Lecture 24
(Analyzing I/O and Cache Performance)

Rezaul A. Chowdhury
Department of Computer Science
SUNY Stony Brook
Fall 2012
Memory: Fast, Large & Cheap!

For efficient computation we need

- fast processors
- fast and large (but not so expensive) memory

But memory cannot be cheap, large and fast at the same time, because of

- finite signal speed
- lack of space to put enough connecting wires

A reasonable compromise is to use a memory hierarchy.
A memory hierarchy is

- almost as fast as its fastest level
- almost as large as its largest level
- inexpensive
To perform well on a memory hierarchy algorithms must have **high locality** in their memory access patterns.
The two-level I/O (or cache-aware) model [Aggarwal & Vitter, CACM’88] consists of:

- an internal memory of size $M$
- an arbitrarily large external memory partitioned into blocks of size $B$.

**I/O complexity** of an algorithm

= number of blocks transferred between these two levels

Basic I/O complexities: $\text{scan}(N) = \Theta\left(\frac{N}{B}\right)$ and $\text{sort}(N) = \Theta\left(\frac{N}{B} \log_{M/B} \frac{N}{B}\right)$

Algorithms often crucially depend on the knowledge of $M$ and $B$

⇒ algorithms do not adapt well when $M$ or $B$ changes
The ideal-cache model [ Frigo et al., FOCS’99 ] is an extension of the I/O model with the following additional feature:

- algorithms for this model are not allowed to use knowledge of $M$ and $B$.

Consequences of this extension

- algorithms can simultaneously adapt to all levels of a multi-level memory hierarchy
- algorithms become more flexible and portable

Algorithms for this model are known as cache-oblivious algorithms.
The Ideal-Cache Model: Assumptions

The model makes the following assumptions:

- Optimal offline cache replacement policy
- Exactly two levels of memory
- Automatic replacement & full associativity
The Ideal-Cache Model: Assumptions

The model makes the following assumptions:

- **Optimal offline cache replacement policy**
  - LRU & FIFO allow for a constant factor approximation of optimal
    [ Sleator & Tarjan, JACM’85 ]

- **Exactly two levels of memory**

- **Automatic replacement & full associativity**
The Ideal-Cache Model: Assumptions

The model makes the following assumptions:

- Optimal offline cache replacement policy
- Exactly two levels of memory
  - can be effectively removed by making several reasonable assumptions about the memory hierarchy [Frigo et al., FOCS’99]
- Automatic replacement & full associativity
The Ideal-Cache Model: Assumptions

The model makes the following assumptions:

- Optimal offline cache replacement policy
- Exactly two levels of memory
- Automatic replacement & full associativity
  - in practice, cache replacement is automatic (by OS or hardware)
  - fully associative LRU caches can be simulated in software with only a constant factor loss in expected performance [Frigo et al., FOCS’99]
The ideal-cache model: Assumptions

The model makes the following assumptions:

- Optimal offline cache replacement policy
- Exactly two levels of memory
- Automatic replacement & full associativity

Often makes the following assumption, too:

- $M = \Omega(B^2)$, i.e., the cache is *tall*
The Ideal-Cache Model: Assumptions

The model makes the following assumptions:

- Optimal offline cache replacement policy
- Exactly two levels of memory
- Automatic replacement & full associativity

Often makes the following assumption, too:

- \( M = \Omega(B^2) \), i.e., the cache is \textit{tall}
  - most practical caches are tall
The Ideal-Cache Model: I/O Bounds

Cache-oblivious vs. cache-aware bounds:

- Basic I/O bounds (same as the cache-aware bounds):
  - \( scan(N) = \Theta\left(\frac{N}{B}\right) \)
  - \( sort(N) = \Theta\left(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B}\right) \)

- Most cache-oblivious results match the I/O bounds of their cache-aware counterparts

- There are few exceptions; e.g., no cache-oblivious solution to the permutation problem can match cache-aware I/O bounds

[ Brodal & Fagerberg, STOC’03 ]
### Some Known Cache Aware / Oblivious Results

<table>
<thead>
<tr>
<th>Problem</th>
<th>Cache-Aware Results</th>
<th>Cache-Oblivious Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array Scanning ($scan(N)$)</td>
<td>$O\left(\frac{N}{B}\right)$</td>
<td>$O\left(\frac{N}{B}\right)$</td>
</tr>
<tr>
<td>Sorting ($sort(N)$)</td>
<td>$O\left(\frac{N \log M}{B} \cdot \frac{N}{B}\right)$</td>
<td>$O\left(\frac{N \log M}{B} \cdot \frac{N}{B}\right)$</td>
</tr>
<tr>
<td>Selection</td>
<td>$O\left(scan(N)\right)$</td>
<td>$O\left(scan(N)\right)$</td>
</tr>
<tr>
<td>B-Trees [Am] (Insert, Delete)</td>
<td>$O\left(\log_2\frac{N}{B}\right)$</td>
<td>$O\left(\log_2\frac{N}{B}\right)$</td>
</tr>
<tr>
<td>Priority Queue [Am] (Insert, Weak Delete, Delete-Min)</td>
<td>$O\left(\frac{1}{B} \log_2\frac{N}{B}\right)$</td>
<td>$O\left(\frac{1}{B} \log_2\frac{N}{B}\right)$</td>
</tr>
<tr>
<td>Matrix Multiplication</td>
<td>$O\left(\frac{N^3}{B \sqrt{M}}\right)$</td>
<td>$O\left(\frac{N^3}{B \sqrt{M}}\right)$</td>
</tr>
<tr>
<td>Sequence Alignment</td>
<td>$O\left(\frac{N^2}{BM}\right)$</td>
<td>$O\left(\frac{N^2}{BM}\right)$</td>
</tr>
<tr>
<td>Single Source Shortest Paths</td>
<td>$O\left((V + \frac{E}{B}) \cdot \log_2\frac{V}{B}\right)$</td>
<td>$O\left((V + \frac{E}{B}) \cdot \log_2\frac{V}{B}\right)$</td>
</tr>
<tr>
<td>Minimum Spanning Forest</td>
<td>$O\left(\min\left(sort(E) \log_2 \log_2 V, V + sort(E)\right)\right)$</td>
<td>$O\left(\min\left(sort(E) \log_2 \log_2 \frac{VB}{E}, V + sort(E)\right)\right)$</td>
</tr>
</tbody>
</table>

Table 1: $N = \#\text{elements}$, $V = \#\text{vertices}$, $E = \#\text{edges}$, Am = Amortized.
Matrix Multiplication
Matrix Multiplication

\[ z_{ij} = \sum_{k=1}^{n} x_{ik} y_{kj} \]

\[
\begin{array}{cccc}
  z_{11} & z_{12} & \cdots & z_{1n} \\
  z_{21} & z_{22} & \cdots & z_{2n} \\
    \vdots &    \vdots & \ddots & \vdots \\
  z_{n1} & z_{n2} & \cdots & z_{nn} \\
\end{array}
\quad = \quad
\begin{array}{cccc}
  x_{11} & x_{12} & \cdots & x_{1n} \\
  x_{21} & x_{22} & \cdots & x_{2n} \\
    \vdots &    \vdots & \ddots & \vdots \\
  x_{n1} & x_{n2} & \cdots & x_{nn} \\
\end{array}
\times
\begin{array}{cccc}
  y_{11} & y_{12} & \cdots & y_{1n} \\
  y_{21} & y_{22} & \cdots & y_{2n} \\
    \vdots &    \vdots & \ddots & \vdots \\
  y_{n1} & y_{n2} & \cdots & y_{nn} \\
\end{array}
\]

Iter-MM( X, Y, Z, n )

1. for i ← 1 to n do
2. for j ← 1 to n do
3. for k ← 1 to n do
4. \( z_{ij} ← z_{ij} + x_{ik} \times y_{kj} \)
I/O-Complexity: Iter-MM

\[ \text{Iter-MM}(X, Y, Z, n) \]

1. for \( i \leftarrow 1 \) to \( n \) do
2. \hspace{1em} for \( j \leftarrow 1 \) to \( n \) do
3. \hspace{2em} for \( k \leftarrow 1 \) to \( n \) do
4. \hspace{3em} \( z_{ij} \leftarrow z_{ij} + x_{ik} \times y_{kj} \)

Each iteration of the for loop in line 3 incurs \( O(n) \) cache misses.

I/O-complexity of Iter-MM = \( O(n^3) \)
I/O-Complexity: Iter-MM

\[ \text{Iter-MM}(X, Y, Z, n) \]

1. \textbf{for} \( i \leftarrow 1 \) \textbf{to} \( n \) \textbf{do}
2. \textbf{for} \( j \leftarrow 1 \) \textbf{to} \( n \) \textbf{do}
3. \textbf{for} \( k \leftarrow 1 \) \textbf{to} \( n \) \textbf{do}
4. \( z_{ij} \leftarrow z_{ij} + x_{ik} \times y_{kj} \)

Each iteration of the \textit{for} loop in line 3 incurs \( \Omega \left( 1 + \frac{n}{B} \right) \) cache misses.

I/O-complexity of \textit{Iter-MM} = \( O \left( n^2 \left( 1 + \frac{n}{B} \right) \right) = O \left( n^2 + \frac{n^3}{B} \right) = O \left( \frac{n^3}{B} \right) \)
Block Matrix Multiplication

\[ \text{Block-MM}(X, Y, Z, n) \]

1. \( \text{for } i \leftarrow 1 \text { to } n / s \text { do} \)
2. \( \text{for } j \leftarrow 1 \text { to } n / s \text { do} \)
3. \( \text{for } k \leftarrow 1 \text { to } n / s \text { do} \)
4. \( \text{Iter-MM}(X_{ik}, Y_{kj}, Z_{ij}, s) \)
Choose \( s = \Theta\left(\sqrt{M}\right) \), so that \( X_{ik}, Y_{kj} \) and \( Z_{ij} \) just fit into the cache.

Then line 4 incurs \( \Theta\left(s\left(1 + \frac{s}{B}\right)\right) \) cache misses.

I/O-complexity of Block-MM [assuming a tall cache, i.e., \( M = \Omega\left(B^2\right) \)]

\[
= \Theta\left(\left(\frac{n}{s}\right)^3\left(s + \frac{s^2}{B}\right)\right) = \Theta\left(\frac{n^3}{s^2} + \frac{n^3}{Bs}\right) = \Theta\left(\frac{n^3}{M} + \frac{n^3}{B\sqrt{M}}\right) = \Theta\left(\frac{n^3}{B\sqrt{M}}\right)
\]

( Optimal: Hong & Kung, STOC’81 )
Multiple Levels of Cache

\[ \text{Block-MM}(X, Y, Z, n) \]

1. \( \text{for } i \leftarrow 1 \text{ to } n/s \text{ do} \)
2. \( \text{for } j \leftarrow 1 \text{ to } n/s \text{ do} \)
3. \( \text{for } k \leftarrow 1 \text{ to } n/s \text{ do} \)
4. \( \text{Iter-MM}(X_{ik}, Y_{kj}, Z_{ij}, s) \)
Block-MM(\(X, Y, Z, n\))

1. for \(i_1 \leftarrow 1\) to \(n/s\) do
2. \hspace{0.5cm} for \(j_1 \leftarrow 1\) to \(n/s\) do
3. \hspace{1.5cm} for \(k_1 \leftarrow 1\) to \(n/s\) do
4. \hspace{2cm} for \(i_2 \leftarrow 1\) to \(s/t\) do
5. \hspace{3cm} for \(j_2 \leftarrow 1\) to \(s/t\) do
6. \hspace{4cm} for \(k_2 \leftarrow 1\) to \(s/t\) do
7. Iter-MM(\((X_{i_1k_1})_{i_2k_2}, (Y_{k_1j_1})_{k_2j_2}, (X_{i_1j_1})_{i_2j_2}, t)\)
Multiple Levels of Cache

Block-MM( X, Y, Z, n )

1. \( \text{for } i_1 \leftarrow 1 \text{ to } n / s \text{ do} \)
2. \( \text{for } j_1 \leftarrow 1 \text{ to } n / s \text{ do} \)
3. \( \text{for } k_1 \leftarrow 1 \text{ to } n / s \text{ do} \)
4. \( \text{for } i_2 \leftarrow 1 \text{ to } s / t \text{ do} \)
5. \( \text{for } j_2 \leftarrow 1 \text{ to } s / t \text{ do} \)
6. \( \text{for } k_2 \leftarrow 1 \text{ to } s / t \text{ do} \)
7. \( \text{Iter-MM}( (X_{i_1 k_1})_{i_2 k_2}, (Y_{k_1 j_1})_{k_2 j_2}, (X_{i_1 j_1})_{i_2 j_2}, t ) \)
Recursive Matrix Multiplication

\[
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
Z_{11} & Z_{12} \\
\hline
Z_{21} & Z_{22}
\end{array}
\end{array}
= \\
\begin{array}{c}
\begin{array}{c}
X_{11} & X_{12} \\
\hline
X_{21} & X_{22}
\end{array}
\end{array}
\times \\
\begin{array}{c}
\begin{array}{c}
Y_{11} & Y_{12} \\
\hline
Y_{21} & Y_{22}
\end{array}
\end{array}
= \\
\begin{array}{c}
\begin{array}{c}
X_{11} Y_{11} + X_{12} Y_{21} & X_{11} Y_{12} + X_{12} Y_{22} \\
\hline
X_{21} Y_{11} + X_{22} Y_{21} & X_{21} Y_{12} + X_{22} Y_{22}
\end{array}
\end{array}
\end{array}
\]
Recursive Matrix Multiplication

Rec-MM( X, Y, Z, n )

1. if n = 1 then Z ← Z + X · Y
2. else

3. Rec-MM( X_{11}, Y_{11}, Z_{11}, n / 2 ), Rec-MM( X_{12}, Y_{21}, Z_{11}, n / 2 )
4. Rec-MM( X_{11}, Y_{12}, Z_{12}, n / 2 ), Rec-MM( X_{12}, Y_{22}, Z_{12}, n / 2 )
5. Rec-MM( X_{21}, Y_{11}, Z_{21}, n / 2 ), Rec-MM( X_{22}, Y_{21}, Z_{21}, n / 2 )
6. Rec-MM( X_{21}, Y_{12}, Z_{22}, n / 2 ), Rec-MM( X_{22}, Y_{22}, Z_{22}, n / 2 )
I/O-Complexity: Rec-MM

Rec-MM( X, Y, Z, n )

1. if  \( n = 1 \) then  \( Z \leftarrow Z + X \cdot Y \)
2. else

3. \( \text{Rec-MM}( X_{11}, Y_{11}, Z_{11}, n/2 ) \), \( \text{Rec-MM}( X_{12}, Y_{21}, Z_{11}, n/2 ) \)
4. \( \text{Rec-MM}( X_{11}, Y_{12}, Z_{12}, n/2 ) \), \( \text{Rec-MM}( X_{12}, Y_{22}, Z_{12}, n/2 ) \)
5. \( \text{Rec-MM}( X_{21}, Y_{11}, Z_{21}, n/2 ) \), \( \text{Rec-MM}( X_{22}, Y_{21}, Z_{21}, n/2 ) \)
6. \( \text{Rec-MM}( X_{21}, Y_{12}, Z_{22}, n/2 ) \), \( \text{Rec-MM}( X_{22}, Y_{22}, Z_{22}, n/2 ) \)

I/O-complexity of \( \text{Rec-MM} \),  \( I(n) = \begin{cases} O\left(n + \frac{n^2}{B}\right), & \text{if } n^2 \leq \alpha M \\ 8I\left(\frac{n}{2}\right) + O(1), & \text{otherwise} \end{cases} \)

\[
= O\left(\frac{n^3}{M} + \frac{n^3}{B\sqrt{M}}\right) = O\left(\frac{n^3}{B\sqrt{M}}\right), \text{ when } M = \Omega\left(B^2\right)
\]

( Optimal: Hong & Kung, STOC’81 )
Searching
( Static B-Trees )
A Static Search Tree

- A perfectly balanced binary search tree
- Static: no insertions or deletions
- Height of the tree, $h = \Theta(\log_2 n)$
A perfectly balanced binary search tree

Static: no insertions or deletions

Height of the tree, \( h = \Theta(\log_2 n) \)

A search path visits \( O(h) \) nodes, and incurs \( O(h) = O(\log_2 n) \) I/Os
I/O-Efficient Static B-Trees

- Each node stores $B$ keys, and has degree $B + 1$
- Height of the tree, $h = \Theta(\log_B n)$
Each node stores $B$ keys, and has degree $B + 1$

Height of the tree, $h = \Theta(\log_B n)$

A search path visits $O(h)$ nodes, and incurs $O(h) = O(\log_B n)$ I/Os
Cache-Oblivious Static B-Trees?
van Emde Boas Layout

A binary search tree
van Emde Boas Layout

A binary search tree

If the tree contains \( n \) nodes,
each subtree contains \( \Theta\left(\frac{h}{2^2}\right) = \Theta\left(\sqrt{n}\right) \) nodes,
and \( k = \Theta\left(\sqrt{n}\right) \)
van Emde Boas Layout

If the tree contains \( n \) nodes,
each subtree contains \( \Theta \left( \frac{h}{2^2} \right) = \Theta \left( \sqrt{n} \right) \) nodes,
and \( k = \Theta \left( \sqrt{n} \right) \)
van Emde Boas Layout

Recursive Subdivision

If the tree contains $n$ nodes,
each subtree contains $\Theta\left(\frac{h}{2^2}\right) = \Theta\left(\sqrt{n}\right)$ nodes,
and $k = \Theta\left(\sqrt{n}\right)$
van Emde Boas Layout

A binary search tree

Recursive Subdivision

If the tree contains $n$ nodes,
each subtree contains $\Theta\left(\frac{h}{2^2}\right) = \Theta\left(\sqrt{n}\right)$ nodes,
and $k = \Theta\left(\sqrt{n}\right)$
van Emde Boas Layout

If the tree contains $n$ nodes,
each subtree contains $\Theta\left(\frac{h}{2^2}\right) = \Theta\left(\sqrt{n}\right)$ nodes,
and $k = \Theta\left(\sqrt{n}\right)$
I/O-Complexity of a Search

- The height of the tree is $\log n$.
- Each $\Delta$ has height between $\frac{1}{2} \log B$ and $\log B$.
- Each $\Delta$ spans at most 2 blocks of size $B$. 
The height of the tree is \( \log n \).

Each triangle has height between \( \frac{1}{2} \log B \) and \( \log B \).

Each triangle spans at most 2 blocks of size \( B \).

\[ p = \text{number of triangles visited by a search path} \]

Then \( p \geq \frac{\log n}{\log B} = \log_B n \), and \( p \leq \frac{\log n}{\frac{1}{2} \log B} = 2 \log_B n \).

The number of blocks transferred is \( \leq 2 \times 2 \log_B n = 4 \log_B n \).
Sorting

( Distribution Sort )
# Cache-Complexity of Sorting

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Cache-Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traditional (e.g., mergesort and heapsort)</td>
<td>$O(N \log N)$</td>
</tr>
<tr>
<td>Cache-Aware (e.g., external-memory versions of mergesort and distribution sort)</td>
<td>$O\left(\frac{N}{B} \log \frac{M}{B} \frac{N}{B}\right)$</td>
</tr>
<tr>
<td>Cache-Oblivious (e.g. funnelsort, cache-oblivious distribution sort and proximity mergesort)</td>
<td>$O\left(\frac{N}{B} \log \frac{M}{B} \frac{N}{B}\right)$</td>
</tr>
</tbody>
</table>
## Cache-Complexity of Sorting

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Cache-Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traditional</td>
<td>$O\left( \frac{N}{B} \log_2 N \right) \right)$</td>
</tr>
<tr>
<td>(e.g., mergesort and heapsort)</td>
<td></td>
</tr>
<tr>
<td>Cache-Aware</td>
<td>$O\left( \frac{N}{B} \log_\frac{M}{B} N \right) \right)$</td>
</tr>
<tr>
<td>(e.g., external-memory versions of mergesort and distribution sort)</td>
<td></td>
</tr>
<tr>
<td>Cache-Oblivious</td>
<td>$O\left( \frac{N}{B} \log_\frac{M}{B} N \right) \right)$</td>
</tr>
<tr>
<td>(e.g. funnelsort, cache-oblivious distribution sort and proximity mergesort)</td>
<td></td>
</tr>
</tbody>
</table>

optimal
Cache-Oblivious Distribution Sort

**Step 1:** Partition, and recursively sort partitions.

**Step 2:** Distribute partitions into buckets.

**Step 3:** Recursively sort buckets.
Step 1: Partition & Recursively Sort Partitions

\[ \sqrt{n} \text{ sub-arrays} \]

\[ \sqrt{n} \text{ elements} \]

Partitioned

Recursively Sorted

Order:
Step 2: Distribute to Buckets

Recursively Sorted

$A_1$:

$A_2$:

$A_3$:

$A_{\sqrt{n}}$:

\[\sqrt{n} \text{ elements}\]

Distributed to Buckets

$B_1$:

$B_2$:

$B_3$:

$B_q$:

- Number of buckets, $q \leq \sqrt{n}$
- Number of elements in $B_i = n_i \leq 2\sqrt{n}$
- $\max \{x \mid x \in B_i\} \leq \min \{x \mid x \in B_{i+1}\}$
Step 3: Recursively Sort Buckets

Recursively Sort Each Bucket

\[ B_1 : \text{[Diagram of sorted bucket]} \]
\[ B_2 : \text{[Diagram of sorted bucket]} \]
\[ B_3 : \text{[Diagram of sorted bucket]} \]
\[ \vdots \]
\[ B_q : \text{[Diagram of sorted bucket]} \]

Done!
Step 1: Partition, and recursively sort partitions.

Step 2: Distribute partitions into buckets.

Step 3: Recursively sort buckets.
We can take the partitions one by one, and distribute all elements of current partition to buckets

Has very poor cache performance: $\Theta\left(\sqrt{n} \times \sqrt{n}\right) = \Theta\left(n\right)$ I/Os
Recursive Distribution

Sorted Partitions

A₁
A₂
A₃

⋯

A_\sqrt{n}

B₃
B₂
B₁

⋯

B_\sqrt{n}

Distribute (i, j, m)

1. if \( m = 1 \) then copy elements from \( A_i \) to \( B_j \)
2. else
3. Distribute (\( i \), \( j \), \( m / 2 \))
4. Distribute (\( i + m / 2 \), \( j \), \( m / 2 \))
5. Distribute (\( i \), \( j + m / 2 \), \( m / 2 \))
6. Distribute (\( i + m / 2 \), \( j + m / 2 \), \( m / 2 \))

may need to split \( B_j \) to maintain \( B_j \leq 2\sqrt{n} \)
Recursive Distribution

Distribute \((i, j, m)\)

1. if \(m = 1\) then copy elements from \(A_i\) to \(B_j\)
2. else
3. Distribute \((i, j, m/2)\)
4. Distribute \((i + m/2, j, m/2)\)
5. Distribute \((i, j + m/2, m/2)\)
6. Distribute \((i + m/2, j + m/2, m/2)\)

Let \(R(m, d)\) denote the cache misses incurred by Distribute \((i, j, m)\) that copies \(d\) elements from \(m\) partitions to \(m\) buckets. Then

\[
R(m, d) = \begin{cases} 
O\left(B + \frac{d}{B}\right), & \text{if } m \leq \alpha B, \\
\sum_{1 \leq i \leq 4} R\left(\frac{m}{2}, d_i\right), & \text{otherwise, where } d = \sum_{1 \leq i \leq 4} d_i
\end{cases}
\]

\[
= O\left(B + \frac{m^2}{B} + \frac{d}{B}\right)
\]

\[
\therefore R(\sqrt{n}, n) = O\left(\frac{n}{B}\right)
\]
Recursive Distribution

Distribute \((i, j, m)\)

1. if \(m = 1\) then copy elements from \(A_i\) to \(B_j\)
2. else
3. \(Distribute\ (i, j, m/2)\)
4. \(Distribute\ (i + m/2, j, m/2)\)
5. \(Distribute\ (i, j + m/2, m/2)\)
6. \(Distribute\ (i + m/2, j + m/2, m/2)\)

Ignore the cost of splits for the time being.
Recursive Distribution

**Distribute** (i, j, m)

1. if \( m = 1 \) then copy elements from \( A_i \) to \( B_j \)
2. else
3. \( \text{Distribute} (i, j, m / 2) \)
4. \( \text{Distribute} (i + m / 2, j, m / 2) \)
5. \( \text{Distribute} (i, j + m / 2, m / 2) \)
6. \( \text{Distribute} (i + m / 2, j + m / 2, m / 2) \)

total cache misses incurred by all splits

\[ = \sqrt{n} \times O\left(\frac{\sqrt{n}}{B}\right) = O\left(\frac{n}{B}\right) \]

I/O-complexity of \( \text{Distribute} (1, 1, \sqrt{n}) \) is

\[ = R\left(\sqrt{n}, n\right) + O\left(\frac{n}{B}\right) = O\left(\frac{n}{B}\right) \]
I/O-Complexity of Distribution Sort

**Step 1:** Partition into \( \sqrt{n} \) sub-arrays containing \( \sqrt{n} \) elements each and sort the sub-arrays recursively.

**Step 2:** Distribute sub-arrays into buckets \( B_1, B_2, ..., B_q \).

**Step 3:** Recursively sort the buckets.

I/O-complexity of Distribution Sort:

\[
Q(n) = \begin{cases} 
O\left(1 + \frac{n}{B}\right), & \text{if } n \leq \alpha M \\
\sqrt{n}Q\left(\sqrt{n}\right) + \sum_{i=1}^{q} Q(n_i) + O\left(1 + \frac{n}{B}\right), & \text{otherwise} \\
= O\left(\frac{n}{B} \log_M n\right), & \text{when } M = \Omega\left(B^2\right)
\end{cases}
\]