Introduction to parallel computers and parallel programming

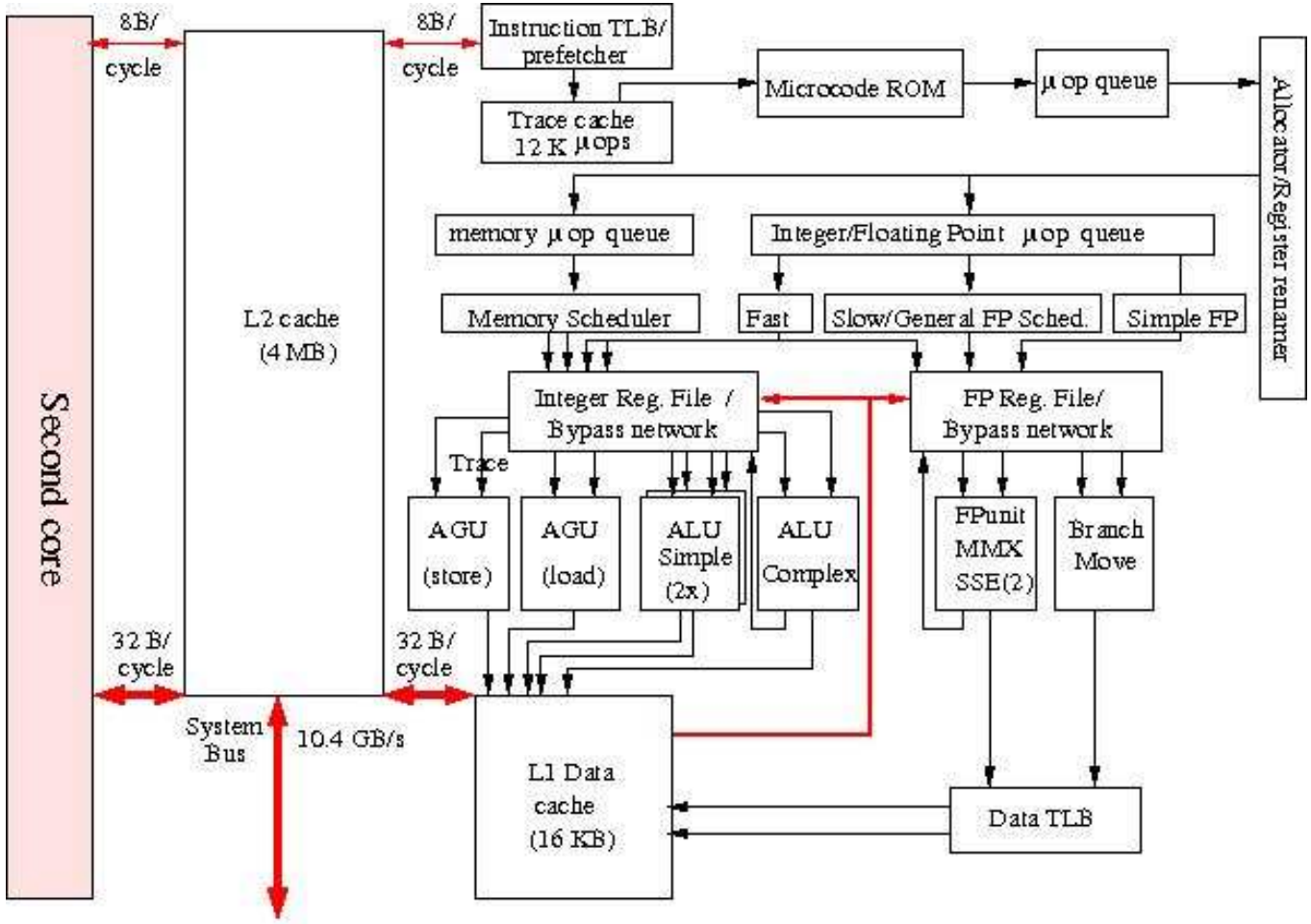
Content

- A quick overview of morden parallel hardware
  - Parallelism within a chip
    - Pipelining
    - Superscaler execution
    - SIMD
    - Multiple cores
  - Parallelism within a compute node
    - Multiple sockets
    - UMA vs. NUMA
  - Parallelism across multiple nodes
- A very quick overview of parallel programming

First things first

- CPU—*central processing unit*—is the “brain” of a computer
- CPU processes *instructions*, many of which require data transfers from/to the *memory* on a computer
- CPU integrates many components (registers, FPUs, caches...)
- CPU has a “clock”, which at each *clock cycle* synchronizes the logic units within the CPU to process instructions

An example of a CPU core

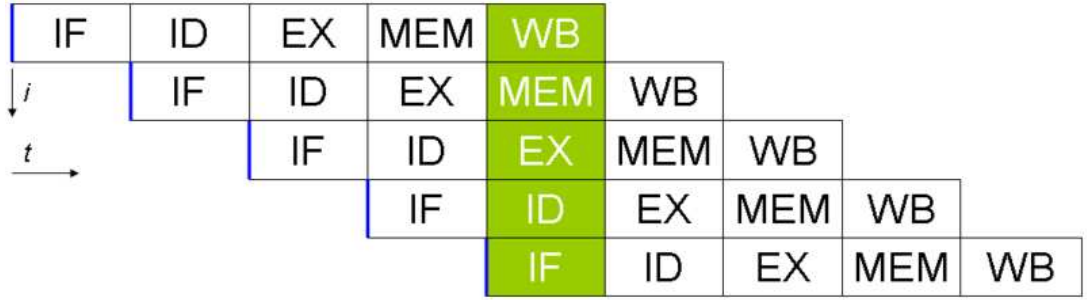


Block diagram of an Intel Xeon Woodcrest CPU core

Instruction pipelining

Suppose every instruction has five stages, each taking one cycle

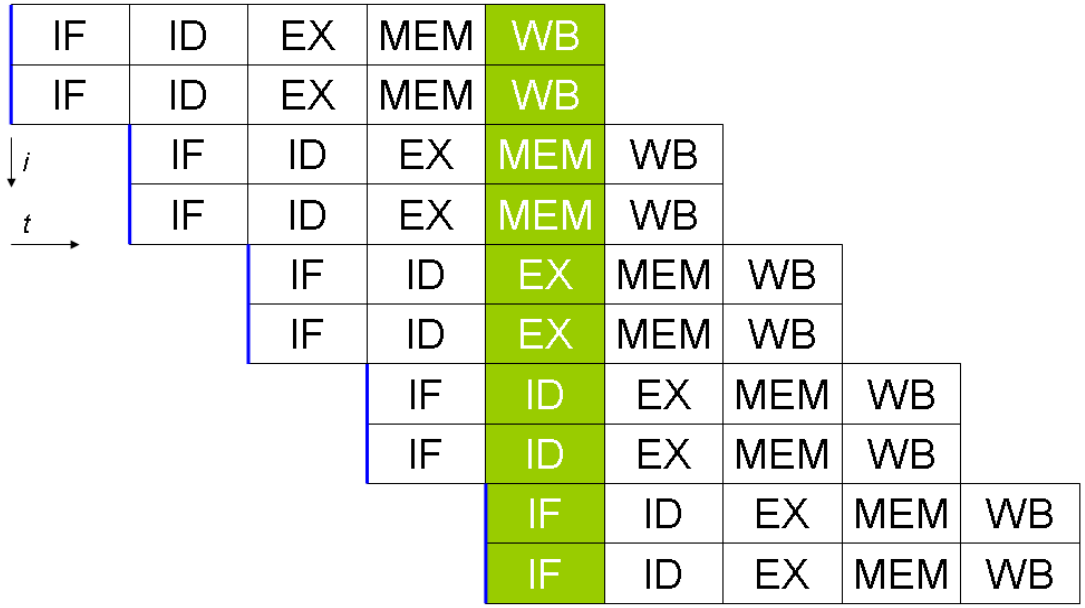
Without instruction pipelining



With instruction pipelining

Superscalar execution

Multiple execution units $\Rightarrow$ more than one instruction can finish per cycle

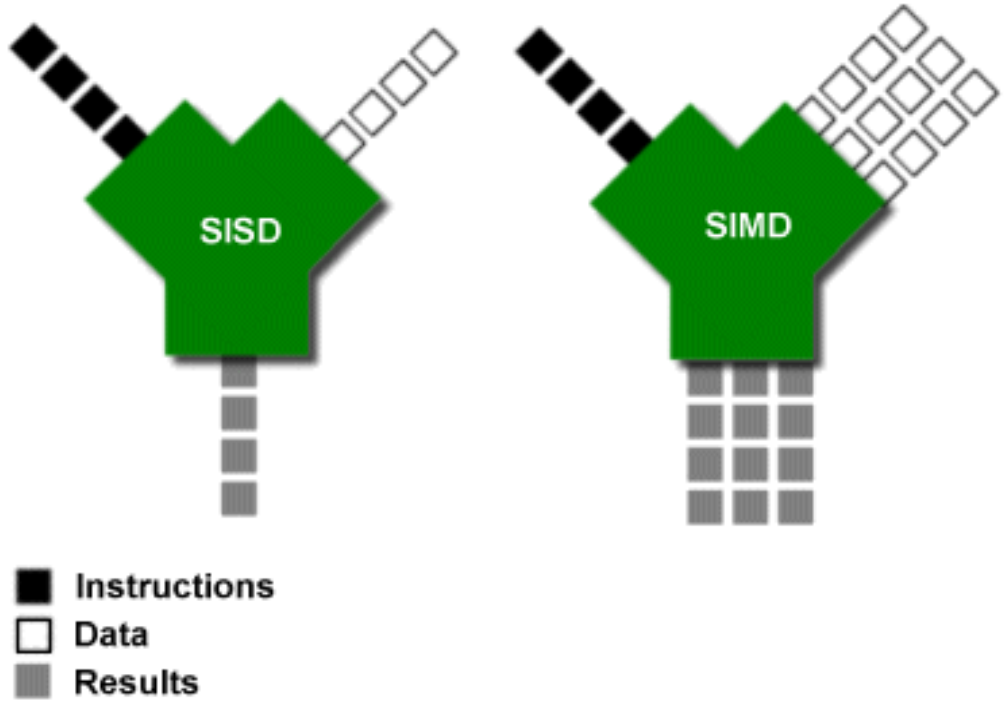


An enhanced form of instruction-level parallelism

Data

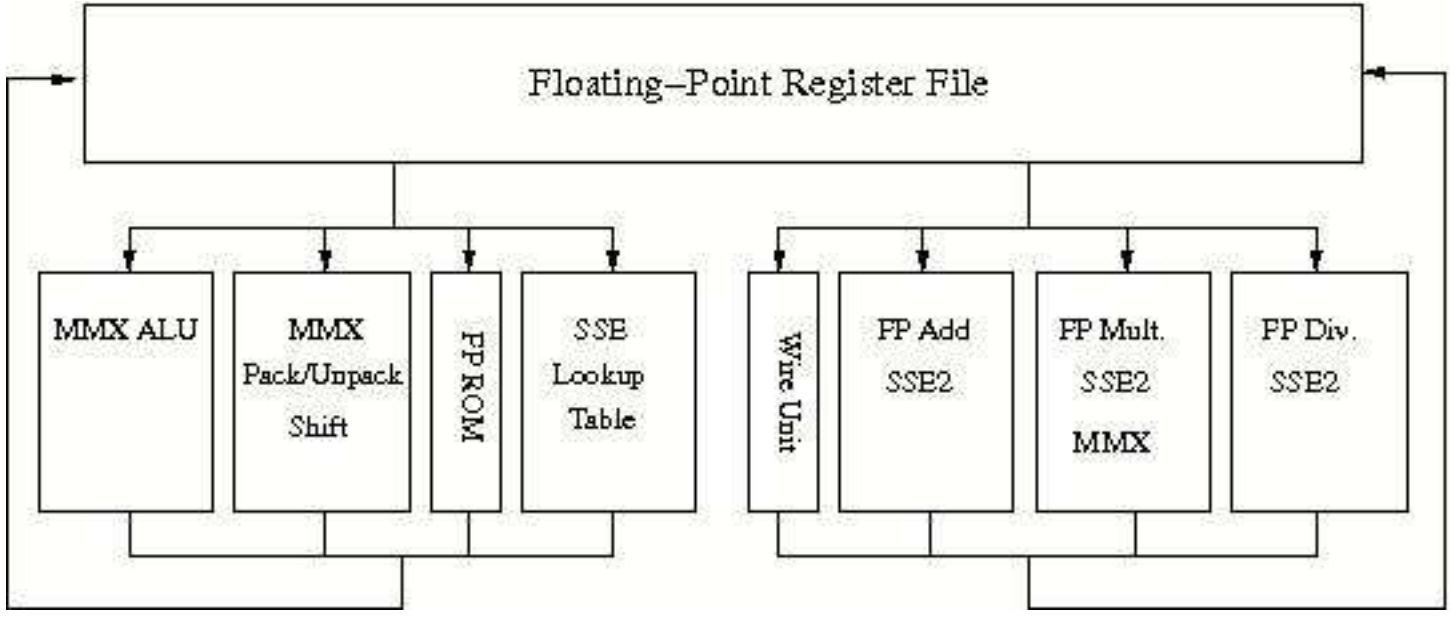
- Data are stored in computer memory as sequence of 0s and 1s
- Each 0 or 1 occupies one *bit*
- 8 bits constitute one *byte*
- Normally, in the C language:
  - `char`: 1 byte
  - `int`: 4 bytes
  - `float`: 4 bytes
  - `double`: 8 bytes
- Bandwidth—the speed of data transfer—is measured as number of bytes transferred per second

SIMD



- SISD: *single instruction stream single data stream*
- SIMD: *single instruction stream multiple data streams*

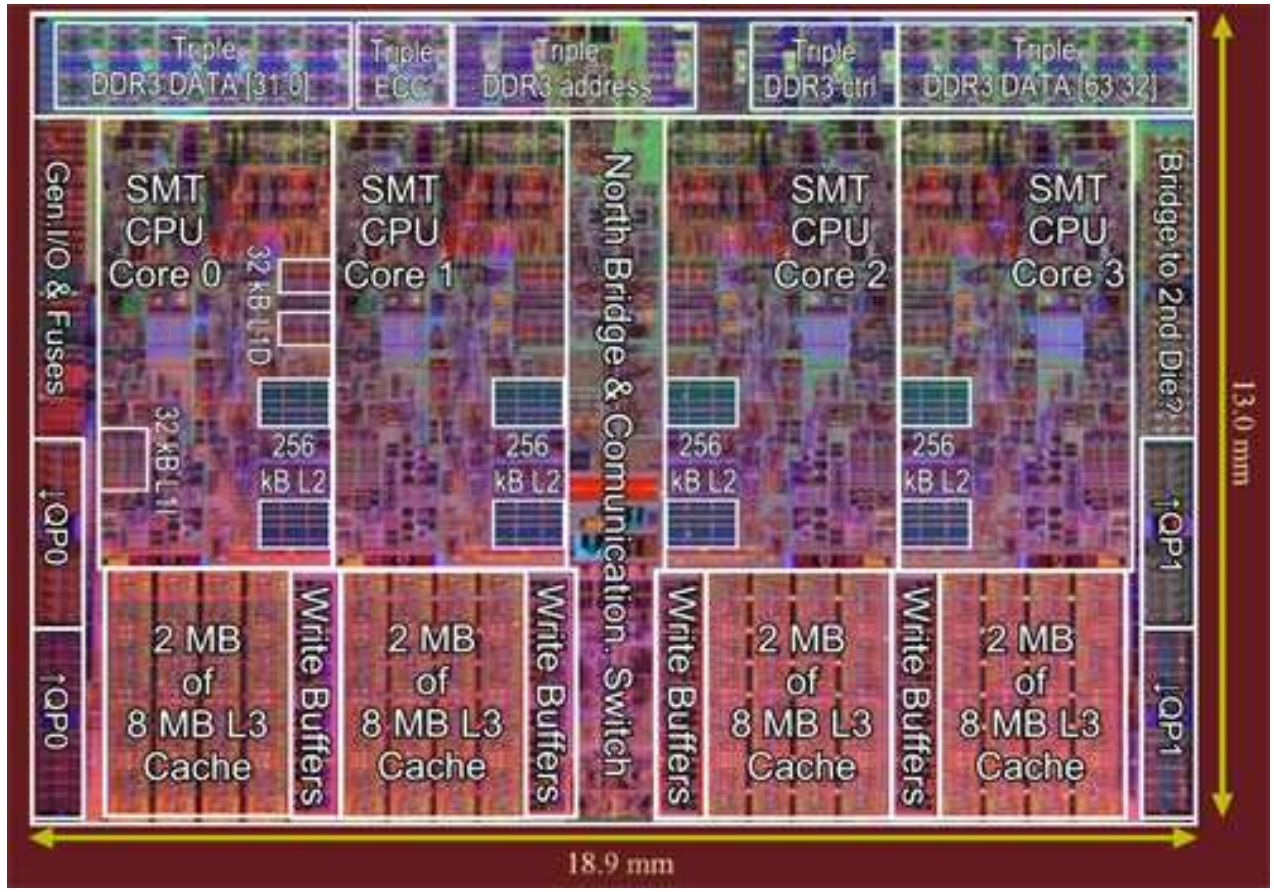
An example of a floating-point unit



FP unit on an Intel Xeon CPU

Multicore processor

Modern hardware technology can put several independent CPU cores on the same chip—a multicore processor



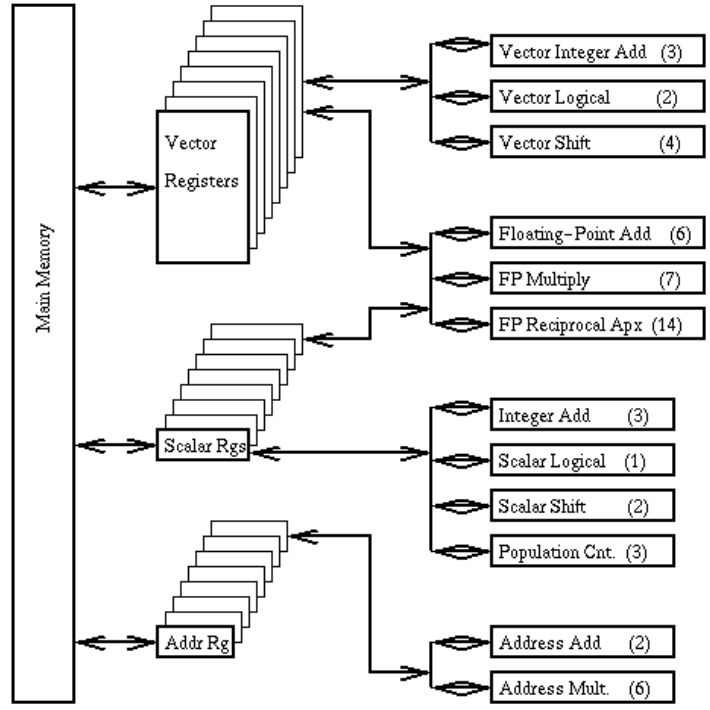
Intel Xeon Nehalem quad-core processor

Multi-threading

- Modern CPU cores often have threading capability
- Hardware support for multiple threads to be executed within a core
- However, threads have to share resources of a core
  - computing units
  - caches
  - translation lookaside buffer

Vector processor

- Another approach, different from multicore (and multi-threading)
- Massive SIMD
  - Vector registers
  - Direct pipes into main memory with high bandwidth
- Used to be the dominating high-performance computing hardware, but now only niche technology



Multi-socket

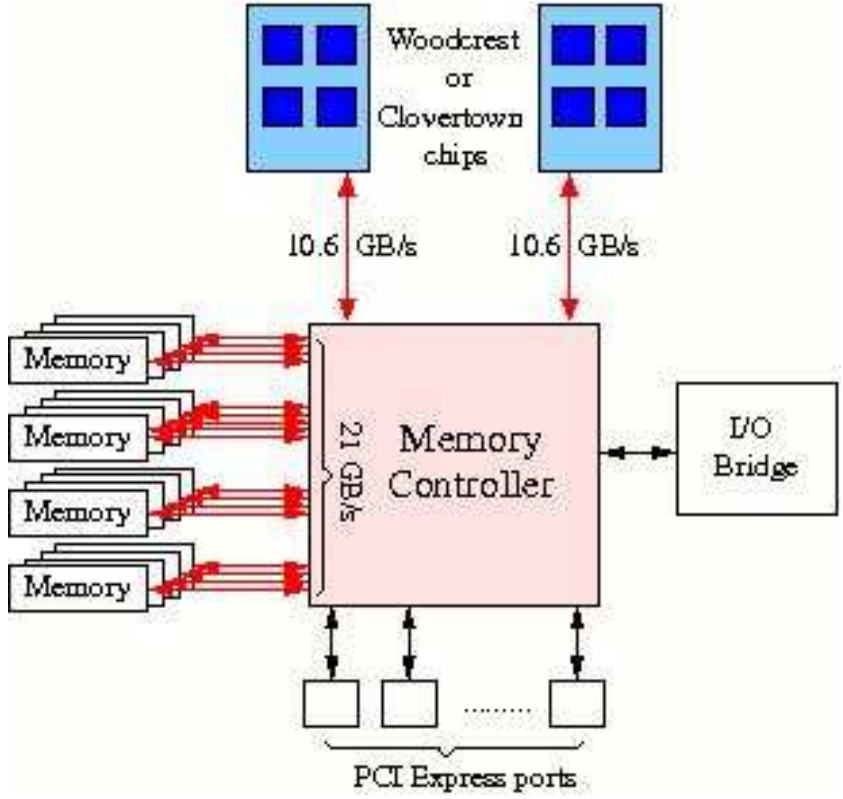
- Socket—a connection on motherboard that a processor is plugged into
- Modern computers often have several sockets
  - Each socket holds a multicore processor
  - Example: Nehalem-EP (2$\times$socket, quad-core CPUs, 8 cores in total)

Shared memory

- *Shared memory*: all CPU cores can access all memory as global address space
- Traditionally called “multiprocessor”

UMA

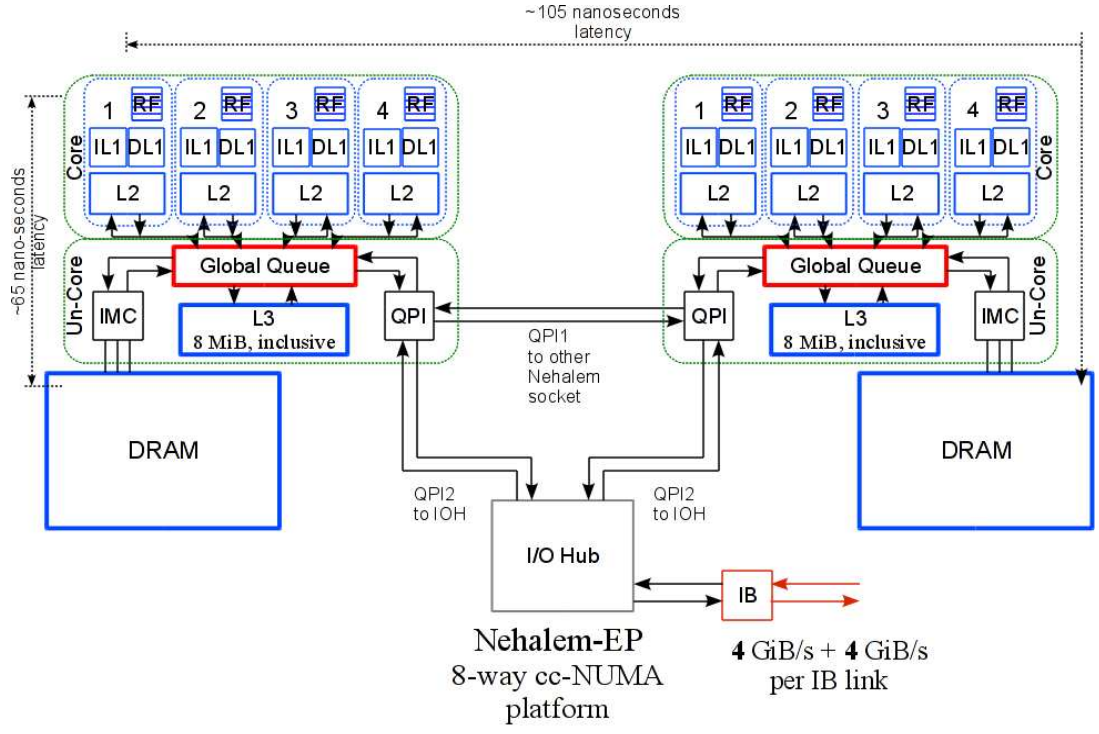
- UMA—*uniform memory access*, one type of shared memory
- Another name for symmetric multi-processing



Dual-socket Xeon Clovertown CPUs

NUMA

- NUMA—*non-uniform memory access*, another type of shared memory
- Several symmetric multi-processing units are linked together
- Each core should access its closest memory unit, as much as possible



Dual-socket Xeon Nehalem CPUs

Cache coherence

- Important for shared-memory systems
- If one CPU core updates a value in its private cache, all the other cores “know” about the update
- Cache coherence is accomplished by hardware

Chapter 2.4.6 of *Introduction to Parallel Computing* describes several strategies of achieving cache coherence

“Competition” among the cores

- Within a multi-socket multicore computer, some resources are shared
- Within a socket, the cores share the last-level cache
- The memory bandwidth is also shared to a great extent

# cores	1	2	4	6	8
BW	3.42 GB/s	4.56 GB/s	4.57 GB/s	4.32 GB/s	5.28 GB/s

Actual memory bandwidth measured on a 2$\times$socket quad-core Xeon Harpertown

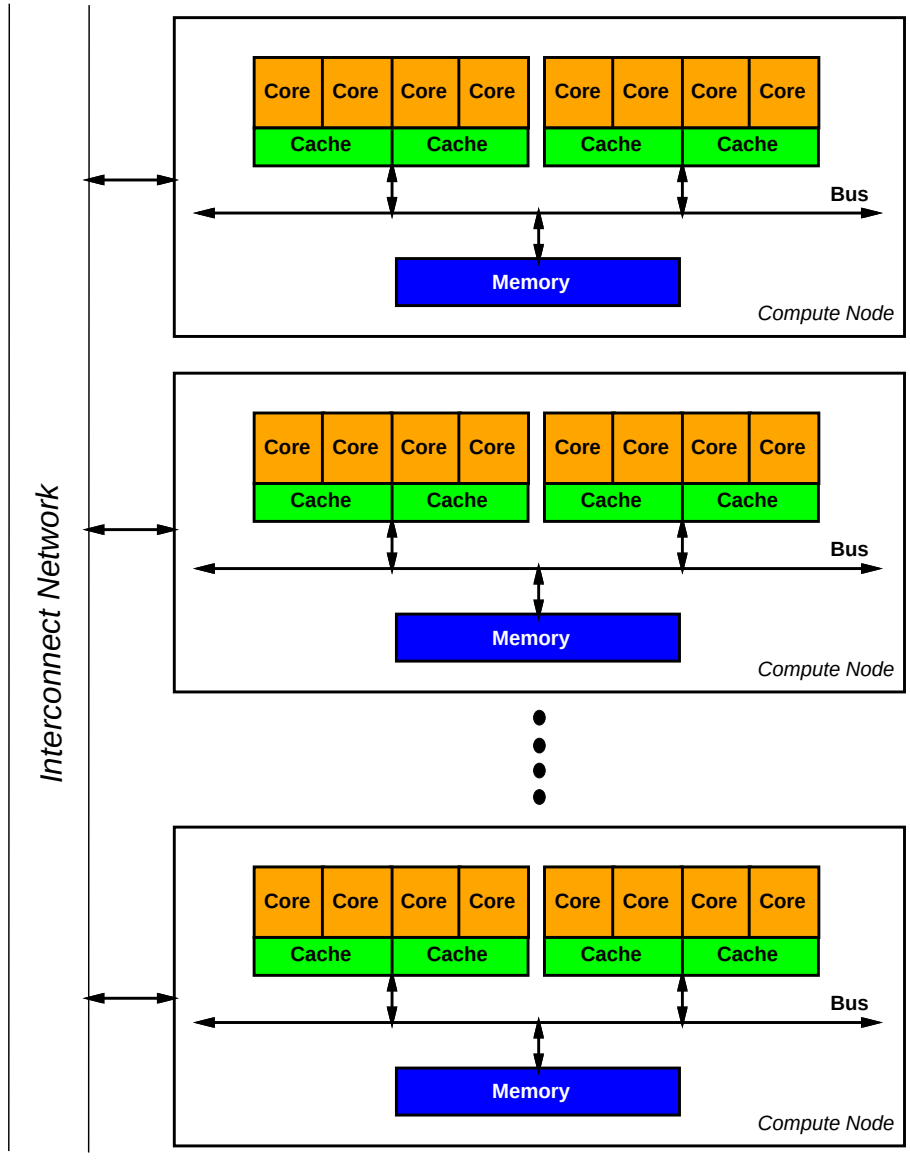
Distributed memory

- The entire memory consists of several disjoint parts
- A communication network is needed in between
- There is not a single global memory space
- A CPU (core) can directly access its own local memory
- A CPU (core) cannot directly access a remote memory
- A distributed-memory system is traditionally called a “multicomputer”

Comparing shared memory and distributed memory

- Sharedmemory
  - User-friendly programming
  - Data sharing between processors
  - Not cost effective
  - Synchronization needed
- Distributed-memory
  - Memory is scalable with the number of processors
  - Cost effective
  - Programmer responsible for data communication

Hybrid memory system



Different ways of parallel programming

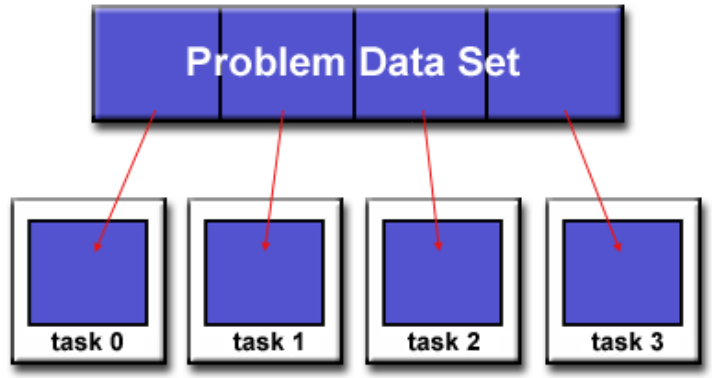
- Threads model using OpenMP
  - Easy to program (inserting a few OpenMP directives)
  - Parallelism "behind the scene" (little user control)
  - Difficult to scale to many CPUs (NUMA, cache coherence)
- Message passing model using MPI
  - Many programming details
  - Better user control (data & work decomposition)
  - Larger systems and better performance
- Stream-based programming (for using GPUs)
- Some special parallel languages
  - Co-Array Fortran, Unified Parallel C, Titanium
- Hybrid parallel programming

Designing parallel programs

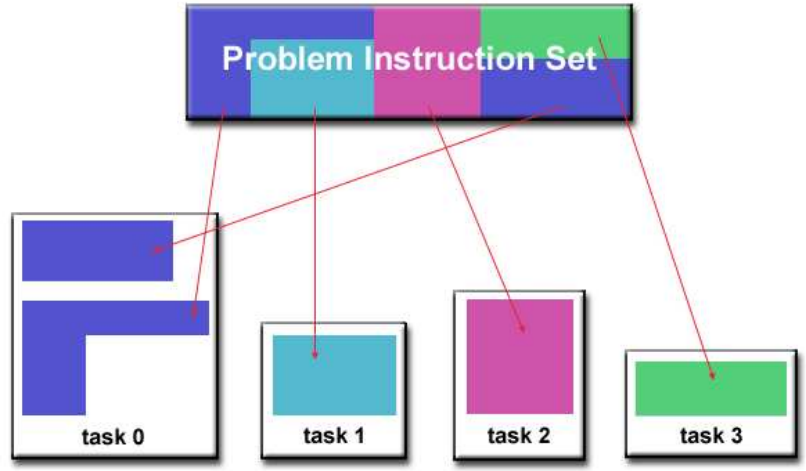
- Determine whether or not the problem is parallelizable
- Identify “hotspots”
  - Where are most of the computations?
  - Parallelization should focus on the hotspots
- Partition the problem
- Insert collaboration (unless embarrassingly parallel)

Partitioning

- Break the problem into “chunks”
- Domain decomposition (data decomposition)

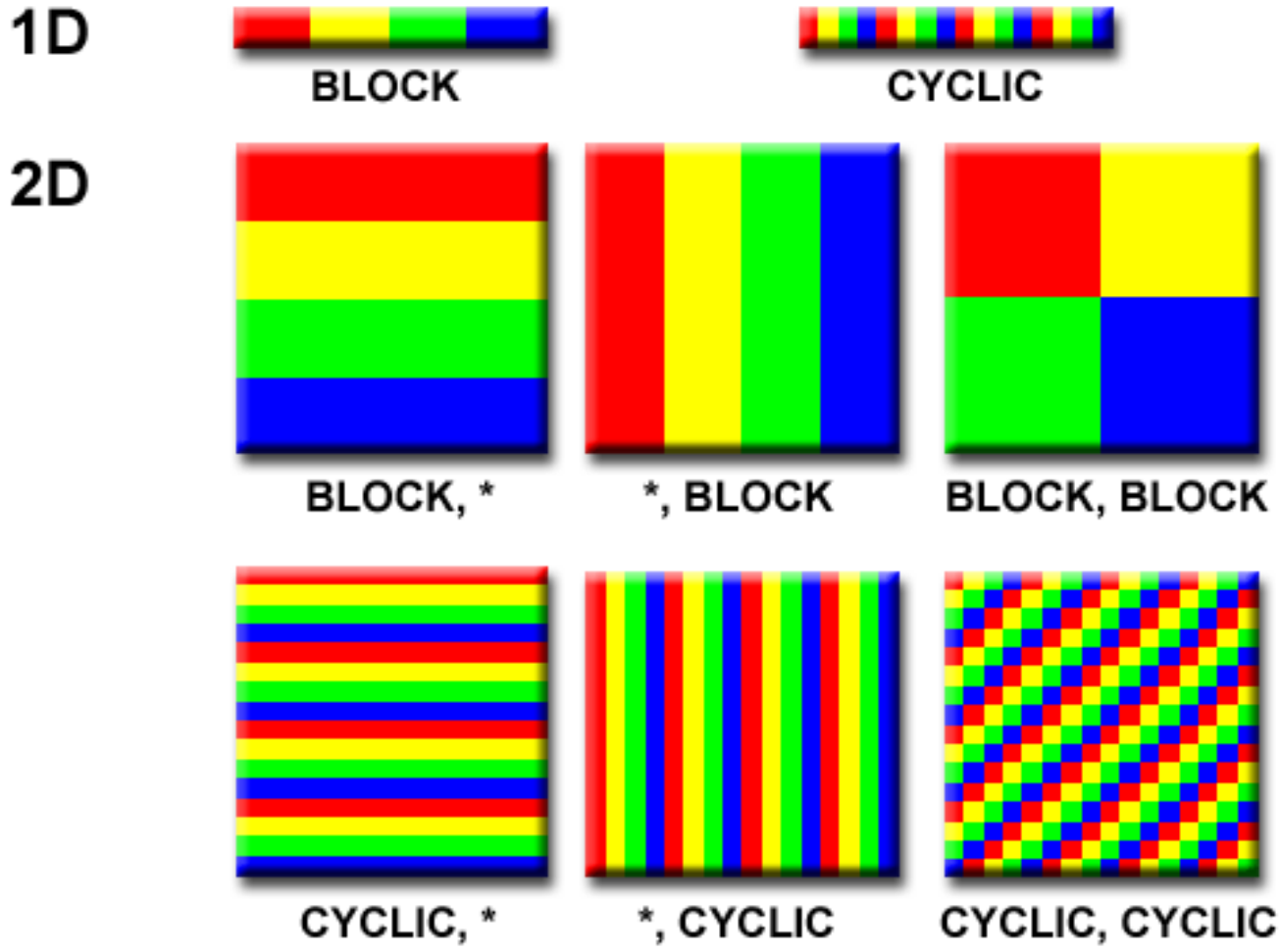


- Functional decomposition



https://computing.llnl.gov/tutorials/parallel_comp/

Examples of domain decomposition



https://computing.llnl.gov/tutorials/parallel_comp/

Collaboration

- Communication
  - Overhead depends on both the number and size of messages
  - Overlap communication with computation, if possible
  - Different types of communications (one-to-one, collective)
- Synchronization
  - Barrier
  - Lock & semaphore
  - Synchronous communication operations

Load balancing

- Objective: idle time is minimized
- Important for parallel performance
- Balanced partitioning of work (and/or data)
- Dynamic work assignment may be necessary

Granularity

- Computations are typically separated from communications by synchronization events
- *Granularity*: ratio of computation to communication
- Fine-grain parallelism
  - Individual tasks are relatively small
  - More overhead incurred
  - Might be easier for load balancing
- Coarse-grain parallelism
  - Individual tasks are relatively large
  - Advantageous for performance due to lower overhead
  - Might be harder for load balancing

`https://computing.llnl.gov/tutorials/parallel_comp/`