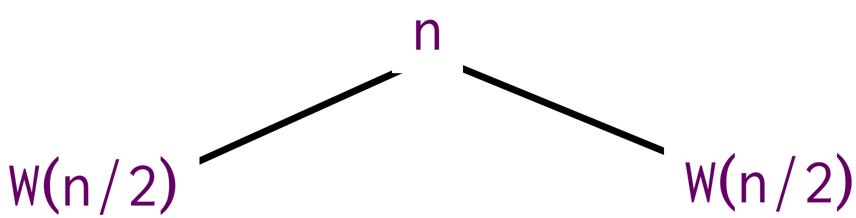
Recursion Tree

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

$$W(n)$$

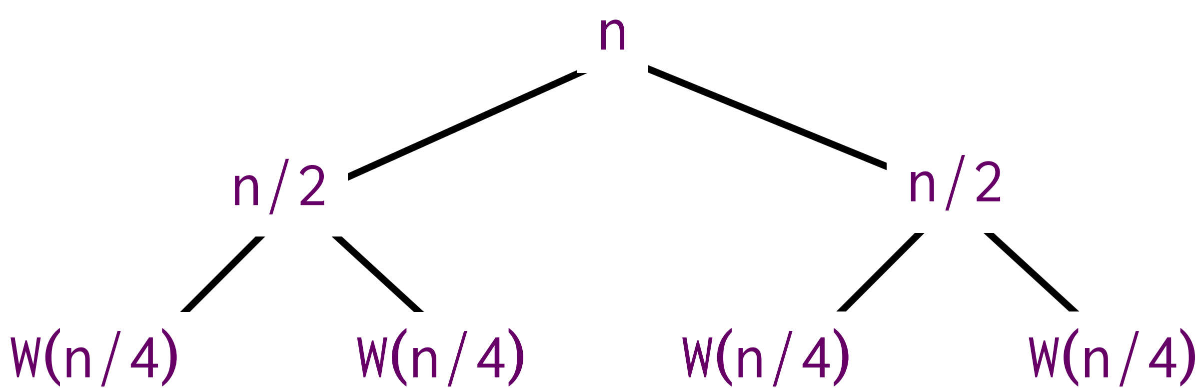
Recursion Tree

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



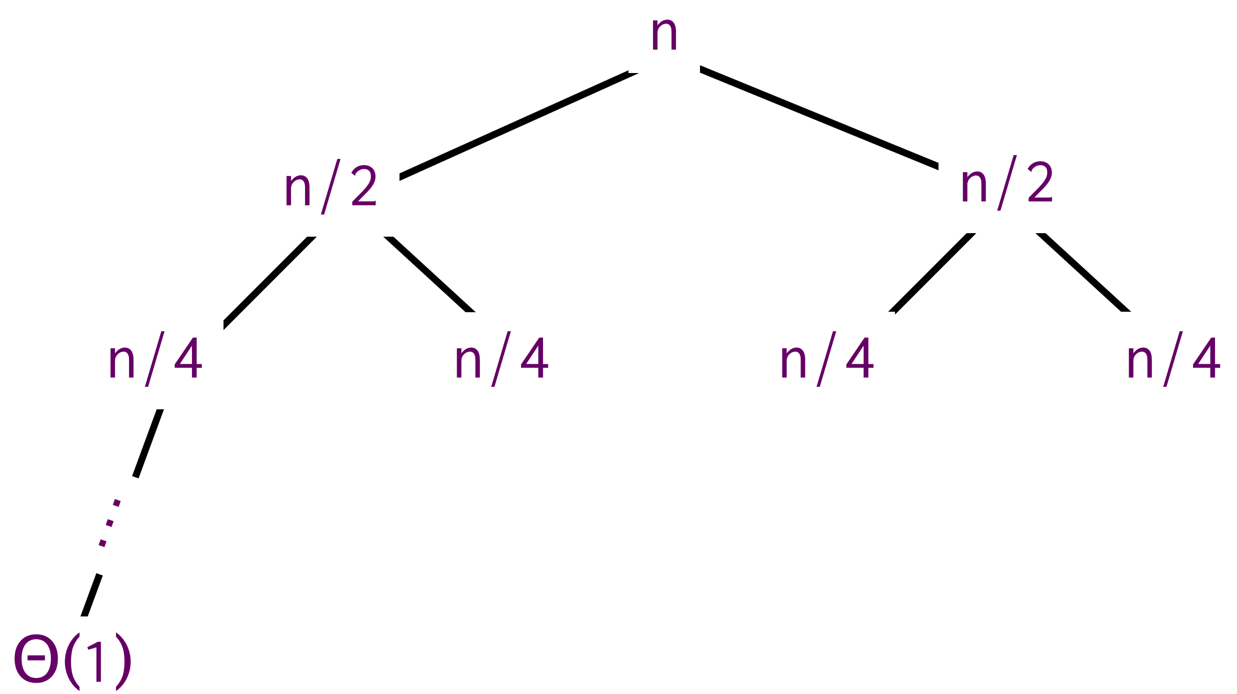
Recursion Tree

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



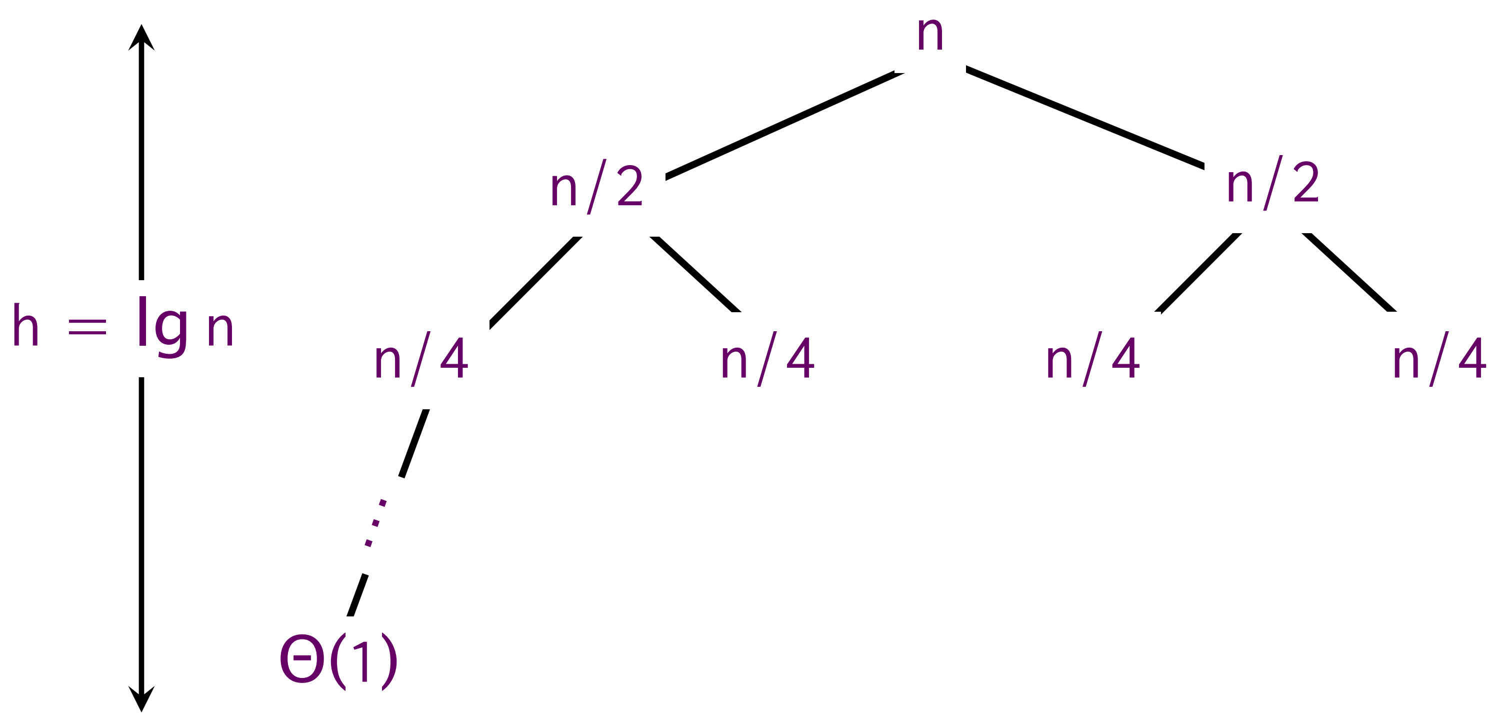
Recursion Tree

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



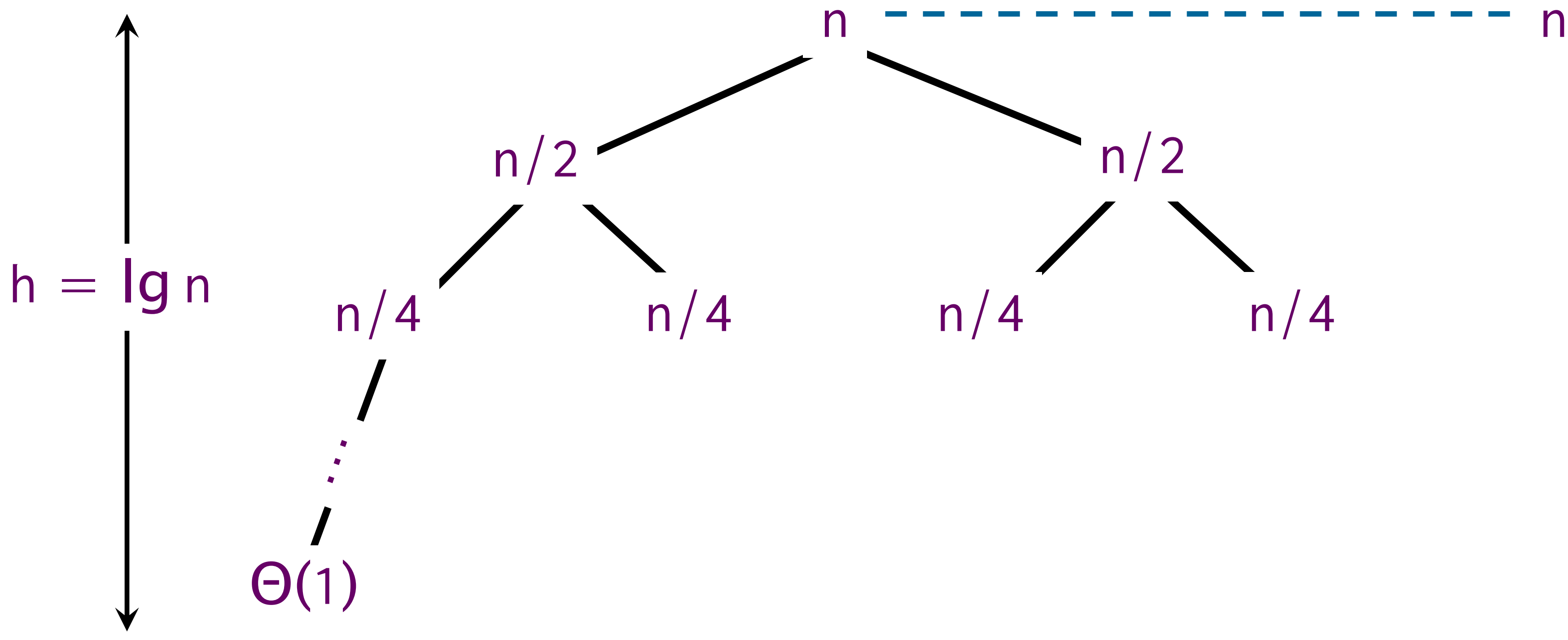
Recursion Tree

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



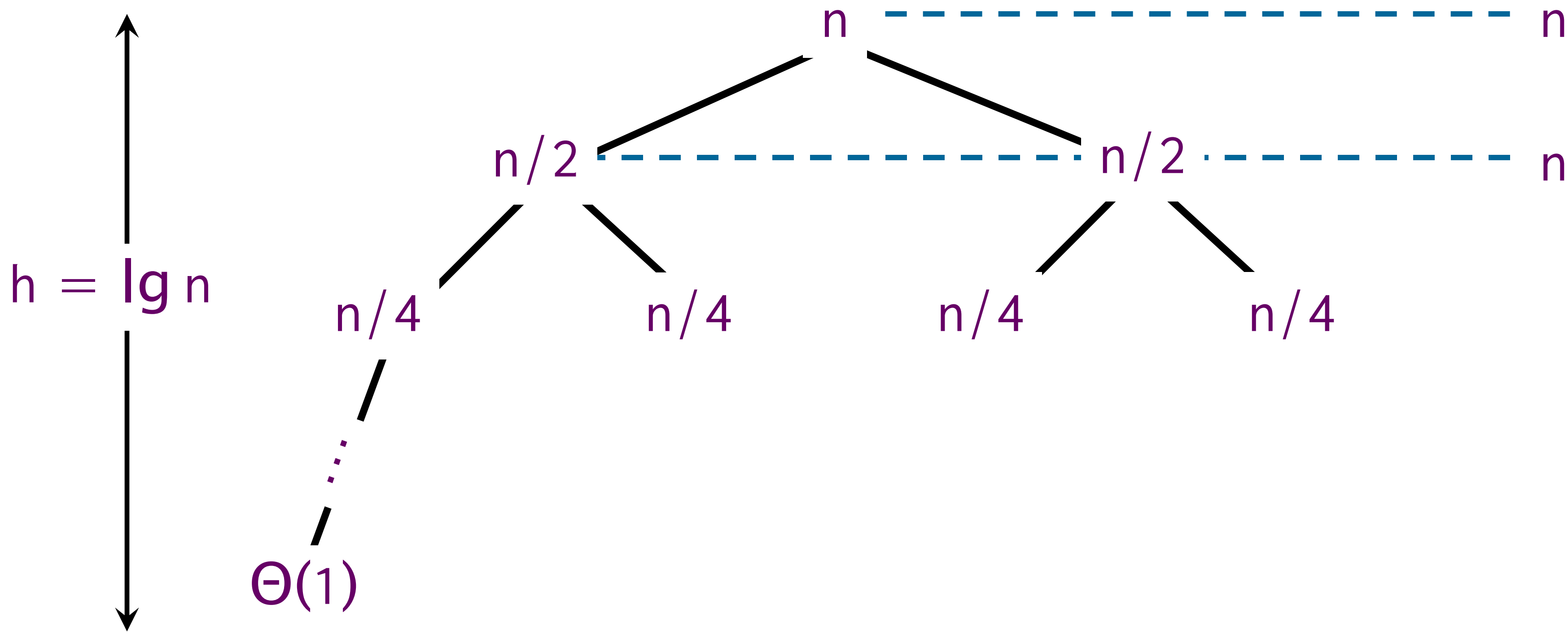
Recursion Tree

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



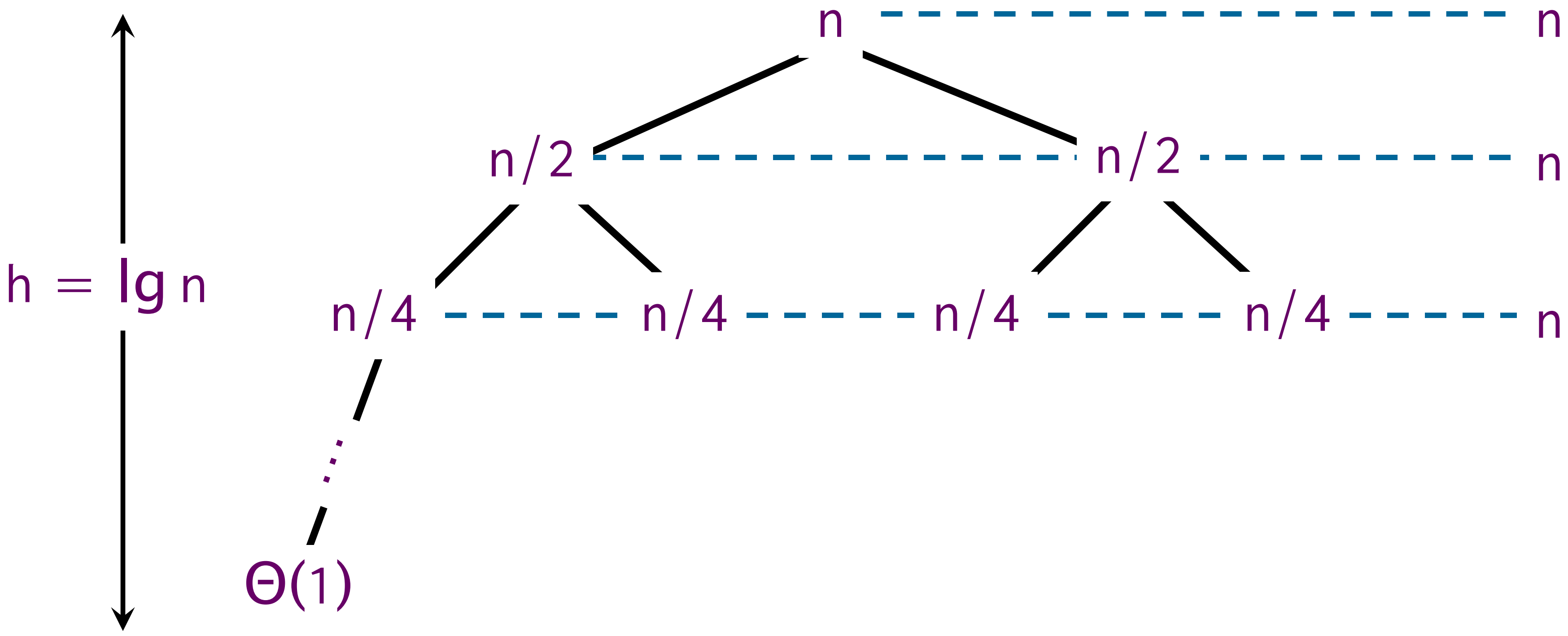
Recursion Tree

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



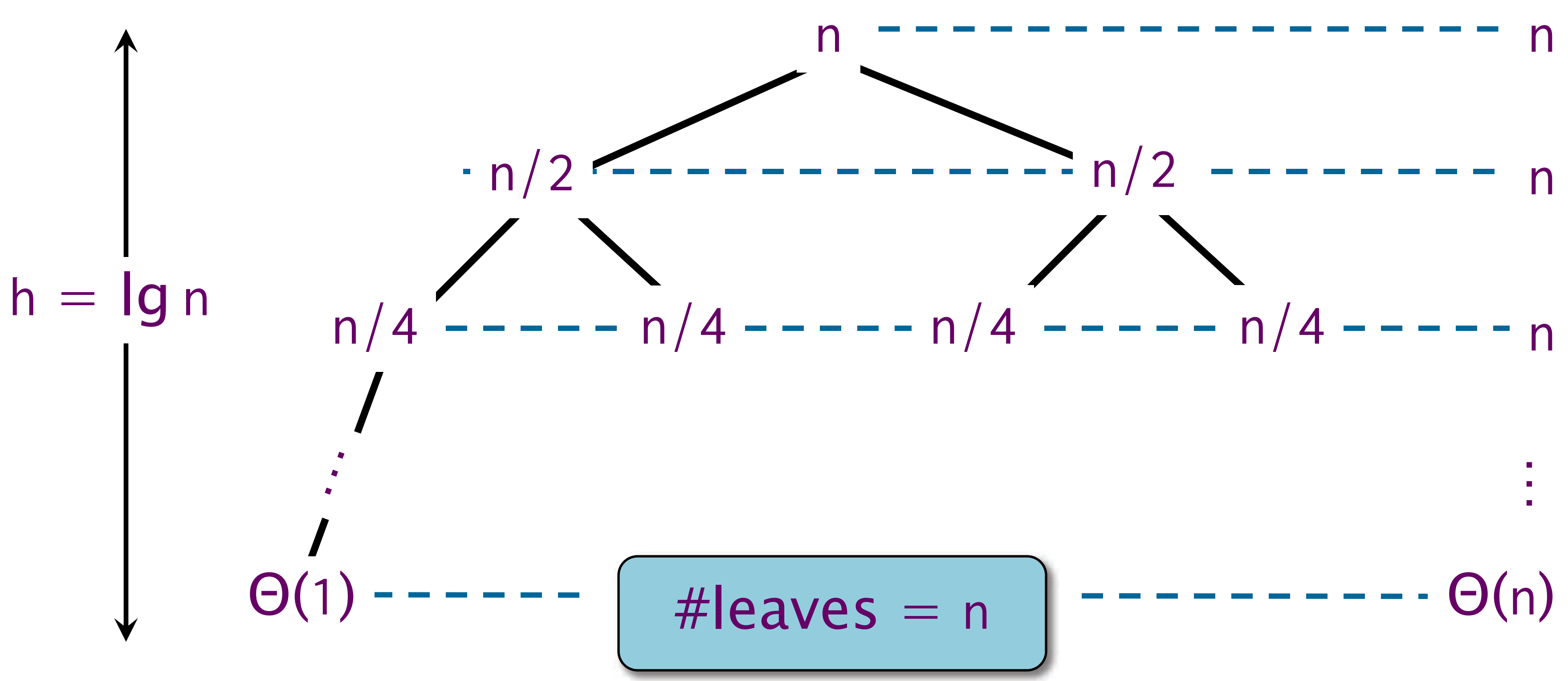
Recursion Tree

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



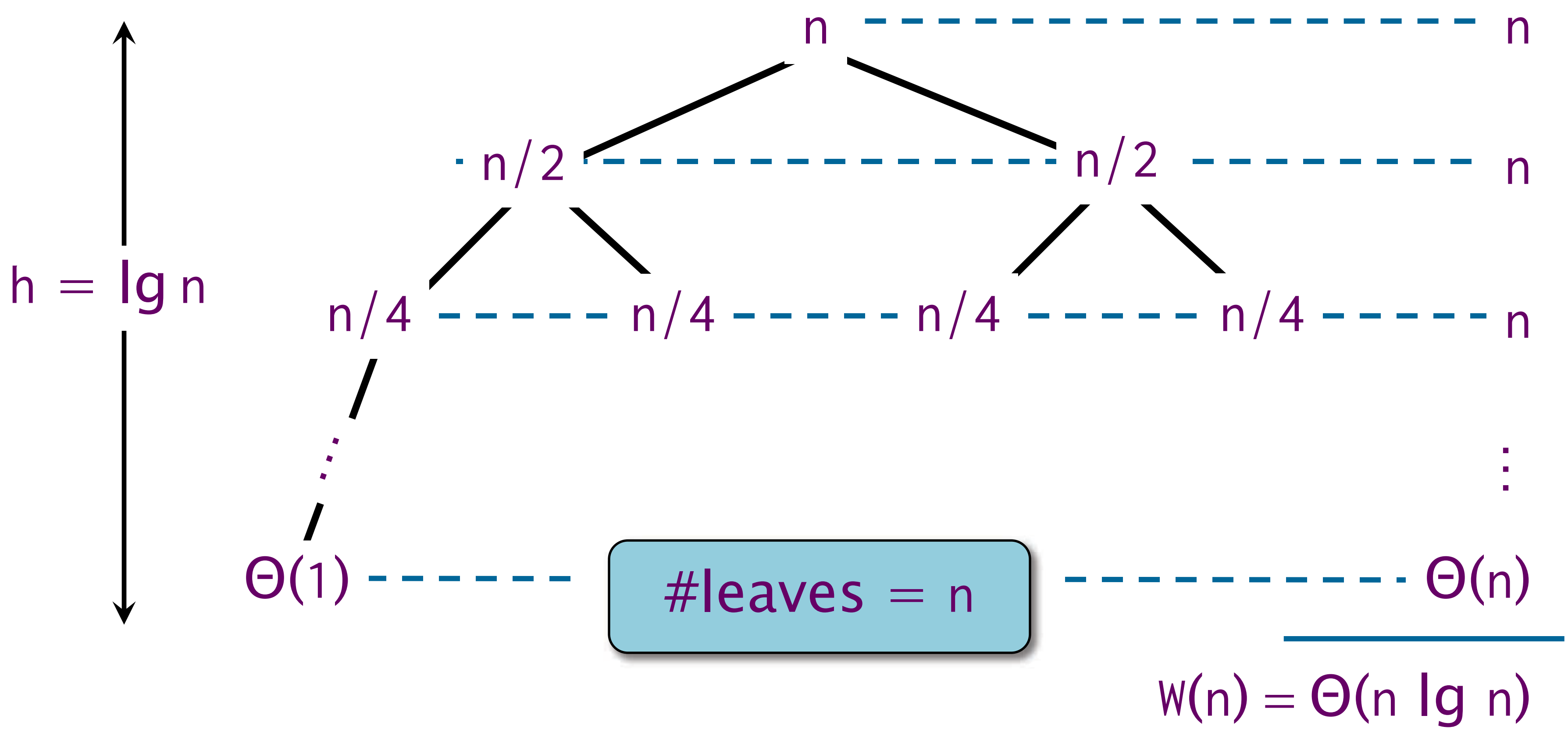
Recursion Tree

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



Recursion Tree

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



Now with Caching

Merge subroutine

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

Merge sort

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

Cache Analysis of Merge Sort

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

Recursion tree

$Q(n)$

Cache Analysis of Merge Sort

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

Recursion tree

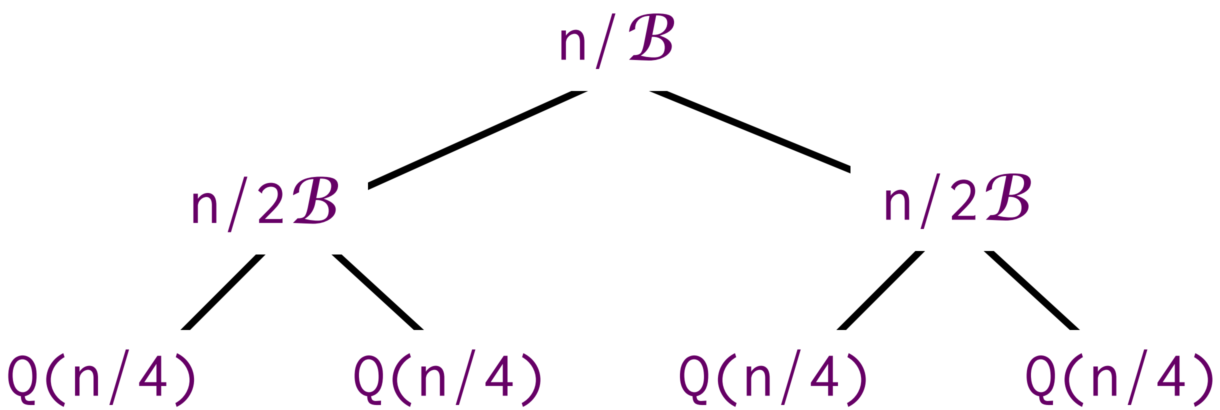
$n/\mathcal{B}$

$Q(n/2)$ $Q(n/2)$

Cache Analysis of Merge Sort

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

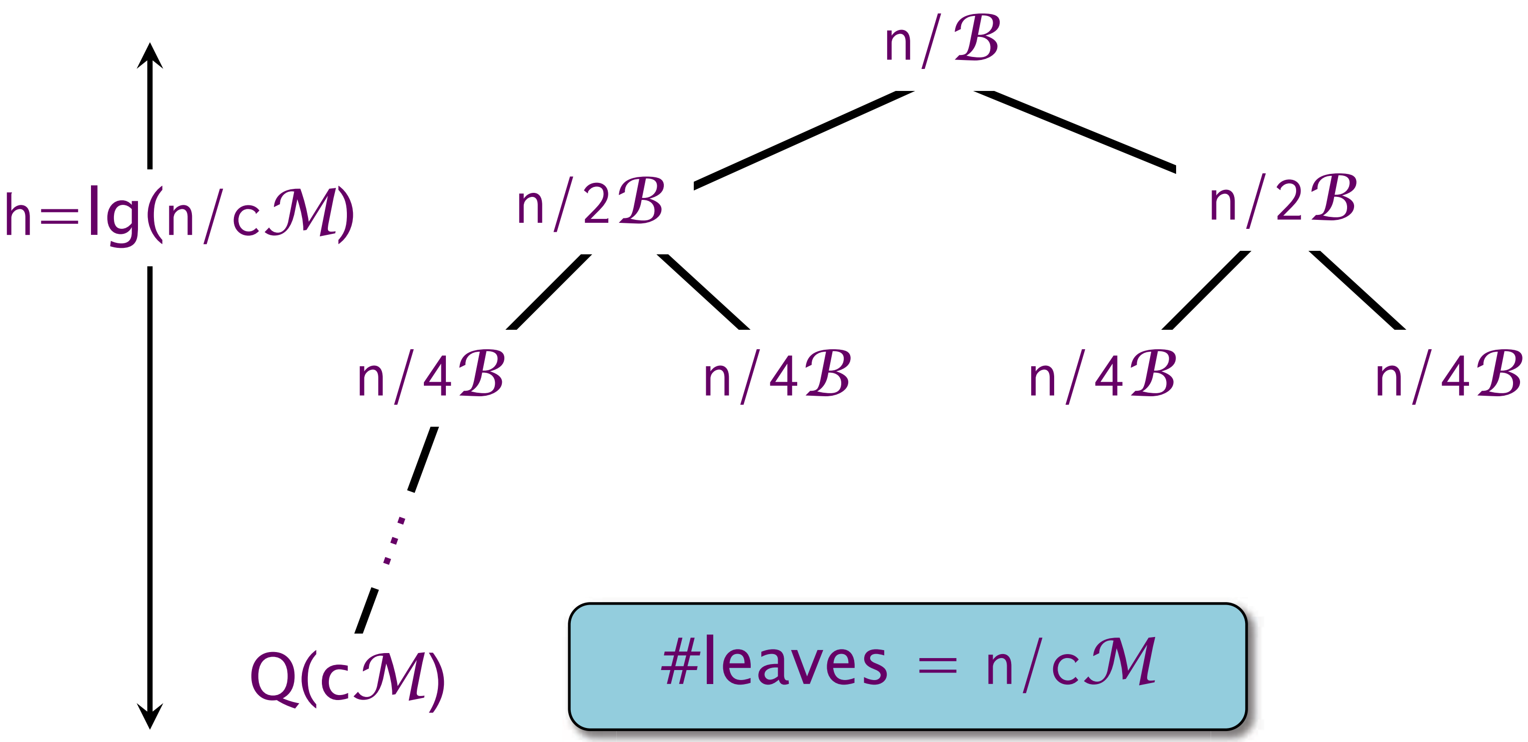
Recursion tree



Cache Analysis of Merge Sort

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

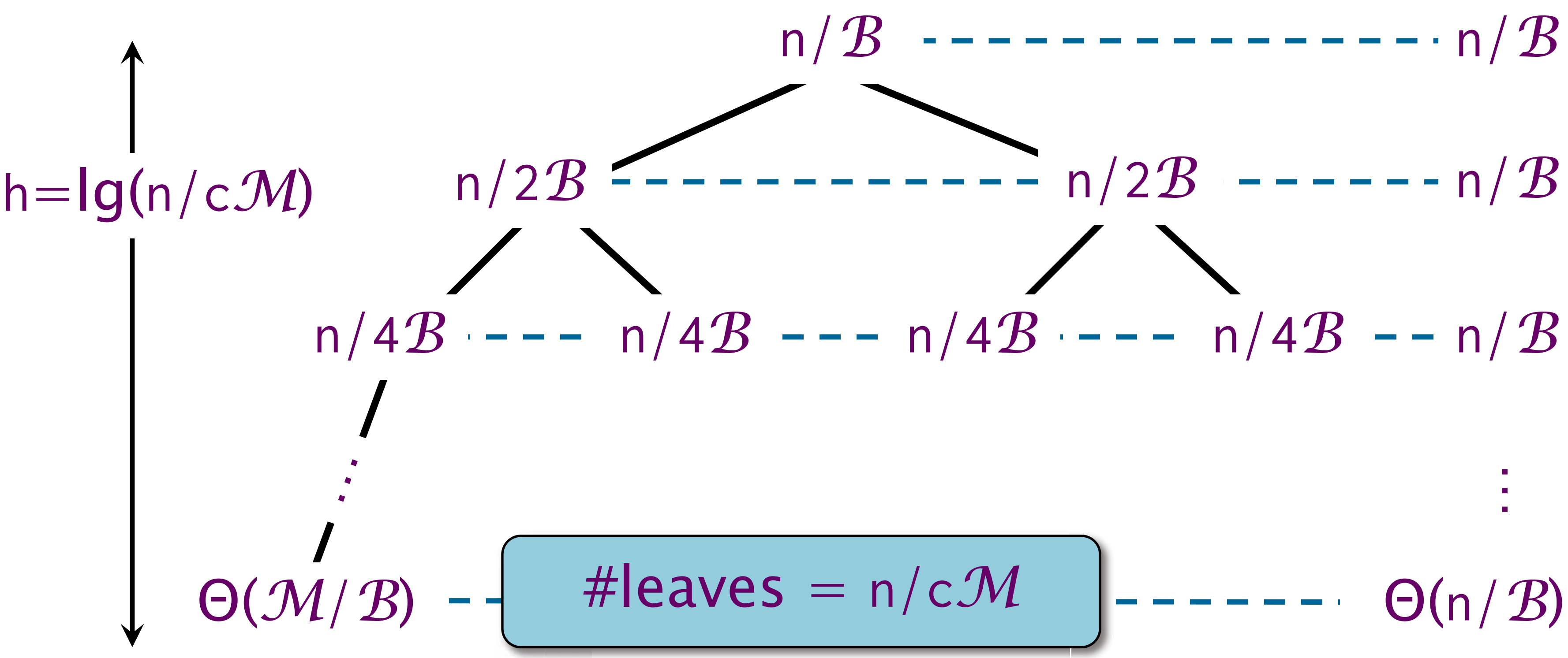
Recursion tree



Cache Analysis of Merge Sort

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

Recursion tree



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

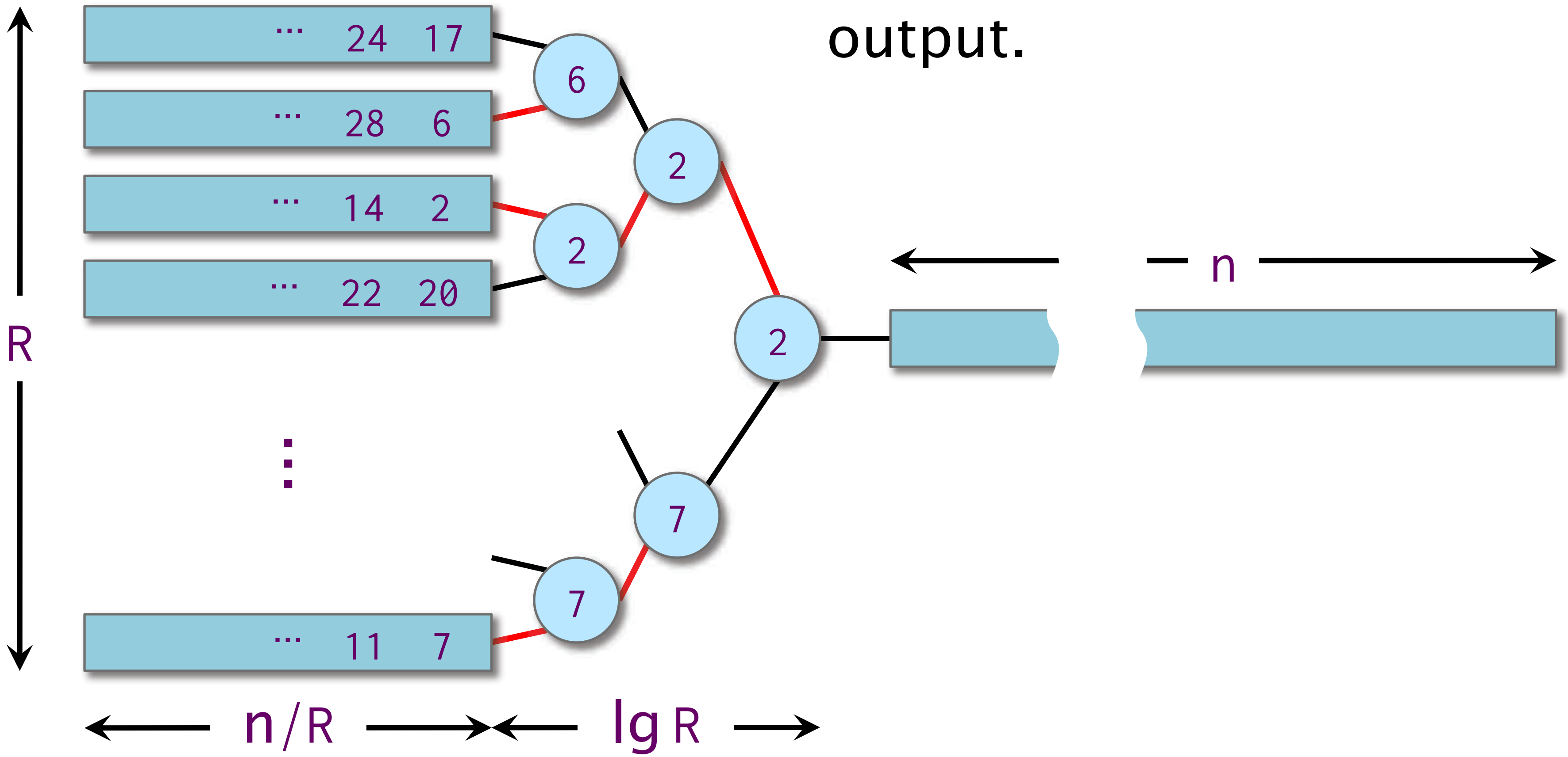
Bottom Line for Merge Sort

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

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

Multiway Merging

IDEA: Merge $R < n$ subarrays with a tournament.

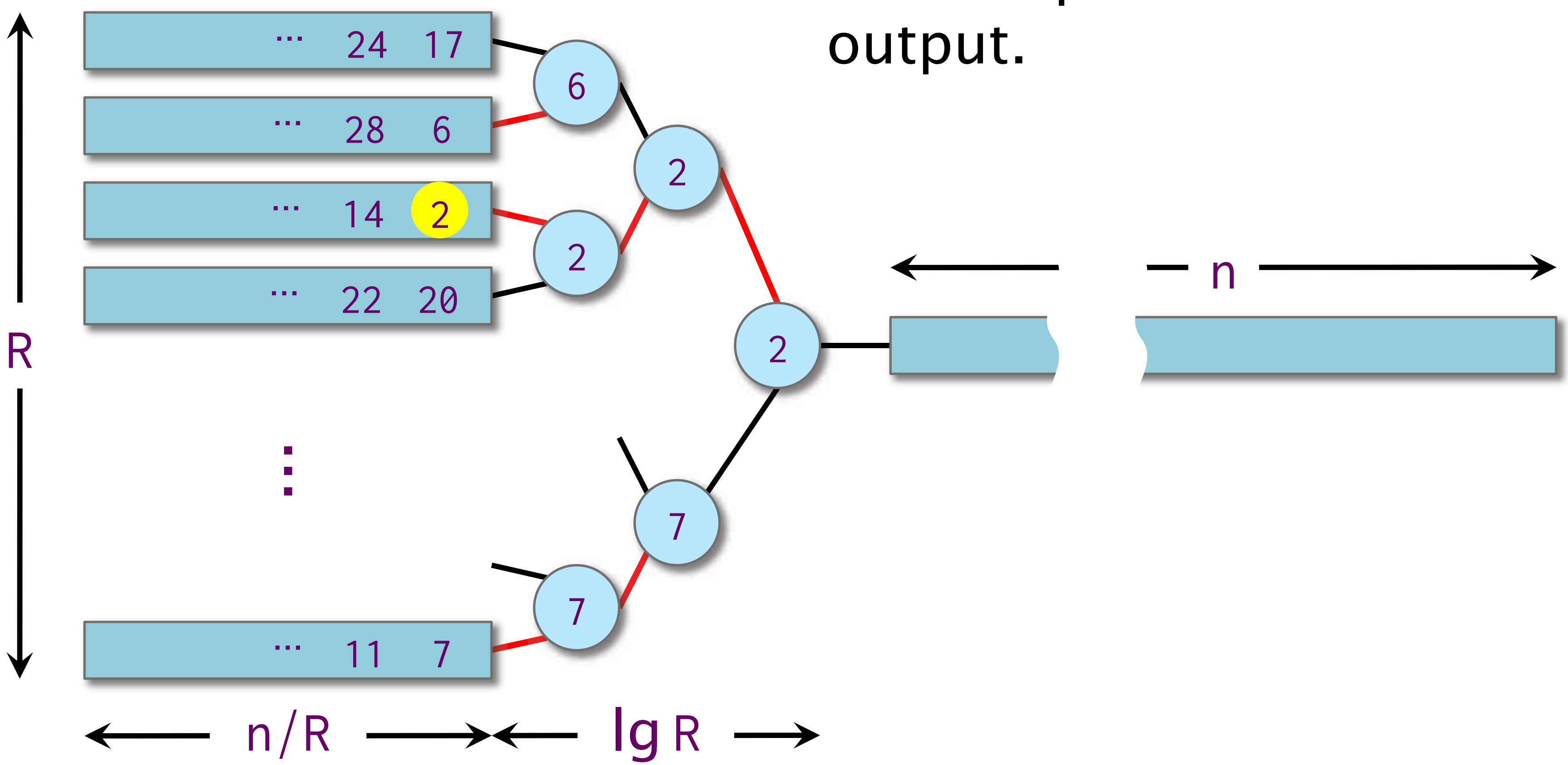


- Tournament takes $\Theta(R)$ work to produce the first output.

Multiway Merging

IDEA: Merge $R < n$ subarrays with a tournament.

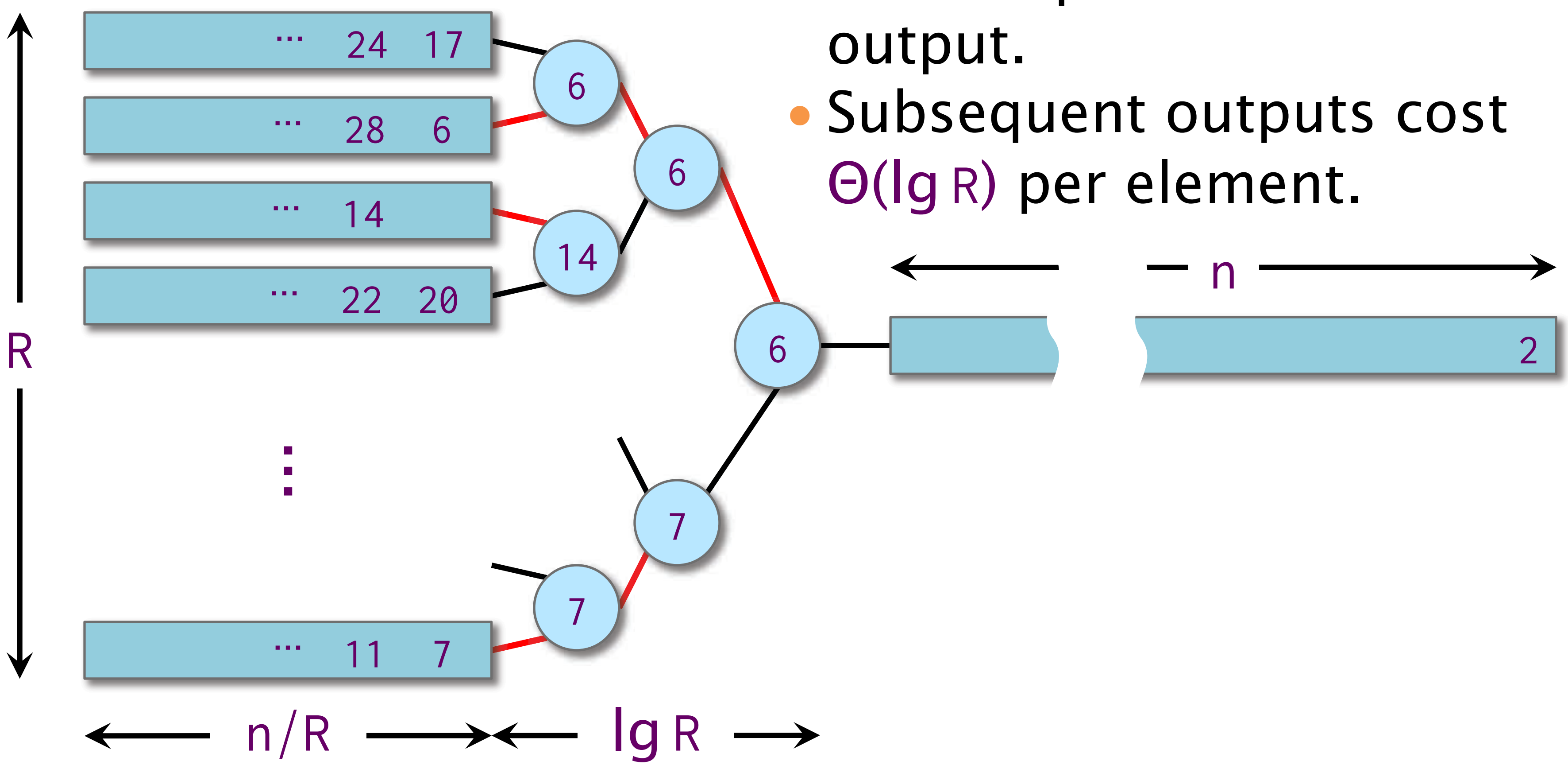
- Tournament takes $\Theta(R)$ work to produce the first output.



Multiway Merging

IDEA: Merge $R < n$ subarrays with a tournament.

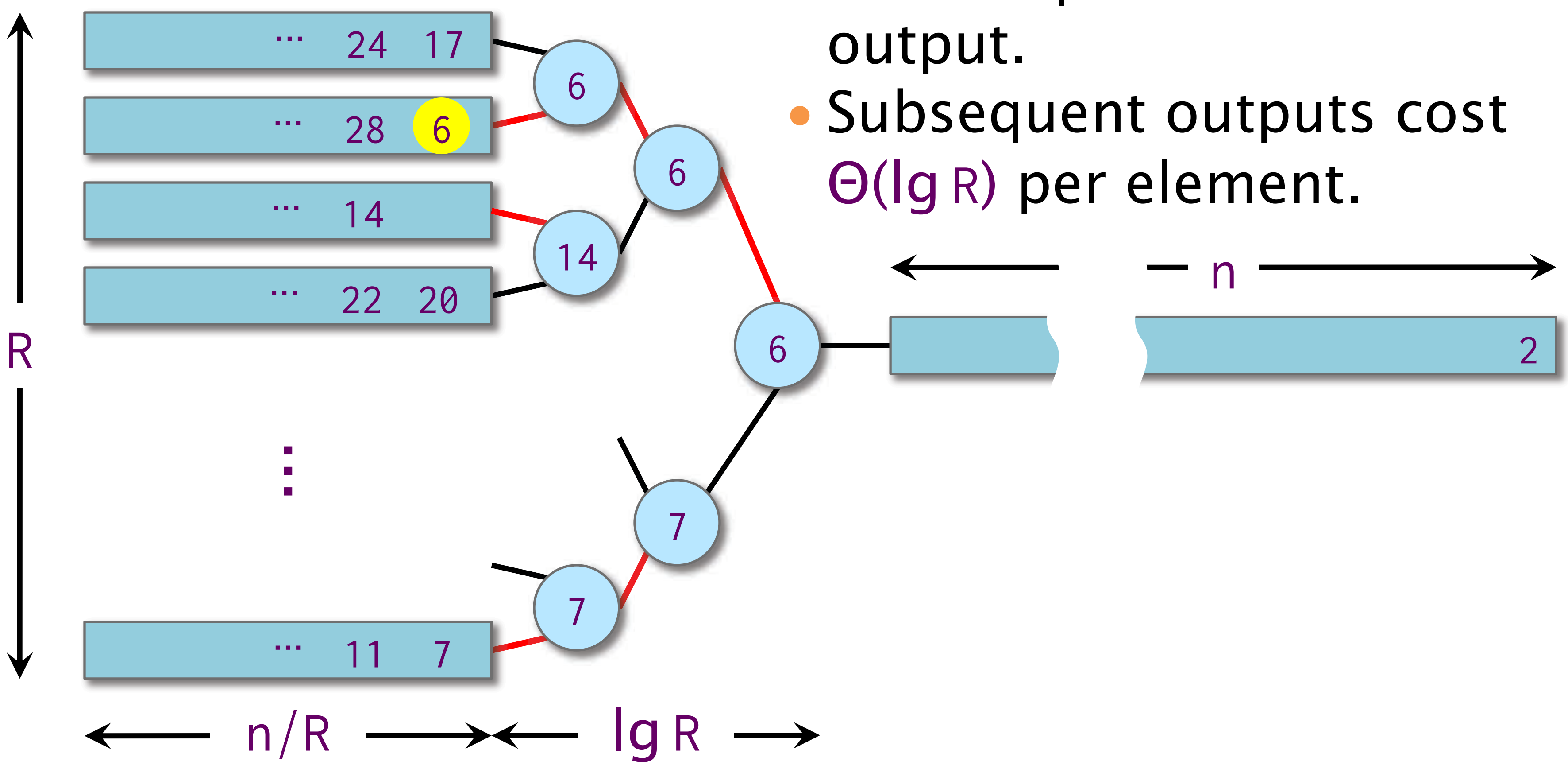
- Tournament takes $\Theta(R)$ work to produce the first output.
- Subsequent outputs cost $\Theta(\lg R)$ per element.



Multiway Merging

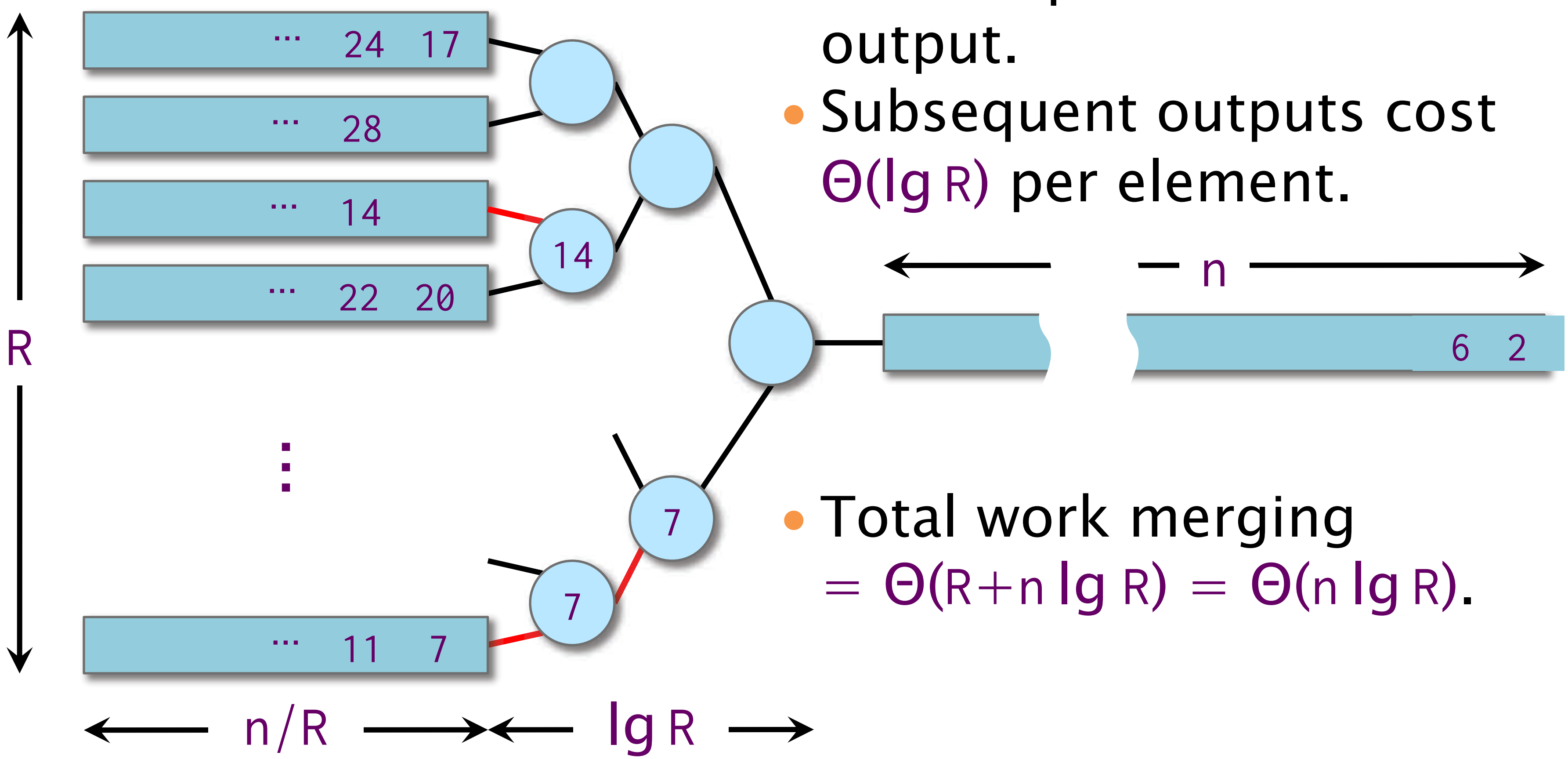
IDEA: Merge $R < n$ subarrays with a tournament.

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



Multiway Merging

IDEA: Merge $R < n$ subarrays with a tournament.

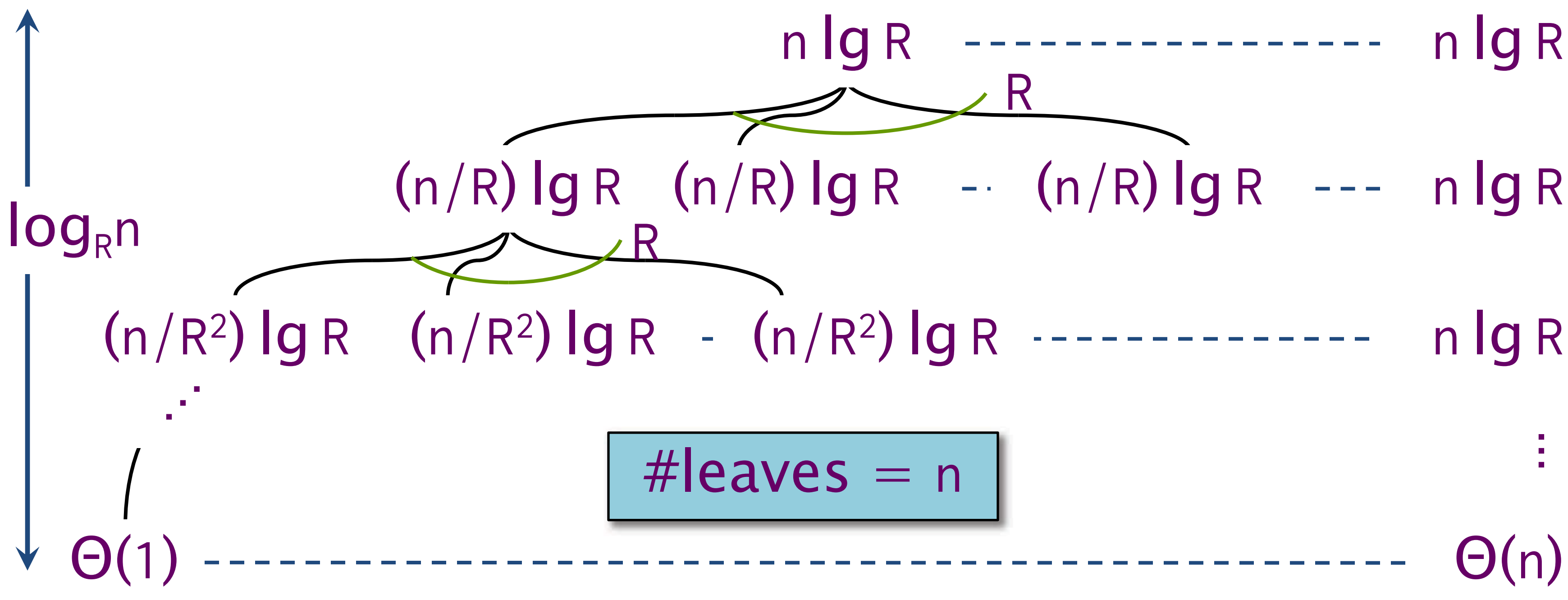


- Tournament takes $\Theta(R)$ work to produce the first output.
- Subsequent outputs cost $\Theta(\lg R)$ per element.
- Total work merging $= \Theta(R + n \lg R) = \Theta(n \lg R)$.

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 $c \le 1$.

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.
$\Rightarrow$ $Q(n) \le \Theta(n/\mathcal{B})$ for merging.

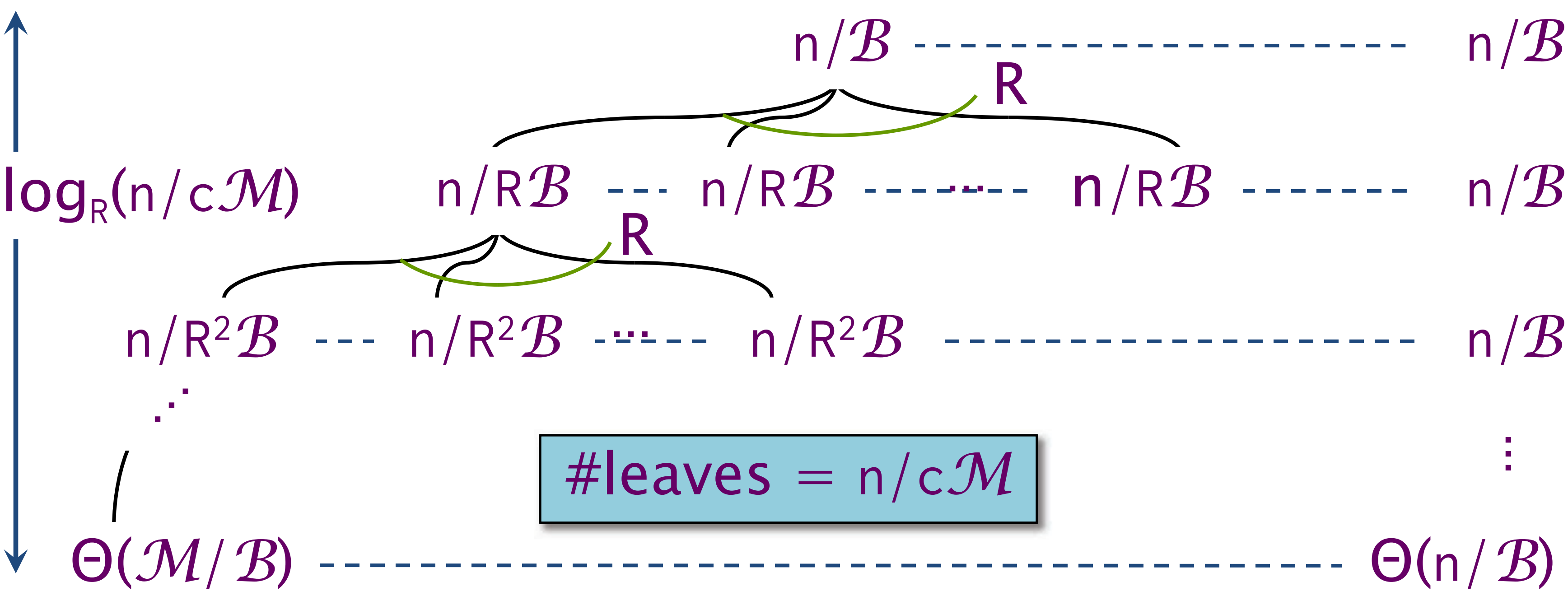
R–way merge sort

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

Cache Analysis

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

Recursion tree



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

Tuning the Voodoo Parameter

We have

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

which decreases as $R \le c\mathcal{M}/\mathcal{B}$ 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 versus 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

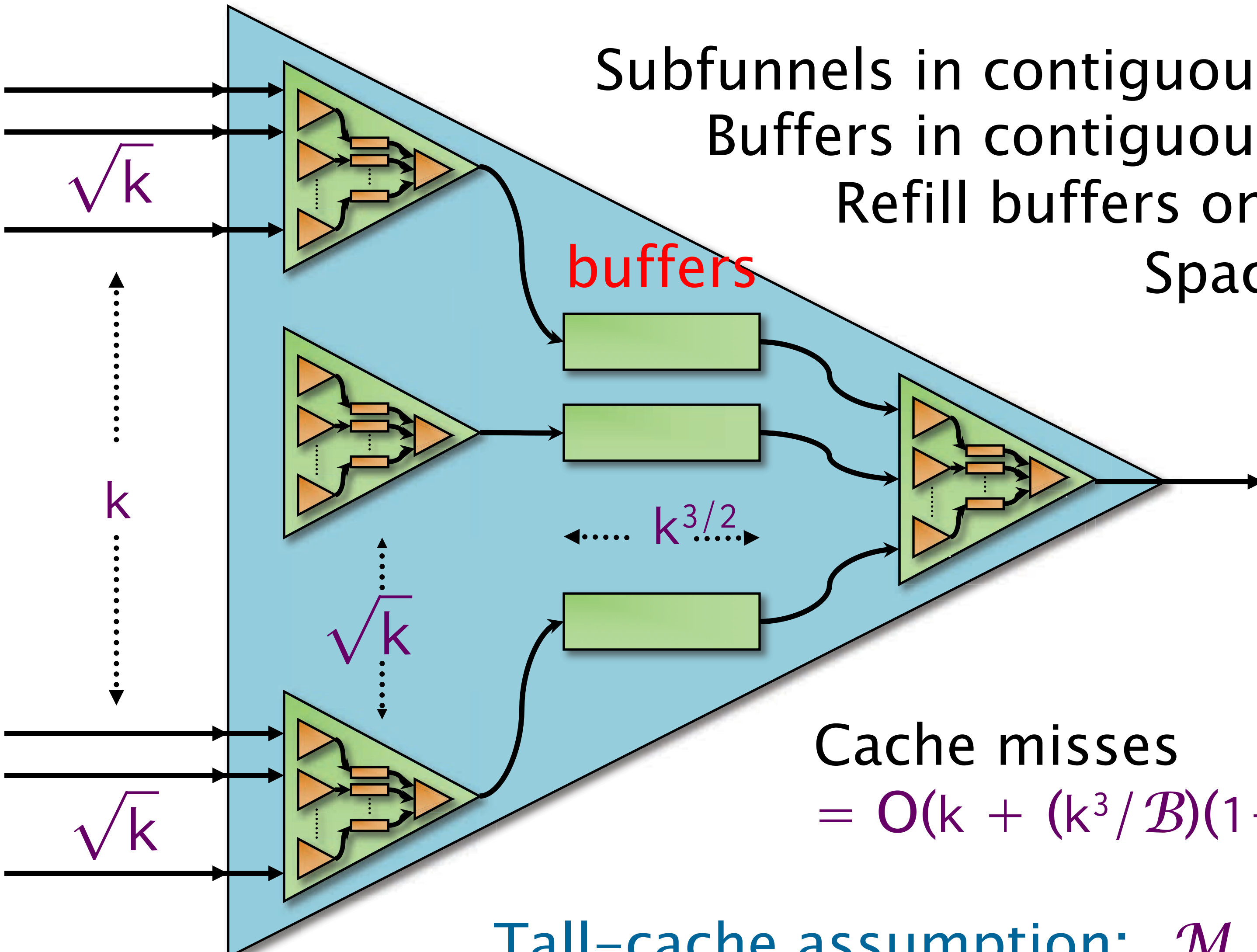
$$\Theta(k + (k^3/\mathcal{B})(1 + \log_{\mathcal{M}} k))$$

cache misses. Thus, funnelsort incurs

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

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 $= O(k^2)$.

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

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

Other C-O 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].

C-O 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.