Course name: Parallel Programming

Course code: DSE 3254

No of contact hours/week: 4

Credit: 4

By
Sandhya Parasnath Dubey and Girisha Surathkal
Assistant Professor,
Department of Data Science and Computer Applications - MIT, Manipal Academy of Higher Education, Manipal-576104, Karnataka, India.
Mob: +91-9886542135
Email: sandhyadubey24@manipal.edu

"Our greatest weakness lies in giving up. The most certain way to succeed is always to try just one more time"

OBJECTIVES

- To introduce you to the basic concepts and ideas in parallel computing
- To familiarize you with the major programming models in parallel computing
- To provide you with guidance for designing efficient parallel programs

Introduction to parallel programming

Why parallel computing?

- From 1986-2002
  - Performance of microprocessor increased on an average by 50%
  - But later, the performance gain was reduced by 20%
  - Because computer designers started focusing on designing parallel computers
    - Rather than designing complex single core processors
    - Multi-core processors
- But software developers used to develop **serial programs**
- *Aren't single processor systems fast enough?*
- *Why build parallel systems?*
- *Why we need parallel programs?*

Why we need ever-lasting increase in performance?

- Past improvements in performance of microprocessor resulted in quicker web searcher, accurate and quick medical diagnosis, realistic computer games, etc
- Higher computation power means we can solve larger problems:
    - Climate modelling
    - Protein folding
    - Drug discovery
    - Energy research
    - Data analysis

Why we are building parallel systems?

- Increase in single processor performance has been due the ever-increasing density of transistors
- As the size of transistors decreases, their speed can be increased
    - Their power consumption also increases
    - Dissipates heats
    - Highly unreliable
- Hence, it was impossible to increase the speed of integrated circuits
- But, increasing transistor density can continue
- Rather than building ever-faster, more complex, monolithic processors
- They started bringing out multiple, relatively simple, complete processors on a single chip

Why we need to write parallel programs?

- Most serial programs are designed to run on single core
- They are unaware of multiple processors
- We can at max run multiple instances of same program on multiple cores
  - This is not what we want. Why?

Applications of parallel computing

- Applications in Engineering and Design
  - Optimization problems
  - Internal combustion engines
  - Airfoils designs in aircraft
- Scientific Applications
  - Sequencing of the human genome
  - Weather modeling, mineral prospecting, flood prediction, etc.
- Commercial Application
  - Web and database servers
  - Data mining
  - Analysis for optimizing business and marketing decisions
- Applications in Computer Systems

Introduction

- **Parallel Computing:** It is the use of parallel computer to reduce the time needed to solve computational problem

- **Parallel Computers:** It is a multi-processor computer system that supports parallel programming
  - Two types of parallel computers:
    - **Multi-computer:** Parallel computer constructed out of multiple computers and an inter-connection network
    - **Centralized multi-processors:** An integrated system in which all CPUs share access to a single global memory

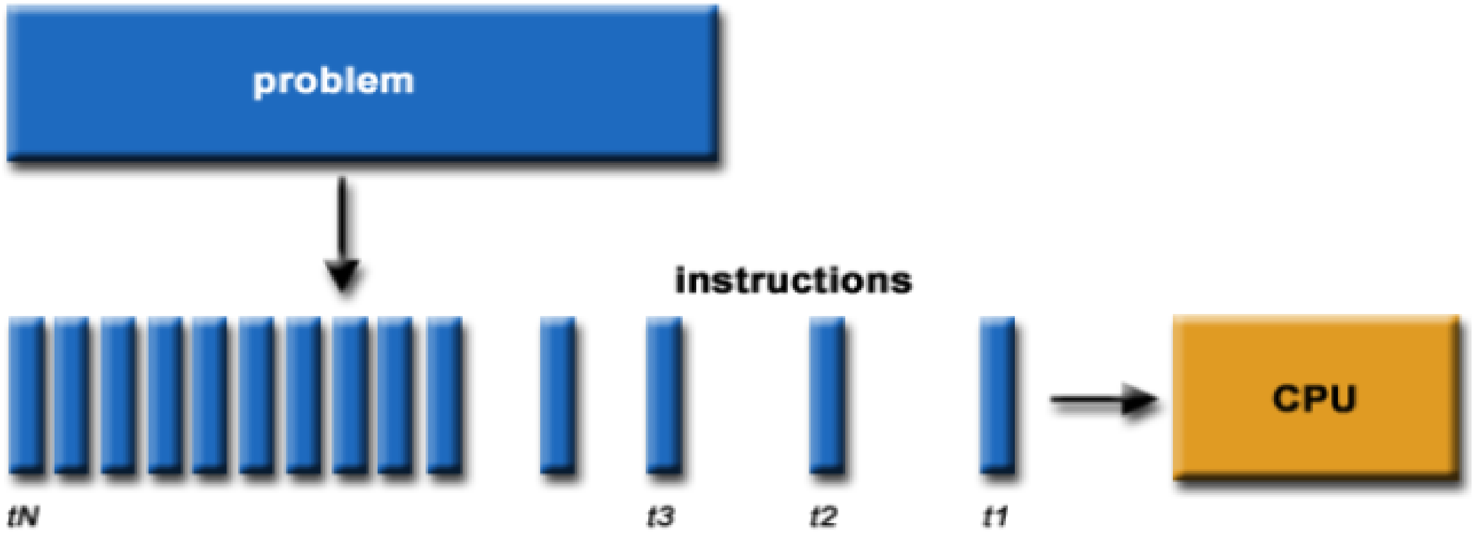
Introduction

- **Parallel Programming:** is programming in a language that allows that allows you to explicitly indicate how different portion of the computation may be executed concurrently

Serial Computation:

Traditionally software has been written for serial computations:

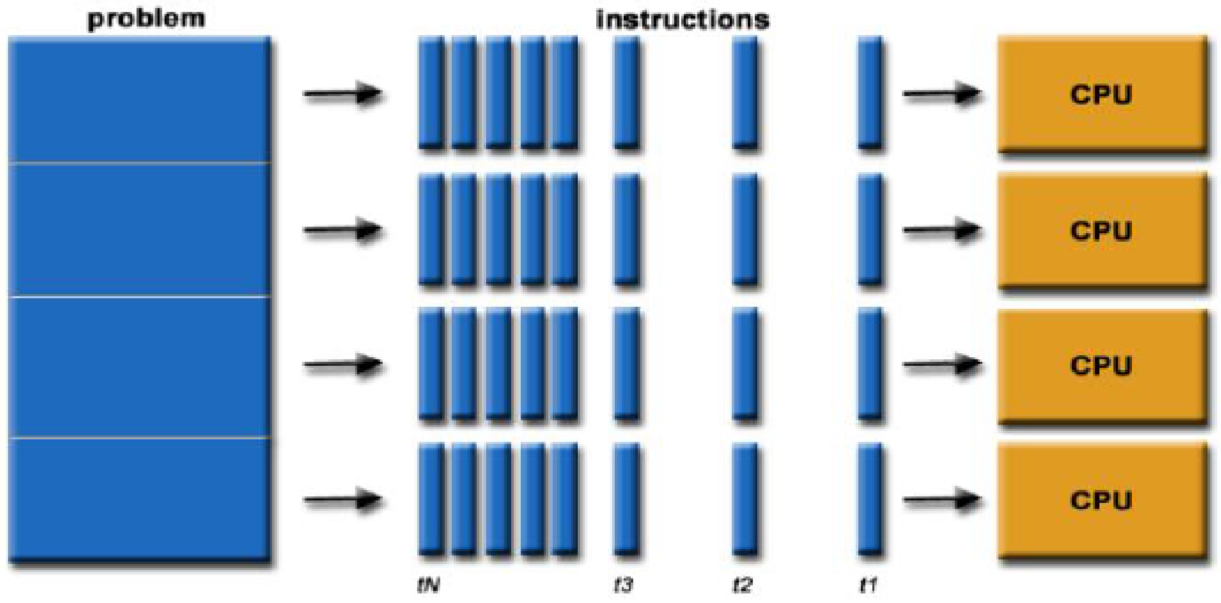
- ❑ To be run on a single computer having a single Central Processing Unit (CPU)
- ❑ A problem is broken into a discrete set of instructions
- ❑ Instructions are executed one after another
- ❑ Only one instruction can be executed at any moment in time



Parallel Computing:

In the simplest sense, parallel computing is the simultaneous use of multiple compute resources to solve a computational problem:

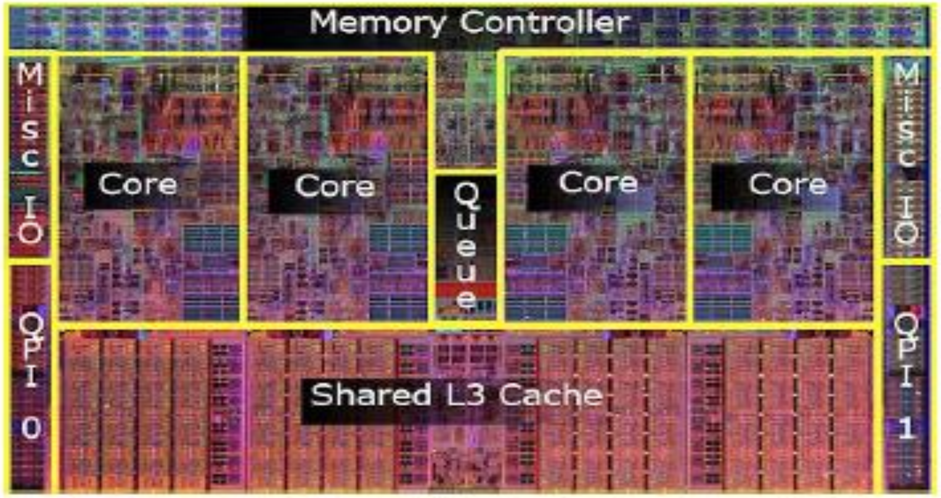
- ❑ To be run using multiple CPUs
- ❑ A problem is broken into discrete parts that can be solved concurrently
- ❑ Each part is further broken down to a series of instructions
- ❑ Instructions from each part execute simultaneously on different CPUs



Parallel Computers:

Virtually all stand-alone computers today are parallel from a hardware perspective:

- Multiple functional units (floating point, integer, GPU, etc.)
- Multiple execution units / cores
- Multiple hardware threads



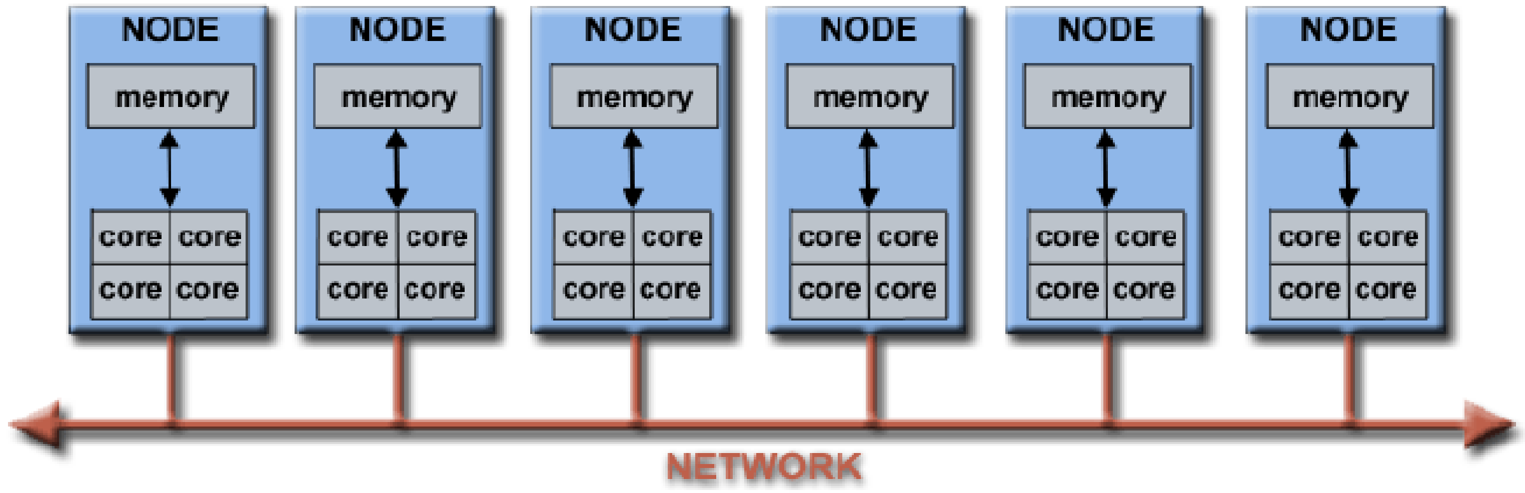
Intel Core i7 CPU and its major components

Image Credit: Intel

Parallel Computers:

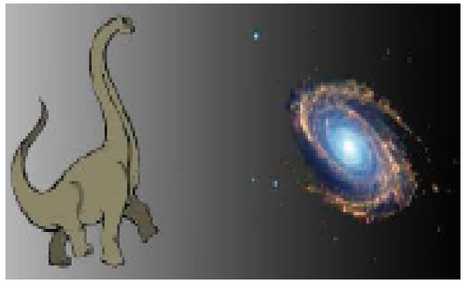
Networks connect multiple stand-alone computers (nodes) to create larger parallel computer clusters

- ❑ Each compute node is a multi-processor parallel computer in itself
- ❑ Multiple compute nodes are networked together with an InfiniBand network
- ❑ Special purpose nodes, also multi-processor, are used for other purposes



Why Use Parallel Programming?

Save time and/or money: In theory, throwing more resources at a task will shorten its time to completion, with potential cost savings. Parallel clusters can be built from cheap, commodity components.

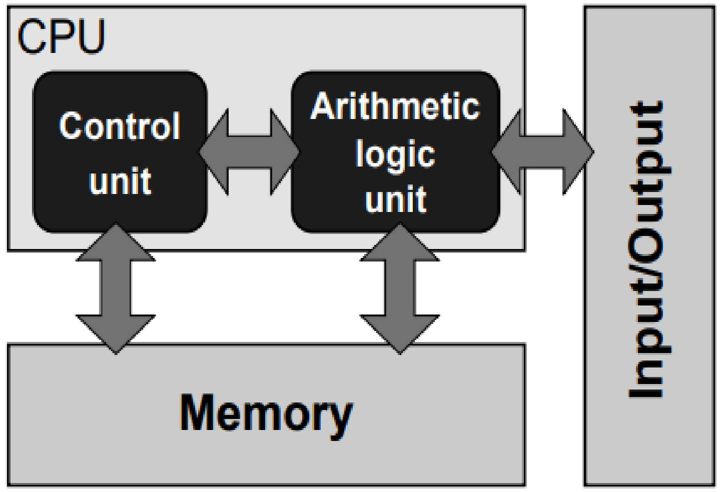


Solve larger problems: Many problems are so large and/or complex that it is impractical or impossible to solve them on a single computer, especially given limited computer memory.

Provide concurrency: A single compute resource can only do one thing at a time. Multiple computing resources can be doing many things simultaneously.

Stored-program computer architecture

- Instructions are numbers that are stored as data in memory
- Instructions are read and executed by a control unit
- Arithmetic logic unit is responsible for actual computation. It manipulates the data along with the instructions
- I/O facilities allows for the communication with the users
- Control unit and ALU along with appropriate interfaces to memory and I/O is called as CPU

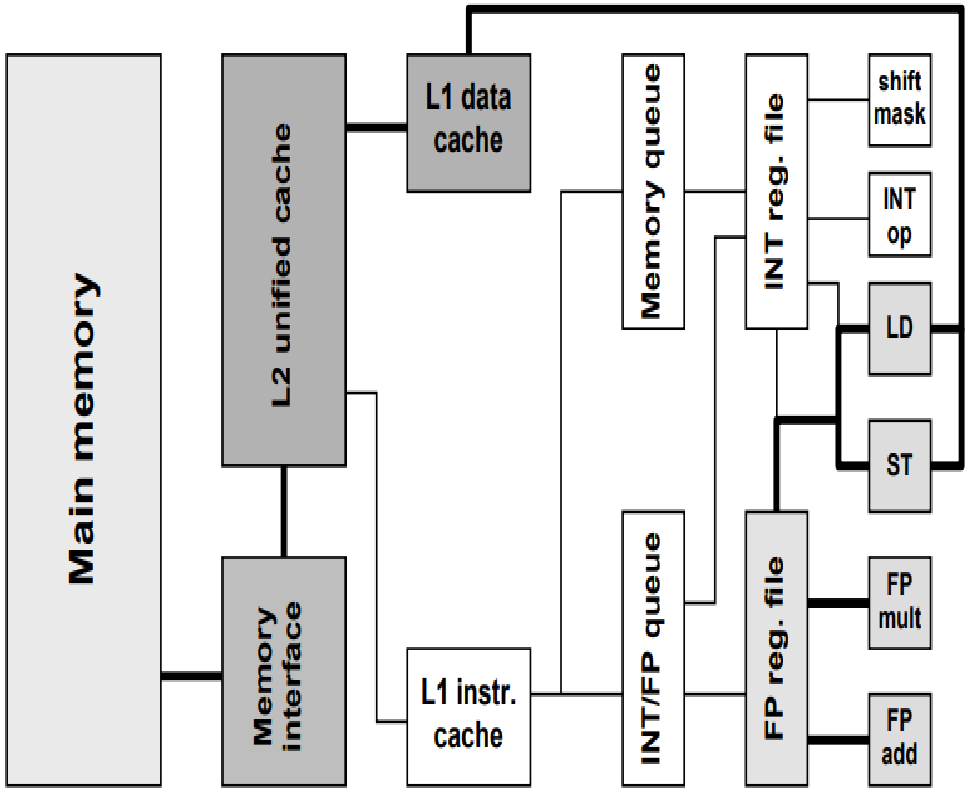


Stored-program computer architecture

- Programming a stored program computer requires us to modify the instructions stored in memory
- This is generally done by another program called as **compiler**
- **This is the general blueprint for all mainstream computers**
- Has few drawbacks:
  - Instructions and data must be continuously fed to the control and arithmetic units: This is known as von Neumann bottleneck
  - The architecture is sequential, processing a single instruction with (possibly) a single operand or a group of operands from memory

General purpose cache-based microprocessor architecture

- Arithmetic units are responsible for running the applications
  - FP (Floating point) and INT (Integer)
- CPU registers hold operands to be accessed by instructions
  - INT reg. file and FP reg. file
  - 16-128 such registers are generally available
- LD and ST units handle instructions that transfer data to and from registers
- Instructions are stored in queues to be executed
- Cache holds the data for re-use



Performance metrics

- Let Π be an arbitrary computational problem which is to be solved by a computer
  - Sequential algorithm performs one operation in each step
  - **Parallel algorithm** may perform multiple operations in a single step
- Let P be a parallel algorithm that has parallelism
- Let $C(p)$ be a parallel computer of the kind C which contains p processing units

Performance metrics

- The performance of P depends on both C and p
- We must consider two things:
  - Potential parallelism in P
  - Ability of $C(p)$ to execute, in parallel, multiple operations of P
- So, the performance of the algorithm P on the parallel computer $C(p)$ depends on $C(p)$'s capability to exploit P's potential parallelism
- The ***"performance"*** means the time required to execute P on $C(p)$
  - This is called the ***parallel execution time*** (or, ***parallel runtime***) of P on $C(p)$
  - Denoted by $\boldsymbol{T_{par}}$

Performance metrics

- **Speedup:** How many times is the parallel execution of P in $C(p)$ faster than the sequential execution of P

$$S = \frac{T_{seq}}{T_{par}}$$

- Parallel execution of P on $C(p)$ is S times faster than sequential execution

Performance metrics and enhancement

- **Efficiency:** Average contribution of each of the p processing units of $C(p)$ to the speedup

$$E = \frac{S}{p}$$

- Since, $T_{par} \leq T_{seq} \leq p.T_{par}$, Speedup is bounded by p and efficiency is bounded by 1

$$E \leq 1$$

- "For any C and p, the parallel execution of P on $C(p)$ can be at most p times faster than the execution of P on a single processor"

Oversimplified example:

p → fraction of program that can be parallelized

1 - p → fraction of program that cannot be parallelized

n → number of processors

Then the time of running the parallel program will be

$1 - p + p/n$ of the time for running the serial program

80% can be parallelized

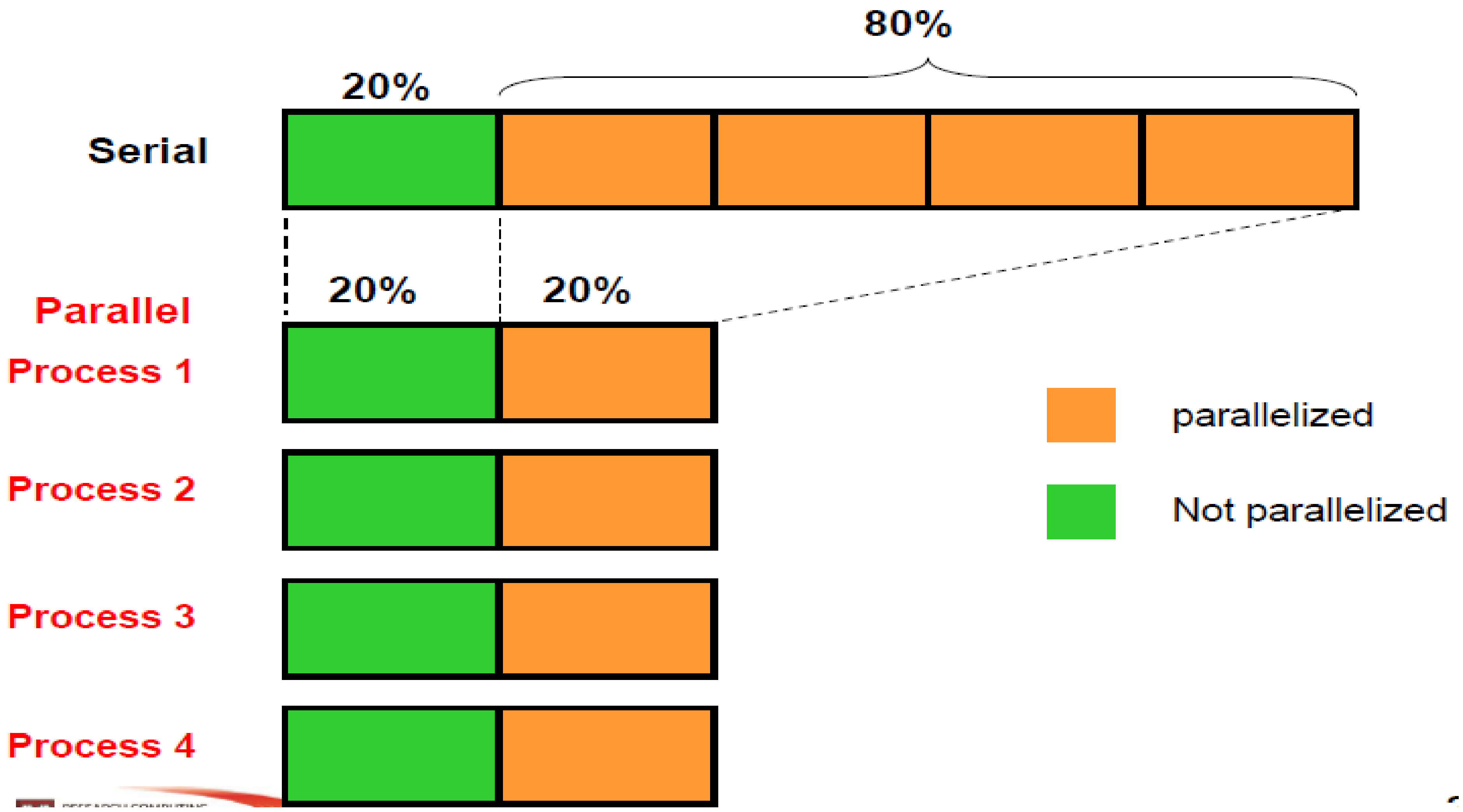
20 % cannot be parallelized

$n = 4$

$1 - 0.8 + 0.8 / 4 = 0.4$ i.e., 40% of the time for running the serial code

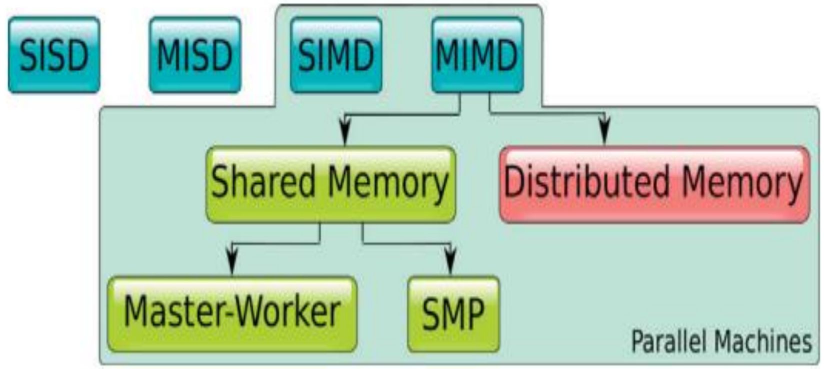
You get 2.5 speed up although you run on 4 cores since only 80% of your code can be parallelized (assuming that all parts in the code can complete in equal time)

Oversimplified example, cont'd:



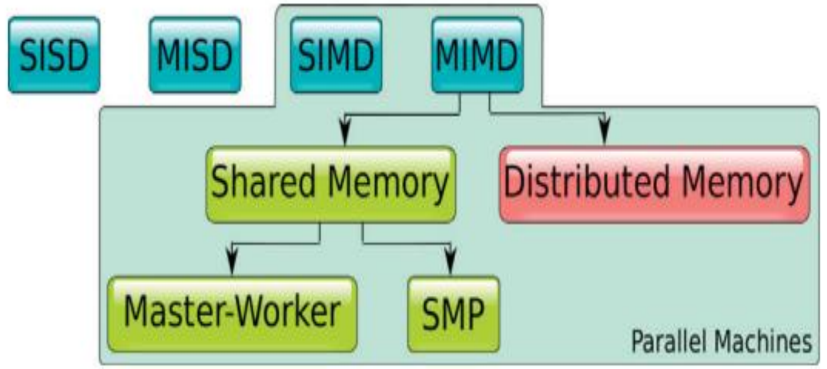
Taxonomy of parallel computers

- In 1966, Michael Flynn proposed a taxonomy of computer architectures
- Based on how many instructions and data items they can process concurrently
  - **SISD:** A sequential machine that can execute one instruction at a time on a single data item (Conventional non-parallel systems)
  - **SIMD:** A single instruction stream, either on a single processor (core) or on multiple compute elements, provides parallelism by operating on multiple data streams concurrently
    - (Examples are vector processors: GPUs)



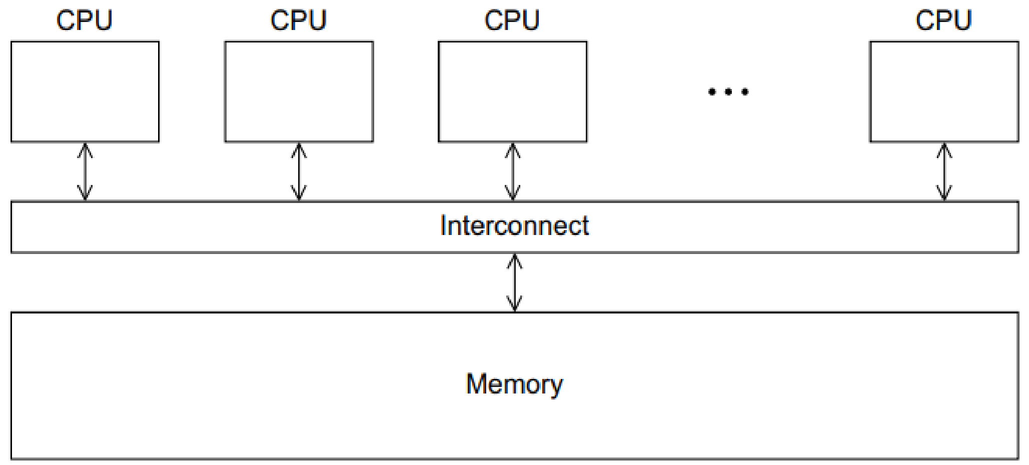
Taxonomy of parallel computers

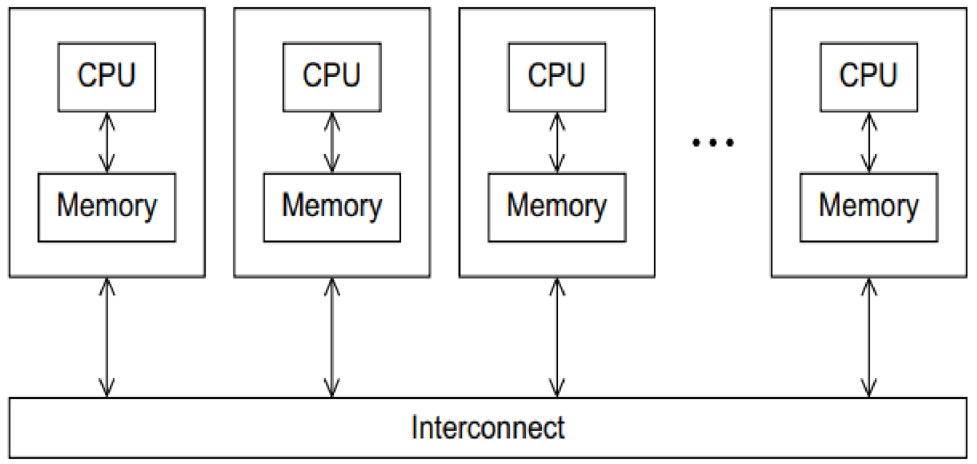
- In 1966, Michael Flynn proposed a taxonomy of computer architectures
- Based on how many instructions and data items they can process concurrently
  - **MISD:** Multiple instructions are applied on a single data
  - **MIMD:** Multiple instruction streams on multiple processors (cores) operate on different data items concurrently. The shared memory and distributed-memory parallel computers
    - Multiple instructions are applied on multiple data



Taxonomy of parallel computers

- MIMD can be further divided into two categories:
  - Shared-memory MIMD
  - Distributed-memory MIMD



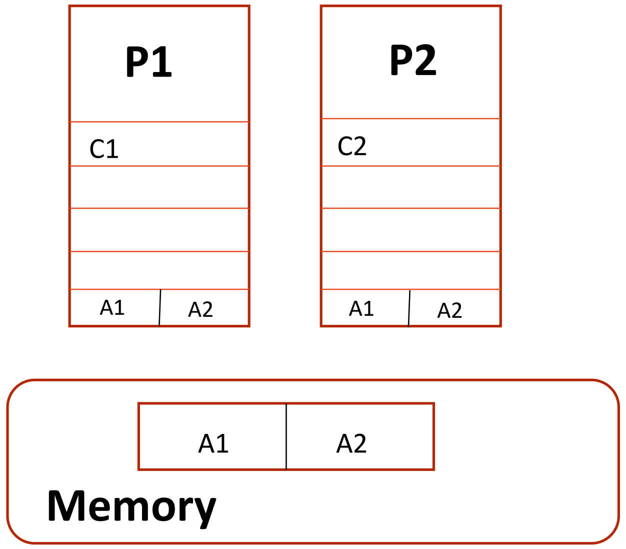


Shared-memory systems

- A number of CPUs work on a common, shared physical address space
- Two varieties of shared-memory systems
  - Uniform Memory Access
    - Latency and bandwidth are same for all processors and memory locations
    - Also called as Symmetric Multi-Processing (SMP)
  - On cache-coherent Nonuniform Memory Access
    - Memory is physically distributed but logically shared
    - The physical layout of such systems are similar to distributed systems
    - **The network logic makes the aggregated memory of the whole system appear as one single address space**
- In both these cases, copy of same information may reside in different caches, probably in modified state

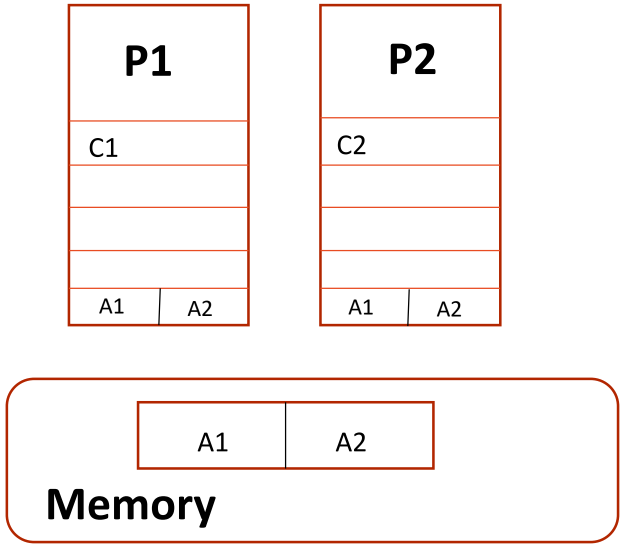
Cache-coherence

- Copies of the same cache line could potentially reside in several CPU caches
- One of those gets modified and evicted to memory
- The other caches' contents reflect outdated data
- **Cache coherence** protocols ensure a consistent view of memory under all circumstances



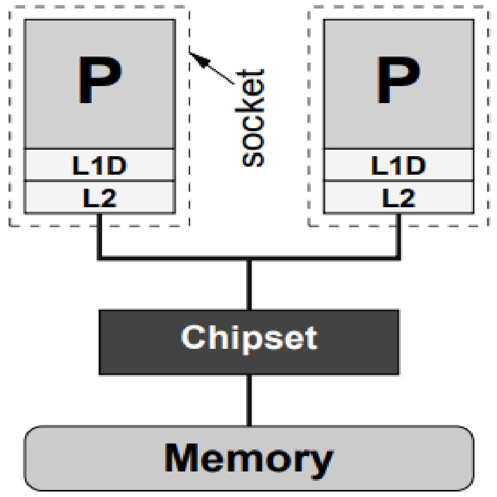
Cache-coherence (MESI protocol)

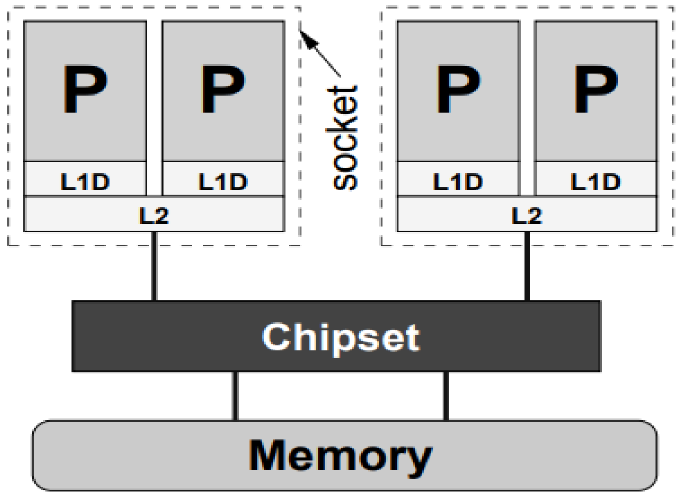
- Under control of cache coherence logic discrepancy can be avoided
- **M modified:** The cache line has been modified in this cache, and it resides in no other cache than this one. Only upon eviction will memory reflect the most current state.
- **E exclusive:** The cache line has been read from memory but not (yet) modified. However, it resides in no other cache.
- **S shared:** The cache line has been read from memory but not (yet) modified. There may be other copies in other caches of the machine.
- **I invalid:** The cache line does not reflect any sensible data. Under normal circumstances this happens if the cache line was in the shared state and another processor has requested exclusive ownership.



Uniform Memory Access (UMA)

- Simplest implementation of a UMA system is a dual-core processor, in which two CPUs on one chip share a single path to memory

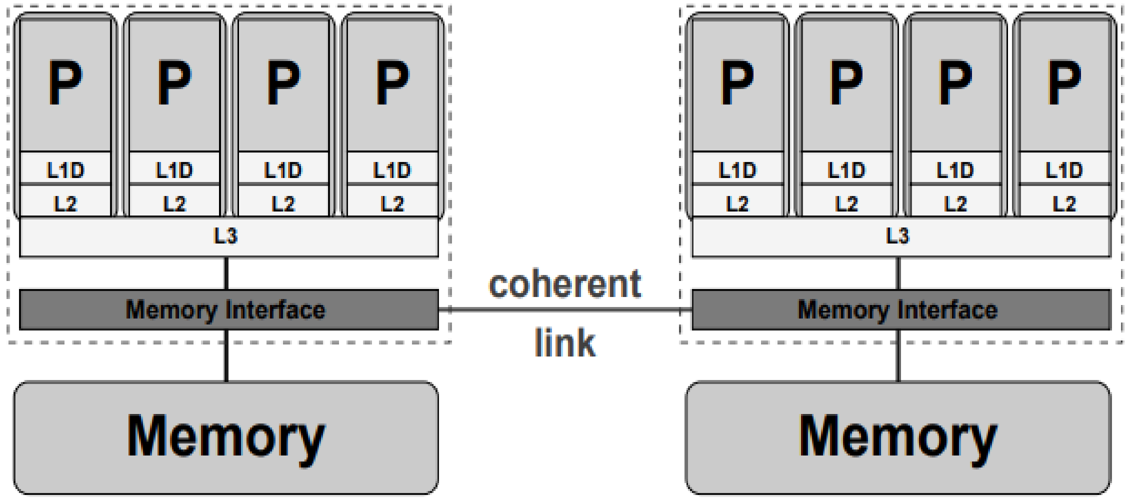




- Problem of UMA systems is that bandwidth bottlenecks are bound to occur

ccNUMA

- Locality domain (LD) is a set of processor cores together with locally connected memory
- Multiple LDs are linked via a coherent interconnect
  - Provides transparent access from any processor to any other processor’s memory

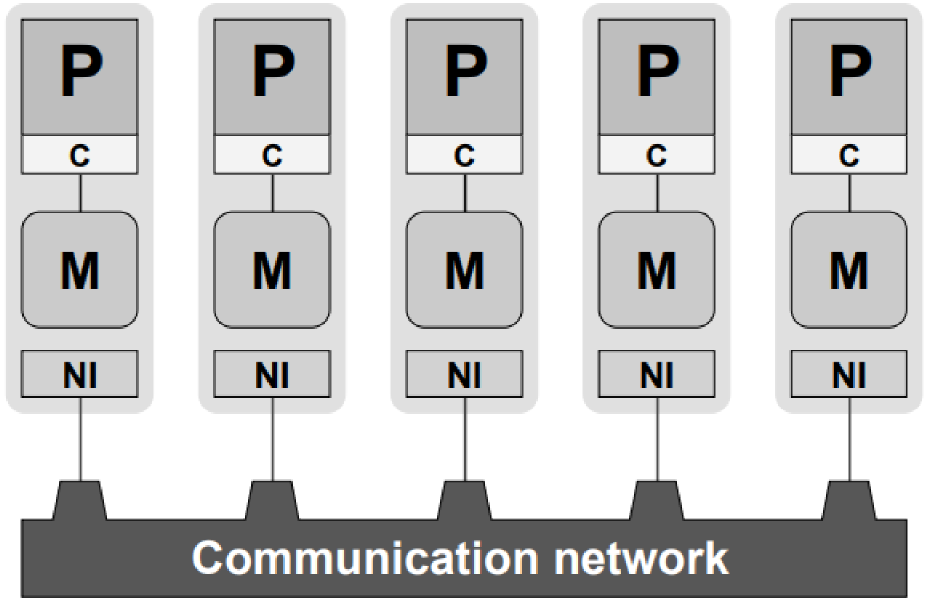


Shared-memory systems

- Advantages:
  - Global address space provides a user-friendly programming perspective to memory
  - Fast and uniform data sharing due to proximity of memory to CPUs
- Disadvantages
  - Lack of scalability between memory and CPUs. Adding more CPUs increases traffic on the shared memory-CPU path
  - Programmers responsibility for correct access to global memory

Distributed-memory systems

- Each processor P is connected to exclusive local memory
- No other CPU has direct access to it
- Each node comprises at least one network interface
- A serial process runs on each CPU that can communicate with other processes on other CPUs by means of the network

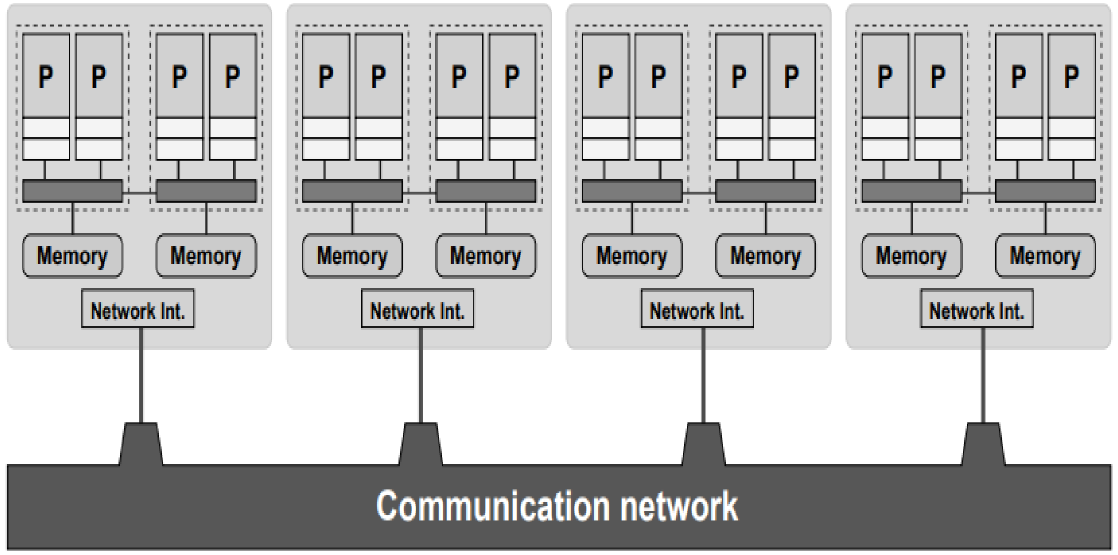


Distributed-memory systems

- Advantages:
  - Memory is scalable with number of CPUs
  - Each CPU can rapidly access its own memory without overhead incurred with trying to maintain global cache coherence
- Disadvantages
  - Programmer is responsible for many of the details associated with data communication between processors
  - It is usually difficult to map existing data structures to this memory organization, based on global memory

Hybrid systems

- Large-scale parallel computers are neither of the purely shared-memory nor of the purely distributed-memory
- Shared-memory building blocks connected via a fast network
- Advantages:
  - Increased scalability
- Disadvantages:
  - Increased programming complexity


P
Memory
Network Int.
Communication network