Lecture 15
Cache-Oblivious Algorithms

Julian Shun
SIMULATION OF HEAT DIFFUSION
Heat Diffusion

2D heat equation

Let $u(t, x, y) = \text{temperature at time } t \text{ of point } (x, y)$.

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

$\alpha$ is the \textit{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

The 1D heat equation thus reduces to

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

**Update rule**

\[
\Delta t = 1, \quad \Delta x = 1
\]

\[
u_{t+1}(x) = u_t(x) + \alpha \left( \frac{u_t(x + \Delta x) - 2u_t(x) + u_t(x - \Delta x)}{(\Delta x)^2} \right)
\]
CACHE–OBLIVIOUS STENCIL COMPUTATIONS
### Recall: Ideal–Cache Model

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

**Performance Measures**
- work $w$ (ordinary running time)
- cache misses $Q$
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 \( N > M \), then
\[
Q = \Theta(NT/B).
\]
Cache–Oblivious 3–Point Stencil

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 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(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 $\Theta(w/B)$ misses, where $w = \Theta(M)$.
- $\Theta(NT/hw)$ leaves.
- $\#\text{internal nodes} = \#\text{leaves} - 1$ do not contribute substantially to $Q$.
- $Q = \Theta(NT/hw) \cdot \Theta(w/B) = \Theta(NT/M^2) \cdot \Theta(M/B) = \Theta(NT/MB)$.
- For $d$ dimensions, $Q = \Theta(NT/M^{1/d}B)$.
Simulation: 3–Point Stencil

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

- Fully associative LRU cache
  - \( B = 4 \) points
  - \( M = 32 \) points

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

Plenty of bandwidth for 1 core
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

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

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 $M/B$ (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 $\geq 2 \cdot$ 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.
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.
Performance Comparison

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

<table>
<thead>
<tr>
<th>Code</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial looping</td>
<td>128.95s</td>
</tr>
<tr>
<td>Parallel looping</td>
<td>66.97s</td>
</tr>
<tr>
<td>Serial trapezoidal</td>
<td>66.76s</td>
</tr>
<tr>
<td>Parallel trapezoidal</td>
<td>16.86s</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)

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
• Simulation of Heat Diffusion
• Cache–Oblivious Stencil Computations
• Caching and Parallelism
• Cache–Oblivious Sorting
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 = Θ(n).

Number of cache misses = Θ(n/B).
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);
    }
}
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);
    }
}
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);
    }
}
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);
    }
}

© 2008–2018 by the MIT 6.172 Lecturers
Merge Sort

```c
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);
    }
}
```
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

```c
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);
    }
}
```
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);
    }
}

Work:
\[
W(n) = 2W(n/2) + \Theta(n) = \Theta(n \log n)
\]
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 $w(n) = 2w(n/2) + \Theta(n)$. 

\[
\begin{array}{c}
\text{n} \\
\text{n/2} & \text{n/2} & \text{n/2} \\
\text{W(n/4)} & \text{W(n/4)} & \text{W(n/4)} & \text{W(n/4)} \\
\end{array}
\]
Recursion Tree

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

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

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

$h = \lg n$

$\Theta(1)$
Recursion Tree

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

\[ h = \lg n \]

\[ n \]

\[ n/2 \]

\[ n/4 \]

\[ \Theta(1) \]
Recursion Tree

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

$h = \lg n$

#leaves = $n$

© 2008–2018 by the MIT 6.172 Lecturers
Recursion Tree

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

\[ h = \lg n \]

\[ \Theta(1) \quad \text{#leaves} = n \quad \Theta(n) \]

\[ w(n) = \Theta(n \lg n) \]
Now with Caching

Merge subroutine

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

Merge sort

\[ Q(n) = \begin{cases} 
\Theta(n / B) & \text{if } n \leq cM, \text{ constant } c \leq 1; \\
2Q(n/2) + \Theta(n/B) & \text{otherwise.}
\end{cases} \]
Cache Analysis of Merge Sort

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

Recursion tree

\[ Q(n) \]
Cache Analysis of Merge Sort

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

Recursion tree

\[
\begin{array}{c}
\text{n/B} \\
\text{Q(n/2)} \\
\text{Q(n/2)}
\end{array}
\]
Cache Analysis of Merge Sort

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

Recursion tree
Cache Analysis of Merge Sort

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

Recursion tree

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

#leaves = \[ n / cM \]
Cache Analysis of Merge Sort

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

Recursion tree

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

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

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

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

- For \( n \gg M \), we have \( \log(n/M) \approx \log n \), and thus \( w(n)/Q(n) \approx \Theta(B) \).
- For \( n \approx M \), we have \( \log(n/M) \approx \Theta(1) \), and thus \( w(n)/Q(n) \approx \Theta(B \log n) \).
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.

---

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

© 2008–2018 by the MIT 6.172 Lecturers
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

Same as binary merge sort.

\[ W(n) = \Theta((n \lg R) \log_{R} n + n) \]
\[ = \Theta((n \lg R)(\lg n)/\lg (R+n)) \]
\[ = \Theta(n \lg n) \]
Assume that we have $R < c\frac{M}{B}$ for a sufficiently small constant $c \leq 1$.

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

**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

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

Recursion tree

\[
\log_R(n/cM) \quad n/RB \quad \ldots \quad n/RB \quad \ldots \quad n/RB \quad n/B
\]

\[
\Theta(M/B) \quad \ldots \quad 0
\]

\[
\#leaves = n/cM
\]

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

\[ Q(n) = \Theta((n/B) \log_{R}(n/M)) , \]

which decreases as \( R \leq cM/B \) increases. Choosing \( R \) as big as possible yields

\[ R = \Theta(M/B) . \]

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

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

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

Example (ignoring constants)

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

$$\Theta(k + (k^3/B)(1 + \log_M k))$$

cache misses. Thus, funnelsort incurs

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

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 \( \Theta(1 + \frac{mn}{B}) \)
Straightforward recursive algorithm.

Strassen’s Algorithm \( \Theta(n + \frac{n^2}{B} + \frac{n^{\log_7 7}}{BM^{(\log_7 7)/2 - 1}}) \)
Straightforward recursive algorithm.

Fast Fourier Transform \( \Theta(1 + \frac{n}{B}(1 + \log_M n)) \)

LUP–Decomposition \( \Theta(1 + \frac{n^2}{B} + \frac{n^3}{BM^{1/2}}) \)
Recursive algorithm due to Sivan Toledo [T97].
C–O Data Structures

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:** $O(1 + \frac{(\lg^2 n)}{B})$

**SEARCH:**

**TRAVERSE:** $O(1 + \frac{\log_{B+1} n + (\lg^2 n)}{B})$

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.