



Computer Hardware Engineering (IS1200)

Computer Organization and Components (IS1500)

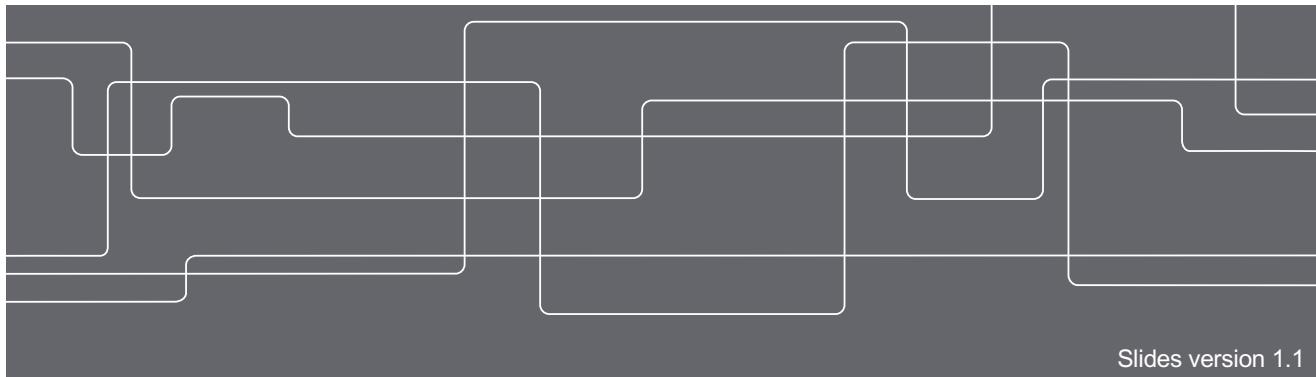
Spring 2019

Lecture 13: SIMD, MIMD, and Parallel Programming

Elias Castegren

Postdoc, KTH Royal Institute of Technology

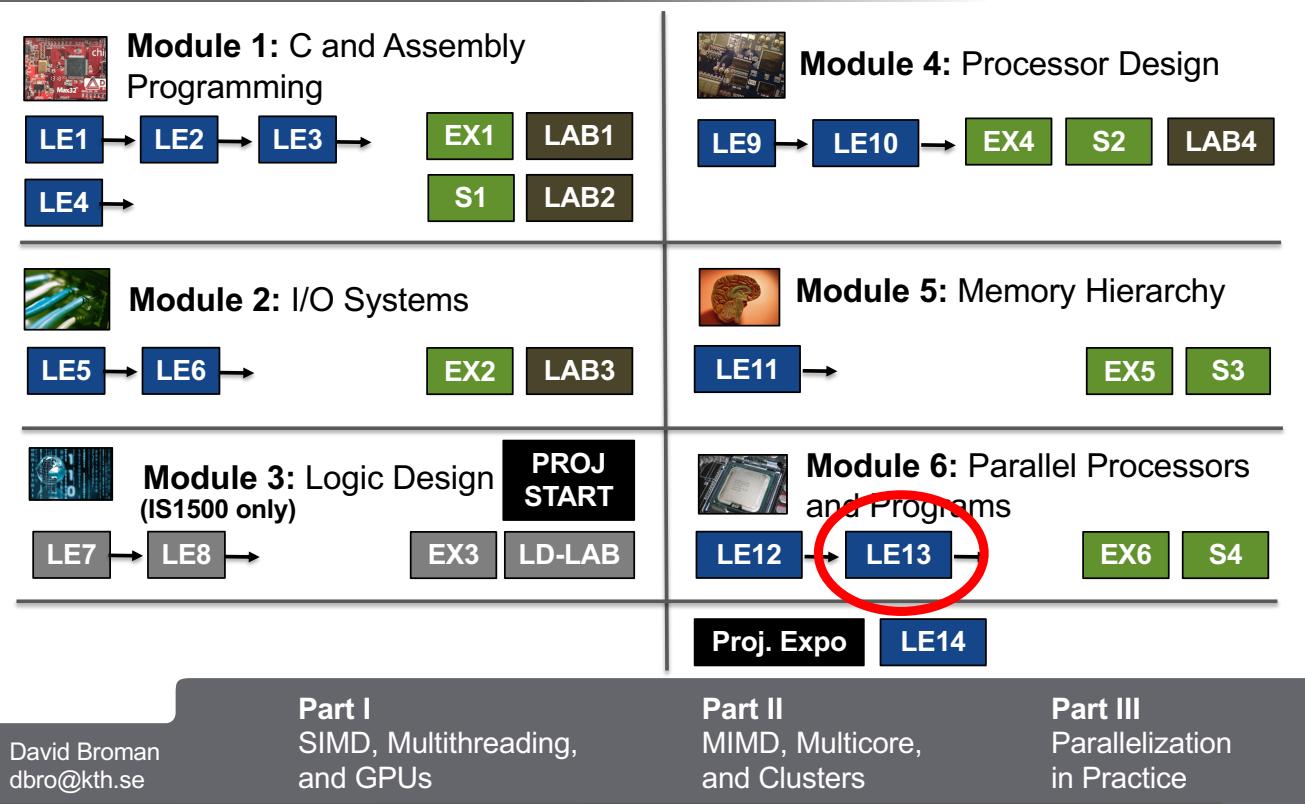
Slides by David Broman



2

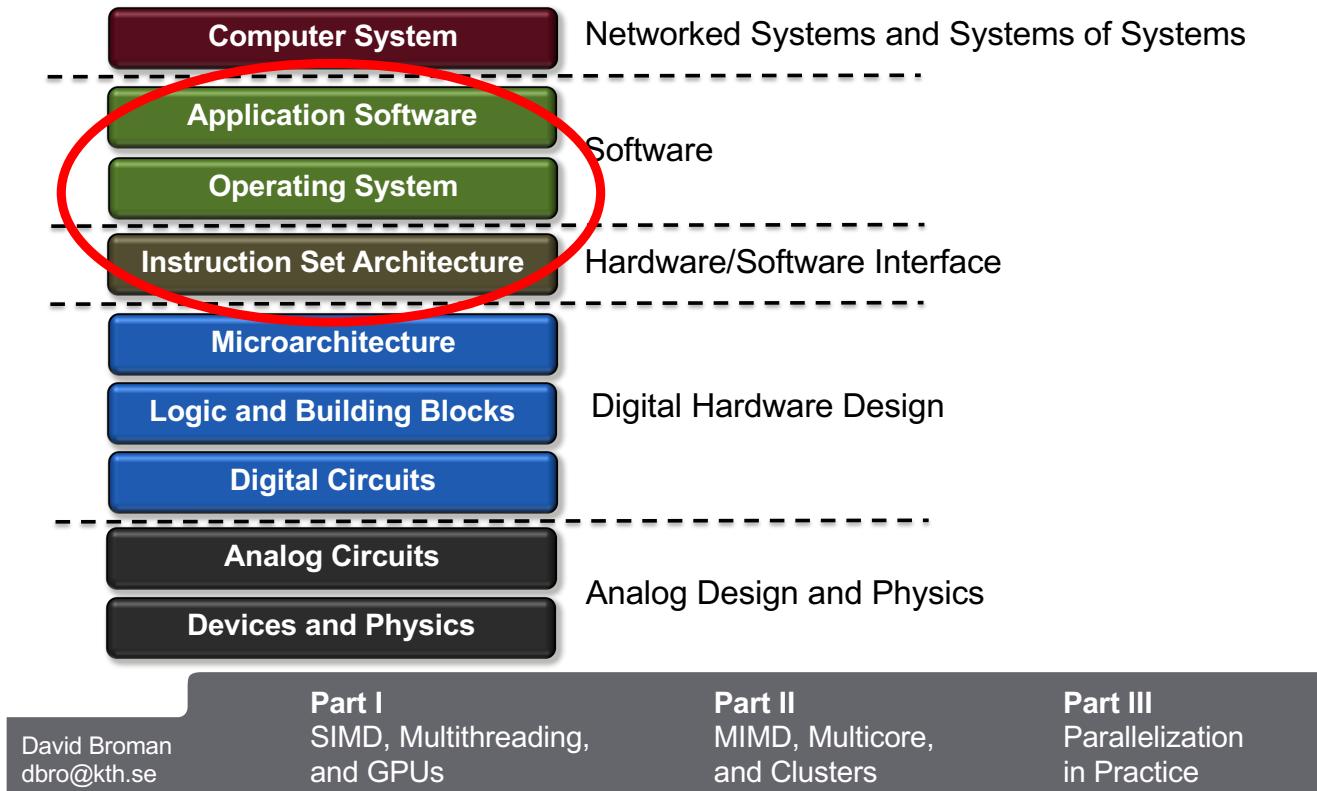


Course Structure

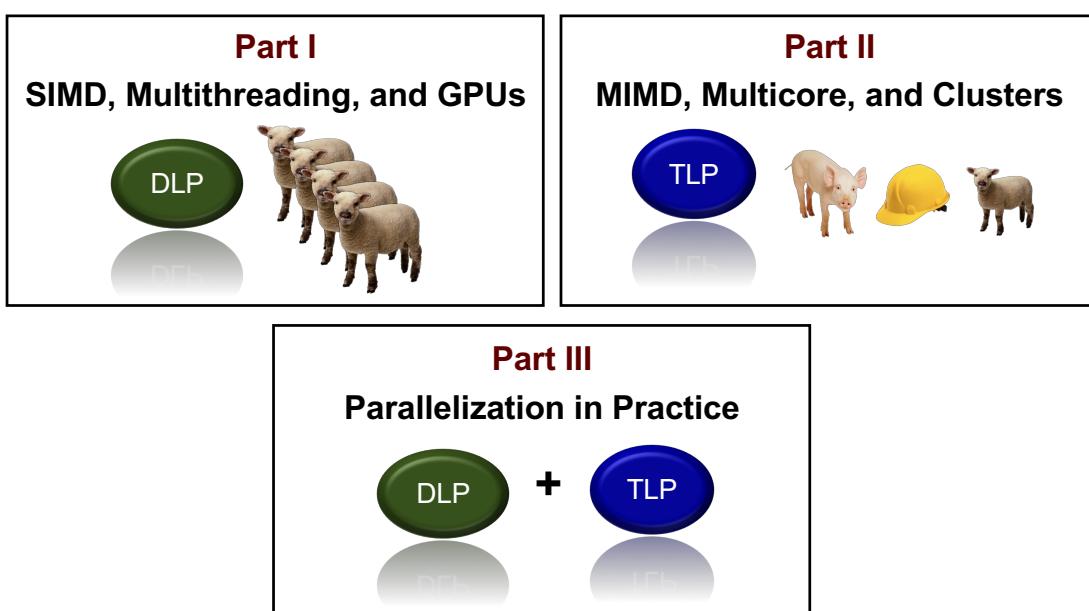




Abstractions in Computer Systems



Agenda



David Broman
dbro@kth.se

Part I
SIMD, Multithreading,
and GPUs

Part II
MIMD, Multicore,
and Clusters

Part III
Parallelization
in Practice

Part I

SIMD, Multithreading, and GPUs



Acknowledgement: The structure and several of the good examples are derived from the book "Computer Organization and Design" (2014) by David A. Patterson and John L. Hennessy

David Broman
dbro@kth.se

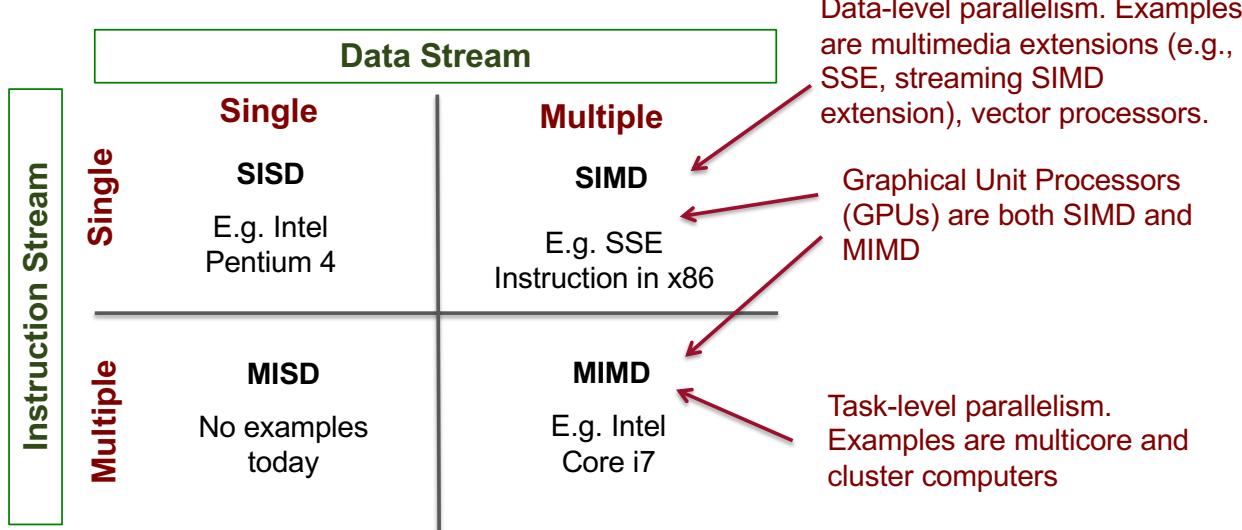


Part I
SIMD, Multithreading,
and GPUs

Part II
MIMD, Multicore,
and Clusters

Part III
Parallelization
in Practice

SISD, SIMD, and MIMD (Revisited)



David Broman
dbro@kth.se



Part I
SIMD, Multithreading,
and GPUs

Part II
MIMD, Multicore,
and Clusters

Part III
Parallelization
in Practice

Subword Parallelism and Multimedia Extensions

7

Subword parallelism is when a wide data word is operated on in parallel.

This is the same as SIMD or data-level parallelism.

One instruction operates on multiple data items.



MMX (MultiMedia eXtension), first SIMD by Intel Pentium processors (introduced 1997). Only on Integers.

3D Now! AMD, included single-precision floating-point (1998)

SSE (Streaming SIMD Extension) introduced by Intel in Pentium III (year 1999). Included single-precision FP.

AVX (Advanced Vector Extension), supported by both Intel and AMD (processors available in 2011). Added support for 256 bits and double-precision FP.

NEON multimedia extension for ARMv7 and ARMv8
(32 registers 8 bytes wide or 16 registers 16 bytes wide)

David Broman
dbro@kth.se

Part I

SIMD, Multithreading,
and GPUs

Part II

MIMD, Multicore,
and Clusters

Part III

Parallelization
in Practice

8



Streaming SIMD Extension (SSE) and Advanced Vector Extension (AVX)

In SSE (and the later version SSE2), assembly instructions are using two-operand format.

`addpd %xmm0, %xmm4`

meaning: $\%xmm4 = \%xmm4 + \%xmm0$
Note the reversed order.

Registers (e.g. `%xmm4`) are 128-bits in SSE/SSE2.

Added the “v” for vector to distinguish AVX from SSE and renamed registers to `%ymm` that are now 256-bit

“pd” means Packed Double precision FP. It can operate on as many FP that fits in the register

`vaddpd %ymm0, %ymm1, %ymm4`
`vmovapd %ymm4, (%r11)`

Question: How many FP additions does `vaddpd` perform in parallel?

Answer: 4

Moves the result to the memory address stored in `%r11` (a 64-bit register). Stores the four 64-bit FP in consecutive order in memory.

AVX introduced three-operand format
Meaning: $\%ymm4 = \%ymm0 + \%ymm1$

David Broman
dbro@kth.se

Part I

SIMD, Multithreading,
and GPUs

Part II

MIMD, Multicore,
and Clusters

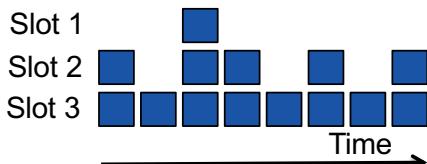
Part III

Parallelization
in Practice



Recall the idea of a multi-issue uniprocessor

Thread A Thread B Thread C



Typically, all functional units cannot be fully utilized in a single-threaded program (white space is unused slot/functional unit).

David Broman
dbro@kth.se



Part I
SIMD, Multithreading,
and GPUs

Part II
MIMD, Multicore,
and Clusters

Part III
Parallelization
in Practice

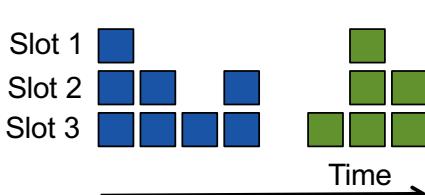


Hardware Multithreading

In a **multithreaded processor**, several hardware threads share the same functional units.

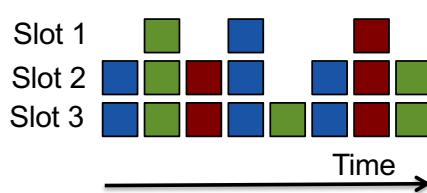
Thread A Thread B Thread C

The purpose of multithreading is to hide latencies and avoid stalls due to cache misses etc.



Coarse-grained multithreading, switches threads only at costly stalls, e.g., last-level cache misses.

Cannot overcome throughput losses in short stalls.



Fine-grained multithreading, switches between hardware threads every cycle. Better utilization.

David Broman
dbro@kth.se



Part I
SIMD, Multithreading,
and GPUs

Part II
MIMD, Multicore,
and Clusters

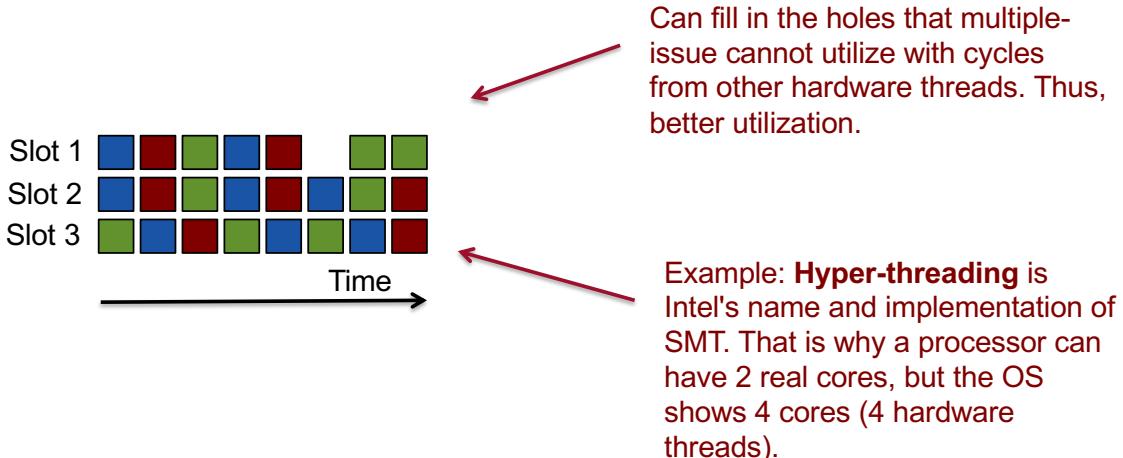
Part III
Parallelization
in Practice



Simultaneous multithreading (SMT)

Simultaneous multithreading (SMT) combines multithreading with a multiple-issue, dynamically scheduled pipeline.

Thread A Thread B Thread C



David Broman
dbro@kth.se



Part I
SIMD, Multithreading,
and GPUs

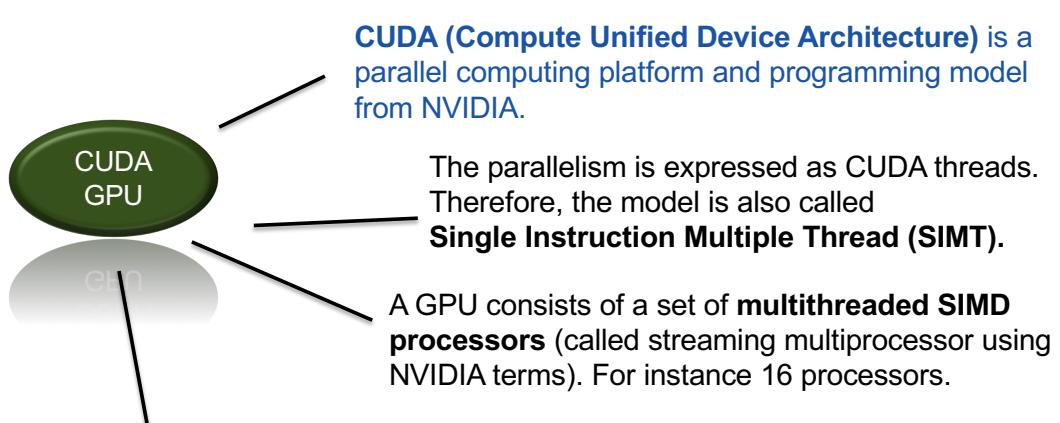
Part II
MIMD, Multicore,
and Clusters

Part III
Parallelization
in Practice



Graphical Processing Units (GPUs)

A **Graphical Processing Unit (GPU)** utilizes multithreading, MIMD, SIMD, and ILP. The main form of parallelism that can be used is data-level parallelism.



The main idea is to execute a massive number of threads and to use **multithreading** to hide latency. However, the latest GPUs also include caches.

David Broman
dbro@kth.se



Part I
SIMD, Multithreading,
and GPUs

Part II
MIMD, Multicore,
and Clusters

Part III
Parallelization
in Practice



Part II

MIMD, Multicore, and Clusters



Acknowledgement: The structure and several of the good examples are derived from the book "Computer Organization and Design" (2014) by David A. Patterson and John L. Hennessy

David Broman
dbro@kth.se

Part I
SIMD, Multithreading,
and GPUs

Part II
MIMD, Multicore,
and Clusters

Part III
Parallelization
in Practice



Shared Memory Multiprocessor (SMP)

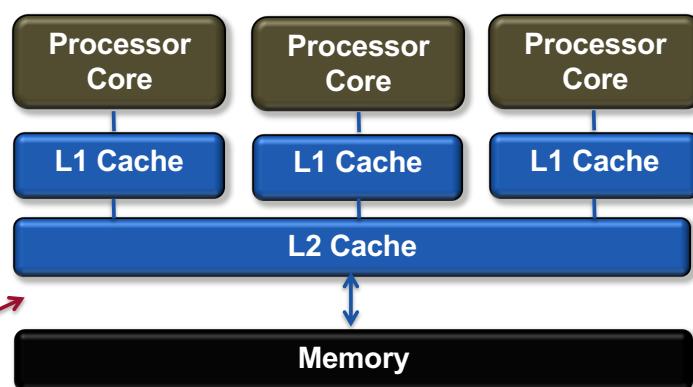
A Shared Memory Multiprocessor (SMP) has a *single physical address space* across all processors.

An SMP is almost always the same as a multicore processor.

In a **uniform memory access (UMA)** multiprocessor, the latency of accessing memory does not depend on the processor.

In a **nonuniform memory access (NUMA)** multiprocessor, memory can be divided between processor and result in different latencies.

Processors (cores) in a SMP communicate via **shared memory**.



Alternative: Network on Chip (NoC)

David Broman
dbro@kth.se

Part I
SIMD, Multithreading,
and GPUs

Part II
MIMD, Multicore,
and Clusters

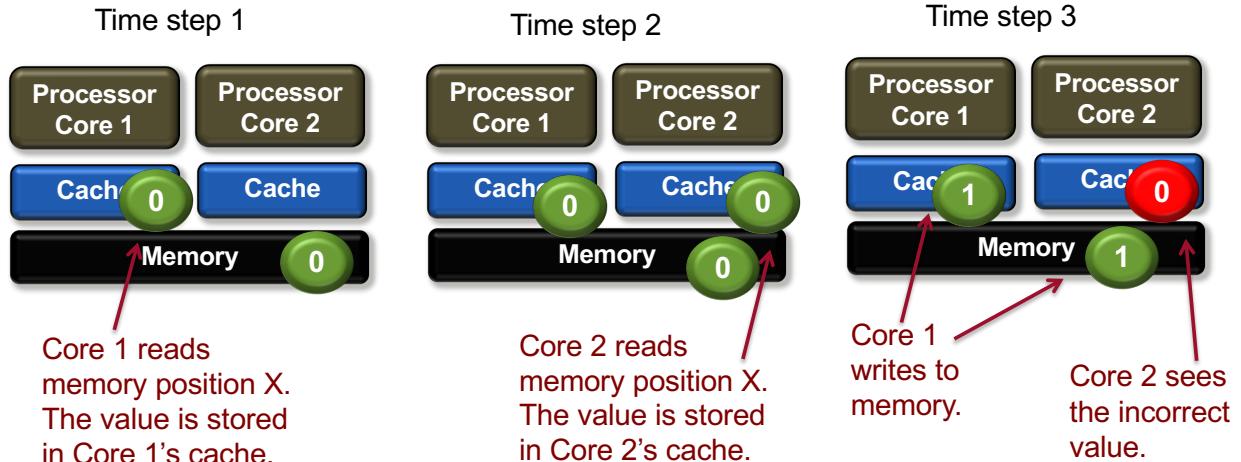
Part III
Parallelization
in Practice



Cache Coherence

Different cores' local caches could result in that different cores see different values for the same memory address.

This is called the **cache coherency** problem.



David Broman
dbro@kth.se

Part I
SIMD, Multithreading,
and GPUs

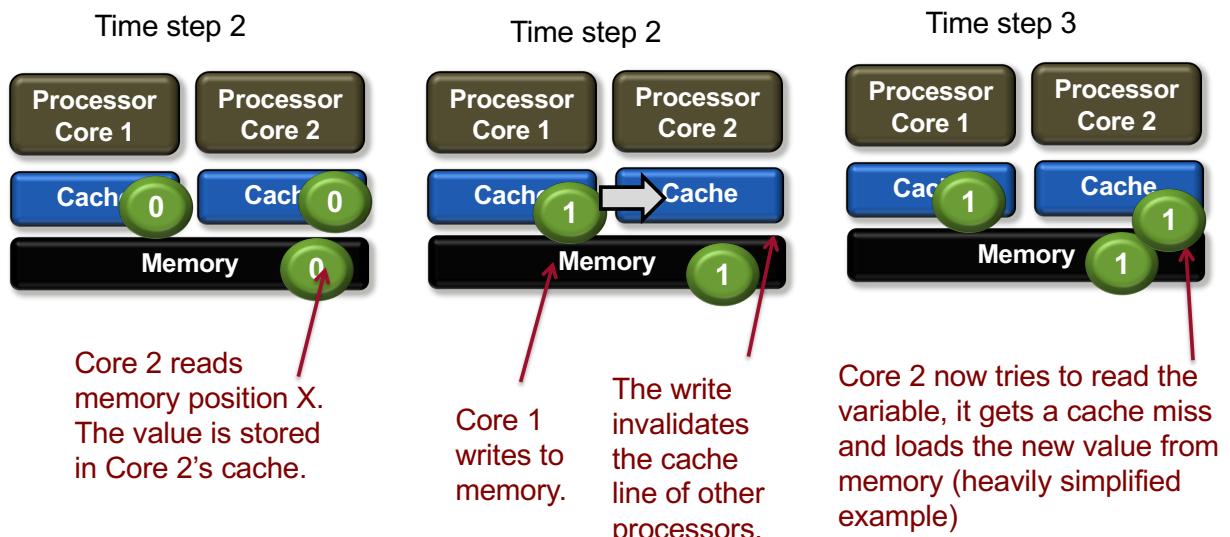
Part II
MIMD, Multicore,
and Clusters

Part III
Parallelization
in Practice



Snooping Protocol

Cache coherence can be enforced using a cache coherence protocol. For instance a *write invalidate protocol*, such as the **snooping protocol**.



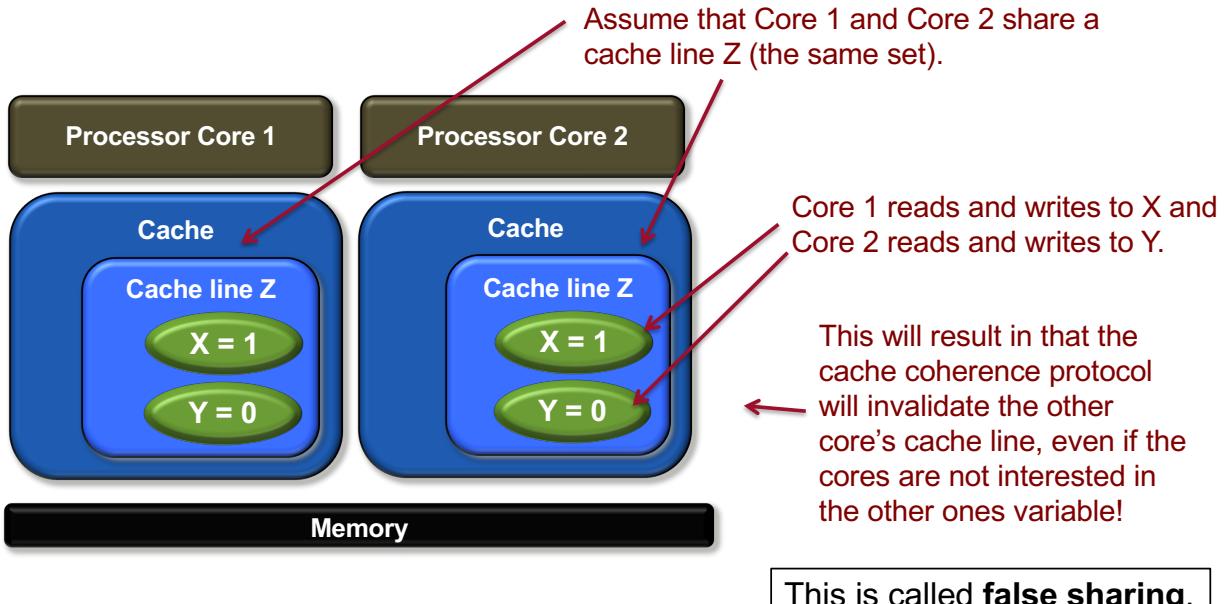
David Broman
dbro@kth.se

Part I
SIMD, Multithreading,
and GPUs

Part II
MIMD, Multicore,
and Clusters

Part III
Parallelization
in Practice

False Sharing



David Broman
dbro@kth.se

Part I
SIMD, Multithreading,
and GPUs

Part II
MIMD, Multicore,
and Clusters

Part III
Parallelization
in Practice



Processes, Threads, and Cores

A modern **operating system (OS)** can execute several **processes** concurrently.

A **process context** includes its own virtual memory space, IO files, read-only code, heap, shared library, process id (PID) etc.

Concurrent threads are **scheduled** by the OS to execute in **parallel** on different cores.

Note: All threads share the process context, including virtual memory etc.

Each process can have N number of **concurrent threads**. The **thread context** includes thread ID, stack, stack pointer, program counter etc.



David Broman
dbro@kth.se

Part I
SIMD, Multithreading,
and GPUs

Part II
MIMD, Multicore,
and Clusters

Part III
Parallelization
in Practice

POSIX threads (pthreads) is a common way of programming concurrency and utilizing multicores for parallel computation.

```
#include <stdio.h>
#include <pthread.h>

volatile int counter = 0;

void *count(void *data) {
    int i;
    int max = *((int*)data);
    for(i=0; i<max; i++)
        counter++;
    pthread_exit(NULL);
}
```

Creates two threads, each is counting a shared variable.

```
int main(){
    pthread_t tid1, tid2;
    int max;
    max = 40000;
    pthread_create(&tid1, NULL, count, &max);

    max = 60000;
    pthread_create(&tid2, NULL, count, &max);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("counter = %d\n", counter);
    pthread_exit(NULL);
}
```

Exercise: What is the output?

Answer: Different values each time...

David Broman
dbro@kth.se

Part I
SIMD, Multithreading,
and GPUs

Part II
MIMD, Multicore,
and Clusters

Part III
Parallelization
in Practice



Semaphores

A **semaphore** is a global variable that can hold a nonnegative integer value. It can only be changed by the following two operations.



P(s): If $s > 0$, then decrement s and return.
If $s = 0$, then wait until $s > 0$, then decrement s and return.



V(s): Increment s .

Note that the check and return of **P(s)** and increment of **V(s)** must be **atomic**, meaning that appears to be “instantaneously”.

Semaphores were invented by Edsger Dijkstra, who was originally from the Netherlands. P and V is supposed to stand for **Prolaag** (probeer te verlagen, “try to reduce”) and **Verhogen** (*increase*).

David Broman
dbro@kth.se

Part I
SIMD, Multithreading,
and GPUs

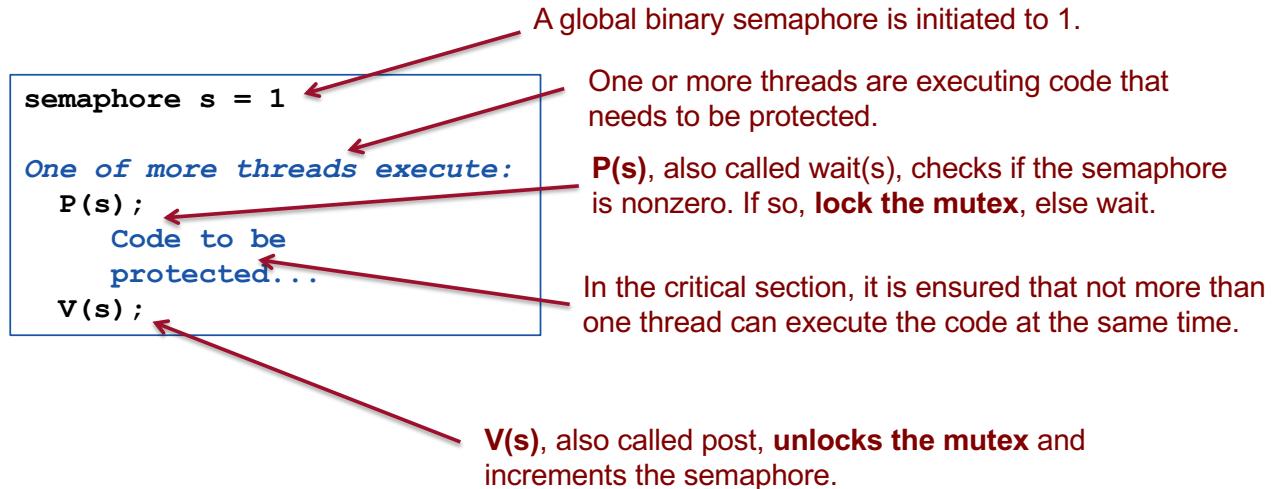
Part II
MIMD, Multicore,
and Clusters

Part III
Parallelization
in Practice



Mutex

A semaphore can be used for mutual exclusion, meaning that only one thread can access a particular resource at the same time. Such a **binary semaphore** is called a **mutex**.



David Broman
dbro@kth.se

Part I
SIMD, Multithreading,
and GPUs

Part II
MIMD, Multicore,
and Clusters

Part III
Parallelization
in Practice



Programming with Threads and Shared Variables with Mutexes

22
E

A Mutex (binary semaphore starting at 1)

Problem. We update the value max, that is also shared...

```
volatile int counter = 0;
pthread_mutex_t mutex;

void *count(void *data) {
    int i;
    int max = *((int*)data);
    for(i=0; i<max; i++) {
        pthread_mutex_lock(&mutex);
        counter++;
        pthread_mutex_unlock(&mutex);
    }
    pthread_exit(NULL);
}
```

Exercise: Is it correct this time?

```
int main(){
    pthread_t tid1, tid2;
    int max;

    pthread_mutex_init(&mutex, NULL);
    max = 40000;
    pthread_create(&tid1, NULL, count, &max);
    max = 60000;
    pthread_create(&tid2, NULL, count, &max);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("counter = %d\n", counter);
    pthread_mutex_destroy(&mutex);
    pthread_exit(NULL);
}
```

David Broman
dbro@kth.se

Part I
SIMD, Multithreading,
and GPUs

Part II
MIMD, Multicore,
and Clusters

Part III
Parallelization
in Practice

Correct solution...

```

volatile int counter = 0;
pthread_mutex_t mutex;

void *count(void *data) {
    int i;
    int max = *((int*)data);
    for(i=0; i<max; i++){
        pthread_mutex_lock(&mutex);
        counter++;
        pthread_mutex_unlock(&mutex);
    }
    pthread_exit(NULL);
}

```

```

int main(){
    pthread_t tid1, tid2;
    int max1 = 40000;
    int max2 = 60000;

    pthread_mutex_init(&mutex, NULL);
    pthread_create(&tid1, NULL, count, &max1);
    pthread_create(&tid2, NULL, count, &max2);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    printf("counter = %d\n", counter);
    pthread_mutex_destroy(&mutex);
    pthread_exit(NULL);
}

```

Simple solution. Use different variables.

David Broman
dbro@kth.se

Part I
SIMD, Multithreading,
and GPUs

Part II
MIMD, Multicore,
and Clusters

Part III
Parallelization
in Practice

Clusters and Warehouse Scale Computers

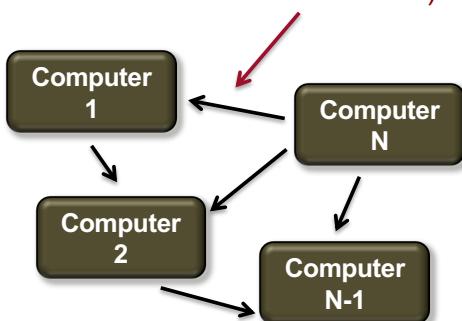


Photo by Robert Harker

A **cluster** is a set of computers that are connected over a local area network (LAN). May be viewed as one large multiprocessor.

Warehouse-Scale Computers are very large cluster that can include 100 000 servers that act as one giant computer (e.g., Facebook, Google, Apple).

Clusters do not communicate over shared memory (as for SMP) but are using **message passing**.



MapReduce is a programming model that is popular for batch processing.

1. **Map** applies a programmer defined function on all data items.
2. **Reduce** collects the output and collapse the data using another programmer defined function.

The map step is highly parallel. The reduce stage may be parallelized to some extent.

David Broman
dbro@kth.se

Part I
SIMD, Multithreading,
and GPUs

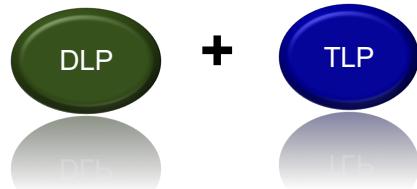
Part II
MIMD, Multicore,
and Clusters

Part III
Parallelization
in Practice



Part III

Parallelization in Practice



David Broman
dbro@kth.se

Part I
SIMD, Multithreading,
and GPUs

Part II
MIMD, Multicore,
and Clusters

 **Part III**
Parallelization
in Practice



General Matrix Multiplication (GEMM)

Simple matrix multiplication

Uses matrix size n as a parameter and single dimension for performance.

```
void dgemm(int n, double* A, double* B, double* C) {
    for(int i = 0; i < n; ++i)
        for(int j = 0; j < n; ++j){
            double cij = C[i+j*n];
            for(int k = 0; k < n; k++)
                cij += A[i+k*n] * B[k+j*n];
            C[i+j*n] = cij;
        }
}
```

David Broman
dbro@kth.se

Part I
SIMD, Multithreading,
and GPUs

Part II
MIMD, Multicore,
and Clusters

 **Part III**
Parallelization
in Practice



Parallelizing GEMM

Unoptimized

Unoptimized C version (previous page). Using one core.

1.7 GigaFLOPS (32x32)
0.8 GigaFLOPS (960x960)

SIMD

Use AVX instructions `vaddpd` and `vmulpd` to do 4 double precision floating-point operations in parallel.

6.4 GigaFLOPS (32x32)
2.5 GigaFLOPS (960x960)

ILP

AVX + unroll parts of the loop, so that the multiple-issue, out-of-order processor have more instructions to schedule.

14.6 GigaFLOPS (32x32)
5.1 GigaFLOPS (960x960)

Cache

AVX + unroll + blocking (dividing the problem into submatrices). This avoids cache misses.

13.6 GigaFLOPS (32x32)
12.0 GigaFLOPS (960x960)

Multi-core

AVX + unroll + blocking + multi core (multithreading using OpenMP)

23 GigaFLOPS (960x960, 2 cores)
44 GigaFLOPS (960x960, 4 cores)
174 GigaFLOPS (960x960, 16 cores)

Experiment by P&H on a 2.6GHz Intel Core i7 with Turbo mode turned off.

For details see P&H, 5th edition, sections 3.8, 4.12, 5.14, and 6.12

David Broman
 dbro@kth.se

Part I

SIMD, Multithreading,
 and GPUs

Part II

MIMD, Multicore,
 and Clusters

Part III

Parallelization
 in Practice



Future perspective: MIMD, SIMD, ILP, and Caches

“For x86 computers, we expect to see two additional cores per chip every two years and the SIMD width to double every four years.”

Hennessy & Patterson, Computer Architecture – A Quantitative Approach, 5th edition, 2013 (page 263)



We must understand and utilize **both MIMD** and **SIMD** to gain maximal speedups in the future, although MIMD (multicore) has gained much more attention lately.

The previous example showed that **the way** we program for **ILP** and **caches**, also matters significantly.

David Broman
 dbro@kth.se

Part I

SIMD, Multithreading,
 and GPUs

Part II

MIMD, Multicore,
 and Clusters

Part III

Parallelization
 in Practice



It's about time to... ...relax



David Broman
dbro@kth.se

Part I
SIMD, Multithreading,
and GPUs

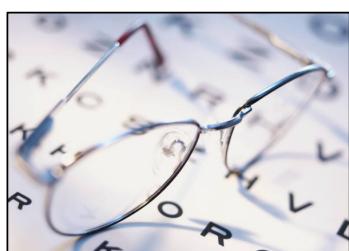
Part II
MIMD, Multicore,
and Clusters

Part III
Parallelization
in Practice



Reading Guidelines

Module 6: Parallel Processors and Programs



Lecture 12: Parallelism, Concurrency, Speedup, and ILP

- H&H Chapter 1.8, 3.6, 7.8.3-7.8.5
- P&H5 Chapters 1.7-1.8, 1.10, 4.10, 6.1-6.2
or P&H4 Chapters 1.5-1.6, 1.8, 4.10, 7.1-7.2

Lecture 13: SIMD, MIMD, and Parallel Programming

- H&H Chapter 7.8.6-7.8.9
- P&H5 Chapters 2.11, 3.6, 5.10, 6.3-6.7
or P&H4 Chapters 2.11, 3.6, 5.8, 7.3-7.7

Reading Guidelines

See the course webpage
for more information.

David Broman
dbro@kth.se

Part I
SIMD, Multithreading,
and GPUs

Part II
MIMD, Multicore,
and Clusters

Part III
Parallelization
in Practice



Summary

Some key take away points:

- **SIMD and GPUs** can efficiently parallelize problems that have data-level parallelism
- **MIMD, Multicores, and Clusters** can be used to parallelize problems that have task-level parallelism.
- In the future, we should try to **combine** and use both **SIMD** and **MIMD**!



Thanks for listening!

David Broman
dbro@kth.se

Part I
SIMD, Multithreading,
and GPUs

Part II
MIMD, Multicore,
and Clusters

Part III
Parallelization
in Practice