Computer Hardware Engineering

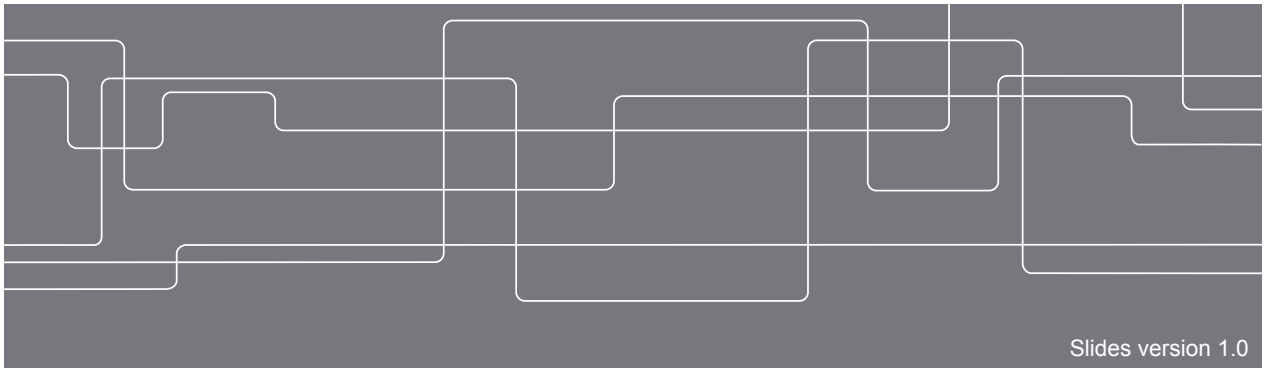
IS1200, spring 2015

Lecture 10: SIMD, MIMD, and Parallel Programming

David Broman

Associate Professor, KTH Royal Institute of Technology

Assistant Research Engineer, University of California, Berkeley



Course Structure

Module 1: Logic Design

L → L → DCÖ1 → DCÖ2 → Lab:dicom

Module 4: I/O Systems

L7 → E6 → E7 → Lab: nios2io

E9 → Lab: nios2int

Module 2: C and Assembly Programming

L1 → L2 → E1

L3 → E2 → E3 → Lab: nios2time

L4 → E4 → Home lab: C

Module 5: Memory Hierarchy

L8 → E8 → Home Lab: cache

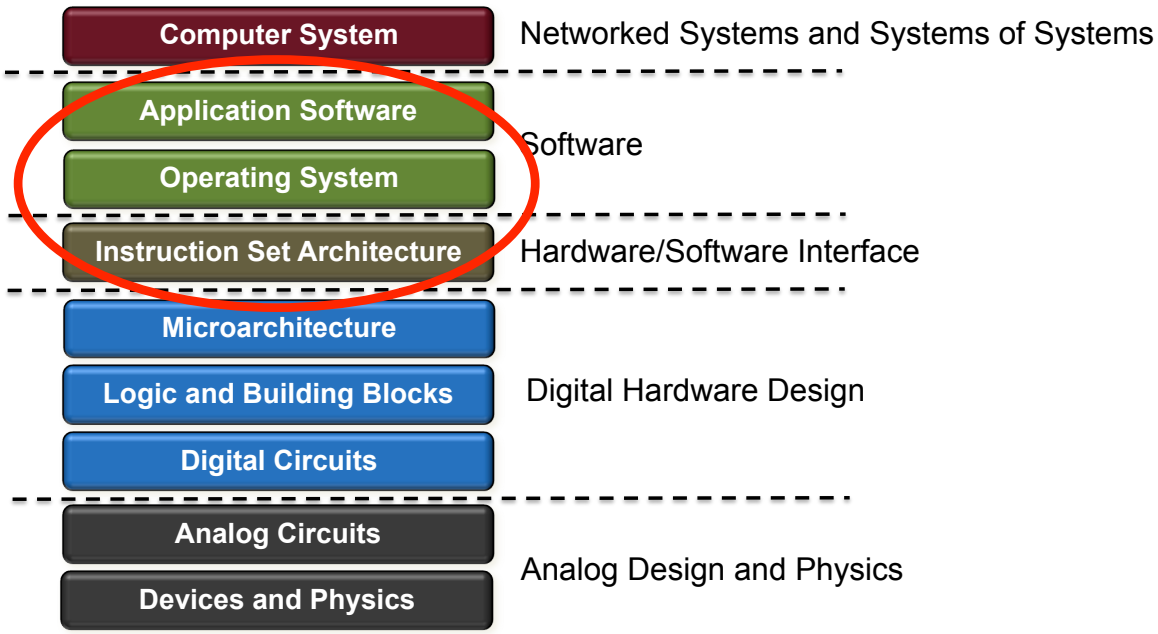
Module 3: Processor Design

L5 → L6 → E5

Module 6: Parallel Processors and Programs

L9 → L10 → E10

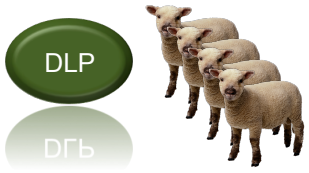
Abstractions in Computer Systems



Agenda

Part I

SIMD, Multithreading, and GPUs

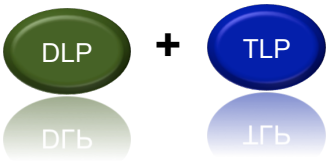


Part II

MIMD, Multicore, and Clusters

Part III

Parallelization in Practice

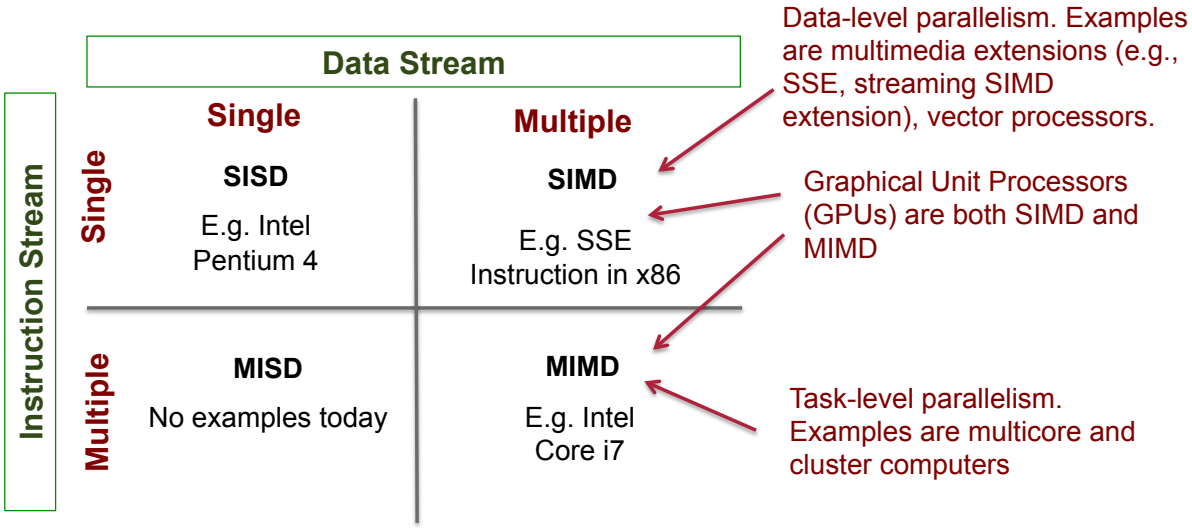


Part I

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

SISD, SIMD, and MIMD (Revisited)

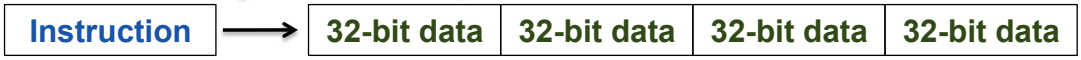


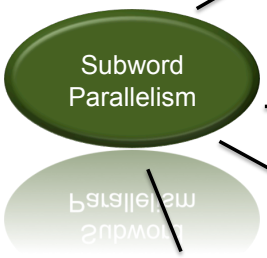
Subword Parallelism and Multimedia Extensions

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 (MutliMedia 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)

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 (Intel assembly in general)

Registers (e.g. `%xmm4`) are 128-bits in SSE/SEE2.

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)
```

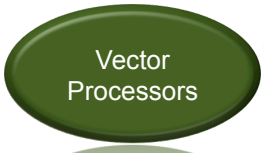
Question: How many FP additions does `vaddpd` perform in parallel?

Answer: 4

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

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.

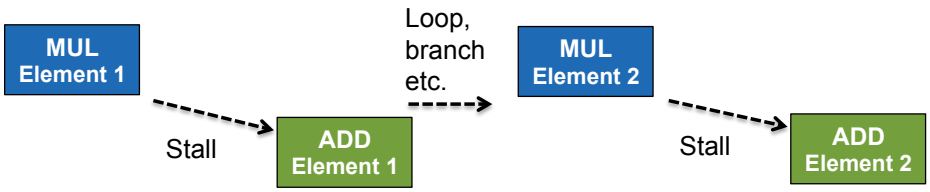
Vector Processors



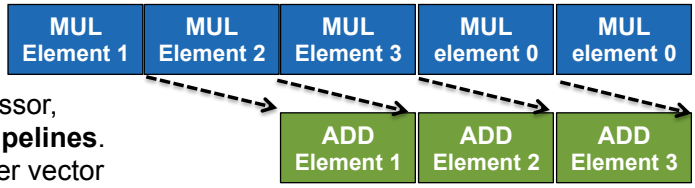
Older, but elegant version if SIMD. Dates back to the supercomputers in the 70s (Seymour Cray)

Vector processors have large **vector registers**, e.g., 32 vector registers, each having 64 64-bit elements.

Sequential code in a loop. Stall between instructions.



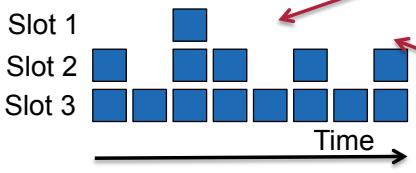
In a vector processor, operations are **pipelines**. Stall only once per vector operation.



Efficient use of memory and low instruction bandwidth give good energy characteristics.

Recall the idea of a multi-issue uniprocesor

Executes only one hardware thread (context switching must be done in software)



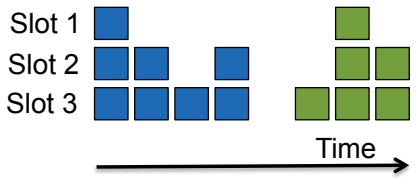
Typically, all functional units cannot be fully utilized in a single-threaded program (white space is unused slot/functional unit).

Hardware Multithreading

Thread A | Thread B | Thread C

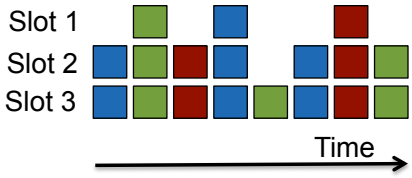
In a **multithreaded processor**, several hardware threads share the same functional units.

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.



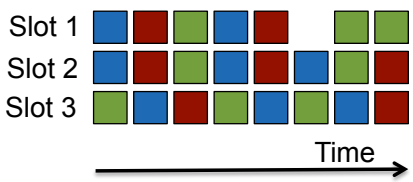
Fine-grained multithreading switches between hardware threads every cycle. Better utilization.

Simultaneous multithreading (SMT)

Thread A | Thread B | Thread C

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

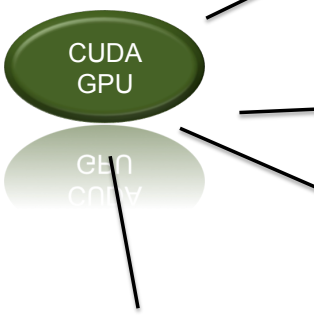
Can fill in the wholes that multiple-issue cannot utilize with cycles from other hardware threads. Thus, better utilization.



Example: **Hyper-threading** is Intel's name and implementation of SMT. That is why a processor can have 2 real cores, but the OS shows 4 cores (4 hardware threads).

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.



CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model from NVIDIA.

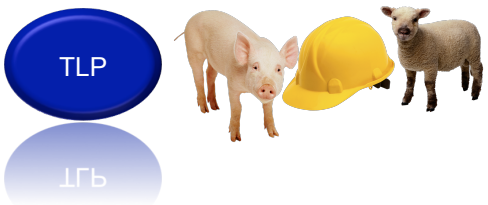
All parallelism are expressed as CUDA threads. Therefore, the model is also called **Single Instruction Multiple Thread (SIMT).**

A GPU consists of a set of **multithreaded SIMD processors** (called streaming multiprocessor using NVIDIA terms). For instance 16 processors.

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

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

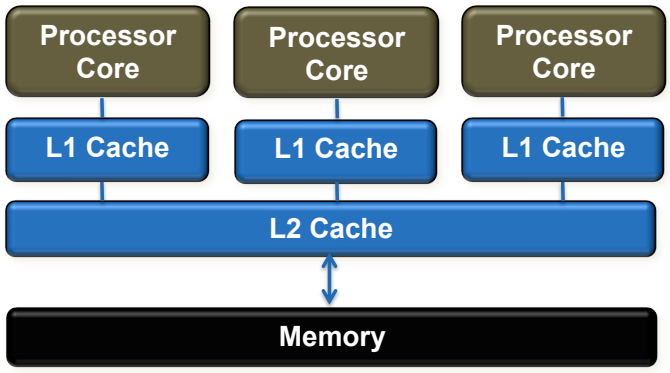
Shared Memory Multiprocessor (SMP)

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.

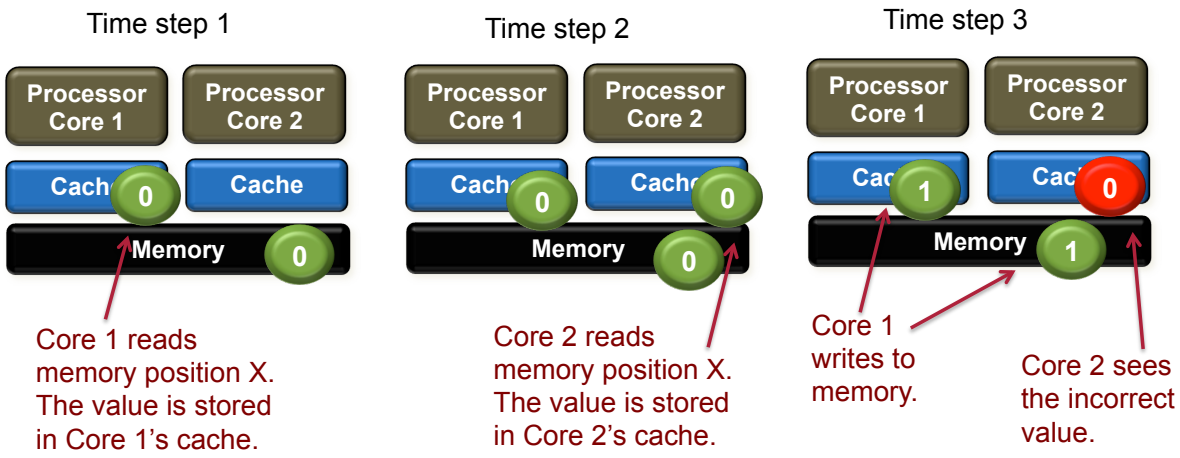


Processors (cores) in a SMP communicate via **shared memory**.

Cache Coherence

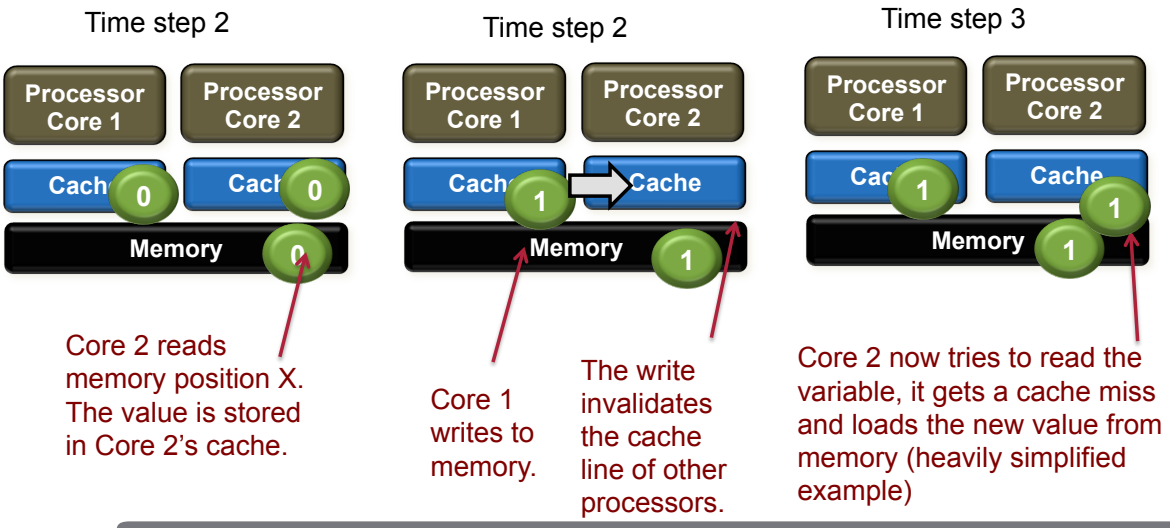
Different cores' local caches could result in that different cores see different values for the same memory address.

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

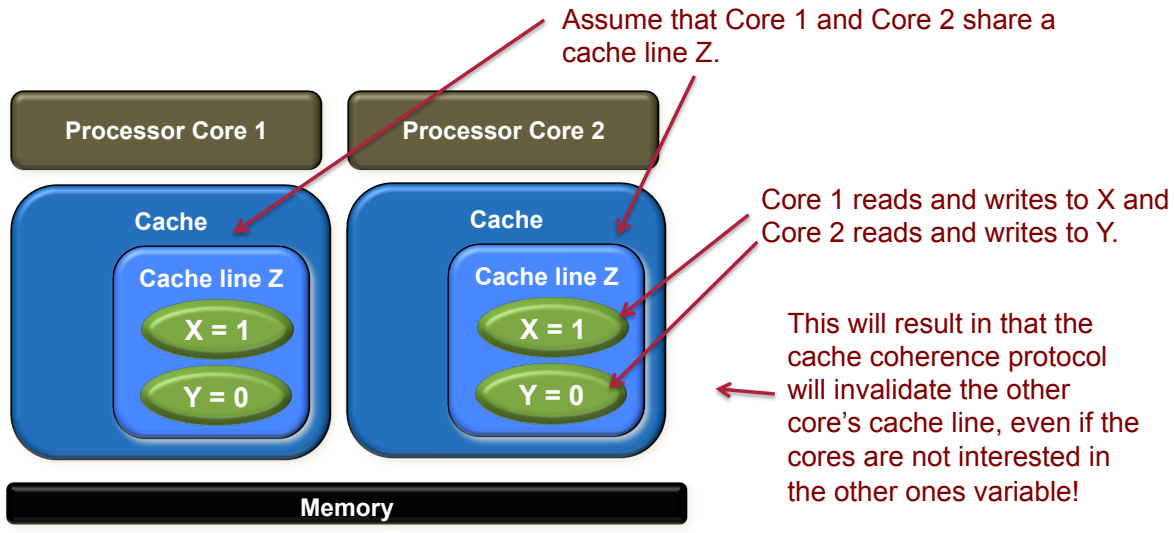


Snooping Protocol

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



False Sharing

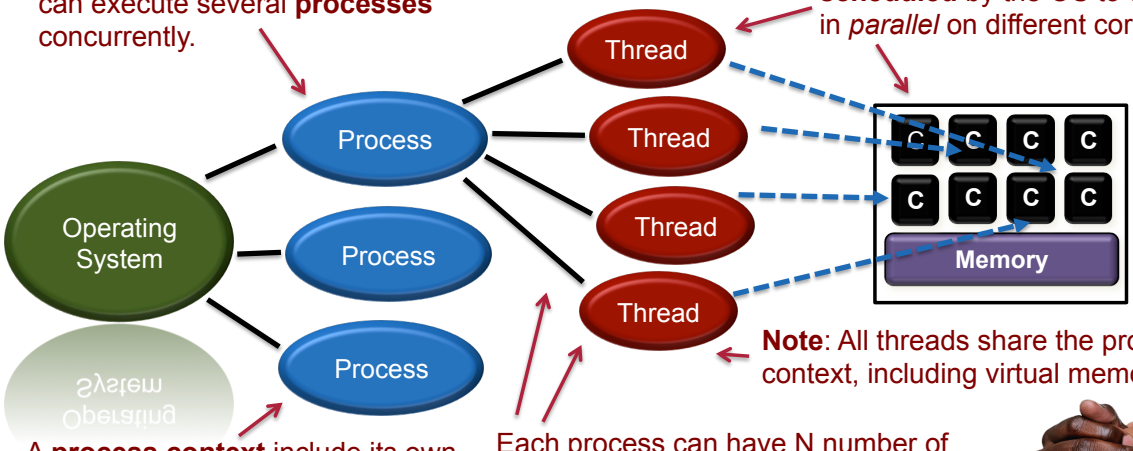


This is called **false sharing**.

Processes, Threads, and Cores

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

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.

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

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

Programming with Threads and Shared Variables

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

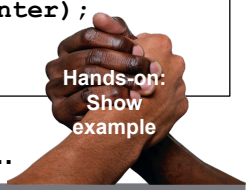
```
#include <stdio.h>
#include <pthread.h>

volatile int counter = 0;

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

```
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);
}
```

Creates two threads, each is counting a shared variable.

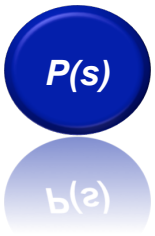
Exercise: What is the output?

Answer: Different values each time…

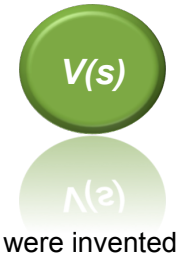


Semaphores

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 y Edsger Dijkstra, who was originally from the Neatherlands. P and V is supposed to come from the Dutch words **Proberen** (*to test*) and **Verlogen** (*to increment*).

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**.

A global binary semaphore is initiated to 1.

```
semaphore s = 1

One of more threads execute:
  P(s);
     Code to
     protected...
  V(s);
```

One or more threads are executing code that needs to be protected.

P(s), also called wait(s), checks if the semaphore is nonzero. If so, **lock the mutex**, else wait.

In the critical section, it is ensured that not more than one thread can execute the code at the same time.

V(s), also called post, **unlocks the mutex** and increments the semaphore.

Programming with Threads and Shared Variables with Semaphores

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

```
volatile int counter = 0;
sem_t *mutex;

void *count(void *data){
  int i;
  int max = *((int*)data);
  for(i=0; i<max; i++){
    sem_wait(mutex); /* P()*/
    counter++;
    sem_post(mutex); /* V(m)*/
  }
  pthread_exit(NULL);
}
```

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

  mutex = sem_open("/semaphore", O_CREAT,
                   O_RDWR, 1);
  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);
  sem_close(mutex);
  pthread_exit(NULL);
}
```

Exercise: Is it correct this time?

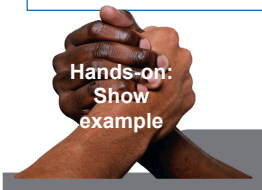
Programming with Threads and Shared Variables with Semaphores

Correct solution…

Simple solution. Use different variables.

```
volatile int counter = 0;
sem_t *mutex;

void *count(void *data){
  int i;
  int max = *((int*)data);
  for(i=0; i<max; i++){
    sem_wait(mutex); /*P()*/
    counter++;
    sem_post(mutex); /*V(m)*/
  }
  pthread_exit(NULL);
}
```

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

  mutex = sem_open("/semaphore", O_CREAT,
                   0777, 1);
  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);
  sem_close(mutex);
  pthread_exit(NULL);
}
```



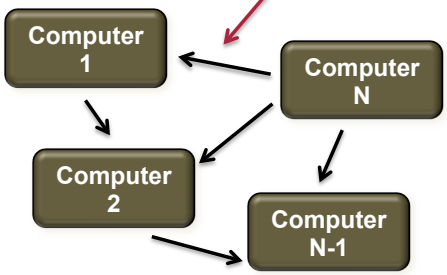
Clusters and Warehouse Scale Computers

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 using **message passing.**



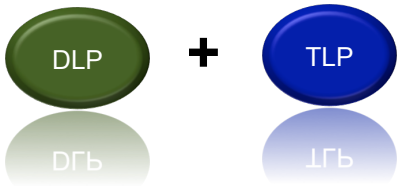
The **MapReduce** and **Hadoop** framework are popular for batch processing.

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.

Both tasks are highly parallel.

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;
    }
}
```


Parallelizing GEMM

Unoptimized	Unoptimzed 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)
Mutli-core	AVX + unroll + blocking + multi core (mutithreading 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 edtion, sections 3.8, 4.12, 5.14, and 6.12

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 **how** we program for **ILP** and **caches**, also matters significantly.

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!