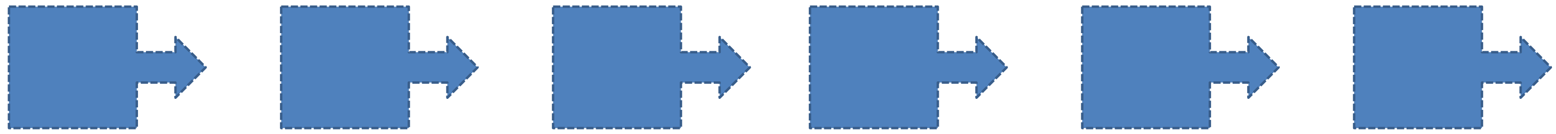
Parallel Scientific Computation

Basic Concepts

J.-H. Parq
IPCST
Seoul National University

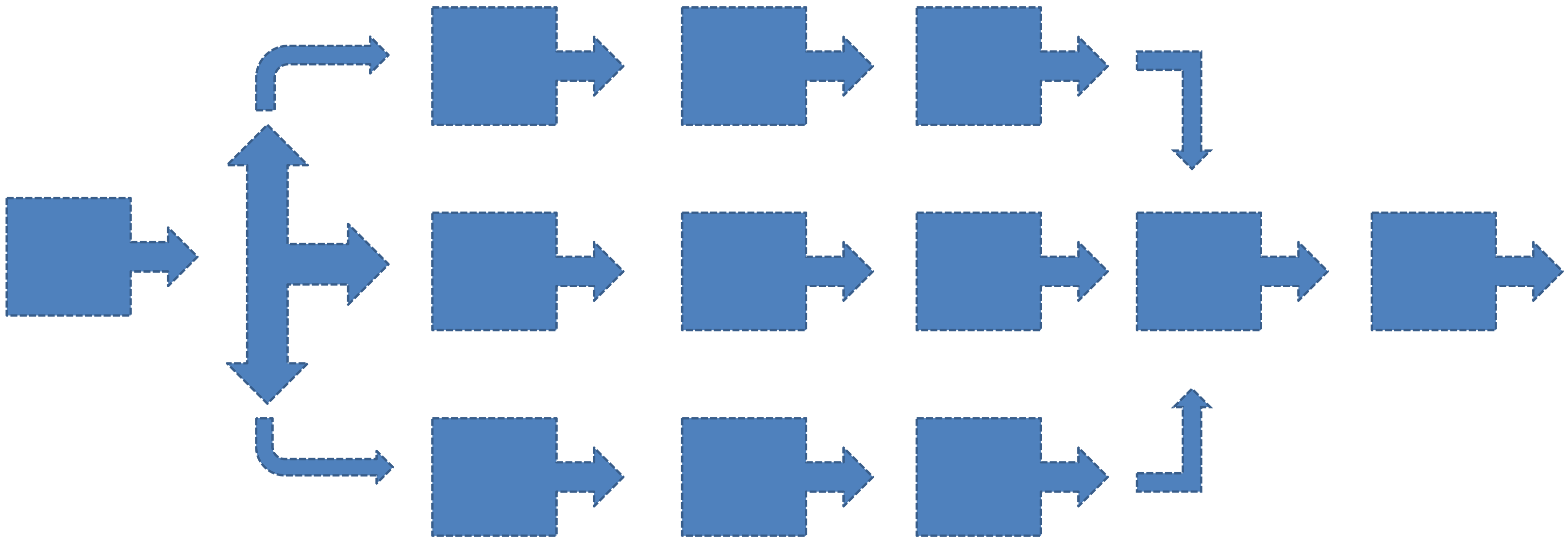
Serial vs Parallel

- Serial computing
  - Program: instructions in one sequence
  - Run on single core of one CPU (Central Processing Unit) in one computer



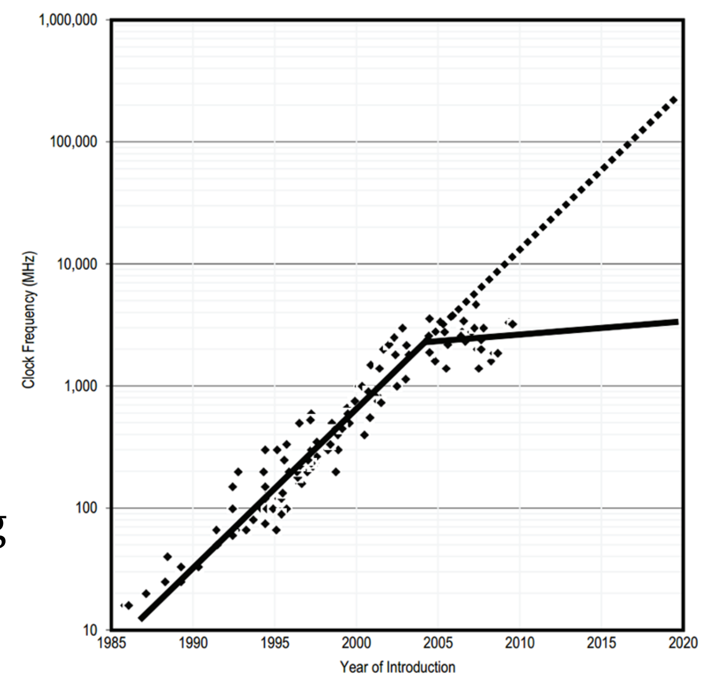
Serial vs Parallel

- Parallel computing
  - Program: divided instructions or data
  - Run on different CPUs or cores (concurrently)



Why Parallel Computing?

- Overcoming limits of serial computing
  - **Reducing time cost**
    - Since about 2005, it is very difficult to enhance the speed of one CPU due to its physical limit
    - Now multi-core CPUs are the mainstream



• **CPU frequency from 1986 to 2008**
Picture from 'The Future of Computing Performance' (2011)

Why Parallel Computing?

- Overcoming limits of serial computing
  - Dividing memory cost
    - Some problems require huge memory usage, and there is a memory capacity limit for a computer → connecting computers to divide data
    - Too much memory access also increases total computation time (but often negligible)

Spread of Parallel Computing

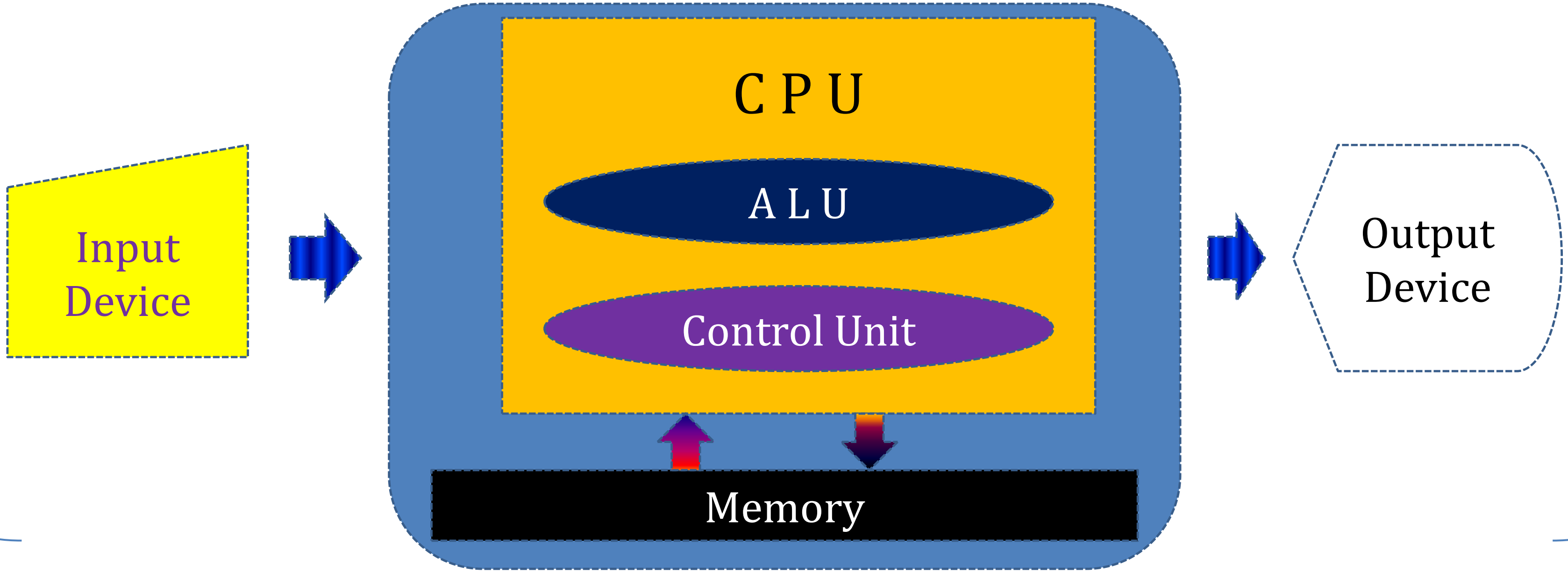
- Large scale problems
  - Astronomical problems
  - Global earth science predictions
    - Weather (or climate), ocean, plate, ……
  - Nano-scale irregular materials
  - Virtual product design
  - Big data and data visualization
  - See ‘grand challenges’ in Wikipedia for more.

Spread of Parallel Computing

- Development of hardware
  - Multi-core CPUs
  - GPGPU (General Purpose GPU)
  - FPGA (Field-Programmable Gate Array)
  - InfiniBand network
  - Supercomputers
  - Cloud computing
  - ……

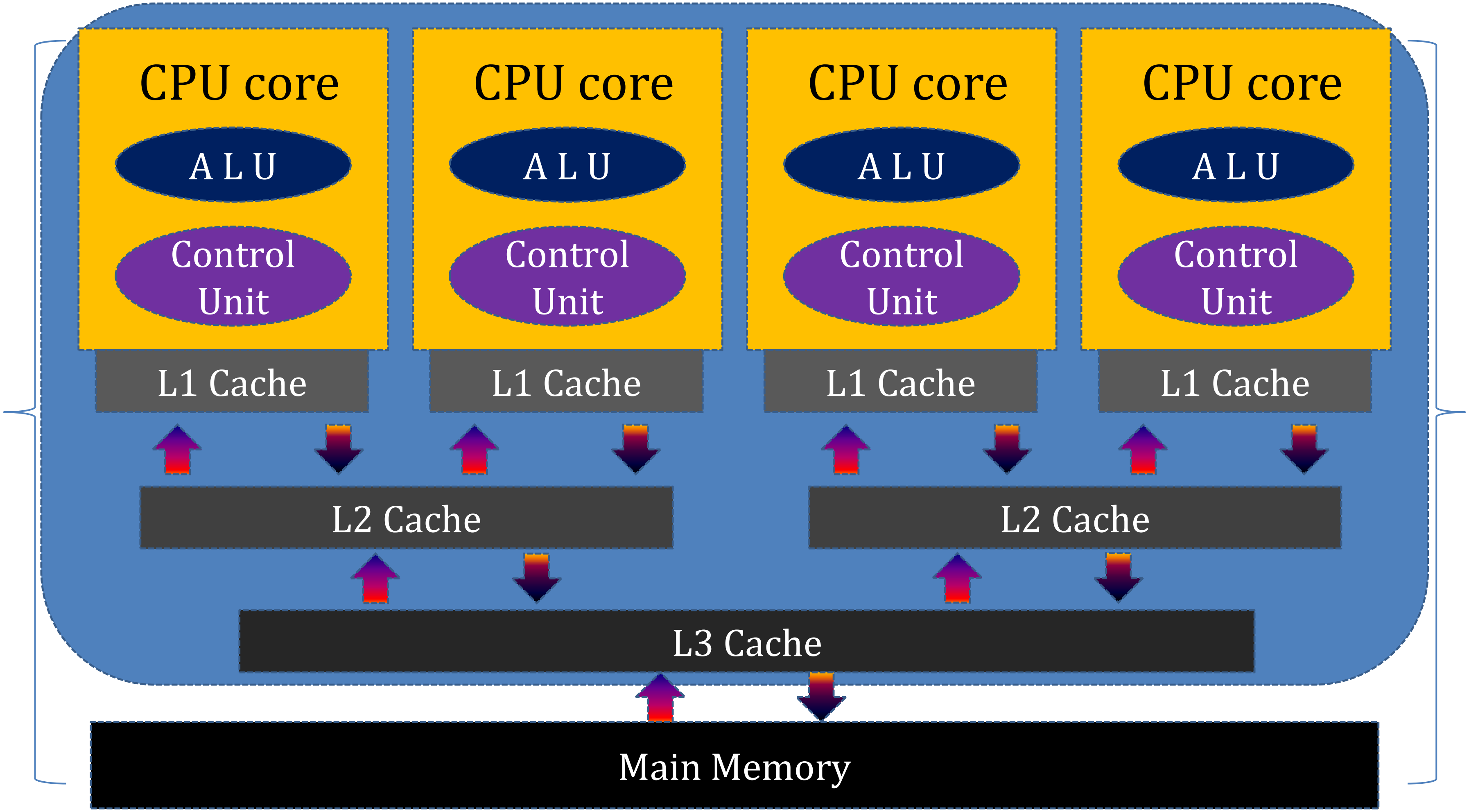
Von Neumann Architecture

- Most commonly used computer model
- A stored-program system
  - Both program and data are stored in memory.



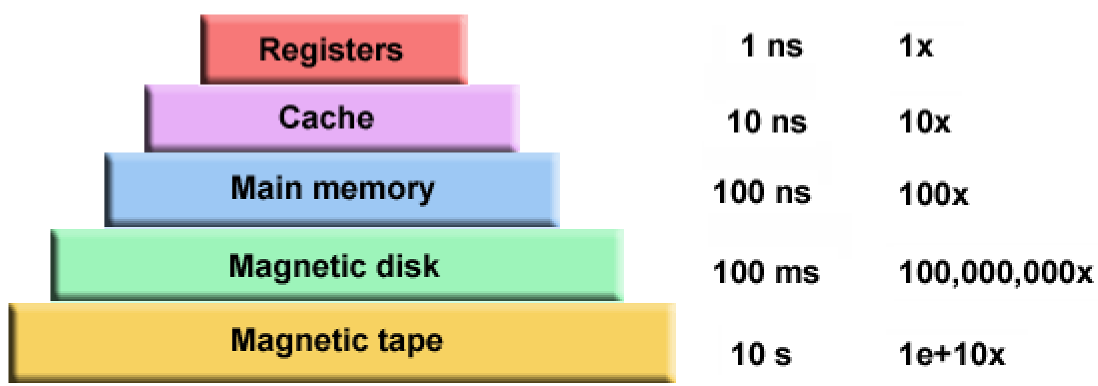
ALU = Arithmetic/Logic Unit

Multi-core CPU



Memory Access Time

- Cache: temporary storage to access fast
  - Speed: L1 > L2 > L3
  - Capacity: L1 < L2 < L3



- Picture from Barney's

Flynn's Classical Taxonomy

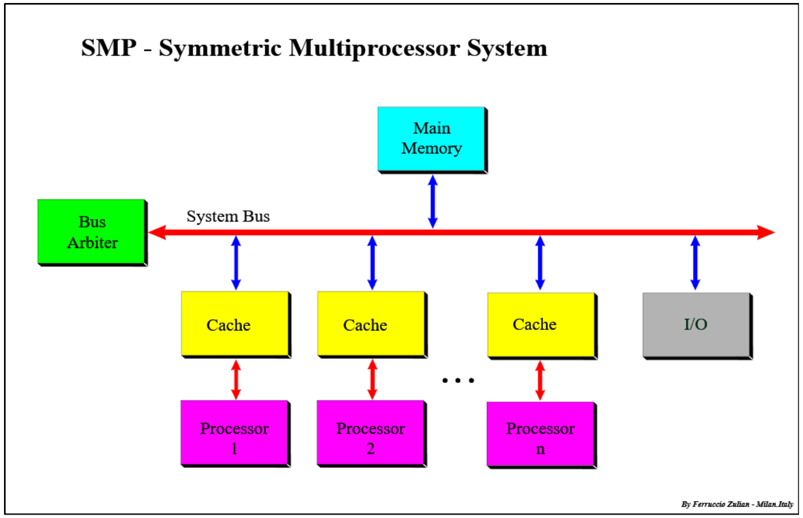
- Old classification of parallel computers
  - Single Instruction, Single Data
    - Serial computers
  - Single Instruction, Multiple Data
    - Vector pipelines, GPUs
  - Multiple Instruction, Single Data
  - Multiple Instruction, Multiple Data
    - The most common type

Terminology

- CPU, processor, socket, core
  - Some vendors call CPUs with multiple cores as ‘sockets’.
- Task: logical portion of computational work
- Pipelining: a type of parallelism
  - Dividing a task into steps for processor units
    - The output of one unit is the input of the next unit.
  - Data pipelines are used in vector processors or array processors or GPUs

Terminology

- Shared memory
  - All tasks access directly to the same memory
- SMP(Symmetric Multi-Processor)
  - A hardware architecture where multiple identical processors share main memory and all input/output devices



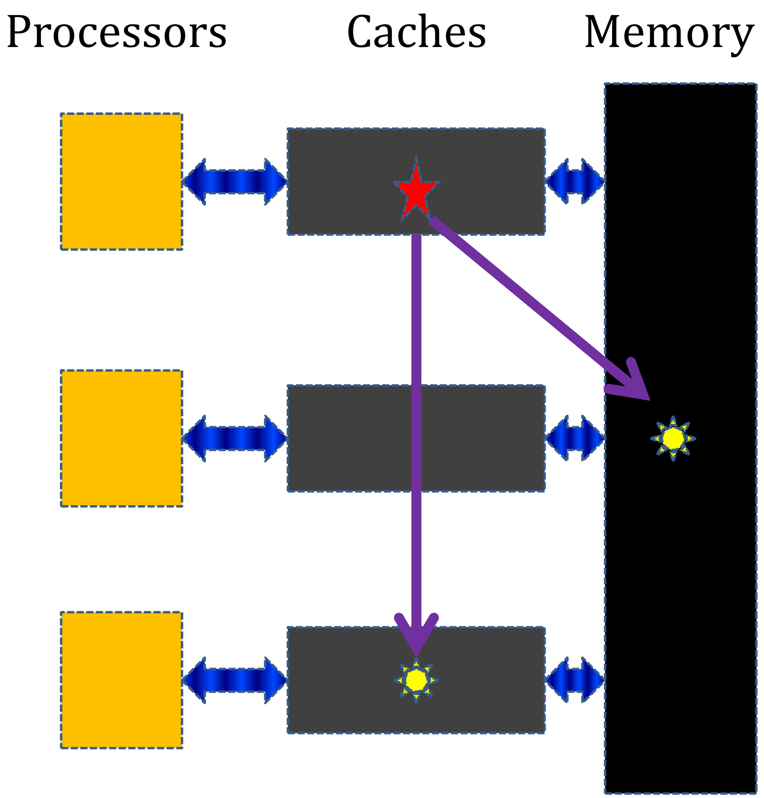
- Picture from Wikipedia

Terminology

- Nodes
  - Individual computers networked together to comprise a supercomputer or a cluster
- Communications
  - Exchanging data among nodes or processors
- Distributed memory
  - Physical memories distributed in nodes connected through network to communicate data

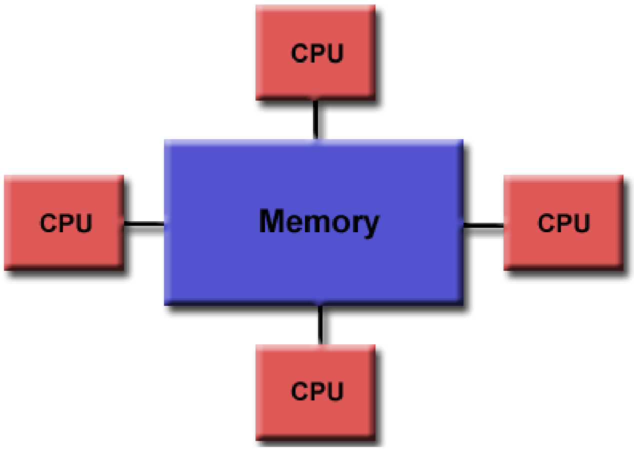
Cache Coherence

- Data consistence
  - Data change in a cache can induce discrepancy of data in caches for the same address.
  - Copying changed data recovers cache coherence.



Parallel Computer Memory Architectures

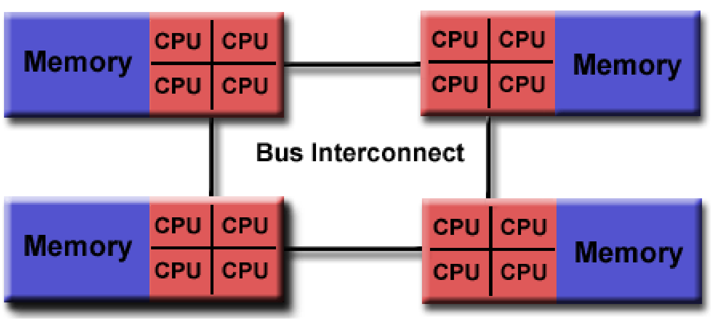
- Shared memory architectures
  - Uniform Memory Access (UMA)
    - Equal access to memory
    - SMP, crossbar switches, or multistage interconnection networks



- Picture from Barney's

Parallel Computer Memory Architectures

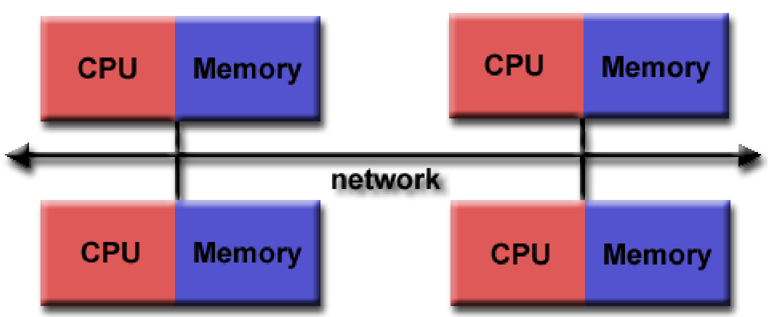
- Shared memory architectures
  - Non-Uniform Memory Access (NUMA)
    - Memory access time depends on location.
    - Most common: physically linked SMPs
    - CC(Cache Coherent)-NUMA: CPU cache controllers communicate to each other.



- Pictures from Barney's

Parallel Computer Memory Architectures

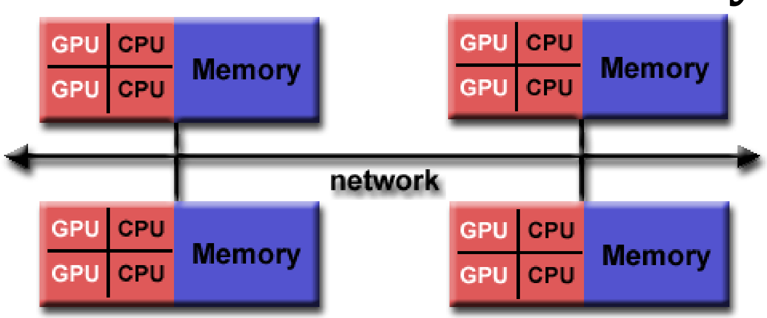
- Distributed memory architecture
  - Each CPU has its own local memory (not cache).
  - Network communication is indispensible.

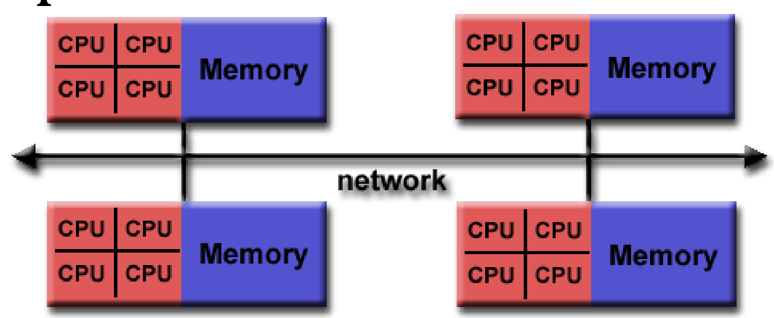


- Picture from Barney's

Parallel Computer Memory Architectures

- Hybrid memory architecture
  - Most widely used for supercomputers or clusters
  - Each node: shared memory – usually cache coherent SMP or GPU
  - The network among nodes represents the distributed memory component





- Pictures from Barney's

Speedup

- Definition

$$\frac{\text{Elapsed real time of serial execution}}{\text{Elapsed real time of parallel execution}}$$

- Efficiency = (speedup)/(# of processors)
- Related factors
  - Granularity
  - Parallel overhead
  - Software algorithm
  - Hardware properties
  - Performance of compiler or other related applications

Terminology

- Granularity
  - Ratio of computation time of a task to communication time
    - **Fine-grained**: large number of small tasks
    - **Coarse-grained**: small number of large tasks
- Parallel overhead
  - Additional time to the time sum of all tasks
    - Task start and termination
    - Communication
    - Synchronization
    - Software overhead by libraries or OS

Terminology

- Synchronization
  - **Timing** between or among tasks
  - Important during communications or joins
  - Usually, at least one task waits
- *Embarrassingly parallel* problems
      - Perfectly parallel, delightfully parallel, ……
  - A whole program for this kind of problems is inherently parallel.
  - Every task is **independent**.
  - No or little communications are necessary.

Amdahl's law

- If the parallelizable fraction of a code is P, the speedup is, ignoring parallel overheads,

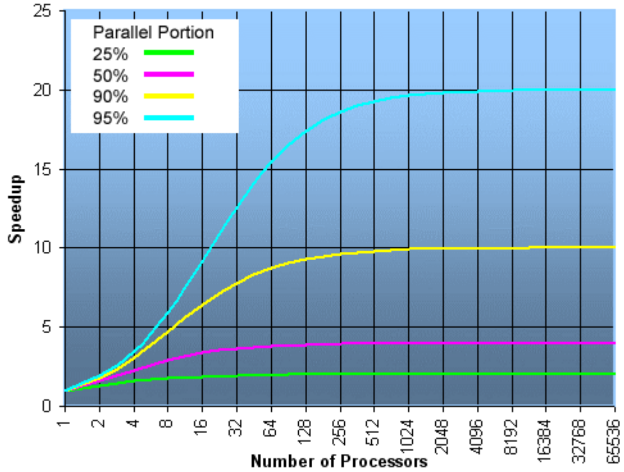
  $$S = 1/(P/n + 1 - P)$$

  where n is the number of working processors

  – or

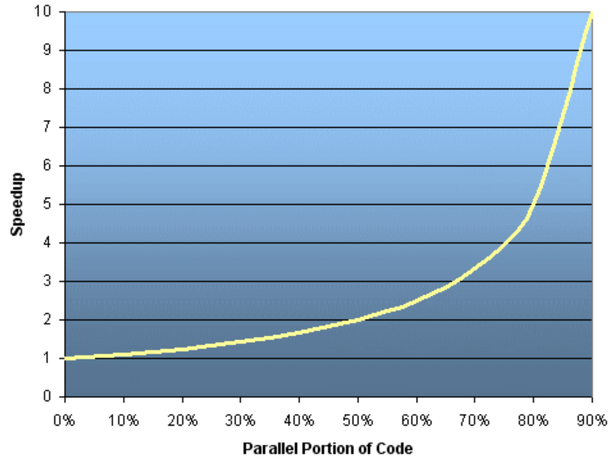
  $$S < 1/(P/n + 1 - P)$$

  considering overheads



- Picture from Barney's

Amdahl's law



- Picture from Barney's
  - Speedup limit: $S \rightarrow 1/(1 - P)$ as $n \rightarrow \infty$
    - About 99 % of max. speedup when $n \cong 99P/(1 - P)$
      - Ex.) 1.98 speedup if $P = 50$ % and $n = 99$
      - Ex.) 9.9 speedup if $P = 90$ % and $n = 891$
  - Embarrassingly parallel: $S = n$

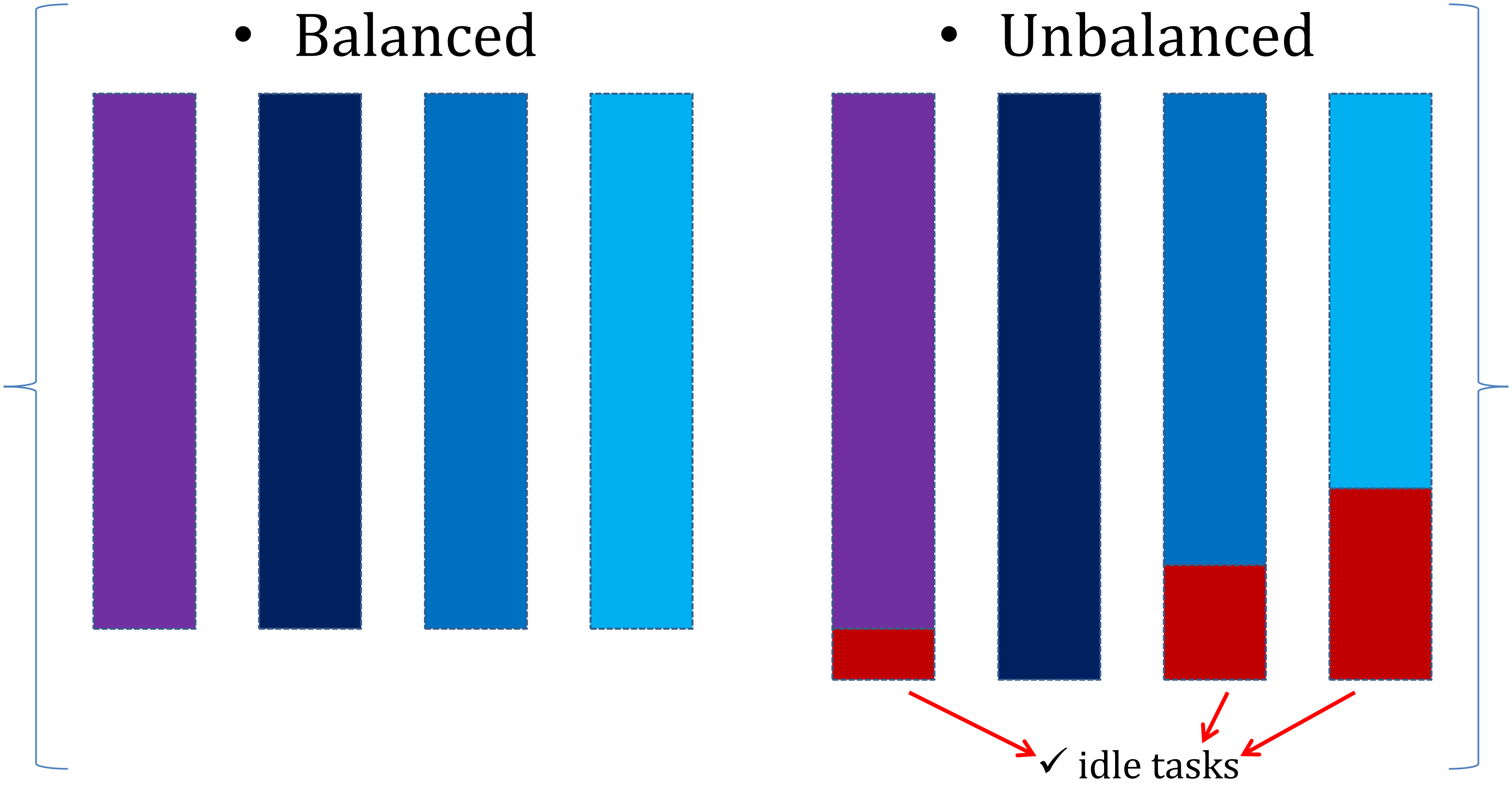
Load Balancing

- Even for an embarrassingly parallel problem, usually speedup $S < n$, why?
  - Parallel overhead
  - Load imbalance

- Ideally, all tasks should end *at the same time*.
- If not, tasks should be *rearranged* so that load of every task can be *about the same*.

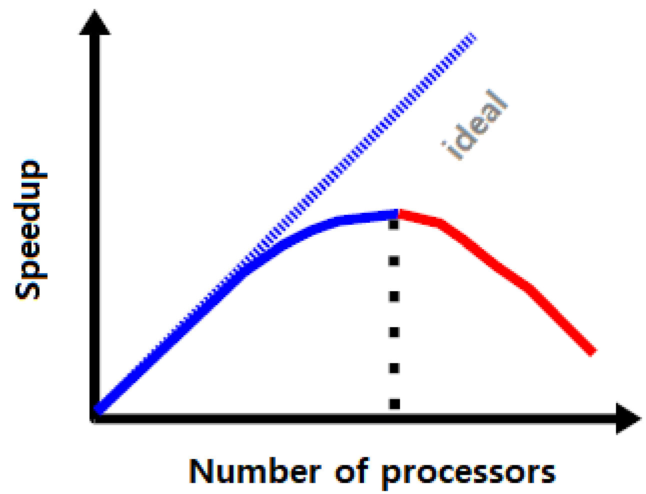
Load Balancing

- Balanced
- Unbalanced



Scalability

- – Ability to increase speedup by adding resources (for example, more CPUs)
- Actual scaling
  - – The speedup curve usually declines after passing a maximum.



- Limiting factors
  - – Algorithm
  - – Hardware: significant in scalability
    - Communication, memory, CPU frequency
  - – Parallel support libraries and subsystems software

Scalability

- Two types of scaling concepts in terms of high-performance computing
  - **Strong scaling**
    - Fixed total problem size
    - Aim: *Faster performance* for the *same problem* size
  - **Weak scaling**
    - Fixed problem size per processor
    - Aim: *Larger problem* in the *same* amount of *time*

Parallel Programming Models

- Shared memory without threads
  - One task assigned to one processor (1 core)
  - Asynchronous access to memory
  - Ex.) POSIX shared memory API, SHMEM, X10, Chapel, ……
- Multi-threading (for shared memory)
  - One task assigned to one thread, but one processor can take two or more threads.
  - Ex.) OpenMP, OpenACC, CUDA, POSIX Threads, …

Parallel Programming Models

- Message passing (for distributed memory)
  - Tasks communicate data.
  - MPI
- Data parallel model
  - Partitioned Global Address Space (PGAS) model
  - Data structure is common (ex. Arrays).
  - Every task executes the same operation on its partition of the data.
  - Ex.) Coarray Fortran, Unified Parallel C, Global Arrays, ……

Parallel Programming Models

- Hybrid
  - Ex.) MPI+OpenMP, MPI+CUDA
- Single program multiple data
  - Tasks may treat different instructions and data.
  - Usually with MPI or the hybrid model
- Multiple program multiple data
  - Rarely used model

Designing Parallel Programs

1. Understanding the **problem**
2. **Serial** program
3. Determining if it is **parallelizable** or not
4. Finding the **hot spot** of the serial program
   - Hot spot: the most time consuming part
5. Finding **bottlenecks**
   - Bottlenecks: slow parts unable to be parallelized
6. Choosing a parallel **algorithm**
7. Execution and **debugging**
8. Performance analysis and **optimization**
   - You may change your parallel algorithm

Designing Parallel Programs

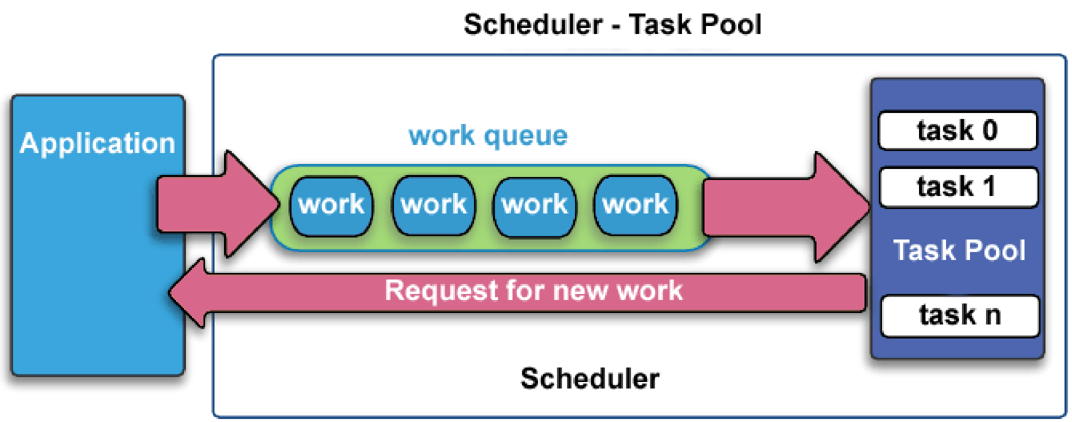
- Partitioning choice
  - Data division
  - Functional division
- Synchronization types
  - Barrier: Early-end tasks standby.
  - Lock/Block: One task monopolizes some data for a while.
  - Synchronous communication: in MPI

Designing Parallel Programs

- Communication
  - To be discussed in the MPI lectures
- Data dependencies
  - Two computations have a data dependence if **one** uses a set of memory locations for data **writing** where **the other** uses for data **access** (reading or writing)
  - If consecutive computations have **no data dependencies**, they can be computed independently in **parallel**.
  - Loop parallelism: to be discussed later

Designing Parallel Programs

- Load balancing
  - A few ways to get load balance
    - Equal partition
    - Dynamic redistribution: estimating and realloting
    - Scheduler



- Picture from Barney's

Designing Parallel Programs

- Granularity
  - Fine-grained
    - Easy to get load balance but high overhead
  - Coarse-grained
    - Low communication and synchronization overhead
- Input/Output
  - Usually unable to be parallelized
  - There are several parallel file systems.
    - GPFS, PVFS, GFS, QFS, Hadoop, Lustre, ……

References

- L. R. Scott, T. Clark, and B. Bagheri,
  Scientific Parallel Computing
- B. Barney,
  Introduction to Parallel Computing
- Wikipedia