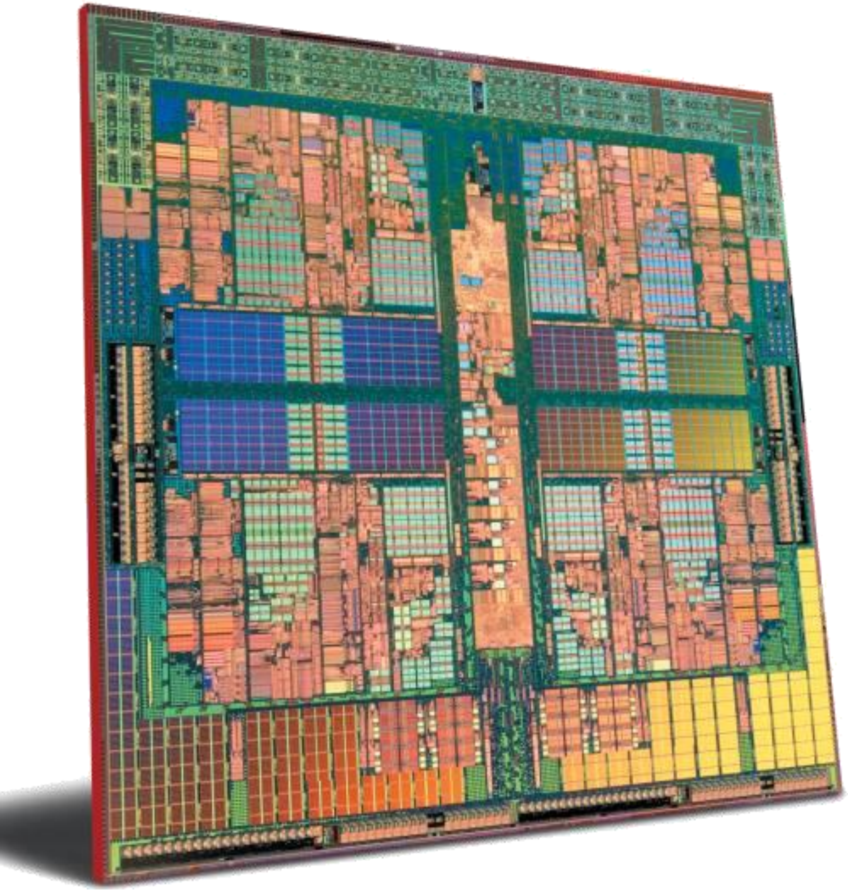
Parallel Architectures



ESCALab@PoliMi

Embedded Systems and Computer Architecture Laboratory

2011 December 1st

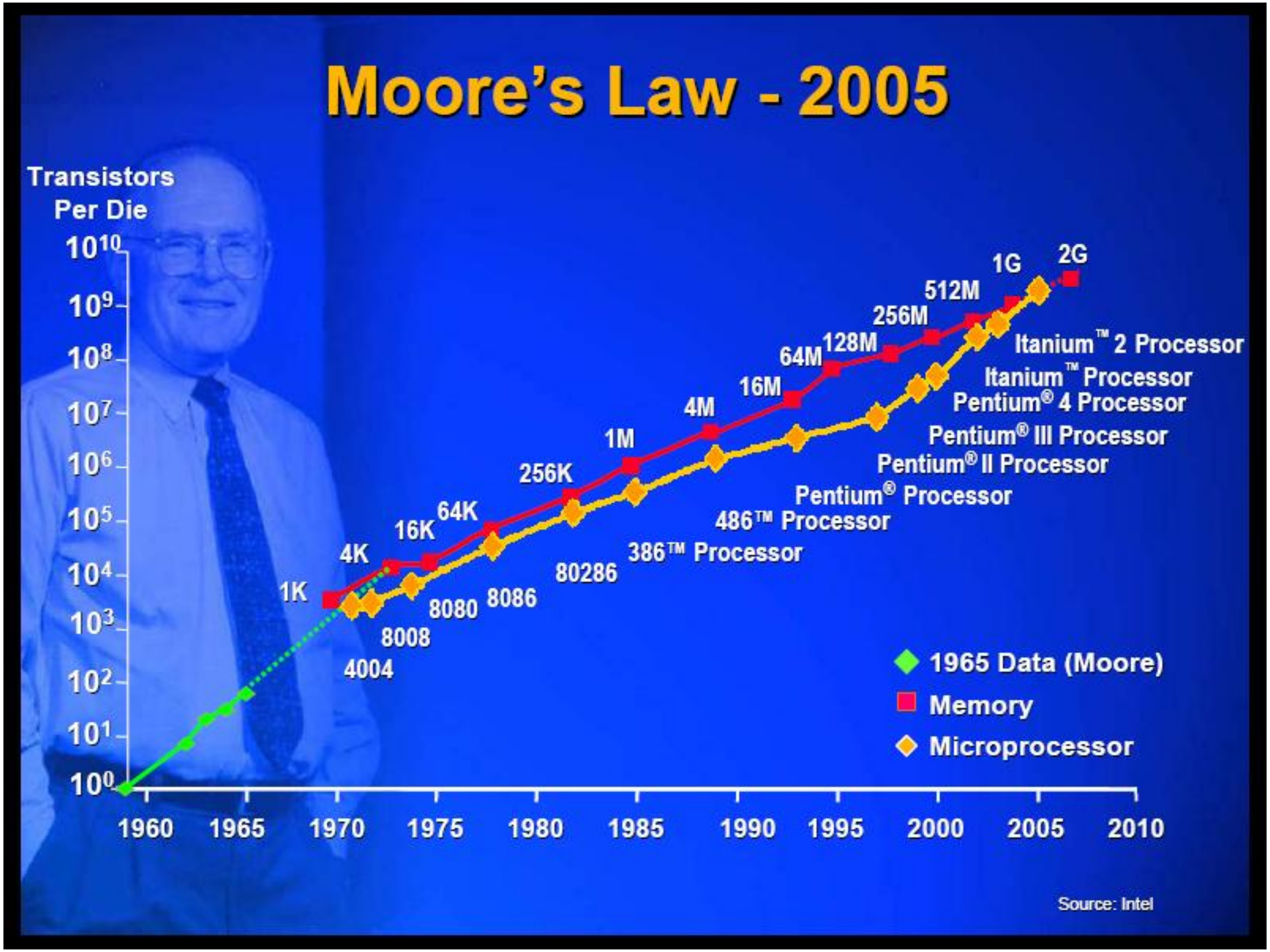
Outline

- **Introduction**
- **Multi-core architectures**
  - Memory hierarchy
  - Interconnect design
- **Commercial examples**
- **Parallel programming issues**
  - Forms of parallelism in software (overview)
  - Main issues in parallel programming
- **Wrap-up**

What’s next

- **Introduction**
- **Multi-core architectures**
  - Memory hierarchy
  - Interconnect design
- **Commercial examples**
- **Parallel programming issues**
  - Forms of parallelism in software (overview)
  - Main issues in parallel programming
- **Wrap-up**

Moore’s Law at Intel: 1965

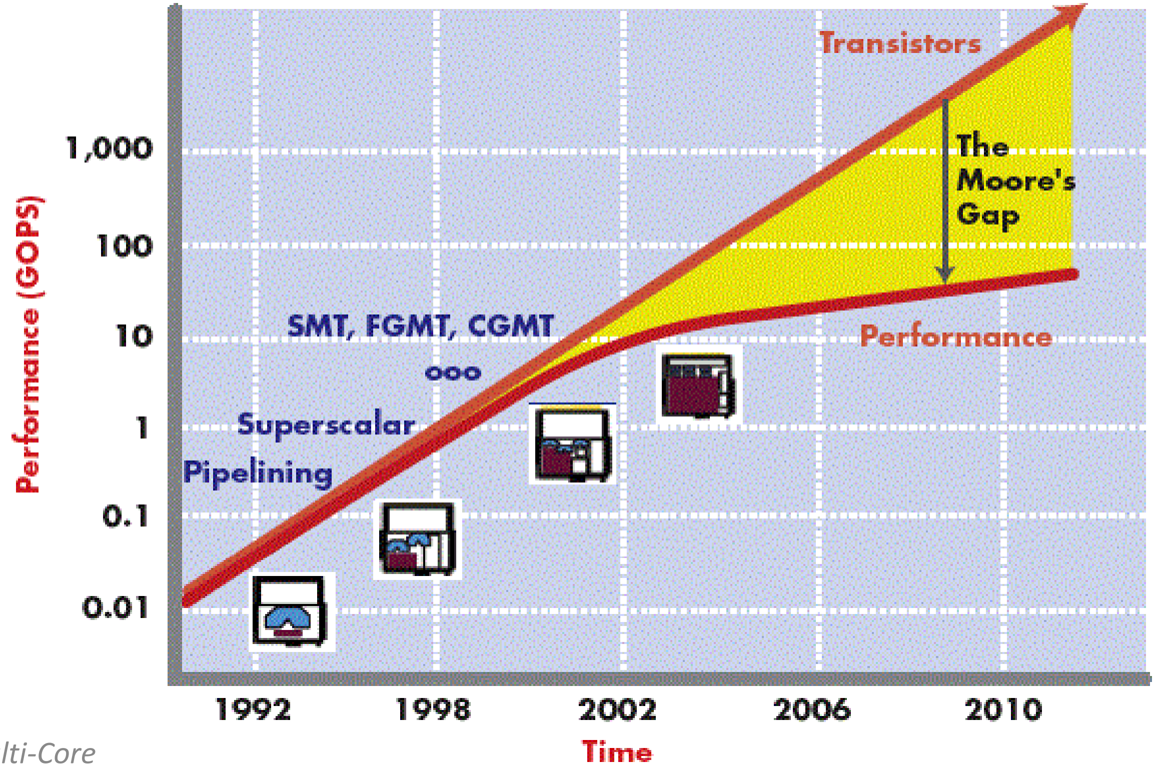


Source: Intel Co.

Moore’s Law turning point: 2002

- **Microarchitectural techniques for ILP**
  - Pipelining, superscalar processors, VLIW, out-of-order execution, register renaming, speculative execution, branch prediction

Moore's Law ran smoothly until 2002, when the gap between performance and gate count started to appear.

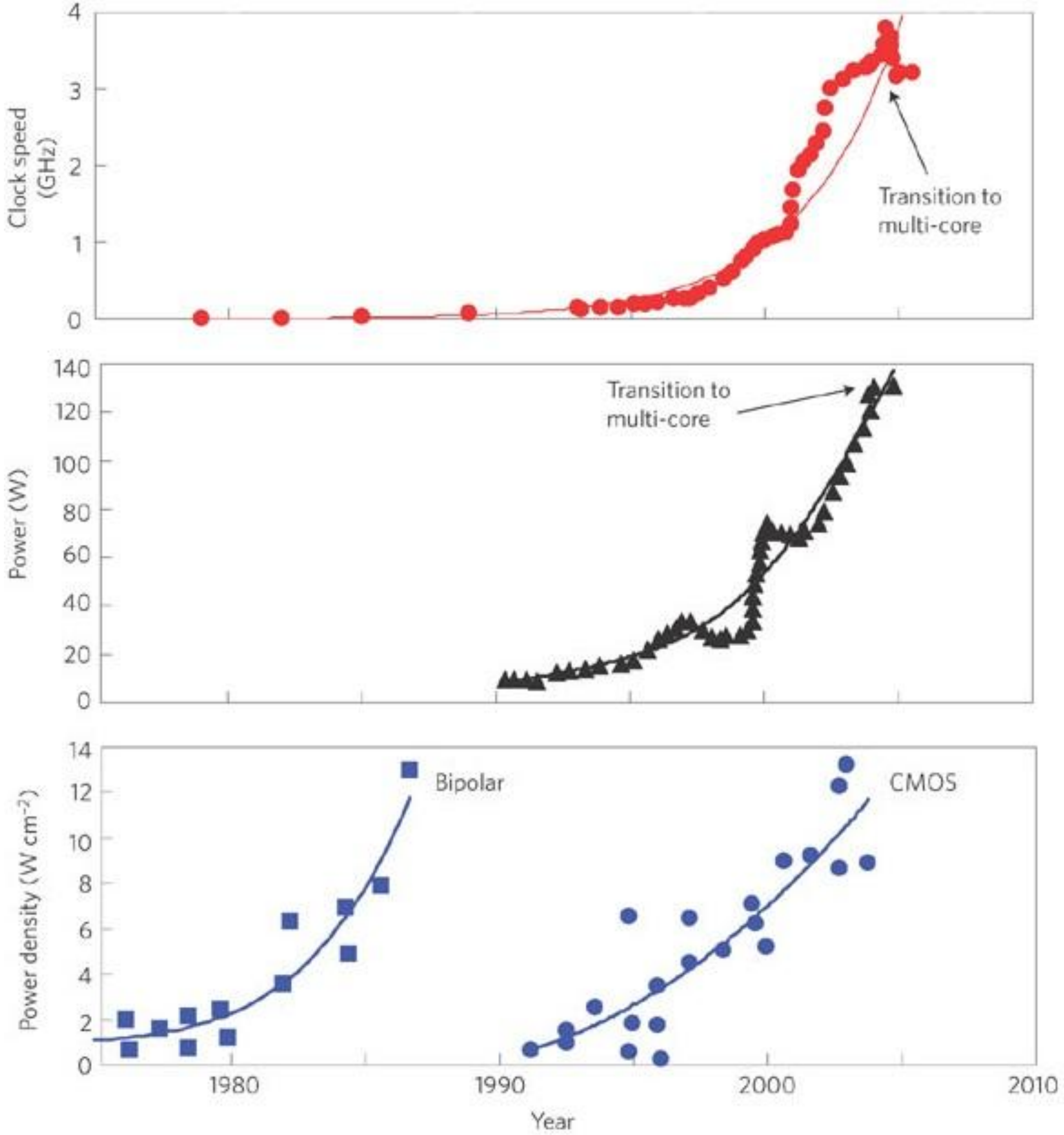


- Moore’s Gap implies that not enough ILP can be exploited from using more transistors!

- Why?
  - Power consumption
  - Signal propagation delay > transistor delay
  - Memory bottleneck
    - Memory stalls dominate

Microelectronics industry trend

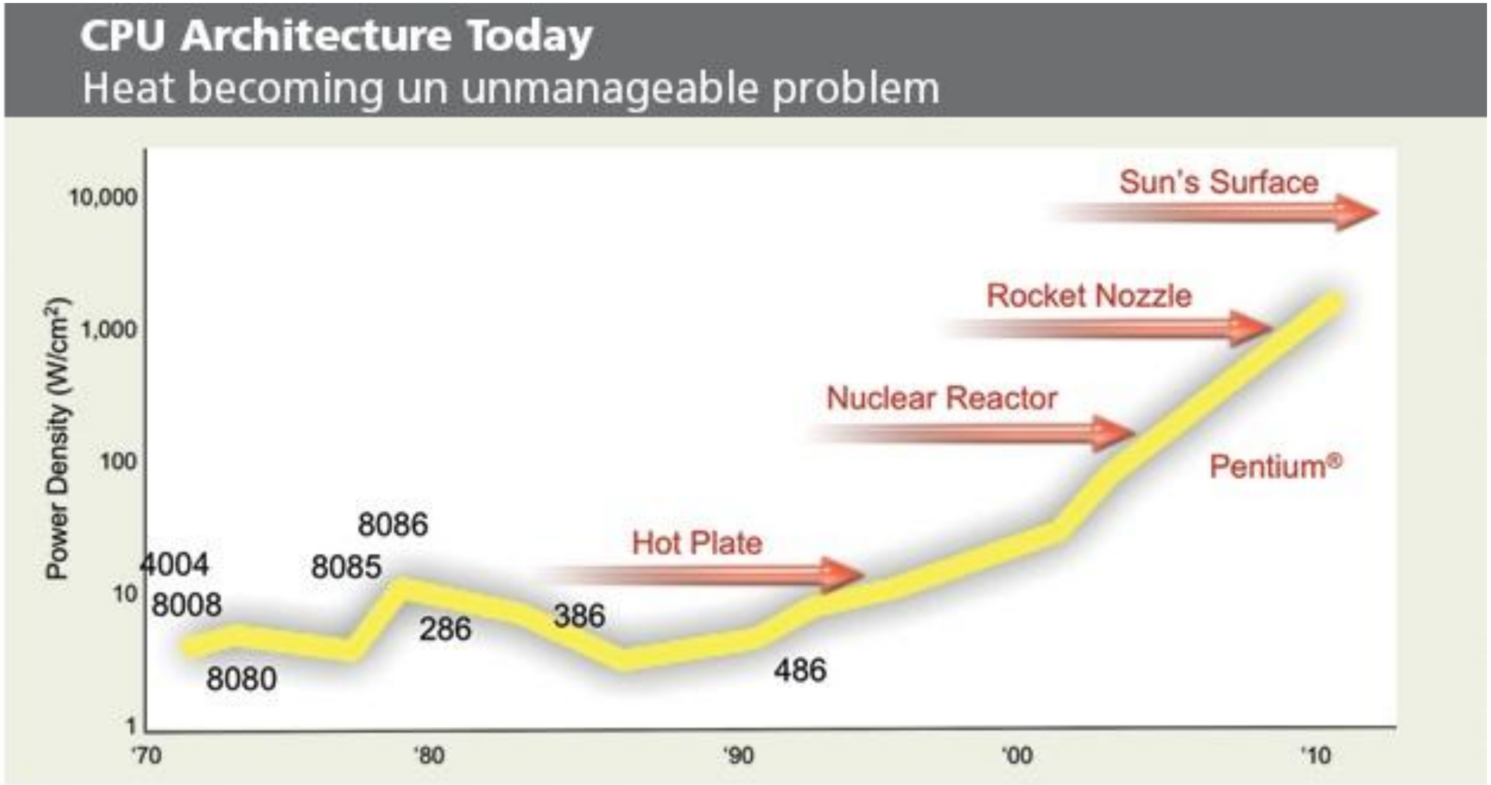
- Higher performance
- Higher power
- Higher power density



Power consumption

$$P_{dynamic} \propto V^2 f C_{load}$$

- Density [W/cm^2]

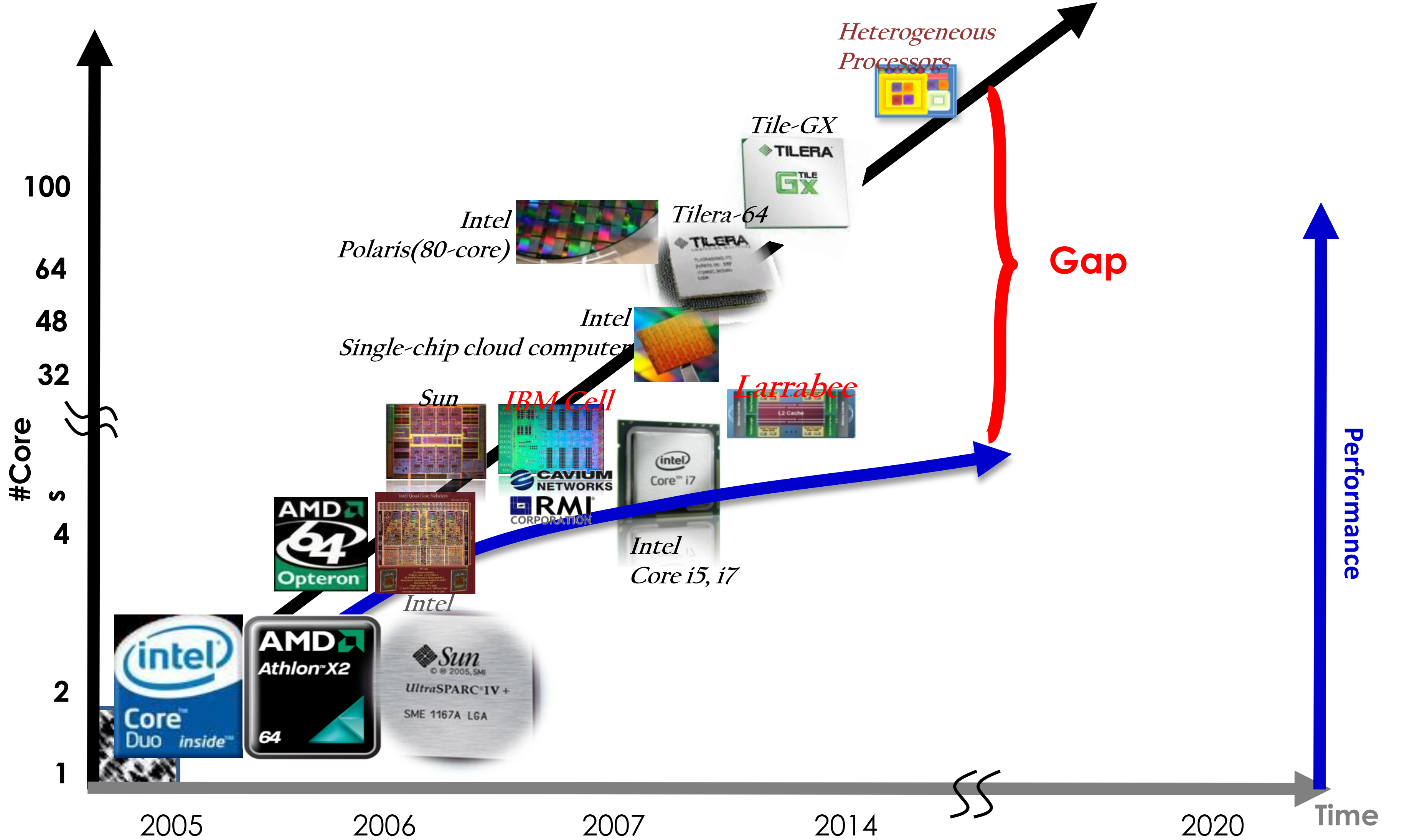


CPU Architecture Today

Heat becoming un unmanageable problem

(Courtesy of Pat Gelsinger, Intel Developer Forum, Spring 2004)

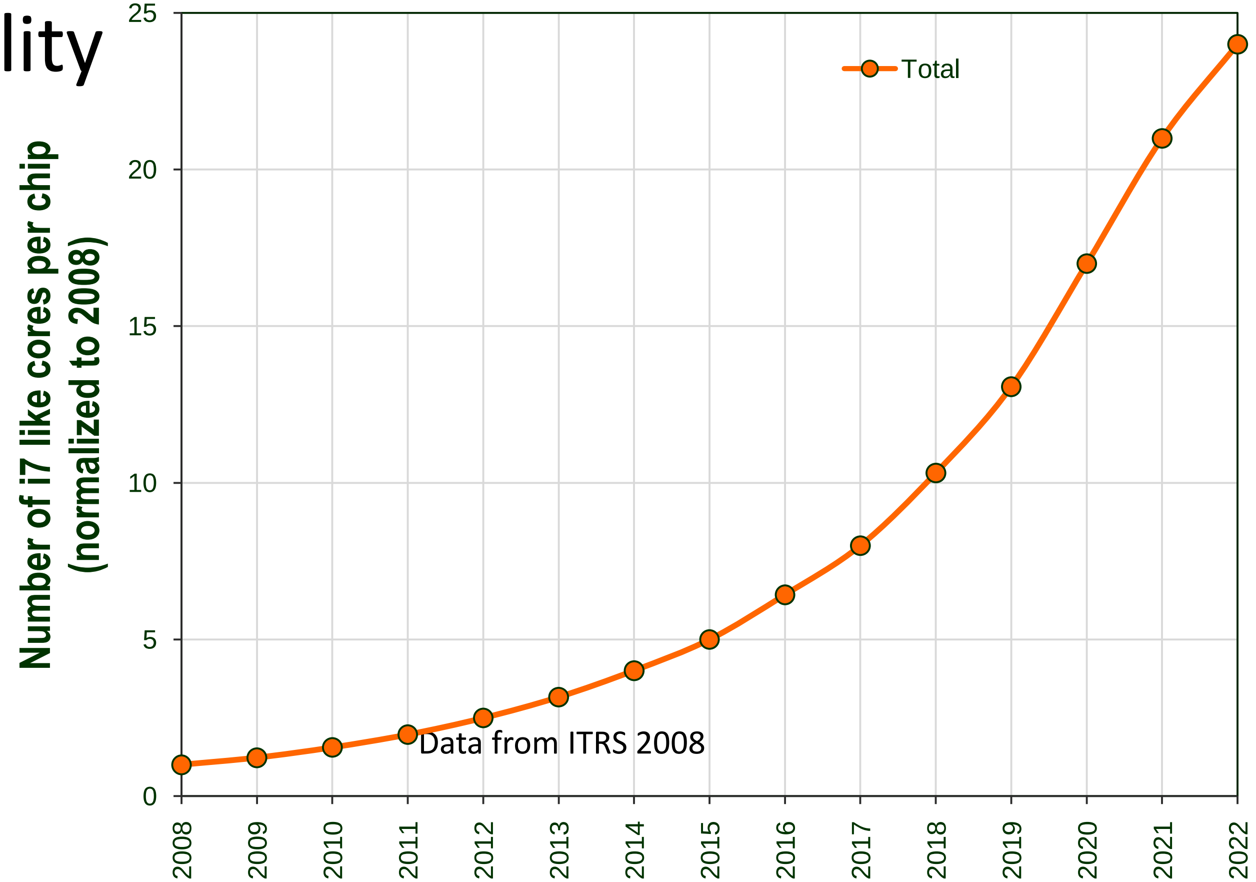
Many-core evolution



Source: Das, C. "**Interconnection networks**".
7th ACACES Summer School, 2011, Fiuggi, July 2011

Many-core evolution (cont'd)

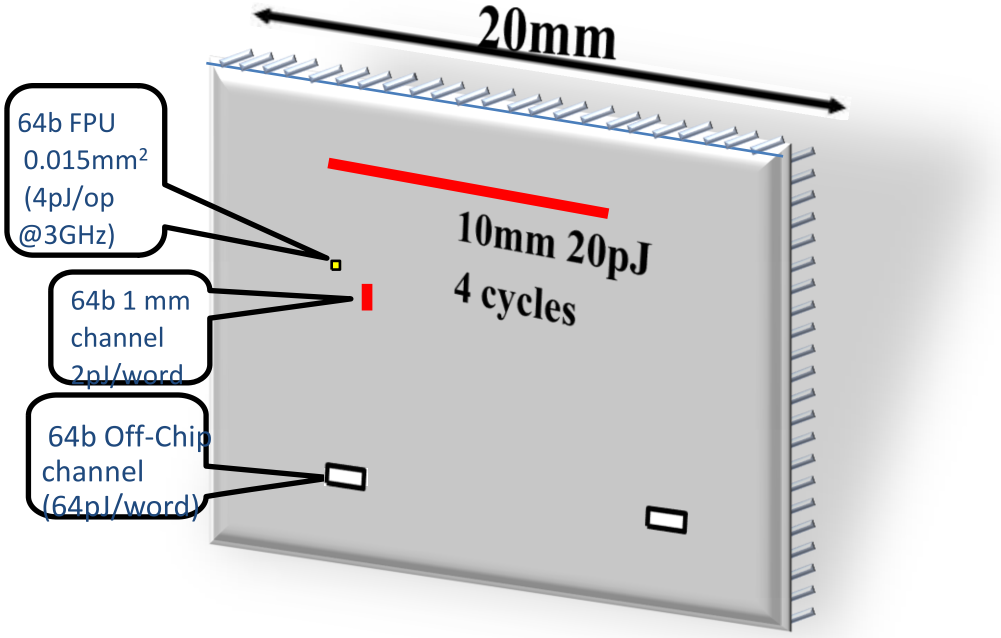
Increasing number of cores per chip demands for increasing communication bandwidth and capability

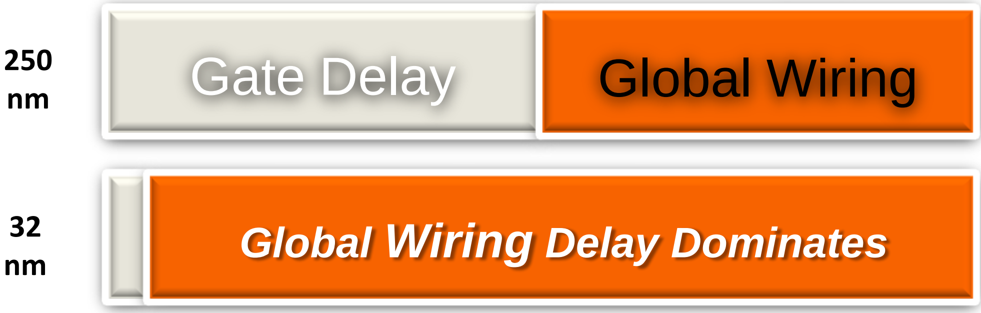


Source: International Technology Roadmap for Semiconductors, Design chapter, 2008.

Energy requirements

State-of-the-Art architecture design



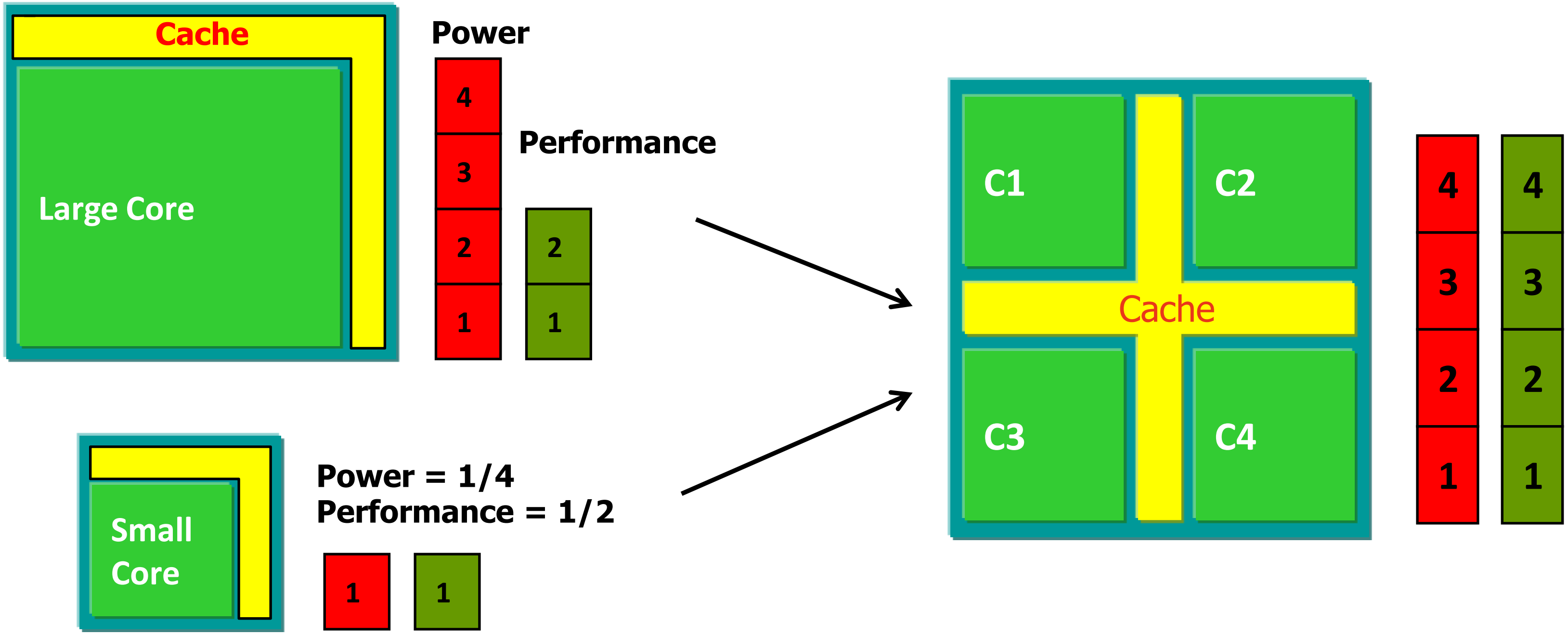


Operation	Energy (pJ)
64bIntegerAdd	1
I$ Fetch	33
Read64bRegister(64x32 bank)	3.5
Read64bRAM(64x2K)	25
Readtags(24x2K)	8
Move64b1mm	6
Move64b20mm	120
Move64boffchip	256
Read64bfromDRAM	2000

Multi-core benefits

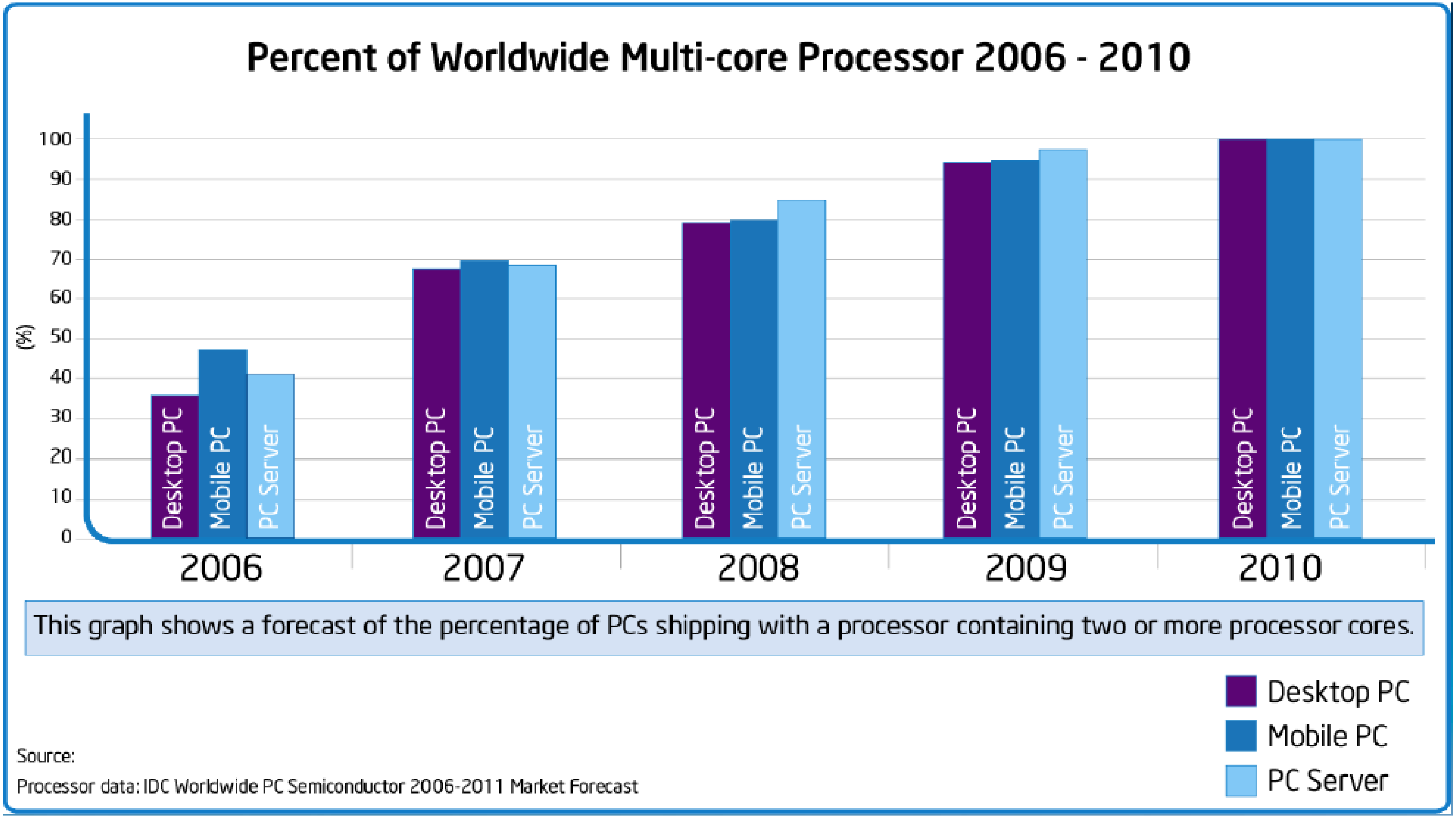
Multi-chip cores are not appropriate for power/performance optimization because of the off-chip energy requirements

Multi-core chips provide better power management



Multi-core processors adoption

- Multi-core processors are adopted world-wide



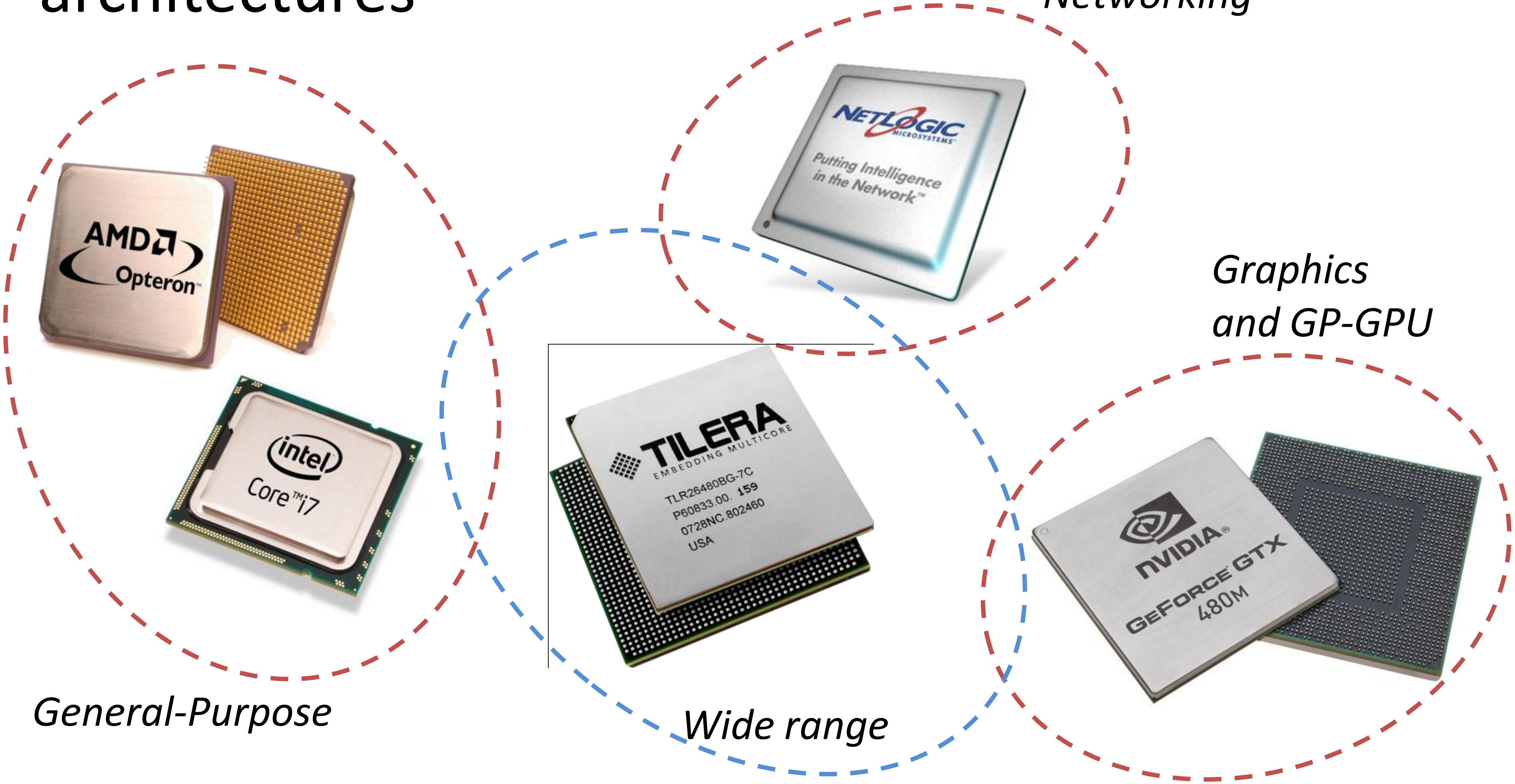
Percent of Worldwide Multi-core Processor 2006 - 2010

This graph shows a forecast of the percentage of PCs shipping with a processor containing two or more processor cores.

Source:
Processor data: IDC Worldwide PC Semiconductor 2006-2011 Market Forecast

Market segments

- Different markets for multi-core and many-core architectures



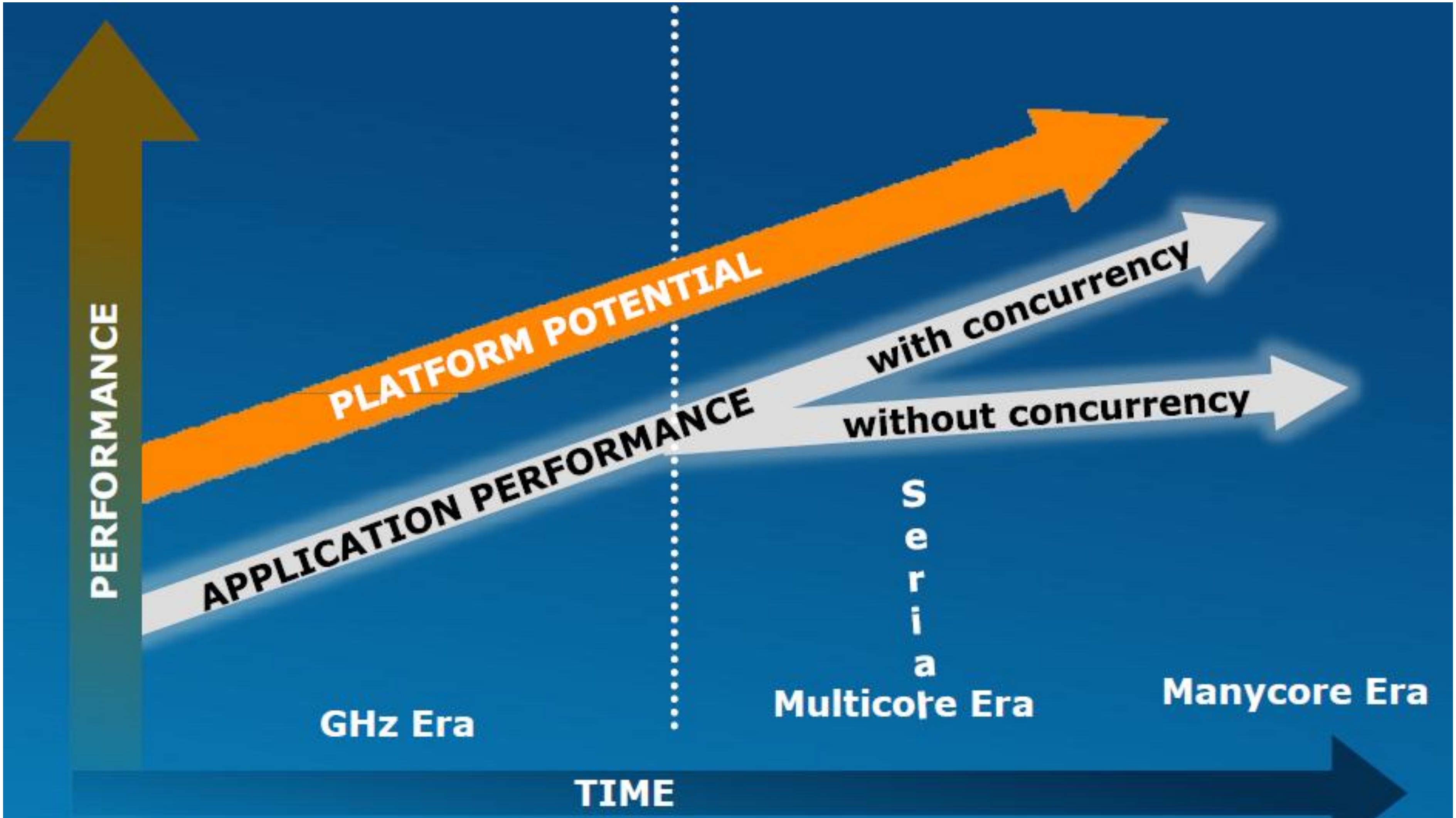
Networking

Graphics and GP-GPU

General-Purpose

Wide range

The opportunity from concurrency



Source: “Parallelism for Multicore and manycore”, James Reinders. Multicore Days ‘08, Stockholm

Parallel programming

- What should we learn from that?
  - – Current parallelism support (e.g., ILP) is no longer suitable for ensuring increasing performance requirements
  - – Need a programming paradigm shift
    - Parallel programming

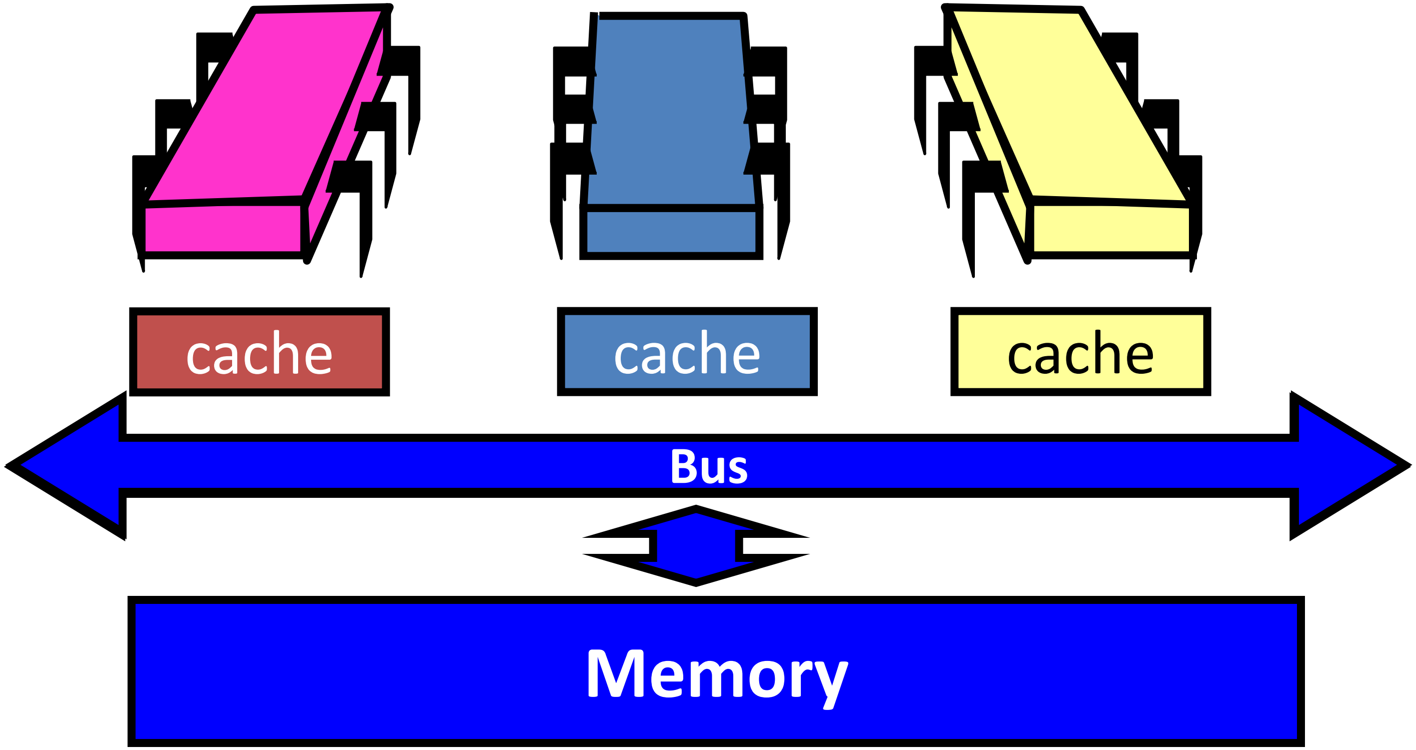
- This comes with non-negligible cost
  - – Steep (?) learning-curve

What’s next

- **Introduction**
- **Multi-core architectures**
  - Memory hierarchy
  - Interconnect design
- **Commercial examples**
- **Parallel programming issues**
  - Forms of parallelism in software (overview)
  - Main issues in parallel programming
- **Wrap-up**

Old-school multiprocessors

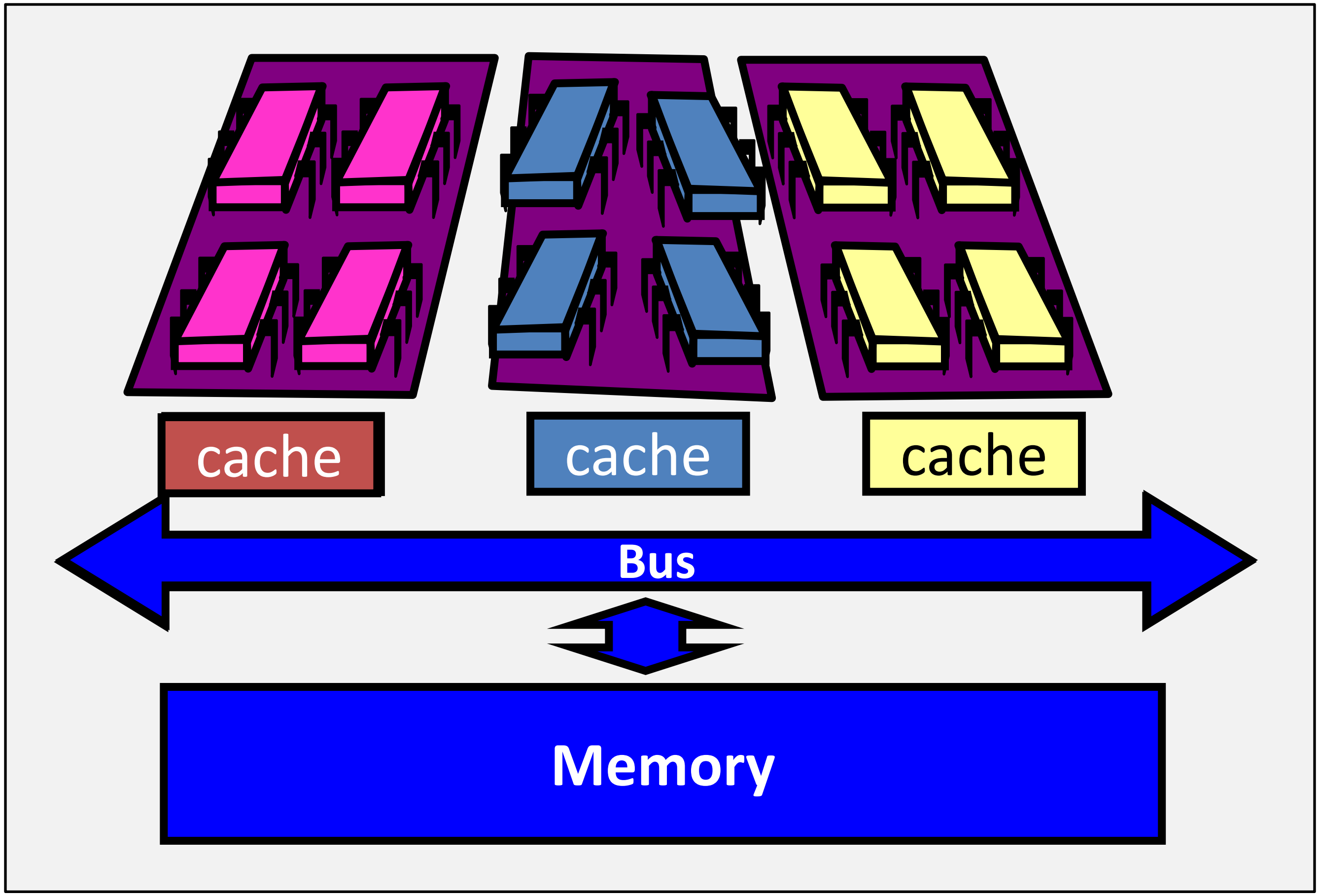
- Processors reside on different chips
  - Off-chip memory and communication resources



Source: "The Art of Multiprocessor Programming",
Herlihy, Maurice; Shavit, Nir. Morgan Kauffman

Multicore architectures

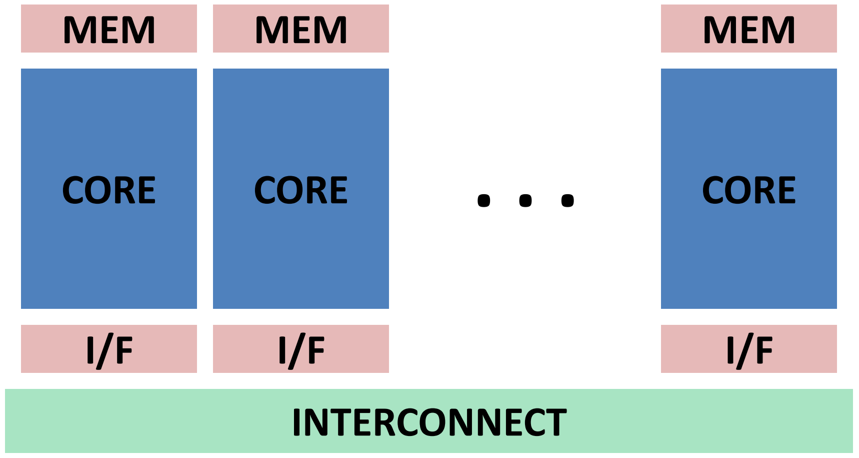
- Processors on a single chip
  - **MPSoCs**



Implementation

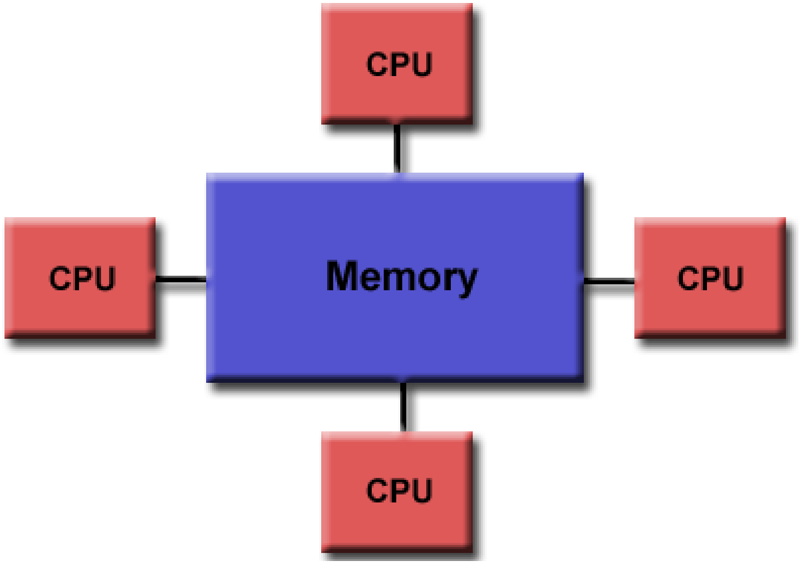
- Multi-core architectures present a variety of shapes
  - Application-specific
  - Performance-constrained
  - Ad-hoc solutions

- Challenges
  - Interconnect
  - Memory susbsystem
  - Chip layout



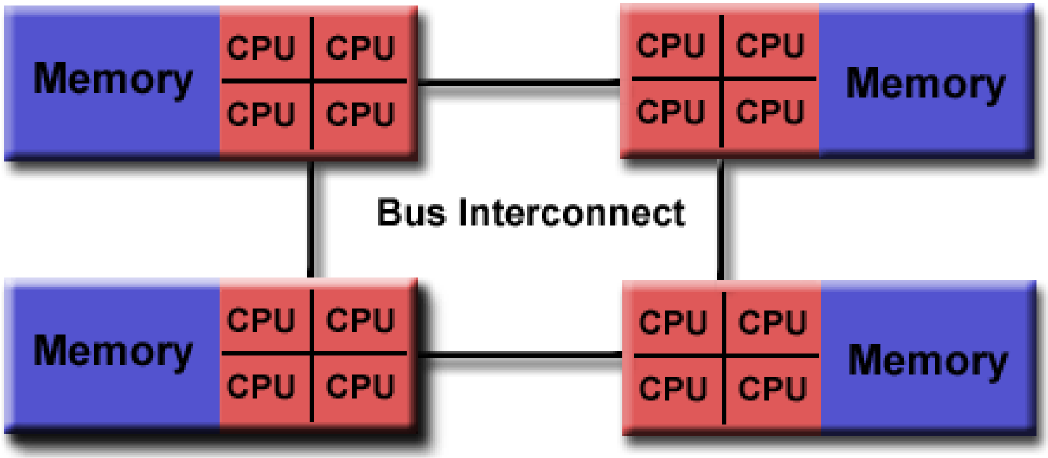
Shared memory

- A unique global address space



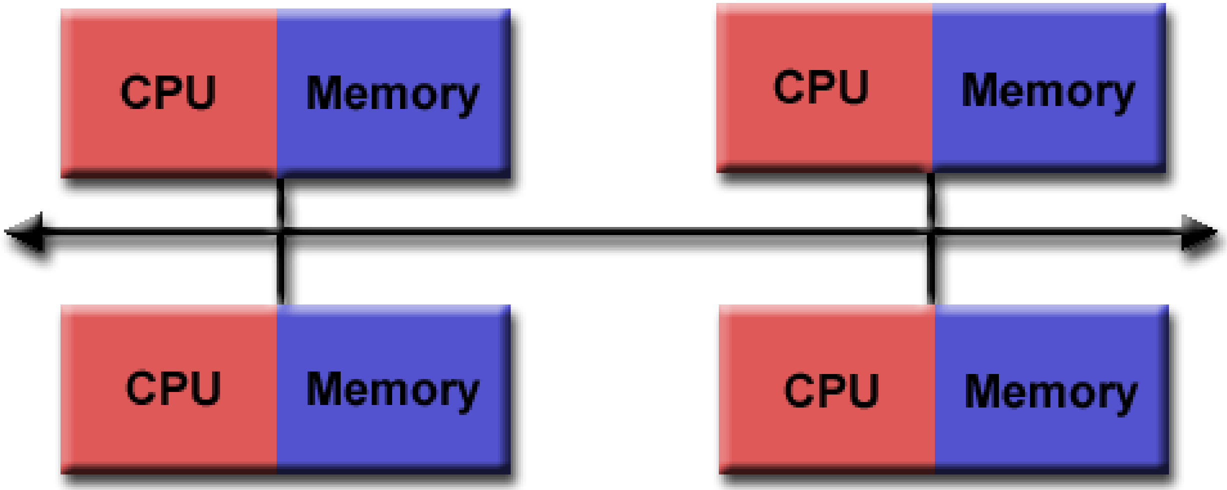
Cache-Coherent
Uniform Memory Access
(single physical bank)

(Cache-Coherent)
Non-Uniform
Memory Access
(multiple banks)



Distributed memory

- Memory addresses in one processor do not map on another processor
- Cache coherence concept does not apply

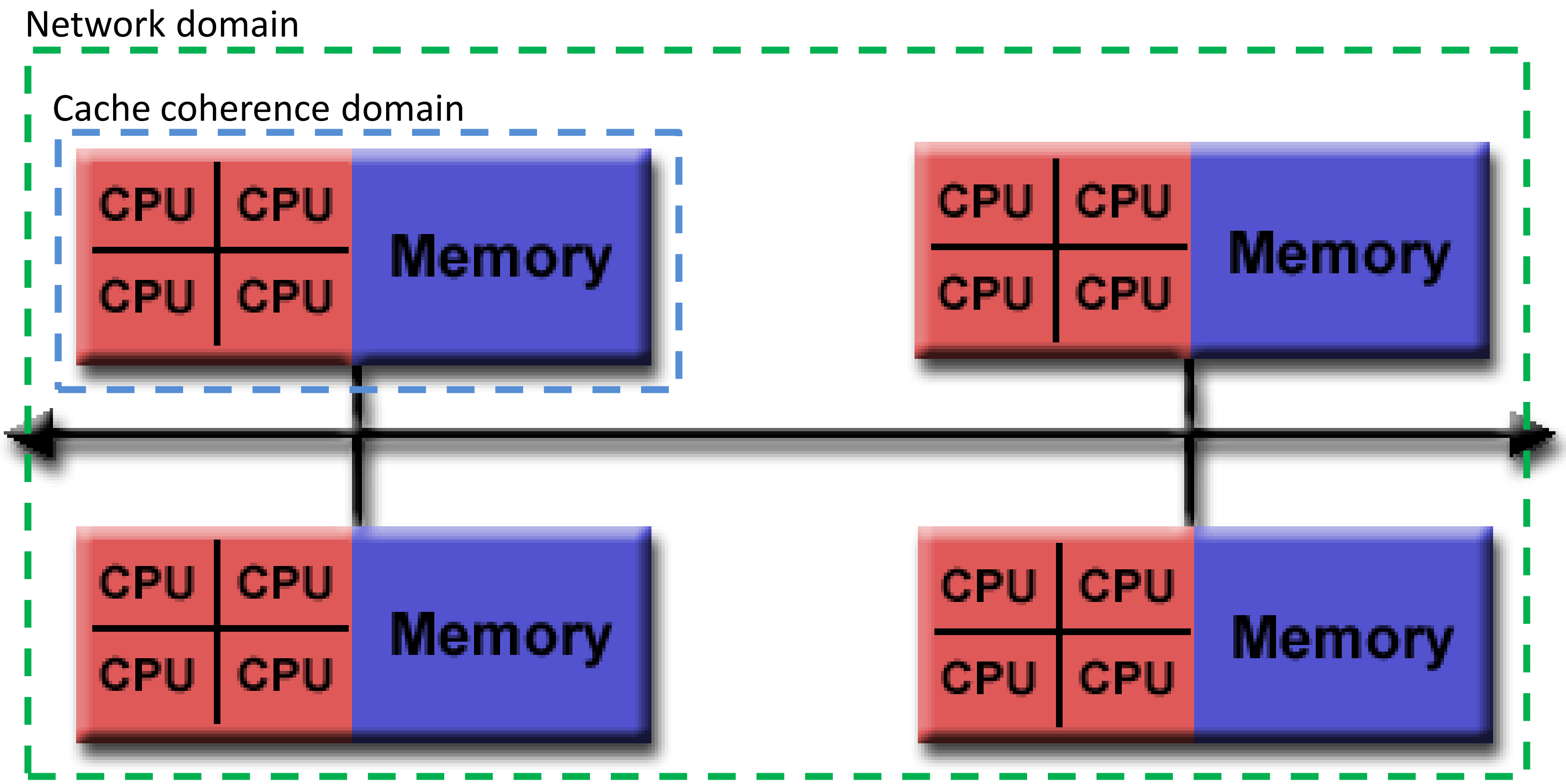


Shared versus distributed memory

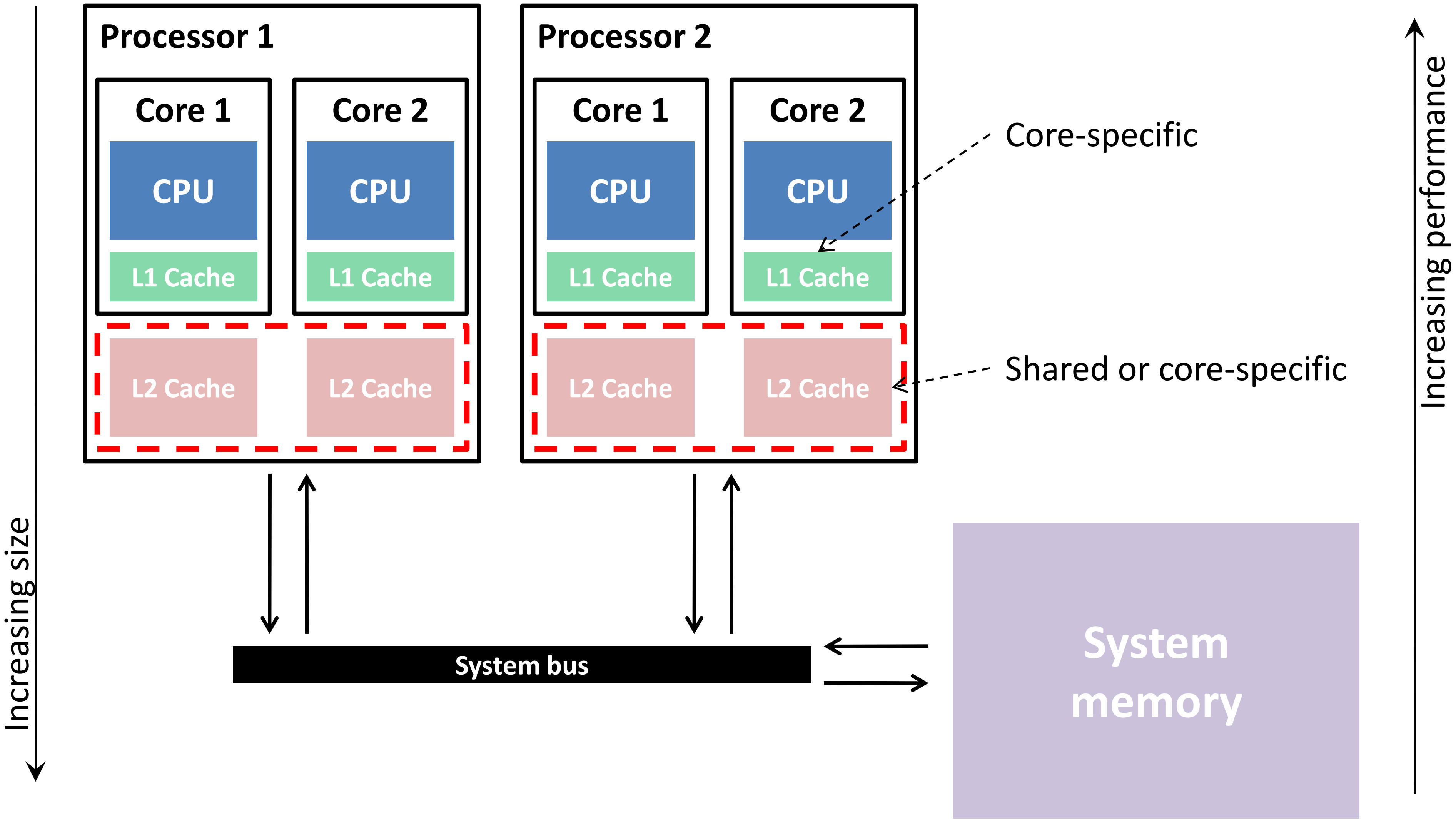
SHARED MEMORY	DISTRIBUTED MEMORY
+ Global address space simplifies programming efforts to memory + Fast data sharing mechanism	+ Scalability + Each processor can access its local memory without impacting on other processors
- Scalability (increasing number of processors increase traffic) - It is in charge of the programmer to explicitely use synchronize constructs to access shared resources - Cost with increasing number of processors	- Many of the communication details between processors is in charge of the programmer - NUMA in nature

An hybrid approach

- Clustered architecture



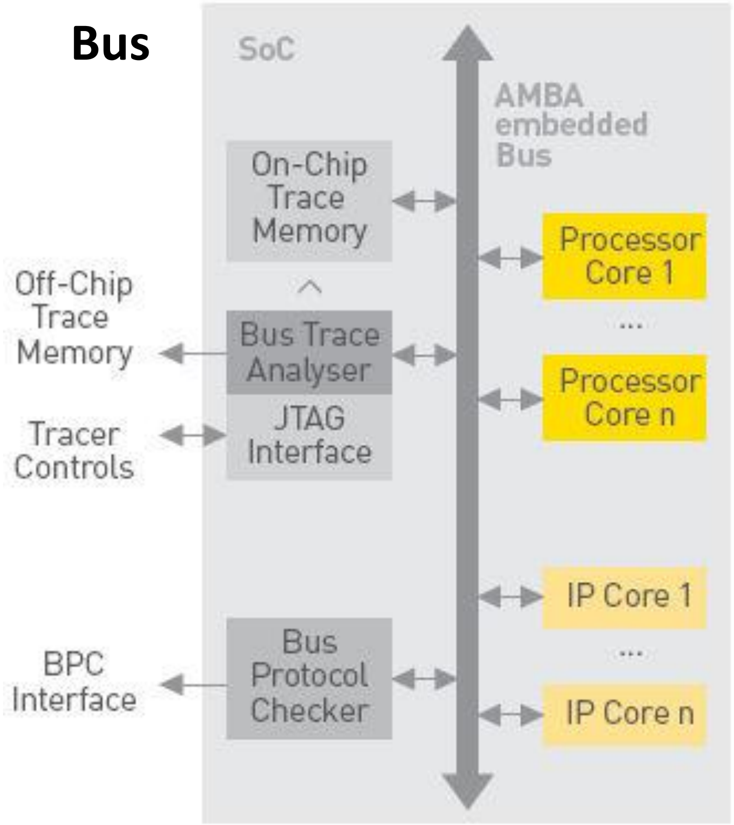
Memory hierarchy



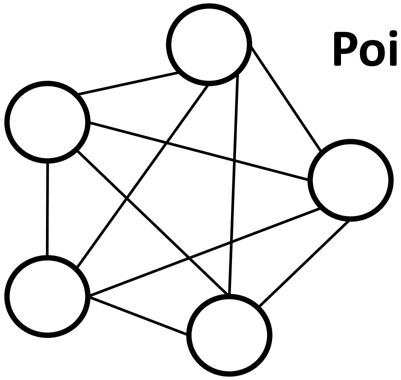
Interconnects

- Classical approaches are no longer suitable for interconnect design in MPSoC architectures

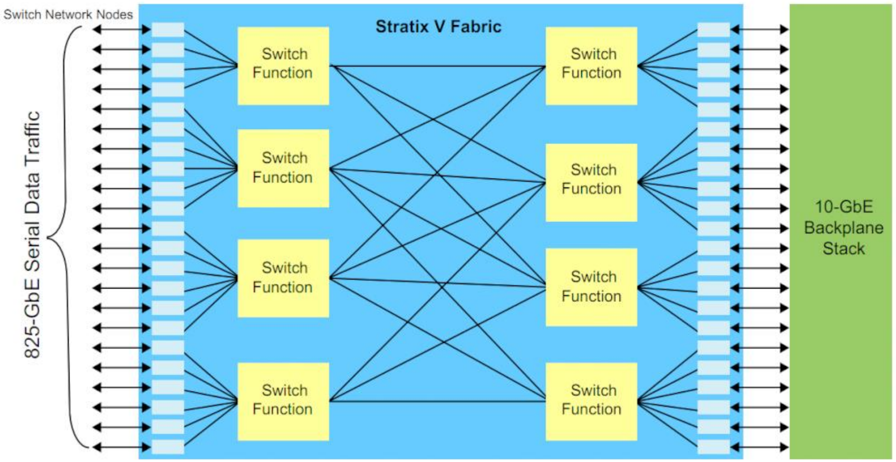
Bus



Point-to-Point

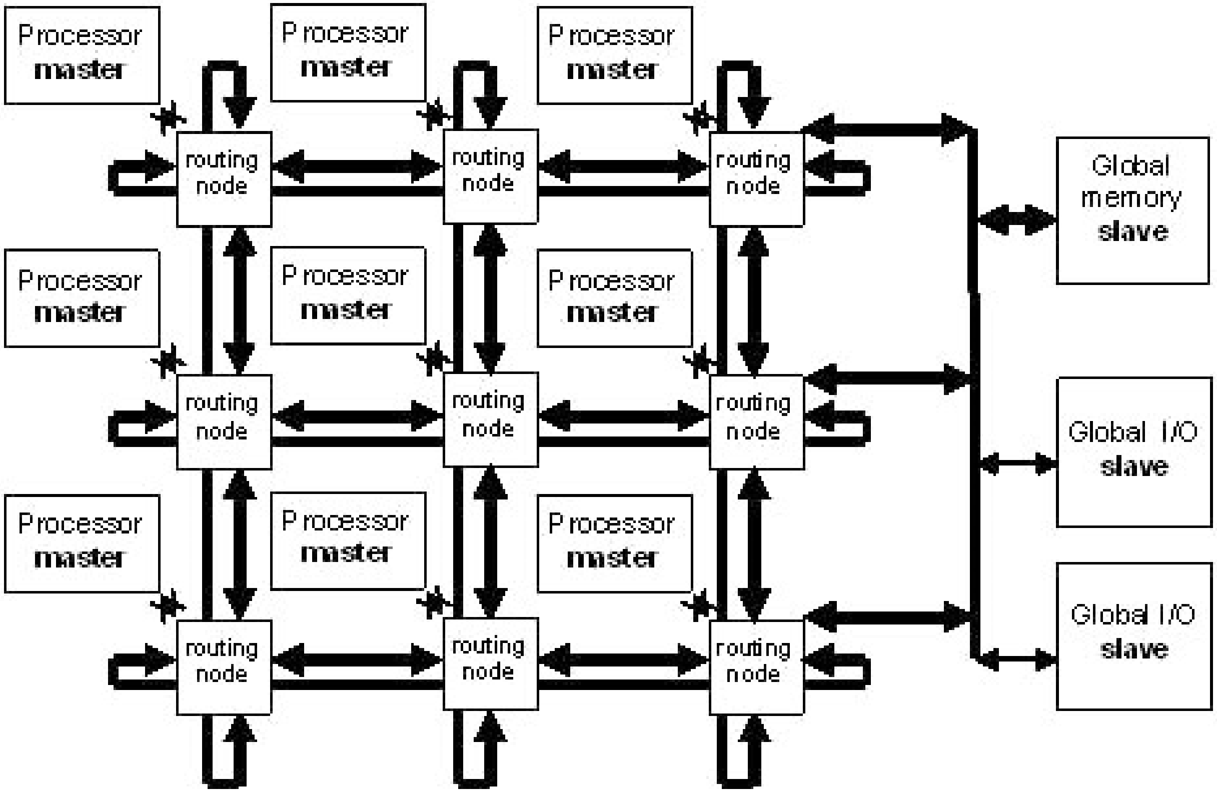


Crossbar



Network-on-Chip

- NoC is a good candidate for scalable communication architecture design

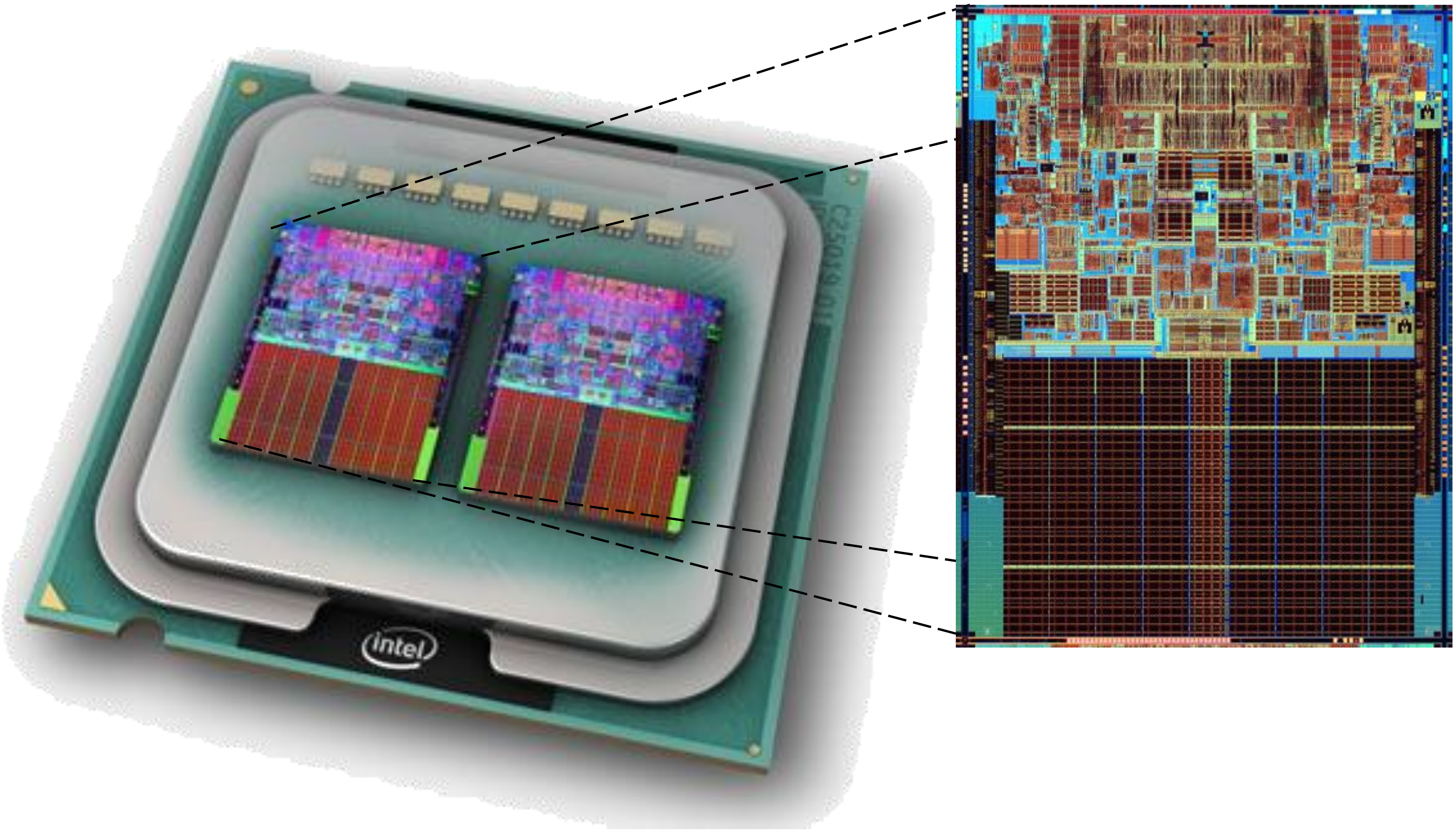


What’s next

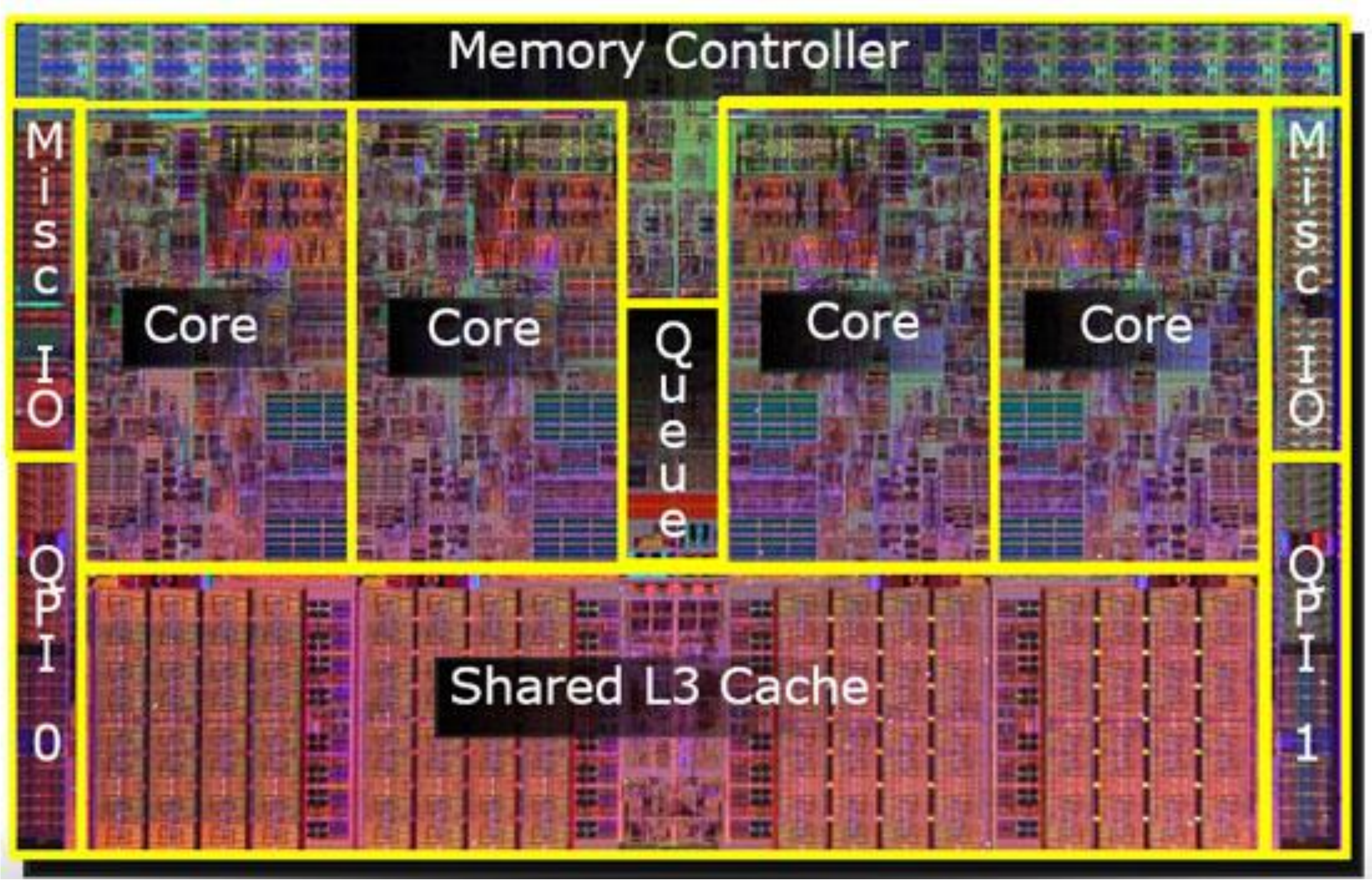
- **Introduction**
- **Multi-core architectures**
  - Memory hierarchy
  - Interconnect design
- **Commercial examples**
- **Parallel programming issues**
  - Forms of parallelism in software (overview)
  - Main issues in parallel programming
- **Wrap-up**

Commercial processors examples

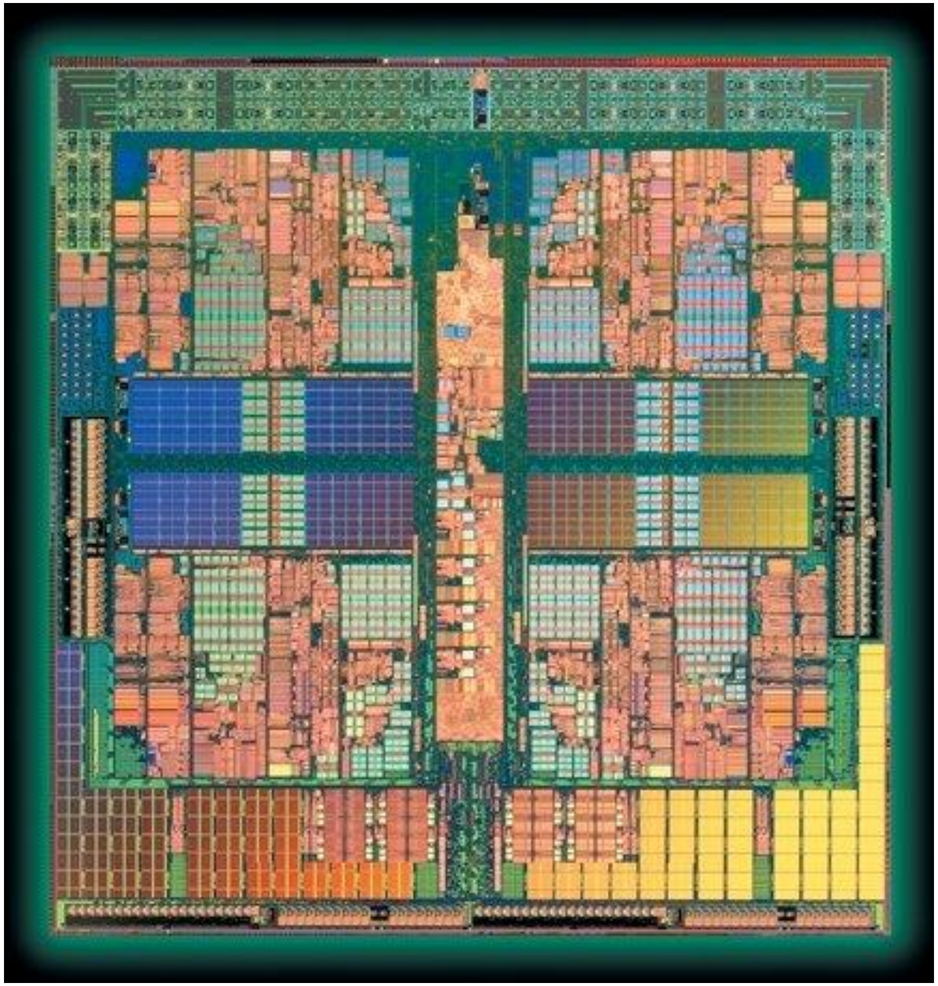
- Intel Core2-Duo Quad

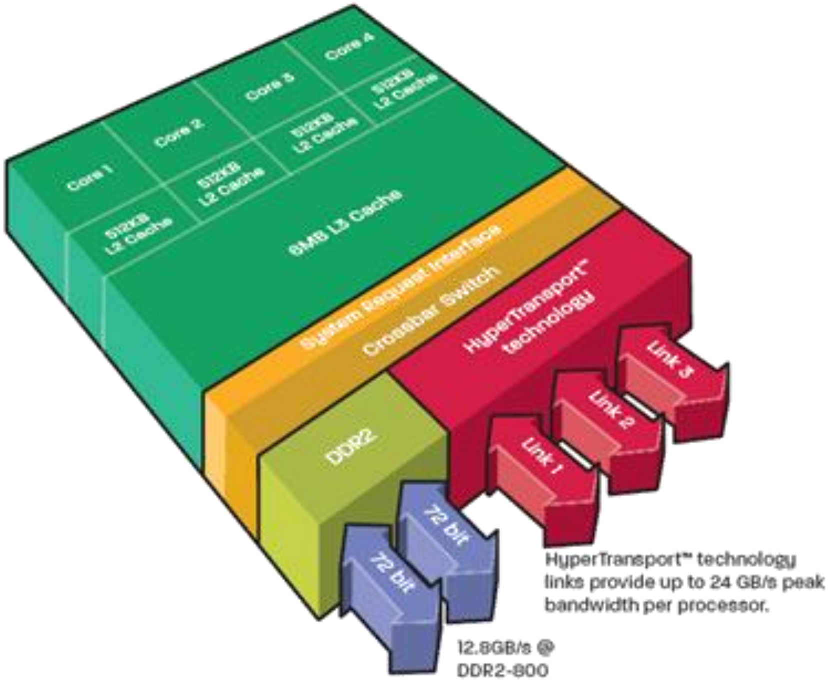


- Intel Core i7 Quad-core (Nehalem)

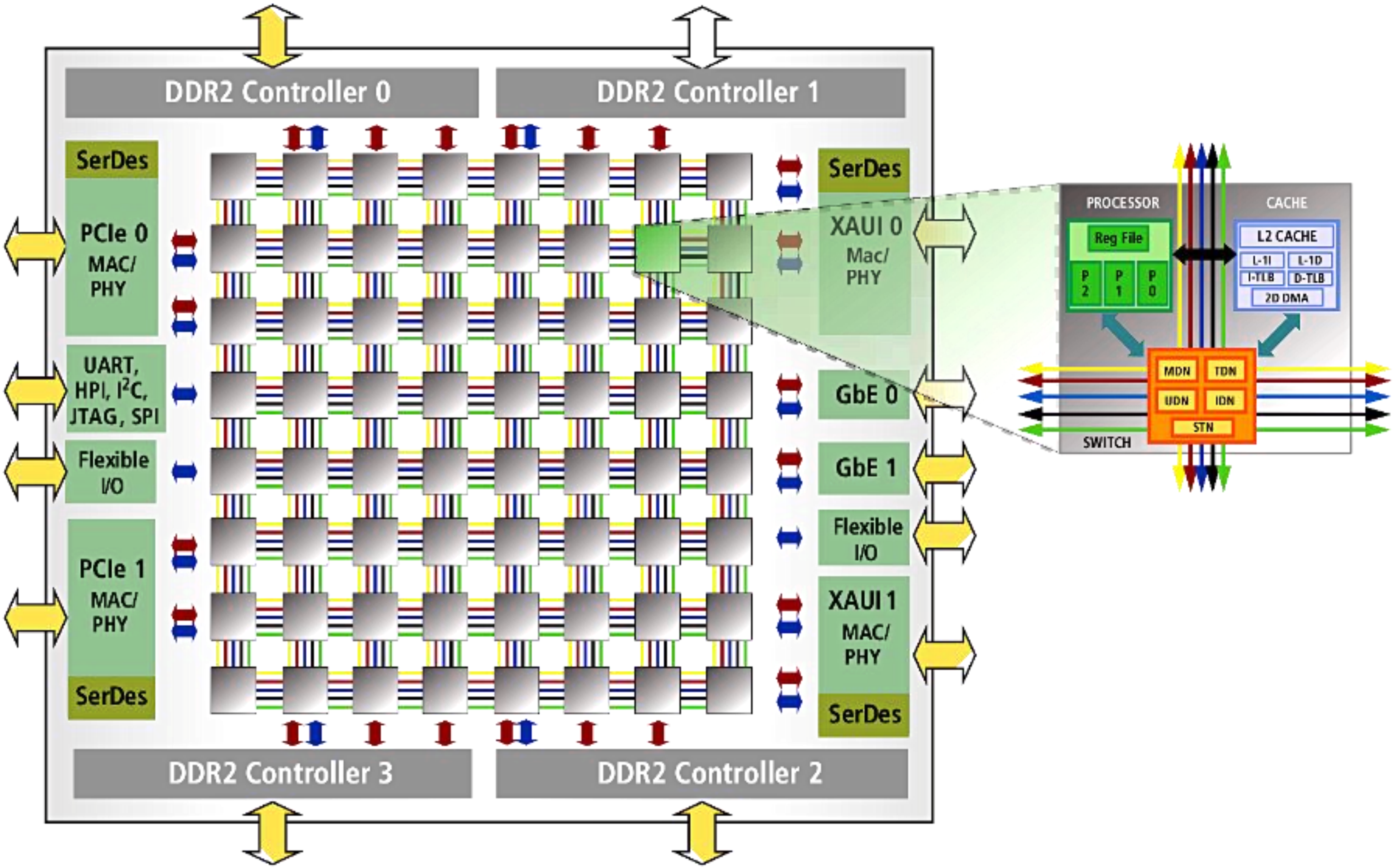


- AMD Opteron Quad-core

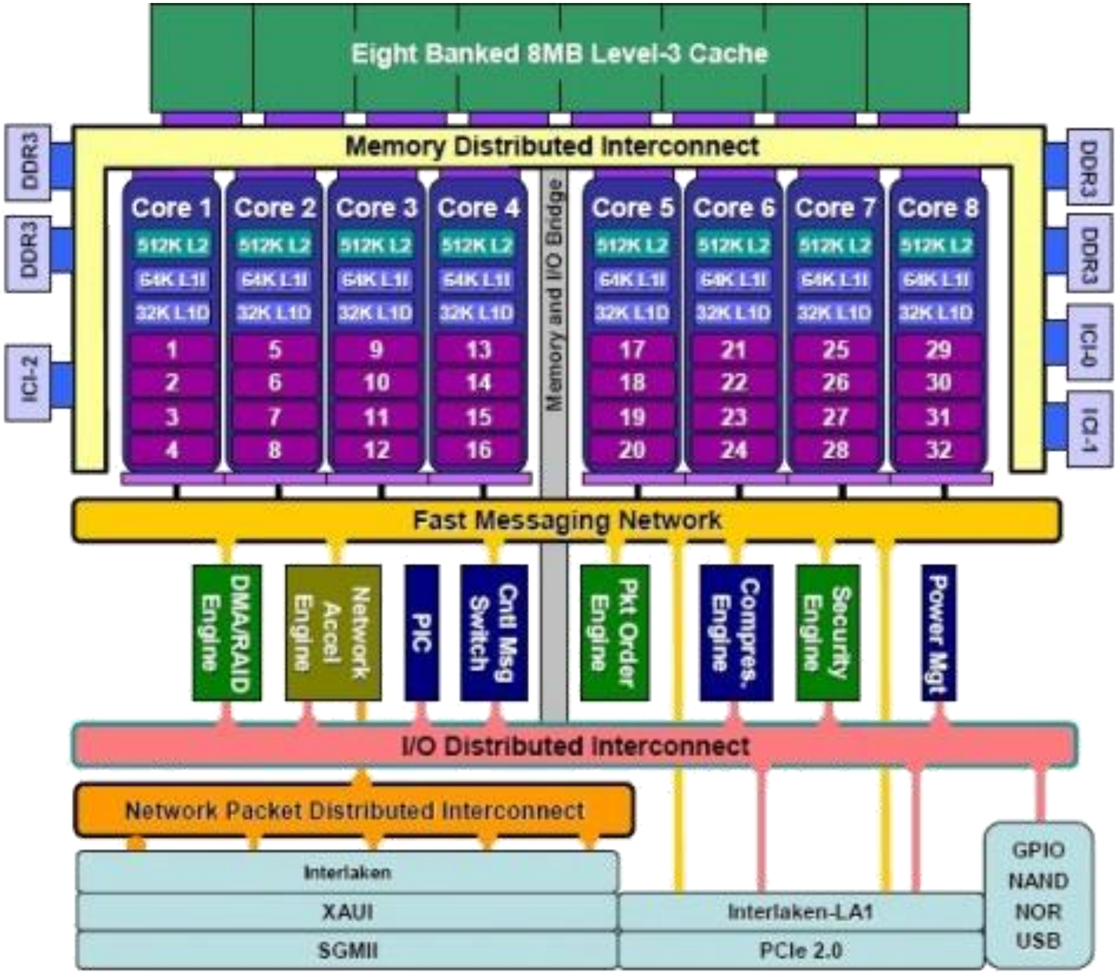




- Tilera TILE64



- NetLogic XLP series



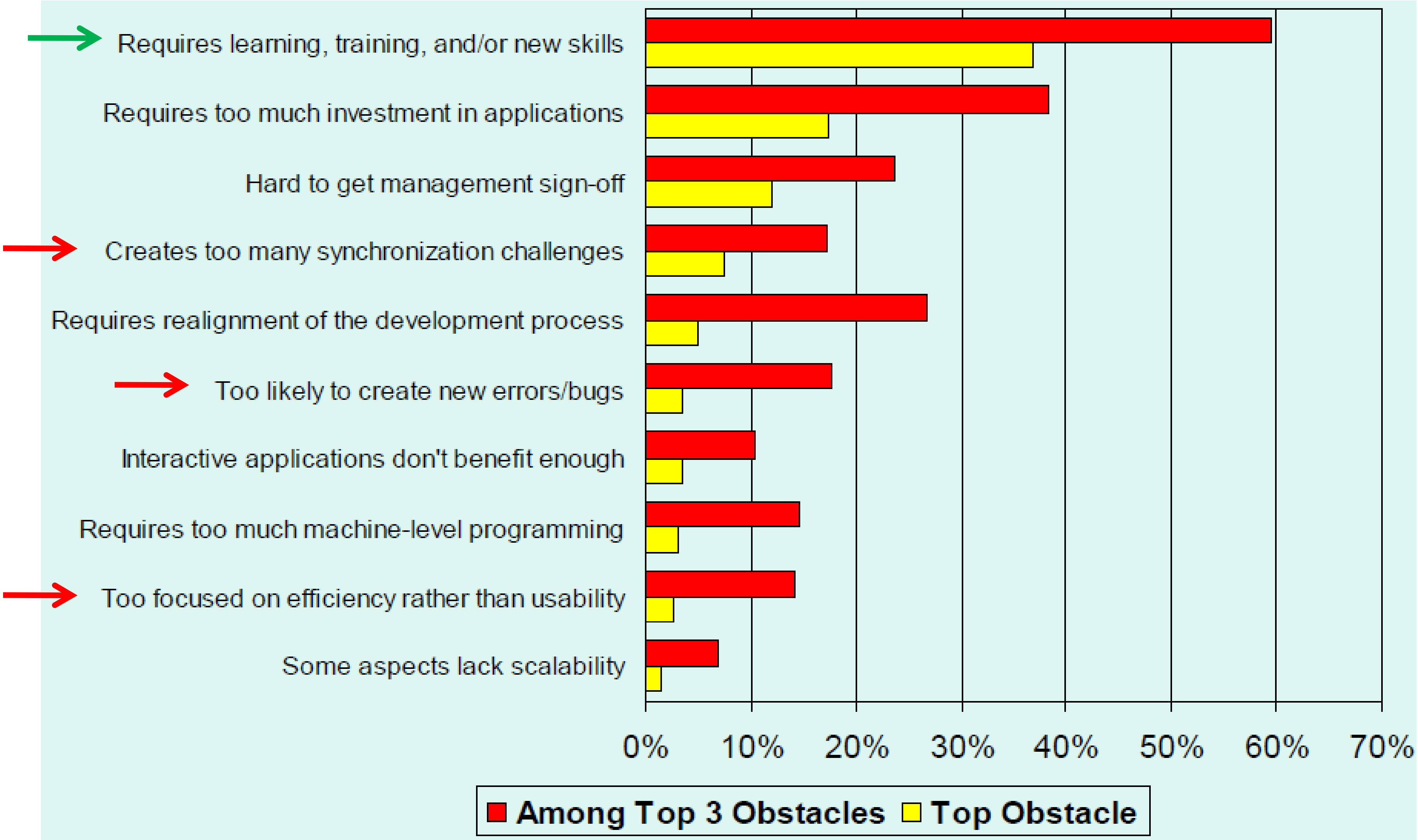
XLP832 Processor Block Diagram

What's next

- **Introduction**
- **Multi-core architectures**
  - Memory hierarchy
  - Interconnect design
- **Commercial examples**
- **Parallel programming issues**
  - Forms of parallelism in software (overview)
  - Main issues in parallel programming
- **Wrap-up**

Parallel programming efforts

Source: "Parallelism for Multicore and manycore", James Reinders. Multicode Days '08, Stockholm



Forms of parallelism in software

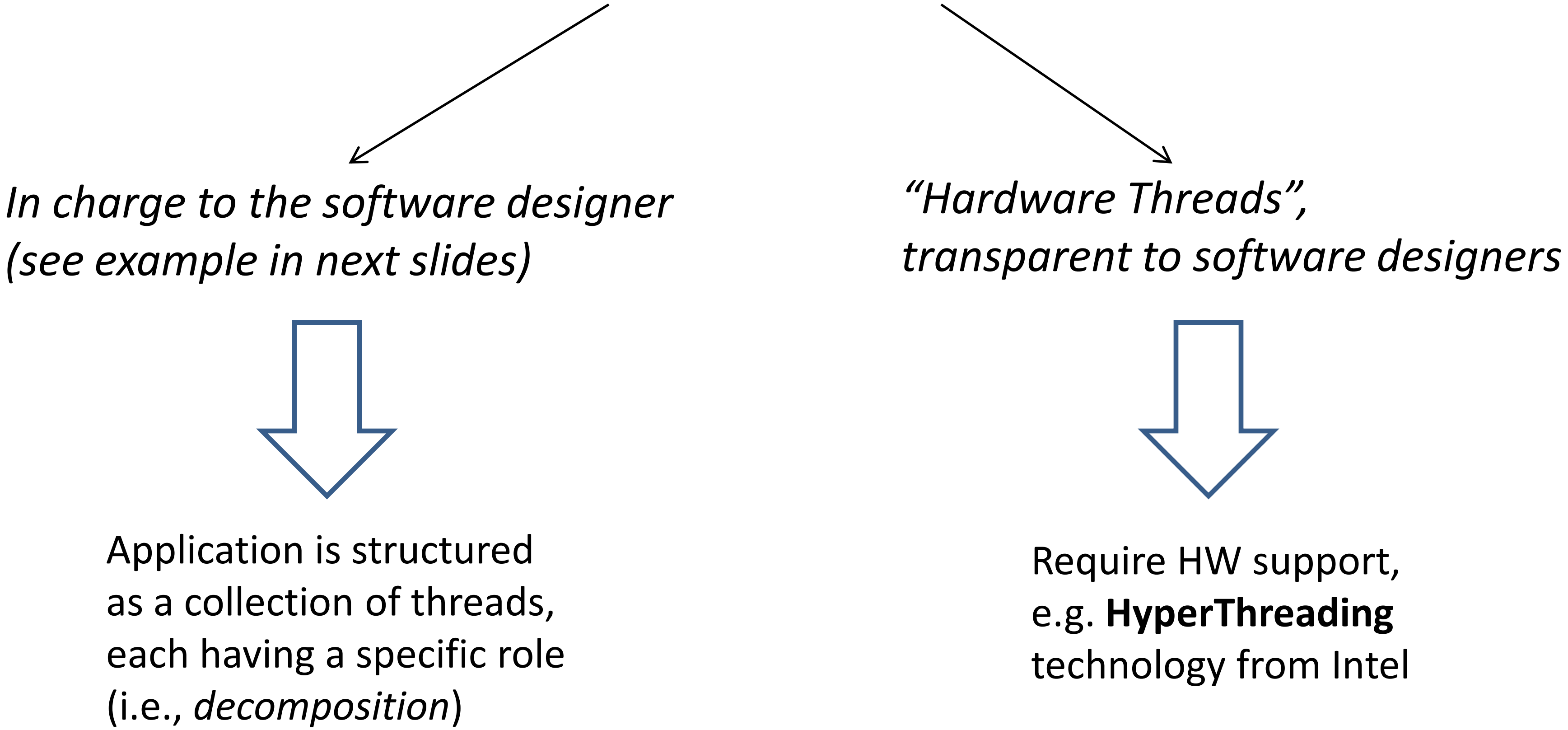
- Instruction Level Parallelism (ILP)
  - (General term) require hardware support (e.g., instruction pipelining, VLIW processors, out-of-order execution, speculative control, ...)

- Multi-threading (MT)
  - Might be implicit (HW support) or explicit (SW structure)

- Task Level Parallelism (TLP)
- Data Level Parallelism (DLP)
- Pipelining (aka streaming)

Multi-threading

- Can be either **implicit** or **explicit**



In charge to the software designer (see example in next slides)

Application is structured as a collection of threads, each having a specific role (i.e., *decomposition*)

“Hardware Threads”, transparent to software designers

Require HW support, e.g. **HyperThreading** technology from Intel

Task-Level Parallelism

- Parallelism is exploited at process-level granularity
  - Multiple applications running concurrently on different processors
  - Processes from the same application running concurrently on different processors

- Processes execute different functionalities on different data
  - Single Program Multiple Data (SPMD) or Multiple Programs Multiple Data (MPMD)

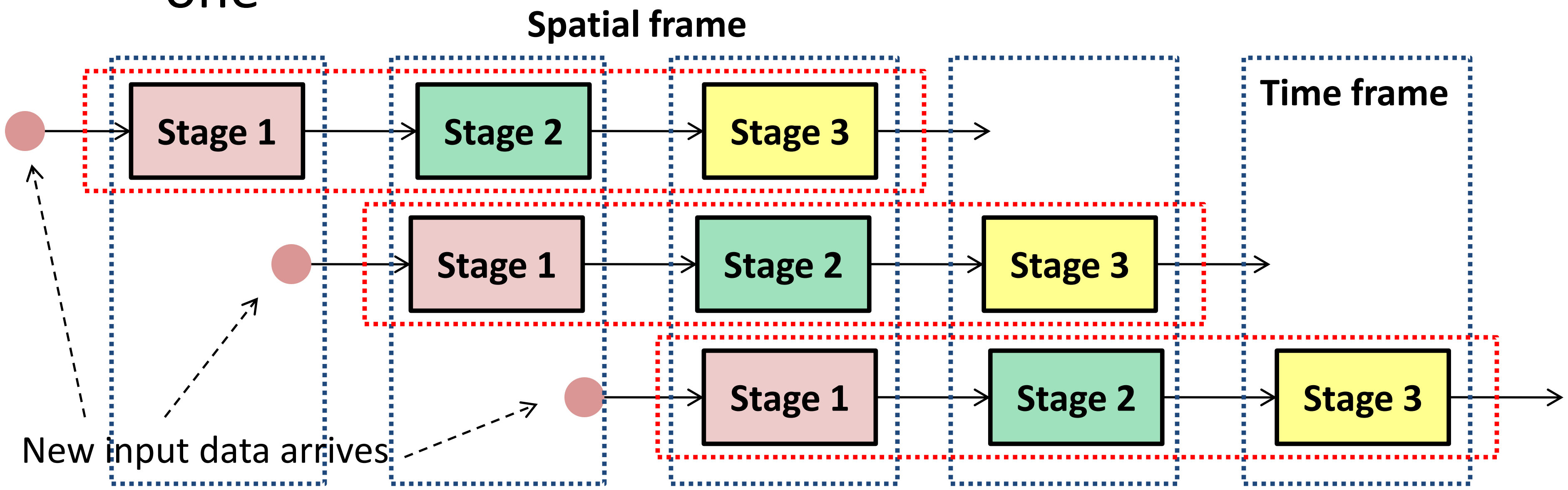
Data-Level Parallelism

- Computational load is balanced on different processes running the same functionality on different data
  - e.g., image visualization algorithms

- Parallelism is done at the data level, not at the functionality level
  - Similar to the Single Instruction Multiple Data (SIMD) cathegory defined by Flynn's taxonomy

Pipelining (streaming)

- An application is divded into elementary basic blocks (basic functionalities)
  - Applications is seen as a cascade of operations
  - Input in current stage is output from the previous one

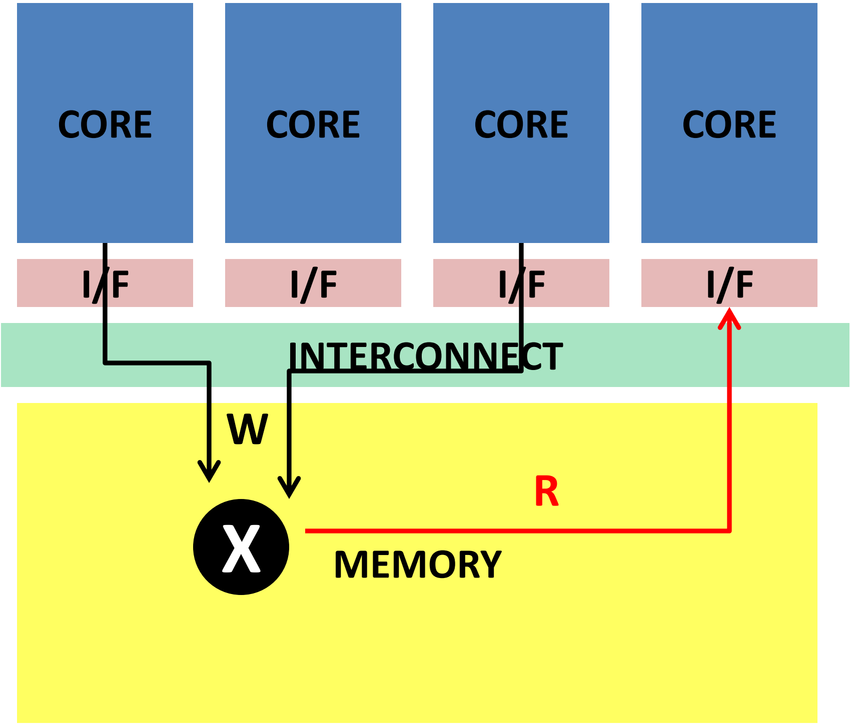


Parallel programming

- Main challenges in parallel programming
  - Access to shared resources
  - Synchronization
  - Load balance
- Solutions
  - Hardware
  - Software
  - OS-supported

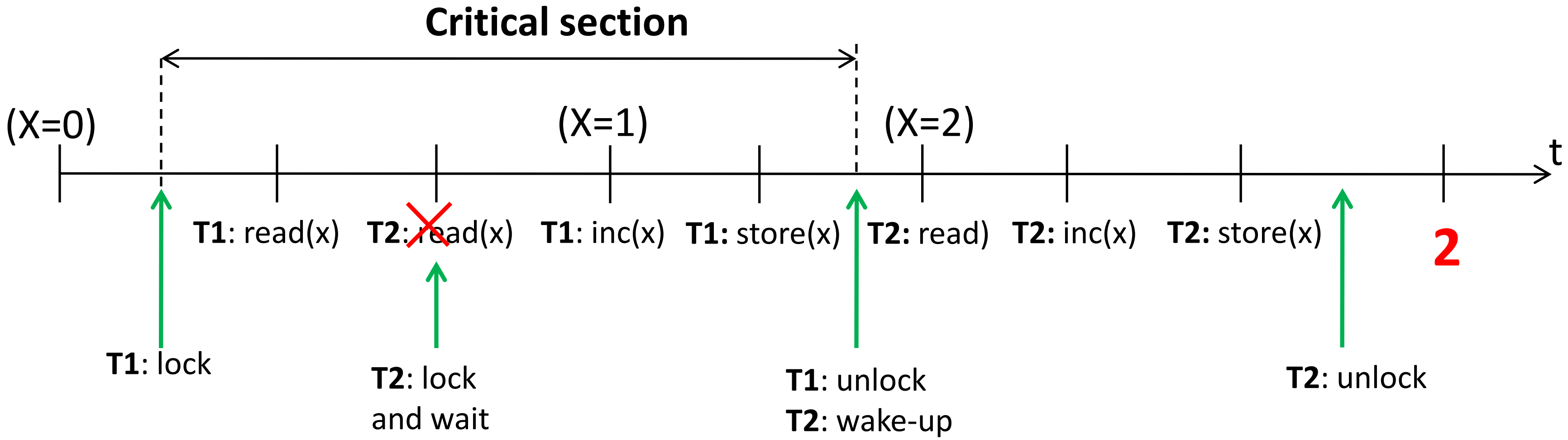
Shared resources

- A resource **X** can be shared among different processors
  - R/R operations aren't real issues
  - Issues when an operation is a W



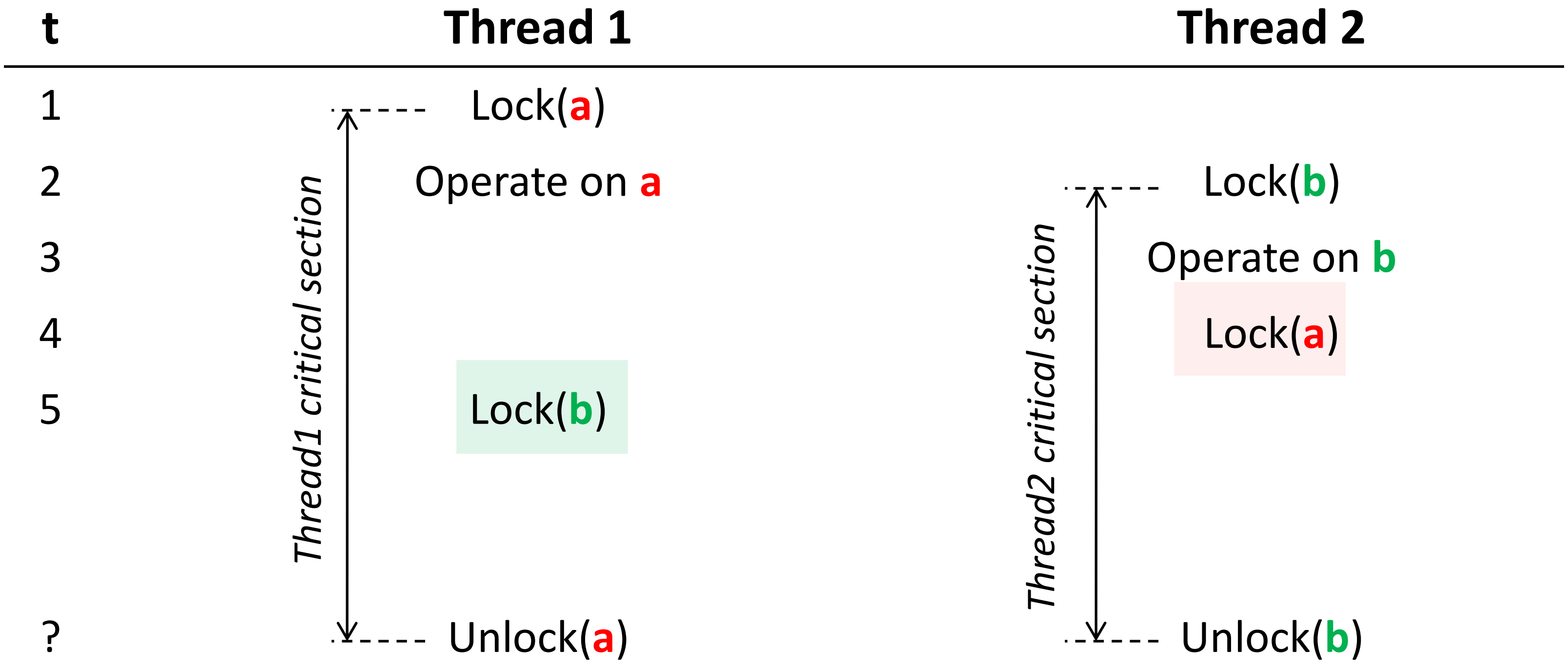
Locking resources

- A typical solution is to lock resources
  - Only one among the executing threads is allowed to access shared resource (**mutual exclusion**)
  - The other threads are blocked and wait until resource is released (unlocked)



Locking issues

- Consider the following example



t	Thread 1	Thread 2
1	Lock(**a**)	
2	Operate on **a**	Lock(**b**)
3		Operate on **b**
4		Lock(**a**)
5	Lock(**b**)	
?	Unlock(**a**)	Unlock(**b**)

POSIX locks

- POSIX definition for locks is employed in the mutex concept
- A `mutex_t` variable defines the access port to shared resources

```
#include <pthread.h>

pthread_t my_mutex = PTHREAD_MUTEX_INITIALIZER;
int shared_counter = 0;

/* Executed at generic process Pi */
pthread_mutex_lock(&my_mutex);
Shared_counter++;
pthread_mutex_unlock(&my_mutex);

/* When all done, destroy the lock */
pthread_mutex_destroy(&my_mutex);
```

RLock in POSIX standard

- What it process generic *Pi* locks twice the same lock object?
  - POSIX recursive locks can be acquired an arbitrary number of times by a process

```
#include <pthread.h>

pthread_t my_mutex;
pthread_mutexattr_t my_mutex_attr;
pthread_mutexattr_settype(&my_mutex_attr, PTHREAD_MUTEX_RECUR
SIVE);
pthread_mutex_init(& my_mutex_attr);

pthread_mutex_lock(&my_mutex);
//...
pthread_mutex_lock(&my_mutex);
//...
pthread_mutex_unlock(&my_mutex);
pthread_mutex_unlock(&my_mutex);
```

Must be called an equal number of times as *lock*()

Semaphores

- More advanced
  - Have an internal counter and block if a given number of threads attempted to hold it

- How do they work
  - Counter is decremented at each call to lock (`wait()`)
    - If counter is 0 then block
  - If counter becomes >0 (`post()`) wake up any waiting thread

The producer/consumer problem

- Generally speaking, parallelization leads to a (generalized) producer/consumer problem
  - A producer **P** generates data and send them through ad-hoc interfaces to the consumer(s) **C**
  - A consumer **C** reads incoming data and process them locally

- A producer cannot produce data if the communication via is full
- A consumer cannot read empty buffer

- Example, naive implementation with single semaphore (mutex)

PRODUCER

```
while(1)
{
      lock(LockObject);
      write(data, &buffer);
      release(LockObject);
}
```

CONSUMER

```
while(1)
{
      lock(LockObject);
      read(data, &buffer);
      release(LockObject);
}
```

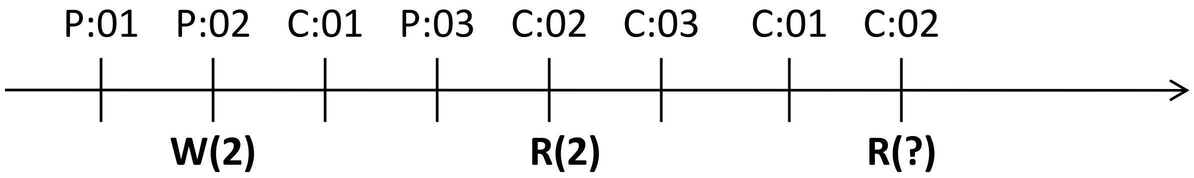
- Assume shared buffer is an integer
  - What happens in next execution scenario?

PRODUCER

```
while(1)
{
01    lock(LockObject);
02    write(data, &buffer);
03    release(LockObject);
}
```

CONSUMER

```
while(1)
{
01    lock(LockObject);
02    read(data, &buffer);
03    release(LockObject);
}
```



```
  lock: NONE
buffer: EMPTY
```

- Previous example shows how a naive solution with a single locking mechanism is not suitable
  - Race condition is **not** solved

- Generally, the producer/consumer problem cannot be solved using one semaphore
  - We need two semaphores to keep track of free and full buffer events

- `wait()` and `post()` are semaphore-related functions, equivalent to previous *lock* and *unlock*

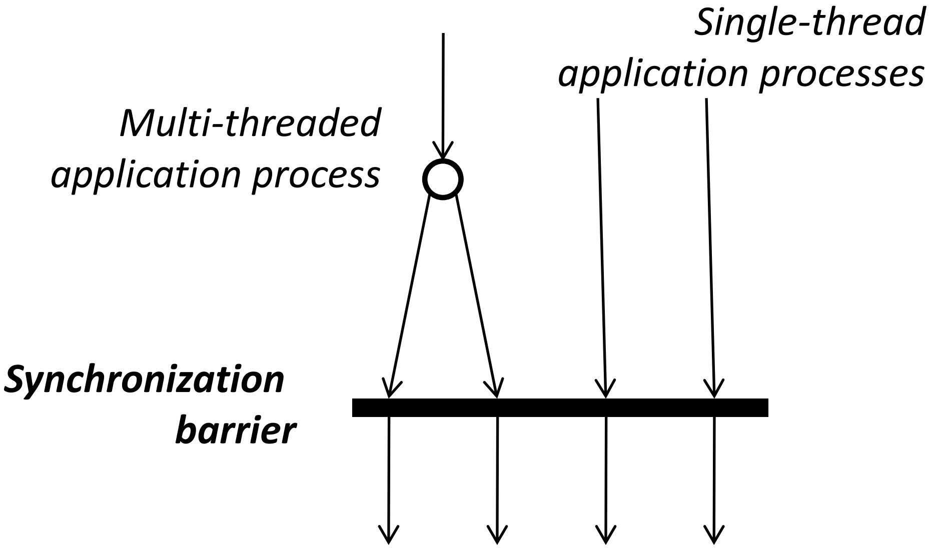
```
INIT

initialize(empty_buffer,1);
Initialize(full_buffer,0);
```

```
PRODUCER

while(1)
{
    wait(empty_buffer);
    write(data, &buffer);
    post(full_buffer);
}
```

```
CONSUMER

while(1)
{
    wait(full_buffer);
    read(data, &buffer);
    post(empty_buffer);
}
```

Barriers

- Barriers are used to synchronize a set of processes up to a specific point
  - Processes reaching the barrier are allowed to continue execution if **all** processes reached that barrier


Single-thread application processes
Multi-threaded application process
Synchronization barrier

Enforcing mutual exclusion

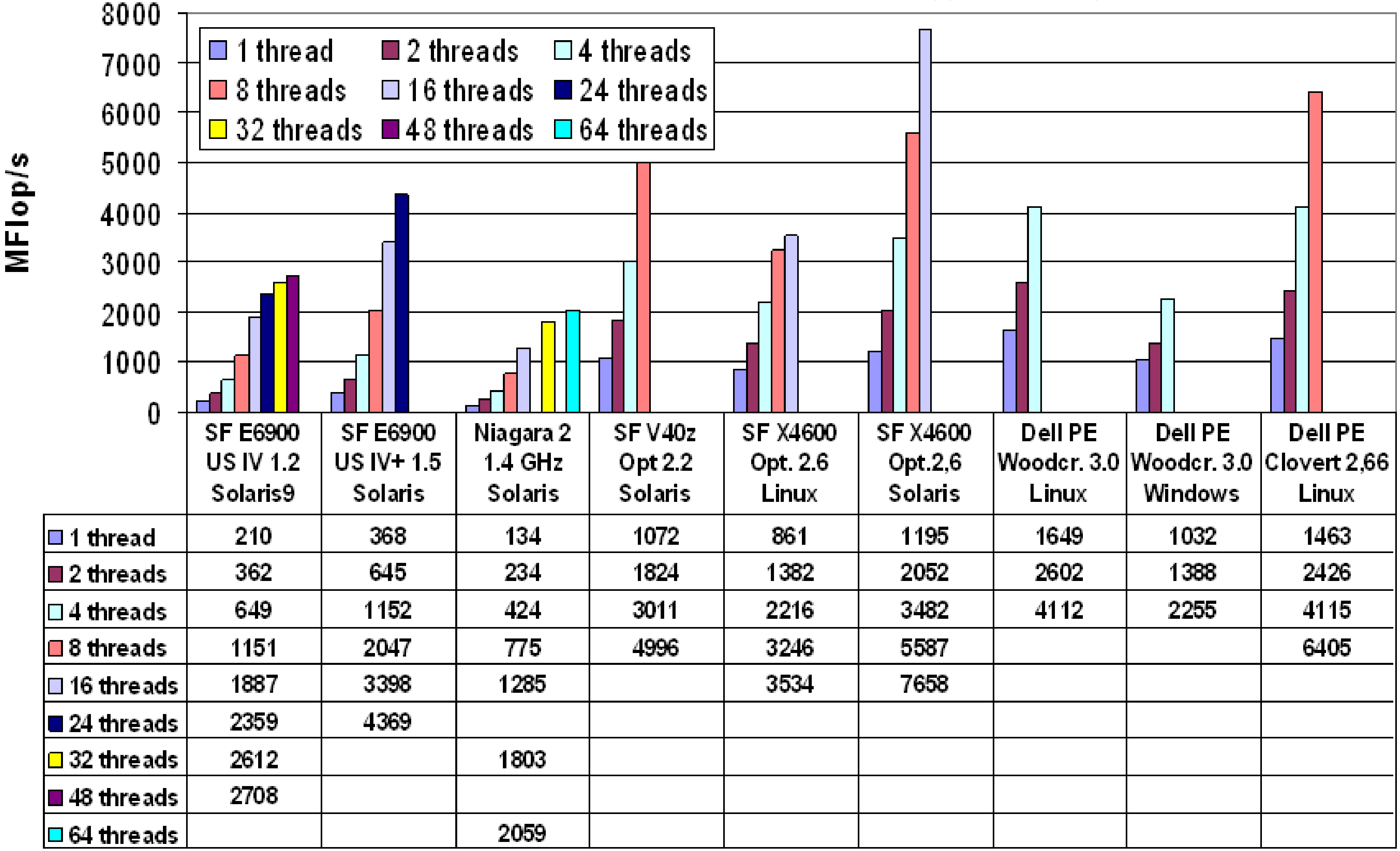
- **Hardware** solutions
  - Disable interrupts, avoiding ISR code to be executed while in the CS and avoiding context switch
    - Not suitable for multiprocessors
  - Test&Set instruction (on shared memory)
    - Spinlock
  - Compare&Swap
  - Hardware synchronizer for MPSoC architectures
- **Software** solutions
  - Locks
  - Mutexes (reentrant mutexes)
  - Semaphores

What’s next

- **Introduction**
- **Multi-core architectures**
  - Memory hierarchy
  - Interconnect design
- **Commercial examples**
- **Parallel programming issues**
  - Forms of parallelism in software (overview)
  - Main issues in parallel programming
- **Wrap-up**

Speedup examples

Source: "Application Performance", RTWH-Aachen



	SF E6900 US IV 1.2 Solaris9	SF E6900 US IV+ 1.5 Solaris	Niagara 2 1.4 GHz Solaris	SF V40z Opt 2.2 Solaris	SF X4600 Opt. 2.6 Linux	SF X4600 Opt.2,6 Solaris	Dell PE Woodcr. 3.0 Linux	Dell PE Woodcr. 3.0 Windows	Dell PE Clovert 2,66 Linux
■ 1 thread	210	368	134	1072	861	1195	1649	1032	1463
■ 2 threads	362	645	234	1824	1382	2052	2602	1388	2426
□ 4 threads	649	1152	424	3011	2216	3482	4112	2255	4115
■ 8 threads	1151	2047	775	4996	3246	5587			6405
□ 16 threads	1887	3398	1285		3534	7658			
■ 24 threads	2359	4369							
■ 32 threads	2612		1803						
■ 48 threads	2708								
■ 64 threads			2059						

The Multicore Association

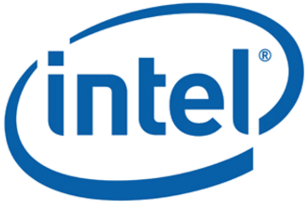
- An open membership organization including companies implementing products embracing multicore technology

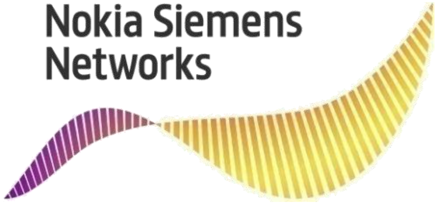
- To establish an industry-supported set of multicore programming practices and services

Copyright The Multicore Association^{TM}

http://www.multicore-association.org/home.php

- (Some of the) relevant parnters





References

[1] Culler, David E.; Singh, Jaswinder Pal. "Parallel Computer Architecture: A Hardware/Software Approach", Morgan Kaufmann

[2] Hennessy, John L.; Patterson, David A. "Computer Architecture: A Quantitative Approach", Elsevier

[3] Herlihy, Maurice; Shavit, Nir. "The Art of Multiprocessor Programming", Morgan Kaufmann

[4] Shavit, Nir; Korland, Guy; Cohen, Hila. Slideset and lecture notes for the "*Multicore Programming*" course at Tel Aviv University, Spring 2010. Available at http://www.cs.tau.ac.il/~multi/?p=slides

[5] Nagarajan, Rjagopal. "Multicore technologies and software challenges" Design article. Available at http://www.eetimes.com/design/eda-design/4008860/Multicore-technologies-and-software-challenges?cid=NL_Embedded&Ecosystem=embedded

[6] Daily, Steve (Intel Co.). "Software Design Issues for Multi-core/Multiprocessor Systems" Design article. Available at http://www.eetimes.com/design/programmable-logic/4006624/Software-Design-Issues-for-Multi-core-Multiprocessor-Systems?cid=NL_Embedded&Ecosystem=embedded

[7] Culler, David E.; Singh, Jaswinder Pal. "Matrix Operation on the GPU", Slideset and lecture notes available at http://www.seas.upenn.edu/~cis665/LECTURES/Lecture11.ppt

[8] Mitchell, Mark; Oldham, Jeffrey; Samuel, Alex. "Advanced Linux Programming", Freely available at http://www.advancedlinuxprogramming.com/downloads.html

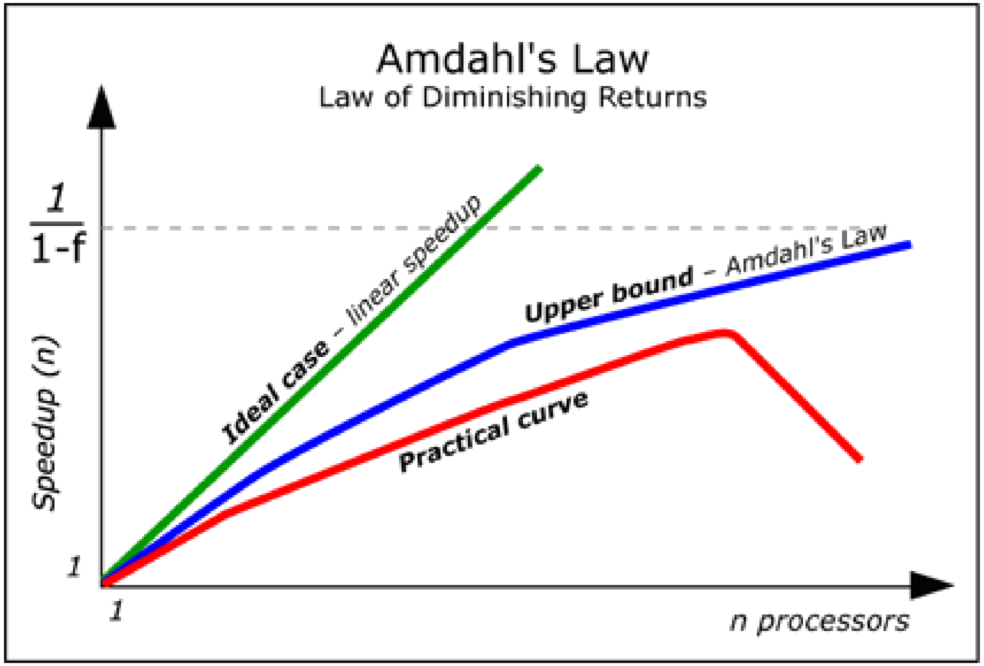
Amdahl’s Law

$$Speedup = \frac{1}{(1-P)+\frac{P}{S}}$$

Speedup of an algorithm due to an improvement with speedup *S* to the parallel portion *P*

$$Speedup = \frac{1}{(1-P)+\frac{P}{N}}$$

Speedup of a program due to an increased number *N* of processors



Source: Hennessy, J.; Patterson, D.A. “Computer Architecture – A Quantitative Approach”, Morgan Kaufman, 2007