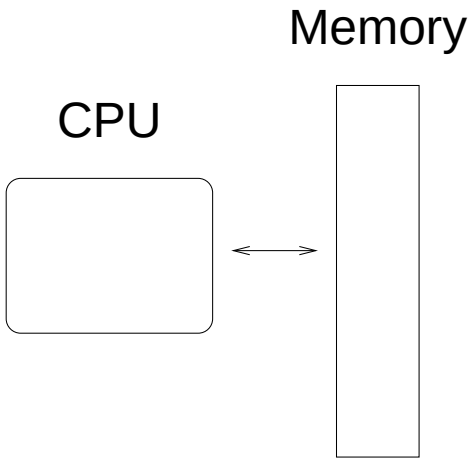
DM207
I/O-Efficient Algorithms and Data Structures

Fall 2009

Rolf Fagerberg

Analysis of algorithms (DM507, DM508,...)

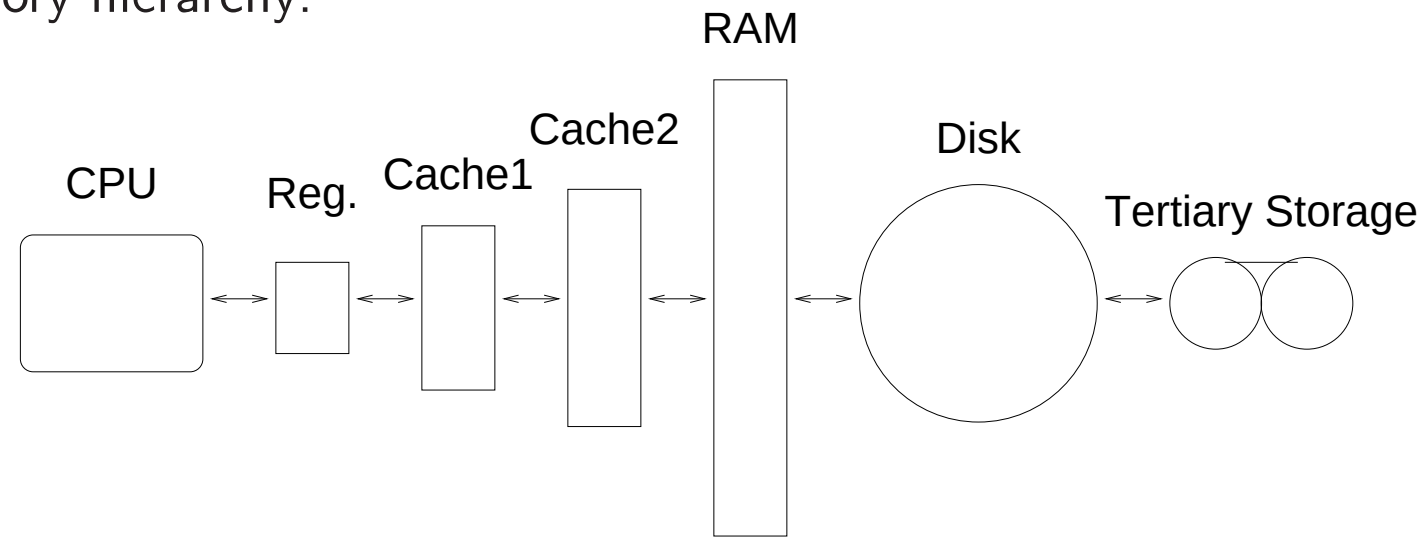
The standard model:



- ADD: 1 unit of time
- MULT: 1 unit of time
- BRANCH: 1 unit of time
- MEMACCESS: 1 unit of time

Reality

Memory hierarchy:



	Access time	*Volume*
Registers	1 cycle	1 Kb
Cache	5–10 cycles	1 Mb
RAM	50–100 cycles	1 Gb
Disk	30,000,000 cycles	1 Tb

CPU speed has improved faster than RAM access time and much faster than disk access time

Reality

Many real-life problems of **Terabyte** and even **Petabyte** size:

- Meteorology
- Geology, Geography, GIS
- Scientific computing
- Financial sector, banks
- Phone companies
- WWW

The Data Explosion Problem

From Danish Center for Scientific Computing:

> A major challenge... is the data explosion problem: Collecting scientific data has progressed from the time when a scientist scribbled notes on paper to automated data collection, using many inexpensive sensors, which gives rise to a huge explosion in the amount of collected data. Examples are numerous and include meteorological data, seismic data, genetic data, astronomical data, medical imaging, etc. To this must be added the ever growing information archives in the arts and social sciences, which require further large storage and processing facilities.

I/O bottleneck

I/O is the bottleneck

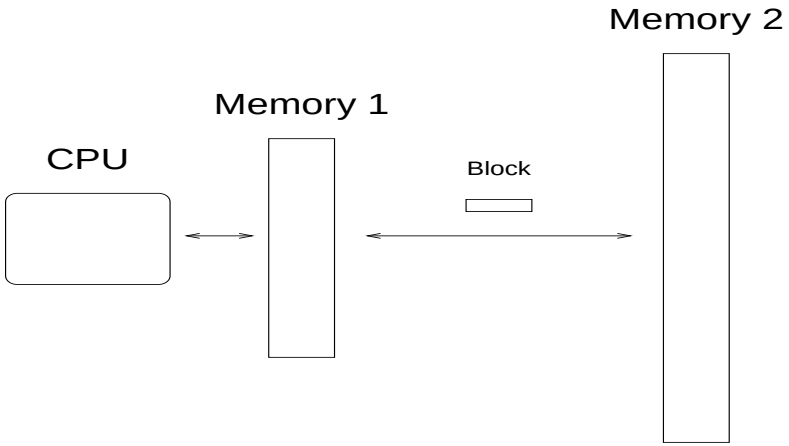
$\Downarrow$

I/O should be optimized (not instruction count)

We need new models for this.

Analysis of algorithms

New **I/O-model:**



Parameters:

Aggarwal, Vitter, 1988

N = no. of elements in problem.

M = no. of elements that fits in RAM.

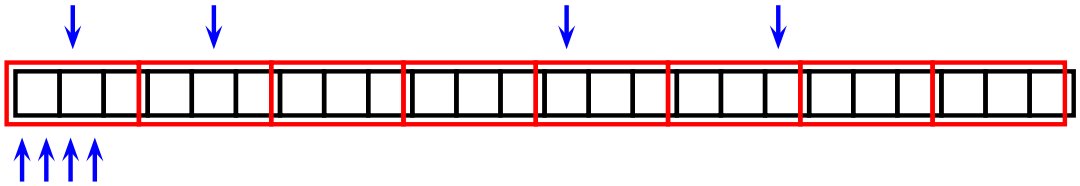
B = no. of elements in a block on disk.

Cost: Number of I/O's (block transfers) between Memory 1 and Memory 2.

Generic Example

Consider two $O(N)$ algorithms:

1. Memory accessed randomly $\Rightarrow$ page fault at each memory access.
2. Memory accessed sequentially $\Rightarrow$ page fault every B memory accesses.



$O(N)$ I/Os *vs.* $O(N/B)$ I/Os

Typically for disk: $B = 10^3 - 10^5$.

(Note: 10^5 minutes $=$ 70 days, 10^5 days $=$ 274 years.)

Specific Examples

Three $O(N \log N)$ CPU-time sorting algorithms:

	Worstcase	Inplace
QuickSort		+
MergeSort	+	
HeapSort	+	+

Specific Examples

But:

QuickSort, MergeSort $\sim$ sequential access

HeapSort $\sim$ random access

So in terms of I/Os:

QuickSort:	$O(N \log_2(N/M)/B)$
MergeSort:	$O(N \log_2(N/M)/B)$
HeapSort:	$O(N \log_2(N/M))$

Goals of Course

- Goal I: Learn principles for designing I/O-efficient algorithms.
- Goal II: Get hands-on experience via projects.
- Goal III: See lots of beautiful algorithmic ideas.
- Goal IV: Sharpen analytical skills.

Course Contents

- The I/O model(s).
- Algorithms, data structures, and lower bounds for basic problems:
  - Permuting
  - Sorting
  - Searching (search trees, priority queues)
- I/O efficient algorithms and data structures for problems from
  - computational geometry,
  - strings,
  - graphs.

Course Formalities

Lectures:

- Theoretical (in the style of DM507, DM508, DM206, DM802,...).
- New stuff: 1995-2009.
- Aim: Principles and methods.

Project work:

- Several small/medium projects (3 ECTS in total).
- Aim: Hands-on (programming), thinking (theory).

Course Formalities

Literature:

- Based on lecture notes and articles.

Prerequisites:

- DM507, DM508 (and a BA-degree).

Duration:

- 1st and 2nd quarter.

Credits:

- 10 ECTS (including project).

Exam:

- The projects (pass/fail), oral exam (7-scale).

Statement of Aims

After the course, the participant is expected to be able to:

- Describe general methods and results relevant for developing I/O-efficient algorithms and data structures, as covered in the course.
- Give proofs of correctness and complexity of algorithms and data structures covered in the course.
- Formulate the above in precise language and notation.
- Implement algorithms and data structures from the course.
- Do experiments on these implementations and reflect on the results achieved.
- Describe the implementation and experimental work done in clear and precise language, and in a structured fashion.

Basic Results in the I/O-Model

Scanning: $\Theta(\frac{N}{B})$	Permuting: $\Theta(\min\{N, \frac{N}{B}\log_{\frac{M}{B}}(\frac{N}{M}))\})$
Sorting: $\Theta(\frac{N}{B}\log_{\frac{M}{B}}(\frac{N}{M}))$	Searching: $\Theta(\log_B(N))$

Notable differences from standard internal model:

- Linear time $= O(\frac{N}{B}) \neq O(N)$
- Sorting very close to linear time for normal parameters
- Sorting $=$ permuting for normal parameters
- Permuting $>$ linear time
- Sorting using search trees is far from optimal (search $>>$ sort/N).

Basic Results in the I/O-Model

Scanning: $\Theta(\frac{N}{B})$	Permuting: $\Theta(\min\{N, \frac{N}{B}\log_{\frac{M}{B}}(\frac{N}{M}))\})$
Sorting: $\Theta(\frac{N}{B}\log_{\frac{M}{B}}(\frac{N}{M}))$	Searching: $\Theta(\log_B(N))$

Scanning is I/O-efficient ($O(1/B)$ per operation). Hence, a few algorithms and data structures (selection, stacks, queues) are I/O-efficient ($O(1/B)$ per operation) out of the box.

Most other algorithmic tasks need rethinking and new ideas.

Stacks and Queues

With constant number of blocks in RAM:

$O(1/B)$ I/Os per Push/Pop operation.

$O(1/B)$ I/Os per Dequeue/Enqueue operation.

Selection

Recall the problem: For (unsorted) set of elements, find the kth largest.

Classic linear time (CPU-wise) algorithm:

1. Split into groups of 5 elements, select median of each.
2. Recursively find the median of this set of selected elements.
3. Split entire input into two parts using this element as pivot.
4. Recursively select in relevant part.

Step 1 and 3 are scans, step 2 recurse on $N/5$ elements, and none of the lists made in step 3 are larger than around $7N/10$ elements.

As (N/B) is the solution to $T(N) = O(N/B) + T(N/5) + T(7N/10)$, $T(M) = O(M/B)$, the algorithm is also linear in terms of I/Os.

(This recurrence holds assuming the memory touched by a recursive call (including all sub-calls) is contiguous, and that LRU caching is done.)