Karlsruhe Institute of Technology

Programmierparadigmen

Prof. Dr. Ralf Reussner

Topic 5.1

Parallel Programming Fundamentals

DEPENDABILITY OF SOFTWARE-INTENSIVE SYSTEMS
INSTITUTE OF INFORMATION SECURITY AND DEPENDABILITY, FACULTY OF INFORMATICS

dsis.kastel.kit.edu

Overview on Today's Lecture

- Content
  - Motivation for parallel programming
  - Fundamentals of parallel programming
  - Exemplary parallel algorithms
- Learning goals: participants –
  - understand why parallel programming becomes increasingly relevant
  - understand basic concepts and approaches for parallel programming
  - know the basic terminology in parallel programming
  - can estimate the parallelization potential for given programs
- Parallelization topics in the next weeks
  - C basics & parallel programming with MPI
  - Parallel programming in Java
  - Design by Contract

Computer Performance Development

Atari
Mega ST 2
8 MHz

1987

Umax
Pulsar
150 MHz

1998

Samsung
Omnia II
800 MHz

2009

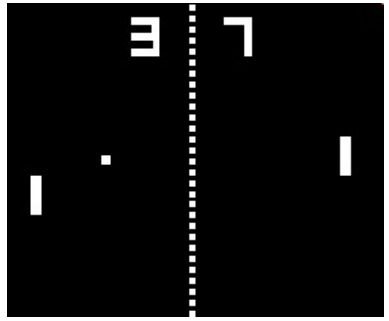
Samsung
Galaxy S25
4470 MHz

2025

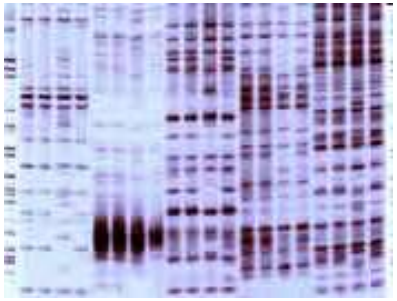
Motivation

- Application of Murphy‘s Law on computer performance:

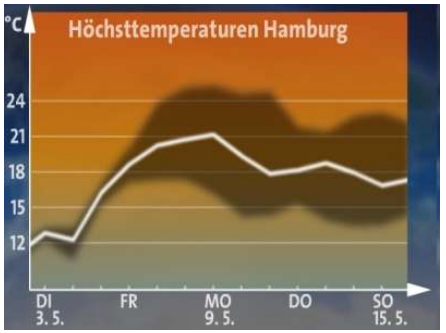
  Every computer is too slow ☺



- Why does that bother us?
  - Computer graphics:
    Medicine, movies, games, …
  - Genome analysis

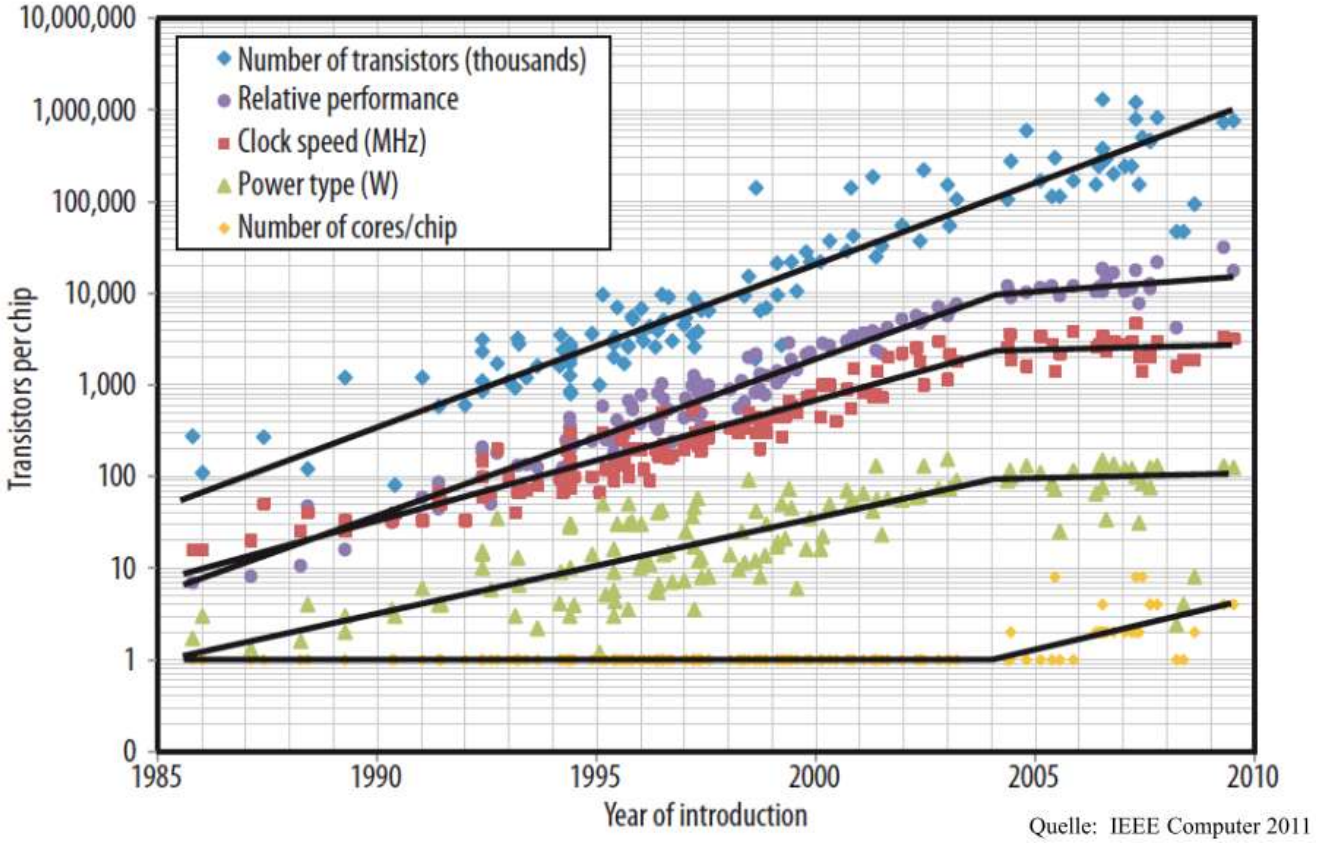


  - Weather forecast
  - etc.



Moore‘s Law (1)

- Moore’s law states that the number of transistors in integrated circuits doubles every two years



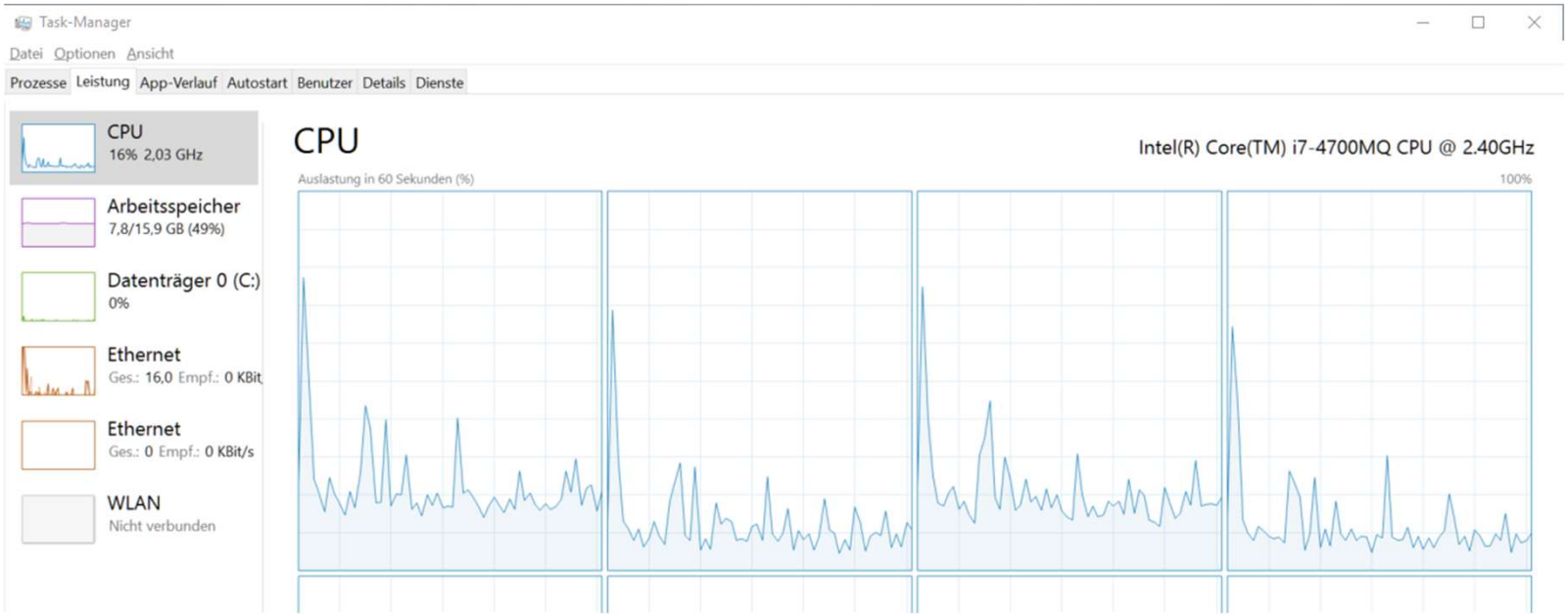
[SWT I]

- Meanwhile, Moore‘s law ceases to be valid

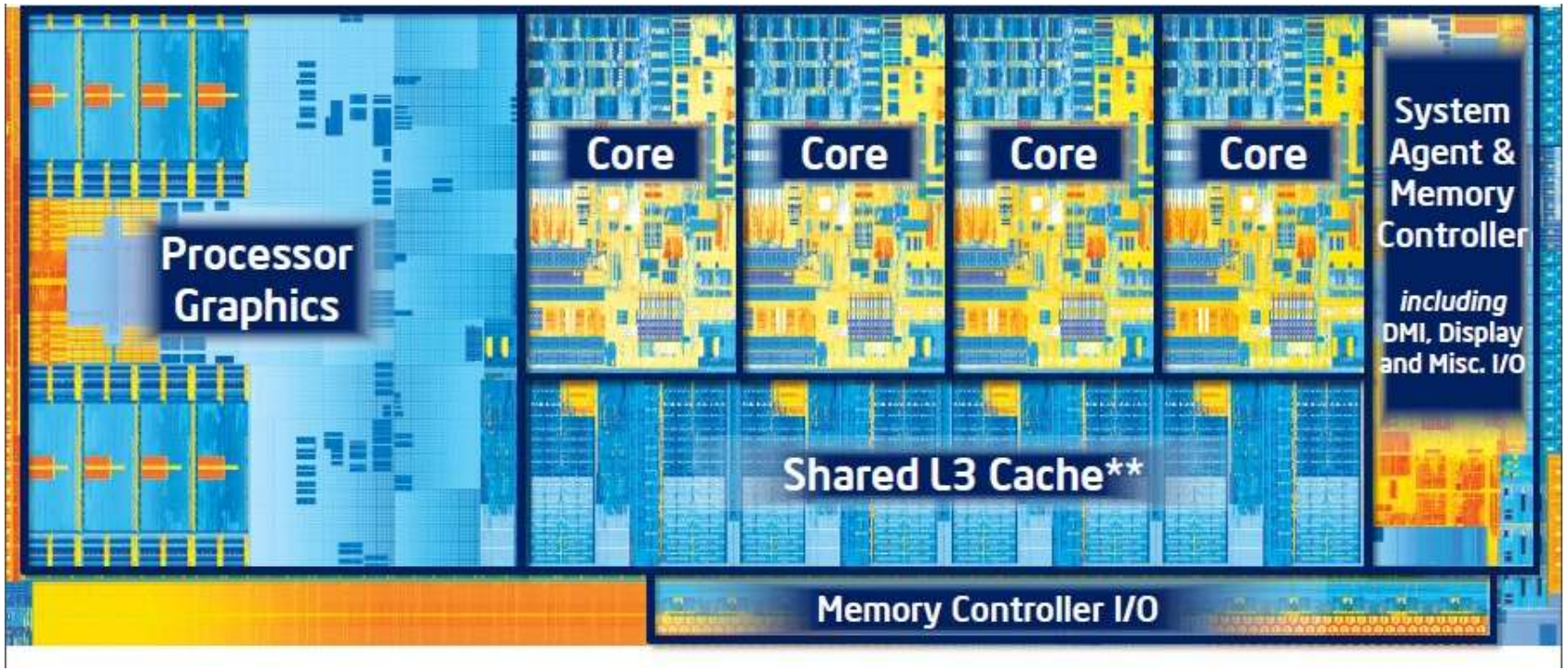
Moore‘s Law (2)

- Often not only applied on complexity, but also on –
  - performance
  - or even on clock rates
- The clock rates stagnate since several years at about 3 – 4 GHz

- How can performance be further increased?
  - e.g. with parallel processing on multi-core processors or distributed systems



Intel Multicore Architectures



Ivy-Bridge Architecture [Intel]

- 2012 (Ivy-Bridge Desktop): up to 6 Cores
- 2018 (Skylake-X Desktop): up to 18 Cores
- 2025 (Bartlett-Lake Desktop): up to 24 Cores
- Future: ?

Consequences (1)

- Performance can be increased by parallelization
- Main products of processor manufacturers are multi-core processors
  - Servers ship with multi-core processors since 2005
  - Laptops ship with dual-core processors since 2006
  - Even smartphones use multi-core processors since several years
- **Parallel processing becomes the rule**
- Fundamental transition from sequential to parallel processing
  - Dramatic impacts on applications, research and education
  - Buzzwords: Grid and Cloud Computing, Cluster, Supercomputer
- **Computer scientists have to learn how to master „parallel applications“**

Consequences (2)

- Software producers (and the computer science education) have to adapt to parallel processing techniques to stay competitive
  - Migration of existing applications
  - Development of new, parallel applications

- More details on parallel programming in advanced lectures
  - Parallelrechner und Parallelprogrammierung
  - Parallele Algorithmen
  - Effizientes paralleles C++
  - etc.

Parallel Processing

- No programming paradigm in the proper sense
  - Can be implemented in different paradigms and languages

- The following lectures demonstrate basic concepts exemplarily in –
  - C: Message Passing Interface (MPI)
  - Java: Advanced concepts and actors

- Modern processors support parallel execution on different levels
  - Bit level (e.g. two 32-bit additions in a 64-bit architecture)
  - Instruction level (e.g. pipelining)
  - Data level (e.g. loops)
  - Processing level (common in modern systems)

Parallelism vs. Concurrency

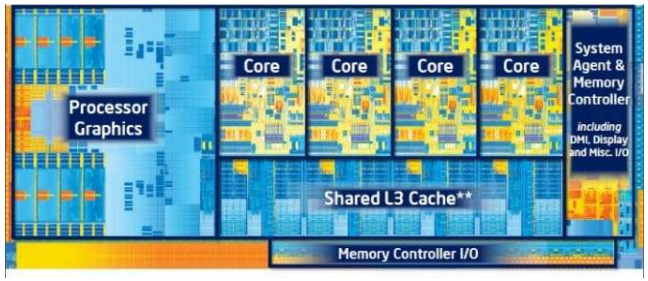
- In the context of parallel computing, the terms parallelism and concurrency are commonly used
- We follow the definition of Oracle for these terms [Oracle]

- **Parallelism**: At least two threads are *executed simulateously*
- **Concurrency**: At least two threads *make progress*

- Concurrency is a generalized form of parallelism
  - Includes time-slicing as a form of virtual parallelism, which computers support for many years

Parallel Programming Approaches

- Two different parallel programming approaches
- For computer systems with shared memory
  - Each processor can access each memory cell
  - Processors communicate by manipulating shares memory cells
  - e.g. on multi-core processors



[Intel]

- For computer systems with distributed memory
  - Each processor has its own memory, which is only accessible by that processor
  - Processors communicate via messages (message passing)
  - e.g. on computer clusters

[NASA]

Shared memory (1)

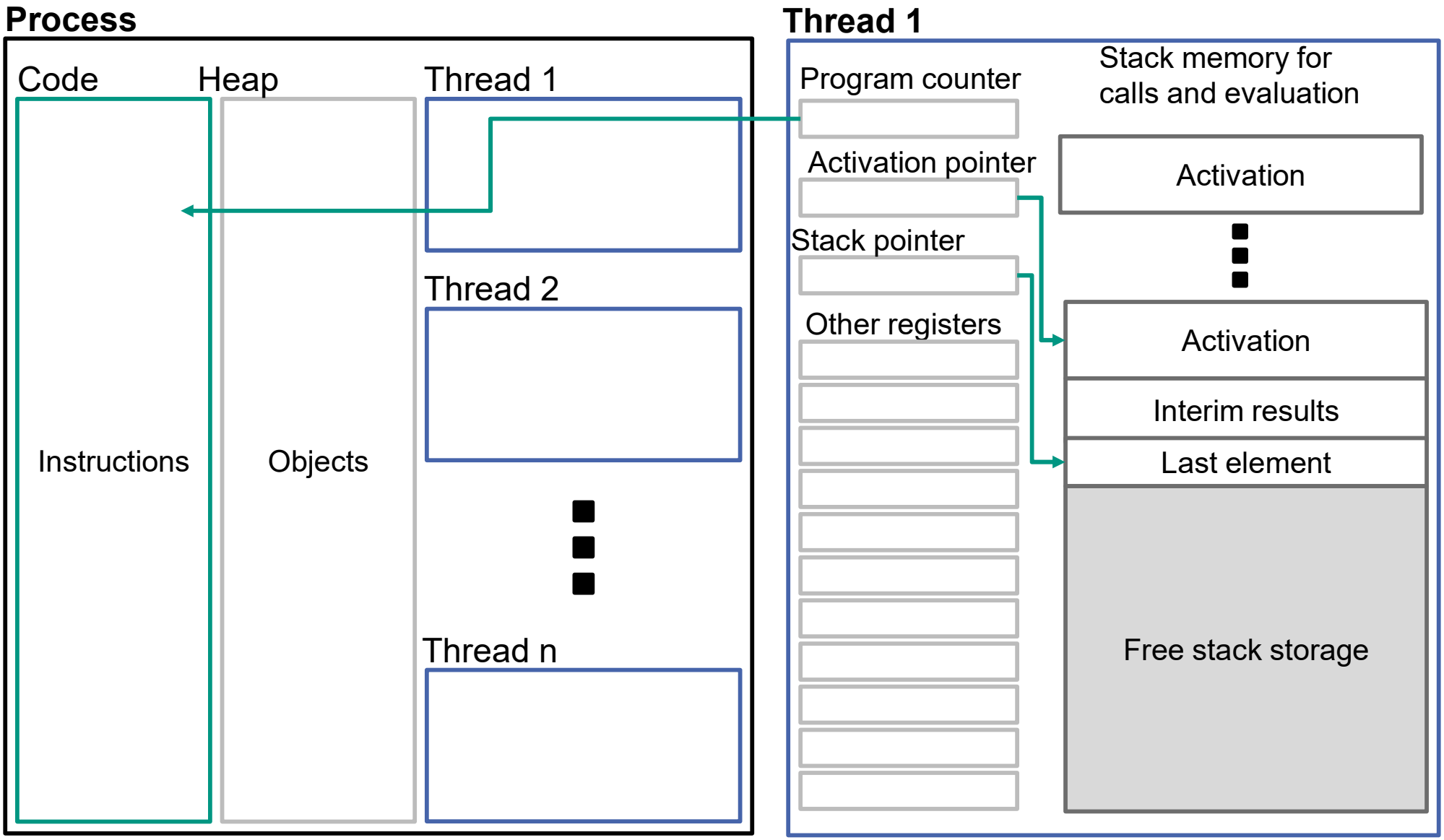
Process

- Generated by operating system
- Contains information about program resources and execution status, e.g.
  - Code segment (program instructions)
  - Data segment (for global variables, stack, heap)
  - At least 1 control thread
- CPU context switch between processes slow

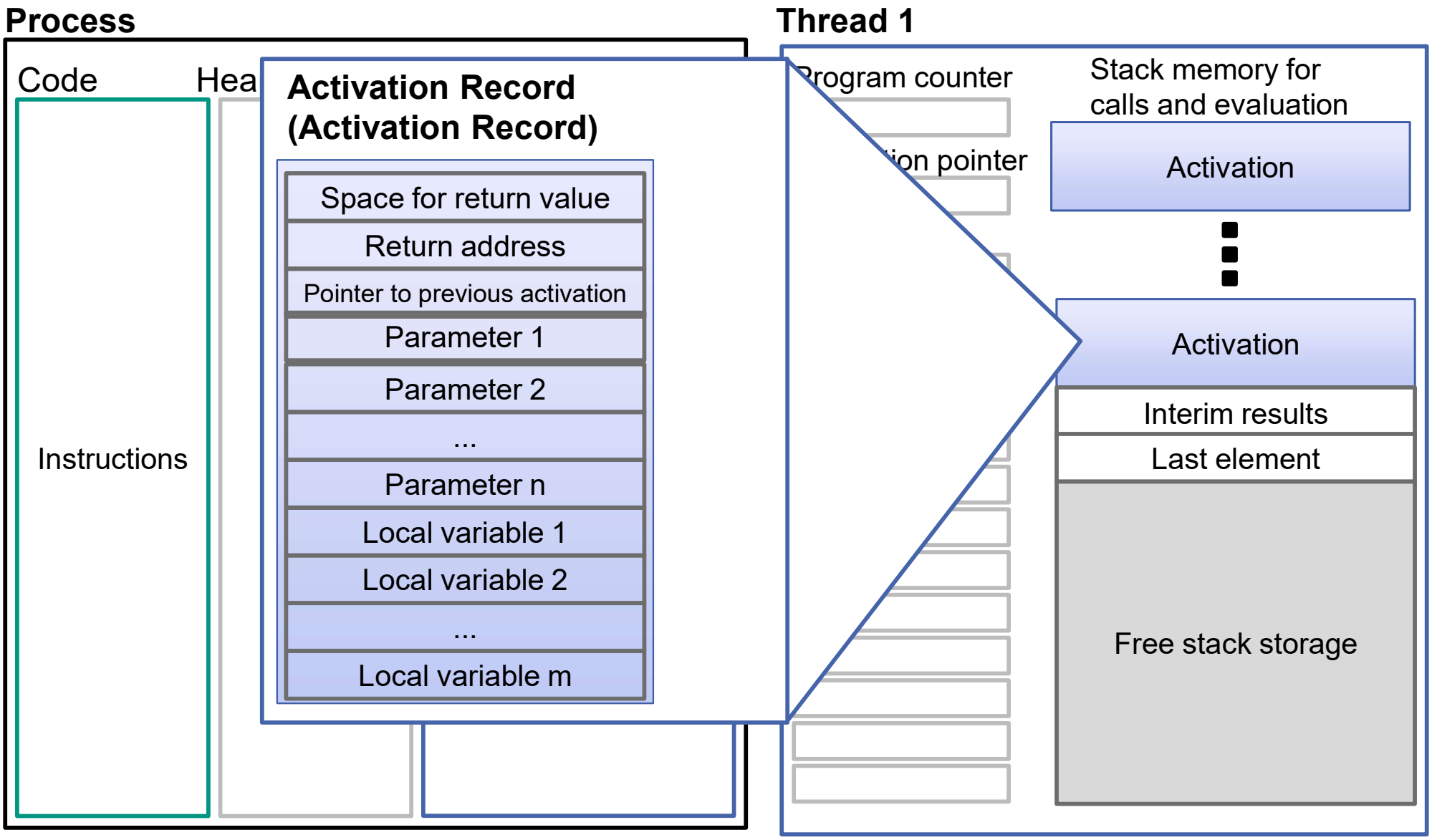
Control thread

- Instruction sequence that is executed
- Exists in a process
- A thread has its own
  - Command pointer
  - Stack
  - Register
- Shares with other threads in the same process
  - Address space
  - Code/data segments
  - Other resources (e.g. open files, locks, etc.)
- CPU context change between threads of the same process faster than between different processes.

Processes and control threads



Processes and control threads



Shared memory (2)

- Basic procedure for shared memory:
  - Threads perform parallel tasks (instructions)
  - Exchange of information about shared variables in the memory
    - In Java, this is only possible via objects on the heap. The stack is thread-local.
  - Synchronization constructs coordinate execution in the event of data or control dependencies
  - Processes and their threads are always generated by the operating system and distributed to processors or cores
    - Interface usually built into programming languages (e.g. Java threads) or via libraries (C Pthreads)

Flynn's Taxonomy

- {Single, Multiple} Instruction/Program x {Single, Multiple} Data
  - SISD, SIMD, MIMD, MISD
- SISD: Single Instruction x Single Data
  - von Neumann architecture:
    a single instruction stream operates on a single memory
- SIMD: Single Instruction x Multiple Data
  - One instruction is applied on homogeneous data (e.g. an array)
  - e.g. vector processors of early supercomputers
- MIMD: Multiple Instruction x Multiple Data
  - Different processors operate on different data
  - e.g. today's multi-core processors
- MISD: Multiple Instruction x Single Data
  - Multiple instructions are executed simultaneously on the same data
  - e.g. in redundant architectures or in pipelines of modern processors (controversal view)

Challenges

- The programmer is responsible for partitioning the problem
- Two approaches:
  - Task parallelism: functional decomposition
    - Define tasks that can be executed in parallel
    - Tasks should be as independent as possible
  - Data parallelism: data decomposition
    - Partition the data on which the same operation is executed in parallel
    - Tasks should be able to work on the partitions as indepdently as possible
- „Existing“ challenges are still relevant, such as –
  - inter-process communication
    - UMA vs. NUMA ((non-)uniform memory access) and data exchange
    - Threads, shared memory, message passing
  - potentially arising race conditions and deadlocks
    - Can be avoided with techniques for mutual exclusion, such as monitors or semaphores

Theoretical Example Parallel Addition (1)

- Ideally, problems can be partitioned, such that the partitions are independent and can be processed in parallel
  - e.g. many algorithms for image processing or a vector addition
- Even a piecewise parallelization is possible:
  - Given the following array of integers

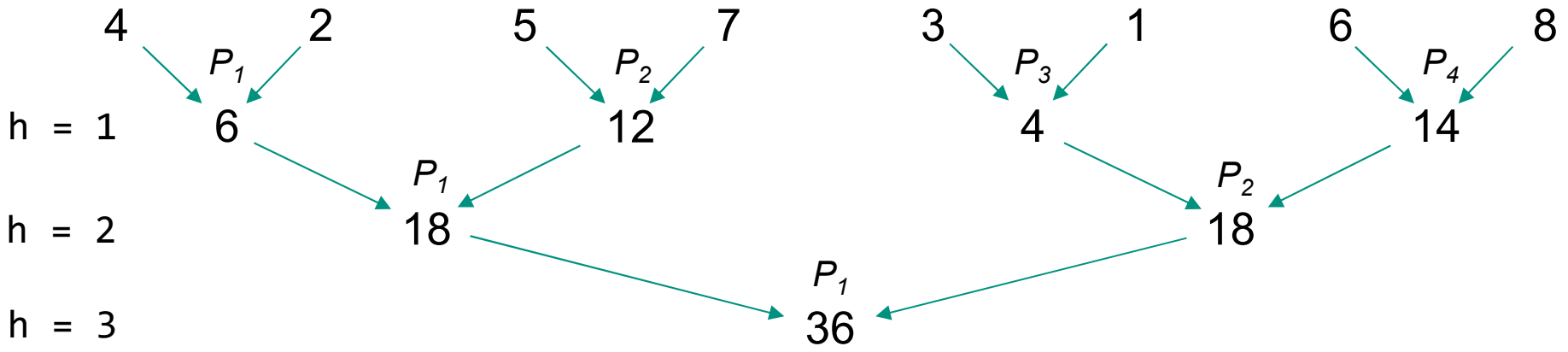
```
a[] = {4, 2, 5, 7, 3, 1, 6, 8}
```

adding the numbers sequentially requires eight cycles

  - e.g. within a for-loop
  - 4 + 2 + 5 + 7 + 3 + 1 + 6 + 8 = 36
  - For *n* numbers, the operation requires *O(n)* cycles
- How can we sum up “in parallel“? (e.g. on 4 processors)

Parallel Addition (2)

- Summing up the entries of a[] in parallel using 4 processors
  - Per divide and conquer on a SIMD architecture



The algorithm can be executed in O(log n)

```
for h := 1 to log2(n) do
   for i := 0 to n / 2^h -1 pardo
      a[i] := a[2i] + a[2i + 1]
   endfor
endfor
```

This instruction can be distributed on the different P_i

cf. [Wagner08]

Amdahl's Law

- The possible speedup of an algorithm executed on *n* processors can be computed as follows

$$S(n) = \frac{T(1)}{T(n)} = \frac{\text{execution time if processed by one processor}}{\text{execution time if processed by } n \text{ processors}}$$

- With Amdahl's law (1967) the **maximal** speedup that can be achieved with parallel processing on *n* processors can be computed
  - $p = \text{parallelizable percentage of the program}$

$$S(n) = \frac{1}{(1-p) + \frac{p}{n}}$$

  - Amdahl's law shows the speedup limitation due to the non-parallelizable part of a program since even for large *n*
  - Due to synchronization and communication overhead, this speedup is hardly reachable in practice

Example: Sorting Numbers Sequentially

- … with bubble sort
  - Given the following array of integers

```
a[] = {7, 3, 6, 5, 8, 1, 4, 2}
```

  - An optimized sequential bubble sort requires 7 + 6 + 5 + 4 + 3 + 2 + 1 = 28 comparisons in this example
  - In general, the algorithm requires $n(n - 1)/2$ operations → **$O(n^2)$**

Turing Award winner
Donald Knuth

Bubble sort seems to have nothing to recommend it, except a catchy name and the fact that it leads to some interesting theoretical problems.

Serial Java Bubble Sort

```
public class SerialBubbleSort {
   static int[] a = new int[]{7, 3, 6, 5, 8, 1, 4, 2};

   public static void main(String[] args) {
      for (int i = a.length-1; i > 0; i--) {
         for (int j = 0; j < i; j++) {
            if (a[j] > a[j+1]) {
               int temp = a[j+1];
               a[j+1] = a[j];
               a[j] = temp;
            } // endif
         } // for j
      } // for i
   } // main
} // eof
```

Example: Sorting Numbers In Parallel

- … with bubble sort
  - Given the following array of integers

```
a[] = {7, 3, 6, 5, 8, 1, 4, 2}
```

```
for h := 0 to n div 2 do                          → O(n)
   for i := 0, 2, 4, ... pardo                    → O(1)
      compareAndExchange(a[i], a[i+1])
   endfor
   for i := 1, 3, 5, ... pardo                    → O(1)
      compareAndExchange(a[i], a[i+1])
   endfor
endfor
```

```
if (a[i] > a[i+1])
   swap(a[i], a[i+1])
```

Each loop run is in *O(1)*, because it is completely executed in parallel on *n* processors

Conclusion

- Moore‘s Law for clock-rate is at an end → Further performance improvements primarily through parallel computing
- Two parallel processing approaches
  - For computer systems with shared memory
  - For computer systems with distributed memory
- Parallelization through task parallelism and data parallelism
- Flynn‘s Taxonomy
  - {Single, Multiple} Instruction x {Single, Multiple} Data
  - Multi-core processors allow MIMD
- Amdahl‘s Law
  - Gives the theoretical speedup of a program executed in parallel on *n* processors
  - Possible speedup is limited by the part of a program that cannot be parallelized

Literature and References

[Wagner08] http://i11www.iti.uni-karlsruhe.de/_media/information/scripts/scripte/algotech08.pdf

[Intel] http://spmarchitecture.com/ivy-bridge-architecture/ivy-bridge-architecture-layout-147275/

[NASA] https://www.nasa.gov/sites/default/files/goddard_servers.jpg

[Oracle] https://docs.oracle.com/cd/E19455-01/806-5257/6je9h032b/index.html