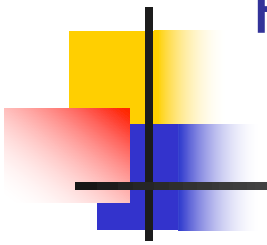


IN3030/IN4330, L02, våren 2024
Cache

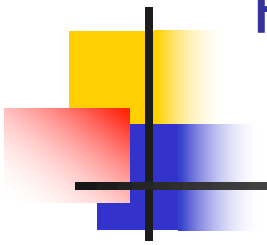
Michael Kirkedal Thomsen
Programming Technology Group
Programming Section
Department of Informatics
University of Oslo

Resume of first Lecture L01v25

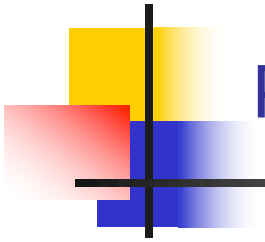


- Motivation
  - Utilization of Multi-core – by changing sequential programs into parallel programs
  - Multi-core hardware driving this trend
- Purpose
  - Convert sequential program into parallel versions
  - Achieve speedup
- Requirements
  - Correctness – Effective!
  - Efficient – MUST be FASTER – measured by speedup
- Approach
  - Empirical – *the proof of the pudding is in the eating!*
  - *Timings have the ULTIMATIVE SAY!*
  - *(Theory is fine; but practise is essential!)*

Resume of first Lecture L01v25 (second page)

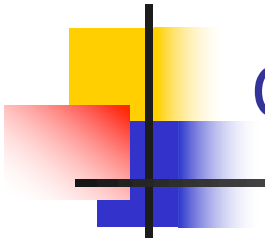


- Central metric: SPEEDUP
  - Speedup: sequential time / parallel time
  - Want speedup > 1
  - Really want speedup = *number of cores*
- How?
  - Parallel threads in Java
  - Must synchronize
- Measuring programs - how?
  - Real-time clock times!!
- Threads in Java
  - Template for parallel programs
- Synchonization
  - Cyclic Barrier in Java
  - Other synchronization methods in Java



Plan for L02v25

- Hva er Cache – intro
- Effekten på eksekveringstid av cache
- Presentasjon av Oblig1



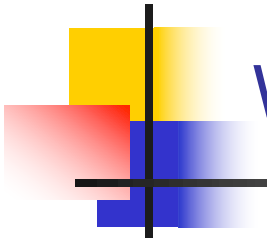
Caching

What is caching?

The principle of caching

Multi-level caching

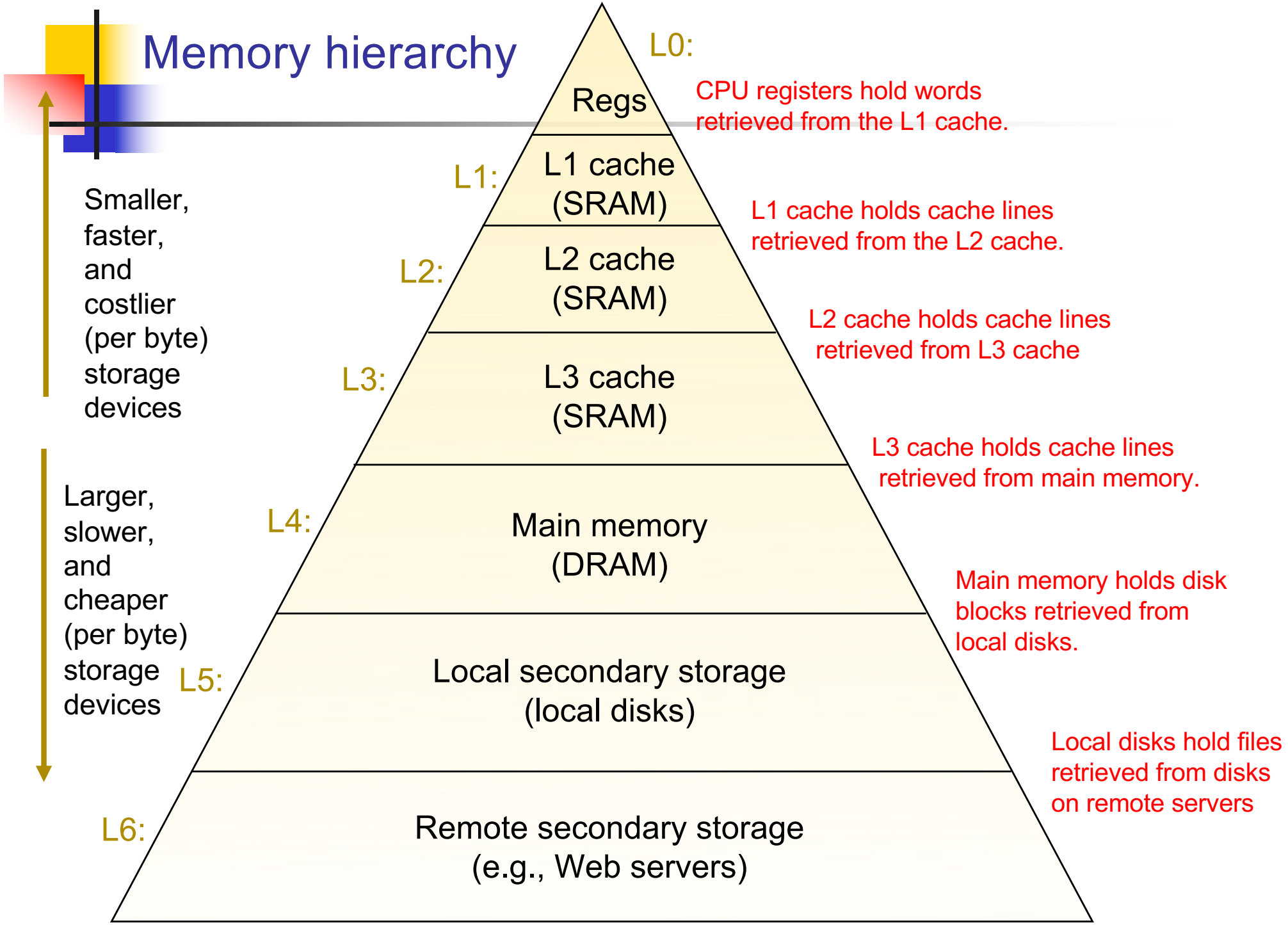
Prefetching



WHAT is a cache?

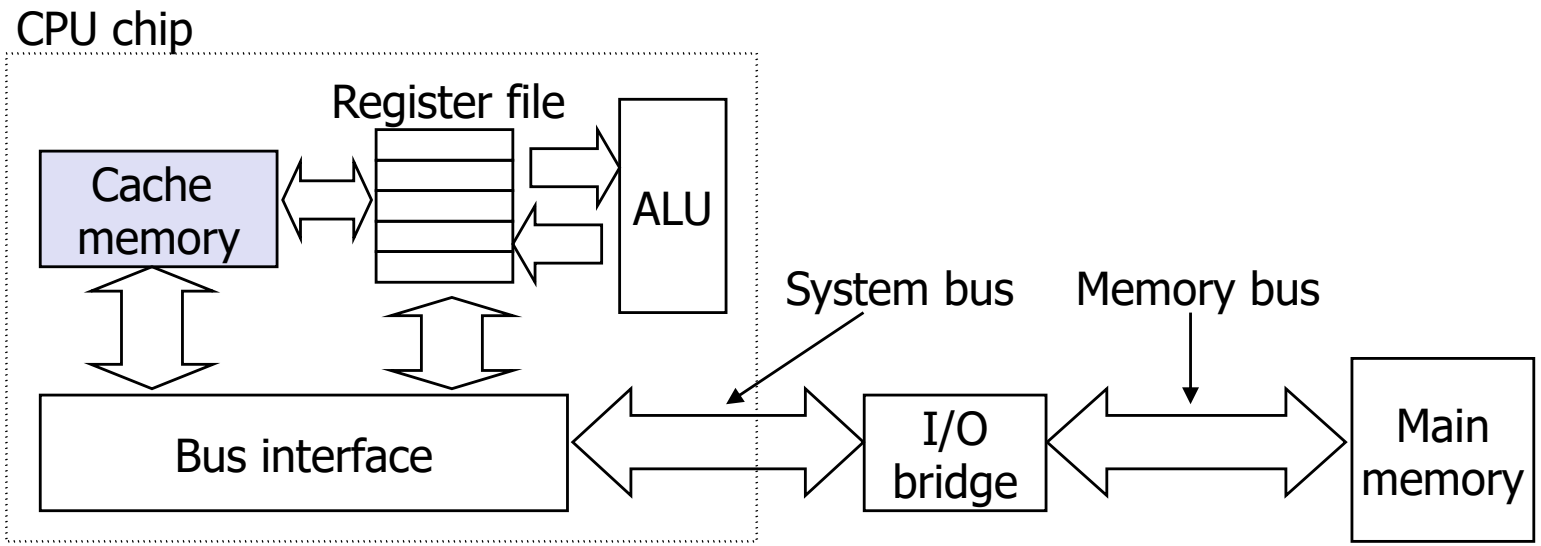
- Principle of a cache
  - CPUs fast – memory access slow
  - Move data into a fast cache
  - Make room: kick out Least Recently Used (LRU Strategy)
- Multiple level cache
- Speed of memory vs size
- Further performance enhancement: Prefetching

Memory hierarchy

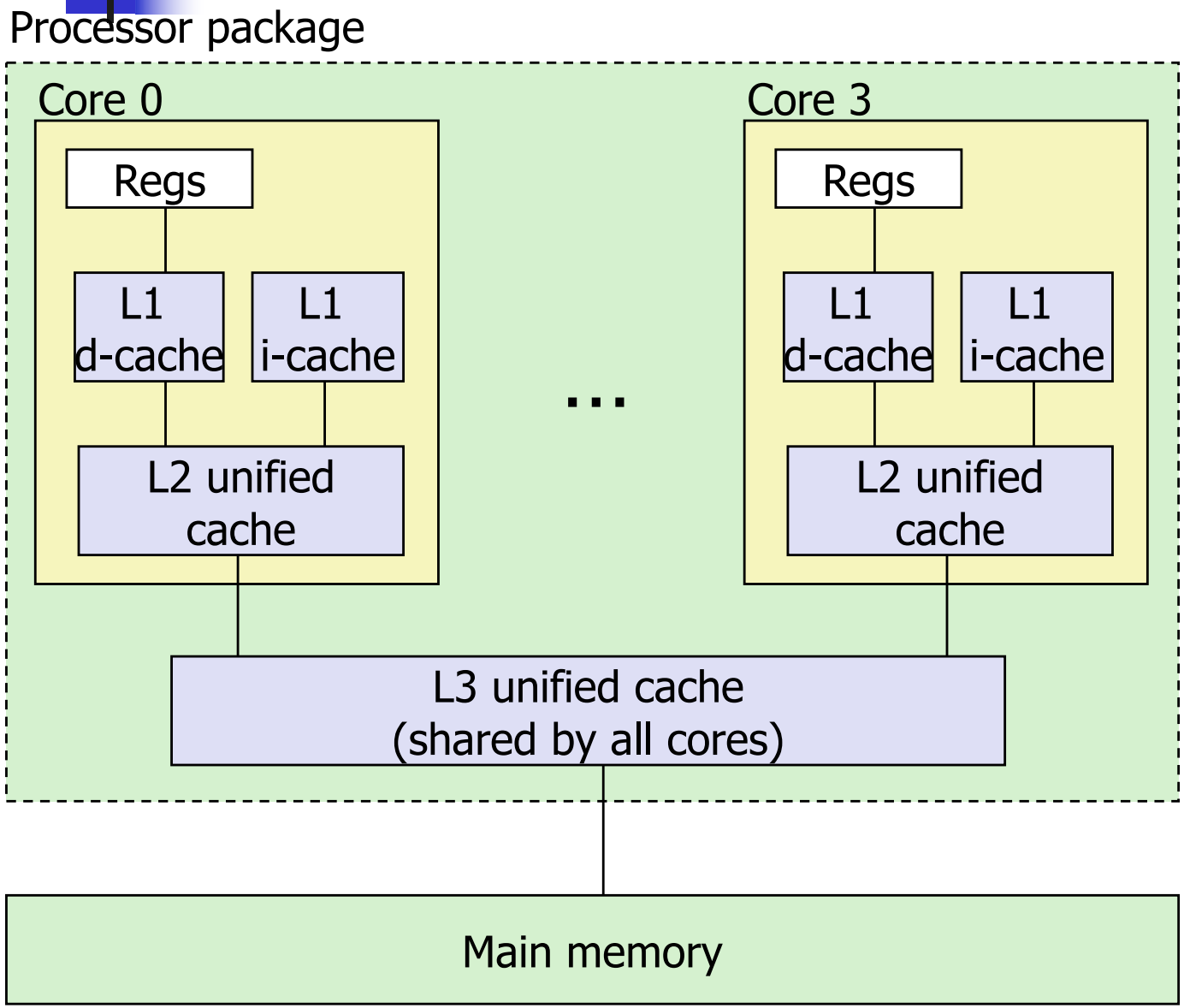


Cache Memories

- Cache memories are small, fast SRAM-based memories managed automatically in hardware
  - Hold frequently accessed blocks of main memory
- CPU looks first for data in cache
- Typical system structure:



Intel Core i7 Cache Hierarchy



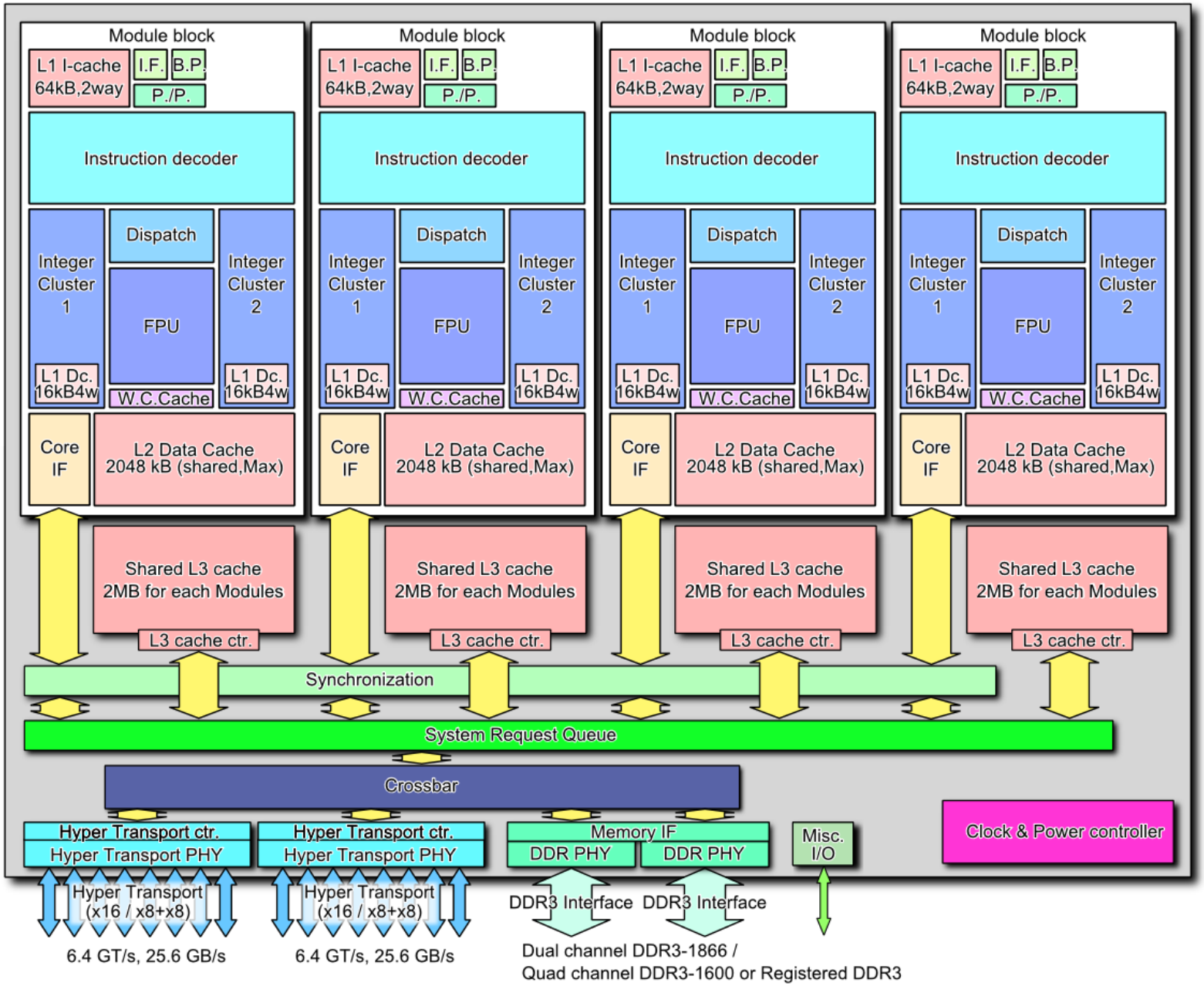
L1 i-cache and d-cache:
32 KB, 8-way,
Access: 4 cycles

L2 unified cache:
256 KB, 8-way,
Access: 10 cycles

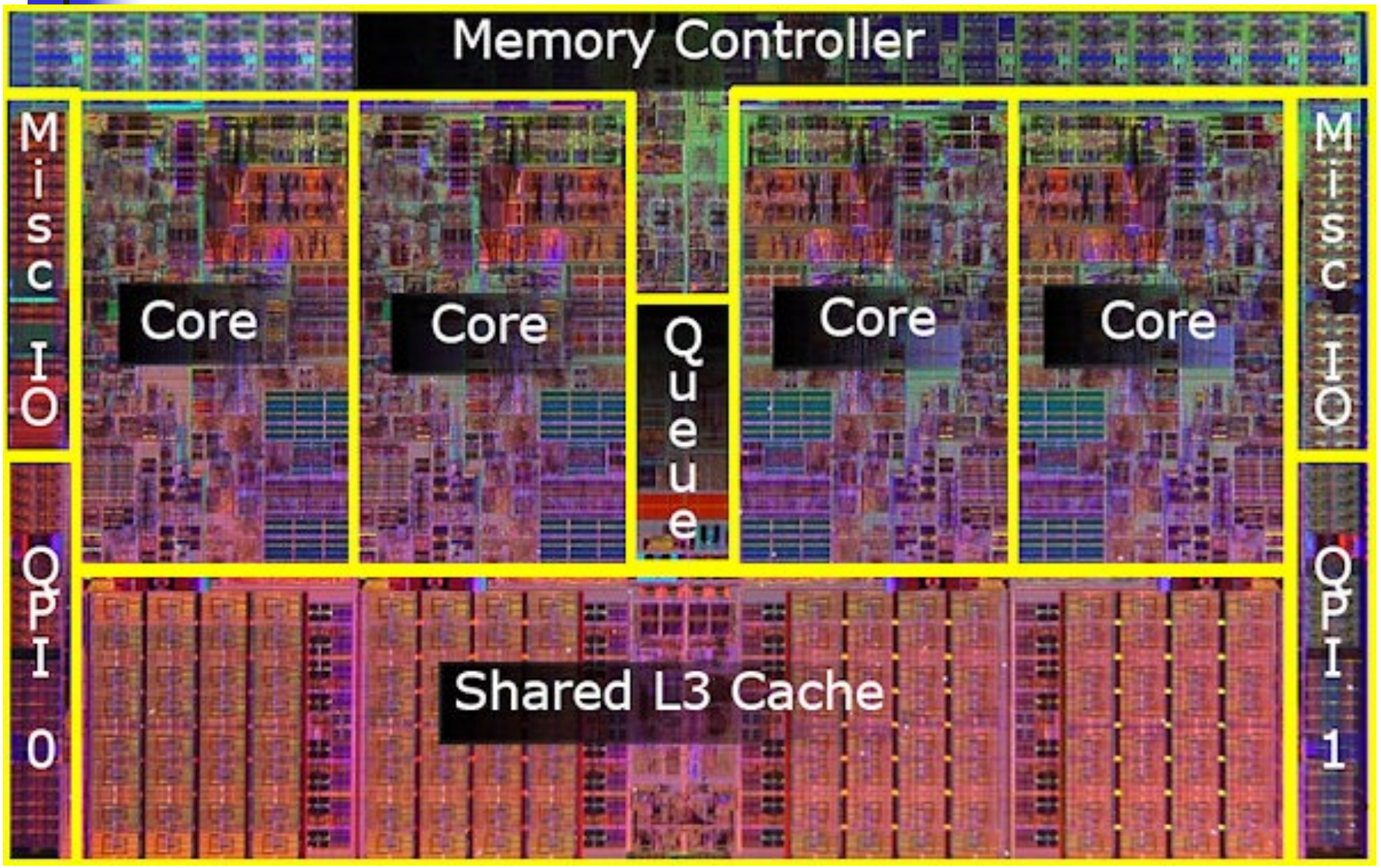
L3 unified cache:
8 MB, 16-way,
Access: 40-75 cycles

Block size: 64 bytes for all caches.

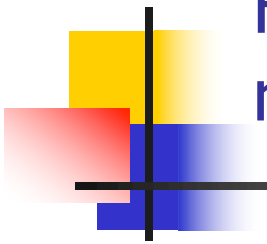
Hukommelses-systemet i en 4 kjerne CPU – mange lag og flere ulike beregningsmoduler i hver kjerne.:



Intel Core i7 - 4-core CPU



Hvordan tar vi hensyn til cache-systemet for å få raskere programmer?



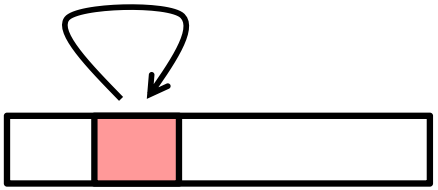
- Vi ser bare på data-cachene (lite å hente på instruksjonene)
- Viktig å vite er at hver gang vi skal hente data i hovedlageret , får vi en cach-linje = 64 byte = f.eks 8 heltall (int)
- Det er svært begrenset plass i cachene, og en cache-linje som ikke har vært brukt på 'lenge' vil bli 'kastet ut' (overskrevet av en annen, nyere) cache-linje
- Slik er raskest:
  - Jobber på få data (korte deler av en array) 'lenge' av gangen – ikke hoppe rundt.
  - Helst gå forlengs eller baklengs gjennom data (arrayene) (i, i+1,.. eller: i, i-1,..)

Vi må lage slike cache-vennlige programmer !

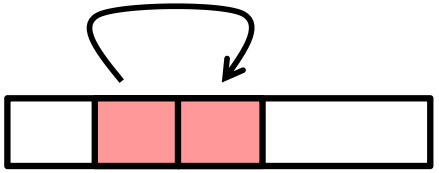
Locality

- Principle of Locality: Programs tend to use data and instructions with addresses near or equal to those they have used recently

- Temporal locality:
  - Recently referenced items are likely to be referenced again in the near future



- Spatial locality:
  - Items with nearby addresses tend to be referenced close together in time



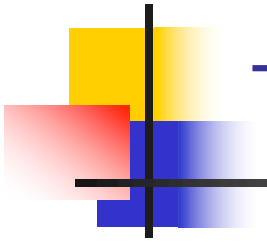


Writing Cache Friendly Code

- Make the common case go fast
  - Focus on the inner loops of the core functions

- Minimize the misses in the inner loops
  - Repeated references to variables are good (temporal locality)
  - Stride-1 reference patterns are good (spatial locality)

Key idea: Our qualitative notion of locality is quantified through our understanding of cache memories



The Memory Mountain

- Read throughput (read bandwidth)
  - Number of bytes read from memory per second (MB/s)

- Memory mountain: Measured read throughput as a function of spatial and temporal locality.
  - Compact way to characterize memory system performance.

The Memory Mountain test function

```
long data[MAXELEMS];  /* Global array to traverse */

/* test - Iterate over first "elems" elements of
 *        array "data" with stride of "stride", using
 *        using 4x4 loop unrolling.
 */
int test(int elems, int stride) {
    long i, sx2=stride*2, sx3=stride*3, sx4=stride*4;
    long acc0 = 0, acc1 = 0, acc2 = 0, acc3 = 0;
    long length = elems, limit = length - sx4;

    /* Combine 4 elements at a time */
    for (i = 0; i < limit; i += sx4) {
        acc0 = acc0 + data[i];
        acc1 = acc1 + data[i+stride];
        acc2 = acc2 + data[i+sx2];
        acc3 = acc3 + data[i+sx3];
    }

    /* Finish any remaining elements */
    for (; i < length; i++) {
        acc0 = acc0 + data[i];
    }
    return ((acc0 + acc1) + (acc2 + acc3));
}
```

mountain/mountain.c

Call `test()` with many combinations of `elems` and `stride`.

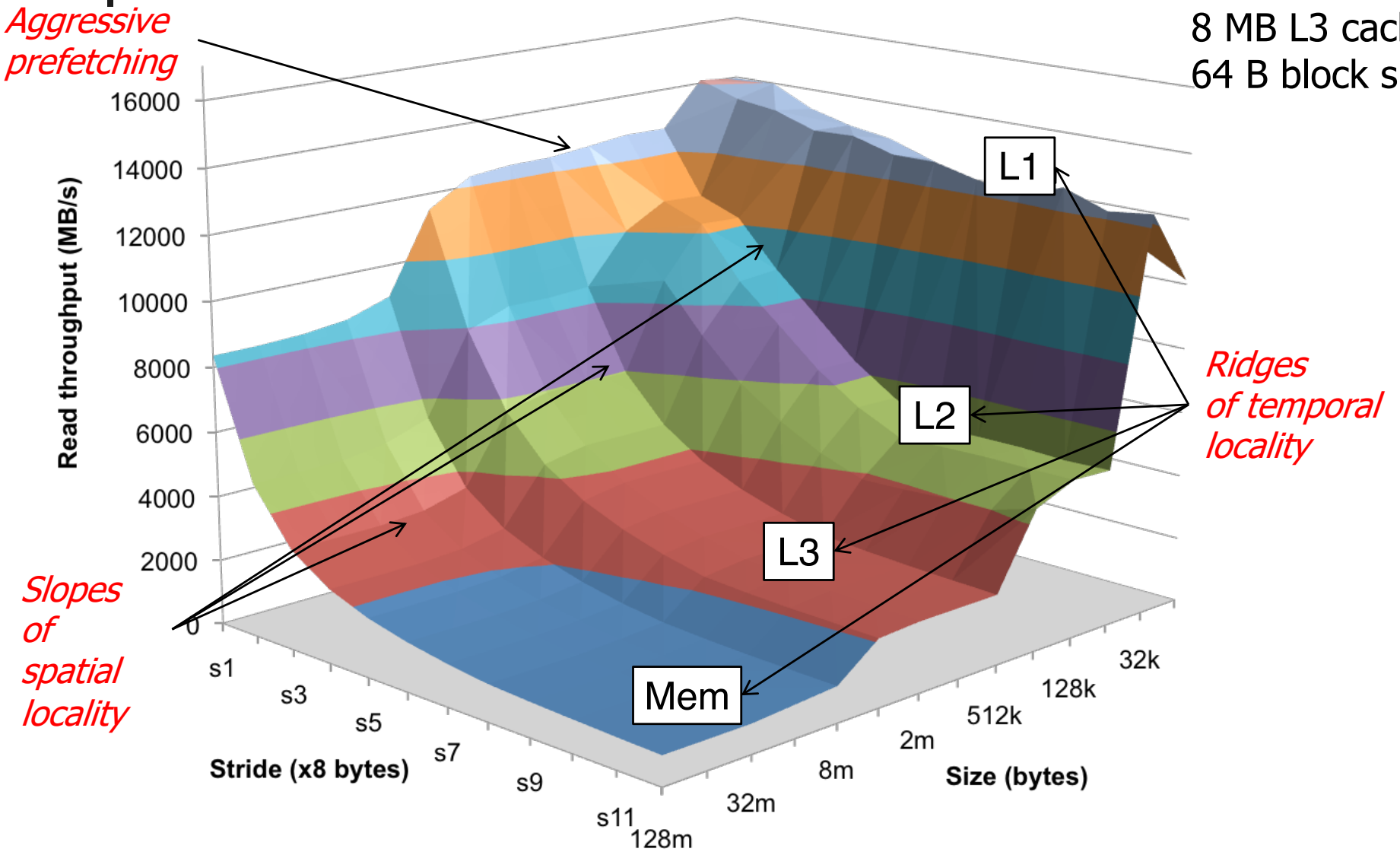
For each `elems` and `stride`:

1. Call `test()` once to warm up the caches.

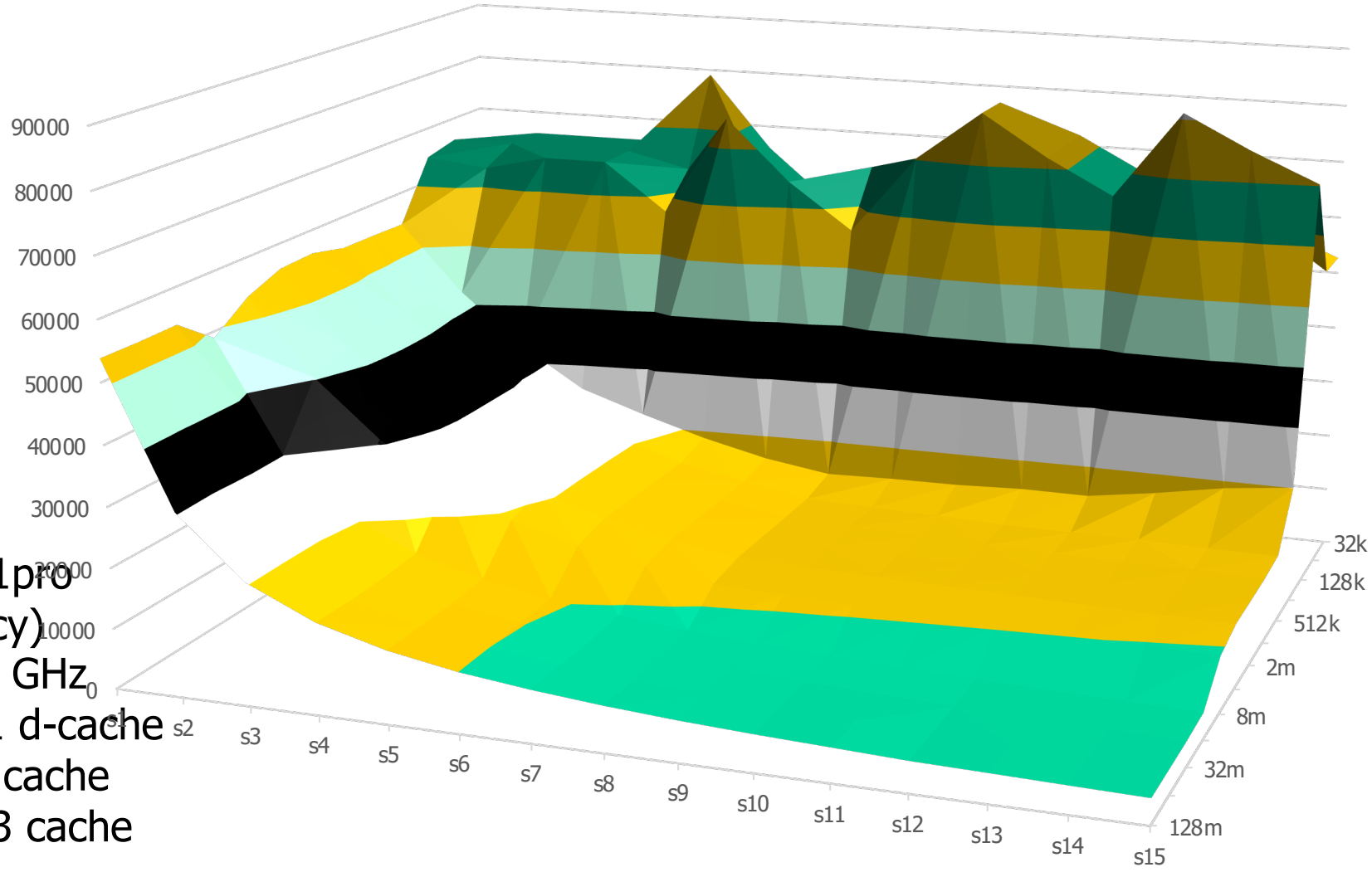
2. Call `test()` again and measure the read throughput (MB/s)

The Memory Mountain

Core i7 Haswell
2.1 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size

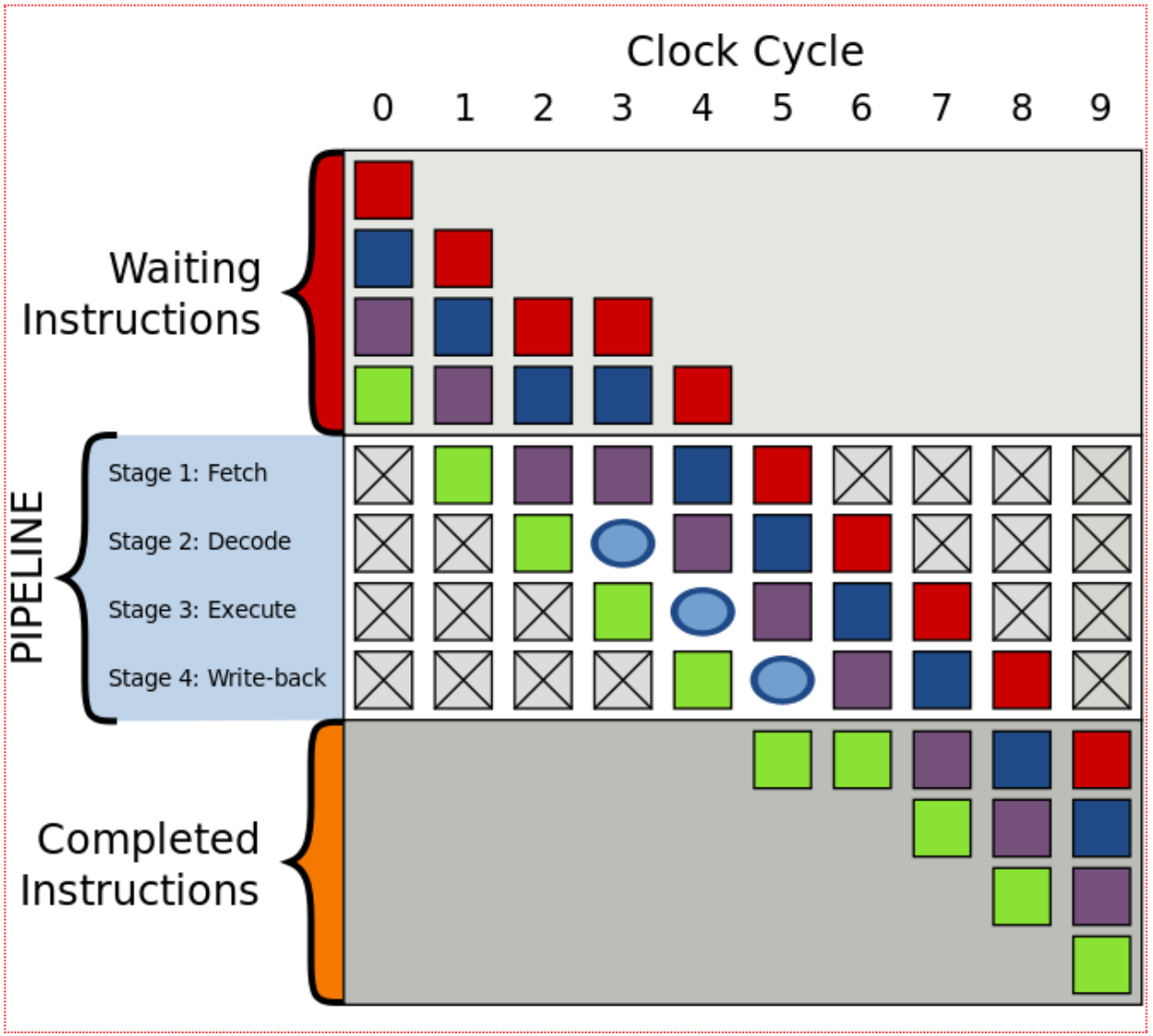


The memory mountain – my Apple M1pro



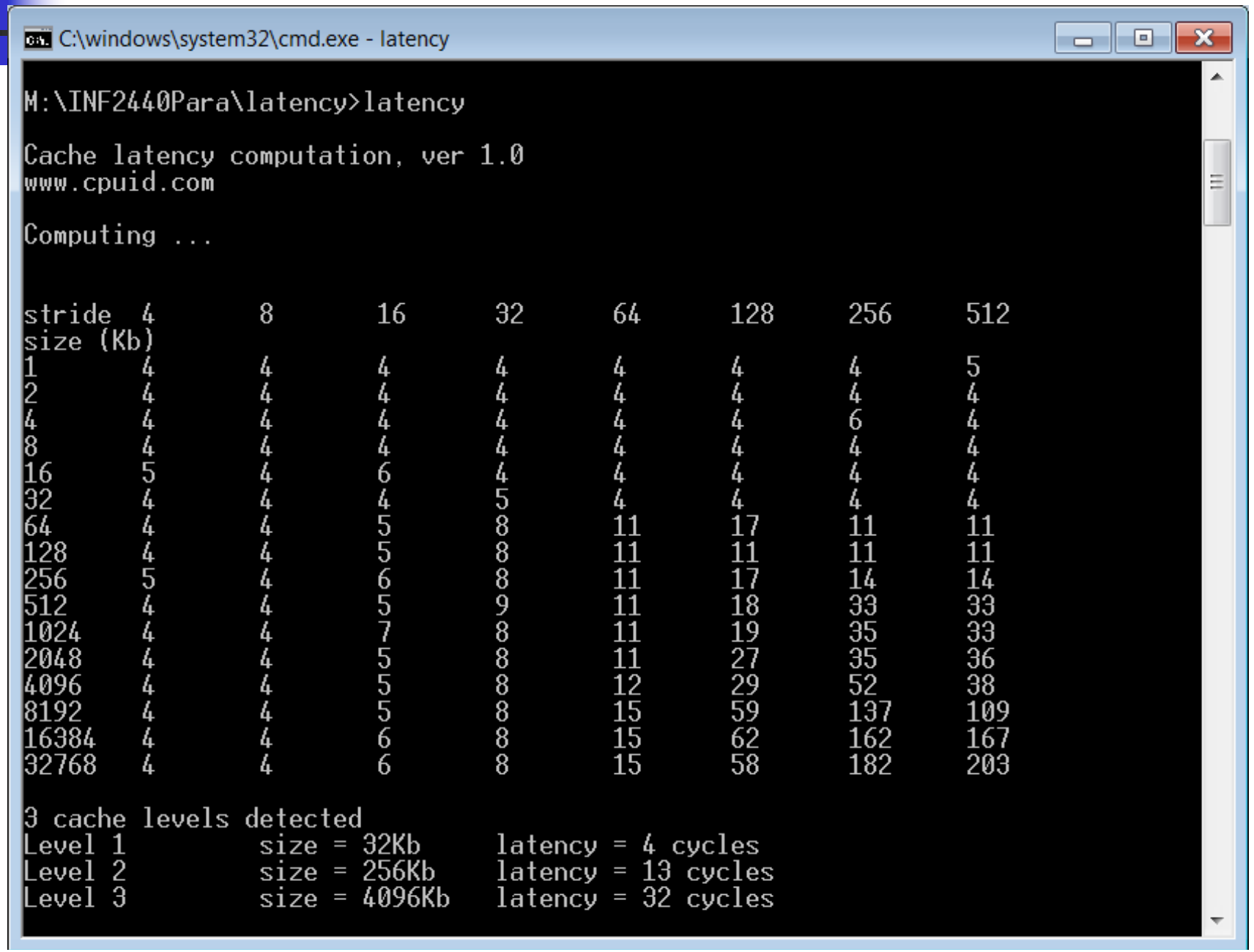
Apple M1pro (efficiency)
2.4 - 3.2 GHz
64 KB L1 d-cache
4 MB L2 cache
32 MB L3 cache

Instruksjonsparallellitet i en CPU-kjerne. Pipeline – flere instruksjoner (her 4) utføres *samtidig* i raskest mulig rekkefølge.



Note, that this in an extremely simplified model of modern machines.

Test av forsinkelse i data-cachene og hovedhukommelsen - latency.exe (fra CPUZ)



Oppsummering – ideen om at vi har *uniform* aksesstid i hukommelsen er helt galt

- Hukommelses-systemet i en multicore CPU ,Intel Core i5-459 3.3 GHz, – mange lag (typisk aksesstid i instruksjonssykler):

1. Registre i kjernen (1) – 8/32 registre
2. L1 cache (3-4) – 32 Kb – separate data and instruction
3. L2 cache (13) – 256 kb – both data and instruction
4. L3 cache (32) – 8Mb – shared between cores
5. Hovedhukommelsen (virtuell hukommelse) (ca. 200) – 8-64 GB
6. Disken (15 000 000 roterende) = 5 ms – 1000 GB – 1-5 TB
   FlashDisk (ca 2 000 000 les, ca. 10 000 000 skriv) = ca. 1 ms

Helt avgjørede for oss – cache-hukommelse

- Hva er cache
  - Raskere (men også dyrere) hukommelse mellom hovedlageret og kjernene.
  - Vi må ha cache fordi det er så store hastighetsforskjeller mellom en CPU-kjerne og hovedlageret ('main memory')
  - Ofte nå 3-4 lag med cache hukommelser + et antall registere i kjernen (enda raskere enn cache-hukommelsene) som holder data eller intruksjoner
  - Når en kjerne trenger data eller en ny instruksjon (og den ikke har det i et register) leter den nedover i cache-hukommelsene. Først cache level 1 (L1), så L2 cachen, .. , før den går til hovedhukommelsen for data eller instruksjoner.
  - Det finns flere teknikker for å gjøre dette raskt (som pre-fetch , dvs at systemet henter neste data/instruksjon uten at kjernen eksplisitt har bedt om det)

Matrix Multiplication Example

- Description:
  - Multiply N x N matrices
  - Matrix elements are doubles (8 bytes)
  - $O(N^3)$ total operations
  - N reads per source element
  - N values summed per destination
    - but may be able to hold in register

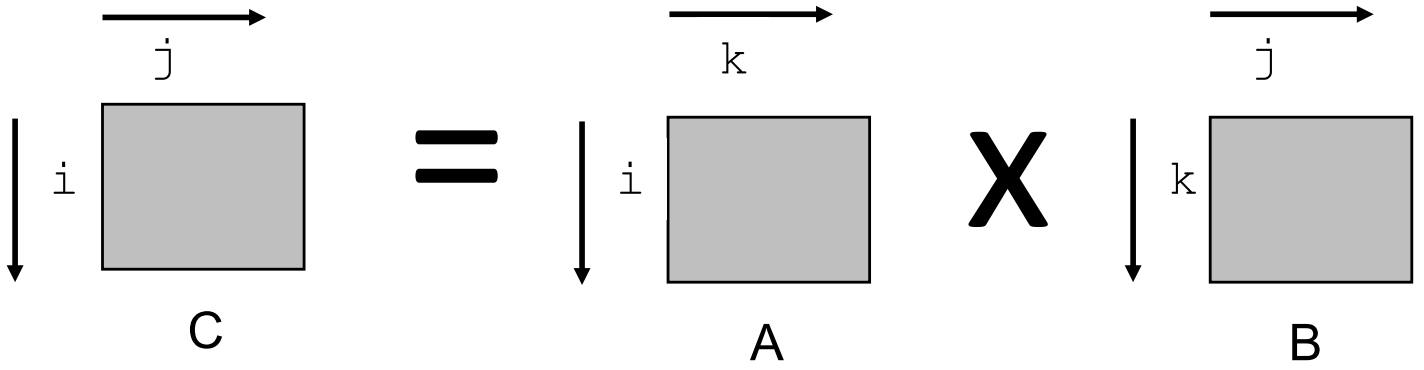
Variable `sum` *held in register*

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

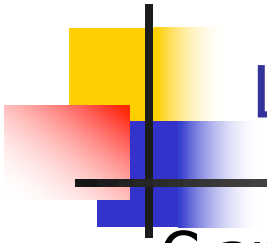
matmult/mm.c

Miss Rate Analysis for Matrix Multiply

- Assume:
  - Block size = 32B (big enough for four doubles)
  - Matrix dimension (N) is very large
    - Approximate 1/N as 0.0
  - Cache is not even big enough to hold multiple rows
- Analysis Method:
  - Look at access pattern of inner loop



Layout of C Arrays in Memory (review)



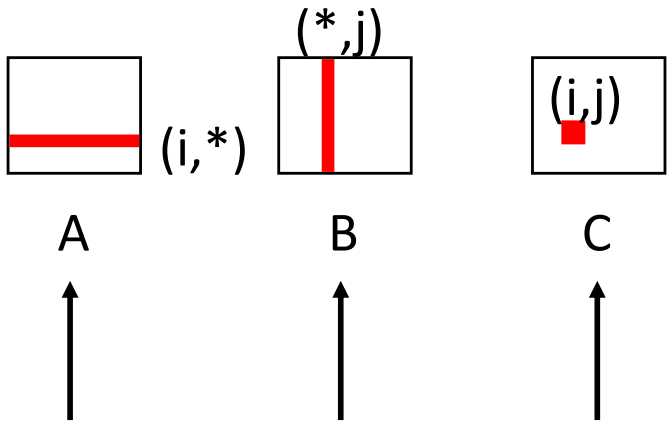
- C arrays allocated in row-major order
  - each row in contiguous memory locations
- Stepping through columns in one row:
  - `for (i = 0; i < N; i++)`
    `sum += a[0][i];`
  - accesses successive elements
  - if block size (B) > sizeof(a_{ij}) bytes, exploit spatial locality
    - miss rate = sizeof(a_{ij}) / B
- Stepping through rows in one column:
  - `for (i = 0; i < n; i++)`
    `sum += a[i][0];`
  - accesses distant elements
  - no spatial locality!
    - miss rate = 1 (i.e. 100%)

Matrix Multiplication (ijk)

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

matmult/mm.c

Inner loop:



Misses per inner loop iteration:

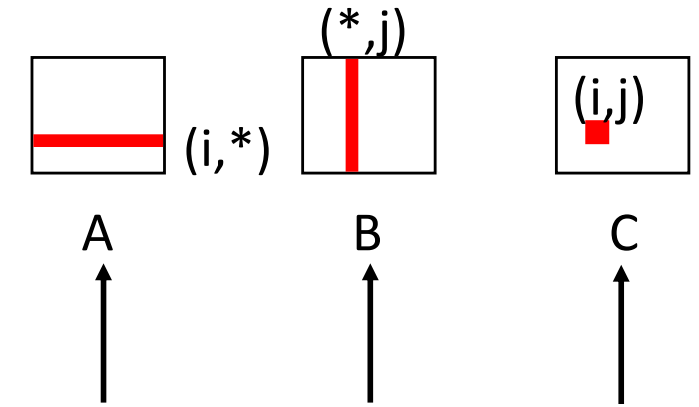
A	B	C
0.25	1.0	0.0

Matrix Multiplication (jik)

```
/* jik */
for (j=0; j<n; j++) {
  for (i=0; i<n; i++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum
  }
}
```

matmult/mm.c

Inner loop:



Misses per inner loop iteration:

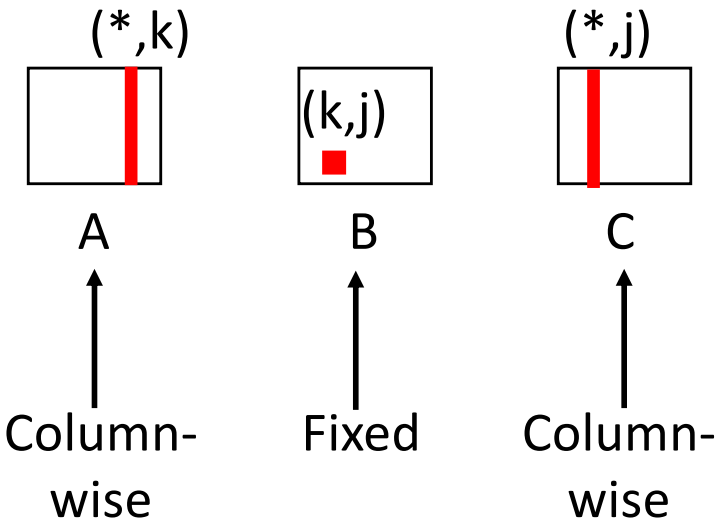
A	B	C
0.25	1.0	0.0

Matrix Multiplication (jki)

```
/* jki */
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

matmult/mm.c

Inner loop:



Misses per inner loop iteration:

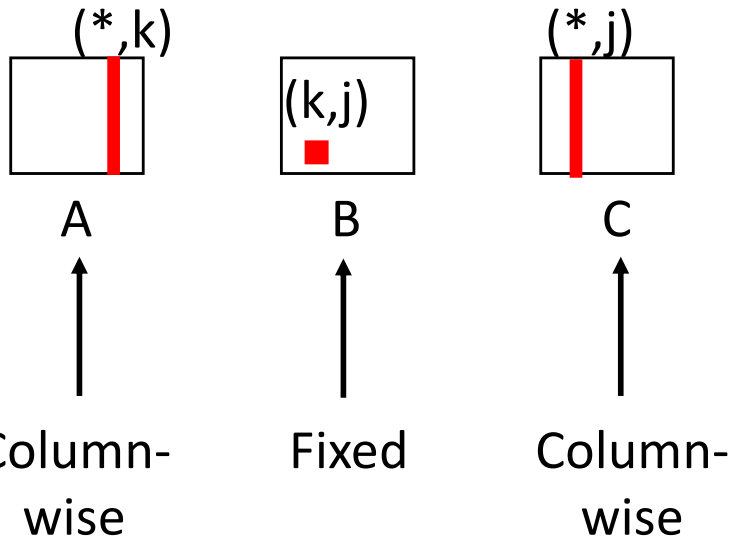
A	B	C
1.0	0.0	1.0

Matrix Multiplication (kji)

```
/* kji */
for (k=0; k<n; k++) {
  for (j=0; j<n; j++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

matmult/mm.c

Inner loop:



Misses per inner loop iteration:

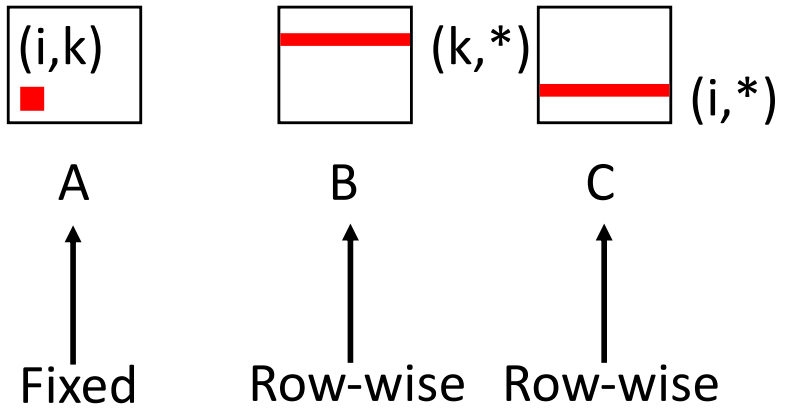
A	B	C
1.0	0.0	1.0

Matrix Multiplication (kij)

```
/* kij */
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

matmult/mm.c

Inner loop:



Misses per inner loop iteration:

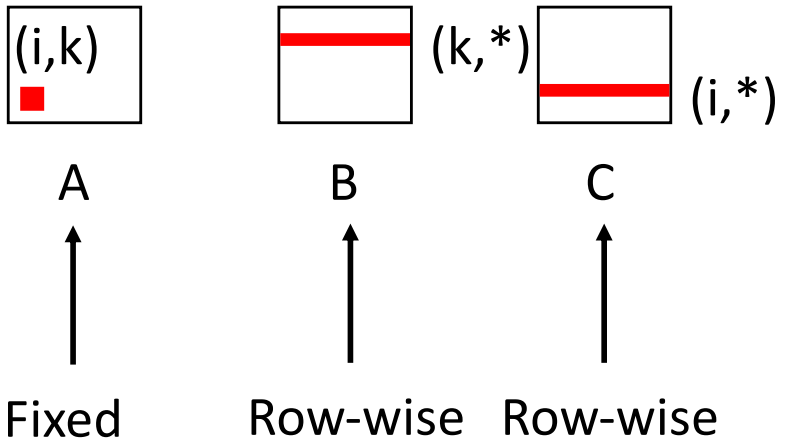
A	B	C
0.0	0.25	0.25

Matrix Multiplication (ikj)

```
/* ikj */
for (i=0; i<n; i++) {
  for (k=0; k<n; k++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

matmult/mm.c

Inner loop:



Misses per inner loop iteration:

A	B	C
0.0	0.25	0.25

Summary of Matrix Multiplication

```
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
   sum = 0.0;
   for (k=0; k<n; k++)
     sum += a[i][k] * b[k][j];
   c[i][j] = sum;
 }
}
```

ijk (& jik):

- 2 loads, 0 stores
- misses/iter = 1.25

```
for (k=0; k<n; k++) {
 for (i=0; i<n; i++) {
  r = a[i][k];
  for (j=0; j<n; j++)
   c[i][j] += r * b[k][j];
 }
}
```

kij (& ikj):

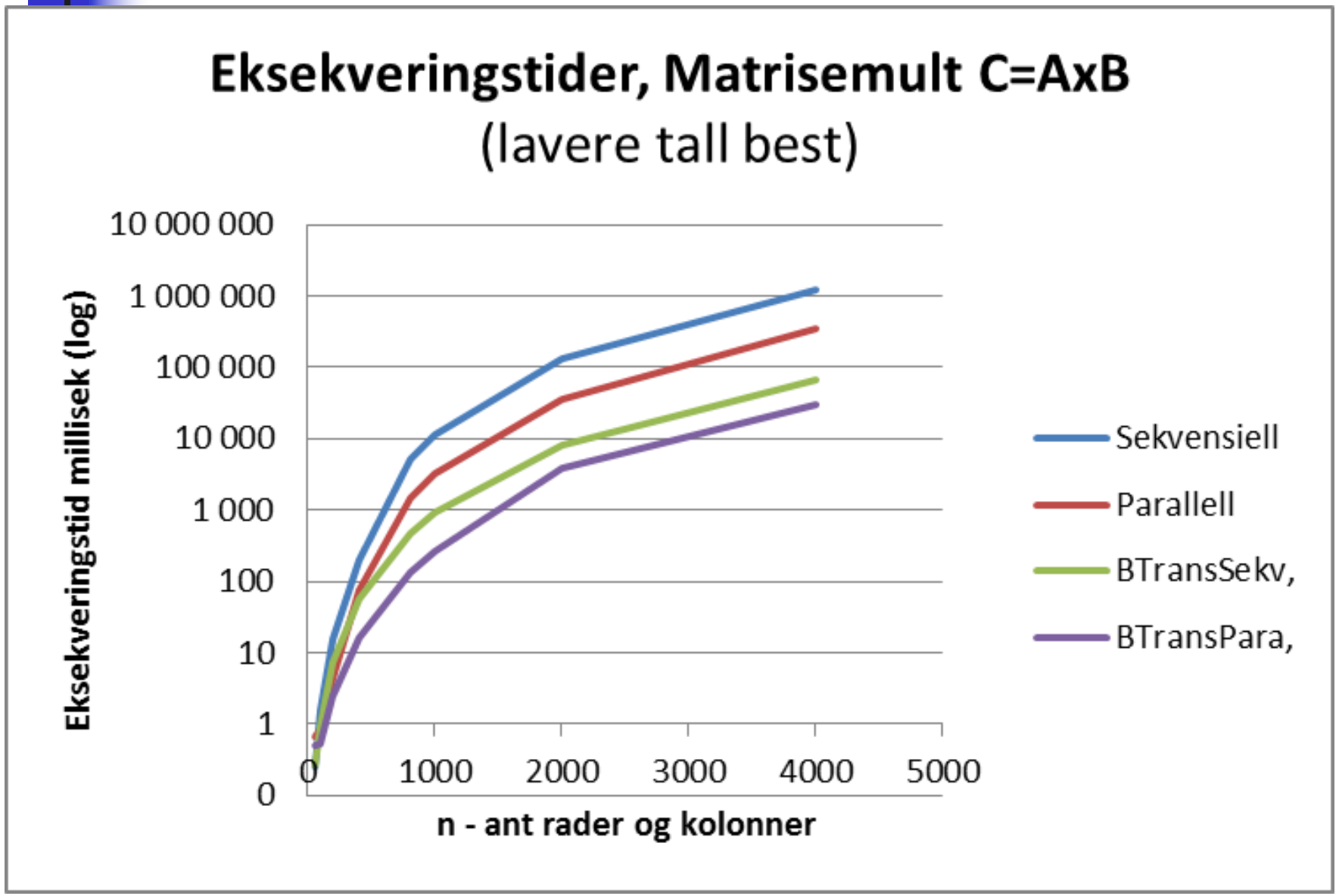
- 2 loads, 1 store
- misses/iter = 0.5

```
for (j=0; j<n; j++) {
 for (k=0; k<n; k++) {
   r = b[k][j];
   for (i=0; i<n; i++)
    c[i][j] += a[i][k] * r;
 }
}
```

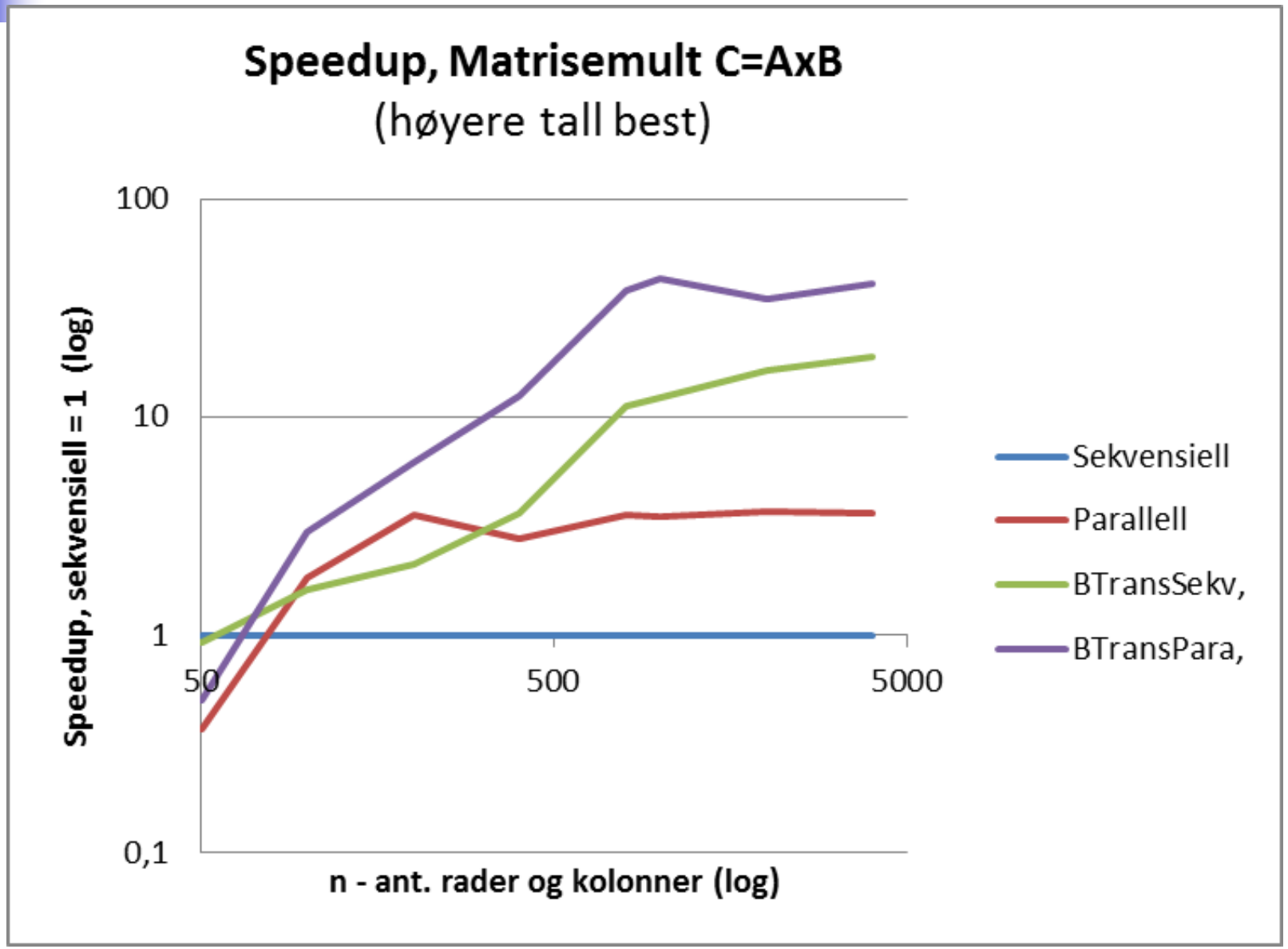
jki (& kji):

- 2 loads, 1 store
- misses/iter = 2.0

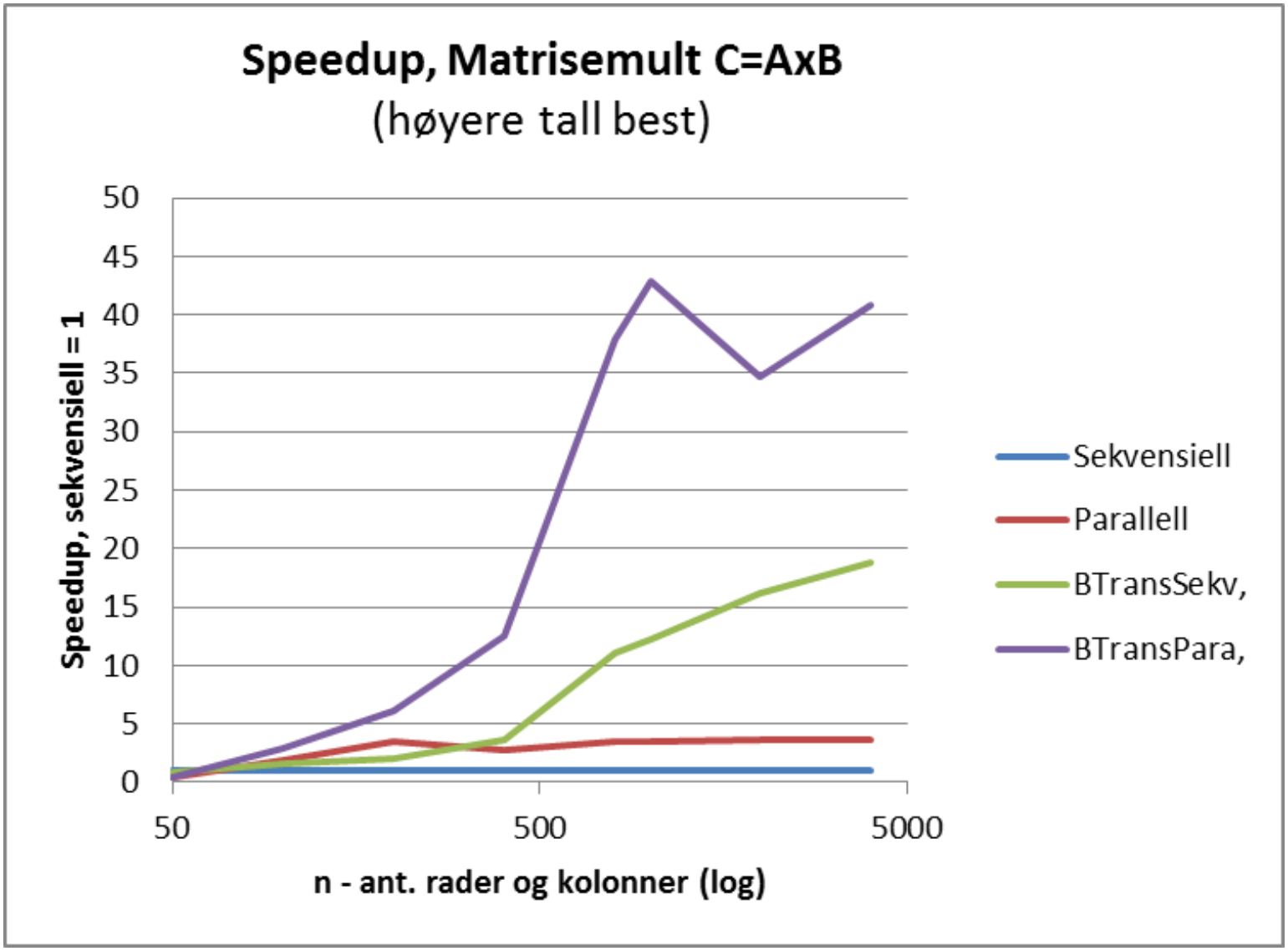
Kjøretider – i ms (y-aksen logaritmisk)



Kjøretidsresultater – Speedup , y-aksen logaritmisk



Speedup – med lineær y-akse

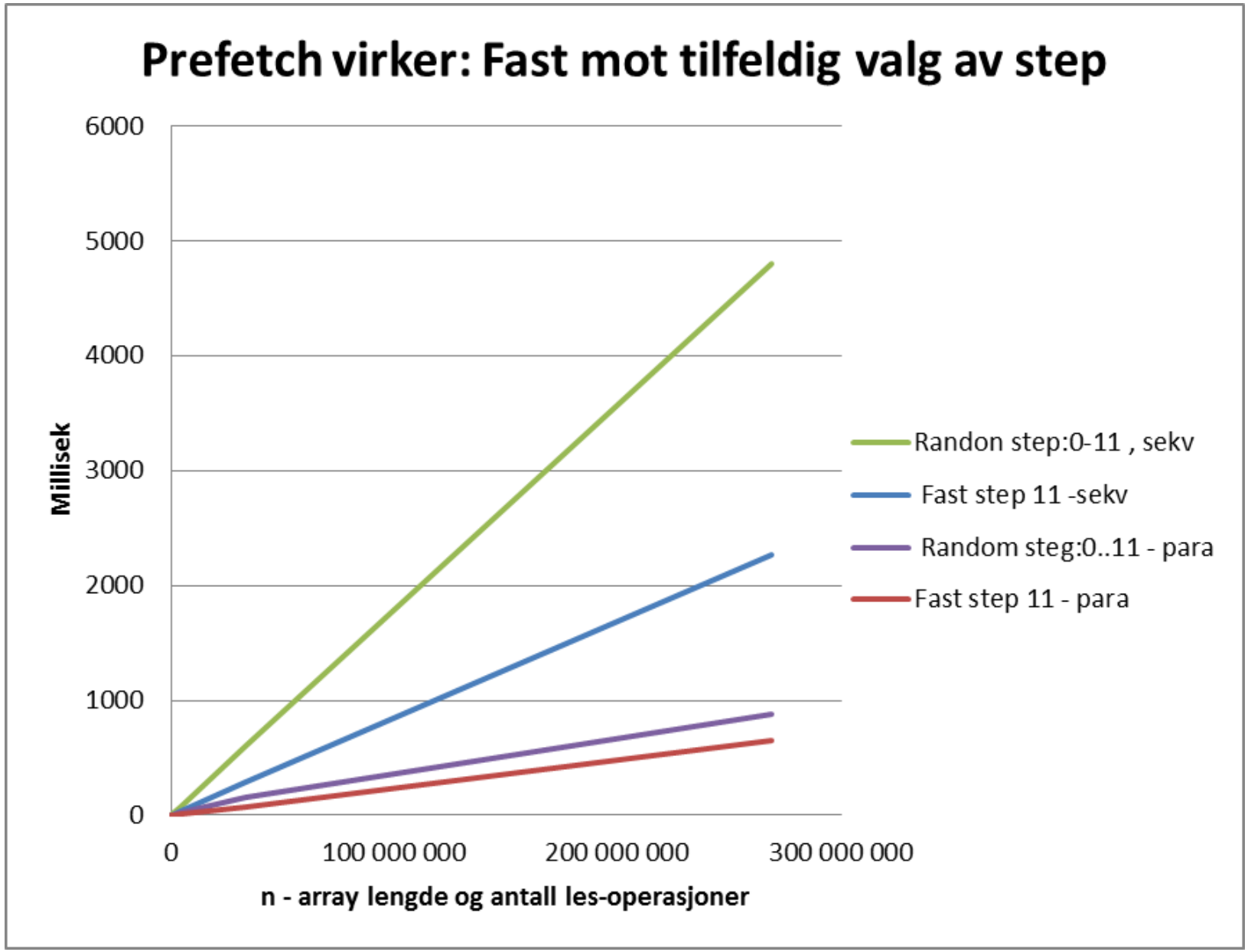


Prefetch-mekanismen på brikken

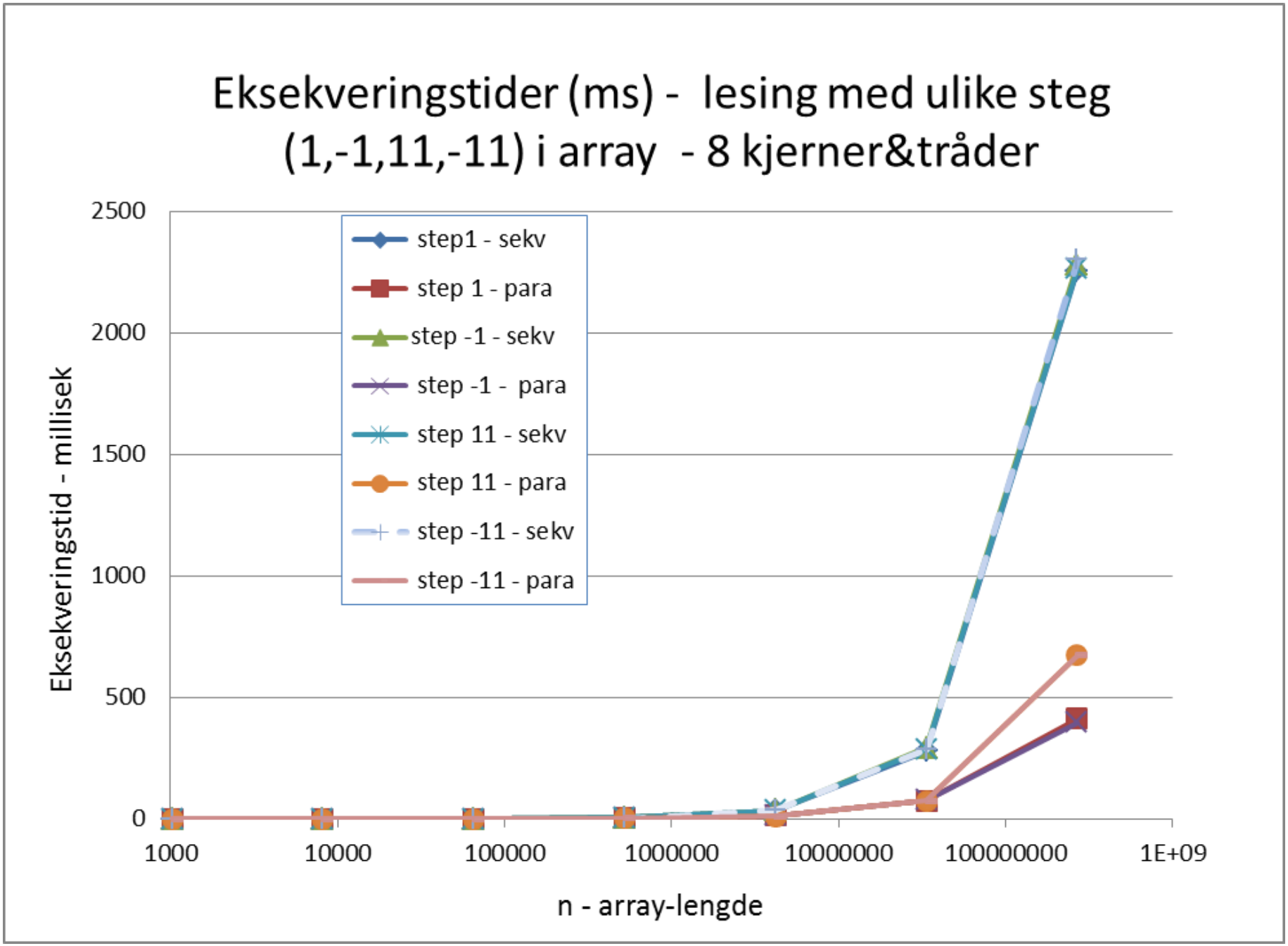
- Består i at hvis vi aksesserer (leser eller skriver) element i k i en array a[] og så element k+m, så prøver elektronikken å hente element a[k+2m], a[k+3m],…, ***før*** vi har bedt om det.
- Tillegget m kan være både positivt og negativt .
- Hvis elementene a[k+2m] ligger i samme cachelinje, går dette spesielt fort.
- Skrev testprogram for dette og testet m = 1, -1,11 og -11

```
for (int i=0;i < a.length; i++){
    // index = Math.abs((index+r.nextInt(step+1))%a.length);
    index = Math.abs((i+step)%a.length );
    sum += a[index];
}
```

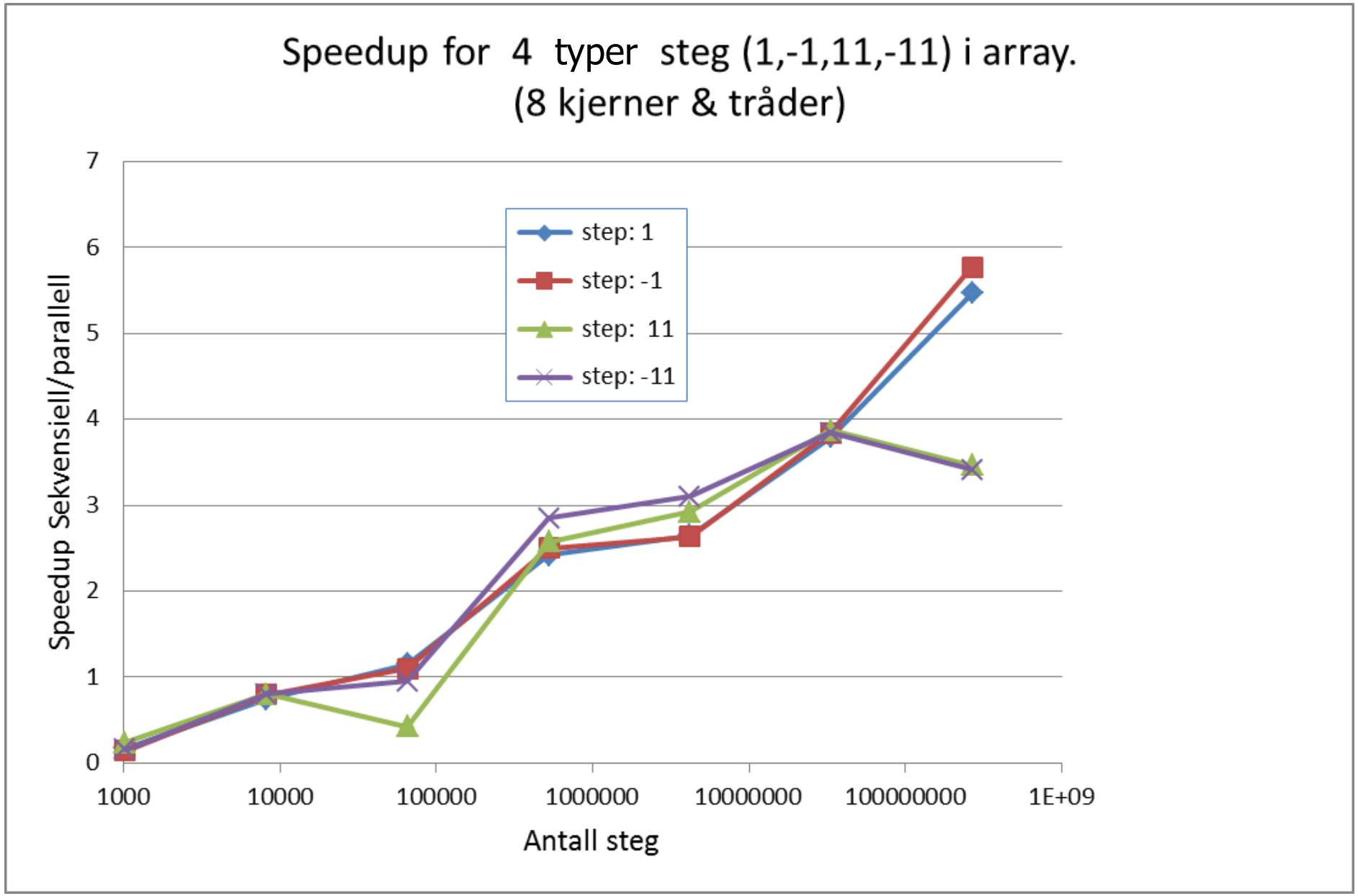
- Hva fant vi og hva kan vi slutte av det.
- Først grafer



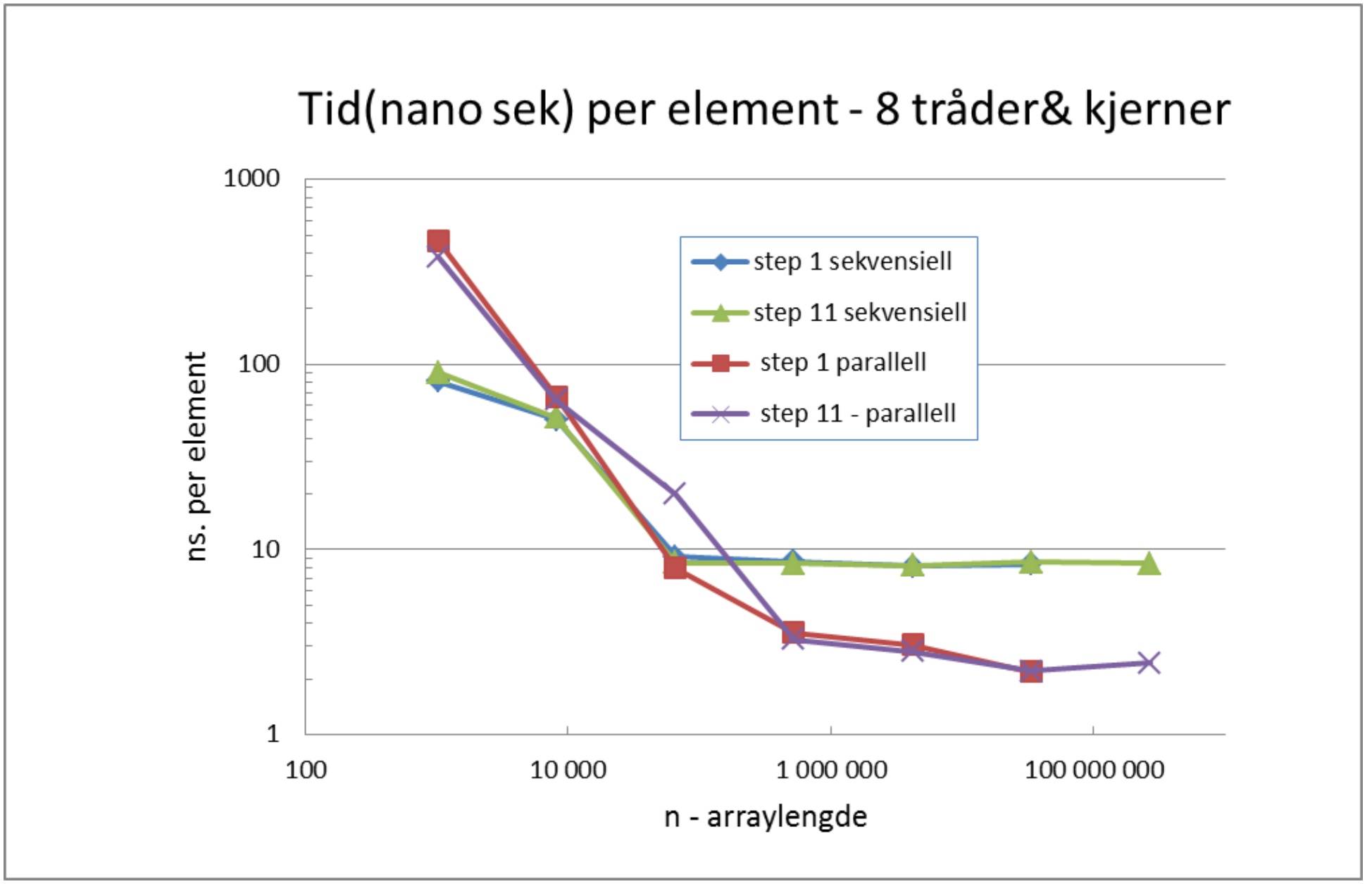
Konklusjon1: Fast steglengde er ca. dobbelt så raskt som tilfeldig (pga prefetch)



Konklusjon2: negative og positive tillegg er like raske.

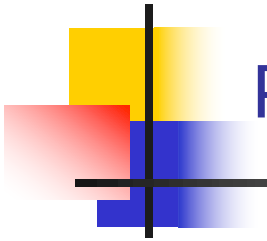


Konklusjon3: Spesielt steg 1,-1 har bra speedup pga. samme cachelinje



Konklusjon 4: Prosesseringstiden per element synker med økende n (JIT?)

Konklusjon 5: Per element tar det litt over 2*2,8 = ca. 6 instruksjoner å summere et element til en sum i parallell.



Prefetch-mekanismen hjelper en del

- Ikke så viktig som cache-systemet
- Ikke så viktig som JIT-kompilering
- men hjelper til og går på ingen måte i veien for de to viktigste mekanismene – ca. 2x raskere
- Programmet som laget data til disse grafene er laget av programmet Prefetch.java som er lagt ut på hjemmesida
- Grafene er laget i Excel (velg graftype:scatter diagram):
  - sett inn et slikt i regnearket og trykk så Select Data

Effekten på eksekveringstider av cache

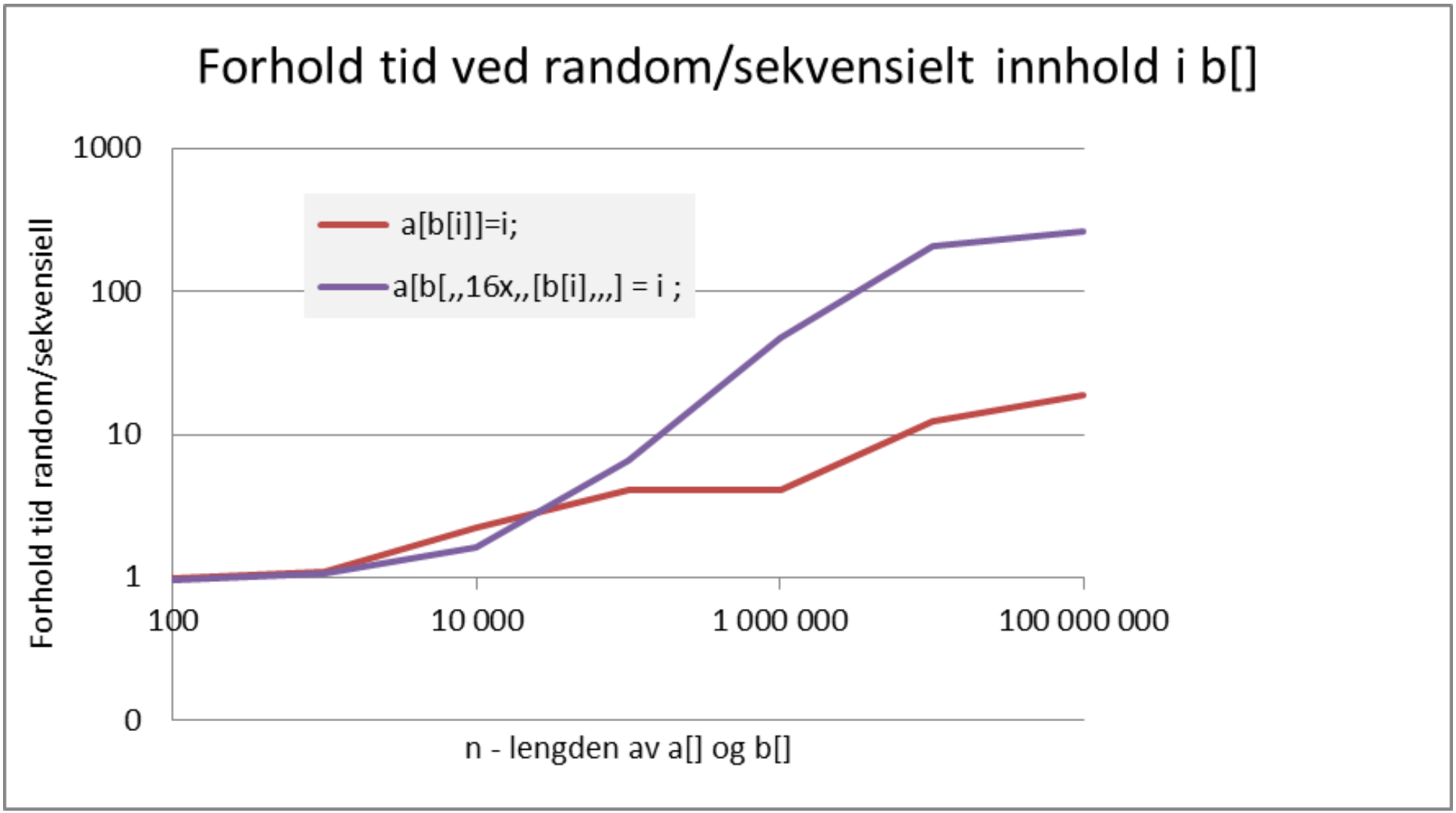
1) Hvor lang tid tar det å utføre n ganger (n=100, 1000, 10 000,….,100 mill):

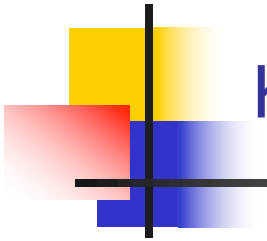
   a[b[i]]=i;

2) Avhenger av hva b[] inneholder:
   1) Hvis b[i] = i (sekvensiell) , så er a[b[i]] = a[i] og vi har 'alt' i cachen
   2) Hvis innholdet i b[] er tilfeldig trukket mellom 0:n-1, så er hver les/skriv i lageret en hopping frem og tilbake i a[] – ingen nytte av cachen

3) Neste graf viser hvor mange ganger lenger tid det tar å utføre ganger de to måtene å fylle b[]
   – enten b[i] = i, eller b[i] = random(0..n-1)

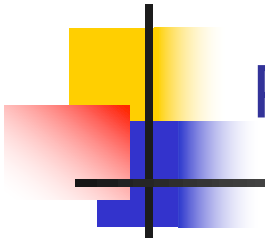
Hvor mange ganger tregere går random innhold i b[] enn b[] = 0,1,2,3,.. ?



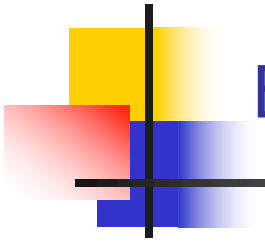


Konklusjon – nestet aksess a[b[i]]

- For 'små' verdier av n <1000, gir cachen god aksess til både hele a[] (viktigst), og til b[].
- For store verdier av n > 100 0000 blir det meget langsommere, og vi kan få mellom 12 – 240 ganger langsommere kode (pga. cache-miss) når innholdet av b[] er 'tilfeldig'.
- Slike uttrykk a[b[i]] og a[b[c[i]]] finner vi i Radix-sortering som vi skal granske i en senere forelesning.



Presentasjon av Oblig 1



End L02v2

- End of lecture L02v25