6. Realistic Computer Models

AEA 2025

Content

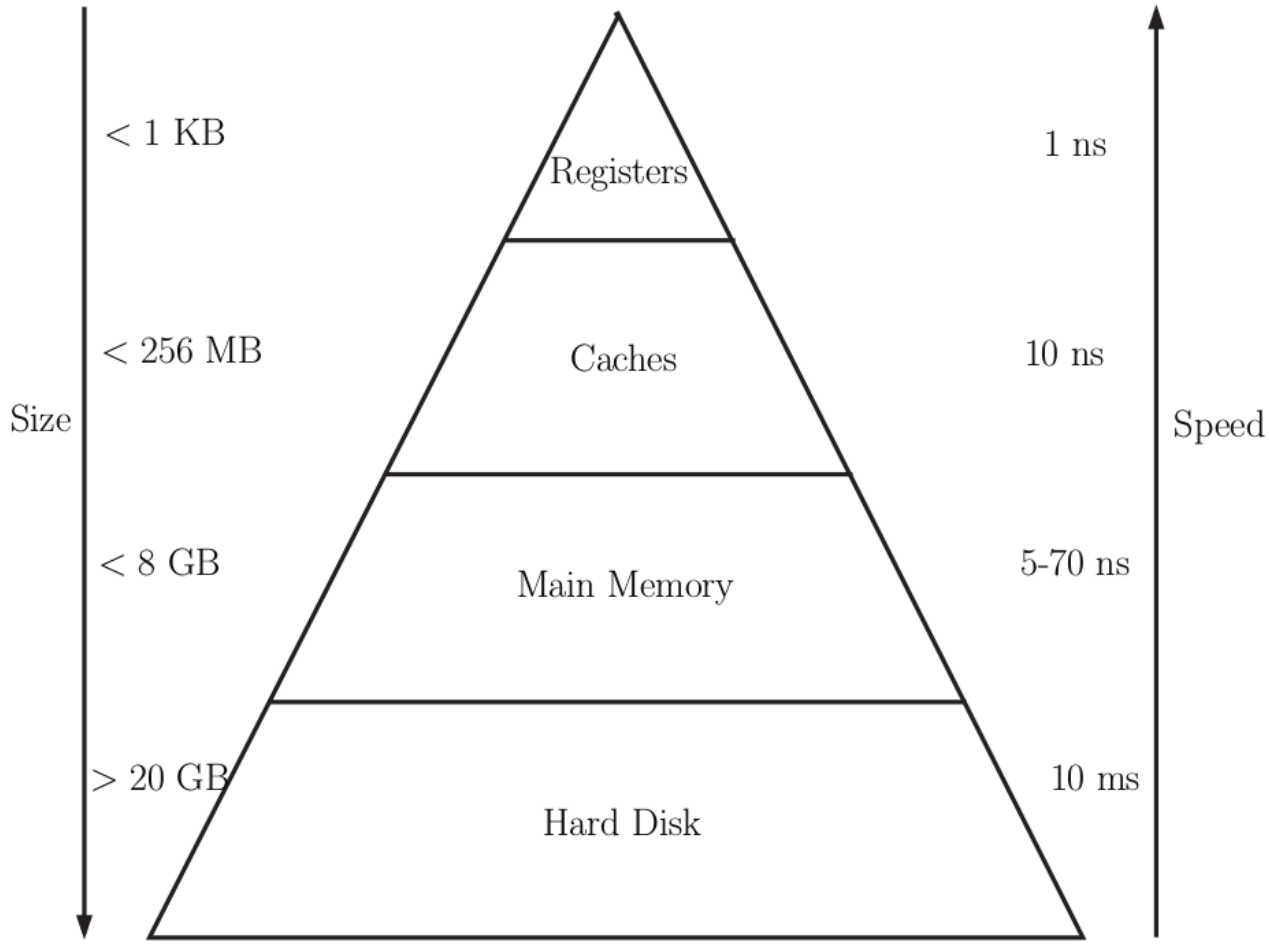
The RAM model of computation

The *Random-access machine* (RAM) model or the "von Neumann model of computation": a computing device attached to a storage device

- Instructions are executed one after another, no concurrent operations
- Every instruction takes the same amount of time
- Unbounded amount of available memory
- Memory stores words of size $O(\log n)$ bits, n the input size
- Accessing a memory location - in unit time
- For numerical and geometric algorithms: it is assumed that we represent real numbers accurately
- Exact arithmetic on real numbers - in const. time

Real Architecture

A hierarchy of storage devices with different access times and storage capacities



The RAM model

Advantages:

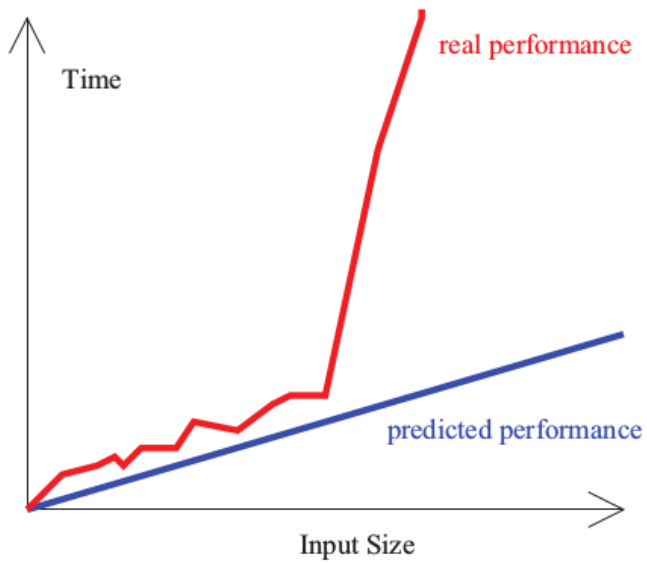
- hides the "messy" details of computer architecture from the algorithm designer
- captures the essential behavior of computers, while being simple to work with
- encapsulates well the comparative performance of algorithms
- the performance guarantees are not architecture-specific (robust)

The RAM model

Disadvantages:

- fails when the input data/intermediate data structure is too large; the dominant part: the time the algorithms spend waiting for data to be brought from hard disk to internal memory

Figure: Predicted performance of RAM model vs. its real performance

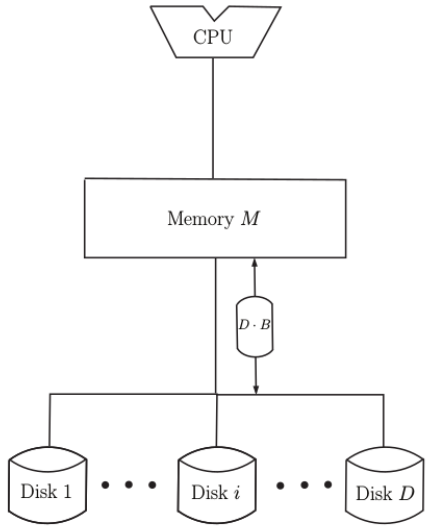


- not suited for parallel architecture

Content

External Memory Model (EM)

- a central processing unit and 2 levels of memory hierarchy: the internal memory (a limited size of M words), the external memory accessed using I/Os (move B contiguous words between the two memories)
- *the measure of performance: # I/Os*
- disk parallelism: D parallel disks; D arbitrary blocks can be accessed in parallel in an I/O
- *cache-aware algorithms*



Ideal Cache Model

- a faster level of memory (*cache*) of size M
- data transfers in chunks of B elements
- the memory is managed automatically by an optimal offline *cache-replacement strategy*; the cache is fully associative

Cache-Oblivious Model

In practice B, M need to be tuned for optimal performance.

Cache-Oblivious Model

- A **two-level memory hierarchy**, but the algorithm doesn't have any knowledge of the values M and B

- The guarantees on I/O-efficient algorithms hold on any machine with multi-level memory hierarchy.

  I/O-efficient algorithms perform well on different architectures without the need of any machine-specific optimization.

Content

Fundamental Techniques

Spatial locality: data close in address space to the currently accessed item is likely to be accessed soon.

Exploiting spatial locality:

- the data transfer in the EM/cache-oblivious model happens in terms of block of elements
- the entire block when accessed should contain as much useful information as possible
- **graph clustering and partitioning** techniques exploit "nearness"

Fundamental Techniques

Temporal locality: an instruction/data item issued/accessed currently is likely to be issued/accessed in the near future.

Exploiting temporal locality:

- use the data in the internal memory as much as possible, before writing back to the external memory
- **divide and conquer**: data is divided into small chunks to fit into the internal memory

Fundamental Techniques

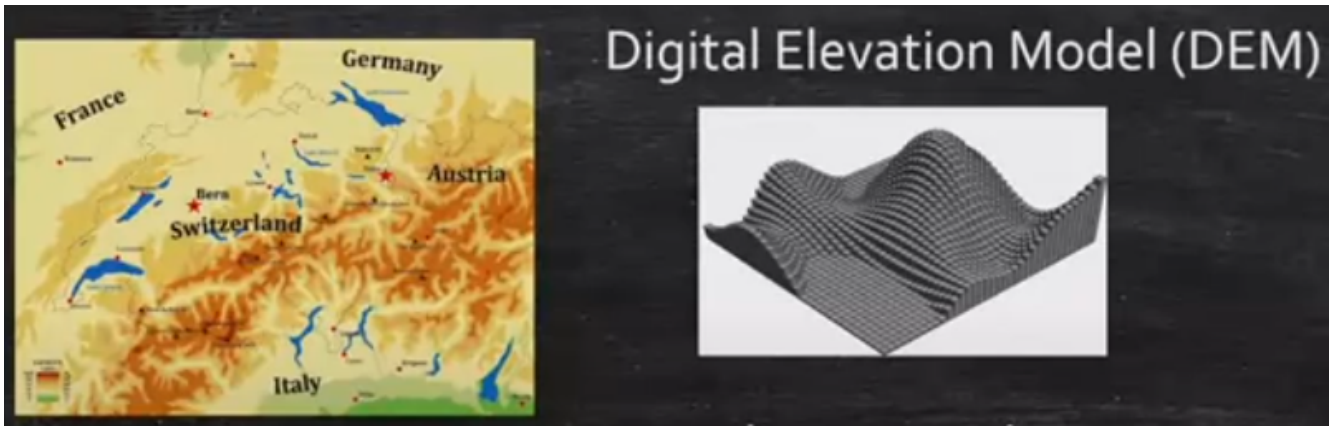
Batching: wait before issuing an operation until enough data needs to be processed s.t. the operation's cost is worthwhile.

Batching the operations:

- when performing one operation is nearly as costly as performing multiple operations of the same kind
- do lazy processing: batch a large # operations and then perform them "in parallel"
- **external priority queue**: lazy processing of *decrease-key* operations after collecting them in a batch

I/O Efficient algorithms: Motivation

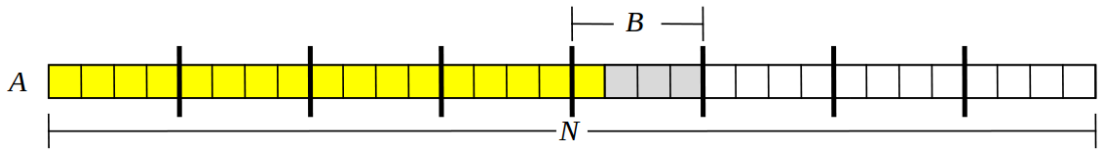
Computing the average elevation in the Alps



- 2-dimensional array $A[i, j]$: the elevation of center of cell (i, j)
- 200km x 100km, cell size 1m x 1m, 8bytes per cell $\rightarrow$ 160 GB (doesn't fit into RAM)

Sorting and Scanning

sum $= 0$;
for $i = 1$ *to* N **do**
    $sum = sum + A[i]$;



I/Os for scanning N data items: $scan(N) = \Theta(N/B)$.

How to obtain the $O(N/B)$ upper-bound for scanning?
Bring B contiguous elements in internal memory using a single I/O; do a simple memory access, rather than an expensive disk I/O.

Difference between N and N/B: $N = 256x10^6$, $B = 8000$, 1ms disk access time

- N I/Os takes $256x10^3$ s $= 4266$ min $= 71$h
- N/B I/Os takes $256/8$s $= 32$s

Sorting and Scanning

The I/O complexity of sorting n elements:
$sort(n) = \Theta(\frac{n}{B} log_{M/B} \frac{n}{B})$.

Merge-sort: in the *run formation phase*, the input data is partitioned into sorted sequences ("runs"); in the *merging phase*, the runs are merged until one sorted run remains.

- *External memory sorting algorithm* (Aggarwal&Vitter '88):
  - the 1^{st} phase produces sorted runs of M elements
  - the 2^{nd} phase does a $\frac{M}{B}$-way merge

  $O(\frac{n}{B} log_{M/B} \frac{n}{B})$ I/Os

- In the cache-oblivious setting, *funnel-sort* and *lazy funnelsort*, lead to sorting algorithms with a similar I/O complexity.

External Merge-sort

5GB of data (files on disk)
1GB RAM

1. divide in 5 files (1GB each)
   load in RAM each file, sort, write on disk
   $(F1\_sorted, \ldots, F5\_sorted)$

2. load in RAM first 150MB from each sorted file Fi_sorted
   $\rightarrow 250MB$ *free*
   5-way merge $\rightarrow$ write to disk

Sorting and Scanning

On large data sets, $scan(n) < sort(n) \ll n$, for all practical values of B, M and n.

Reading and writing data in sequential order/sorting the data is less expensive than accessing data at random.

Remove duplicates (1)

- **Read one page** in memory from the input file:

for $i = 1$ *to* $numPages(A)$ **do**
  $p \leftarrow$ **read the next page from** A;
  read pages $i + 1, \ldots,$ $numPages(A)$, using $m - 1$ pages as a read buffer;
  clean p from duplicates;
  write the remaining records of p to the end of the output (beginning of A);

$\theta(n^2)$ I/O operations, n # pages in the input file

Remove duplicates (2)

- **Read** $m-1$ **pages** in memory, use 1 page as read buffer:

  for $i = 1$ *to* $\lceil numPages(A)/(m-1) \rceil$ **do**
  $B \leftarrow$ **read the next** $m-1$ **pages from** A;
  read pages $i(m-1)+1, \ldots, numPages(A)$, using 1 page as a read buffer;
  clean B from duplicates;
  write the remaining records of B to the end of the output (beginning of A);

  $\theta(n^2/m)$ I/O operations

Content

Stacks and Queues

Represent dynamic sets of elements (LIFO/FIFO); implement them using an array of length n and pointers $\rightarrow$ **1 I/O per INSERT and DELETE** in worst-case.

Stack: keep a buffer of size $2B$ in the internal memory; it contains k most recently added elements, $k \leq 2B$.

- *INSERT*: no I/Os, except when the buffer runs full
  1 I/O to write B least recent elements to a block in EM

- *REMOVE*: no I/Os, except when the buffer is empty
  1 I/O to retrieve the block of B elements most recently written to EM

Any **sequence of B INSERT/DELETE** operations need at most **one I/O**; the **amortized cost** per operation: $1/B$ I/Os.

Stacks and Queues

Queues: 2 buffers, a read and a write buffer of size B consisting of least, respectively most recently inserted elements.

- *INSERT* to the write buffer (when full, write to EM).
- *REMOVE*: work on the read buffer; no I/O until the buffer is empty (read the appropriate EM block).

  Amortized complexity of $1/B$ I/Os per operation.

Linked Lists

An efficient implementation of *ordered lists* of elements: sequential *SEARCH*, *DELETE* and *INSERT* in arbitrary locations. Perform 1 I/O every time a pointer is followed.

An I/O-efficient implementation of **linked lists**: keep the elements in blocks; *invariant*: there are more than $\frac{2}{3}B$ elements in every pair of consecutive blocks.

- *INSERT*: 1 I/O, if the block is not full; otherwise, if any of its two neighbors has spare capacity, push an element to that block; otherwise, split the block.

- *DELETE*: check if the operation results in violating the invariant; if so, merge the two violating blocks.

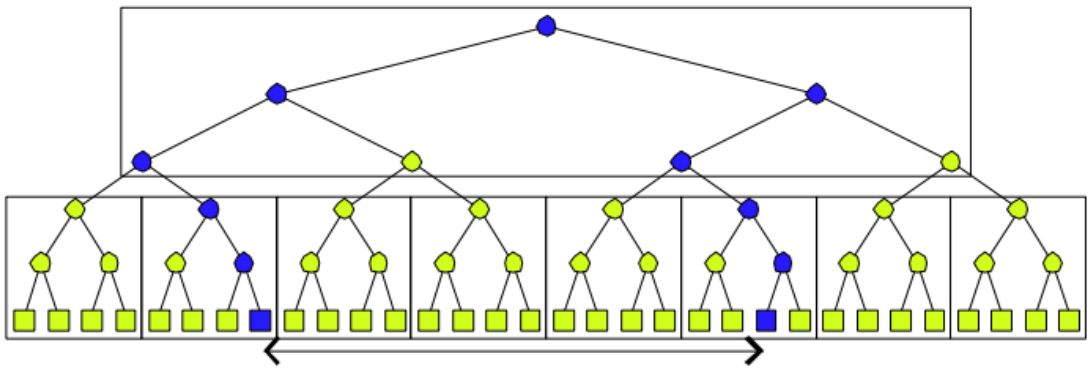
$O(1)$ I/O INSERT, DELETE, MERGE, SPLIT; $O(i/B)$ I/O access the i^{th} element.

B-tree

Storing binary trees arbitrarily in external memory: $O(log_2 N)$ I/Os per query/update.

In external memory, B-**tree** is the basis for a wide range of efficient queries on sets. A balanced BST of degree $\theta(B)$:

- n data items are stored in sorted order in $\theta(n/B)$ leaves; **each leaf:** $\theta(B)$ **elements**
- leaves are on the same level; the tree height $O(log_B n)$.



B-tree

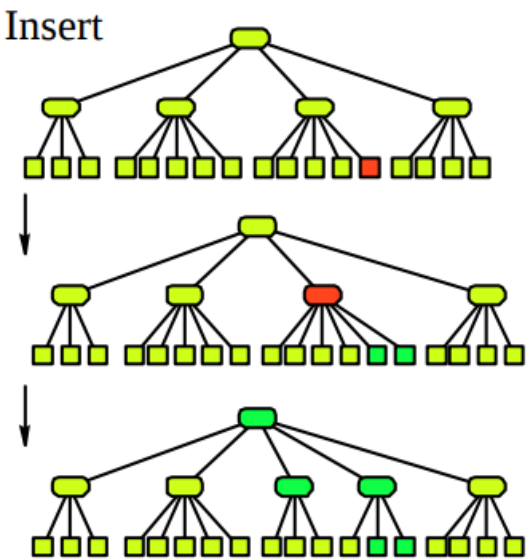
$O(log_B n)$ I/O INSERT, DELETE and SEARCH.

- *SEARCH*: traverse the tree from the root to the appropriate leaf in $O(log_B n)$ I/Os.

  One-dimensional *range queries*: $O(log_B n + T/B)$ I/Os, T the output size.

B-tree

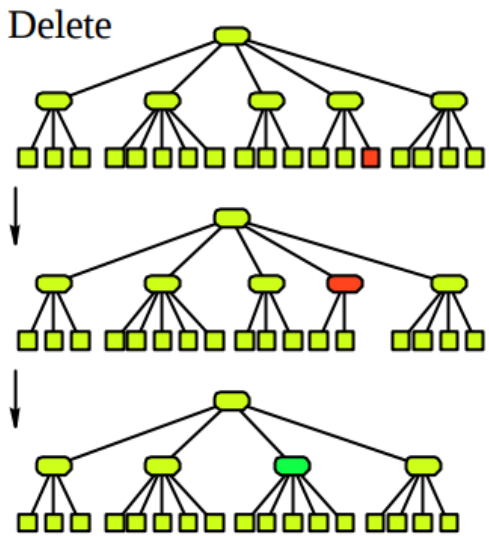
- *INSERT*
  1. Search the relevant leaf l
  2. If it is not full, insert the new element
  3. Otherwise, split l into leaves l' and l'' of approx. the same size and insert the element in the relevant leaf.
     The split results in the insertion of a new routing element in the parent of l; the need for a split may propagate up the tree (the height of the tree grows by 1).



Complexity: $O(log_B n)$ I/Os.

B-tree

- *DELETE*: search the appropriate leaf and remove the element; if too few elements in the leaf, fuse it with one of its siblings; Fuse operations may propagate up the tree (the height of the tree decreases by 1).



Complexity: $O(log_B n)$ I/Os.

Content

Cache-aware Optimization

Caches are part of the memory hierarchy between registers and the main memory.

Cache miss

- if the code does not respect the locality properties, a required data item is likely to be not in the cache
- load **contiguous** data words from memory into cache

Detecting poor cache performance

- the cache simulator *cachegrind* from Valgrind tool suite[1] performs simulations of L1 and L2 cache to determine the origins of cache misses; kcachegrind[2] - profile data visualization

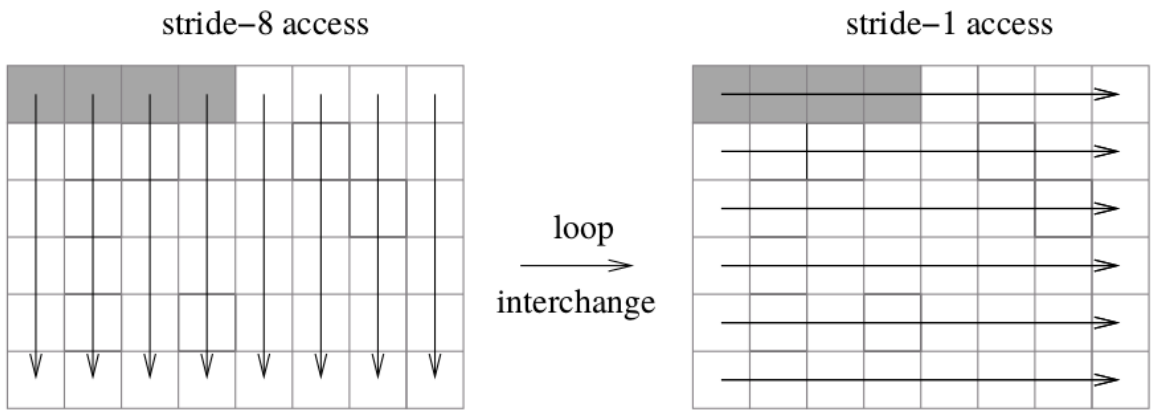
[1] http://valgrind.org/info/tools.html
[2] https://github.com/KDE/kcachegrind

Fundamental Cache-Aware Techniques

Loop Interchange and Array Transpose

- Since data is fetched blockwise into the cache, access contiguous data consecutively.

- *Cache miss*: if the **data access doesn't respect the data layout**, memory references aren't performed on contiguous data.
  Solution: access $A[i][j]$ accordant to row-major.



Stride: the distance of array elements in memory, accessed within consecutive loop iterations.

Fundamental Cache-Aware Techniques

Loop fusion: combines 2 loops (executed after another, with the same iteration space) into a loop

- the transformation is legal if no dependencies from the 1st loop to the 2nd one

Algorithm 3.2 Loop fusion

```
1: // Original code:
2: for i = 1 to n do
3:     b[i] = a[i] + 1.0;
4: end for
5: for i = 1 to n do
6:     c[i] = b[i] * 4.0;
7: end for
```

```
1: // After loop fusion:
2: for i = 1 to n do
3:     b[i] = a[i] + 1.0;
4:     c[i] = b[i] * 4.0;
5: end for
```

- a higher instruction level parallelism; reduces the loop overhead; may improve data locality

Fundamental Cache-Aware Techniques

Array merging: if the elements of a and b are typically accessed together,

instead of declaring 2 arrays with the same dimension and type `double a[n], b[n]`,

combine them to one multidimensional array (`double ab[n][2]`)

Array Padding

Inter-array padding inserts unused variables (*pads*) between 2 arrays to avoid cross interference (both arrays are mapped to different parts of the cache).

Algorithm 3.4 Inter-array padding.

```
1: // Original code:                    1: // Code after applying inter-array padding:
2: double a[1024];                     2: double a[1024];
3: double b[1024];                     3: double pad[x];
4: for i = 1 to 1023 do                4: double b[1024];
5:     sum+ = a[i] * b[i];             5: for i = 1 to 1023 do
6: end for                             6:     sum+ = a[i] * b[i];
                                       7: end for
```

Cache-Aware Numerical Linear Algebra

Computational kernels in linear algebra that achieve a high cache performance:

- *Basic Linear Algebra Subprograms (BLAS)*[1] - basic vector and matrix operations
- *Linear Algebra Package (LAPACK)*[2] - solvers for linear equations, linear least-square and eigenvalue problems, etc.
- *Automatically Tuned Linear Algebra Software (ATLAS)*[3] - determines the hardware parameters during its installation and adapts its parameters accordingly to achieve a high cache efficiency on a variety of platforms

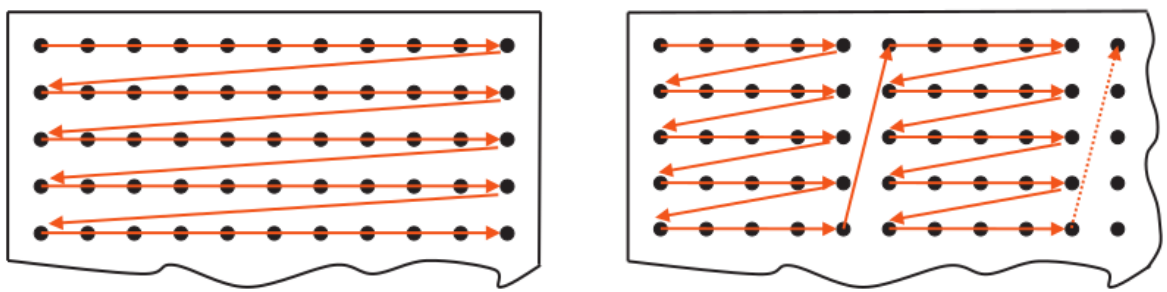
[1] http://www.netlib.org/blas/
[2] http://www.netlib.org/lapack/
[3] http://math-atlas.sourceforge.net/

Cache-Aware Numerical Linear Algebra

Loop blocking: for the improvement of data access and therefore temporal locality in loop nests.

Changes the way in which the matrix and the corresponding vector elements are accessed:

- rather than iterating over one row after the other, the matrix is **divided into small block matrices** that fit into the cache
- new inner loops that iterate within the blocks are introduced into the original one

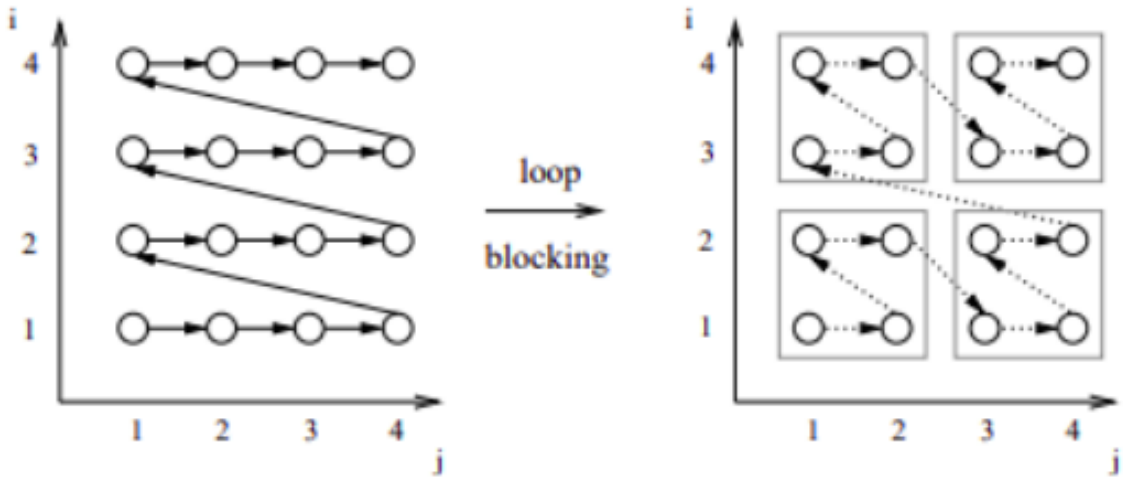


Loop blocking

Algorithm 3.3 Loop blocking for matrix transposition

```
1: // Original code:
2: for i = 1 to n do
3:    for j = 1 to n do
4:       a[i, j] = b[j, i];
5:    end for
6: end for
```

```
1: // Loop blocked code:
2: for ii = 1 to n by B do
3:    for jj = 1 to n by B do
4:       for i = ii to min(ii + B − 1, n) do
5:          for j = jj to min(jj + B − 1, n) do
6:             a[i, j] = b[j, i];
7:          end for
8:       end for
9:    end for
10: end for
```



The outer loop traverses the original iteration space with an increment equal to the size B of the block which is traversed by the inner loop.

Matrix transposition

Matrix $A[m, m]$, $A^T[i, j] = A[j, i]$, for all i, j
Goal: convert A to A^T

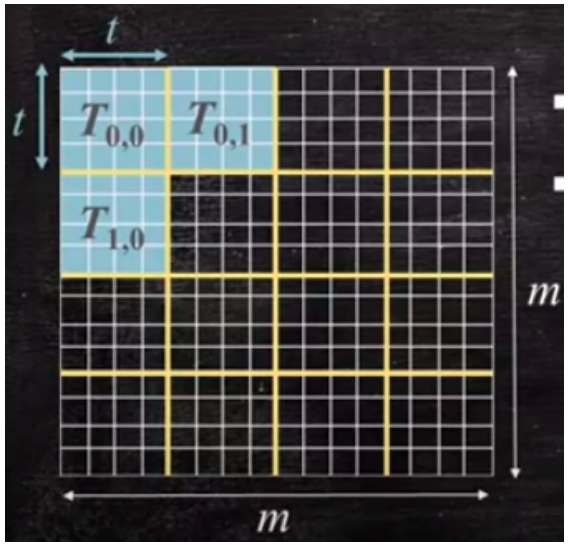
- size of the matrix $n = m^2$
- matrix is stored in row-major order
- cannot keep one column in memory

for $i = 1$ *to* $m - 1$ **do**
  for $j = 0$ *to* $i - 1$ **do**
    swap $A[i, j]$ and $A[j, i]$;

- for $A[i, j]$: blocks are read at most once, $O(n/B)$ I/Os
- for $A[j, i]$: blocks are read many times, $O(n)$ I/Os

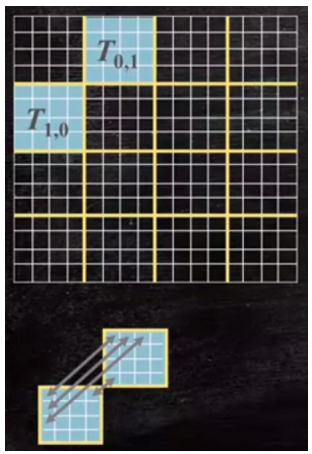
A cache-aware algorithm for matrix transposition

- partition matrix into tiles $T_{i,j}$ of size $t \times t$
- 2 tiles should fit into main memory
- # tiles: $\lceil m/t \rceil \times \lceil m/t \rceil = O(n/t^2)$



A cache-aware algorithm for matrix transposition[1]

$t = \ldots;$
for $i = 0$ *to* $\lceil m/t \rceil$ **do**
    for $j = 0$ *to* i **do**
        read tiles $T_{i,j}$ and $T_{j,i}$;
        swap elements in tiles;
        write tiles back to disk;



$O(n/B)$ I/Os
Pick tile size s.t. 2 tiles fit into internal memory: $t \approx \sqrt{M/2}$.

[1] I/O-efficient algorithms course

Content

Cache-Oblivious Algorithms

The portability of cache-aware optimization methods from one machine to another is often difficult

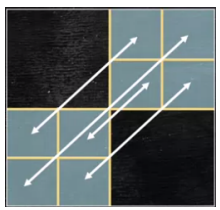
- interested in algorithms that **don't require specific hardware parameters**

Idea:

- use a **recursive** algorithm: recursion will reach a point where the subproblem fits into the internal memory

a. Cache-oblivious matrix transposition[2]

- To swap two matrices: swap 4 pairs of submatrices of 1/2 size $\rightarrow$ 4 recursive calls
- Base case: submatrix of 1×1



Exception: the cells on the diagonal (3 recursive calls).

[2] I/O-efficient algorithms

Cache-oblivious matrix transposition

//swap $A[i_1...i_2, j_1...j_2]$ with $A[j_1...j_2, i_1...i_2]$;
if $i_1 = i_2$ *or* $j_1 = j_2$ **then**
    swap $A[i_1...i_2, j_1...j_2]$ with $A[j_1...j_2, i_1...i_2]$;
else
    $i_{mid} \leftarrow \lfloor \frac{i_1+i_2}{2} \rfloor, j_{mid} \leftarrow \lfloor \frac{j_1+j_2}{2} \rfloor$;
    Cache-obliv-transpose$(A, i_1, i_{mid}, j_1, j_{mid})$;
    Cache-obliv-transpose$(A, i_{mid+1}, i_2, j_1, j_{mid})$;
    Cache-obliv-transpose$(A, i_{mid+1}, i_2, j_{mid+1}, j_2)$;
    if $i_1 \geq j_{mid} + 1$ **then**
        Cache-obliv-transpose$(A, i_1, i_{mid}, j_{mid+1}, j_2)$;

Algorithm 1: Cache-obliv-transpose(A, i_1, i_2, j_1, j_2)

$O(n/B)I/Os$

b. Cache-Oblivious matrix multiplication

A, B $n \times n$ matrices stored in the memory. Compute the matrix product $C := AB$.

for $i = 1$ *to* n **do**
    for $j = 1$ *to* n **do**
        $C[i, j] \leftarrow 0.0$;
        for $k = 1$ *to* n **do**
            $C[i, j] = C[i, j] + A[i, k] \cdot B[k, j]$;

Algorithm 2: Naive matrix multiplication

Two arrays of length n are accessed at the same time, one with stride 1, another with stride n.
Apply *loop blocking*: cached entries of all matrices can be reused.

Cache-oblivious matrix multiplication algorithm[3]

A cache-oblivious blocking of the main loop can be achieved by **recursive** block building.

Guide this recursion by *space-filling curves*

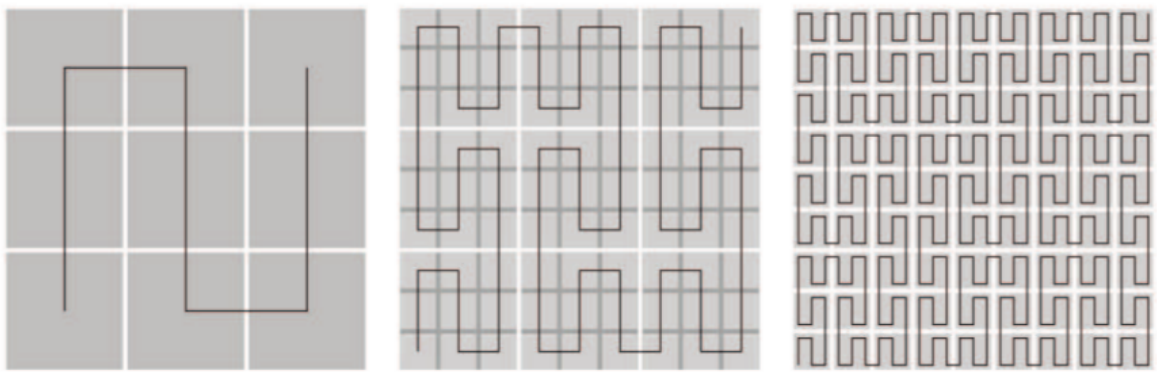
- a method based on the *Peano curve* to increase spatial and temporal locality

[3] Cache Oblivious Matrix Operations Using Peano Curves

Cache-oblivious matrix multiplication algorithm

Cache-efficient computation: processing of matrix blocks.

- Matrices are subdivided recursively into $n_x \times n_y$ block matrices, until small enough (some fraction of the cache size).
- Any block size $n_x \times n_y$, n_x, n_y odd.
- 9 recursive blocks; the recursion stops with submatrices of 3x3.

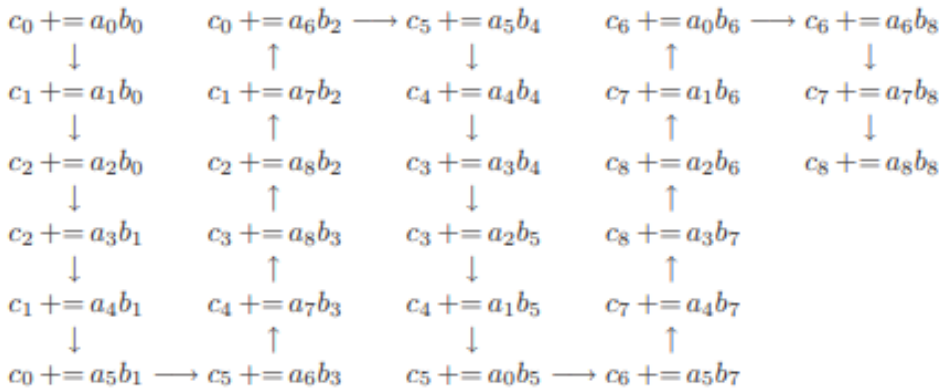


Cache-oblivious matrix multiplication algorithm

Each submatrix of 3 x 3 is stored in a *Peano-like ordering*:

$$\begin{pmatrix} a_0 & a_5 & a_6 \\ a_1 & a_4 & a_7 \\ a_2 & a_3 & a_8 \end{pmatrix} \cdot \begin{pmatrix} b_0 & b_5 & b_6 \\ b_1 & b_4 & b_7 \\ b_2 & b_3 & b_8 \end{pmatrix} = \begin{pmatrix} c_0 & c_5 & c_6 \\ c_1 & c_4 & c_7 \\ c_2 & c_3 & c_8 \end{pmatrix}$$

Find an operation order where consecutive triples differ by no more than 1 in each element (for spatial & temporal locality).



Similarly for the outer iteration: the blocks are also accessed in the Peano order due to the recursive construction.

c. Funnel sort: a cache-oblivious version of Mergesort[4]

To sort a (contiguous) array of n elements:

1. split the input into $n^{1/3}$ contiguous arrays, of size $n^{2/3}$
2. sort the arrays recursively
3. merge the $n^{1/3}$ sorted sequences using a $n^{1/3}$-merger.

A k-**merger** inputs k sorted sequences and merges them (recursively).

Invariant: Each invocation of a k-merger outputs the next k^3 elements of the sorted sequence, obtained by merging the k input sequences.

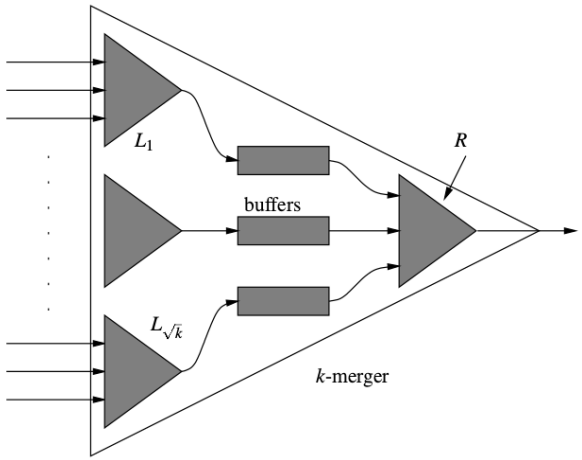
[4] Cache-Oblivious Algorithms and Data Structures

Funnel sort

A k-**merger** is built recursively out of $\sqrt{k}$-mergers:

- k inputs are partitioned into $\sqrt{k}$ sets of $\sqrt{k}$ elements (the input to the $\sqrt{k}$ $\sqrt{k}$-mergers $L_1, \ldots L_{\sqrt{k}}$).
- The outputs of the mergers are connected to the inputs of $\sqrt{k}$ *buffers* (a queue that can hold $2k^{3/2}$ elements).
- The outputs of the buffers are connected to the $\sqrt{k}$ inputs of the$\sqrt{k}$-merger R.

The base case of the recursion is a k-merger with $k = 2$, which produces $k^3 = 8$ elements.

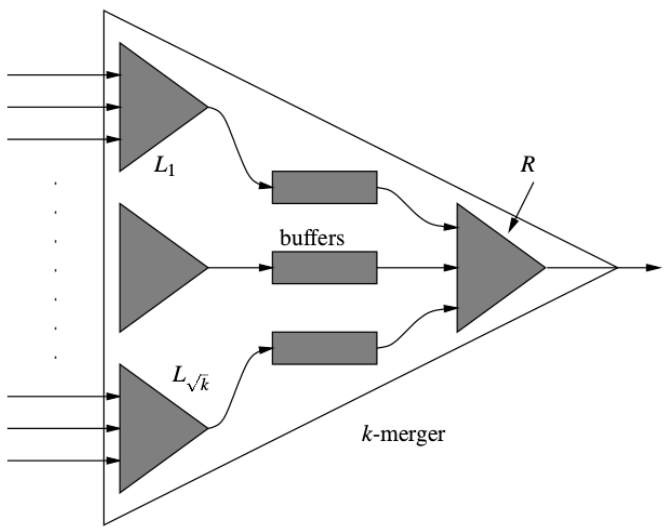


Funnel sort

A k-merger operates recursively: **the k-merger invokes R for $k^{3/2}$ times to output k^3 elements**.

- before each invocation, the k-merger fills all buffers that are less than half full (contain less than $k^{3/2}$ elements)

- to fill buffer i, the algorithm invokes the merger L_i once; since L_i outputs $k^{3/2}$ elements, the buffer contains at least $k^{3/2}$ elements after L_i finishes



Funnel sort

A careful implementation of a cache-oblivious *lazy funnel sort* outperforms library implementations of quicksort on uniformly distributed data

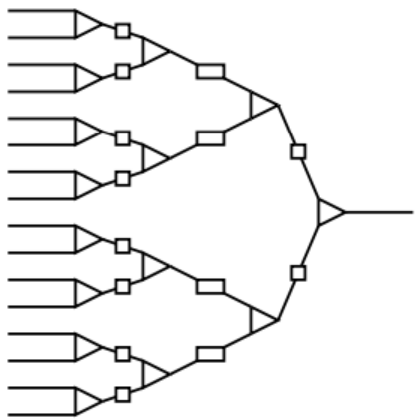
- for the largest instances in the RAM, it outperforms `std::sort` from the STL library, GCC 3.2, by 10-40%

Lazy Funnel sort

Modification: a buffer is filled when it runs empty.
A tree of binary mergers with buffers on the edges.

A *k-merger*: a perfectly balanced binary tree with k leaves

- a leaf contains a sorted input stream; an internal node contains a binary merger
- output of the root: output stream of the k-merger
- the edge between two internal nodes contains a buffer



Bibliography

- *Algorithm Engineering: Bridging the Gap Between Algorithm Theory and Practice* - Chapter 5. Realistic Computer Models