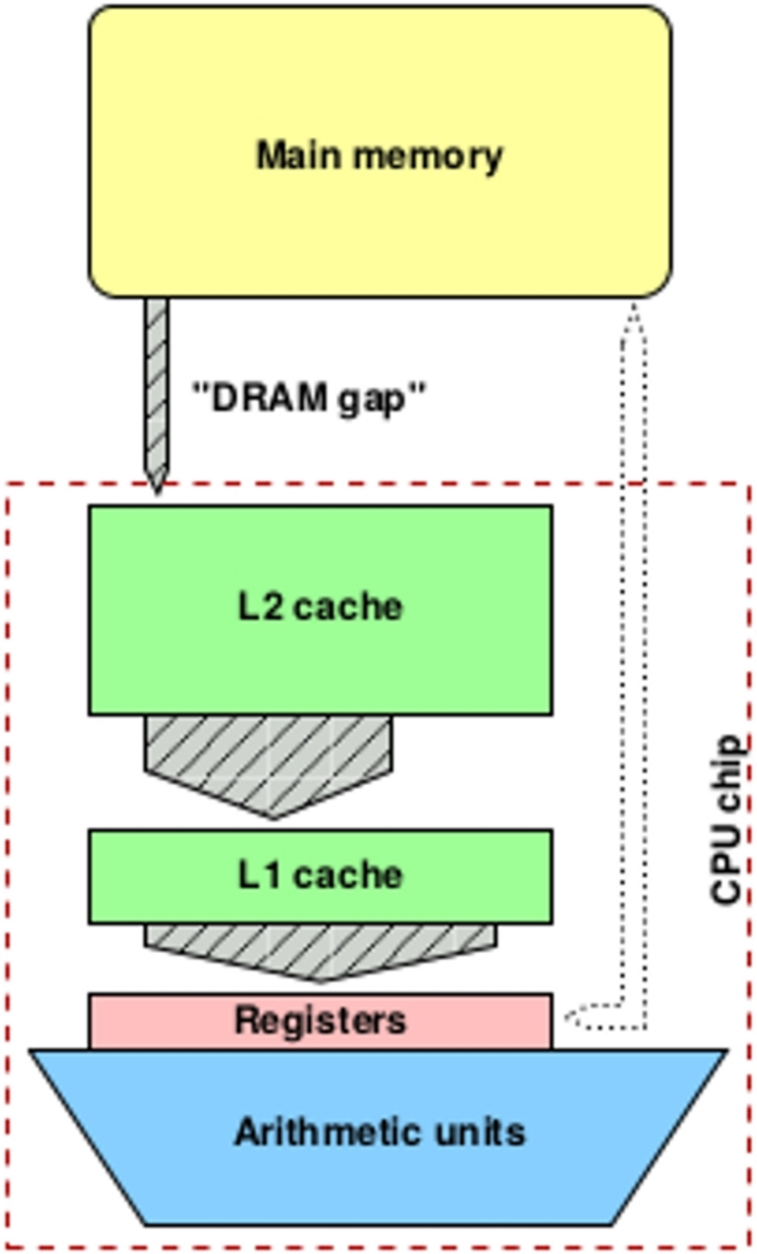
Primena super-računara u astronomiji

Memory Hierarchies



Memory Hierarchies

- data transfer speeds to main memory are very slow compared to the CPU's arithmetic performance
- to transfer a single data item from memory, an initial waiting time, called latency, occurs until bytes can actually flow
- latency is typically defined as the time it takes to transfer a zero-byte message
- Memory latency is usually of the order of several hundred CPU cycles
- caches can alleviate the effects of the DRAM gap in many cases

Cache Memory

- Caches are low-capacity, high-speed memories
- Main memory (RAM) is much slower but also much larger than cache
- Generally, the “closer” a cache is to the CPU’s registers, its bandwidth is higher while the latency is lower,
- But, the smaller it must be to keep administration overhead low
- Cache miss and cache hit
- Caches can only have a positive effect on performance if the data access pattern of an application shows some locality of reference, that is: data items that have been loaded into cache are to be used again “soon enough”

Cache Memory

- Caches can only have a positive effect on performance if the data access pattern of an application shows some locality of reference, that is: data items that have been loaded into cache are to be used again “soon enough”

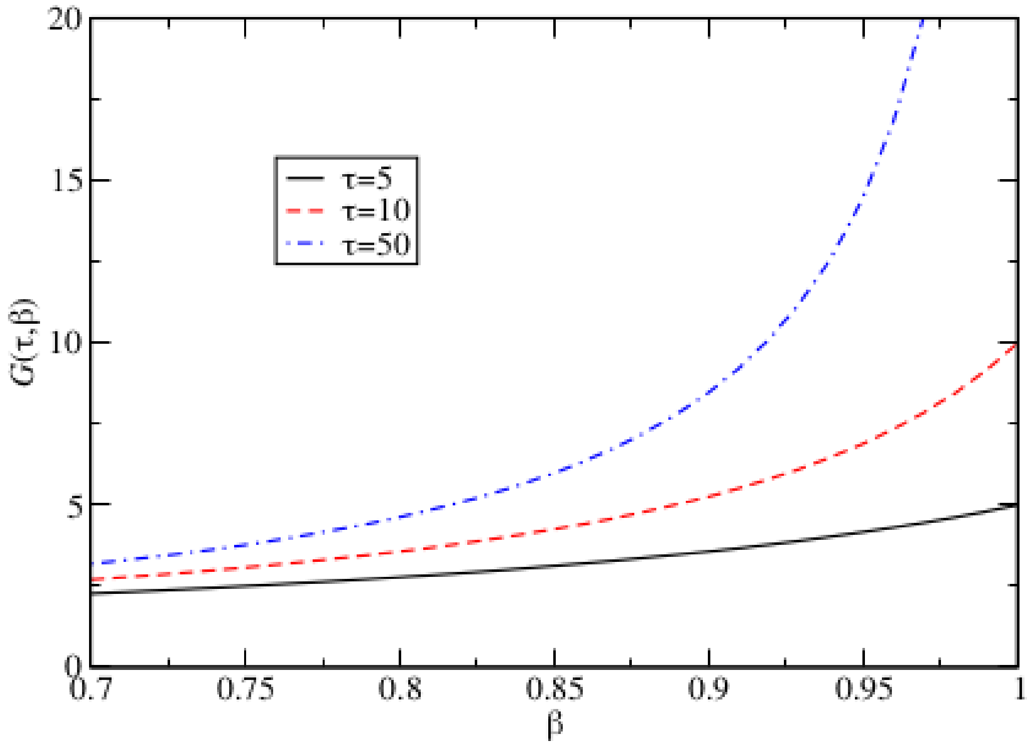


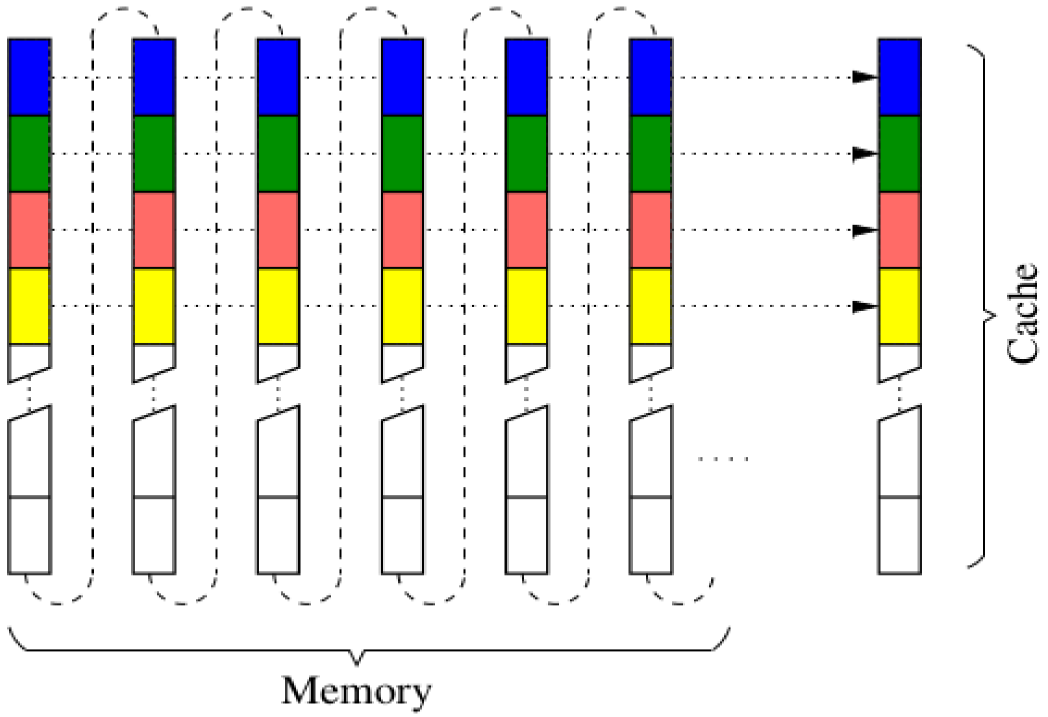
Figure:
Performance gain
vs
cache reuse ratio

Cache Memory Mapping

- no restriction on which cache line can be associated with which memory locations - fully associative cache
- **limitations**:
  - for each cache line the cache logic must store its location in the CPU’s address space, and each memory access must be checked against the list of all those addresses
  - the decision which cache line to replace next if the cache is full is made by some algorithm implemented in hardware

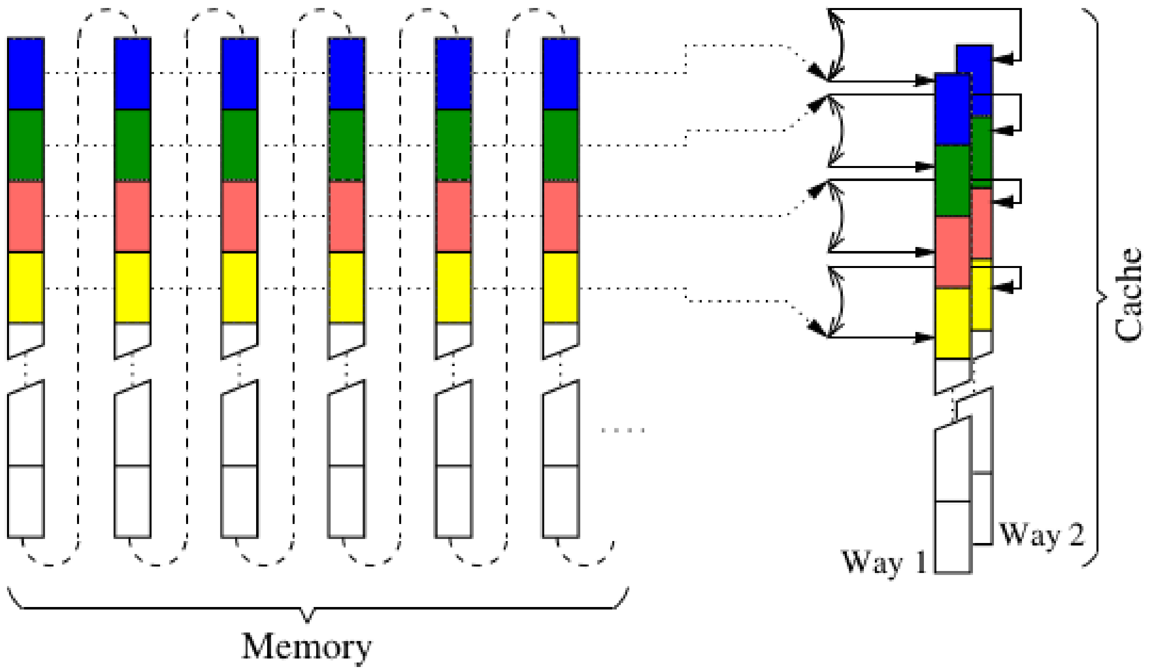
Cache Memory Mapping

- cache which maps the full cache size repeatedly into memory - direct-mapped cache
- memory locations that lie a multiple of the cache size apart are always mapped to the same cache line
- the cache line that corresponds to some address can be obtained very quickly by masking out the most significant bits
- disposed toward cache thrashing: cache lines are loaded into and evicted from cache in rapid succession



Cache Memory Mapping

- set-associative cache is divided into m direct-mapped caches equal in size, so-called ways
- the number of ways m is the number of different cache lines a memory address can be mapped to
- for each cache level, the tradeoff between low latency and prevention of thrashing must be considered by processor designers
- innermost (L1) caches tend to be less set-associative than outer cache levels



Prefetching

- The problem of latency on the first miss
- Assuming a typical commodity system with a memory latency of 100 ns and a bandwidth of 4 GBytes/sec, a single 128-byte cache line transfer takes 32 ns, so 75 % of the potential bus bandwidth is unused
- Obviously, latency has an even more severe impact on performance than bandwidth

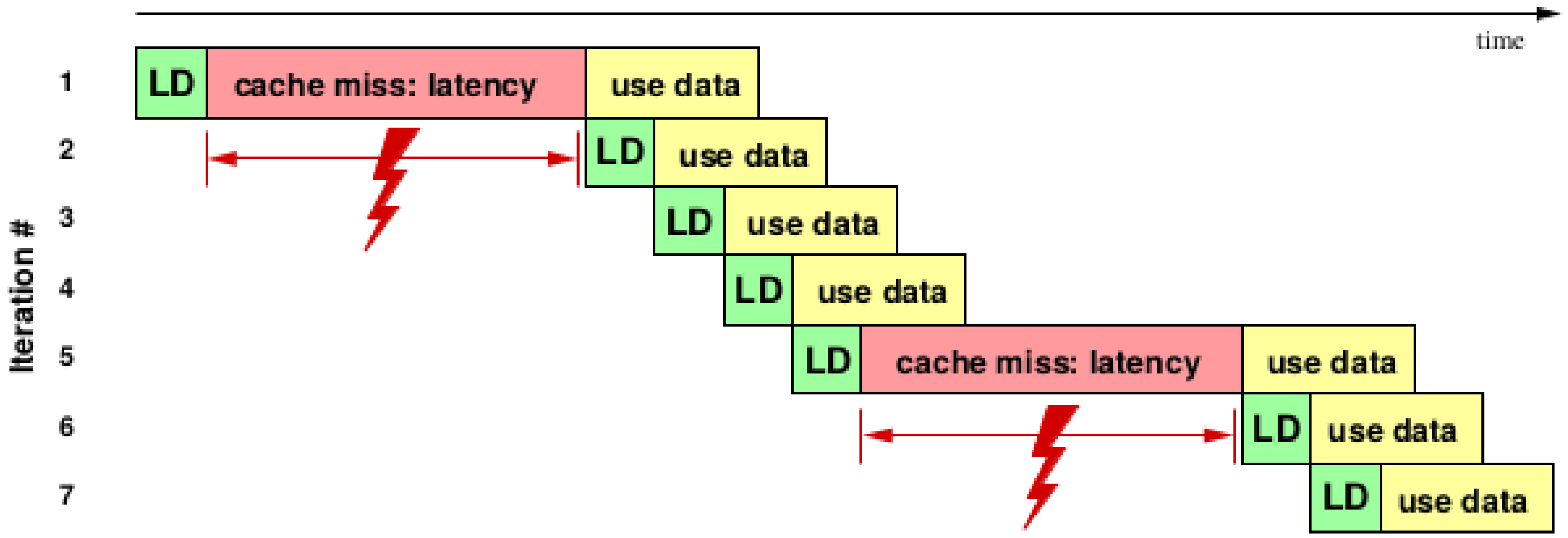
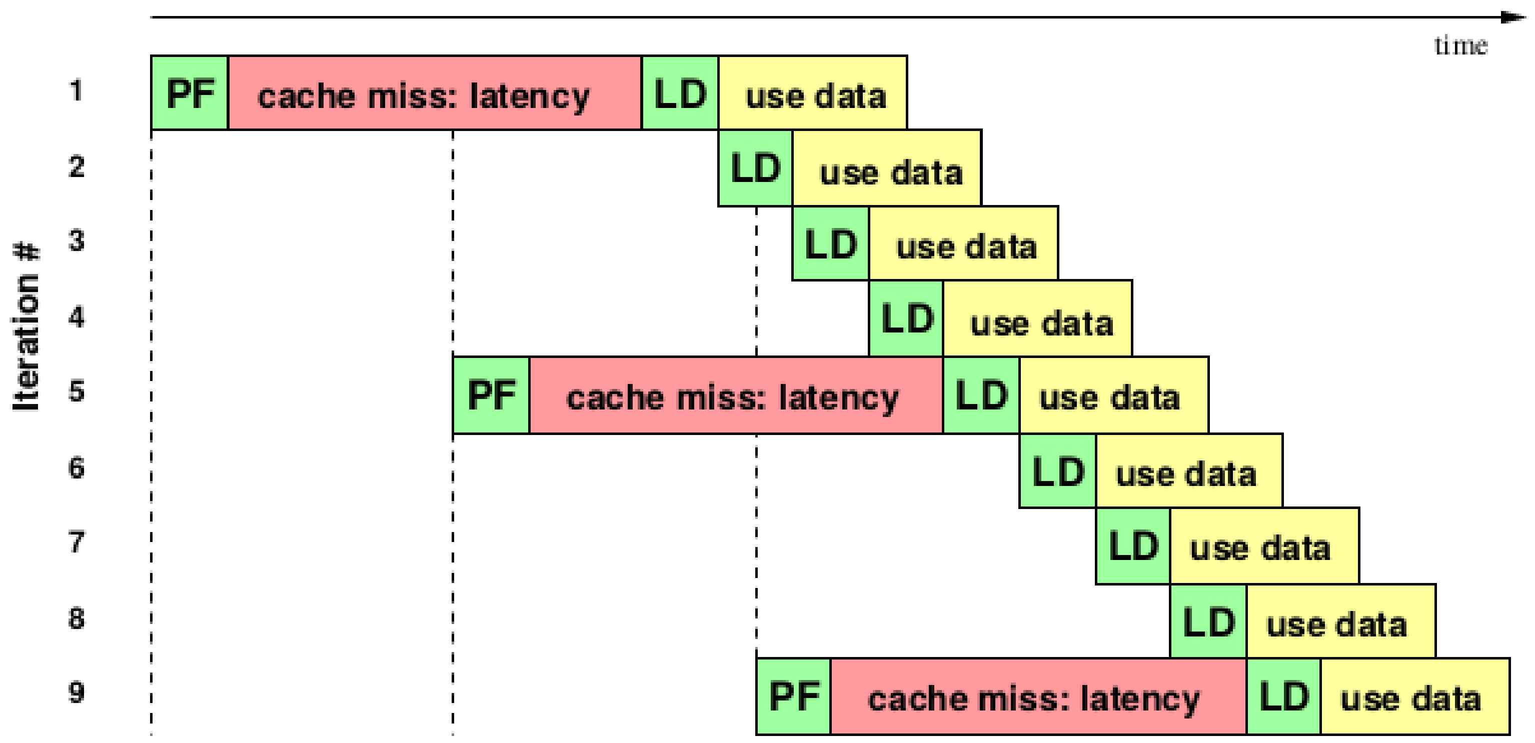


Figure 1.10: Timing diagram on the influence of cache misses and subsequent latency penalties for a vector norm loop. The penalty occurs on each new miss.

Prefetching

- The latency problem can be solved in many cases, however, by prefetching
- Prefetching supplies the cache with data ahead of the actual requirements of an application



Prefetching

- **The number of prefetches required for hiding the latency completely:** If T_l is the latency and B is the bandwidth, the transfer of a whole line of length L_c DP words takes a time of

$$T = T_l + \frac{8L_c}{B}$$

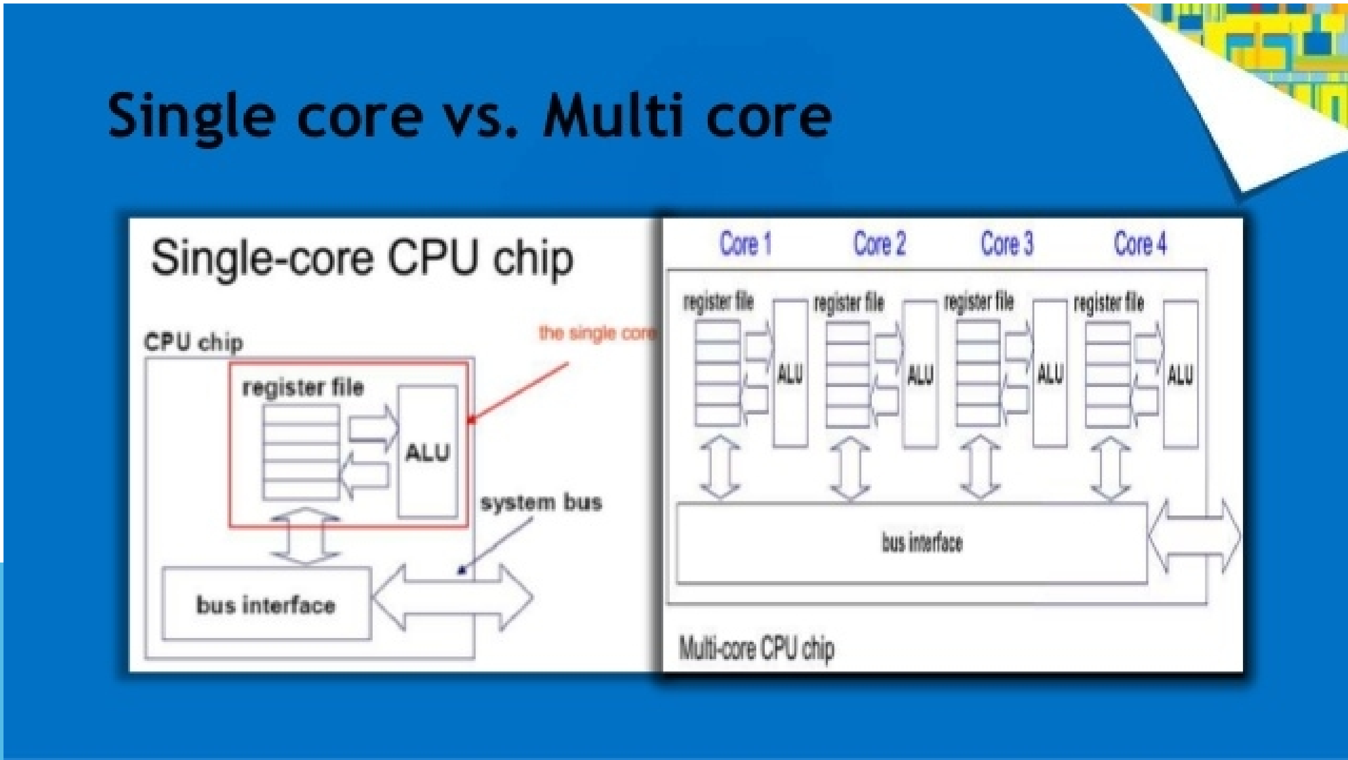
- One prefetch operation must be initiated per cache line transfer, and the number of cache lines that can be transferred during time T is the number of prefetches P that the processor must be able to sustain

$$P = \frac{T}{8L_c/B}$$

- As an example, for a cache line length of 128 bytes (16 DP words), B = 6.4 Gbytes/sec and T_l = 140 ns we get P = 160/20 = 8 prefetches

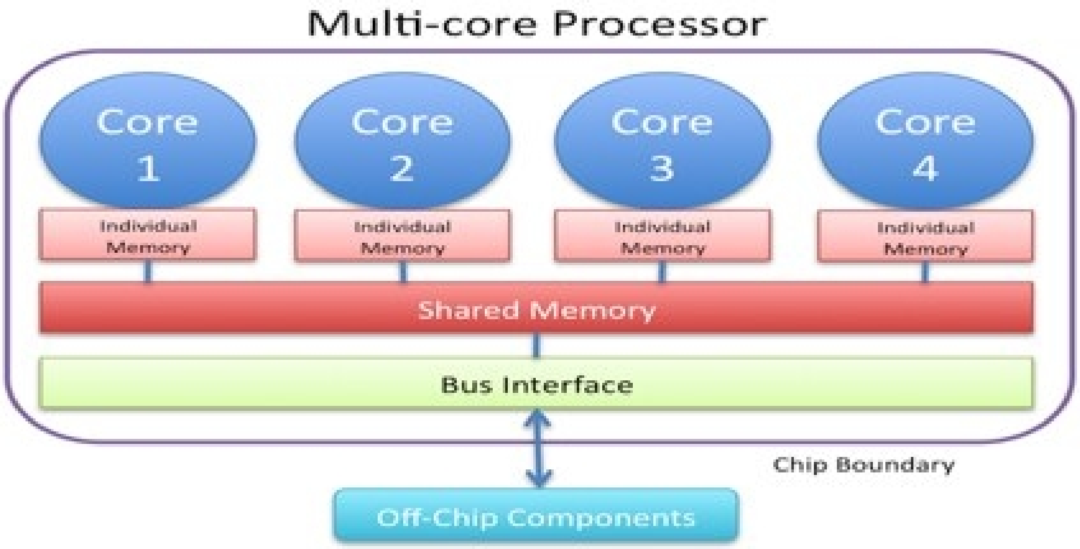
Multi-core processors

- although Moore’s Law is still valid, standard microprocessors are starting to hit the “**heat barrier**”
- architectural advances and growing cache sizes alone will not be sufficient to keep up the one-to-one correspondence of Moore’s Law with application performance
- possible solution is in the form of **multi-core** designs
- Power dissipation of modern CPUs is proportional to the third power of clock frequency f_c

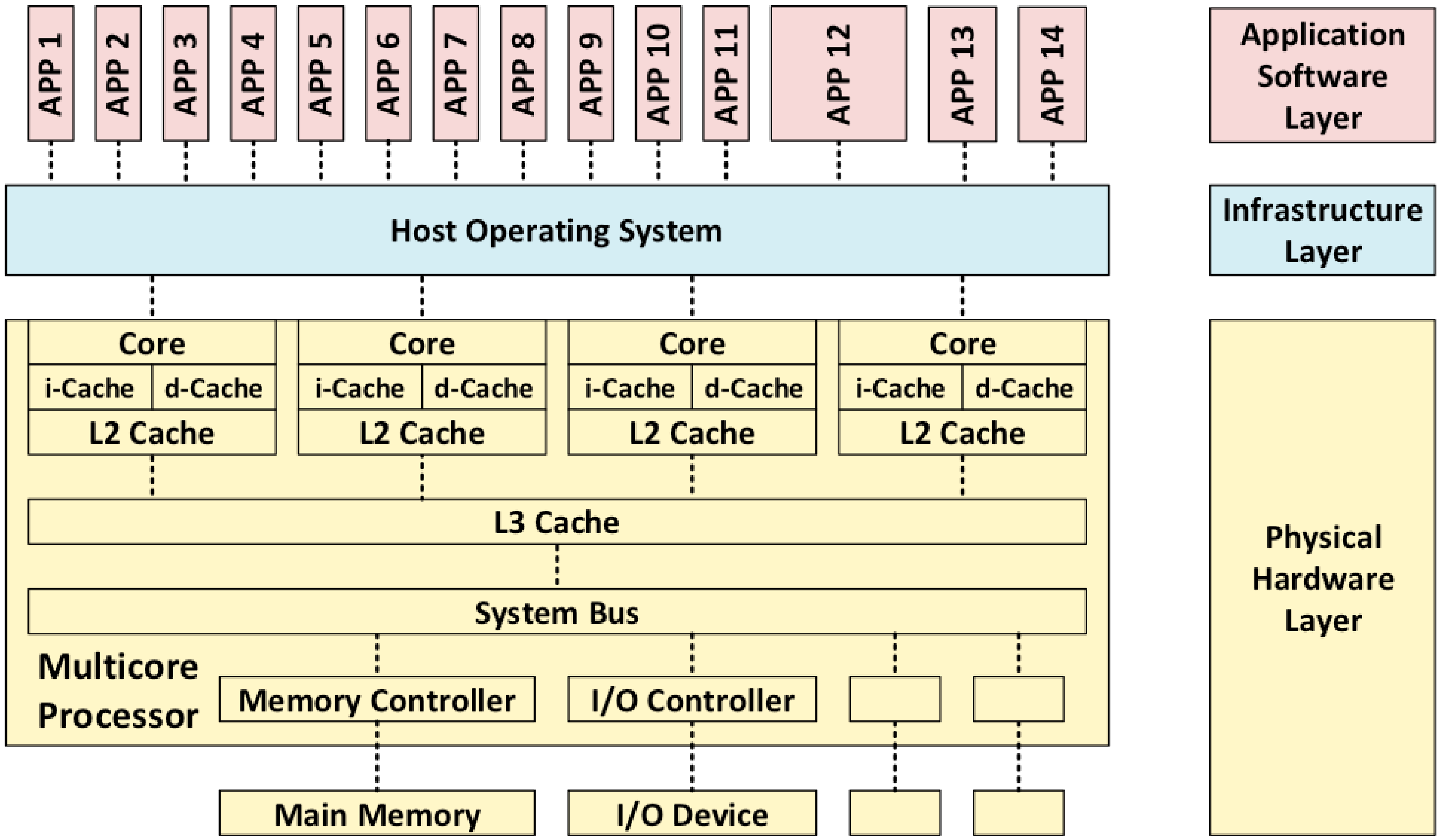


Multi-core processors

- Assuming that a single core with clock frequency f_c has a performance of p and a power dissipation of W, some relative change in performance $\varepsilon_p = \Delta p/p$ will emerge for a relative clock change of $\varepsilon_f = \Delta f_c/f_c$
- $|\varepsilon_f|$ is an upper limit of $|\varepsilon_p|$
- Power dissipation is $W + \Delta W = W (1+\varepsilon_f)^3$

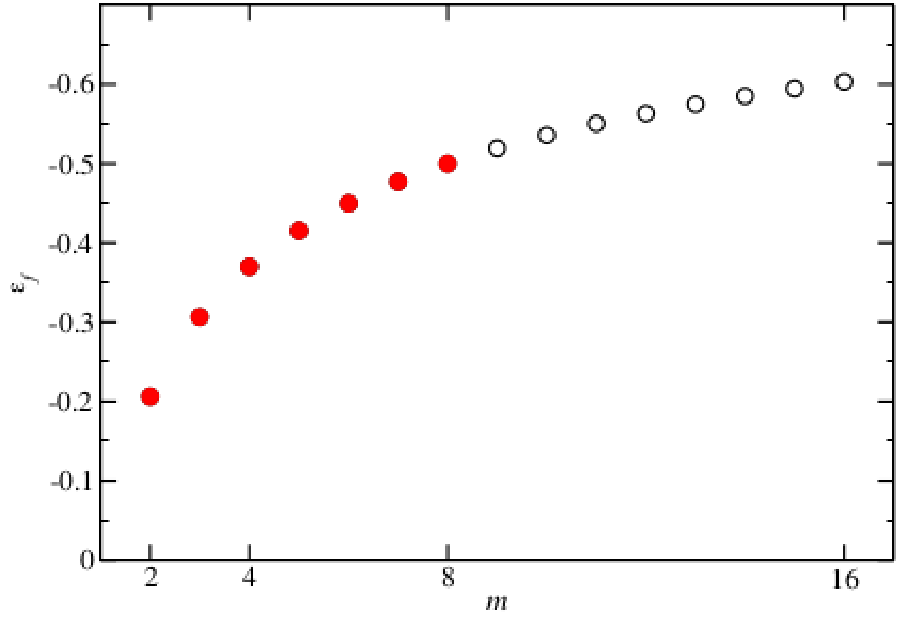


Multi-core processors



Multi-core processors

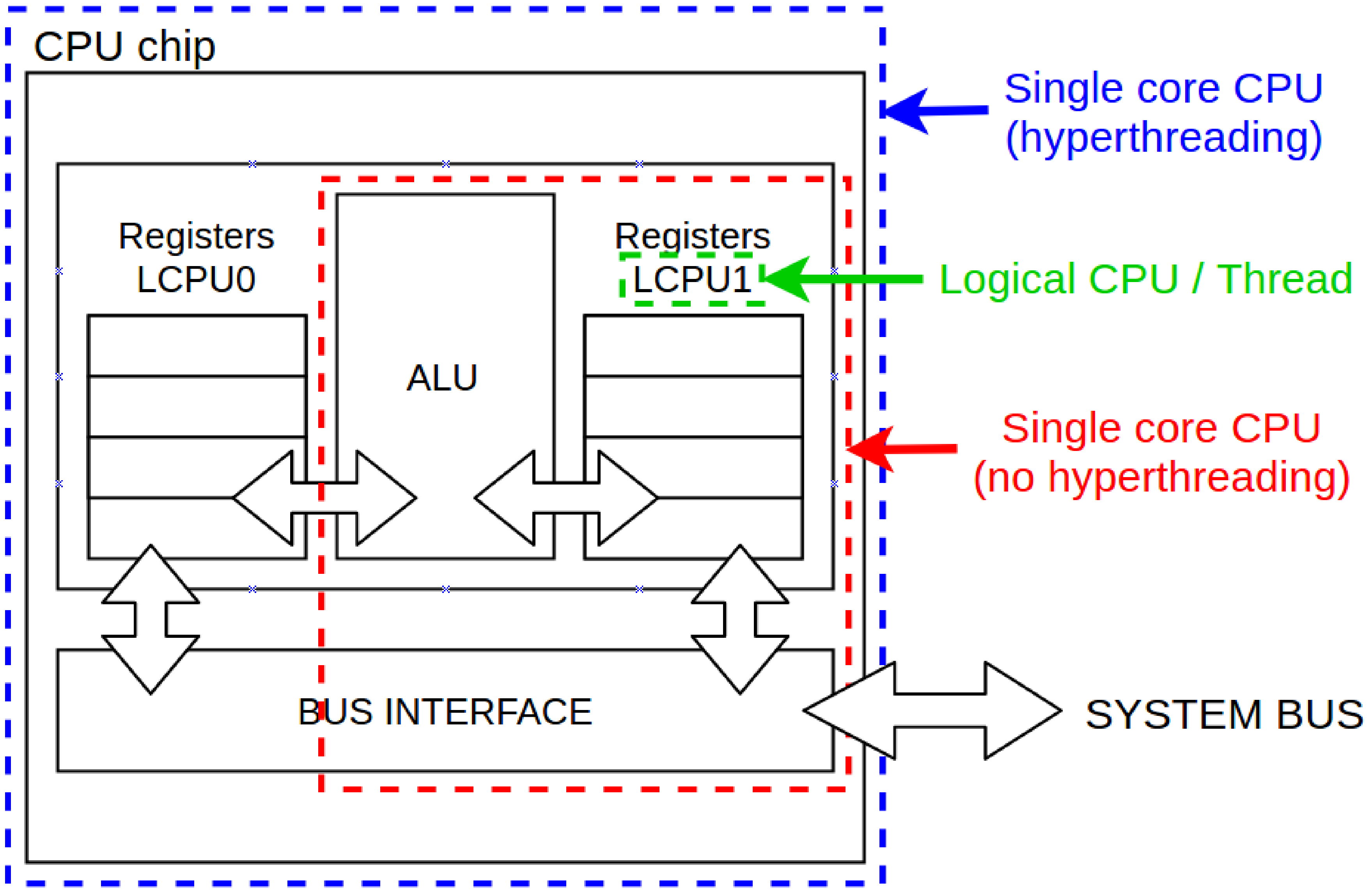
- place more than one CPU core on the same die while keeping the same power envelope as before
- $(1+\varepsilon_f)^3 m = 1 \Rightarrow \varepsilon_f = m^{-1/3} - 1$
- the required relative frequency reduction with respect to the number of cores
- the overall performance of the multi-core chip: $p_m = (1+\varepsilon_p)p_m$
- limit on the performance penalty for a relative clock frequency reduction of ε_f that should be observed for multi-core to stay useful: $\varepsilon_p > 1/m - 1$



CPU vs Core vs Thread

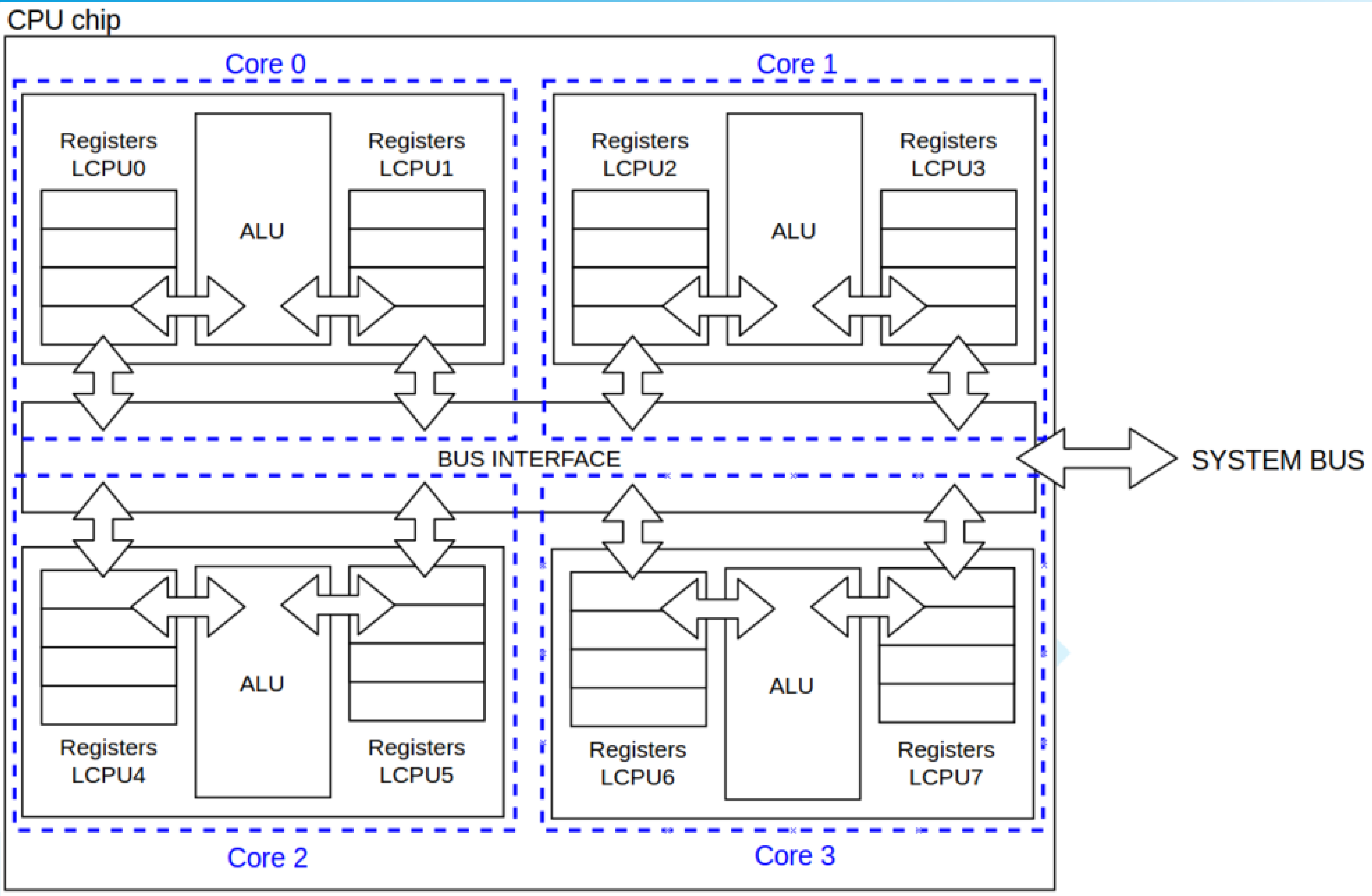
- microprocessor, processor or simply CPU, is a single chip
- communications between the different processors of a multiprocessor system are very inefficient since they have to be done through the **system bus**
- in order to improve this situation the **HyperThreading** technology was invented
- duplicating some CPU internal components within the same chip, such as registers or first level caches
- miniaturizing all processor components and encapsulating them next to others in a single chip
- each of these encapsulated processors is called **core**
- communications between them by means of an **internal bus**

CPU vs Core vs Thread



Single core hyperthreading CPU
(2 logical CPU's / threads)

CPU vs Core vs Thread



Quad-core hyperthreading CPU

Parallel computing

- The parallel computing is when a number of processors (cores) solve a problem in a cooperative way
- All supercomputers’ architectures depend on parallelism

Basic principles of parallelism

- **Parallelization** is the process of formulating a problem in a way that lends itself to concurrent execution by several "**execution units**"
- Ideally, the execution units are initially given some amount of work to do which they execute in exactly the same amount of time
- Using **N** units, a problem that takes a time **T** to be solved sequentially, will now take only **T/N**.
- We call this a speedup of **N**.
- How well a task can be parallelized is usually quantified by some **scalability metric**

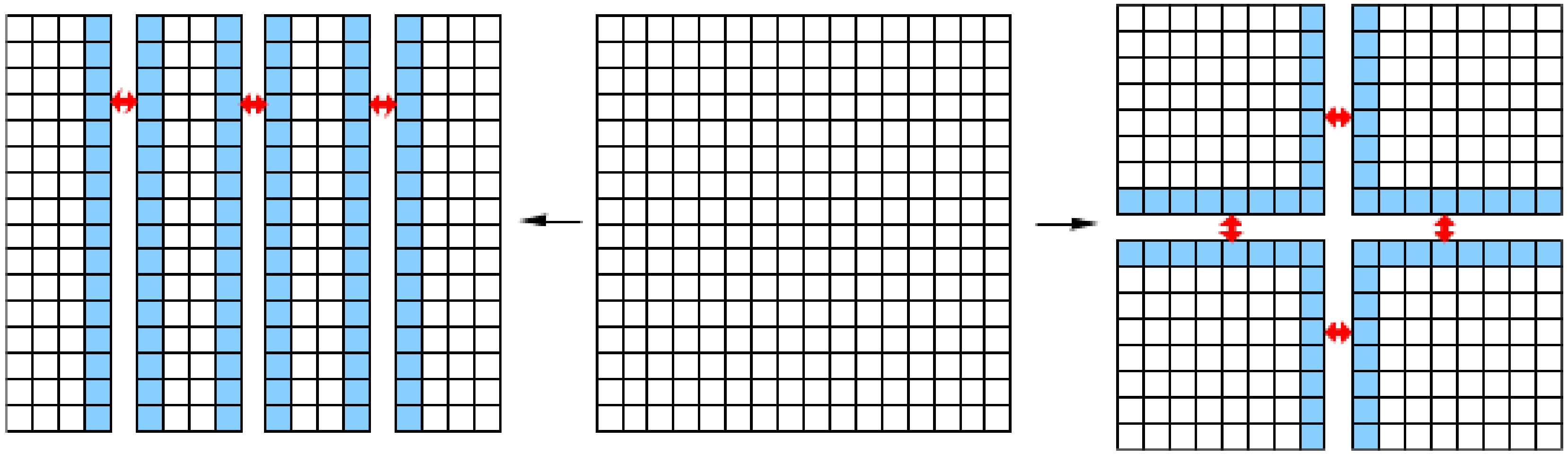
Parallelization strategies

- **Data parallelism:** Many simulations in science could be represented with a simplified picture of reality in which a computational domain is represented as a grid of discrete positions for the physical quantities under consideration. The work is distributed across processors, and a part of the grid is assign to each CPU. This is called **domain decomposition**.

- **Functional (or task) parallelism:** Solving a complete problem can be split into more or less disjoint subtasks. The tasks can be worked on in parallel, using appropriate amounts of resources so that load imbalance is kept under control.

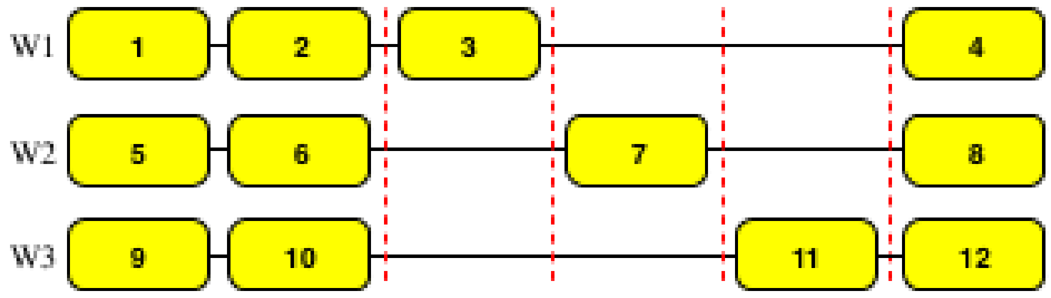
Data parallelism

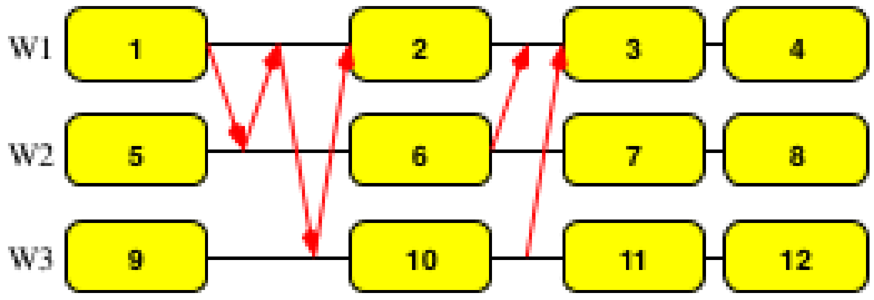
- As an example consider a two-dimensional simulation code that updates physical variables on a **n×n grid**
- Domain decomposition subdivides the computational domain into N subdomains
- The computational effort should be equal for all domains to avoid load imbalance
- It may be necessary to communicate data across domain boundaries
- The communication cost grows linearly with the distance that has to be bridged in order to calculate observables at a certain point of the grid



Functional parallelism

- Spliting a complete problem into disjoint subtasks that can be worked on in parallel
- For instance, assign some resources to communication and others to computational work





Performance models for parallel scalability

- In a simple model, the overall problem size ("amount of work") shall be **s + p = 1**
- The 1-CPU (serial) runtime for this case: $T_{\mathrm{f}}^{\mathrm{s}} = s + p$
- Solving the same problem on N CPUs: $T_{\mathrm{f}}^{\mathrm{p}} = s + \frac{p}{N}$
- This is called strong scaling because the amount of work stays constant no matter how many CPUs are used
- Here the goal of parallelization is minimization of time to solution for a given problem

Performance models for parallel scalability

- For larger problem sizes, for which available memory is the limiting factor, it is appropriate to scale the problem size with some power of N so that the total amount of work is $s + pN^{\alpha}$
- The serial runtime for the scaled problem is defined as:

$$T_{\mathrm{v}}^{\mathrm{s}} = s + pN^{\alpha}$$

- The parallel runtime is: $T_{\mathrm{v}}^{\mathrm{p}} = s + pN^{\alpha - 1}$

- This approuch is called weak scaling

Scalability limitations

Scalability limitations