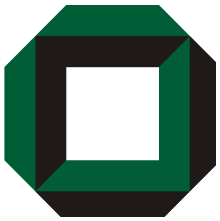


Universität Karlsruhe (TH)

Forschungsuniversität · gegründet 1825

High-Level Multicore Programming with XJava

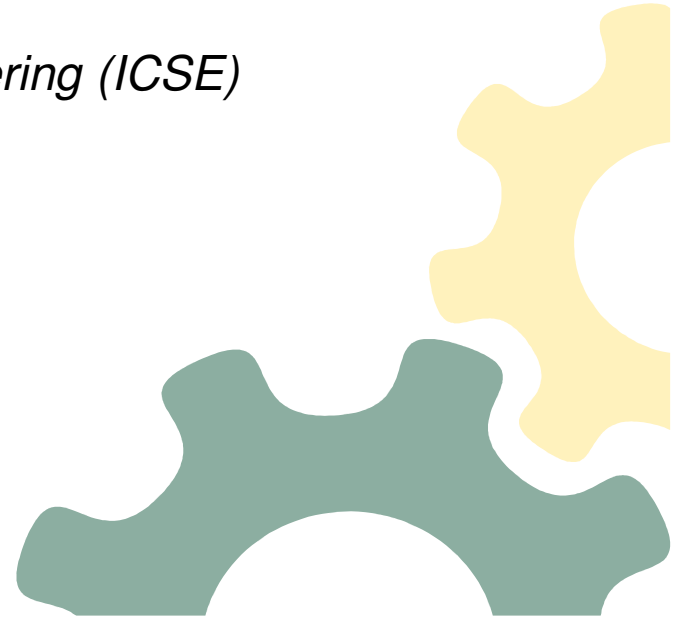
Frank Otto, Victor Pankratius, Walter F. Tichy

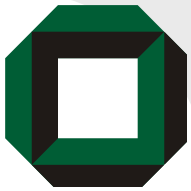
31st International Conference on Software Engineering (ICSE)
New Ideas and Emerging Results (NIER)

20 May 2009, Vancouver, Canada

Fakultät für **Informatik**

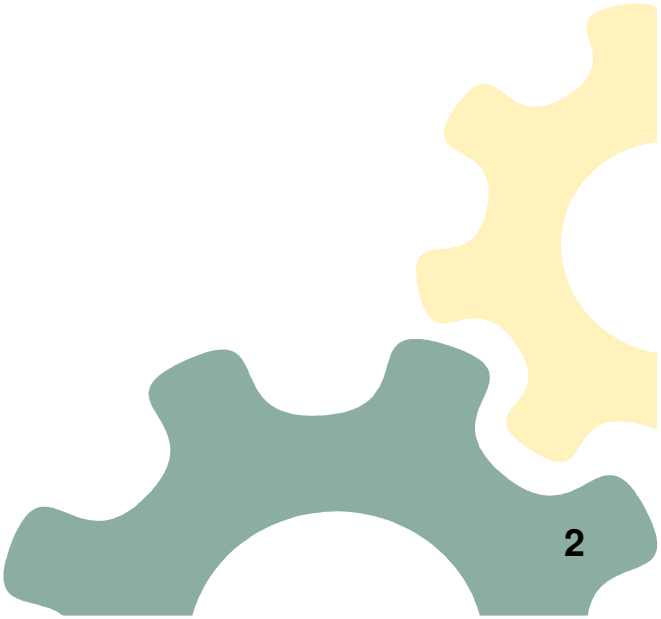
Lehrstuhl für Programmiersysteme



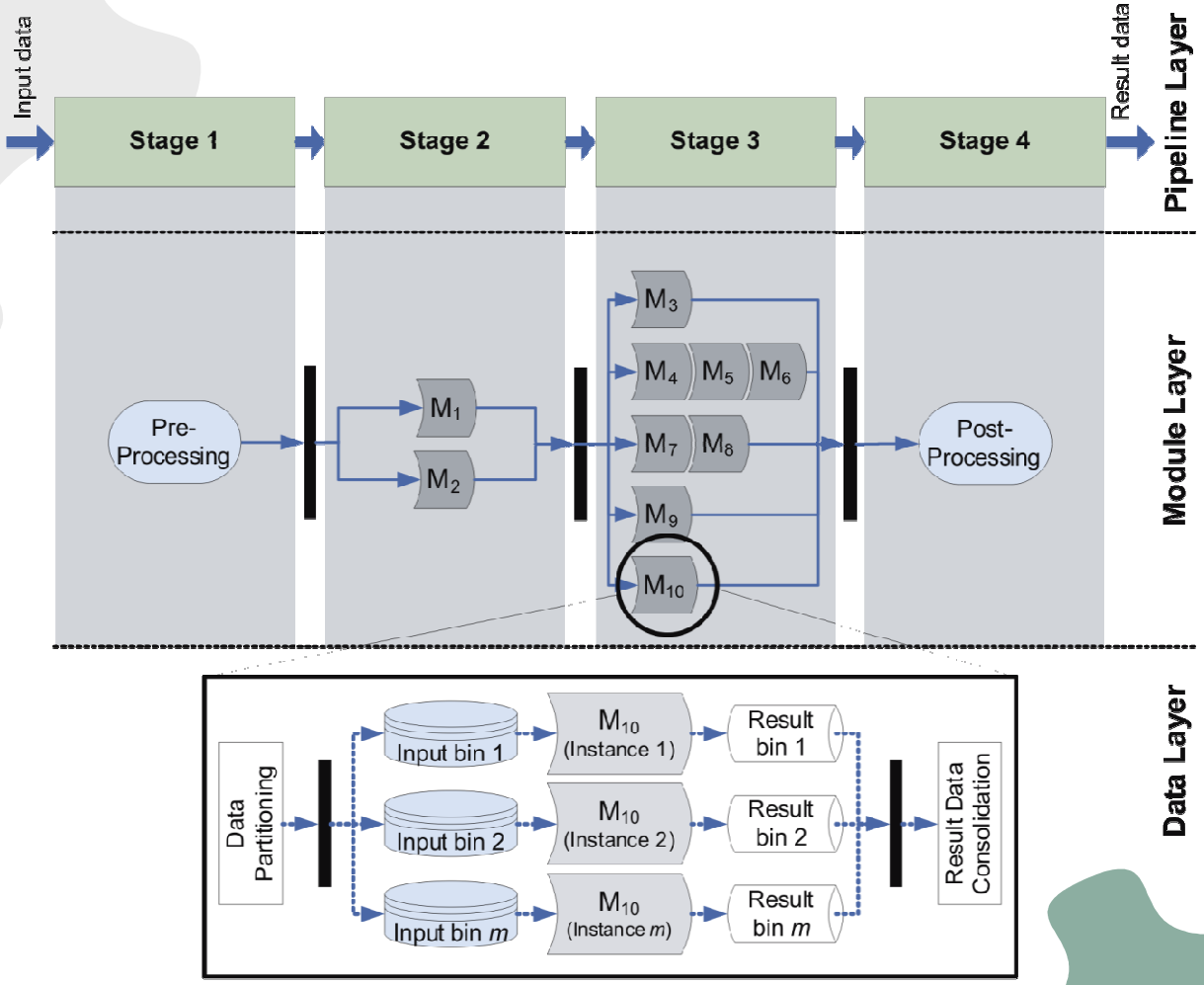


The Problem

- Clock rates are stagnating
  - „The free lunch is over“
- Multicore processors enter the mainstream
  - Renewed interest in parallel programming



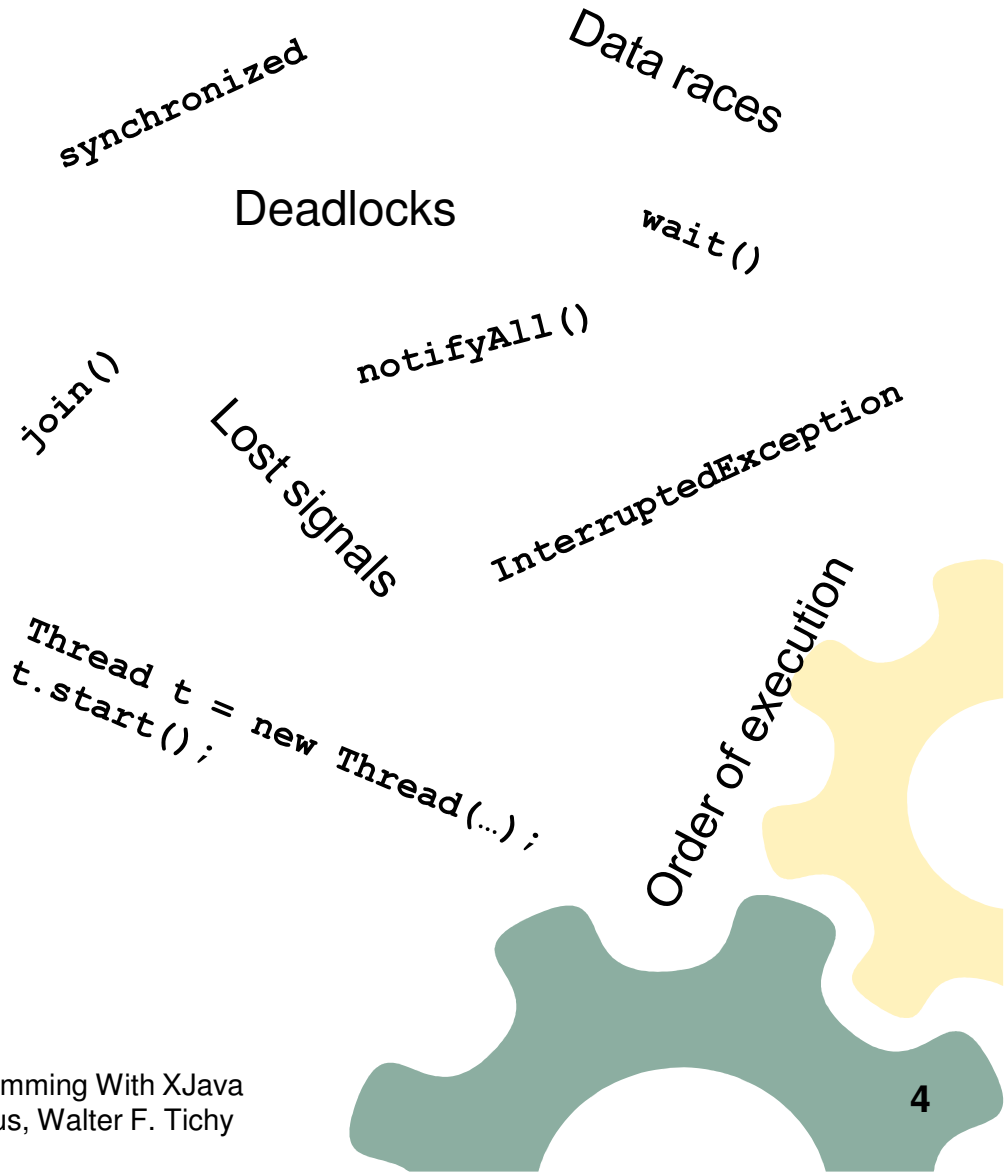
Motivation: A Parallelized Object-Oriented Real-World Application

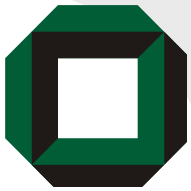


Multithreading and OO

- Create and manage threads manually
- Synchronization is difficult an error-prone

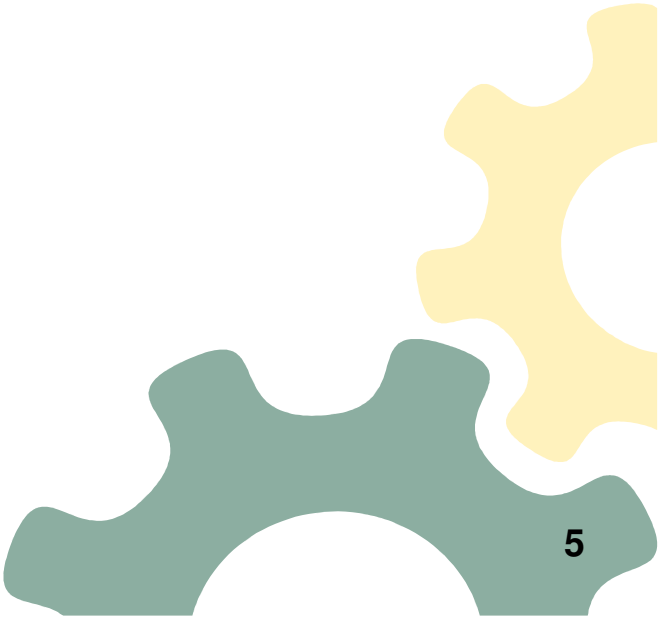
➔ **We need abstractions for parallelism!**

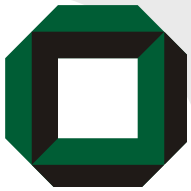




XJava's Goal

- „What you see is what you get“ parallelism
  - More predictable
- High-level programming
  - No explicit threads or manual synchronization

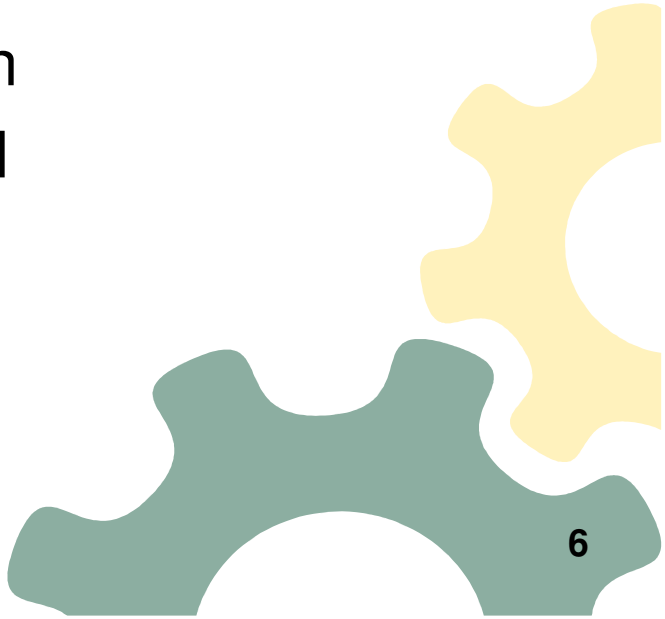


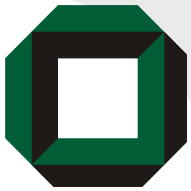


Idea

- Combine the concepts from **object-orientation**, **parallel design patterns**, and **stream languages**

- Stream programs
  - Interconnected filters
  - Input: data stream of arbitrary length
  - Domain-specific, not object-oriented



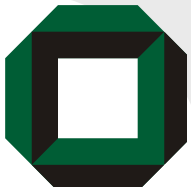


XJava Tasks

- Specialized methods that are concurrently executable

- Typed input and output ports for receiving and generating streams of data

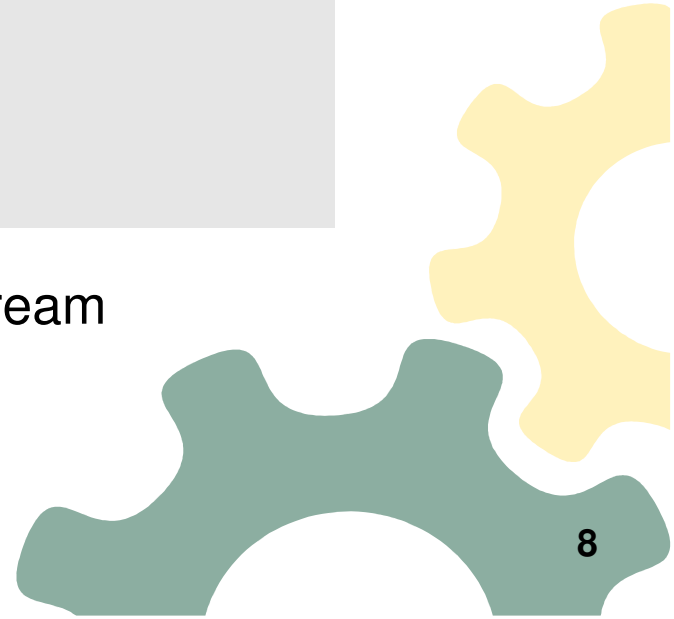
```
public Object => String foo() {
  /* task body */
}
```

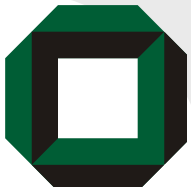


Task Declarations

```
public void => Block read(File f) {
  Iterator i = f.blockIterator();
  while(i.hasNext()) { push (Block) i.next(); }
}
```

`push` puts an element to the output stream

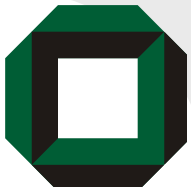




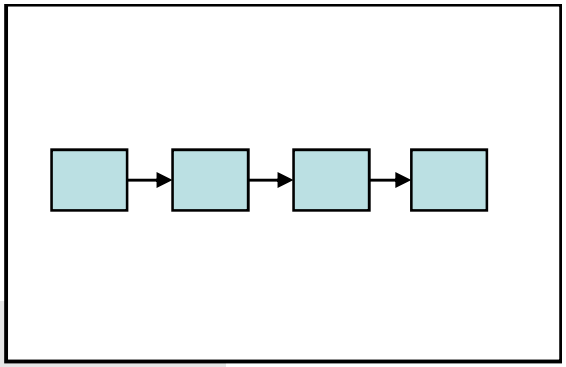
Task Declarations

```
public void => Block read(File f) {
  Iterator i = f.blockIterator();
  while(i.hasNext()) { push (Block) i.next(); }
}

public Block => Block compress() {
  work(Block b) { push b.compressBlock(); }
}

public Block => void write(File f) {
  work(Block b) { f.add(b); /* no push */ }
}
```

A **work**-Block is repeatedly executed for each incoming element of the input data stream

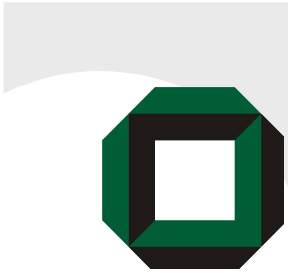


Pipe Statements

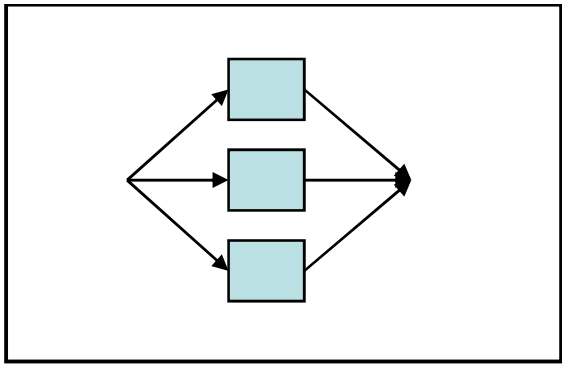


```
public void => Block read(File f) {
  Iterator i = f.blockIterator();
  while(i.hasNext()) { push (Block) i.next(); }
}

public Block => Block compress() {
  work(Block b) { push b.compressBlock(); }
}

public Block => void write(File f) {
  work(Block b) { f.add(b); /* no push */ }
}
```

```
read(inFile) => compress() => write(outFile);
```



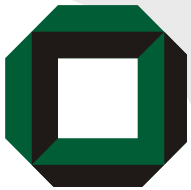
Concurrent Statements



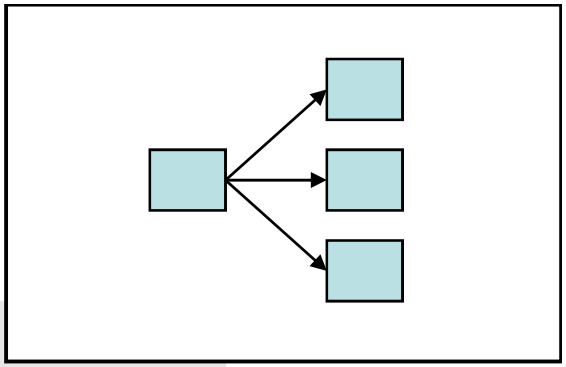
```
public void => void compress(File in, File out) {
  read(in) => compress() => write(out);
}
```

Compress two files in parallel:

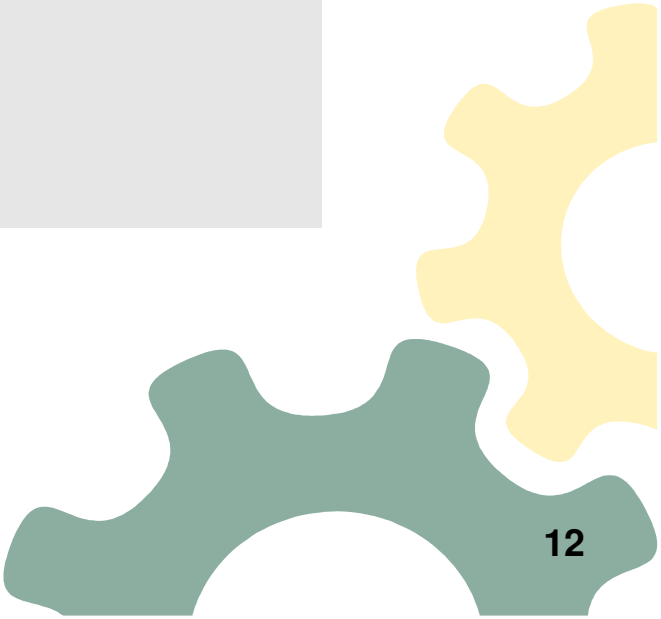
```
compress(f1, f1out) ||| compress (f2, f2out);
```

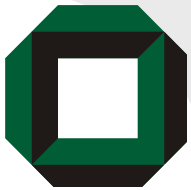


Master/Worker

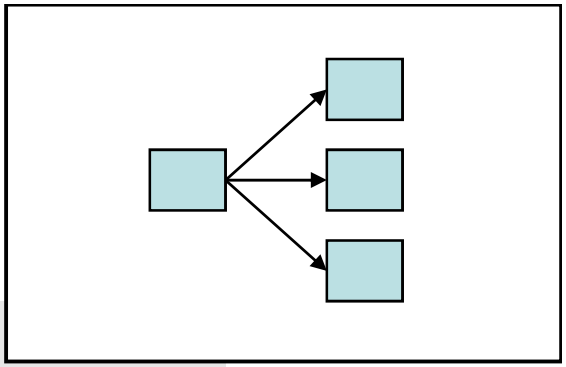


```
public void => Item master() {
  while (moreWork) { Item item = …
    push item; }
}

public Item => void worker() {
  work(Item item) { … }
}

public Item => void wPool(int i) {
  worker():i;    // create i workers
}
```



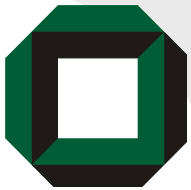


Master/Worker

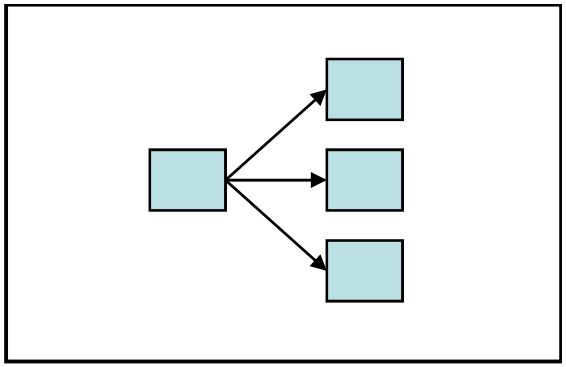


```
public void => Item master() {
  while (moreWork) { …
    push item; }
}

public Item => void worker() {
  work(Item i) { … }
}

public Item => void wPool(int i) {
  worker():i;   // create i workers
}
```

```
master() => wPool(3);
```



Master/Worker



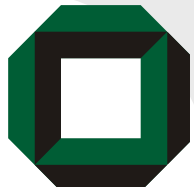
```
public void => Item master() {
  while (moreWork) { …
    push item; }
}

public Item => void worker() {
  work(Item i) { … }
}

public Item => void wPool(int i) {
  worker():i;   // create i workers
}
```

How to distribute work items? What does „=>“ mean?

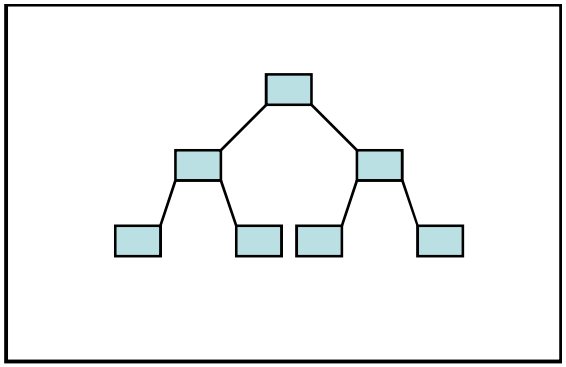
```
master() => wPool(3);     // round robin
```

```
master() =>? wPool(3);    // FCFS
```

```
master() =>* wPool(3);    // broadcast
```



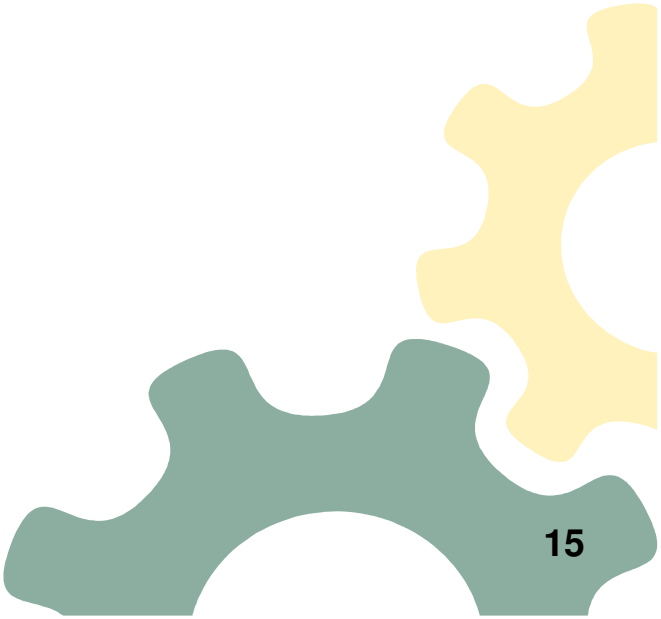
Divide & Conquer Parallelism

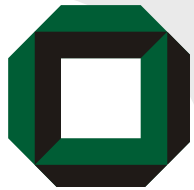


- Compare sequential and parallel MergeSort

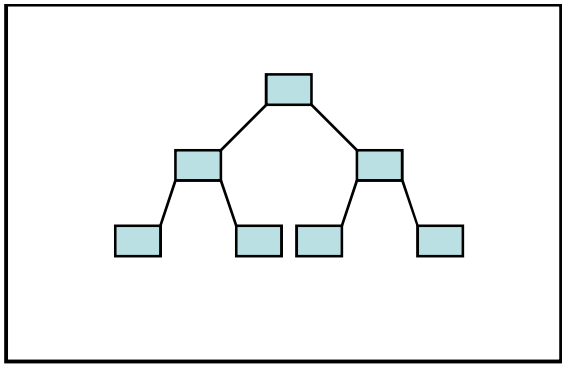
```
void mergesort(int from, int to) {
  if (to - from > threshold) {
    int x = from + (to - from)/2;
    mergesort(from, x);
    mergesort(x + 1, to);
    merge(from, x + 1, to);
  }
  else sort(from, to);
}
```

sequential





Divide & Conquer Parallelism



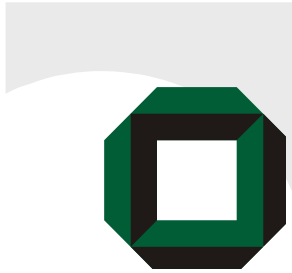
- Compare sequential and parallel MergeSort

```
void mergesort(int from, int to) {
  if (to - from > threshold) {
    int x = from + (to - from)/2;
    mergesort(from, x);
    mergesort(x + 1, to);
    merge(from, x + 1, to);
  }
  else sort(from, to);
}
```

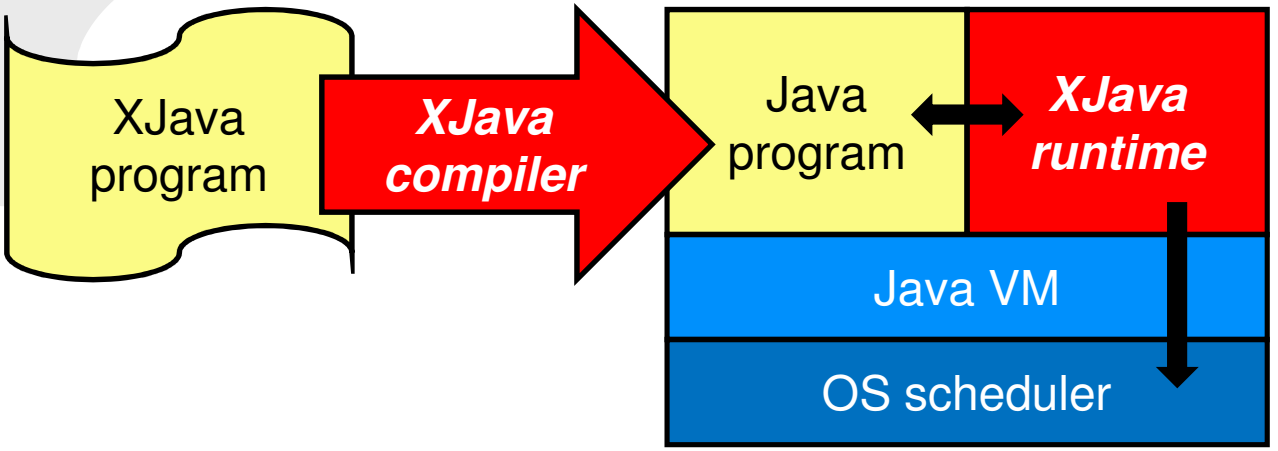
sequential

```
void => void mergesort(int from, int to){
  if (to - from > threshold) {
    int x = from + (to - from)/2;
    mergesort(from, x)
             ||| mergesort(x + 1, to);
    merge(from, x + 1, to);
  }
  else sort(from, to);
}
```

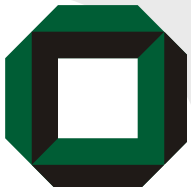
XJava



XJava Compiler and Runtime



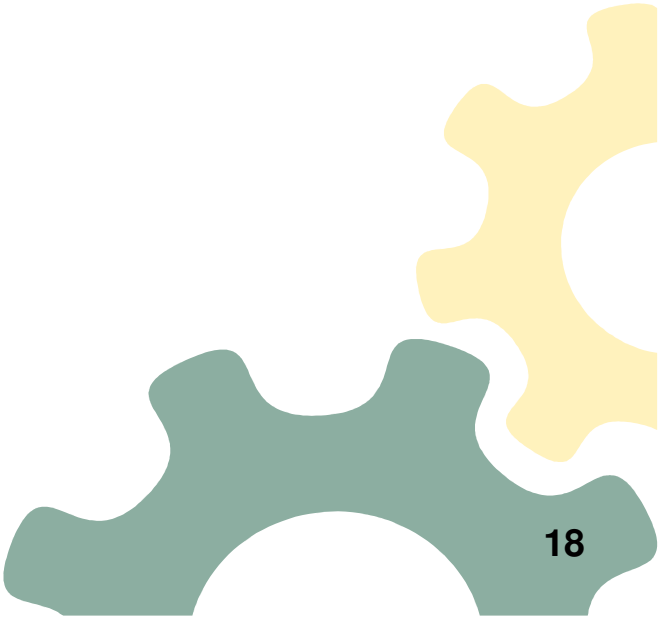
- XJava runtime is the place where parallelism actually happens

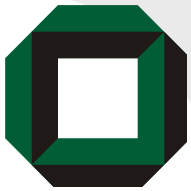


Preliminary Results

- **Up to 40% code savings** over threaded Java
  - Almost no manual synchronization needed

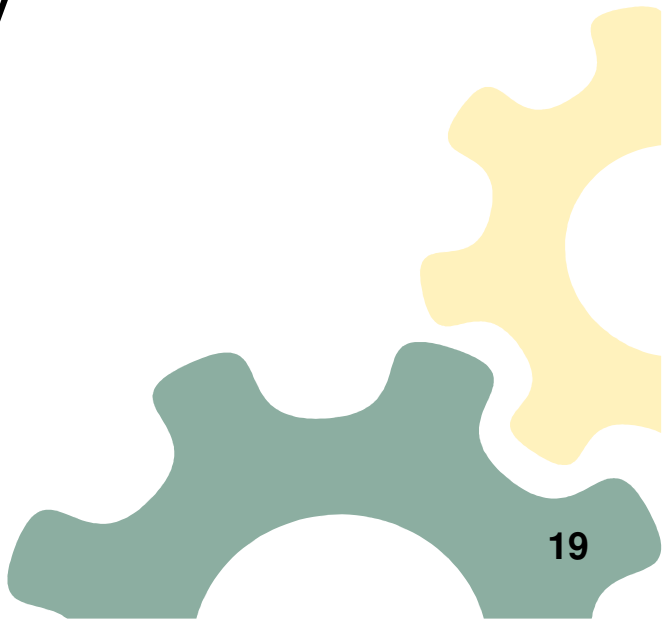
- Good **speedups** over sequential Java
  - **Up to 3.5** (Quadcore)
  - **Up to 31.5** (Sun Niagara 2)

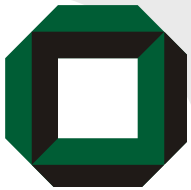




XJava: Conclusion

- High-level parallel OO programming
- Lower risk of synchonization bugs
- Code savings, higher productivity
- Performance





Thank you!

Questions?