System Design – Parallel Programming [05]

Hochschule
für Technik
Stuttgart

Marcus Deininger

WS 2021/22

Overview

- Motivation / Requirements
- Use Cases
- Threads
- Creating Threads
- Controlling Threads
- Master/Worker
- Producer/Consumer
- Pipeline

Motivation

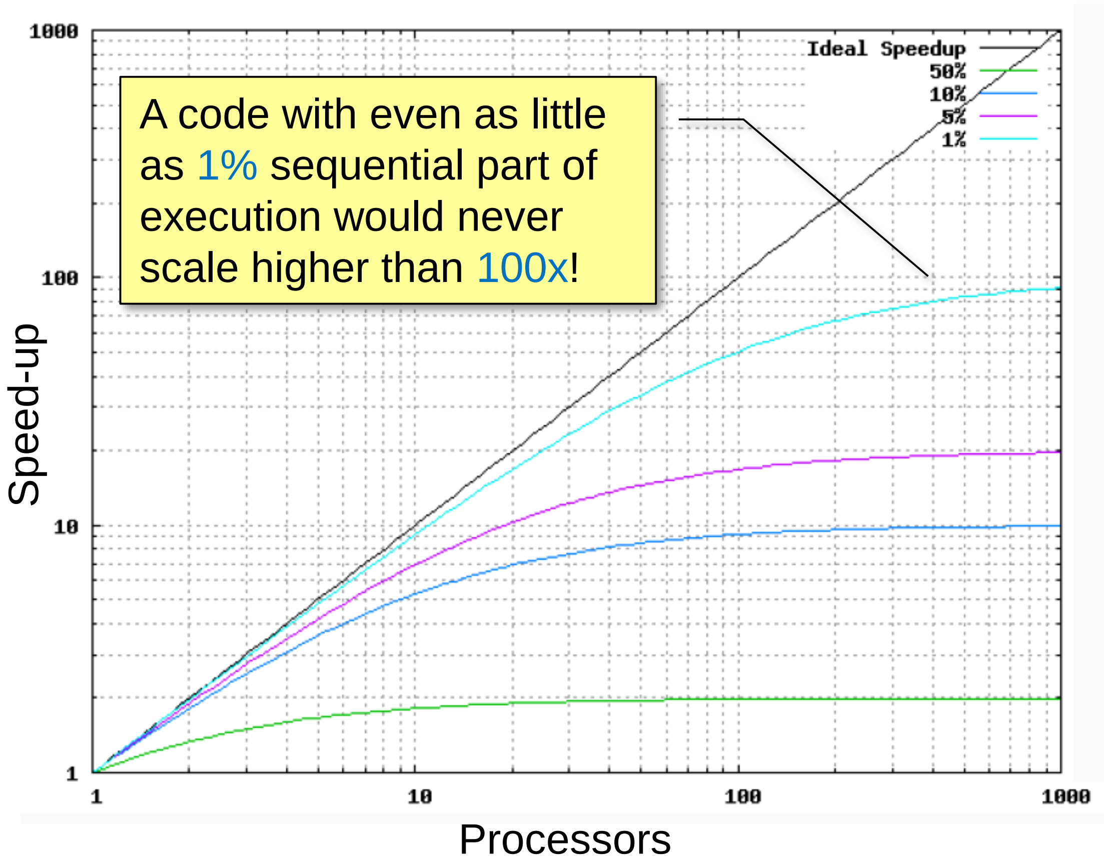
Parallel Programming

- to run programs in shorter period of time
- to solve with higher accuracy in given time
- to solve a computational problem at all (e.g. because of size of data)

Theoretically restricted by

- Amdahl's Law
- Gustafson's Law

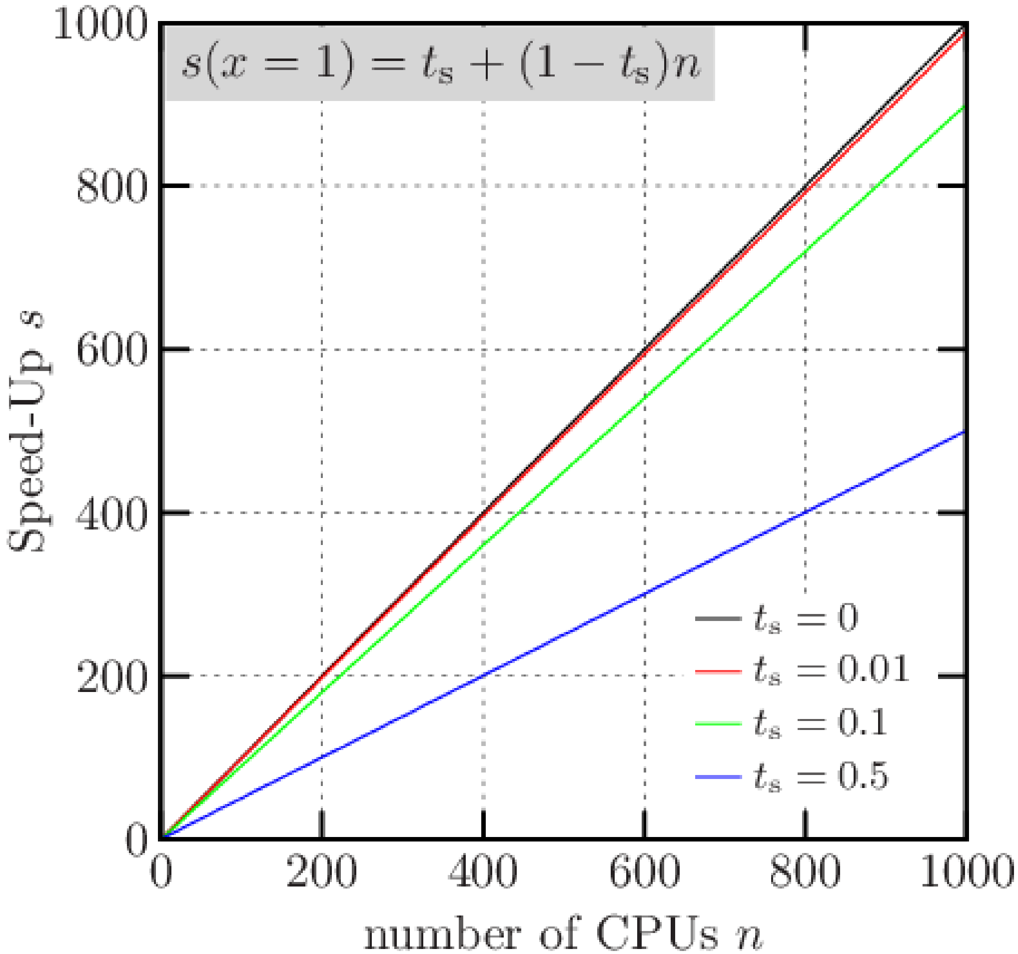
Amdahl's Law (fixed Problem Size)



- Code has a sequential (s) and a parallel (p) proportion, with $s + p = 1$
- Total execution Time:
  $$T = Ts + Tp$$
- Execution Time with n Processors: $T_n = Ts + \frac{Tp}{n}$
- Speed-up:

$$\frac{T}{T_n} = \frac{T}{Ts + \frac{Tp}{n}} = \frac{1}{s + \frac{1-s}{n}}$$

Gustafson's Law (fixed Problem Time)



- Code has a sequential (s) and a parallel (p) proportion, with $s + p = 1$
- Ideal execution time running in parallel on n Processors: $T_n = Ts + Tp$
- Execution time running this code sequentially:
  $$T = Ts + nTp$$
- Speed-up:
  $$\frac{T}{T_n} = \frac{Ts + nTp}{Ts + Tp}$$
  $$= s + n(s-1)$$

Figure: www.researchgate.net

Which Law is Right?

- Amdahl's Law
  - overly pessimistic
  - states that the possibilities are very limited
- Gustafson's Law
  - overly optimistic
  - states that the possibilities are nearly linear
- the truth is somewhere in the middle

Requirements for Parallelization

- Concurrency
  - multiple tasks can be done at the same time
  - data may be worked on independently (for a longer time)
- Scalability
  - if tasks are dependent on each other, there should be little communication and synchronization
- Locality
  - if data is exchanged (shared resources /explicit communication), it should be done rarely – i.e. keep processing local.

The worst algorithm will have dependencies in every step, need to exchange all data between all involved processes.

Processes and Threads

- Processes
  - separated and isolated tasks with different memory
  - heavy-weight, difficult communication
- Threads
  - independent tasks sharing memory and working on common data
  - light-weight, easy communication
  - Java natively supports Threads $\rightarrow$ we will concentrate on Java-Threads

Use Cases

Thread Model	Description
Delegation or Master/Worker	■ Central Master-Thread creates Worker-Treads ■ Master assigs each worker tasks ■ Master may wait until workers complete
Producer-Consumer	■ Producer-Threads produce data ■ Data is stored in a shared memory ■ Consumer-Threads consume the data
Pipeline or Pipes & Filters	■ Processing is done in several stages ■ Each stage is Filter-Thread that performs work on a unit of input.
Peer-to-Peer	■ Peer-Threads have equal status and act on their own ■ working on shared data or private data

Creating Threads in Java

- Direct creation
  - ☑ implement a **`Runnable`** and start it through a Thread
  - create a **`Thread`**-subclass and start it directly
  - implement an anonymous adapter
  - ☑ supply a lambda-expression
  - implement a named adapter
- High-level creation
  - ☑ create an **`Executor`** and supply a **`Runnable`**
  - ☑ create an **`ExecutorService`** and supply a **`Callable`**

☑ *preferred approaches*

Direct Creation: A “Runnable” Class

Main-program

```
public class Target
        implements Runnable {

  public void run() {
    … //do something in parallel
  }
}
```

```
Target target = new Target();
Thread myThread = new Thread(target);
myThread.start();
```

this is the *one* task which should run in parallel

forks a new thread and starts “run()”

The Thread-class hosts the task.

Direct Creation: Lambda Expression

This is the *one* task which should run in parallel – it is **void** and has no parameters (just like “run”)

```
public void doSomething() {
    … //do something in parallel
}


…
Thread myThread = new Thread(() -> doSomething());
myThread.start();
```

forks a new thread and starts “doSomething()”

The Thread-class hosts the task.

The lambda-Expression

Typical Structures of `run()` or `doSomething()`

```
public void run(){
  while (true) {
    try {
      Thread.sleep(1000);
    } catch (InterruptedException ignored){}
    some actions;
  }
}
```

an endless loop running in the Thread after start

suspend for 1000 ms

this is the actual task, which should be performed

Low-Level-Control of Threads – 1

- start / (stop)
  - **`thread.start()`** for starting a thread
  - ~~**`thread.stop()`**~~ for stopping – deprecated!
- wait / release
  - **`thread.join()`** blocking wait until the thread has finished
  - **`thread.yield()`** releases the thread

Low-Level-Control of Threads – 2

- suspend/resume with the help of flags
  - **`anObject.wait()`** suspends an object, waits for **`notify()`**
  - **`anObject.notify()`** resumes an object
  - **`anObject.notifyAll()`** resumes all objects
  - **`Thread.sleep(time)`** suspends the current thread
  - **`thread.interrupt()`** sends an interrupt to an e.g. sleeping thread

Low-Level-Control of Threads – 3

`synchronized` – Method modifier

- a caller entering this method locks the whole object
- all other callers trying to access any other synchronized parts of this object have to wait
- the lock is released when the caller leaves the method or calls **`wait()`**
- typically used for exclusive data access (“race-conditions”)

`synchronized` – Block

- more fine-grained

```
synchronized(anObject){ // mostly this
    exclusive actions;
}
```

Controlling a Thread – 1

```
public class Controlled{

  private Thread thread = new Thread(() -> runInParallel());

  private boolean suspended = false;
  private long sleepTime = 1000;

  public void runInParallel() {
      while (true) {
        try {
          synchronized(this){
            while(suspended)
                  this.wait();
          }
         Thread.sleep(sleepTime);
        }catch (InterruptedException ignored){}
        some actions;
      }
}
  …
}
```

an endless loop running in parallel after start

if the thread wakes up from **`wait`**, he first checks his state and may go to **`wait`** again (if falsely woken up)

Sleep for a given time. This may be interrupted.

this is the actual task, which should be performed

Hochschule für Technik Stuttgart

Controlling a Thread – 2 (continued)

```
public class Controlled{
  …
  public void setSleepTime(long sleepTime) {
    thread.interrupt();
    this.sleepTime = sleepTime;
  }

  public synchronized void start(){
    if(!thread.isAlive())
      thread.start();
    if(suspended){
      suspended = false;
      this.notify();
    }
  }

  public synchronized void stop(){
    if(!suspended)
      suspended = true;
  }

  public boolean isRunning(){
    return thread.isAlive() & !suspended;
  }
  …
}
```

wake up from `sleep`

guarantee that this method is used by only one process

wake up from `wait`

send to `wait`

High-Level-Creation: Executors

- Setting up a Thread-Pool

  with a fixed number of threads

  ```
  Executor executor = Executors.newFixedThreadPool(3);
  ```

  or system-controlled

  ```
  Executor executor = Executors.newCachedThreadPool();
  ```

- Defining and starting the Threads

  ```
  List<Runnable> runnables = …

  for(Runnable runnable : runnables)
    executor.execute(runnable);
  ```

High-Level-Creation: Executor-Services

- Setting up a Thread-Pool

```
ExecutorService executor =
                    Executors.newCachedThreadPool();
```

- Defining and starting the Threads

```
Callable<Type> callable = () -> returnSomeType();
```

Future is a result available later, when the thread is finished

```
Future<Type> future = executor.submit(callable);
```

- Accessing the result

`future.isDone()` → **`true`** when the result is available

`Type value = future.get();` blocks until the result is available and retrieves it.

- Stopping the threads

```
executor.shutdown();
```

Hochschule für Technik Stuttgart

High-Level Access Control

- thread-safe data-structures
  - `AtomicInteger` with atomic operations
  - `AtomicIntegerArray`, `AtomicReference<V>`, …
- thread-safe collections
  - `Collections.`*`synchronized…`*`(…)` returns the respective thread-safe implementation

```
List<Type> l = new ArrayList<>();
List<Type> sl = Collections.synchronizedList(l);
```

Use Case: Master/Worker

- Divide the task into (independent) segments
- Low-Level
  - Master creates a `Thread` for each segment and starts it
  - Master waits with `join()` for each thread
  - If workers access/change shared data use `synchronized` methods
- High-Level
  - Master creates `Executor` or `ExecutorService`
  - assigns `Runnables` or `Callables` for each segment
  - retrieves results from `Futures`

Master/Worker – Sample

```java
public class Worker {

   private String name;
   private int result;

   public Worker(String name) {
      super();
      this.name = name;
   }

   public int getResult() {
      return result;
   }

   public void doSomething(int n) {
      result = 0;
      while(n > 0) {
         System.out.println(name);
         try {
            Thread.sleep(1000);
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
         n--;
         result++;
      }
   }
}
```

```java
public class Master {

   public static void main(String[] args)
                           throws Interrupt
      Worker w1 = new Worker("Worker 1");
      Worker w2 = new Worker("Worker 2");

      Thread t1 = new Thread(() -> w1.doSomething(5));
      Thread t2 = new Thread(() -> w2.doSomething(10));

      t1.start();
      t2.start();
      t1.join();
      t2.join();

      System.out.println("Result 1: " + w1.getResult());
      System.out.println("Result 2: " + w2.getResult());
   }
}
```

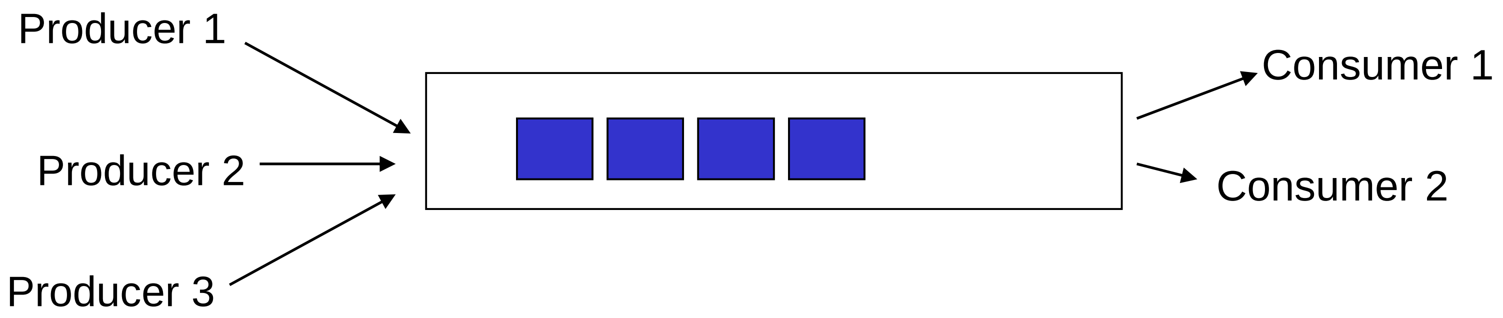
Assign work and start.

Wait for workers to finish.

Get results.

Use Case: Consumer/Producer

- a producer thread sends data into a queue
- a consumer thread takes the data from a queue
- a producer has to wait if the queue is full
- a consumer has to wait if the queue is empty



Consumer/Producer – Sample – 1

```java
public class Queue {

	public static final int MAXQUEUE = 5;
	private List<String> messages = new ArrayList<>();

	//called by the Producer
	public synchronized void putMessage(String message) {
		while(messages.size() >= MAXQUEUE)
			try {
				wait();
			} catch (InterruptedException e) {}
		messages.add(message);
		notifyAll();
	}

	//called by the Consumer
	public synchronized String getMessage() {
		while(messages.size() == 0)
			try {
				notifyAll();
				wait();
			} catch (InterruptedException e) {}
		String message = messages.remove(0);
		notifyAll();
		return message;
	}
}
```

This is the shared data: it is synchronized, not “threaded”

secured data access

resume a possible waiting producer

wait for a queue to be filled

resume possible producers, as there is now space in the queue

Hochschule für Technik Stuttgart

Consumer/Producer – Sample – 2

```
public class NamedProducer implements Runnable {

	private String name;
	private Queue queue;

	public NamedProducer(String name, Queue queue) {
		super();
		this.name = name;
		this.queue = queue;
	}


	@Override
	public void run() {
		while(true){
			queue.putMessage(new Date().toString());
			try {
				Thread.sleep(1000);
			} catch (InterruptedException e) {}
		}
	}

}
```

This is to make the producers / consumers run in parallel later on – otherwise there would be no sense in synchronizing the data.

The producer creates new products and puts it into the queue.

This is just for demo effect – to force some actual delay

Consumer/Producer – Sample – 3

```
public class NamedConsumer implements Runnable {

	private String name;
	private Queue queue;

	public NamedConsumer(String name, Queue queue) {
		super();
		this.name = name;
		this.queue = queue;
	}

	@Override
	public void run() {
		while(true){
			String message = queue.getMessage();
			System.out.println(name + " got message; " + message);
			try {
				Thread.sleep(2000);
			} catch (InterruptedException e) {}
		}
	}
}
```

The consumer receives products from the queue.

This is just for demo effect – to force some actual delay

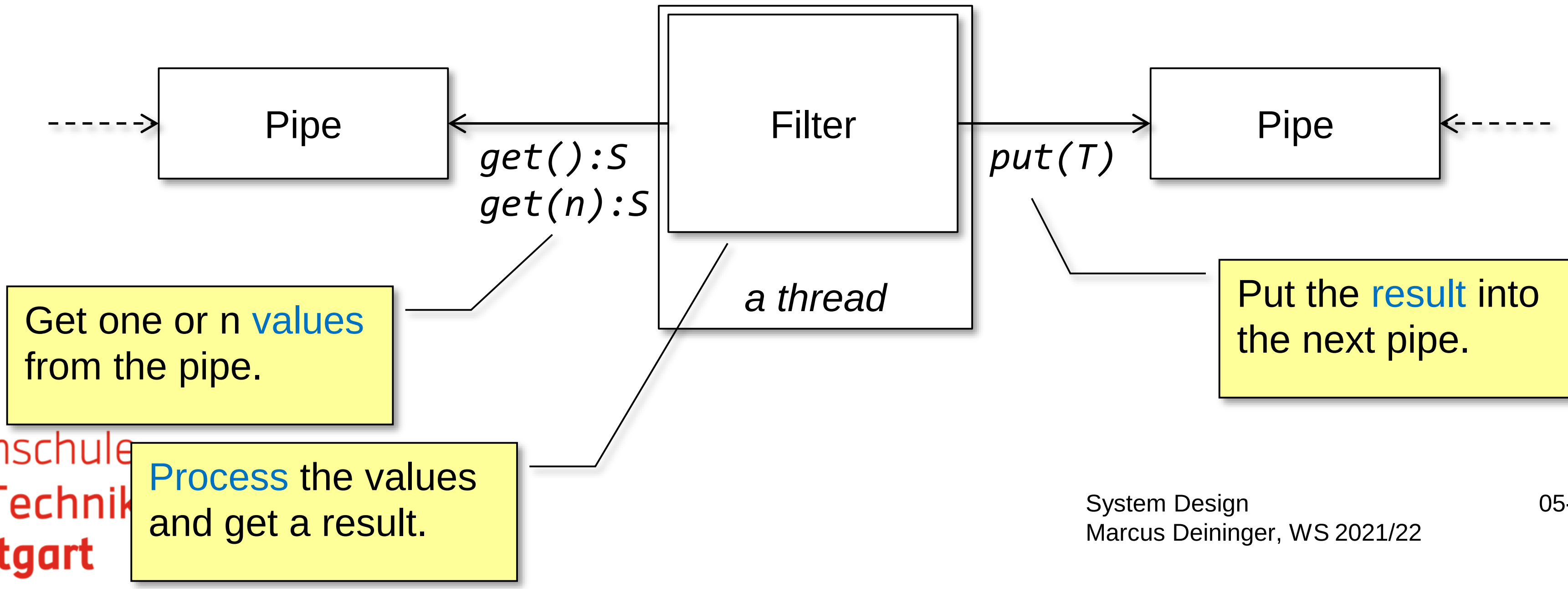
Controlling Data Access – 4

Main program for starting the threads.

```
public class Main {

      public static void main(String[] args){
            Queue queue = new Queue();

            NamedProducer producer1 = new NamedProducer("P1", queue);
            new Thread(producer1).start();
            NamedProducer producer2 = new NamedProducer("P2", queue);
            new Thread(producer2).start();
            NamedConsumer consumer1 = new NamedConsumer("C1", queue);
            new Thread(consumer1).start();
            NamedConsumer consumer2 = new NamedConsumer("C2", queue);
            new Thread(consumer2).start();
      }
}
```

Use Case: Pipeline

- A Pipe is a Queue with synchronized access
- A Filter is a common Consumer/Producer with
  - its own thread
  - an assigned input queue from which he consumes
  - an assigned output queue to which he produces



Pipeline – Sample (→ Exercise)

Converting Data from Source S to Target T

```
public class BasicPipe<T> implements Pipe<T>{

	private static final int MAXQUEUE = 5;
	private List<T> queue = new LinkedList<>();

	//called by the Producer
	@Override
	public synchronized void put(T t) {…}

	//called by the Consumer
	@Override
	public synchronized T get() {…}

	//called by the Consumer
	@Override
	public synchronized List<T> get(int n) {…}

	//called by the Consumer
	@Override
	public synchronized List<T> getAll(int n) {…}

	public synchronized List<T> getAll() {…}
}
```

```
public abstract class Filter<S, T> implements Runnable {

	protected Pipe<S> in = null;
	protected Pipe<T> out = null;
	protected boolean running = true;

	public boolean isRunning() {
		return running;
	}

	public void setIn(Pipe<S> in) {
		this.in = in;
	}

	public void setOut(Pipe<T> ou
		this.out = out;
	}

	public abstract void process();

	@Override
	public void run() {
		while(isRunning())
			process();
	}
}
```

The actual action on the data working on in and out

The Queue of Producer / Consumer

The Producer *and* Consumer

Hochschule für Technik Stuttgart

Pipeline – Sample (→ Exercise)

```
public class Pipeline {

	private List<Thread> threads;

	private Pipeline(List<Thread> threads) {
		this.threads = threads;
	}

	public static class Segment<S>{

		private List<Thread> threads;
		private Pipe<S> in = null;

		…

		public <T> Segment<T> filter(Filter<S, T> filter) {
			filter.in = in;
			Pipe<T> p = new BasicPipe<T>();
			filter.out = p;
			threads.add(new Thread(filter));
			Segment<T> next = new Segment<>(p, threads);
			return next;
		}
		…
	}

	…

	public void start() {
		for(Thread thread : threads)
			thread.start();
	}
}
```

The list of Threads hosting the filters

One pipeline-segment

Connect to previous pipe.

Create following pipe and connect

Host filter

Next segment with new “dangling” pipe

Start all threads.

Pipeline – Sample (→ Exercise)

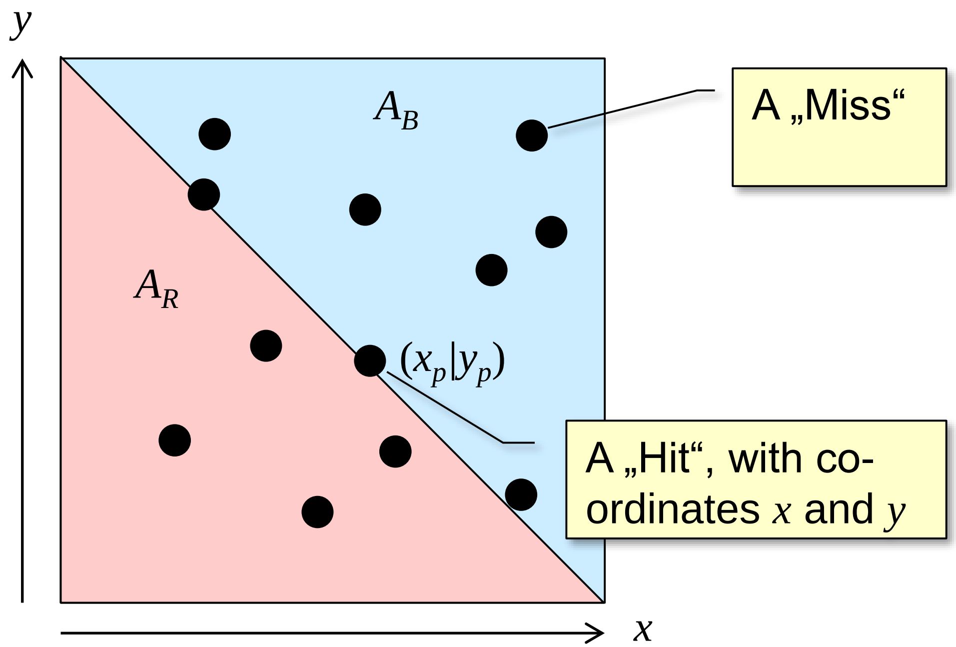
```
public class Sample {

   public static String keep(String s) {
        return s;
   }

   public static void main(String[] args) {
        Pipeline pl = Pipeline.source(new TextFileSource("test.txt"))
                                      .map(s -> keep(s))
                                      .drain(new OutputDrain<String>());
        pl.start();
   }
}
```

Hochschule
für Technik
Stuttgart

Use Case: Peer-to-Peer

- mixture of the previous use-cases
- not really discussed here

Geometrical Probability



Chances are 50%, that a random shot ends up in A_R