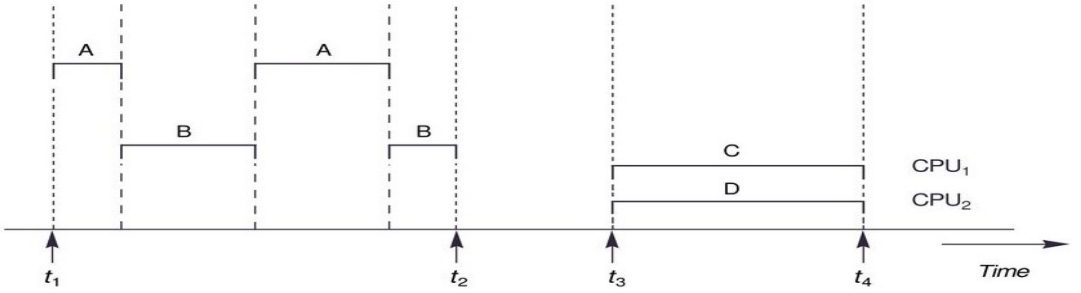
Analysis and modeling of

Computational Performance

Multithreading

- Multithreading
  - several instruction streams (threads), related to a single process
    - multithreading concerns mutiple threads managed by a single instance of an operating system and having access to a shared address space (shared memory)
    - threads execution can be concurrent only: or parallel:



  - parallel execution requires hardware support
    - there are two main types of hardware for multithreading:
      - multi-core microprocessors
      - multi-socket motherboards
      - (multi-processor designs with many motherboards connected using fast communication links are much less common)

Threads versus processes (recall)

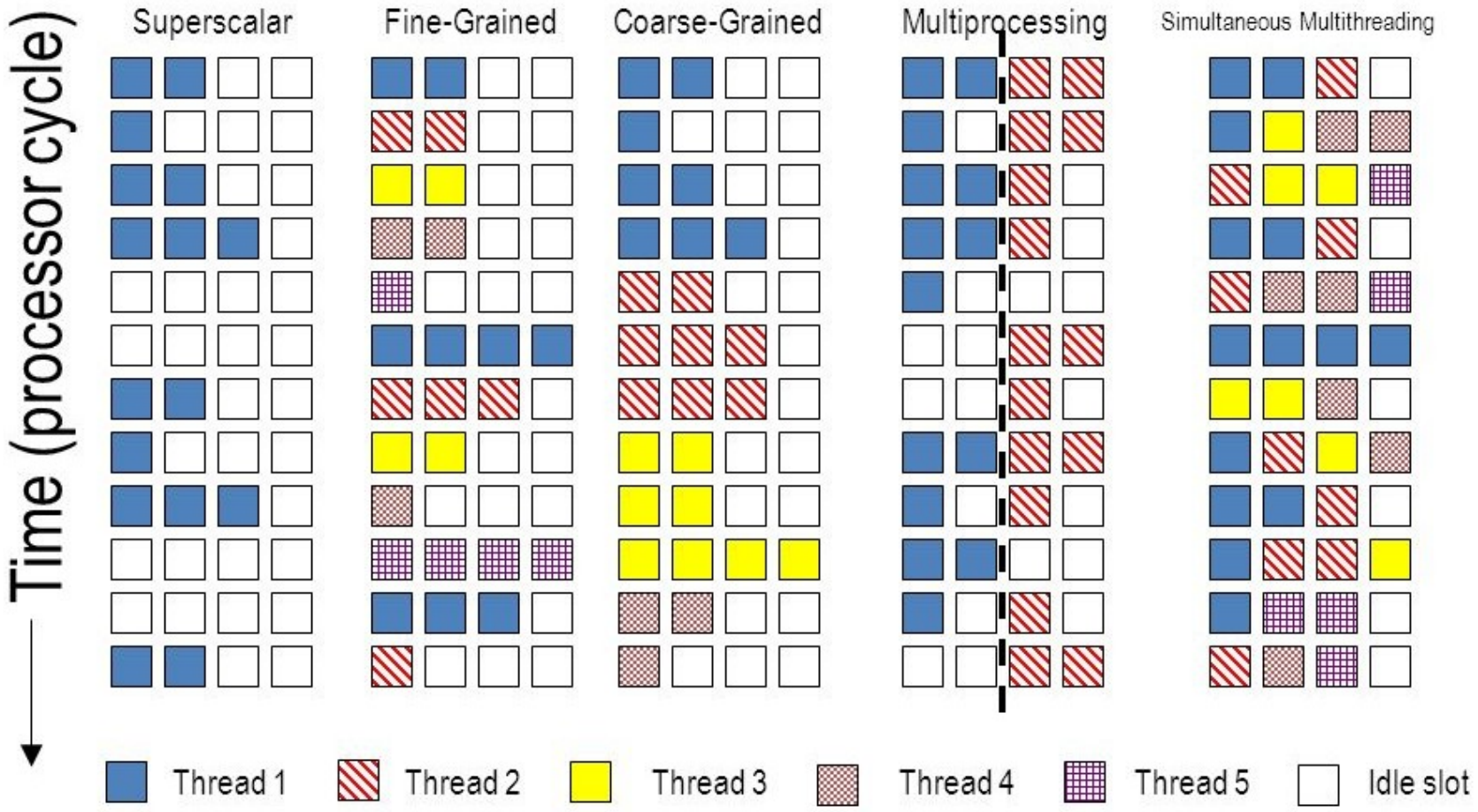
- Processes
  - independently managed by the operating system
    - process state data are separate for different processes
  - having single thread (single instruction sequence) or several threads
  - having own address space separated from spaces of other processes
  - communicating with other processes using system mechanisms
- Threads
  - belong to owning processes
    - private thread state data are subsets of their process state data
    - other state data shared with the other threads (owned by the same process)
  - have no own address space, its address space is within the owning process address space (all the threads share the same code segment)
    - have private stack and some other parts of the address space
    - most of owning process address space shared with the other threads
      - communication with the other threads using shared memory

Multithreading

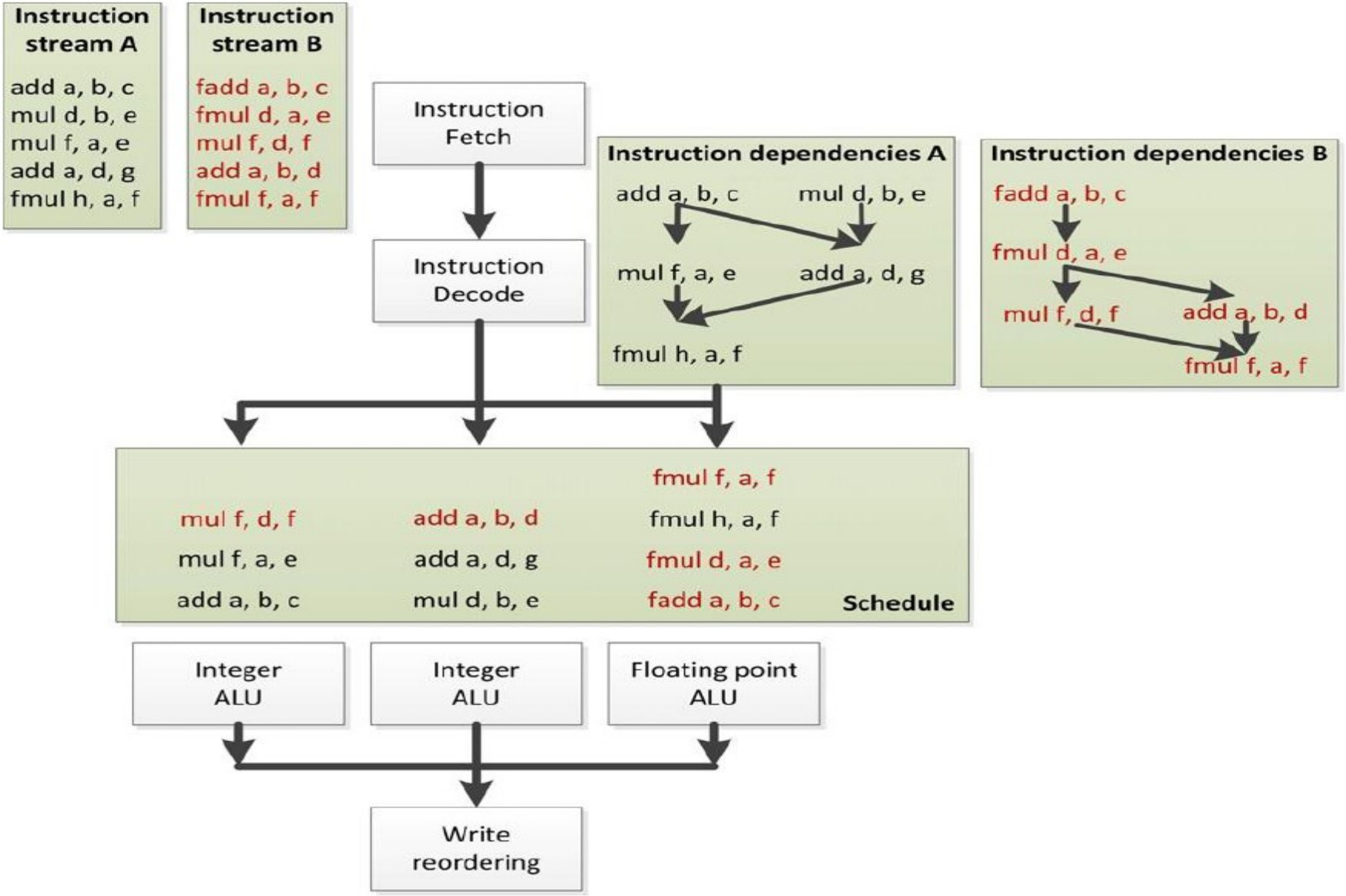
- Multithreading
  - threads can be viewed from different perspectives
    - hardware threads – threads managed by a single core
    - system threads – threads managed by an operating system
    - software threads – threads managed by programmers or virtual machines (interpreters) – mapped to system and hardware threads
- Threads share processor (hardware) resources
  - register file – rewritten during context switch; processors (cores) can have several register sets for simultaneous multithreading
  - execution units – for simultaneous multithreading
  - caches and memory – private thread data may be in the same cache line as private data of another thread
  - files, network devices, etc.
- Context switch between threads is faster than context switch between processes (e.g. there is no TLB flush)

Simultaneous multithreading

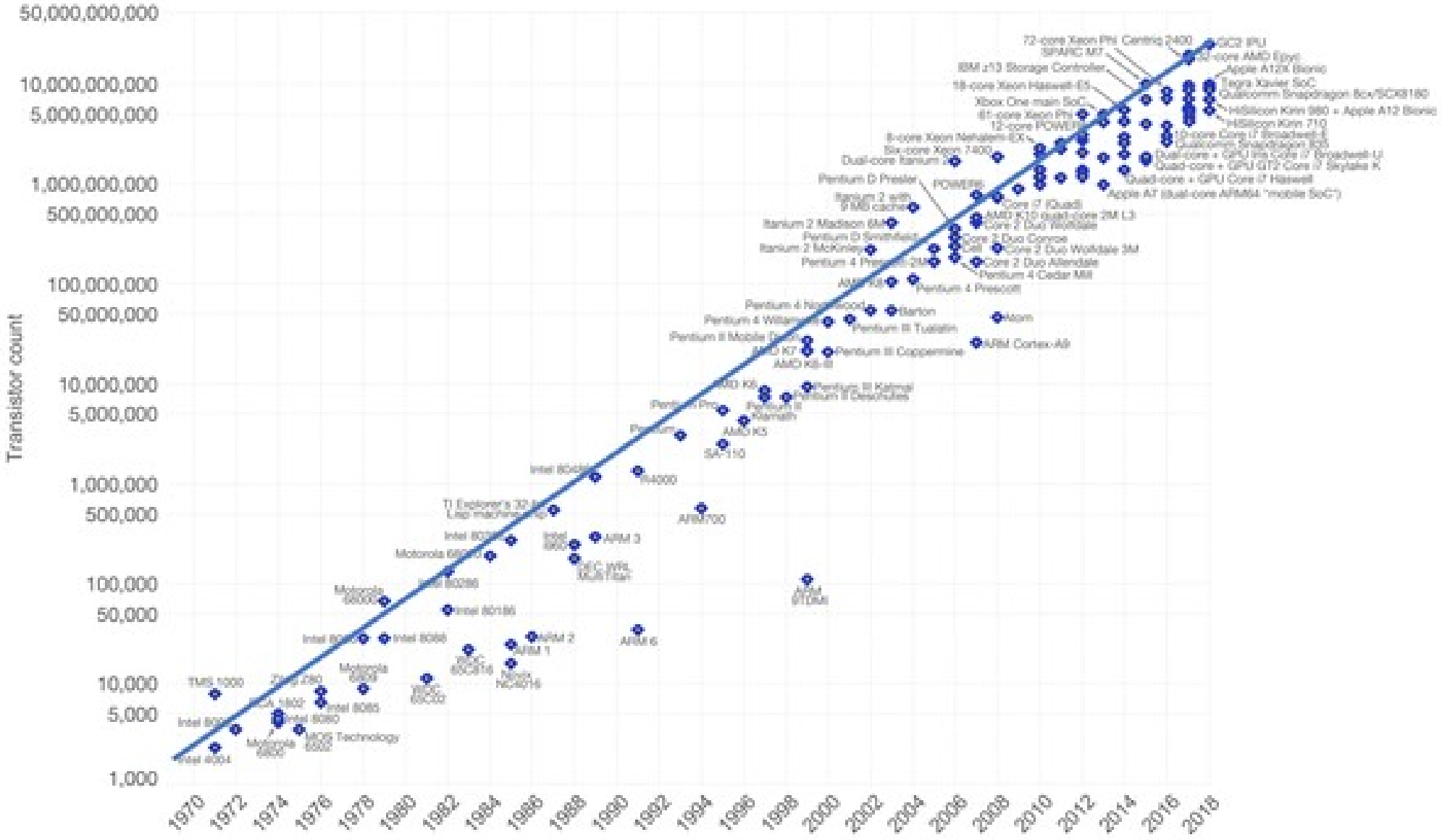
- Hardware management of multithreading (by a single core)



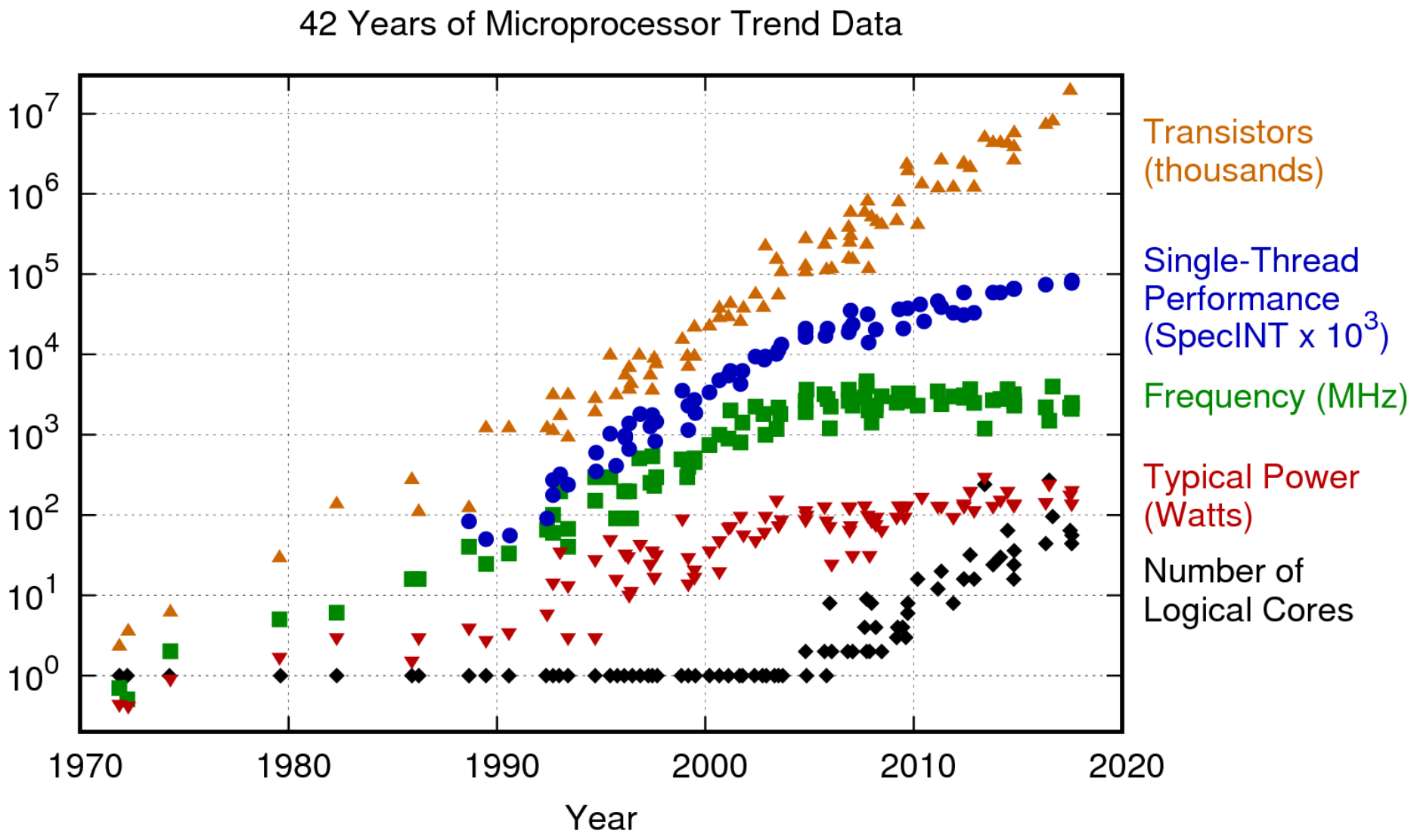
Simultaneous multithreading, SMT



Moore's law (1965)



Microprocessor trends



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

Multi-core microprocessors

- The increasing CPU heat dissipation, due to the increased frequency of operation, forced microprocessor manufacturers to introduce multi-core designs
- The first general purpose multi-core microprocessor was IBM Power 4 in 2001
- Multi-core designs have to deal with the problem of safe and efficient memory access by different cores

Figure 1: POWER4 Chip Logical View

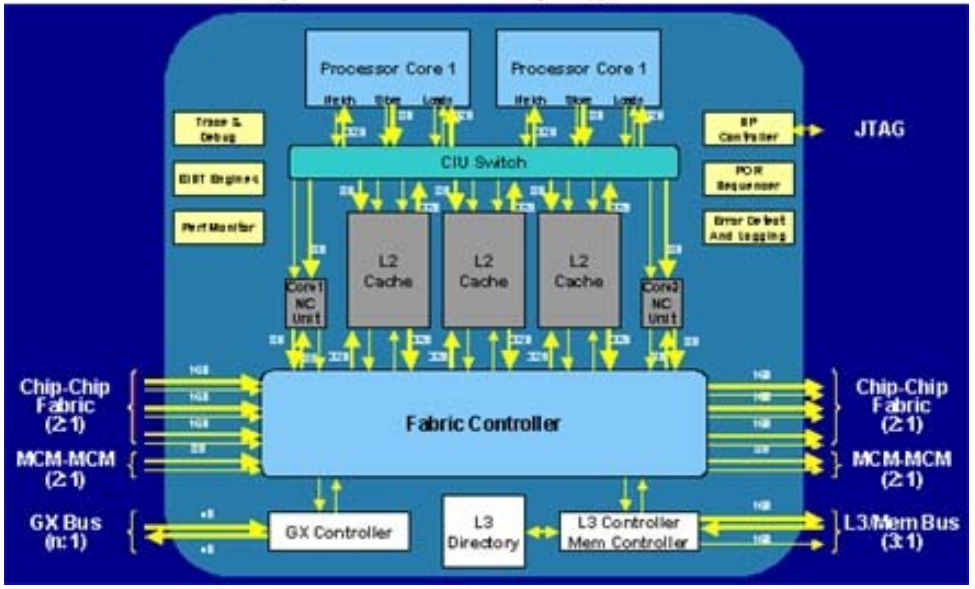
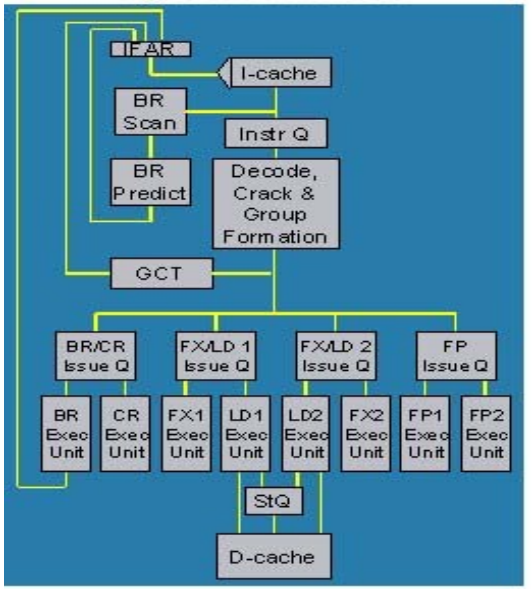
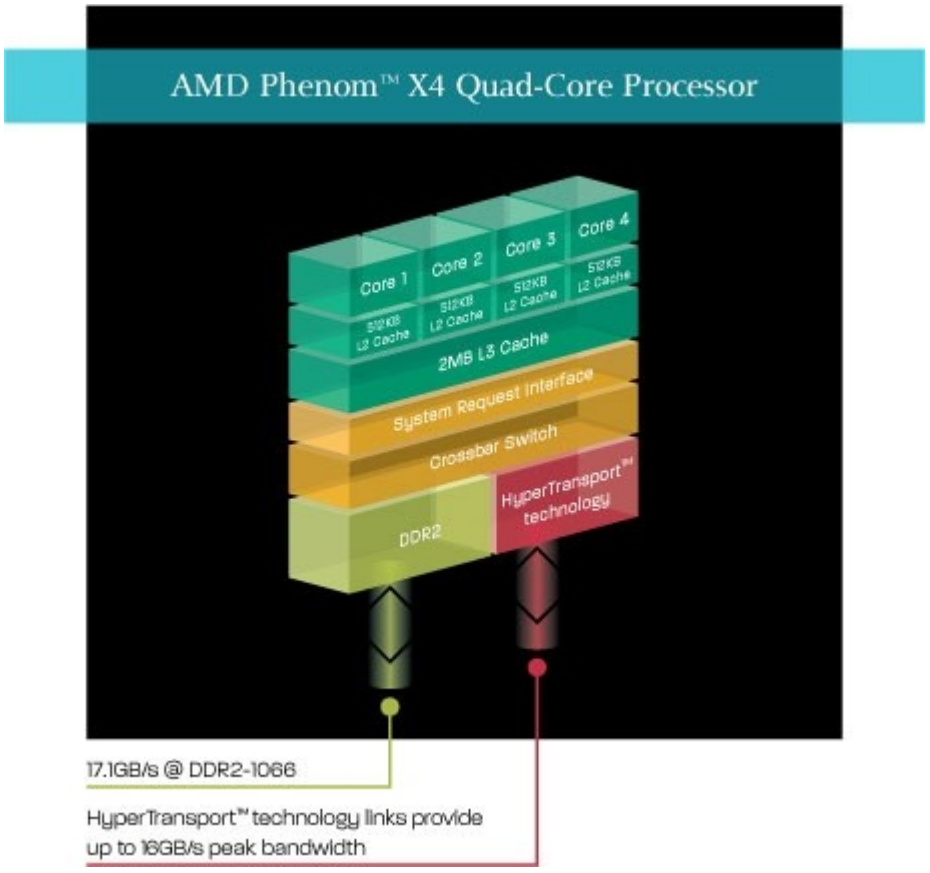
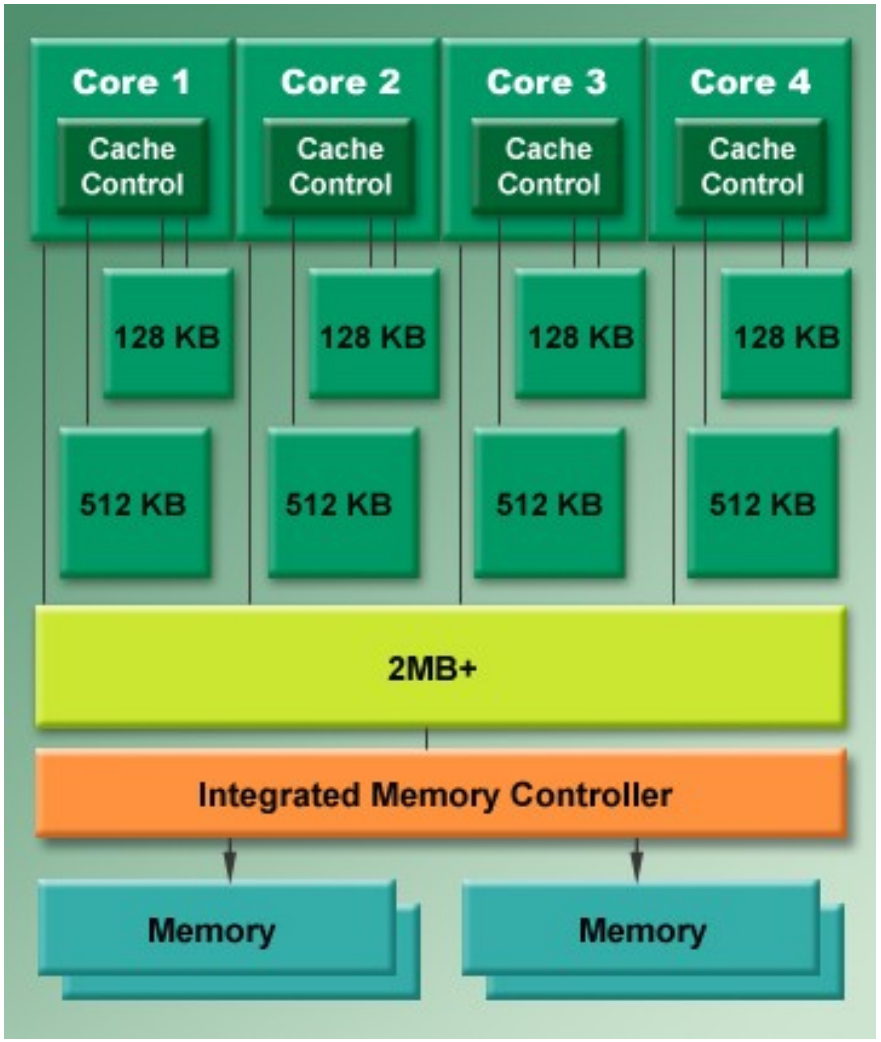


Figure 2: POWER4 Core

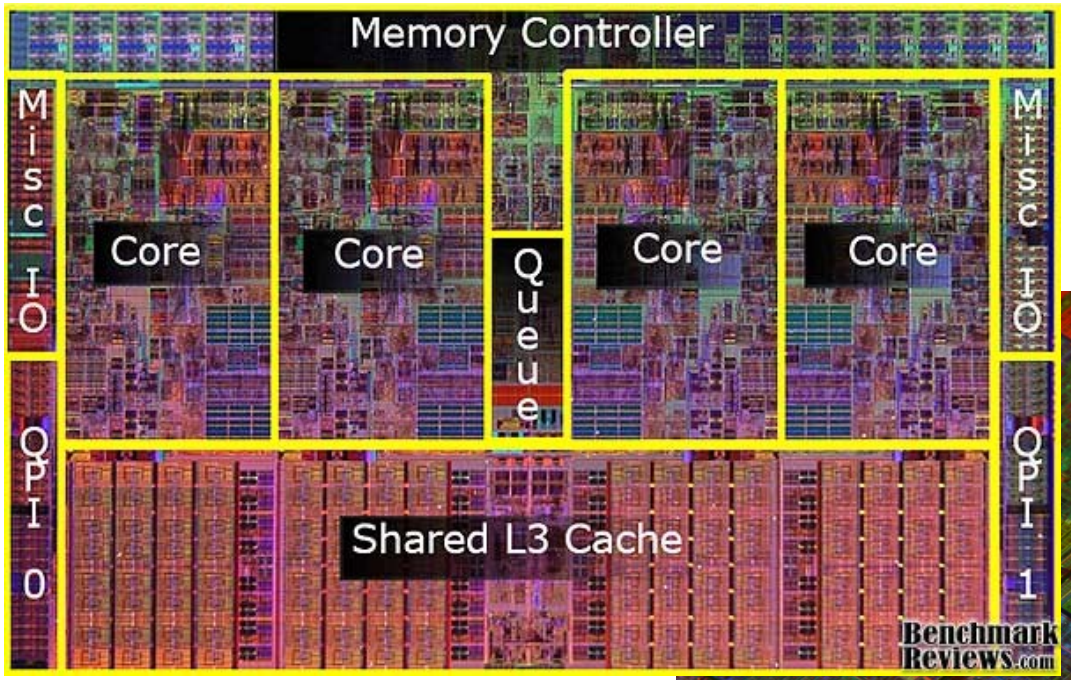


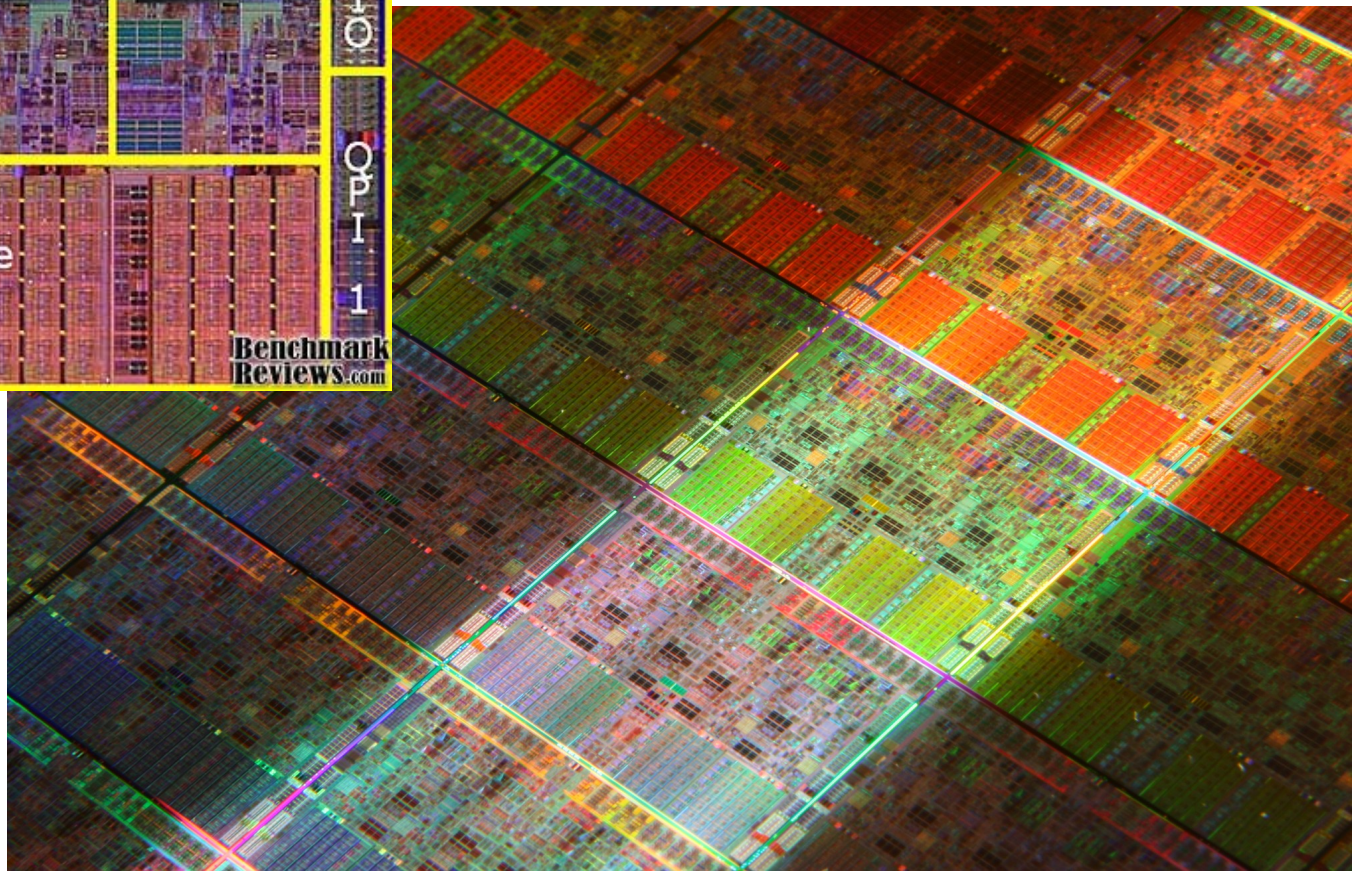
Typical multi-core processor designs





Typical multi-core processor designs

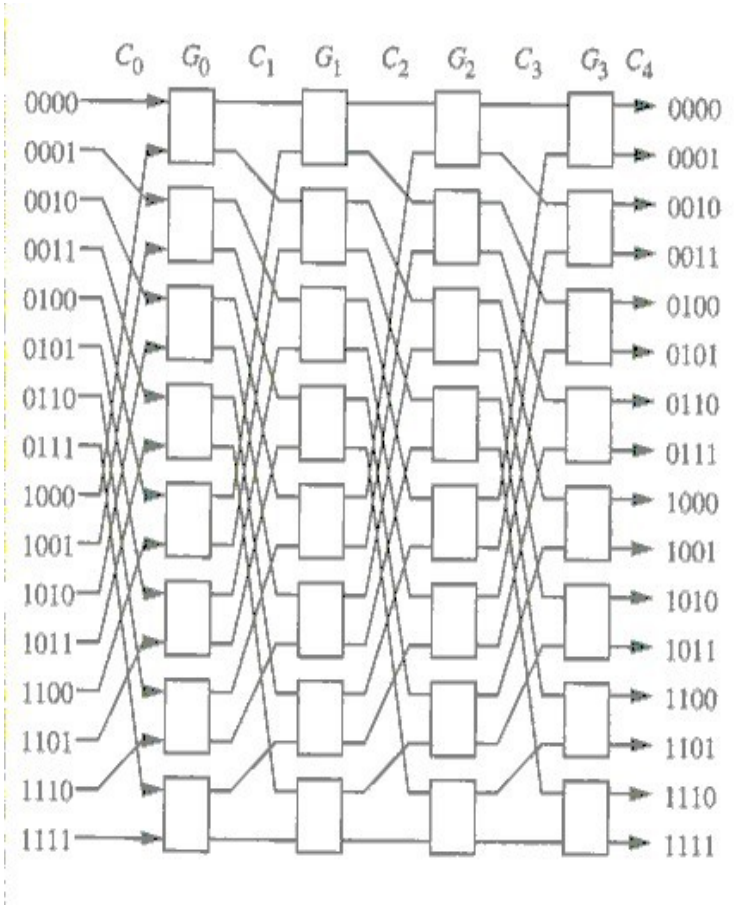




On-chip interconnections

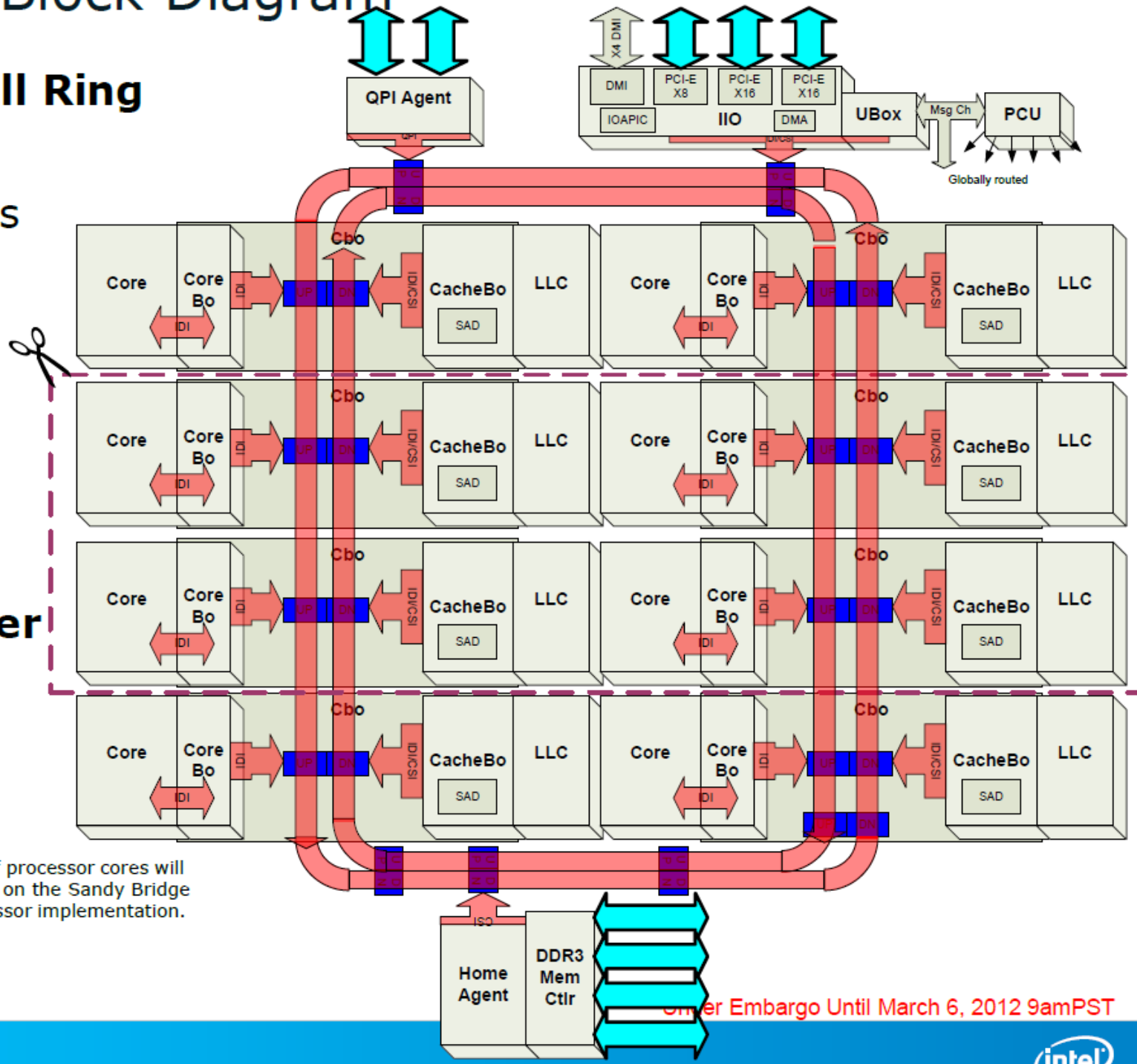
- On-chip interconnections
  - bus, segmented bus with bridges, ring, bidirectional ring
  - crossbar switch, multistage switching network
  - network-on-chip: mesh, torus, tree, hypercube, etc.

- Ω multistage switching network
  - p inputs, p outputs, $\log_2 p$ stages
  - each stage contains p/2 switches 2x2
  - stages connected in a perfect shuffle pattern (output=2*input with round-robin)
  - with bit encoded input and output positions perfect shuffle corresponds to bit rotation, while switches allow for the last bit change



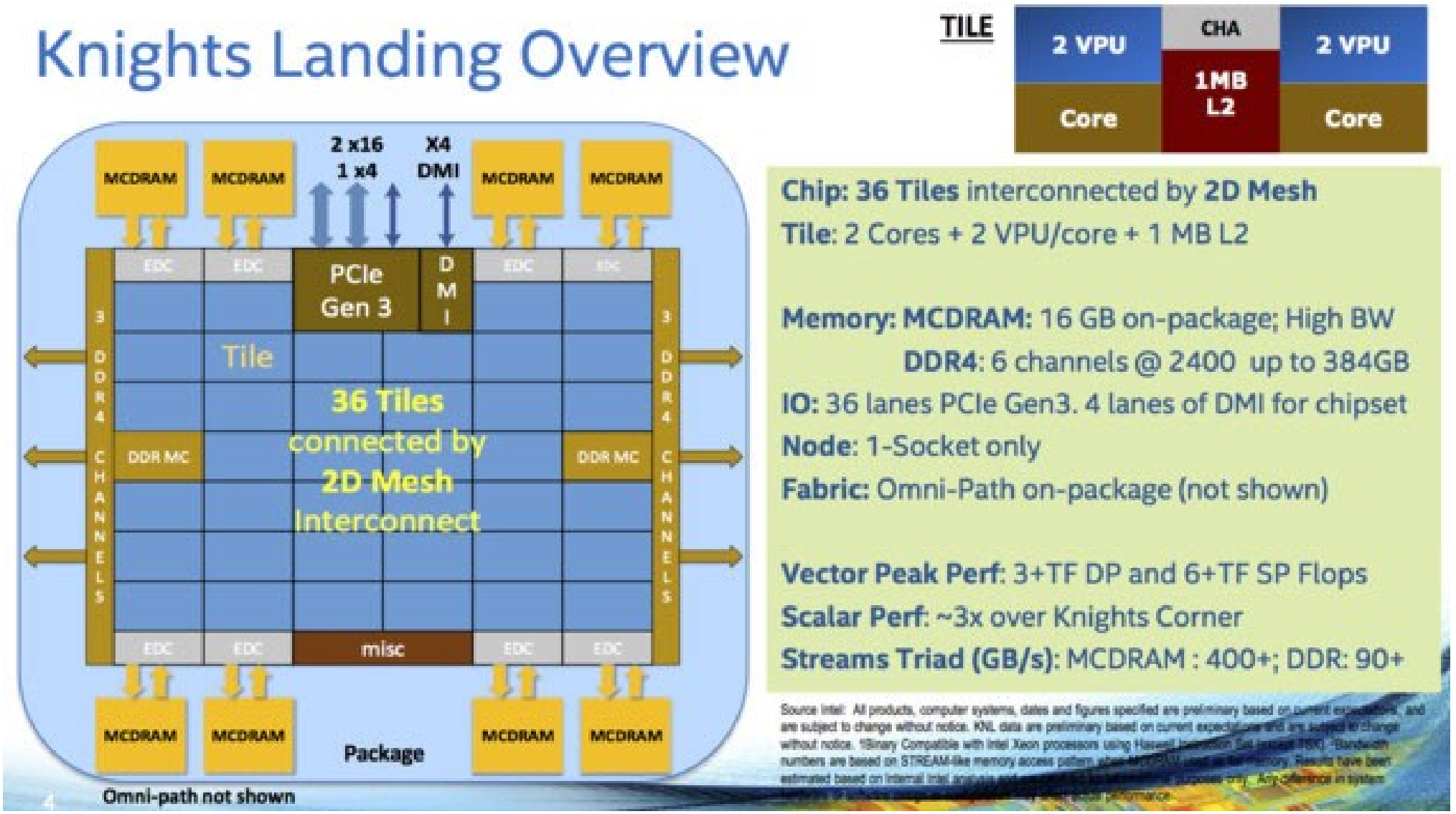
Xeon E5-2600 Block Diagram

- **Bi-Directional Full Ring**
  - 32B/clk/agent
  - 8 Core/LLC slices
- **Last Level Cache**
  - 32B/clk/slice
- **Dual QPI Agent**
- **Integrated I/O**
- **Home Agent**
- **Integerated Memory Controller**
  - Connected to HA
- **PCU**
- **"Ubox"**



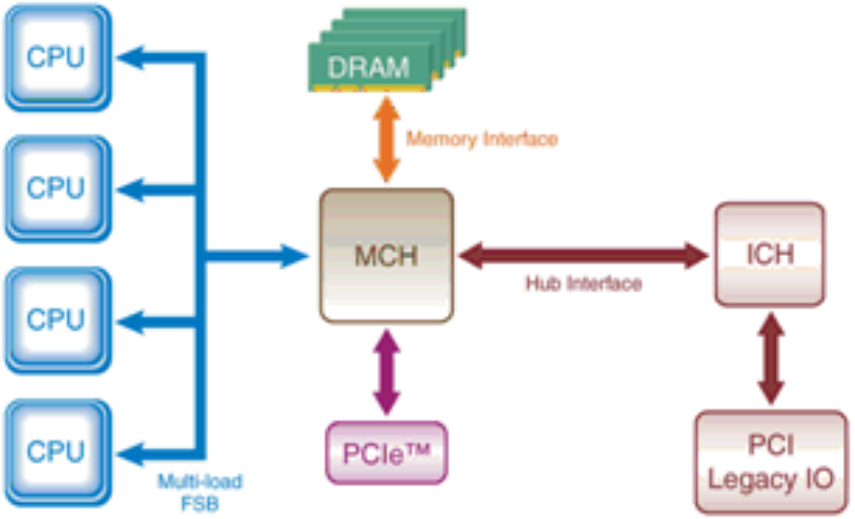
Block Diagram Illustrative only. Number of processor cores will vary with different processor models based on the Sandy Bridge Microarchitecture. Represents server processor implementation.

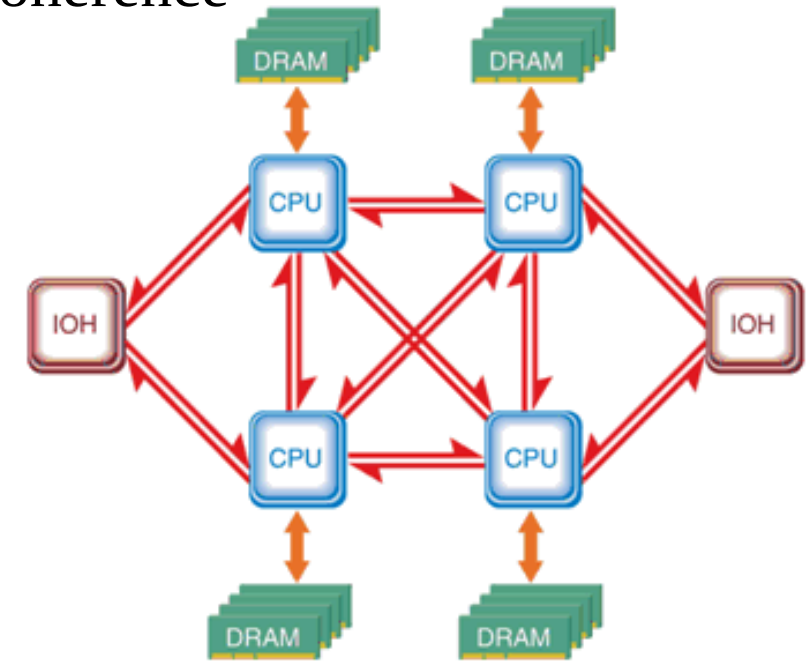
Many-core microprocessors



SMP, UMA, NUMA, etc.

- Multiprocessor systems with shared memory:
  - UMA – uniform memory access ->
  - NUMA – non-uniform memory access
    - ccNUMA – NUMA with cache coherence





Memory layout

```
CPU:   Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz          Intel Core Haswell processor
*************************************************************************************
Hardware Thread Topology:
Sockets:            1       Cores per socket: 4       Threads per core:       2
-------------------------------------------------------------------------------
HWThread  Thread      Core  Socket
0         0           0           0
1         0           1           0
2         0           2           0
3         0           3           0
4         1           0           0
5         1           1           0
6         1           2           0
7         1           3           0
Socket 0:           ( 0 4 1 5 2 6 3 7 )
*************************************************************************************
Cache Topology:
Level:              1       Size:               32 kB
Cache groups:               ( 0 4 ) ( 1 5 ) ( 2 6 ) ( 3 7 )
-------------------------------------------------------------------------------
Level:              2       Size:               256 kB
Cache groups:               ( 0 4 ) ( 1 5 ) ( 2 6 ) ( 3 7 )
-------------------------------------------------------------------------------
Level:              3       Size:               8 MB
Cache groups:               ( 0 4 1 5 2 6 3 7 )
*************************************************************************************
NUMA Topology:
Domain:                     0
Processors:         ( 0 4 1 5 2 6 3 7 )
```

- Example output from likwid-topology tool

Thread CPU affinity

- Standard operating system scheduling assigns threads to different cores (or logical processors in case of SMT cores) based on a complex algorithm that tries to balance the load of cores while retaining the fast context switches for each core
  - Linux in standard form (not real time) uses the value of parameter *nice* to specify scheduling priorities of different threads/processes
- Sometimes standard system scheduling can lead to not optimal performance, when some temporary situation causes the system to assign in a not optimal way
  - process and thread affinity often decides on how memory is used
    - in Linux, assignment of physical frames to virtual pages for dynamically allocated memory is often done on the basis of first write to a memory cell (*lazy page allocation)*
  - when process or thread resumes execution after preemption from one core (i.e. context switch) it can be scheduled to some other core that forces cache reloading

Thread CPU affinity

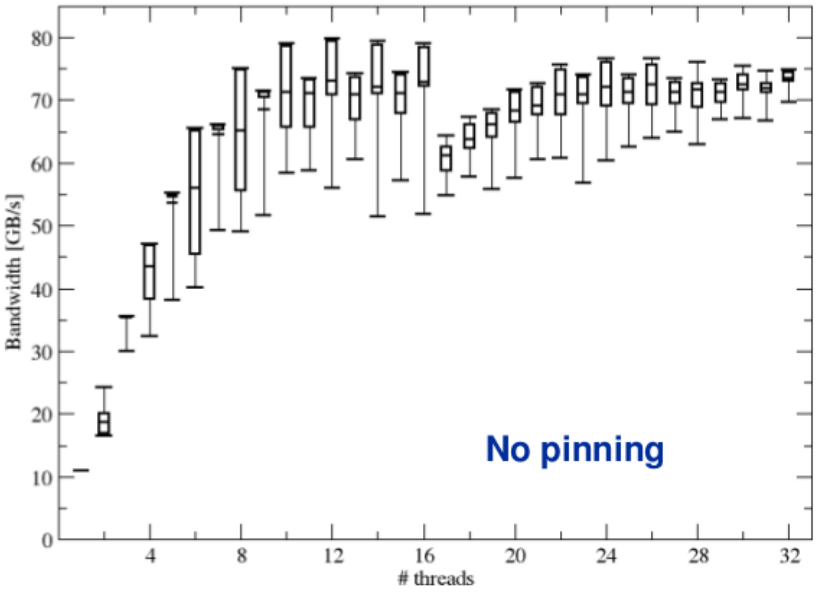
- There are several utilities for manual process and thread pinning (assigning to a given set of cores or logical processors)
  - system commands (e.g. Linux *taskset*) and functions
    - e.g. Linux system function *sched_setaffinity* – using specific form for specifying the set of CPUs to assign the process to
  - external utilities
    - popular Linux utility *numactl*, more complex tool *likwid*
  - environment variables for compilers – specifying also the scheduling policies
    - icc – KMP_AFFINITY
      - *KMP_AFFINITY="granularity=fine,proclist=[<proc_list>],explicit"*
    - gcc – GOMP_AFFINITY
      - *GOMP_CPU_AFFINITY=<proc_list> , np. [0,1,2,3,4,....]*
  - function calls for parallel programming environments
    - *pthread_setaffinity_np*, *pthread_attr_setaffinity_np* – syntax similar to *sched_setaffinity* for POSIX threads

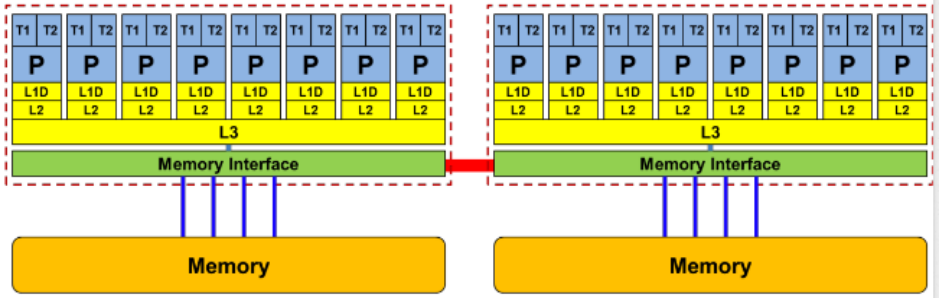
Thread CPU affinity

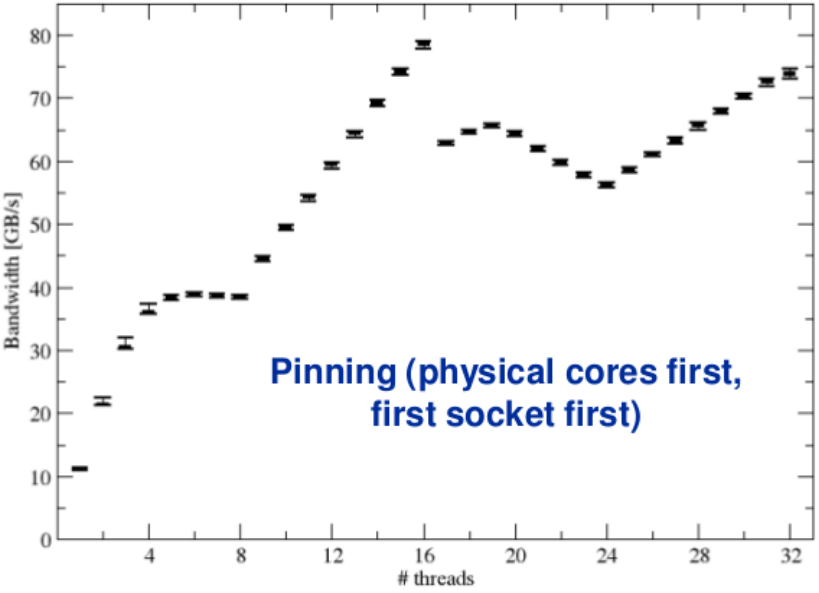
- OpenMP 4.0 introduces a portable form of controlling thread affinity to sockets/cores/logical processors (*places* for OpenMP) with the environment variable OMP_PROC_BIND
  - there are three affinity policies with detailed assignment rules, that can be described in a simplified way as:
    - master – all the threads executing on the same place as the master thread
    - close – the threads are assigned to subsequent places, starting with the master thread place (with wrap around for large threads numbers)
      - e.g. for 2 threads and *num_proc* places, with master thread on the place 0 – assignment to places 0 and 1
    - spread – attempting to spread evenly threads across all the places (with wrap around for too large number of threads)
      - e.g. in the situation as above - assignment to 0 and *num_proc*/2
  - there are also several functions to get information on available sockets, cores and logical processors (hardware threads)

The influence of thread affinity

- An example of memory throughput benchmark STREAM







There are several reasons for caring about affinity:

- **Eliminating performance variation**
- **Making use of architectural features**
- **Avoiding resource contention**