Outline

1. Simulation of Heat Diffusion
2. Cache-Oblivious Stencil Computations
3. 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 thermal diffusivity.

Related problems:
Diffusion of one material within another; Black-Scholes option pricing; Brownian motion.
2D Heat-Diffusion Simulation

Before

After

6.172
1D Heat Equation

\[ \frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2} \]
### Finite-Difference Approximation

#### Approximating partial derivatives

<table>
<thead>
<tr>
<th>Derivative</th>
<th>Approximation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \frac{\partial u}{\partial t}(t, x) )</td>
<td>( \frac{u(t + \Delta t, x) - u(t, x)}{\Delta t} )</td>
</tr>
<tr>
<td>( \frac{\partial u}{\partial x}(t, x) )</td>
<td>( \frac{u(t, x + \Delta x/2) - u(t, x - \Delta x/2)}{\Delta x} )</td>
</tr>
<tr>
<td>( \frac{\partial^2 u}{\partial x^2}(t, x) )</td>
<td>( \frac{(\partial u/\partial x)(t, x + \Delta x/2) - (\partial u/\partial x)(t, x - \Delta x/2)}{\Delta x} )</td>
</tr>
<tr>
<td></td>
<td>( \frac{u(t, x + \Delta x) - 2u(t, x) + u(t, x - \Delta x)}{(\Delta x)^2} )</td>
</tr>
</tbody>
</table>

**Discretized heat equation**

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

**CFL condition**

Stable only if 
\[
\Delta t < \left( \frac{\Delta x}{2 \alpha} \right)^2
\]

*[Courant, Friedrichs, Lewy 1928]*
Finite-Difference Approximation

Approximating partial derivatives

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

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

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

Discretized heat equation

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

CFL condition

Stable only if \(\Delta t < \frac{(\Delta x)^2}{2\alpha}\).

[Courant, Friedrichs, Lewy 1928]
**Finite-Difference Approximation**

### Approximating partial derivatives

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

### CFL condition

Stable only if \(\Delta t < (\Delta x)^2/(2\alpha)\).

[Courant, Friedrichs, Lewy 1928]
3-point stencil

$$\frac{u(t + \Delta t, x) - u(t, x)}{\Delta t} = \alpha \frac{u(t, x + \Delta x) - 2u(t, x) + u(t, x - \Delta x)}{(\Delta x)^2}$$
3-point stencil

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

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

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

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

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

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

**Stencil computations:**

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

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

Stencil computations:

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

Update rule:

\[
u[t + 1][x] = u[t][x] + \frac{\alpha \Delta t}{(\Delta x)^2} (u[t][x + 1] - 2u[t][x] + u[t][x - 1])
\]
static inline double kernel(double um1, double u0, double u1) {
    return u0 + CONSTANT * (um1 - 2*u0 + u1);
}

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

for (t = 0; t < T; ++t) { // time loop
    u[(t+1)%2][0] = left_boundary();
    for (i = 1; i < N - 1; ++i) // space loop
        u[(t+1)%2][i] =
            kernel(u[t%2][i-1], u[t%2][i], u[t%2][i+1]);
    u[(t+1)%2][N-1] = right_boundary();
}
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 $Q$. 

Symbols:
- $P$: Processor
- $M$: Cache size
- $B$: Cache-line length
- $M/B$: Cache lines
- Cache
- Memory
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 $Q$. 
Cache behavior of looping

Assuming LRU, if \( N > M \) then \( Q = \Theta \left( \frac{NT}{B} \right) \).

On any cache, if \( N > 2M \) then \( Q = \Theta \left( \frac{NT}{B} \right) \).
Cache behavior of looping

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

On any cache, if $N > 2M$ then $Q = \Theta \left( \frac{NT}{B} \right)$. 

(read miss)
Cache behavior of looping

\[ Q = \Theta \left( \frac{NT}{B} \right) \]

On any cache, if \( N > 2M \) then \( Q = \Theta \left( \frac{NT}{B} \right) \).
Cache behavior of looping

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

On any cache, if $N > 2M$ then $Q = \Theta\left(\frac{NT}{B}\right)$.
Cache behavior of looping

Assuming LRU, if $N > M$ then $Q = \Theta \left( \frac{NT}{B} \right)$. On any cache, if $N > 2M$ then $Q = \Theta \left( \frac{NT}{B} \right)$. 

\[ t \]

\[ x \]
Cache behavior of looping

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

On any cache, if $N > 2M$ then $Q = \Theta(NT/B)$. 
Cache behavior of looping

Assuming LRU, if \( N > M \) then 
\[
Q = \Theta \left( \frac{NT}{B} \right)
\]

On any cache, if \( N > 2M \) then 
\[
Q = \Theta \left( \frac{NT}{B} \right)
\]
Cache behavior of looping

Cache misses:
Assuming LRU, if \( N > M \) then \( Q = \Theta \left( \frac{NT}{B} \right) \).

On any cache, if \( N > 2M \) then \( Q = \Theta \left( \frac{NT}{B} \right) \).
Cache behavior of looping

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

On any cache, if $N > 2M$ then $Q = \Theta \left( \frac{NT}{B} \right)$. 
Cache behavior of looping

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

On any cache, if $N > 2M$ then $Q = \Theta \left( \frac{NT}{B} \right)$. 
Cache behavior of looping

Cache misses:

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

On any cache, if $N > 2M$ then $Q = \Theta \left( \frac{NT}{B} \right)$. 

\(t\)

\(X\)

line evicted
Cache behavior of looping

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

On any cache, if $N > 2M$ then $Q = \Theta \left( \frac{NT}{B} \right)$. 

*t*  

\[ N > M \]

\[ N > 2M \]

\[ Q = \Theta \left( \frac{NT}{B} \right) \]
Cache behavior of looping

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

On any cache, if $N > 2M$ then $Q = \Theta(NT/B)$. 
Cache behavior of looping

Assuming LRU, if \( N > 2M \) then
\[
Q = \Theta \left( \frac{NT}{B} \right)
\]

On any cache, if \( N > 2M \) then
\[
Q = \Theta \left( \frac{NT}{B} \right)
\]
Cache behavior of looping

Assuming LRU, if \( N > M \) then \( Q = \Theta \left( \frac{NT}{B} \right) \).

On any cache, if \( N > 2M \) then \( Q = \Theta \left( \frac{NT}{B} \right) \).

**read miss (again)**
Cache behavior of looping

Cache misses:
Assuming LRU, if $N > M$ then $Q = \Theta(NT/B)$.

On any cache, if $N > 2M$ then $Q = \Theta(NT/B)$. 
Outline

1. Simulation of Heat Diffusion
2. Cache-Oblivious Stencil Computations
3. Cache-Oblivious Sorting
Cache-Oblivious Algorithm for 3-point Stencil

Recursively traverse trapezoidal regions of spacetime points \((t, x)\) such that:

\[
\begin{align*}
t_0 & \leq t < t_1 \\
x_0 + \dot{x}_0(t - t_0) & \leq x < x_1 + \dot{x}_1(t - t_0) \\
\dot{x}_i & \in \{-1, 0, 1\}
\end{align*}
\]
Cache-Oblivious Algorithm for 3-point Stencil

Recursively traverse trapezoidal regions of spacetime points \((t, x)\) such that:

\[
t_0 \leq t < t_1
\]

\[
x_0 + \dot{x}_0(t - t_0) \leq x < x_1 + \dot{x}_1(t - t_0)
\]

\[
\dot{x}_i \in \{-1, 0, 1\}
\]

Newton’s notation:

\[
\dot{x} = \frac{dx}{dt}.
\]
Cache-Oblivious Algorithm for 3-point Stencil

Recursively traverse trapezoidal regions of spacetime points \((t, x)\) such that:

\[
t_0 \leq t < t_1 \\
{x_0 + \dot{x}_0(t - t_0) \leq x < x_1 + \dot{x}_1(t - t_0)} \\
\dot{x}_i \in \{-1, 0, 1\}
\]
If height $= 1$, compute all spacetime points in the trapezoid.

Any order of computation is valid, because these points do not depend upon each other.
If width $\geq 2 \cdot$ height, cut the trapezoid with a line of slope $-1$ through the center.

Traverse first the trapezoid on the left, then the one on the right.
If width $\geq 2 \cdot$ height, cut the trapezoid with a line of slope $-1$ through the center.

Traverse first the trapezoid on the left, then the one on the right.

*no dependencies from right to left*
Time Cut

If width $< 2 \cdot$ height, cut the trapezoid with a horizontal line through the center.

Traverse the bottom trapezoid first, then the top one.
void trapezoid(int \(t_0\), int \(t_1\), int \(x_0\), int \(\dot{x}_0\), int \(x_1\), int \(\dot{x}_1\))
{
    int \(\Delta t = t_1 - t_0\);
    if (\(\Delta t == 1\)) {
        for (int \(x = x_0\); \(x < x_1\); ++\(x\))
            kernel(t_0, \(x\));
    } else if (\(\Delta t > 1\)) {
        if (\(2 * (x_1 - x_0) + (\dot{x}_1 - \dot{x}_0) * \Delta t >= 4 * \Delta t\)) {
            int \(x_m = (2 * (x_0 + x_1) + (2 + \dot{x}_0 + \dot{x}_1) * \Delta t) / 4\);
            trapezoid(t_0, \(t_1\), \(x_0\), \(\dot{x}_0\), \(x_m\), -1);
            trapezoid(t_0, \(t_1\), \(x_m\), -1, \(x_1\), \(\dot{x}_1\));
        } else {
            int \(s = \Delta t / 2\);
            trapezoid(t_0, \(t_0 + s\), \(x_0\), \(\dot{x}_0\), \(x_1\), \(\dot{x}_1\));
            trapezoid(t_0 + \(s\), \(t_1\), \(x_0 + \dot{x}_0 * s\), \(\dot{x}_0\), \(x_1 + \dot{x}_1 * s\), \(\dot{x}_1\));
        }
    }
}
Cache Analysis

Each leaf: $\Theta(hw)$ points with $h = \Theta(w)$.

Each leaf incurs $O((w + h)/B) = O(w/B)$ misses with $w + h = \Theta(M)$.

$\Theta(NT/hw)$ leaves.

$\#\text{internal nodes} = \#\text{leaves} - 1$ do not contribute substantially to $Q$.

$Q = \Theta(NT/hw) \cdot O(w/B) = \Theta(NT/M^2) \cdot O(M/B) = O(NT/MB)$. 
Simulation: 3-Point Stencil

- Rectangular region
  - \( N = 95 \).
  - \( T = 87 \).

- Fully associative LRU cache.
  - \( B = 4 \) points.
  - \( M = 4, 8, 16, \) or 32 cache lines.

- Cache hit latency = 1 cycle.

- Cache miss latency = 10 cycles.
### Performance of Heat Equation Measurements

<table>
<thead>
<tr>
<th>array size</th>
<th>$T$</th>
<th>loop</th>
<th>trapezoid</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1000 \times 1000$</td>
<td>1000</td>
<td>12.2 s</td>
<td>11.0 s</td>
</tr>
<tr>
<td>$1000 \times 1000$</td>
<td>1000</td>
<td>12.5 s</td>
<td>11.1 s</td>
</tr>
<tr>
<td>4 copies</td>
<td>10</td>
<td>17.5 s</td>
<td>10.4 s</td>
</tr>
<tr>
<td>$10000 \times 10000$</td>
<td>10</td>
<td>76.3 s</td>
<td>10.9 s</td>
</tr>
<tr>
<td>4 copies</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Machine: Xeon E31230 3.2 GHz; 32 KB private L1, 8-way associative; 256 KB private L2, 8-way associative; 8 MB shared “LLC” (L3), 16-way associative; 8 GB RAM in 2×DDR3-1333 memory sticks.
### More experiments

<table>
<thead>
<tr>
<th>array size</th>
<th>$T$</th>
<th>one copy</th>
<th>four copies</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1000 \times 1000$</td>
<td>1000</td>
<td>12.2 s</td>
<td>11.0 s</td>
</tr>
<tr>
<td>100000 $\times 10000$</td>
<td>10</td>
<td>17.5 s</td>
<td>10.4 s</td>
</tr>
<tr>
<td>$1000 \times 1000$</td>
<td>1000</td>
<td>27.7 s</td>
<td>20.5 s</td>
</tr>
<tr>
<td>100000 $\times 10000$</td>
<td>10</td>
<td>39.5 s</td>
<td>16.4 s</td>
</tr>
<tr>
<td>1000 $\times 1000$</td>
<td>1000</td>
<td>12.2 s</td>
<td>11.0 s</td>
</tr>
<tr>
<td>100000 $\times 10000$</td>
<td>10</td>
<td>17.6 s</td>
<td>10.8 s</td>
</tr>
<tr>
<td>$1000 \times 1000$</td>
<td>1000</td>
<td>32.3 s</td>
<td>15.6 s</td>
</tr>
<tr>
<td>100000 $\times 10000$</td>
<td>10</td>
<td>277 s</td>
<td>19.9 s</td>
</tr>
</tbody>
</table>

**Baseline:**

- 1000 $\times 1000$: 12.2 s, 11.0 s, 12.5 s, 11.1 s
- 10000 $\times 10000$: 17.5 s, 10.4 s, 76.3 s, 10.9 s

**Other machine (Phenom II 975):**

- 1000 $\times 1000$: 27.7 s, 20.5 s, 28.0 s, 20.5 s
- 10000 $\times 10000$: 39.5 s, 16.4 s, 80.0 s, 16.5 s

**Remove memory stick (“halve” bandwidth):**

- 1000 $\times 1000$: 12.2 s, 11.0 s, 16.4 s, 11.0 s
- 10000 $\times 10000$: 17.6 s, 10.8 s, (out of memory)

**Saturate memory bus:**

- 1000 $\times 1000$: 32.3 s, 15.6 s, N/A
- 10000 $\times 10000$: 277 s, 19.9 s, N/A
Moral of the story

Collect as much data as you can

- Any single experiment can be misleading.
## Moral of the story

<table>
<thead>
<tr>
<th>Collect as much data as you can</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any single experiment can be misleading.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>One core is more than one fourth of a four-core machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-core programs use more than their “fair share” of caches and memory bandwidth.</td>
</tr>
<tr>
<td>Thus, the best single-core algorithm does not necessarily maximize the aggregate performance of a multi-core machine.</td>
</tr>
</tbody>
</table>
Moral of the story

Collect as much data as you can
- Any single experiment can be misleading.

One core is more than one fourth of a four-core machine
- Single-core programs use more than their “fair share” of caches and memory bandwidth.
- Thus, the best single-core algorithm does not necessarily maximize the aggregate performance of a multi-core machine.

Test your software in multiple environments, creatively
- Overclock, underclock, add/remove hardware, add noise.
Moral of the story

Collect as much data as you can
- Any single experiment can be misleading.

One core is more than one fourth of a four-core machine
- Single-core programs use more than their “fair share” of caches and memory bandwidth.
- Thus, the best single-core algorithm does not necessarily maximize the aggregate performance of a multi-core machine.

Test your software in multiple environments, creatively
- Overclock, underclock, add/remove hardware, add noise.

Cache oblivious algorithms tend to be more robust.
- The trapezoid algorithm is (mostly) insensitive to whatever else is running on your machine.
Cilk and Caching

**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 $Q_P = Q_1 + O(S_P M/B)$, 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 $M/B$ 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.
Cilk and Caching

**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 $Q_P = Q_1 + O(S_P M/B)$, 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 $M/B$ 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 $2S_P$. 

□
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 $Q_P = Q_1 + O(S_P M/B)$, 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 $M/B$ 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.
Outline

1 Simulation of Heat Diffusion

2 Cache-Oblivious Stencil Computations

3 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--;
    }
}
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--;
   }
}
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--;
    }
}
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--;
    }
}
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--;
    }
}
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--;
    }
}
Merging Two Sorted Arrays

```c
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 = $\Theta(n)$. 
Merging Two Sorted Arrays

```c
void MergeSort(double *B, double *A, int n) {
    if (n == 1) {
        B[0] = A[0];
    } else {
        double C0[n/2], C1[n-(n/2)];
        MergeSort(C0, A, n/2);
        MergeSort(C1, A+n/2, n-n/2);
        Merge(B, C0, C1, n/2, n-n/2);
    }
}
```
void MergeSort(double *B, double *A, int n) {
    if (n==1) {
        B[0] = A[0];
    } else {
        double C0[n/2], C1[n-(n/2)];
        MergeSort(C0, A, n/2);
        MergeSort(C1, A+n/2, n-n/2);
        Merge(B, C0, C1, n/2, n-n/2);
    }
}

19 3 12 46 33 4 21 14
void MergeSort(double *B, double *A, int n) {
    if (n==1) {
        B[0] = A[0];
    } else {
        double C0[n/2], C1[n-(n/2)];
        MergeSort(C0, A, n/2);
        MergeSort(C1, A+n/2, n-n/2);
        Merge(B, C0, C1, n/2, n-n/2);
    }
}
void MergeSort(double *B, double *A, int n) {
    if (n==1) {
        B[0] = A[0];
    } else {
        double C0[n/2], C1[n-(n/2)];
        MergeSort(C0, A, n/2);
        MergeSort(C1, A+n/2, n-n/2);
        Merge(B, C0, C1, n/2, n-n/2);
    }
}

19 3 12 46

recursively sort

3 12 19 46

33 4 21 14
void MergeSort(double *B, double *A, int n) {
    if (n==1) {
        B[0] = A[0];
    } else {
        double C0[n/2], C1[n-(n/2)];
        MergeSort(C0, A, n/2);
        MergeSort(C1, A+n/2, n-n/2);
        Merge(B, C0, C1, n/2, n-n/2);
    }
}
void MergeSort(double *B, double *A, int n) {
  if (n==1) {
    B[0] = A[0];
  } else {
    double C0[n/2], C1[n-(n/2)];
    MergeSort(C0, A, n/2);
    MergeSort(C1, A+n/2, n-n/2);
    Merge(B, C0, C1, n/2, n-n/2);
  }
}
Work of Merge Sort

```c
void MergeSort(double *B, double *A, int n) {
    if (n==1) {
        B[0] = A[0];
    } else {
        double C0[n/2], C1[n-(n/2)];
        MergeSort(C0, A, n/2);
        MergeSort(C1, A+n/2, n-n/2);
        Merge(B, C0, C1, n/2, n-n/2);
    }
}
```

$$W(n) = \begin{cases} 
\Theta(1) & \text{if } n = 1; \\
2W(n/2) + \Theta(n) & \text{otherwise.}
\end{cases}$$
Solve \( W(n) = 2W(n/2) + \Theta(n) \).
Recursion Tree

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

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

\#leaves = \frac{n}{2} \rightarrow \frac{n}{4} \rightarrow \frac{n}{8} \rightarrow \ldots \rightarrow \Theta(1)

\text{Depth} h = \log_2 n

W(n) = \Theta(n \log n)
Recursion Tree

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

The recursion tree shows:

- The root node is $n$.
- The left child of the root is $n/2$.
- The right child of the root is $n/2$.
- The left child of the left child is $n/4$.
- The right child of the right child is $n/4$.
- The left child of the left child of the left child is $\Theta(1)$.
- The number of leaves is $n$.
Solve $W(n) = 2W(n/2) + \Theta(n)$.

$h = \lg n$

$\Theta(1)$

#leaves $= n$
Recursion Tree

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

$h = \lg n$

#leaves = $n$

$W(n) = \Theta(n \lg n)$
Now with Caching

**Merge subroutine**

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

**Merge sort**

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

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

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

\[ #\text{leaves} = n/cM \]

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

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

\[
= \Theta((n/B) \log(n/M)).
\]

- For \( n \gg M \), we have \( \log(n/M) = \log n - \log M \approx \log n \), and thus \( W(n)/Q(n) = \Theta(B) \). A larger \( M \) does not help.
- For \( n \approx M \), we have \( \log(n/M) = \Theta(1) \), and thus \( W(n)/Q(n) = \Theta(B \log n) \).
Multiway Merging

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

$\ldots 24 17$

$\ldots 28 6$

$\ldots 14 2$

$\ldots 22 9$

$\ldots 11 7$

$R$

$\frac{n}{R}$

$\lg R$
Multiway Merging

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

- ... 24 17
- ... 28 6
- ... 14 2
- ... 22 9
- ... 11 7

- $n/R$ $\leftarrow$ $\lg R$ $\rightarrow$
Idea: Merge $R < n$ subarrays with a tournament.

Total work merging: $\Theta(n \cdot \log R)$.
Multiway Merging

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

- $\ldots \ 24 \ 17$
- $\ldots \ 28 \ 6$
- $\ldots \ 14 \ 2$
- $\ldots \ 22 \ 9$
- $7$
- $7$
- $\ldots \ 11 \ 7$

$R$

$n/R$ $\lfloor \lg R \rfloor$
Multiway Merging

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

![Diagram showing the multiway merging process with a tournament structure.]

- Nodes represent subarrays.
- The tournament structure merges subarrays in rounds.
- The diagram illustrates the merging process with sample subarrays.

Mathematical expressions:
- Tournament takes $\Theta(R)$ work to produce the first output.
- Subsequent outputs cost $\Theta(lg R)$ work per element.
- Total work merging: $\Theta(R + n lg R) = \Theta(n lg R)$. 
Idea: Merge $R < n$ subarrays with a tournament.

Total work merging: $\Theta(R + n \log R) = \Theta(n \log R)$. 

$R$
Multiway Merging

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

The tournament takes $\Theta(R)$ work to produce the first output. Subsequent outputs cost $\Theta(lg R)$ work per element. Total work merging:

$$\Theta(R + n lg R) = \Theta(n lg R).$$
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.

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

\( n/R \quad \text{and} \quad \lg R \)
Idea: Merge $R < n$ subarrays with a tournament.

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

Total work merging: $\Theta(R + n \lg R) = \Theta(n \lg R)$.
Idea: Merge $R < n$ subarrays with a tournament.

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

Total work merging: $\Theta(R + n \lg R) = \Theta(n \lg R)$.
Multiway Merging

Idea: Merge \( R < n \) subarrays with a tournament.

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

Total work merging:

\[
\Theta(R + n \log R) = \Theta(n \log R).
\]
Multiway Merging

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

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

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

$n/R$ $\leftarrow$ $\lg R$ $\rightarrow$
Multiway Merging

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

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

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

### Diagram

- \( h = \log_R n \)
- \( (n/R^2) \lg R \)
- \( (n/R) \lg R \)
- \#leaves = n
- \( \Theta(1) \)
- \( n/\lg R \)
- \( n \lg R \)

### Calculations

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

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

\[
= \Theta(n \lg n) \quad \text{same as binary merge sort}
\]
**Cache Recurrence**

**R-way merge:**

Assume that $R \leq c\mathcal{M}/\mathcal{B}$ for sufficiently small constant $c \leq 1$. Consider the $R$-way merging of contiguous arrays of total size $n$. If $R \leq c\mathcal{M}/\mathcal{B}$, the entire tournament plus 1 block from each array can fit in cache.

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

**R-way merge sort:**

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

\[ Q(n) \leq \begin{cases} 
\Theta\left(\frac{n}{B}\right) & \text{if } n \leq cM; \\
R \cdot Q\left(\frac{n}{R}\right) + \Theta\left(\frac{n}{B}\right) & \text{otherwise.}
\end{cases} \]

\[ h = \log_R\left(\frac{n}{cM}\right) \]

\[ Q(n) = \Theta\left(\left(\frac{n}{B}\right) \log_R\left(\frac{n}{M}\right)\right) \]
The number of cache misses

\[ Q(n) = \Theta\left(\frac{n}{B} \log_R \left(\frac{n}{M}\right)\right), \]

decreases as \( R \) increases. Choose \( R \) as big as possible:

\[ R = \Theta\left(\frac{M}{B}\right). \]

Thus,

\[ Q(n) = \Theta\left(\frac{n}{B} \log_{\frac{M}{B}} \left(\frac{n}{M}\right)\right) \]

\[ = \Theta\left(\frac{n}{B} \log_M \left(\frac{n}{M}\right)\right) \]

\[ = \Theta\left(\frac{n \log n}{B \log M}\right) \]

Hence, we have

\[ W(n)/Q(n) \approx \Theta(\frac{B}{\log M}). \]
Multiway versus Binary Merge Sort

We have

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

\[
Q_{\text{binary}}(n) = \Theta((n/B) \lg(n/M))
\]

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

Example

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

*k-funnel*

A *k-funnel* merges \(k^3\) items in \(k\) sorted lists, incurring at most

\[
\Theta \left( k + \left( \frac{k^3}{B} \right) \left( 1 + \log_M k \right) \right)
\]

cache misses. Thus funnelsort incurs

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

\[
= \Theta \left( 1 + \left( \frac{n}{B} \right) \left( 1 + \log_M n \right) \right)
\]

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:
$\Theta \left( k + \left( \frac{k^3}{B} \right) \left( 1 + \log_M k \right) \right)$,
tall cache $M = \Omega(B^2)$.
Other Cache-Oblivious 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 \( \Theta(1 + (n/B)(1 + \log_M n)) \)
Variant of Cooley-Tukey [CT65] using cache-oblivious matrix transpose.

LUP-Decomposition \( \Theta(1 + n^2/B + n^3/BM^{1/2}) \)
Recursive algorithm by Sivan Toledo [T97].
Ordered-File Maintenance \(O(1 + (\lg^2 n)/B)\)

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

B-Trees

INSERT/DELETE: \(O(1 + \log_B n + (\lg^2 n)/B)\)

SEARCH: \(O(1 + \log_B n)\)

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.