LECTURE 16
Cache–Oblivious Algorithms

Charles E. Leiserson

November 10, 2011
• Simulation of Heat Diffusion
• Cache–Oblivious Stencil Computations
• Caching and Parallelism
• Cache–Oblivious Sorting
• Simulation of Heat Diffusion
• Cache–Oblivious Stencil Computations
• Caching and Parallelism
• Cache–Oblivious Sorting
Heat Diffusion

2D heat equation

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

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

\( \alpha \) is the \textit{thermal diffusivity}.

Acknowledgment
These stencil slides were heavily 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

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

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

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

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

The 1D heat equation thus reduces to

\[
\frac{u(t + \Delta t, x) - u(t, x)}{\Delta t} = \alpha \frac{u(t, x + \Delta x) - 2u(t, x) + u(t, x - \Delta x)}{(\Delta x)^2}.
\]
A stencil computation updates each point in an array by a fixed pattern, called a stencil.

**Update rule**

\[
u[t + 1][x] = u[t][x] + \frac{\alpha \Delta t}{(\Delta x)^2} (u[t][x + 1] - 2u[t][x] + u[t][x - 1]),\]

\[
\frac{u(t + \Delta t, x) - u(t, x)}{\Delta t} = \alpha \frac{u(t, x + \Delta x) - 2u(t, x) + u(t, x - \Delta x)}{(\Delta x)^2},
\]
• Simulation of Heat Diffusion
• Cache–Oblivious Stencil Computations
• Caching and Parallelism
• Cache–Oblivious Sorting
Recall: Ideal–Cache Model

**Features**

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

**Performance Measures**

- *work* $W$ (ordinary running time).
- *cache misses* $Q$. 
double u[2][N]; // even-odd trick

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

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

Assuming LRU, if N > M, then Q = Θ(NT/B).
Recursively traverse trapezoidal regions of space–time points \((t, x)\) such that

\[
\begin{align*}
t_0 & \leq t < t_1 \\
x_0 + dx_0(t - t_0) & \leq x < x_1 + dx_1(t - t_0) \\
dx_0, \ dx_1 & \in \{-1, 0, 1\}
\end{align*}
\]
If $\text{height} = 1$, compute all space–time points in the trapezoid. Any order of computation is valid, since no point depends on another.
If $\text{width} \geq 2 \cdot \text{height}$, 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.
If $\text{width} < 2 \cdot \text{height}$, cut the trapezoid with a horizontal line through the center. Traverse the bottom trapezoid first, and then the top one.
void trapezoid(int t0, int t1, int x0, int dx0, int x1, int dx1) {
    int lt = t1 - t0;
    if (lt == 1) {
        for (int x = x0; x < x1; x++)
            kernel(t, x);
    } else if (lt > 1) {
        if (2 * (x1 - x0) + (dx1 - dx0) * lt >= 4 * lt) {
            int xm = (2 * (x0 + x1) + (2 + dx0 + dx1) * lt) / 4;
            trapezoid(t0, t1, x0, dx0, xm, -1);
            trapezoid(t0, t1, xm, -1, x1, dx1);
        } else {
            int halflt = lt / 2;
            trapezoid(t0, t0 + halflt, x0, dx0, x1, dx1);
            trapezoid(t0 + halflt, t1, x0 + dx0 * halflt, dx0, x1 + dx1 * halflt, dx1);
        }
    }
}
Each leaf represents $\Theta(hw)$ points where $h = \Theta(w)$. 
Each leaf incurs $\Theta(w/B)$ misses where $w = \Theta(M)$. 
$\Theta(NT/hw)$ leaves. 
#internal nodes = #leaves – 1 do not contribute substantially to $Q$. 
$Q = \Theta(NT/hw) \cdot \Theta(w/B) = \Theta(NT/M^2) \cdot \Theta(M/B) = \Theta(NT/MB)$. 

Cache Analysis
Simulation: 3–Point Stencil

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

- Fully associative LRU cache
  - $B = 4$ points
  - $M = 32$

- Cache-hit latency = 1 cycle
- Cache-miss latency = 10 cycles
Looping v. Trapezoid on Heat
Impact on Performance

Heat equation on a $1000 \times 1000$ grid for 1000 time steps

<table>
<thead>
<tr>
<th></th>
<th>Naïve looping</th>
<th>Trapezoid</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>time (sec)</td>
<td>15.92</td>
<td>12.82</td>
<td>1.24</td>
</tr>
<tr>
<td>L1-cache misses</td>
<td>3,126,220,357</td>
<td>2,368,584,479</td>
<td>1.31</td>
</tr>
<tr>
<td>L2-cache misses</td>
<td>1,338,338,806</td>
<td>371,492,727</td>
<td>36.02</td>
</tr>
<tr>
<td>L3-cache misses</td>
<td>166,802,178</td>
<td>1,050,535</td>
<td>158.77</td>
</tr>
</tbody>
</table>

Q. How can the L3-cache misses be so much better for the cache-oblivious trapezoidal decomposition, but the advantage gained be so marginal?

A. Prefetching and an outstanding memory architecture. One core cannot saturate the memory bandwidth.
Memory Bandwidth

Plenty of bandwidth
• Simulation of Heat Diffusion
• Cache–Oblivious Stencil Computations
• Caching and Parallelism
• Cache–Oblivious Sorting
Theorem. Let $Q_p$ be the number of cache misses in a deterministic Cilk computation when run on $P$ processors, and let $S_p$ be the number of successful steals during the computation. In the ideal cache model, we have

$$Q_p = Q_1 + O(S_p \frac{M}{B}) ,$$

where $M$ is the cache size and $B$ 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 $\frac{M}{B}$ cache (cold) misses, its cache is identical to that in the serial execution. The same is true when a worker resumes a stolen subcomputation after a sync. The number of times these two situations can occur is at most $2S_p$. ■

Moral: Minimizing cache misses in a serial execution minimizes them in parallel executions.
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.
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.
Comparison of Parallel Codes

Heat equation on a $1000 \times 1000$ grid for 1000 time steps (12 processor cores)

<table>
<thead>
<tr>
<th></th>
<th>Parallel looping</th>
<th>Parallel trapezoid</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>time (sec)</td>
<td>2.78</td>
<td>1.42</td>
<td>1.96</td>
</tr>
<tr>
<td>speedup over serial</td>
<td>5.73</td>
<td>9.03</td>
<td></td>
</tr>
<tr>
<td>parallelism</td>
<td>902.97</td>
<td>20.80</td>
<td>43.41</td>
</tr>
<tr>
<td>burdened parallelism</td>
<td>207.97</td>
<td>20.36</td>
<td>10.21</td>
</tr>
</tbody>
</table>

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

Potential bottleneck!
Impediments to Speedup

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

Cilkview can diagnose the first two problems.

Q. How can we diagnose the third?
A. Run $P$ copies of the serial code in parallel — if you have enough memory.

Tools exist to detect lock contention. True and false sharing are harder to detect.
• Simulation of Heat Diffusion
• Cache–Oblivious Stencil Computations
• Caching and Parallelism
• Cache–Oblivious Sorting
void Merge(double *C, double *A, double *B, int na, 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--;
    }
}
void Merge(double *C, double *A, double *B, int na, 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--;
    }
}

Time to merge $n$ elements = $\Theta(n)$. 
void MergeSort(double *B, double *A, int n) {
    if (n==1) {
        B[0] = A[0];
    } else {
        double *C = malloc(n*sizeof(double));
        MergeSort(C, A, n/2);
        MergeSort(C+n/2, A+n/2, n-n/2);
        Merge(B, C, C+n/2, n/2, n-n/2);
    }
}
void MergeSort(double *B, double *A, int n) {
    if (n==1) {
        B[0] = A[0];
    } else {
        double *C = malloc(n*sizeof(double));
        MergeSort(C, A, n/2);
        MergeSort(C+n/2, A+n/2, n-n/2);
        Merge(B, C, C+n/2, n/2, n-n/2);
    }
}
void MergeSort(double *B, double *A, int n) {
    if (n==1) {
        B[0] = A[0];
    } else {
        double *C = malloc(n*sizeof(double));
        MergeSort(C, A, n/2);
        MergeSort(C+n/2, A+n/2, n-n/2);
        Merge(B, C, C+n/2, n/2, n-n/2);
    }
}

W(n) = \begin{cases} 
    \Theta(1) & \text{if } n = 1, \\
    2W(n/2) + \Theta(n) & \text{otherwise.}
\end{cases}
Recursion tree

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

$W(n)$
Recursion tree

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

```
          n
         /   \
   W(n/2)  W(n/2)
```

© 2011 Charles E. Leiserson
Recursion tree

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

![Iteration tree]

© 2011 Charles E. Leiserson
Solve $W(n) = 2W(n/2) + \Theta(n)$. 

\[
\begin{array}{c}
\vdots \\
\Theta(1)
\end{array}
\]
Solve \( W(n) = 2W(n/2) + \Theta(n) \).
Solve $W(n) = 2W(n/2) + \Theta(n)$. 
Solve \( W(n) = 2W(n/2) + \Theta(n) \).
Solve $W(n) = 2W(n/2) + \Theta(n)$. 

$h = \lg n$
Solve $W(n) = 2W(n/2) + \Theta(n)$.
Recursion tree

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

\[ h = \lg n \]

\[ W(n) = \Theta(n \lg n) \]
Merge subroutine

\[ Q(n) = \Theta(n/B). \]

Merge sort

\[ Q(n) = \begin{cases} 
\Theta(n/B) & \text{if } n \leq cM, \text{ const } c \leq 1, \\
2Q(n/2) + \Theta(n/B) & \text{otherwise}. 
\end{cases} \]
Recursion tree

\[ Q(n) = \begin{cases} 
\Theta(n/B) & \text{if } n \leq cM, \text{ const } c \leq 1, \\
2Q(n/2) + \Theta(n/B) & \text{otherwise.} 
\end{cases} \]

\[ h = \log(n/cM) \]

\[ Q(n) = \Theta((n/B) \log(n/M)) \]
Bottom Line for Merge Sort

\[ Q(n) = \begin{cases} 
\Theta(n/B) & \text{if } n \leq cM, \text{ const } c \leq 1. \\
2Q(n/2) + \Theta(n/B) & \text{otherwise.}
\end{cases} \]

\[ = \Theta((n/B) \lg(n/M)) \]

- For \( n \gg M \), we have \( \lg(n/M) \approx \lg n \), and thus \( W(n)/Q(n) \approx \Theta(B) \).
- For \( n \approx M \), we have \( \lg(n/M) \approx \Theta(1) \), and thus \( W(n)/Q(n) \approx \Theta(B \lg n) \).
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.
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
  $\quad = \Theta(R + n \cdot lg R) = \Theta(n \cdot lg R)$. 

© 2011 Charles E. Leiserson
Multiway Merge Sort

\[ W(n) = \begin{cases} 
\Theta(1) & \text{if } n = 1, \\
R \cdot W\left(\frac{n}{R}\right) + \Theta(n \lg R) & \text{otherwise.} 
\end{cases} \]

**recursion tree**

\[ \begin{array}{c}
\log_R n \\
(n/R) \lg R \\
(n/R^2) \lg R \\
\vdots \\
\Theta(1) \\
\end{array} \quad \begin{array}{c}
\text{#leaves} = n \\
(n/R) \lg R \\
(n/R^2) \lg R \\
\vdots \\
\Theta(n) \\
\end{array} \]

\[ W(n) = \Theta((n \lg R) \log_R n + n) = \Theta((n \lg R)(\lg n)/\lg R + n) = \Theta(n \lg n) \]

Same as binary merge sort.
Consider the $R$-way merging of contiguous arrays of total size $n$. If $R < c \frac{M}{B}$, the entire tournament plus 1 block from each array can fit in cache.

$\Rightarrow Q(n) \leq \Theta(n/B)$.

**$R$-way merge sort**

$$Q(n) \leq \begin{cases} \Theta(n/B) & \text{if } n < cM, \\ R \cdot Q(n/R) + \Theta(n/B) & \text{otherwise}. \end{cases}$$
Cache Analysis

\[ Q(n) \leq \begin{cases} 
\Theta(n/B) & \text{if } n < cM, \\
R \cdot Q(n/R) + \Theta(n/B) & \text{otherwise.}
\end{cases} \]

\[ \Theta(M/B) \quad \Theta(n/B) \]

\[ Q(n) = \Theta((n/B) \log_R(n/M)) \]

Recursion tree:

- \( \log_R(n/cM) \)
- \( n/B \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
- \( R \)
- \( n/B \)
We have
\[ Q(n) = \Theta((n/B) \log_R(n/M)), \]
which decreases as \( R \) increases. \( \therefore \) Choose \( R \) as big as possible \( \Rightarrow R = \Theta(M/B). \)

By the tall-cache assumption and the fact that \( \log_M(n/M) = \Theta((\lg n)/\lg M) \), we have
\[
Q(n) = \Theta((n/B) \log_{M/B}(n/M))
= \Theta((n/B) \log_M(n/M))
= \Theta((n \lg n)/B \lg M). 
\]

Hence, we have \( W(n)/Q(n) \approx \Theta(B \lg M). \)
Multiway versus Binary Merge Sort

We have

\[ Q_{\text{multiway}}(n) = \Theta\left(\frac{n \lg n}{B \lg M}\right) \]

versus

\[ Q_{\text{binary}}(n) = \Theta\left(\frac{n}{B} \lg \left(\frac{n}{M}\right)\right). \]

If \( n \gg M \), then \( \lg \left(\frac{n}{M}\right) \approx \lg n \), and thus multiway merge sort saves a factor of \( \Theta(\lg M) \) in cache misses.

Example

- L1-cache: \( M = 2^{15}, B = 2^6 \Rightarrow 9 \times \) savings.
- L2-cache: \( M = 2^{18}, B = 2^6 \Rightarrow 12 \times \) savings.
- L3-cache: \( M = 2^{23}, B = 2^6 \Rightarrow 17 \times \) 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

$$
\Theta \left( k + \frac{k^3}{B} \left( 1 + \log_M k \right) \right)
$$
cache misses. Thus, funnelsort incurs

$$
Q(n) \leq n^{1/3}Q\left( n^{2/3} \right) + \Theta \left( n^{1/3} + \frac{n}{B} \left( 1 + \log_M n \right) \right)
= \Theta \left( 1 + \frac{n}{B} \left( 1 + \log_M n \right) \right),
$$
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

$= O(k + (k^3/B)(1 + \log M k))$

Tall–cache assumption: $M = \Omega(B^2)$.
Other C–O Algorithms

Matrix Transposition/Addition

$\Theta(1 + mn/B)$

Straightforward recursive algorithm.

Strassen’s Algorithm

$\Theta(n + n^2/B + n^{\lg 7}/BM^{(\lg 7)/2 - 1})$

Straightforward recursive algorithm.

Fast Fourier Transform

$\Theta(1 + (n/B)(1 + \log_M n))$


LUP–Decomposition

$\Theta(1 + n^2/B + n^3/BM^{1/2})$

Recursive algorithm due to Sivan Toledo [T97].
Ordered–File Maintenance

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

B–Trees

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

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

Priority Queues

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

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