CS3006 Parallel and Distributed Computing

FALL 2022

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

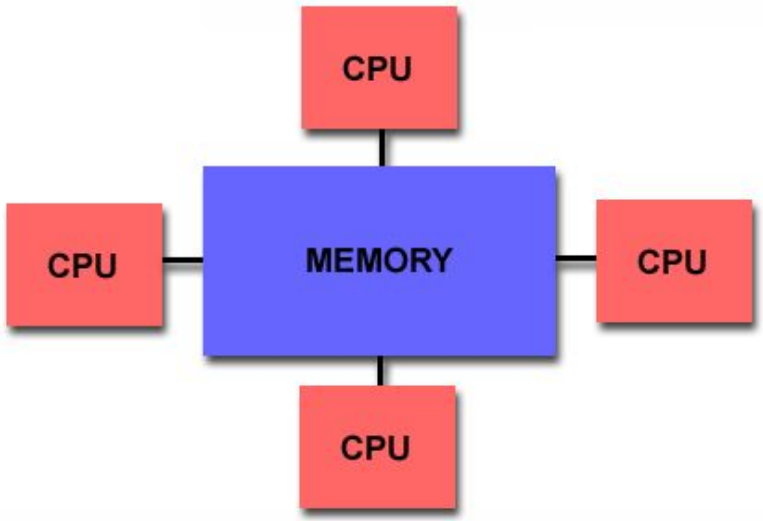
Parallel Computer Memory Architectures

Memory architectures

- Shared Memory
- Distributed Memory
- Hybrid Distributed-Shared Memory

Shared Memory

- Shared memory parallel computers vary widely, but generally have in common the ability for all processors to access all memory as global address space.



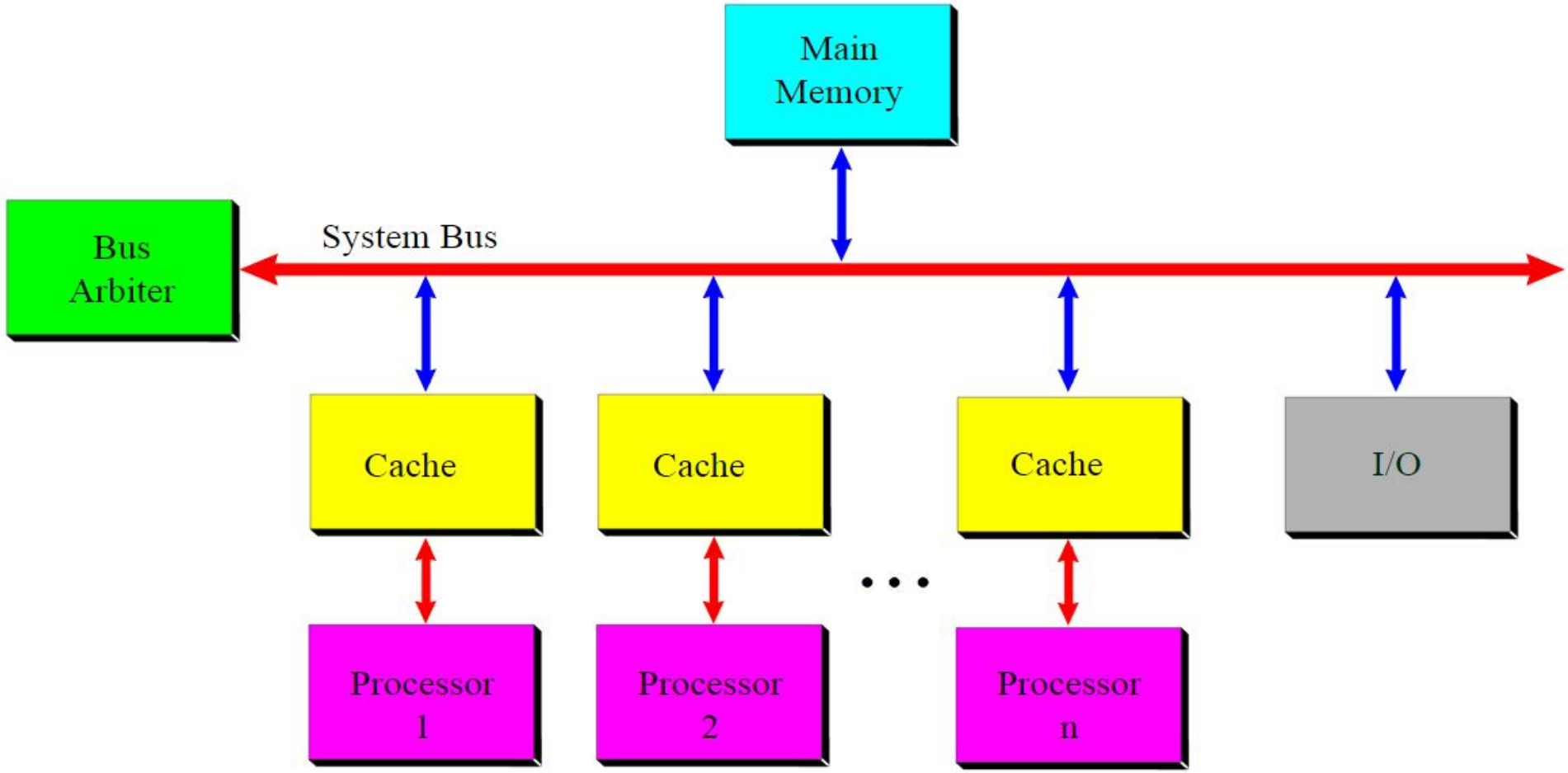
- Multiple processors can operate independently but share the same memory resources.
- Changes in a memory location effected by one processor are visible to all other processors.
- Shared memory machines can be divided into two main classes based upon memory access times: ***UMA*** and ***NUMA***.

Shared Memory : UMA vs. NUMA

- Uniform Memory Access (UMA):
  - Most commonly represented today by Symmetric Multiprocessor (SMP) machines
  - Identical processors with equal access and access times to memory
  - Sometimes called CC-UMA - Cache Coherent UMA.

- Non-Uniform Memory Access (NUMA):
  - Often made by physically linking two or more SMPs
  - One SMP can directly access memory of another SMP
  - Not all processors have equal access time to all memories

SMP - Symmetric Multiprocessor System



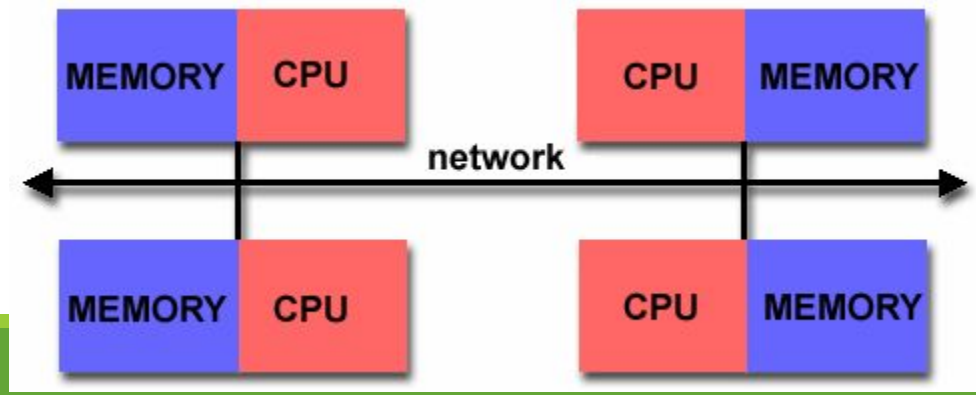
Shared Memory: Pro and Con

- Advantages
  - Global address space provides a user-friendly programming perspective to memory
  - Data sharing between tasks is both fast and uniform due to the proximity of memory to CPUs

- Disadvantages:
  - Primary disadvantage is the lack of scalability between memory and CPUs. Adding more CPUs can geometrically increases traffic on the shared memory-CPU path, and for cache coherent systems, geometrically increase traffic associated with cache/memory management.
  - Programmer responsibility for synchronization constructs that insure "correct" access of global memory.
  - Expense: it becomes increasingly difficult and expensive to design and produce shared memory machines with ever increasing numbers of processors.

Distributed Memory

- Like shared memory systems, distributed memory systems vary widely but share a common characteristic. Distributed memory systems require a communication network to connect inter-processor memory.
- Processors have their own local memory. Memory addresses in one processor do not map to another processor, so there is no concept of global address space across all processors.
- Because each processor has its own local memory, it operates independently. Changes it makes to its local memory have no effect on the memory of other processors. Hence, the concept of cache coherency does not apply.
- When a processor needs access to data in another processor, it is usually the task of the programmer to explicitly define how and when data is communicated. Synchronization between tasks is likewise the programmer's responsibility.
- The network "fabric" used for data transfer varies widely, though it can can be as simple as Ethernet.



Distributed Memory: Pro and Con

- Advantages
  - Memory is scalable with number of processors. Increase the number of processors and the size of memory increases proportionately.
  - Each processor can rapidly access its own memory without interference and without the overhead incurred with trying to maintain cache coherency.
  - Cost effectiveness: can use commodity, off-the-shelf processors and networking.
- Disadvantages
  - The programmer is responsible for many of the details associated with data communication between processors.
  - It may be difficult to map existing data structures, based on global memory, to this memory organization.
  - Non-uniform memory access (NUMA) times

Hybrid Distributed-Shared Memory

Comparison of Shared and Distributed Memory Architectures			
Architecture	CC-UMA	CC-NUMA	Distributed
Examples	SMPs Sun Vexx DEC/Compaq SGI Challenge IBM POWER3	Bull NovaScale SGI Origin Sequent HP Exemplar DEC/Compaq IBM POWER4 (MCM)	Cray T3E Maspar IBM SP2 IBM BlueGene
Communications	MPI Threads OpenMP shmem	MPI Threads OpenMP shmem	MPI
Scalability	to 10s of processors	to 100s of processors	to 1000s of processors
Draw Backs	Memory-CPU bandwidth	Memory-CPU bandwidth Non-uniform access times	System administration Programming is hard to develop and maintain
Software Availability	many 1000s ISVs	many 1000s ISVs	100s ISVs

Chapter2: Parallel Computing Platforms

Implicit Parallelism: Trends in Microprocessor Architectures

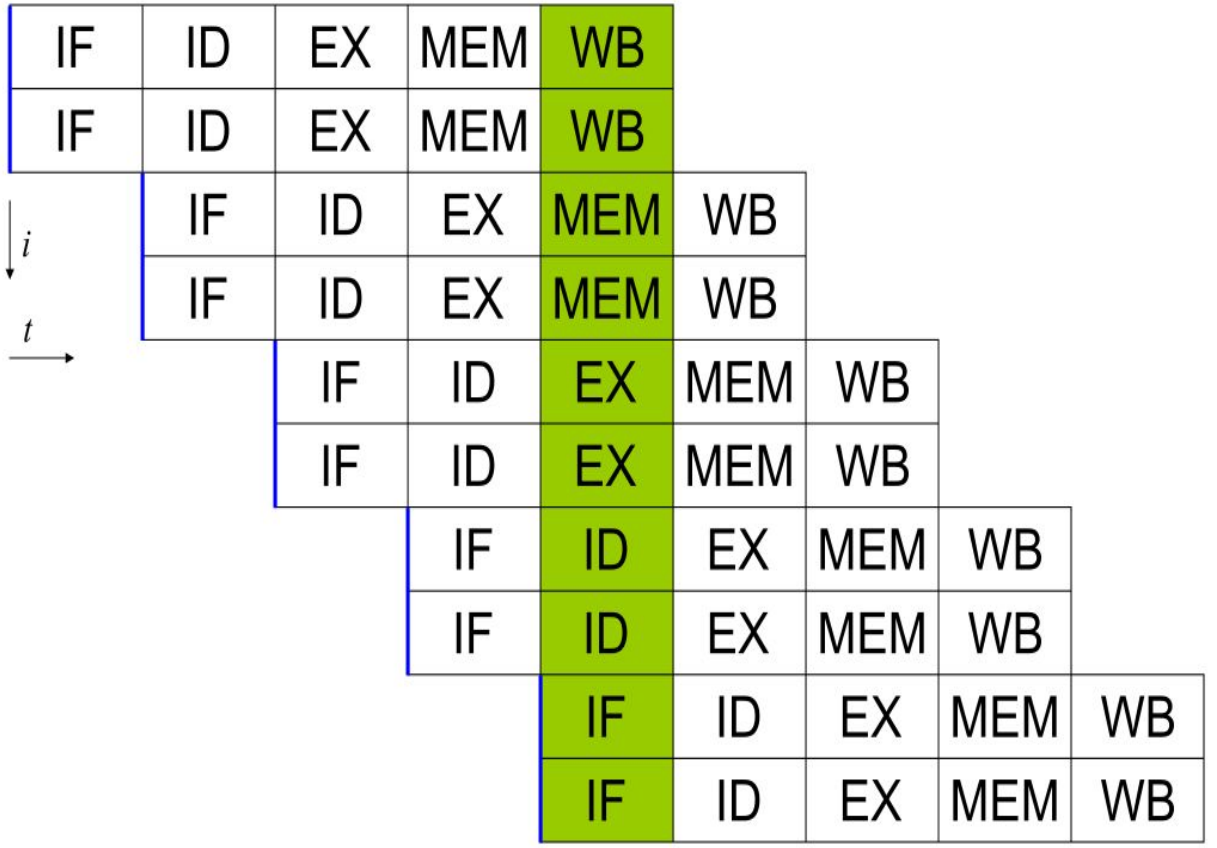
- Traditionally a sequential computer consists of:
  - processor, memory, and datapath
  - all present bottleneck to the performance

- Increments in **clock speed** are severely diluted by the limitations of **memory technology**

Pipelining and Superscalar Execution

- **Pipelining**: overlapping various stages in instruction execution
  - *fetch, schedule, decode, operand fetch, execute, store*, among others
  - Pentium 4, which operates at 2.0 GHz, has a **20 stage** pipeline.
  - Speed of a single pipeline is ultimately limited by the largest **atomic task** in the pipeline.
  - In typical instruction traces, every fifth to sixth instruction is a **branch instruction**.
    - This requires need effective techniques for **predicting branch** destinations so that pipelines can be speculatively filled, a **mis-prediction** will cost a lot.
- Use **multiple pipelines**. The ability of a processor to issue multiple instructions in the same cycle is referred to as **superscalar execution**
  - During each clock cycle, multiple instructions are piped into the processor in parallel.
  - These instructions are executed on multiple **functional units**.

- A superscalar processor can execute more than one instruction during a clock cycle by simultaneously dispatching multiple instructions to different **execution units** on the processor



```
1. load R1, @1000
2. load R2, @1008
3. add R1, @1004
4. add R2, @100C
5. add R1, R2
6. store R1, @2000
```

(i)

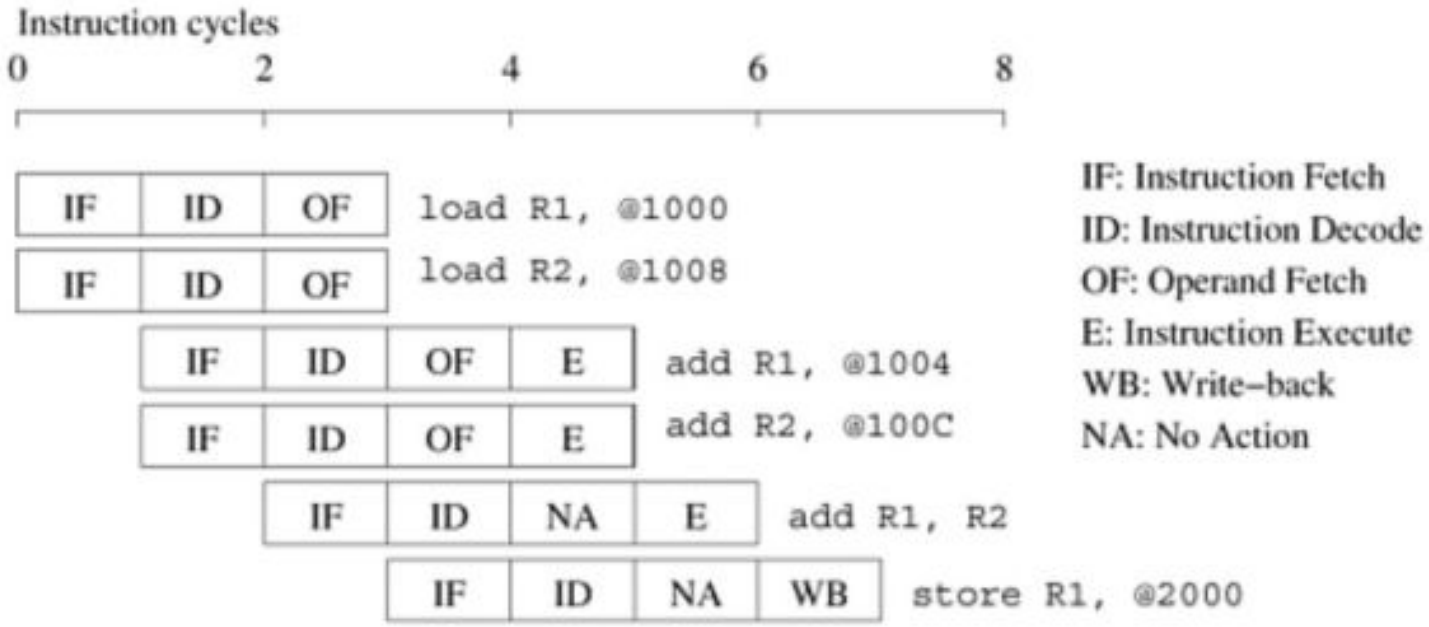
```
1. load R1, @1000
2. add R1, @1004
3. add R1, @1008
4. add R1, @100C
5. store R1, @2000
```

(ii)

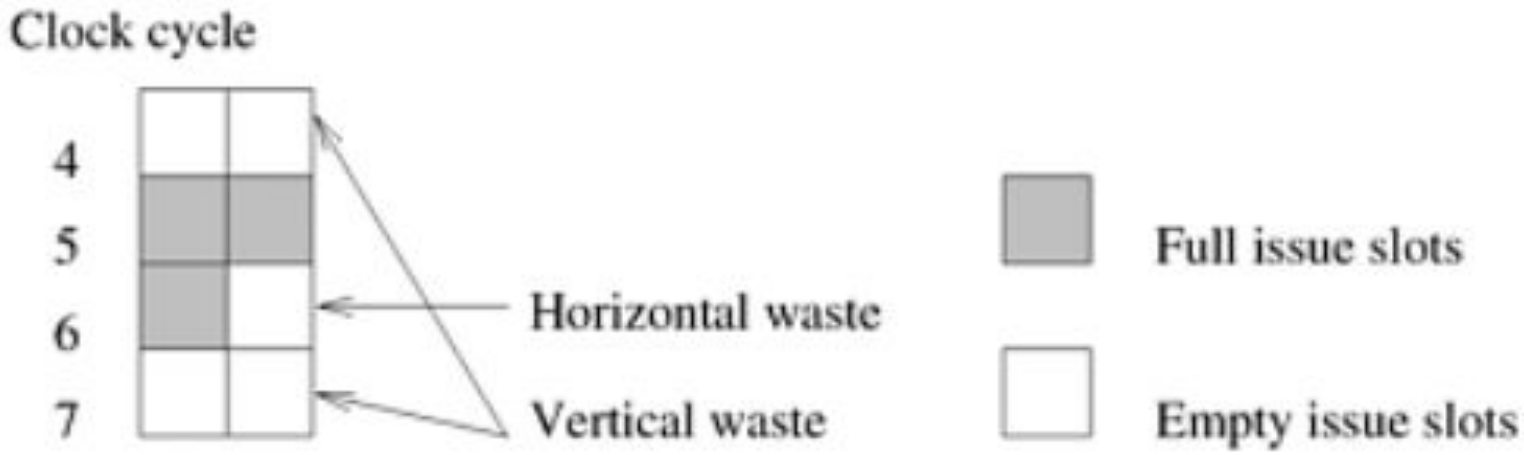
```
1. load R1, @1000
2. add R1, @1004
3. load R2, @1008
4. add R2, @100C
5. add R1, R2
6. store R1, @2000
```

(iii)

(a) Three different code fragments for adding a list of four numbers.



(b) Execution schedule for code fragment (i) above.


Clock cycle
4
5
6
7
Horizontal waste
Vertical waste
Full issue slots
Empty issue slots

- A number of issues needs to be resolved with superscalar execution
  - **Data Dependency**
  - **Resource Dependency**
  - **Branch/Procedural Dependency**
- The processor needs the ability to issue instructions ***out-of-order*** to accomplish desired reordering.
  - The parallelism available in ***in-order*** issue of instructions can be highly limited.
- Most current microprocessors are capable of **out-of-order** issue and completion.
  - This model, also referred to as ***dynamic instruction issue***, exploits maximum instruction level parallelism.

Very Long Instruction Word (VLIW) Processors

- The parallelism extracted by superscalar processors is often limited by the instruction look-ahead.

- Instructions that can be executed concurrently are **packed** into **groups** and **parceled** off to the processor as a single long instruction word to be executed on multiple functional units at the same time.
  - The **compiler** has a larger context from which to select instructions

- The performance of VLIW processors is very sensitive to the compilers‘ ability to detect data and resource dependencies and read and write hazards, and to schedule instructions for maximum parallelism.

- The traditional means to improve performance in processors include dividing instructions into substeps so the instructions can be executed partly at the same time (***pipelining***),
- dispatching individual instructions to be executed independently, in different parts of the processor (***superscalar architectures***), and even executing instructions in an order different from the program (***out-of-order execution***).
- These methods all complicate hardware (larger circuits, higher cost and energy use) because the processor must make all of the decisions internally for these methods to work.
- In contrast, the VLIW method depends on the programs providing all the decisions regarding which instructions to execute simultaneously and how to resolve conflicts. As a practical matter, this means that the **compiler** becomes far more complex, but the hardware is simpler than in many other means of parallelism.

Limitation of Memory System Performance

- The effective performance of a program on a computer relies not just on the speed of the processor but also on the ability of the memory system to feed data to the processor.

- **Latency**: request to receiving time for a memory word.
  - Rate at which data can be pumped from the memory to the processor determines the ***bandwidth*** of the memory system.

Example

- a processor operating at **1 GHz** (1 ns clock) connected to a DRAM with a latency of **100** ns (no caches)

- The processor is capable of executing **four instructions** in each cycle of 1 ns (**4 GFLOPS**).
  - Assuming, the processor has two multiply-add units

- Since the memory latency is equal to **100 cycles** and block size is one word (4-Bytes), every time a memory request is made, the processor must wait **100 cycles** before it can process the data

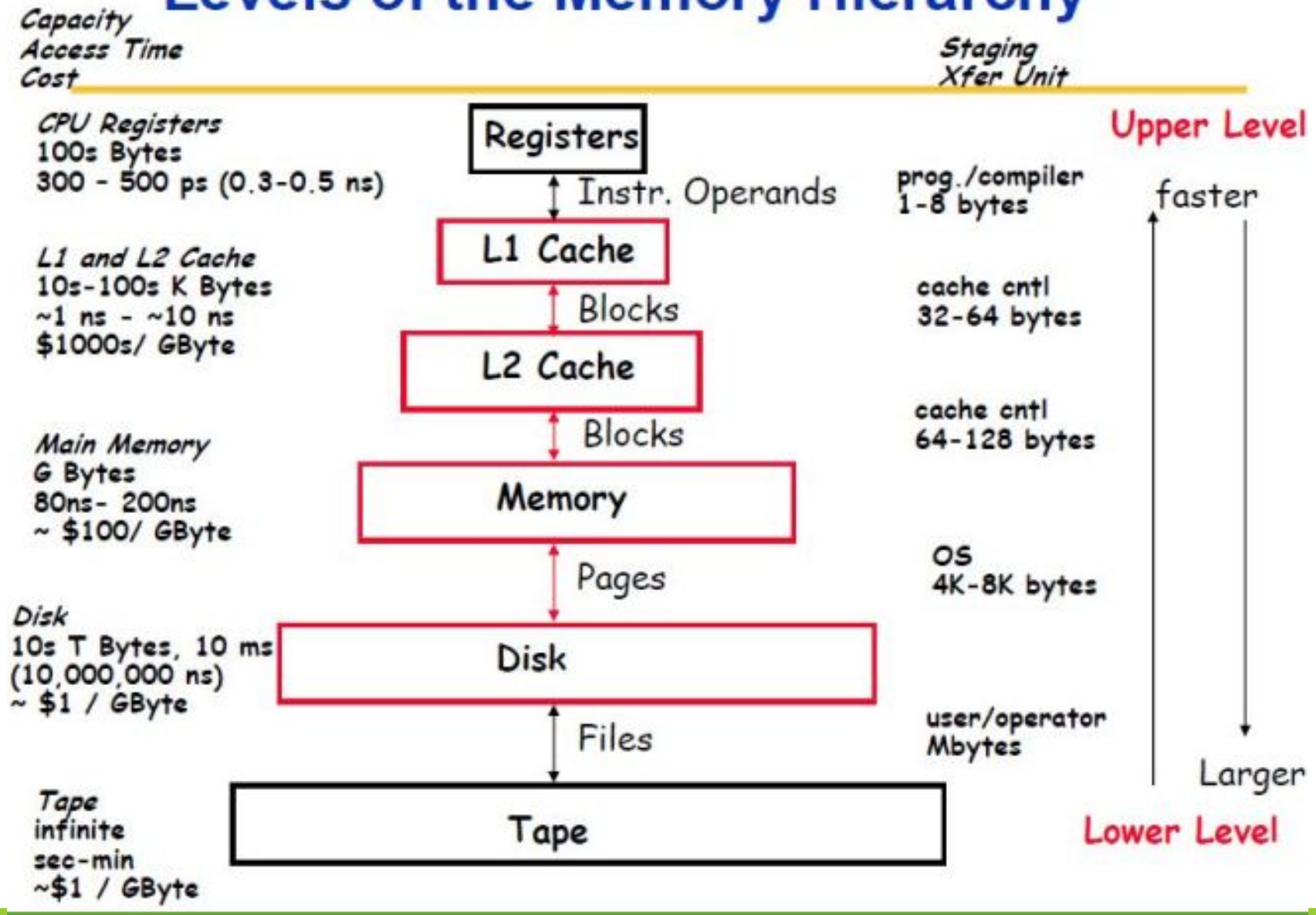
Example 2.2 (Cont'd)

- Consider the problem of computing the dotproduct of two vectors on such a platform.
  - A dot-product computation performs one multiply-add on a single pair of vector elements, i.e., each floating point operation requires one data fetch.
- It is easy to see that the peak speed of this computation is limited to one floating point operation every 100 ns (or a speed of 10 MFLOPS) a very small fraction of the peak processor rating
- This example highlights the need for effective memory system performance in achieving high computation rates.

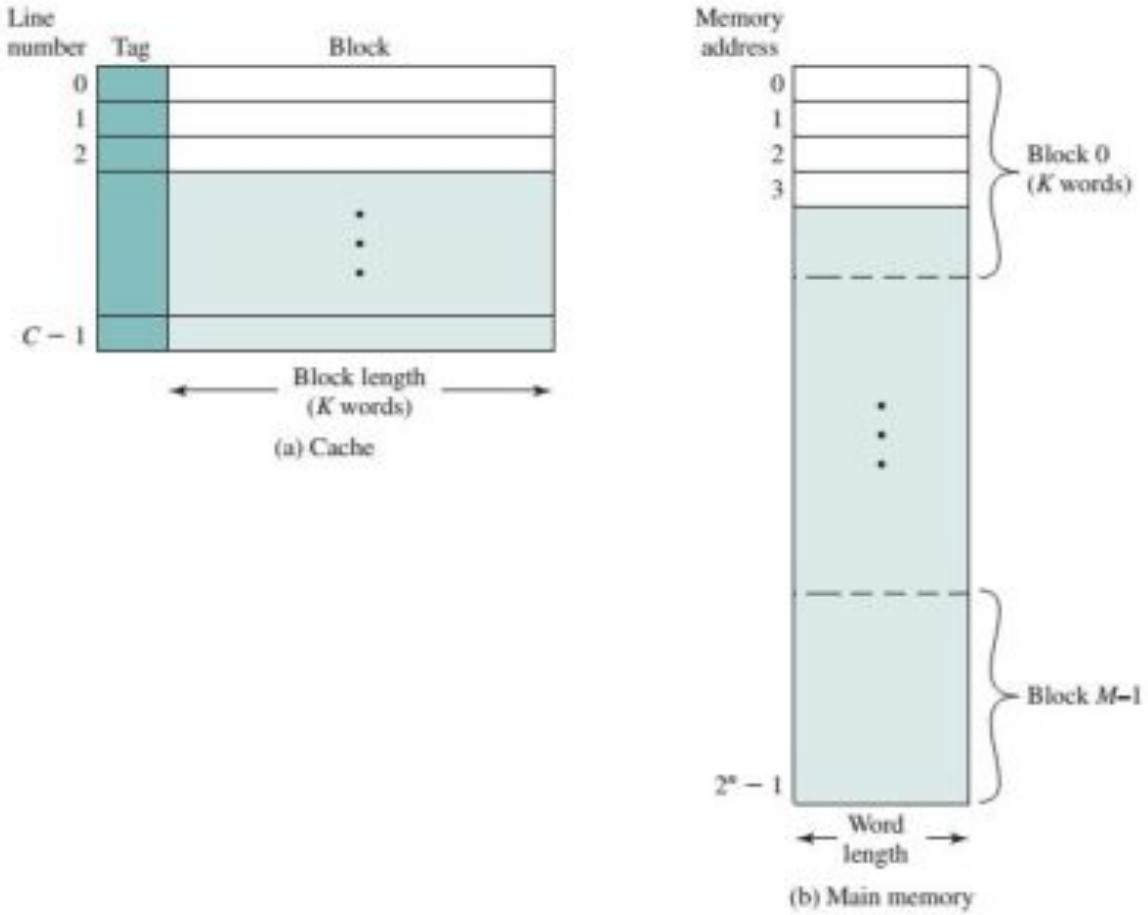
Improving Effective Memory Latency Using Caches

- **Cache**: a *low-latency high-bandwidth* storage between the processor and the DRAM.
  - The data needed by the processor is first fetched into the cache. All subsequent accesses to data items residing in the cache are serviced by the cache.
  - Thus, in principle, if a piece of data is repeatedly used, the effective latency of this memory system can be reduced by the cache.
  - The fraction of data references satisfied by the cache is called the cache ***hit ratio*** of the computation on the system.
  - **Data reuse** is critical for cache performance because if each data item is used only once, it would still have to be fetched once per use from the DRAM

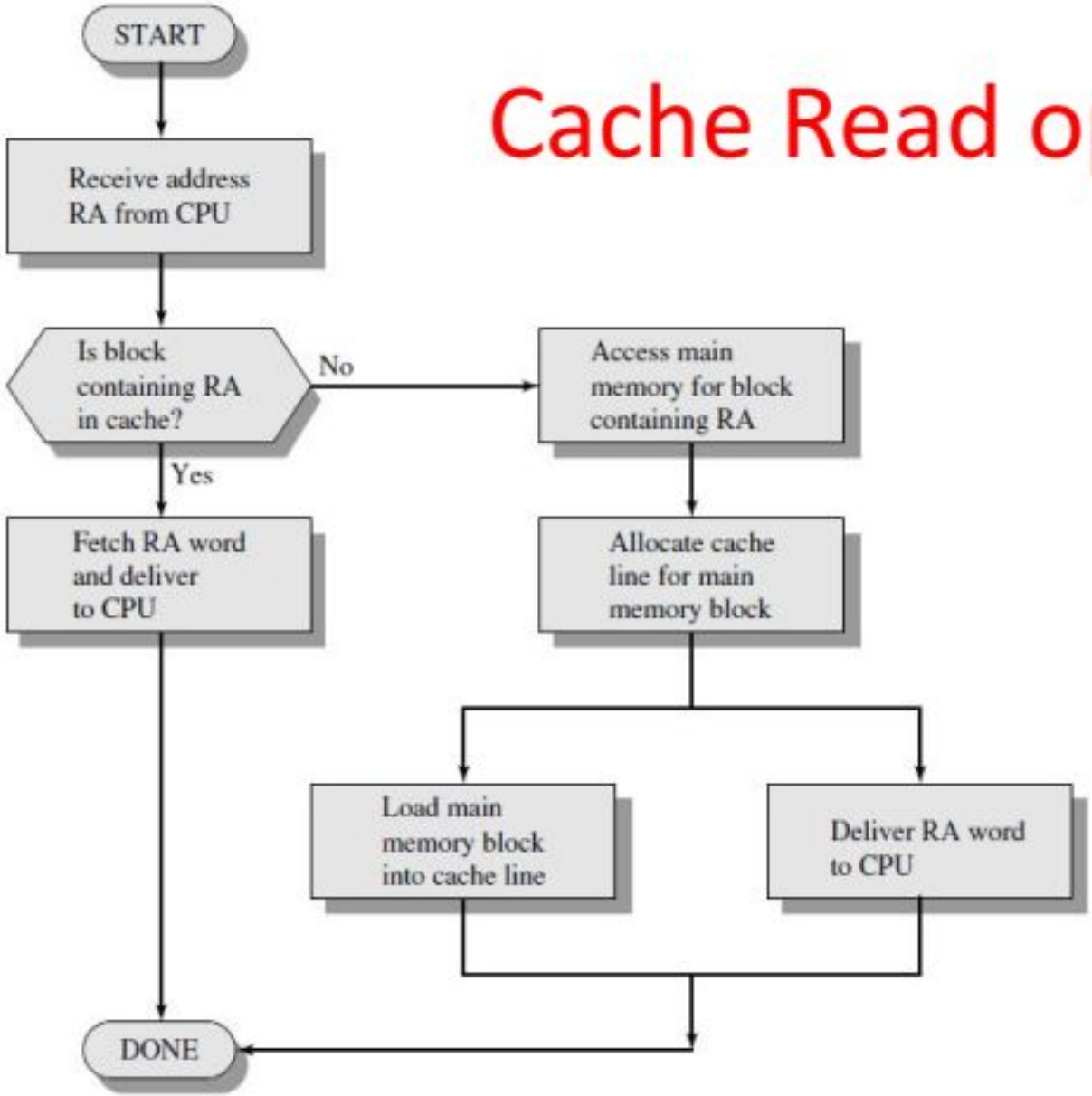
Levels of the Memory Hierarchy



- The cache consists of ***m*** blocks, called **lines**.
  - In referring to the basic unit of the cache, the term *line* is used, rather than the term block.
- **Line Size**: The length of a line.
  - Line size may be as less as 32 bits (a word).
- The number of lines is considerably less than the number of main memory blocks



Cache Read operation



Example 2.3 Impact of caches on memory system performance

- As in the previous example, consider a **1 GHz** processor with a 100 ns latency DRAM.
- In this case, we introduce a cache of size **32 KB** with a latency of 1 ns or one cycle (typically on the processor itself).
- We use this setup to multiply two matrices *A* and *B* of dimensions 32 x 32.
  - We have carefully chosen these numbers so that the cache is large enough to store matrices *A* and *B*, as well as the result matrix *C*.
- Fetching the two matrices into the cache corresponds to fetching 2K words, which takes approximately **200 µs**.

Example 2.3 Impact of caches on memory system performance

- We know from elementary algorithms that multiplying two *n* x *n* matrices takes $\mathbf{2n^3}$ operations.
- For our problem, this corresponds to **64K** operations, which can be performed in 16K cycles (or 16 µs) at four instructions per cycle.
- The total time for the computation is therefore approximately the sum of time for load/store operations and the time for the computation itself, i.e., **200+16 µs**. This corresponds to a peak computation rate of **64K/216** or **303 MFLOPS**.
- Note that this is a thirty-fold improvement over the previous example, although it is still less than 10% of the peak processor performance. We see in this example that by placing a small cache memory, we are able to improve processor utilization considerably.

Impact of Memory Bandwidth

- One commonly used technique to improve memory bandwidth is to increase the size of the memory blocks.
  - Consider again a memory system with a **single cycle cache** and 100 cycle latency DRAM with the processor operating at 1 GHz
  - If the block size is one word, the processor takes **100 cycles** to fetch each word.
  - If the block size is **increased to four words**, i.e., the processor can fetch a **four-word** cache line every 100 cycles.
  - increasing the block size from one to four words did not change the latency of the memory system. However, it increased the **bandwidth four-fold.**

- Increased bandwidth results in higher peak computation rates. The data layouts are assumed to be such that consecutive data words in memory were used by successive instructions.
- In other words, if we take a computation-centric view, there is a ***spatial locality*** of memory access.
- An example of such an access pattern is in reading a dense matrix column-wise when the matrix has been stored in a row-major fashion in memory.

```
1 for (i = 0; i < 1000; i++)
2   column_sum[i] = 0.0;
3       for (j = 0; j < 1000; j++)
4           column_sum[i] += b[j][i];
```

- **Temporal** and **Spatial Locality** of Data:
  - **Temporal locality** refers to the reuse of specific data, and/or resources, within a relatively small time duration.
  - **Spatial locality** (also termed *data locality*) refers to the use of data elements within relatively close storage locations. Sequential locality.
    - a special case of spatial locality, occurs when data elements are arranged and accessed linearly, such as, traversing the elements in a one-dimensional array.

Cache Coherence

- The presence of caches on processors also raises the issue of multiple copies of a single memory word being manipulated by two or more processors at the same time

- Cache coherence is intended to manage such conflicts by maintaining a coherent view of the data values in multiple caches

Cache Coherence (Cont'd)

- **Write Propagation**
  - Changes to the data in any cache must be propagated to other copies (of that cache line) in the peer caches

- **Transaction Serialization**
  - Reads/Writes to a single memory location must be seen by all processors in the same order.

Alternate Approaches for Hiding Memory Latency

- Imagine sitting at your computer browsing the web during peak network traffic hours.

- The lack of response from your browser can be alleviated using one of three simple approaches:
  - **Anticipate** which pages we are going to browse ahead of time and issue requests for them in advance: **Prefetching**
  - we open multiple browsers and access different pages in each browser, thus while we are waiting for one page to load, we could be reading others: **multi-threading**
  - we access a whole bunch of pages in one go – remunerating the latency across various accesses: **Spatial Locality**

Prefetching for Latency Hiding

- **Prefetching** is a technique used by computer processors to boost execution performance by fetching instructions or data from their original storage in slower memory to a faster local memory **before it is actually needed** (hence the term 'prefetch')

- In a typical program, a data item is loaded and used by a processor in a small time window. If the load results in a **cache miss**, then the use **stalls**.

- A simple solution to this problem is to **advance the load operation** so that even if there is a cache miss, the data is likely to have arrived by the time it is used.

- However, if the data item has been overwritten between *load* and *use*, a **fresh load** is issued.

Multithreading for Latency Hiding

- If a thread gets a lot of cache misses, the other threads can continue taking advantage of the unused computing resources, which may lead to faster overall execution, as these resources would have been idle if only a single thread were executed.

- Also, if a thread cannot use all the computing resources of the CPU (because instructions depend on each other's result), running another thread may prevent those resources from becoming idle.

Multithreading for Latency Hiding

- Multiple threads, however can interfere with each other when sharing hardware resources such as caches.

- As a result, execution times of a single thread are not improved and can be degraded, even when only one thread is executing, due to lower frequencies or additional pipeline stages that are necessary to accommodate thread-switching hardware.

Multithreading for Latency Hiding

```
1   for(i=0;i<n;i++)
2       c[i] = dot_product(get_row(a, i), b);
```

- Because each dot-product is independent of the other, and therefore represents a concurrent unit of execution, we may rewrite the above code as:

```
1  for(i=0;i<n;i++)
2      c[i] = create_thread(dot_product, get_row(a, i), b);
```

- Multithreaded processors are capable of maintaining the context of a number of threads of computation with outstanding requests (memory accesses, I/O, or communication requests) and execute them as the requests are satisfied.

- Machines such relying on multithreaded processors that can switch the context of execution in every cycle .
  - they are able to hide latency effectively, provided there is enough concurrency (threads) to keep the processor from idling.

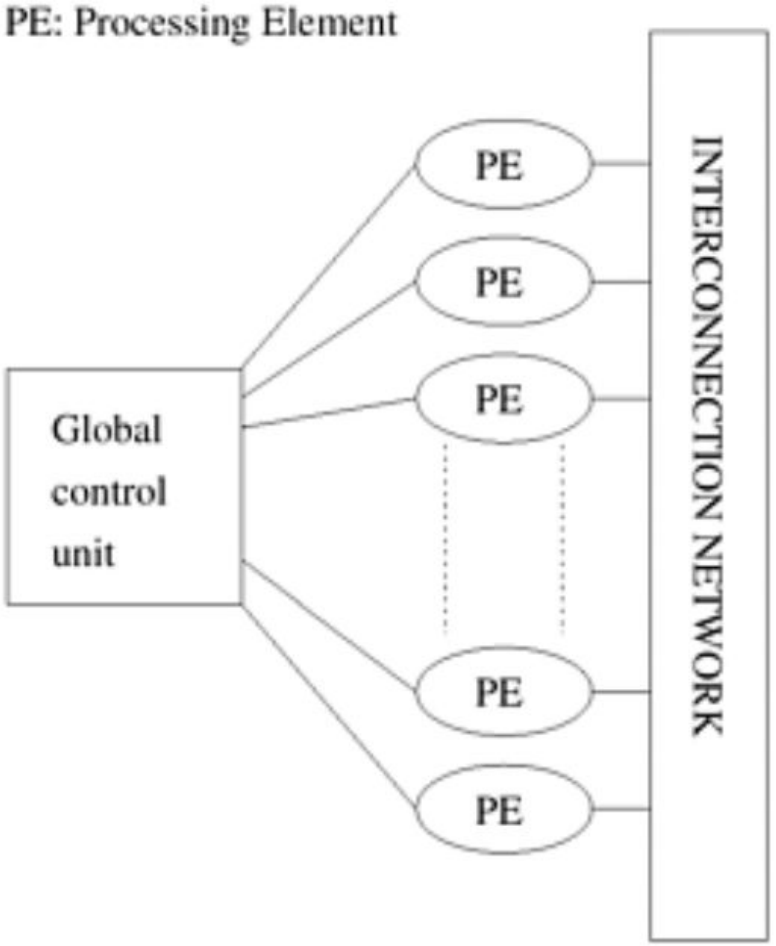
2.3 Dichotomy of Parallel Computing Platforms

Control Structure of Parallel Platforms

- Parallel tasks can be specified at various levels of **granularity**.
  - At one extreme, each program in a set of programs can be viewed as one parallel task.
  - At the other extreme, individual instructions within a program can be viewed as parallel tasks.
  - E.g. Parallelism for single instruction:
  -

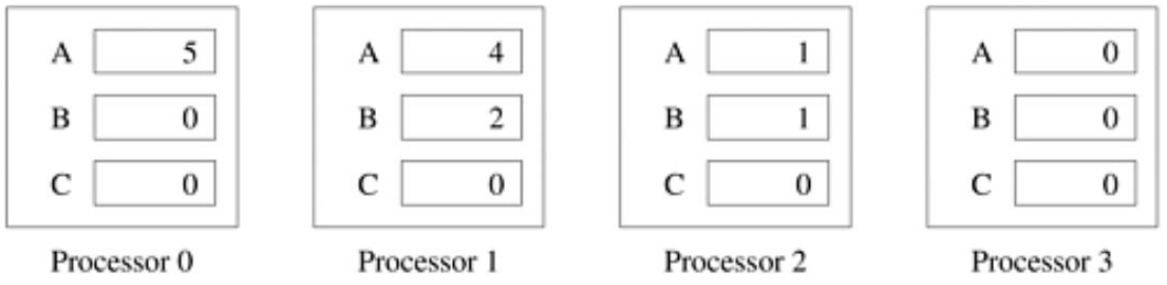
```
1   for (i = 0; i < 1000; i++)
2           c[i] = a[i] + b[i];
```

- Processing units in parallel computers either operate under the **centralized control** of a **single control unit** or work independently.
  - In **SIMD**, a single control unit dispatches instructions to each processing unit.
  - In an SIMD parallel computer, the same instruction is executed synchronously by all processing units.
  - While the SIMD concept works well for structured computations on **parallel data structures** such as arrays, often it is necessary to selectively turn off operations on certain data items, this may limit the utilization of processing elements.

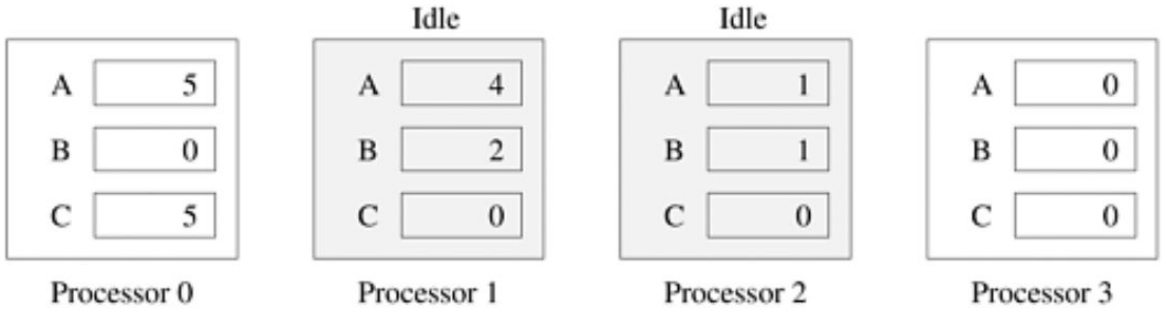


```
if (B == 0)
        C = A;
else
        C = A/B;
```

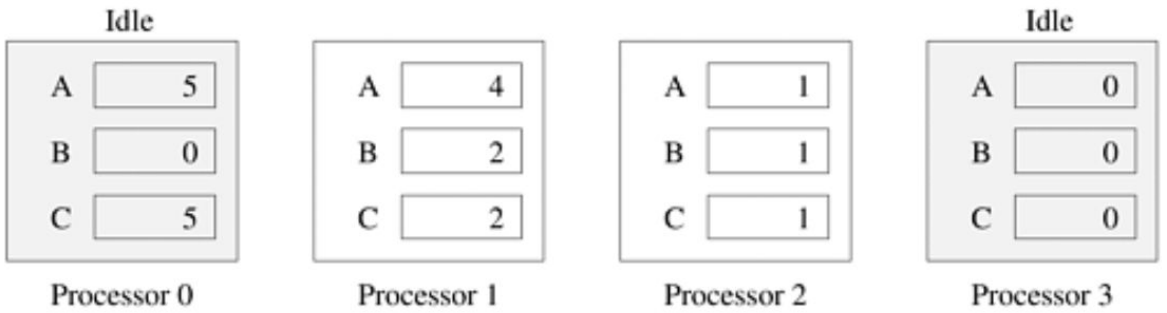
(a)



Initial values



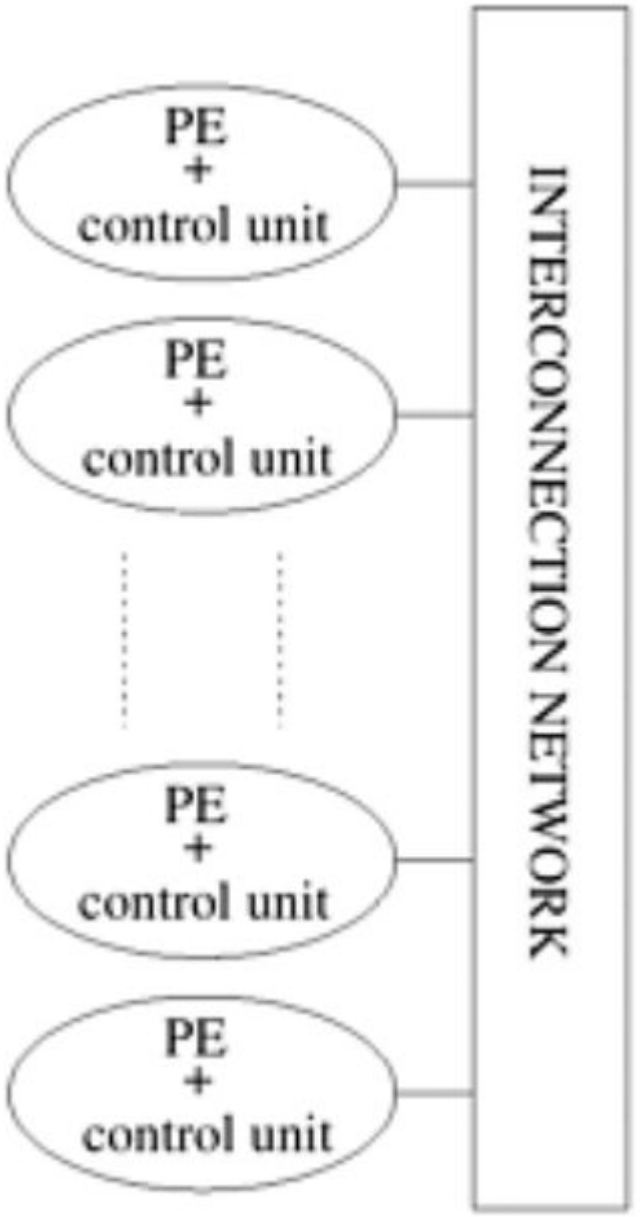
Step 1



Step 2

(b)

- In contrast to SIMD architectures, **MIMD** are the computers in which each processing element is capable of executing a **different program** independent of the other processing elements.
  - **SIMD** : less hardware than MIMD computers because they have only one global control unit, and less memory because only one copy of the program needs to be stored



Communication Model of Parallel Platforms

- There are two primary forms of data exchange between parallel tasks:

1. **Shared-Address-Space Platforms**
2. **Message-Passing Platforms**

Shared-Address-Space Platforms

- common data space is accessible to all processors .
  - If the time taken by a processor to access any memory word in the system (global or local) is identical, the platform is classified as a ***uniform memory access*** (UMA) multicomputer.
  - On the other hand, if the time taken to access certain memory words is longer than others, the platform is called a ***non-uniform memory access*** (NUMA) multicomputer

- The presence of a global memory space makes programming such platforms much easier.
  - All read-only interactions are invisible to the programmer, as they are coded no differently than in a serial program. This greatly eases the burden of writing parallel programs. Read/write interactions are, however, harder to program than the read-only interactions, as these operations require mutual exclusion for concurrent accesses

- The presence of caches on processors also raises the issue of **multiple copies of a single memory** word being manipulated by two or more processors at the same time.
- Supporting a shared address-space in this context involves two major tasks:
  - providing an **address translation** mechanism that locates a memory word in the system,
  - ensuring that concurrent operations on multiple copies of the same memory word have well-defined semantics. ***Cache coherence*** mechanism.

Message-Passing Platforms

- Logically, a message-passing platform consists of *p* processing nodes, each with its own exclusive address space.
  - Interactions between processes running on different nodes must be accomplished using messages (data, work, and to synchronize actions among the processes), hence the name ***message passing***.
  - The basic operations in this programming paradigm are ***send*** and ***receive***

Interconnection Networks for Parallel Computers

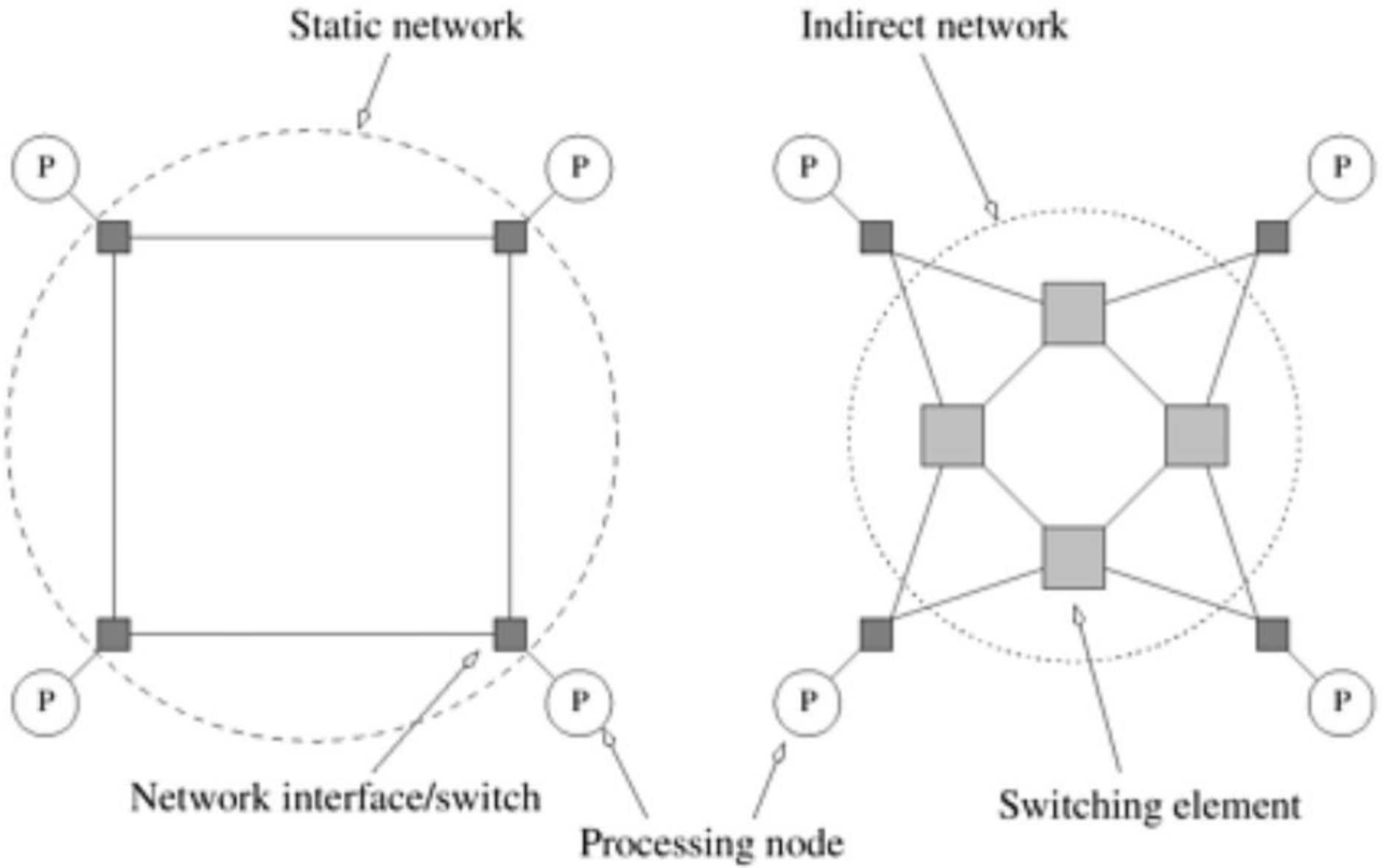
- Interconnection networks provide mechanisms for data transfer between processing nodes or between processors and memory modules.

- The interconnect plays a decisive role in the performance of both distributed and shared-memory systems: even if the processors and memory have virtually unlimited performance, a slow interconnect will seriously degrade the overall performance.
  - Typical interconnection networks are built using **links** and switches.
  - A link corresponds to **physical media** such as a set of wires or fibers capable of carrying information.

Interconnection Networks for Parallel Computers

- **Static Interconnection Networks** consist of point-to-point communication links among processing nodes and are also referred to as ***direct*** networks.

- **Dynamic Interconnection Networks**, on the other hand, are built using **switches** and communication links. Communication links are connected to one another dynamically by the switches. Dynamic networks are also referred to as ***indirect*** networks.
  - A single switch in an interconnection network consists of a set of input ports and a set of output ports. The total number of ports on a switch is also called the ***degree*** of the switch.
  - Switches may also provide support for internal **buffering** (when the requested output port is busy), **routing** (to alleviate congestion on the network), and **multicast** (same output on multiple ports)


Static network
Indirect network
P
P
P
P
P
P
P
P
Network interface/switch
Processing node
Switching element

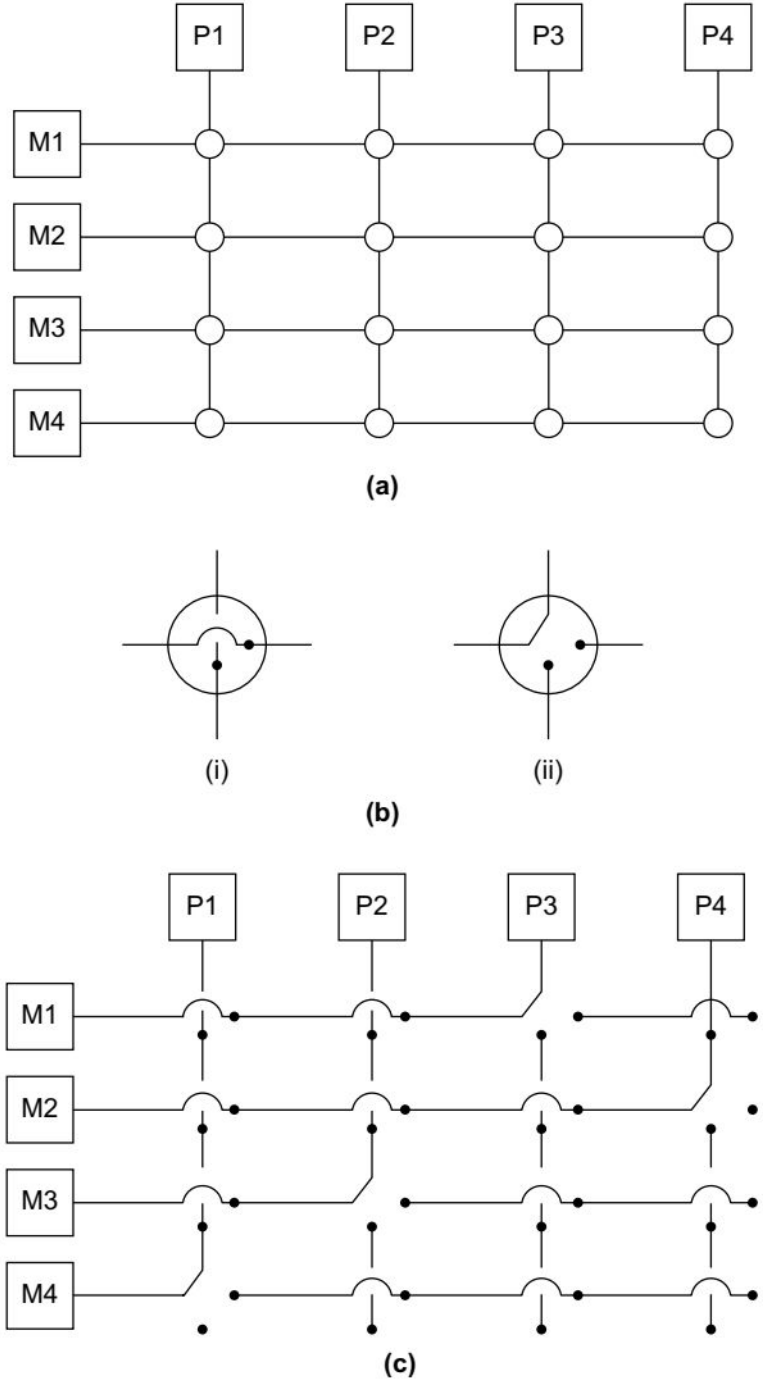
Network Topologies

Bus-Based Networks

- A bus-based network is perhaps the simplest network consisting of a **shared medium** that is common to all the nodes.
  - since the communication wires are shared, as the number of devices connected to the bus increases, the likelihood that there will be **contention** for use of the bus increases.

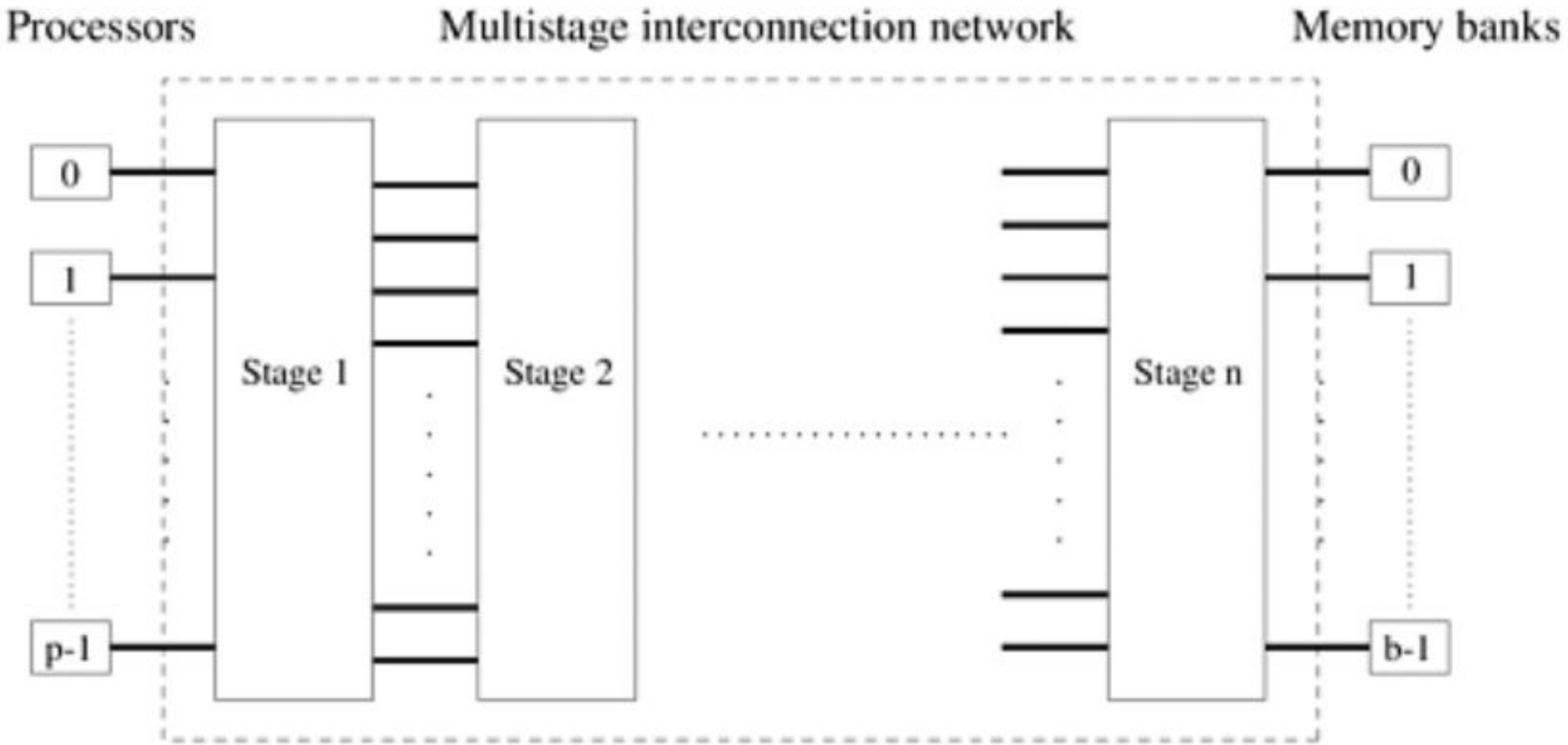
- The distance between any two nodes in the network is constant.
- Buses are also ideal for **broadcasting** information among nodes.
  - Since the transmission medium is shared, there is little overhead associated with broadcast compared to point-to-point message transfer.
- The bounded bandwidth of a bus places limitations on the overall performance of the network as the number of nodes increases. Typical bus based machines are limited to *dozens of nodes*.
- The demands on bus bandwidth can be reduced by making use of the property that in typical programs, a majority of the data accessed is *local* to the node. For such programs, it is possible to provide a **cache** for each node. **Private data** is **cached** at the node and only remote data is accessed through the bus.

A Crossbar Network



(a) A crossbar switch connecting four processors (P_i) and four memory modules (M_j); (b) configuration of internal switches in a crossbar; (c) simultaneous memory accesses by the processors

- The crossbar interconnection network is scalable in terms of performance but unscalable in terms of **cost**.
- Conversely, the shared bus network is scalable in terms of cost but unscalable in terms of **performance**.
- An intermediate class of networks called ***multistage interconnection networks*** lies between these two extremes


Processors
Multistage interconnection network
Memory banks
0
1
p-1
Stage 1
Stage 2
Stage n
0
1
b-1

Message Passing Costs in Parallel Computers

- The total time to transfer a message over a network comprises of the following:
  - ***Startup time*** **(t_s)**: Time spent at sending and receiving nodes (executing the routing algorithm, programming routers, etc.).
  - ***Per-hop time*** **(t_h)**: This time is a function of number of hops and includes factors such as switch latencies, network delays, etc.
  - ***Per-word transfer time*** **(t_w)**: This time includes all overheads that are determined by the length of the message. This includes bandwidth of links, error checking and correction, etc.