Entwurf, Analyse und Umsetzung von Algorithmen
Cache Efficiency, Divide and Conquer

Albert-Ludwigs-Universität Freiburg

Prof. Dr. Rolf Backofen
Bioinformatics Group / Department of Computer Science
Entwurf, Analyse und Umsetzung von Algorithmen

iems
intelligente eingebettete
mikrosysteme

UNI
FREIBURG

Structure

Background:

- Up to now we always counted the number of operations
- Assuming this is a good measure for the runtime of an algorithm/tool
- Today we will see examples where this is not suitable

Cache Efficiency
Introduction

Example:

- We sum up all elements of an array *a* of size *n* in ...
    - natural order:

$$\mathrm{sum}(a) = a[1] + a[2] + \cdots + a[n]$$

    - random order:

$$\mathrm{sum}(a) = a[21] + a[5] + \cdots + a[8]$$

Cache Efficiency
Linear Order - Python

Python:

```python
def init(size):
    """Creates the dataset."""

    # use system time as seed
    random.seed(None)

    # set linear order as accessor
    order = [a for a in range(0, size)]

    # init array with random data
    data = [random.random() for a in order]

    return (order, data)
```

Python:

```python
def run(param):
    """Processes the dataset."""

    # unpack data
    (order, data) = param

    # init the sum value
    s = 0

    for index in order:
        s += data[index]

    return s
```
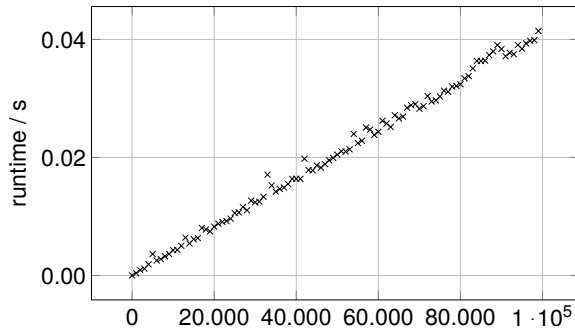
Figure: summing elements in linear order

```python
def init(size):
    """Creates a randomly ordered dataset."""

    # use system time as seed
    random.seed(None)

    # set random order as accessor
    order = [a for a in range(0, size)]
    random.shuffle(order)

    # init array with random data
    data = [random.random() for a in order]

    return (order, data)
```
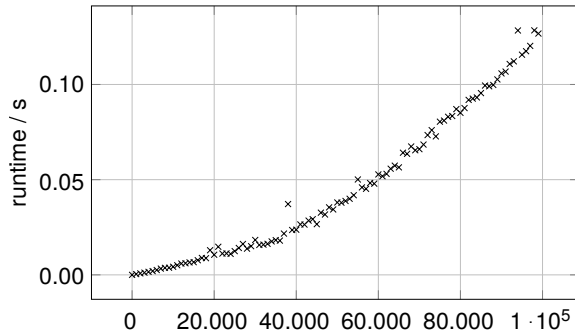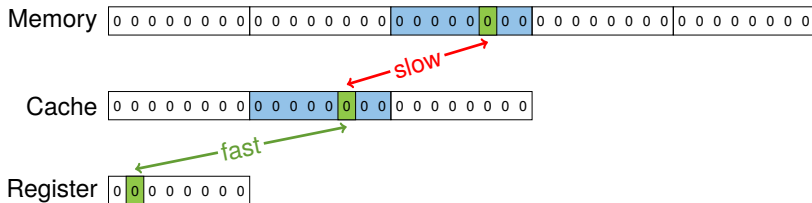
Figure: summing elements in random order

Conclusion:

- The number of operations is identical for both algorithms
- Accessing elements in random order takes a lot longer (factor 10)
- The costs in terms of memory access are very different

Principle / organization:

- Accessing one byte of the main memory takes $\approx 100\,ns$
- Accessing one byte of (L1-)cache takes $\approx 1\,ns$
- Accessing one or more byte/s of main memory loads a whole block $\approx 100\,B$ into the cache
- As long as this block is in the cache, it is not neccessary to access the memory for bytes of this block
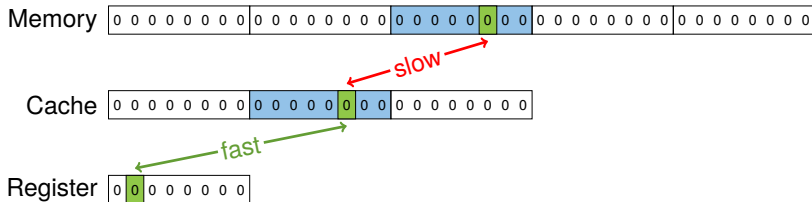
Cache organization:

- The (L1-)cache can hold multiple memory blocks
    - Cache lines ≈ 100 kB
- If the capacity is reached unused blocks are discarded
    - Least recently used (LRU)
    - Least frequently used (LFU)
    - First in first out (FIFO)
- Details of discarding not discussed today

UNI
FREIBURG

B bytes

Memory `0 0 0 0 0 0 0 0` `0 0 0 0 0 0 0 0` `0 0 0 0 0` `0` `0 0` `0 0 0 0 0 0 0 0` `0 0 0 0 0 0 0 0`

Cache `0 0 0 0 0 0 0 0` `0 0 0 0 0` `0` `0 0` `0 0 0 0 0 0 0 0`

M bytes = *M*/*B* blocks
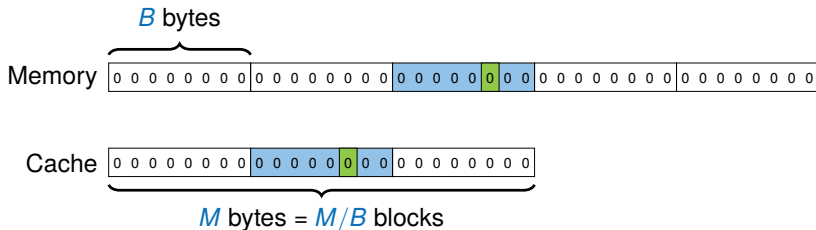
Terminology:

- The system consists of slow and fast memory
- The slow memory is divided in blocks of size *B*
- The fast cache has size *M* an can store *M*/*B* blocks
- If data is not in fast memory, the corresponding block is loaded into the cache

Cache Efficiency
Block Operations

B bytes

Memory `0 0 0 0 0 0 0 0` `0 0 0 0 0 0 0 0` `0 0 0 0 0` `0` `0 0` `0 0 0 0 0 0 0 0` `0 0 0 0 0 0 0 0`

Cache `0 0 0 0 0 0 0 0` `0 0 0 0 0` `0` `0 0` `0 0 0 0 0 0 0 0`

M bytes = *M*/*B* blocks
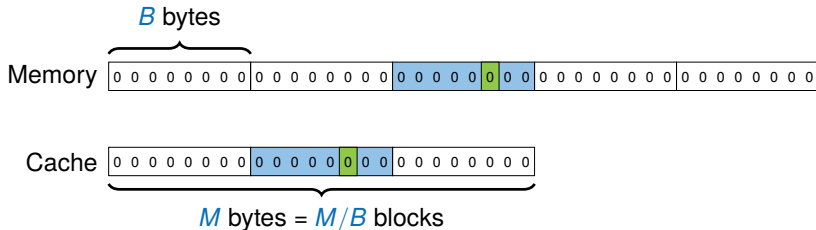
Terminology:

- The program defines which blocks are held in the cache
- We use the number of block operations as runtime estimation
- We ignore runtime costs of cache access / management

Cache Efficiency
Block Operations

good locality

Memory `0 0 0 0 0 0 0 0` `0 0 0 0 0 0 0 0` `0` `0` `0` `0` `0` `0` `0` `0` `0 0 0 0 0 0 0 0` `0 0 0 0 0 0 0 0`

bad locality

Memory `0 0` `0` `0 0 0 0 0` `0 0 0 0 0` `0` `0 0` `0 0` `0` `0 0 0 0 0` `0 0 0 0 0` `0` `0 0` `0 0` `0` `0 0 0 0 0`

Figure: comparison good / bad locality

Accessing the cache B times:

- Best case: 1 block operation $\rightarrow$ good locality
- Worst case: B block operations $\rightarrow$ bad locality

Additional factors:

- The following settings change only a small constant factor in number of block operations
  - Partinining of the slow memory into blocks
  - Regardless of the block size: 1 bytes or 4 bytes or 8 bytes

Note:

- If the input size is smaller than M we load the complete data chunk directly into the cache
- Cache handling is only interesting when the input size is greater than M

UNI
FREIBURG

Typical values: (Intel© i7-4770 Haswell, WD© Blue 2 TB)

- CPU L1 Cache: $B = 64\,B$, $M = 4 \times (32\,kB + 32\,kB)$
- CPU L2 Cache: $B = 64\,B$, $M = 4 \times 256\,kB$
- CPU L3 Cache: $B = 64\,B$, $M = 8\,MB$
- Disk Cache: $B = 64\,kB$, $M = 64\,MB$
    - Many operating systems use free system memory as disk cache

Terminology:

- Block loads on CPU cache are called cache misses
- Block operations on disk cache are called IOs
  (input / output operations)
- These also fall under the term cache efficiency or
  IO efficiency

Example 1 - Linear order:

- We sum up all elements in natural order

$$\mathrm{sum}(a) = a[1] + a[2] + \cdots + a[n]$$

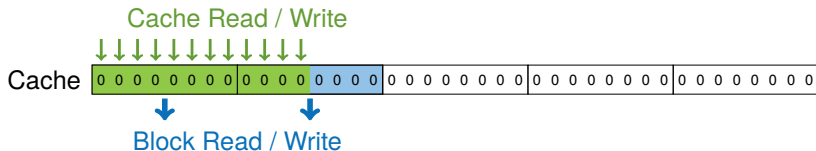- The number of block operations is $\mathrm{ceil}\left(\frac{n}{B}\right)$

Cache Read / Write
↓↓↓↓↓↓↓↓↓↓↓↓

Cache | 0 0 0 0 0 0 0 0 | 0 0 0 0 | 0 0 0 0 | 0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0 | 0 0 0 0 0 0 0 0 |
↓ ↓

Block Read / Write

Figure: good locality of sum operation

Cache Efficiency
Block Operations - Random Order

Example 2 - Random order:

- We sum up all elements in random order

$$\mathrm{sum}(a) = a[21] + a[5] + \cdots + a[8]$$

- The number of block operations is *n* in the worst case
- This leads to a runtime factor difference of *B*

Cache Read / Write

Block Read / Write

Figure: bad locality of sum operation

Generally the factor is substantially $< B$

- Even with a random order we access 4 neighboring bytes at once per `int` (int32_t)
- The next element might already be loaded into the cache
- If not $n \gg M$ this might occur with a high probability

Quicksort:

- **Strategy:** Divide and Conquer
- Divide the data into two parts where the "left" part contains all values $\leq$ the values in the right part
- Choose one element (e.g the first one) as "pivot" element
- Ideally both parts are the same size
- Both parts are sorted recursively

p	list

lower list	p	upper list

Figure: *Quicksort* with pivot element

Idea of Quicksort

- **At start:** pivot in first position, first re-arrange list such that left part contains smaller and right part larger elements
- Do required changes *in place*

Pivot

Iterate this procedur

Index i

Index k
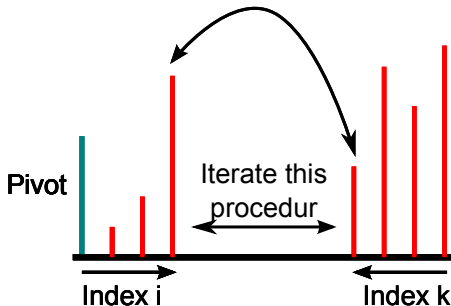
- **End point:** k is left to left-most element greater than pivot
  swap position 0 (pivot) with k (smaller than pivot)

Python:

```python
def quicksort(l, start, end):
  if (end - start) < 1:
    return

  i = start
  k = end
  piv = l[start]

  ...
```

```python
def quicksort(l, start, end):
  ...

  while k > i:
    while l[i] <= piv and i <= end and k > i:
      i += 1
    while l[k] > piv and k >= start and k >= i:
      k -= 1

    if k > i: # swap elements
      (l[i], l[k]) = (l[k], l[i])

  (l[start], l[k]) = (l[k], l[start])
  quicksort(l, start, k - 1)
  quicksort(l, k + 1, end)
```

Number of operations for Quicksort:

- Let $T(n)$ be the runtime for the input size n

Assumptions:

- Arrays are always separated perfectly in the middle
- n is a power-of-two and recursion depth is $k = \log_2 n$

$$
\begin{aligned}
T(n) &\leq & \underbrace{A \cdot n}_{\text{splitting in two parts}} + \underbrace{2 \cdot T\left(\frac{n}{2}\right)}_{\text{recursive sort}} \\
&\leq & A \cdot n + 2\left(A \cdot \frac{n}{2} + 2 \cdot T\left(\frac{n}{4}\right)\right) \\
&= & 2A \cdot n + 4 \cdot T\left(\frac{n}{4}\right) \\
&\leq & 3A \cdot n + 8 \cdot T\left(\frac{n}{8}\right) \\
&\leq & \cdots \\
&\leq & k \cdot A \cdot n + 2^k \cdot T(1) \\
&= & \log_2 n \cdot A \cdot n + n \cdot T(1) \\
&\leq & \log_2 n \cdot A \cdot n + n \cdot A \in \mathcal{O}(n \log_2 n)
\end{aligned}
$$

Cache Read / Write

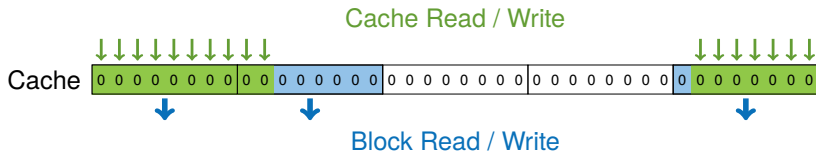
Figure: locality of Quicksort

- Let $IO(n)$ be the number of block operations for input size n
- Assumptions as before but recursion depth is $k = \log_2 \frac{n}{B}$

$$
\begin{aligned}
IO(n) \;\leq\; & \underbrace{A \cdot n/B}_{\text{splitting in two parts}} \;+\; \underbrace{2 \cdot IO(n/2)}_{\text{recursive sort}} \\[2mm]
\leq\; & A \cdot n/B \;+\; 2\,(A \cdot n/2B + 2 \cdot IO(n/4)) \\
\leq\; & 2 \cdot A \cdot n/B \;+\; 4 \cdot IO(n/4) \\
\leq\; & 3 \cdot A \cdot n/B \;+\; 8 \cdot IO(n/8) \\
\leq\; & \cdots \\
\leq\; & k \cdot A \cdot n/B \;+\; 2^k \cdot IO(n/2^k) \\
=\; & \log_2(n/B) \cdot A \cdot (n/B) \;+\; n/B \cdot IO(B) \\
\leq\; & \log_2(n/B) \cdot A \cdot (n/B) \;+\; A \cdot n/B \quad \in \quad \mathcal{O}\left(\frac{n}{B} \cdot \log_2\left(\frac{n}{B}\right)\right)
\end{aligned}
$$

Concept:

- Divide the problem into smaller subproblems
- Conquer the subproblems through recursive solving.
  If subproblems are small enough solve them directly
- Connect all solutions of the subproblems to the solution of
  the full problem
- Recursive application of the algorithm to ever smaller
  subproblems
- Direct solving of sufficiently small subproblems

Divide and Conquer
Introduction - Python

- Function solve for solving a problem of size *n*

```python
def solve(problem):
    if n < threshold:
        return solution # solve directly
    else:
        # divide problem into subproblems
        # P1, P2, ..., Pk with k>=2
        S1 = solve(P1)
        S2 = solve(P2)
        ...
        Sk = solve(Pk)

        # combine solutions
        return S1 + S2 + ... + Sk
```

Divide and Conquer:

- Can help with conceptual hard problems
- Solution of the trivial problems has to be known
- Dividing into subproblems has to be possible
- Combination of solutions has to be possible

Divide and Conquer
Features

Features:

- Realization of efficient solutions
    - If trivial solution is $\in O(1)$
    - And separation / combination of subproblems is $\in O(n)$
    - And the number of subproblems is limited
    - The runtime is $\in O(n \cdot \log n)$
- Suitable for parallel processing
    - Parallel processing of subproblems possible since subproblems are independent of each other

Definition of the trivial case:

- Smaller subproblems are elegant and simple
- On the other hand the efficiency will be improved if relatively big subproblems can be solved directly
- Recursion depth should not get too big (stack / memory overhead)

Division in subproblems:

- Choosing the number of subproblems and the concrete allocation can be demanding

Combination of solutions:

- Typically conceptionally demanding

Example - Maximum Subtotal Input:

- Sequence *X* of *n* integers

Output:

- Maximum sum of related subsequence and its index boundary

Index	0	1	2	3	4	5	6	7	8	9
Value	31	-41	59	26	-53	58	97	-93	-23	84

Output: sum: 187, start: 2, end: 6

Application:

- Maximum profit of buying and selling shares

Figure: stock value over time

Divide and Conquer
Example - Maximum Subtotal - Python

Naive solution (brute force)
```python
def maxSubArray(X):
    # Store sum, start, end
    result = (X[0], 0, 0)
    for i in range(0, len(X)):
        for j in range(i, len(X)):
            subSum = 0
            for k in range(i, j + 1):
                subSum += X[k]
            if result[0] < subSum:
                result = (subSum, i, j)
    return result
```

Runtime - Upper bound

```python
def maxSubArray(X):
    result = (X[0], 0, 0)
    # n loops -> O(n)
    for i in range(0, len(X)):
        # max n loops -> O(n)
        for j in range(i, len(X)):
            # max n loops -> O(n)
            subSum = sum(X[i:j+1])
            if result[0] < subSum: # O(1)
                result = (subSum, i, j)
    return result
```

Upper bound:

- Three nested loops
- Each loop with runtime $O(n)$
- Algorithm runtime of $O(n^3)$

Lower bound:

Table: Operations

i	Additions	j
$\frac{n}{3} \in O(n)$	$\frac{n}{3} \in O(n)$	$\frac{n}{3} \in O(n)$

- We iterate at least $\frac{n}{3}$ values for i
- For each i we iterate at least $\frac{n}{3}$ values for j
- For each j we have at least $\frac{n}{3}$ additions
- We need at least $T(n) = (\frac{n}{3})^3 \in \Omega(n^3)$ steps

Runtime:

- With $T(n) \in O(n^3)$ and $T(n) \in \Omega(n^3)$ we know:

$$T(n) \in \Theta(n^3)$$

- It is hard to solve the problem in a worse way …

Current approach:

- Calculating the sum for range from *i* to *j* with loop

$$S_{i,j} = X[i] + X[i+1] + \cdots + X[j]$$

Better approach:

- Incremental sum instead of loop

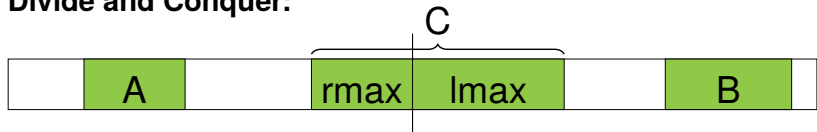$$S_{i,j+1} = X[i] + X[i+1] + \cdots + X[j] + X[j+1]$$
$$S_{i,j+1} = S_{i,j} + X[j+1] \in O(1) \quad \text{instead of} \quad \in O(n)$$

Better solution:

```python
def maxSubArray(X):
    result = (X[0], 0, 0)
    # n loops -> O(n)
    for i in range(0, len(X)):
        subSum = 0
        # max n loops -> O(n)
        for j in range(i, len(X)):
            subSum += X[j] # O(1)
            if result[0] < subSum: # O(1)
                result = (subSum, i, j)
    return result
```

- Runtime $\in O(n^2)$

Divide and Conquer
Example - Maximum Subtotal

Divide and Conquer:

Divide and Conquer idea to solve:

- Split the sequence in the middle
- Solve left half of the problem
- Solve right half and combine both solutions into one
- Maximum might be located in left half (*A*) or right half (*B*)
- Problem: Maximum can overlap the split
- To solve this case we have to calculate rmax and lmax
- The overall solution is the maximum of *A*, *B* and *C*

Principle - Divide and Conquer:

- Small problems are solved directly: $n = 1 \Rightarrow \max = X[0]$
- Bigger problems are partitioned into two subproblems and solved recursively. Subsolutions A and B are returned
- To determine subsolution C, rmax and lmax for the subproblems are computed
- The overall solution is the maximum of *A*, *B* and *C*

Divide and Conquer
Example - Maximum Subtotal - Python

```python
def maxSubArray(X, i, j):
    if i == j: # trivial case
        return (X[i], i, i)

    # recursive subsolutions for A, B
    m = (i + j) // 2
    A = maxSubArray(X, i, m)
    B = maxSubArray(X, m + 1, j)

    # rmax and lmax for cornercase C
    C1, C2 = rmax(X, i, m), lmax(X, m + 1, j)
    C = (C1[0] + C2[0], C1[1], C2[1])

    # compute solution from results A, B, C
    return max([A, B, C], key=lambda i: i[0])
```

Further Literature

- **General**

  [CRL01] Thomas H. Cormen, Ronald L. Rivest, and
  Charles E. Leiserson.
  Introduction to Algorithms.
  MIT Press, Cambridge, Mass, 2001.

  [MS08] Kurt Mehlhorn and Peter Sanders.
  Algorithms and data structures, 2008.
  https://people.mpi-inf.mpg.de/~mehlhorn/
  ftp/Mehlhorn-Sanders-Toolbox.pdf.

- **Caching**

  [Wik] Cache
  https://en.wikipedia.org/wiki/Cache