Conventional Computer Architecture Abstraction

Conventional = Sequential or Single Processor

- **Conventional computer architecture has two aspects:**

Single Processor

1 **<u>The definition of critical abstraction layers:</u>** Interfaces

- **<u>The user/system boundary:</u>**
  - **What is done in user space and what support is provided by the operating system to user programs.**
- **<u>The hardware/software boundary:</u>**
  - **Instruction Set Architecture (ISA).**

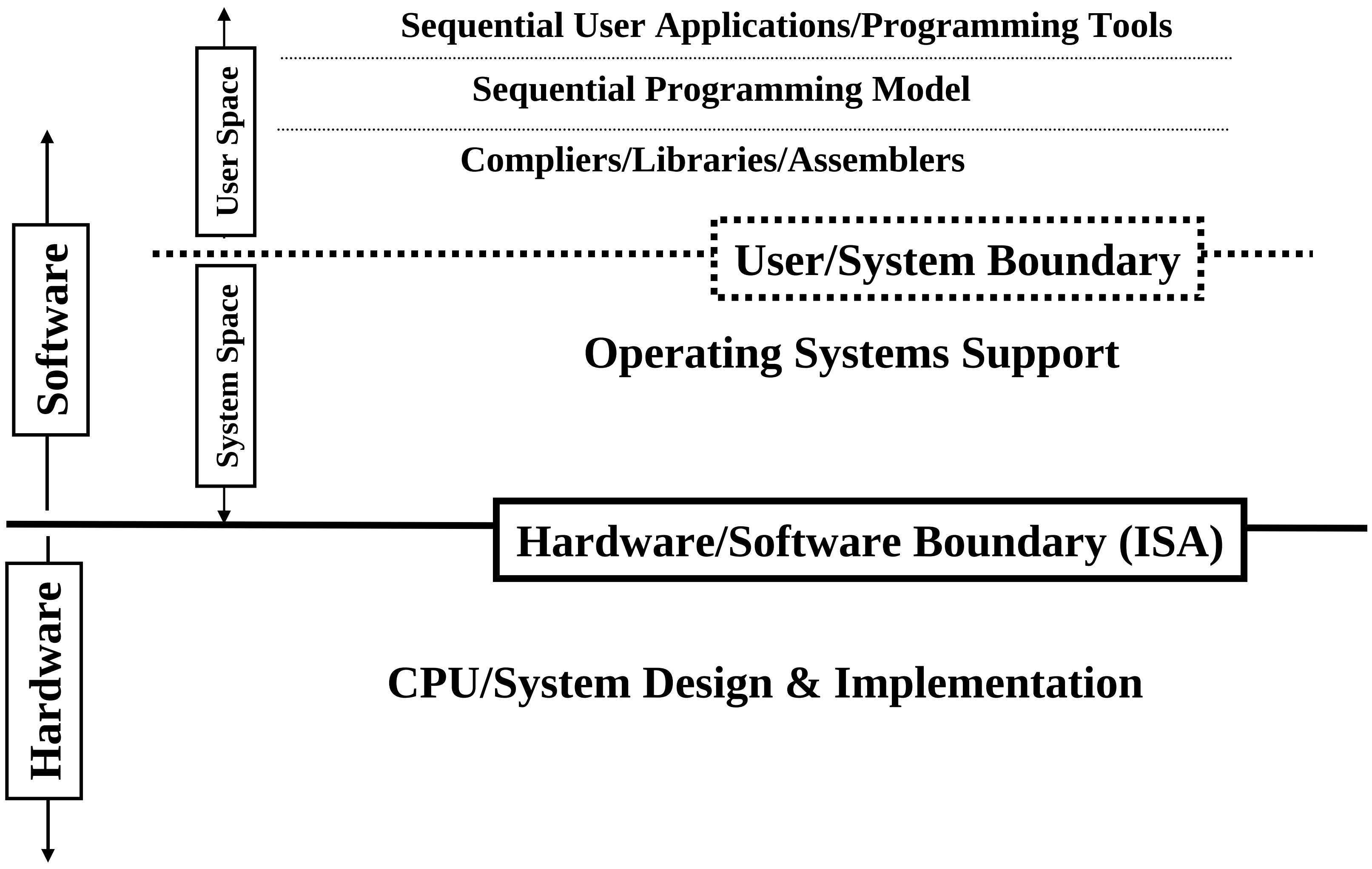
2 **<u>Realization of abstraction layers:</u>**

- **The organizational structures that realize (implement) the abstraction layers to deliver high performance in a cost-effective manner.**
  - **Implementation of abstraction layers in system software (OS)/hardware.**

Conventional = Sequential or Single Processor

PCA Chapter 1.2, 1.3

Conventional Computer Architecture Abstraction:
Critical Abstraction Layers


Sequential User Applications/Programming Tools
Sequential Programming Model
Compliers/Libraries/Assemblers
User Space
User/System Boundary
System Space
Software
Operating Systems Support
Hardware/Software Boundary (ISA)
Hardware
CPU/System Design & Implementation

Conventional = Sequential or Single Processor

Parallel Programming Models

- **<u>A parallel computer system</u> is a collection of communicating processing elements that communicate and cooperate to solve large problems fast.**
- **<u>A parallel program</u> consists of two or more threads of control (parallel tasks) that operate on data.**

Each task only executes on one processor to which it has been mapped or allocated

- **<u>A parallel programming model</u> is the conceptualization of the parallel machine and programming methodology used in coding parallel applications that specifies communication and synchronization.**
  - **Parallel programming models specify how parallel tasks of a parallel program <u>communicate</u> and what <u>synchronization</u> operations are available to <u>coordinate</u> their activities and <u>order</u>. This includes specifying:**

    1. **What data can be named by a task or thread.** ***Naming***
    2. **What operations can be performed on the named data.** ***Operations***
    3. **What order exists among these operations.** ***Order***

How?

- **Typically the parallel programming model is supported at the user level by parallel languages or parallel programming environments in the form of user-level communication and synchronization primitives.**
- **Historically, parallel architectures were <u>tied</u> to parallel programming models.**
- **As parallel programming environments have matured, it led to <u>the separation between parallel programming models and parallel machine organization</u> (system implementation) forming <u>"the communication abstraction".</u>**

Common Parallel Programming Models

<u>Parallel Programming Model (definition):</u>
Parallel programming methodology used in coding parallel applications that specifies <u>communication</u> and <u>synchronization.</u> Or ..

<u>A parallel programming model</u> is the conceptualization of the parallel machine and programming methodology used in coding parallel applications and specifies <u>how parallel tasks</u> of a parallel program <u>communicate</u> and what <u>synchronization operations are available</u>.

Most Common

- ***<u>Shared memory Address Space (SAS):</u>***

  Parallel program threads or tasks communicate using a shared memory address space (shared data in memory).

- ***<u>Message passing</u>*****<u>:</u>**

  Explicit point to point communication is used between parallel program tasks using messages.

- ***<u>Data parallel</u>*****<u>:</u>**

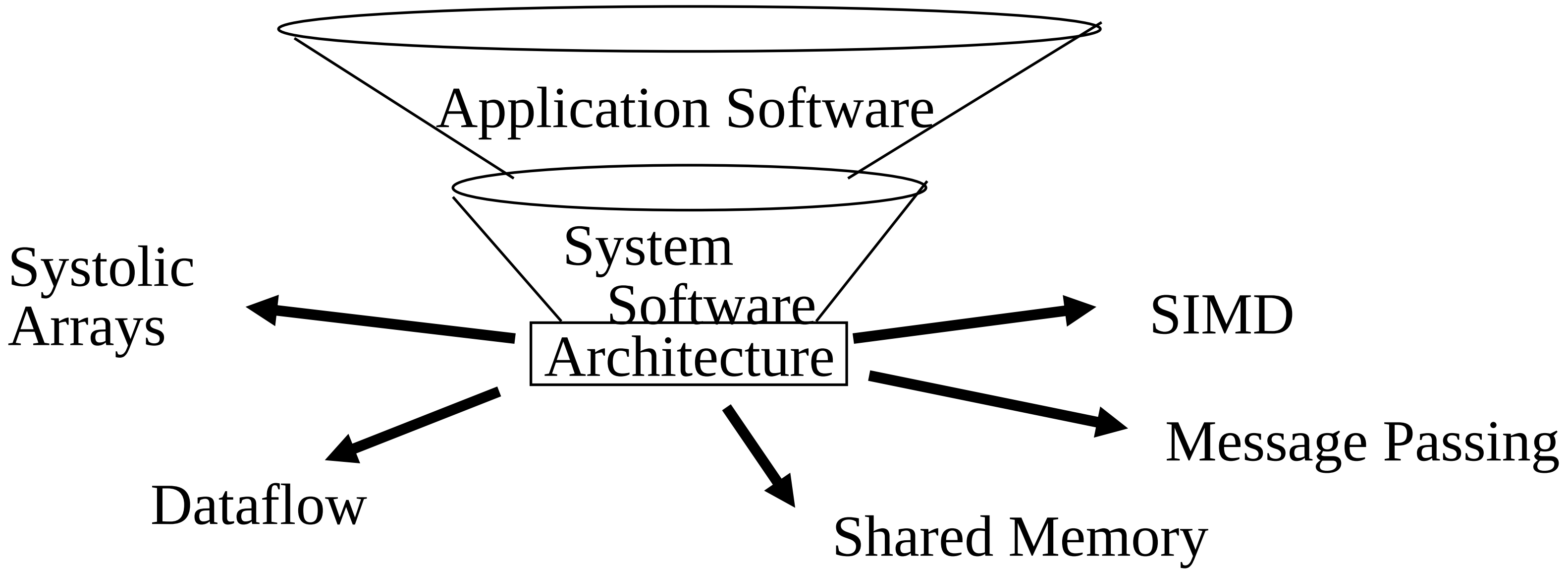
  More regimented, global actions on data (i.e the same operations over all elements on an array or vector)

  – **Can be (and usually) implemented with shared address space (SAS) or message passing**

Parallel Architectures History

Historically, parallel architectures and implementations were <u>tied</u> to parallel programming models:

- **Divergent architectures, with no predictable pattern of growth**.



As parallel programming environments have matured, it led to <u>the separation between parallel programming models and parallel machine organization</u> (system implementation) extending conventional computer architecture abstraction and forming <u>"the communication abstraction".</u>

(PCA Chapter 1.2, 1.3)

Current Trends In Parallel Architectures Abstraction

- As defined earlier, a parallel computer is a collection of processing elements that communicate and cooperate to solve large problems fast.
- This requires the extension of conventional computer architecture abstraction (user/system, ISA) to account for communication and cooperation among processors.
- The extension of "computer architecture" to support communication and cooperation:

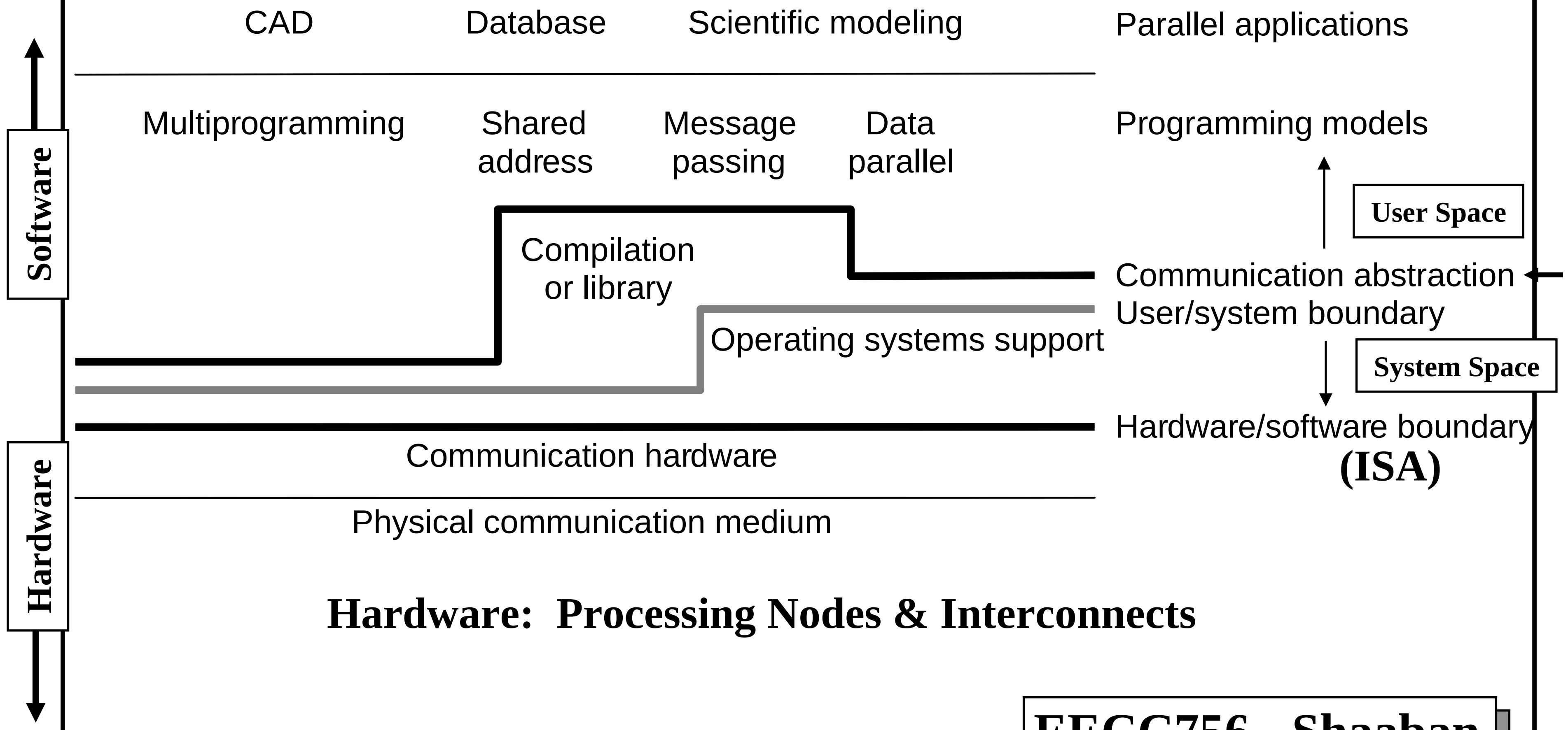
  + ordering/synchronization

  - OLD: Instruction Set Architecture.
  - NEW: *Communication Architecture.*

- The Communication Architecture Defines:
  - 1 – Critical abstractions, boundaries: Interfaces
    - Communication Abstraction
      - Basic user-level communication and synchronization operations (Primitives) that are used to realize a parallel programming model.
    - User/System Boundary.
    - ISA Software/Hardware Boundary.

    } Also in conventional computer architecture abstraction
  - 2 – Organizational structures that implement interfaces (hardware or software).
- Compilers, libraries and OS are important bridges today between programming model requirements and parallel hardware implementation.

Modern Parallel Architecture Abstraction
Layered Framework



Hardware: Processing Nodes & Interconnects

Communication Abstraction

- The communication abstraction forms the <u>key interface</u> between the <u>programming model</u> and <u>system implementation</u>.
- Plays a role in parallel architecture similar to instruction set (ISA) in sequential computer architecture.
- <u>User-level communication/synchronization primitives provided:</u>
  - Realizes the parallel programming model.
  - Mapping exists between language primitives of programming model and these primitives.
- <u>Primitives supported directly by hardware, or via OS, or via user software.</u>
- Lot of debate about what to support in software and gap between layers.

Even for conventional computer architecture

- Today:
  - Hardware/software interface tends to be flat, i.e. complexity roughly uniform.
  - <u>Compilers and software play important roles as bridges.</u>
  - Technology trends exert strong influence
- Result is convergence in organizational structure
  - Relatively simple, general purpose communication primitives.

Communication Abstraction Requirements

- **Key interface** between the **programming model** and **system implementation**.
  - Provides user-level communication/synchronization primitives used to implement parallel programming models via parallel programming environments. *from last slide*
- **Requirements from the software side:**
  - It must have a precise, **well-defined meaning** so the same program will **run correctly on many parallel machine implementations.**
  - The user-level operations "**primitives**" provided by this layer must be **simple** with **clear performance costs** so the software can be optimized for performance.
- **Requirements from the hardware side:**
  - It must have a **well defined meaning** so the machine designer can determine where performance optimization are possible.
  - **Not too overly specific** so it does not prevent useful techniques for performance optimizations that exploit new technologies.
- Thus, the communication abstraction is a set of requirements or **"contract"** between the hardware and software allowing each the flexibility to improve what it does while working together correctly.

Communication Architecture

= *User/System Interface + Implementation*

- <u>**User/System Interface:**</u>
  - **Communication primitives exposed to user-level by hardware and system-level software (e.g. OS).**
- <u>**Implementation:**</u>
  - **Organizational structures that implement the primitives: hardware or OS.**
  - **How optimized are they? How integrated into processing node?**
  - **Structure of network.**
- **Goals:**
  - **Performance**
  - **Broad applicability**
  - **Ease of programmability**
  - **Scalability**
  - **Low cost of implementation**

Toward Architectural Convergence

- **Evolution and role of software have blurred boundary:**

e.g

1. **Send/receive (message passing model) supported on SAS machines via buffers.**
2. **SAS in message-passing machines: Can construct global address space on massively parallel processor (MPPs) message-passing machines by carrying along pointers specifying the process and local virtual address space.**
3. **Shared virtual address space in message-passing machines can also be established at the page level generating a page fault for remote pages handled by sending a message.**

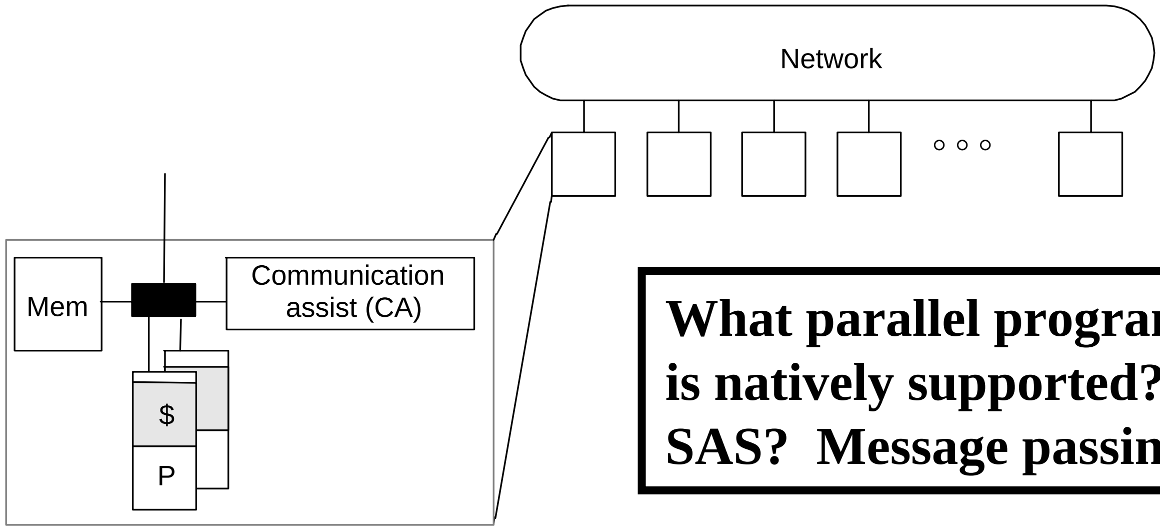
- **Hardware organization converging too:**
  - **Tighter integration even for MPPs (low-latency, high-bandwidth networks):**
    - **Network interface tightly integrated with memory/cache controller.**
    - **Transfer data directly to/from user address space.**
    - **DMA transfers across the network.**
  - **At lower level, even hardware SAS passes hardware messages.**
- **Even clusters of workstations/SMPs are becoming parallel systems:**
  - **Emergence of fast system area networks (SAN): ATM, fiber channel ...**
- **Programming models still distinct, but organizations converging:**
  - **Nodes connected by scalable network and communication assists (CAs).**
  - **Implementations also converging, at least in high-end machines.**

i.e. Architectural convergence between SAS/Message-Passing

Convergence of Scalable Parallel Machines:

Generic Parallel Architecture

- **A generic <u>scalable</u> modern multiprocessor:**



What parallel programming model is natively supported?
SAS? Message passing?

<u>Node:</u> processor(s), memory system, plus *communication assist (CA):*

- **Network interface and communication controller.**

- **<u>Scalable network.</u>**
- **Convergence allows lots of innovation, now within framework**
  - **Integration of assist with node, what operations, how efficiently...**

<u>Scalable:</u> Continue to achieve good parallel performance "speedup"as the sizes of the system/problem are increased

Communication Assist (CA) Design Considerations

- **The performance and capabilities of the communication assist play a very crucial role in today's scalable parallel architectures**
- **Different parallel programming models place different requirements on the design of the communication assist**
  - **This influences which operations are common and should be optimized.**
- **<u>In the shared memory case:</u>** ***(SAS)***
  - **The CA is tightly integrated with the memory system in order to capture (observe" memory events that require interaction with other nodes.**
  - **It must accept messages and perform local memory operations on behalf of other nodes**

i.e remote memory access requests

- **<u>In the message passing case:</u>**
  - **Communication is initiated explicitly by user or system (sends/receives) so observing memory system events is not needed.**
  - **A need exists to initiate messages and respond to incoming messages quickly possibly requiring it to perform tag matching.**

<u>In SAS:</u> Communication is implicit (via loads/stores)

<u>In Message Passing:</u> Communication is explicit (via sends/receives)

Understanding Parallel Architecture

- Traditional taxonomies (e.g. Flynn's SIMD/MIMD ..) not very useful since multiple general-purpose microprocessors are dominant as processing elements.
- Programming models are not enough, nor hardware implementation structures.
  - Programming models can be supported by radically different architectures.

Via software layer(s)

- <u>*Focus on architectural distinctions that affect software*</u>

performance

  - (e.g. That affect Compilers, libraries, programs.)
- Design of user/system and hardware/software interface
  - Constrained from above by programming models and below by technology.
- Guiding principles provided by layers.
  - What primitives are provided at communication abstraction.
  - How programming models map to these.
  - How they are mapped to hardware.

<u>i.e implementation of primitives</u>

Fundamental Design Issues

- At any layer, interface (contract or set of requirements) aspect and performance aspect:

Functionality

  – ***Naming***: How are logically shared data and/or processes referenced?

i.e Data access/transfer/communication operations

  – ***Operations***: What operations are provided on these data.
  – ***Ordering***: How are accesses to data ordered and coordinated to satisfy program threads dependencies?

For Performance

  – **Replication**: How are data replicated to reduce communication overheads?

i.e copied or cached

  – **Communication Cost**: Time added to parallel execution time as a result of communication.

More on this later in the lecture

- Understand these issues at programming model level first, since that sets the requirements on lower layers.

e.g local copies of data (as in cache)

Sequential Programming Model

Contract (or requirements)

of process

1 – **Naming:** Can name any variable in virtual address space
  - Hardware/Software (OS) does translation to physical addresses.

2 – **Operations:** Loads and Stores. + arithmetic .. etc.

3 – **Ordering:** Sequential program order.

Performance

– Compilers and hardware must preserve the data dependence order (for correctness).

– However, compilers and hardware violate other orders without getting caught.
  - **Compiler:** reordering and register allocation, etc…
  - **Hardware:** out of order, register bypassing, write buffers, etc..

– **Replication:** Transparent replication of data in caches:
  - To hide long memory latency (communication time with memory)

SAS Programming Model

In SAS: Communication is implicit via loads/stores of data in shared space
Synchronization is explicit using synchronization operations (e.g locks)

1 • **Naming:** Any process can name any variable in shared space. In addition to naming private variables in its private non-shared space

2 • **Operations:** **Implicit** communication via loads and stores (in shared space), plus those needed for **explicit** ordering and thread synchronization.

3 • **Simplest Ordering Model:**
  – Within a process/task/thread: sequential program order.
  – Across threads: some interleaving (as in time-sharing).
  – Additional orders through synchronization. To satisfy dependencies
  – Again, compilers/hardware can violate orders without getting caught.
  – Different, more subtle ordering models also possible.

i.e Memory Access Ordering Models

Synchronization in SAS

A parallel program must coordinate the ordering of activity of its threads (parallel tasks) to ensure that dependencies within the program are enforced.

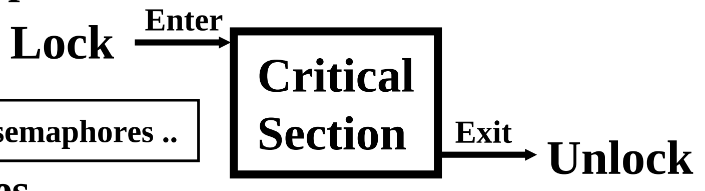
- **This requires explicit synchronization operations when the ordering implicit within each thread is not sufficient.**

Two Types

In SAS synchronization is explicit using synchronization operations: (or primitives)

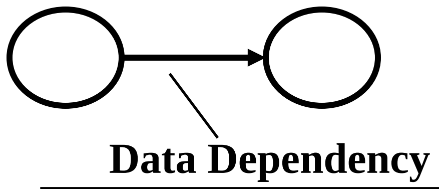
1 Mutual exclusion (locks): One-at-a-time access

- **Ensure certain operations on certain data (in shared space) can be performed by only one process (task) at a time (that acquires the lock).**
  - **Critical Section:** Room that only one task/process can enter at a time.
- **No ordering guarantees.**

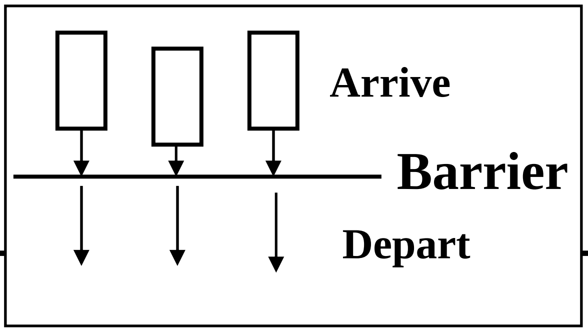


2 Event synchronization: Implemented using locks, flags semaphores ..

- **Ordering of events to preserve dependencies**
  - **e.g. producer —> consumer of data**
- **3 main types:**
  - **Point-to-point**
  - **Global**
  - **Group**



Implies ordering of events



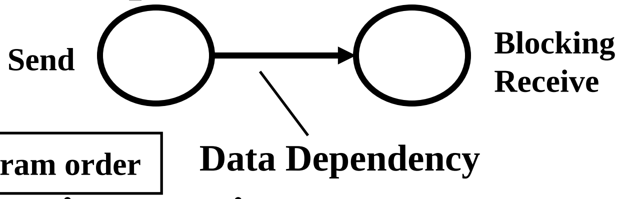
Message Passing Programming Model

In Message Passing : Communication is explicit via Sends/Receives
Synchronization is implicit via blocking send/receive pairs

1 • **Naming:** Processes can only name private data directly. In its private address space
- No shared address space.

2 • **Operations:** Explicit communication through *send* and *receive*
- Send transfers data from private address space to another process.
- Receive copies data from process to private address space.
- Must be able to name processes. i.e Sender /recipient

3 • **Ordering:**
- Program order within a process. i.e sequential program order
- Blocking send and receive can provide implicit point to point synchronization between tasks/processes.
- Mutual exclusion inherent.



• Can construct global address space:
- Process number + address within process address space
- But no direct operations on these names at the communication abstraction level (must be done by user programs/ parallel programming environment).

SAS on Message passing machines

Design Issues Apply At All Layers

- **Programming model's position or requirements provide constraints/goals for the system.**
- **In fact, each interface between layers supports or takes a position on:**
  - **Naming model.**
  - **Set of operations on names**
  - **Ordering model.**

  Functionality

  - **Replication.**
  - **Communication cost and performance.**

  Performance

- **Any set of positions can be mapped to any other by software.**
- **Next: Let's see issues across layers:**
  - **<u>How lower layers can support contracts (requirements) of programming models.</u>**
  - **Performance issues.**

Lower Layers Support of Naming and Operations For SAS

i.e Realizing SAS Parallel Programming Model

- Naming and operations in programming model can be directly supported by lower levels, or translated by compiler, libraries or OS

Example: Shared virtual address space in programming model SAS

More Software Layers

Realizing SAS

1 **Hardware interface supports *shared physical address space***
  - Direct support by hardware through virtual-to-physical mappings, no software layers.

2 **Hardware supports independent physical address spaces:**
  - Can provide SAS through OS, in system/user interface
    - v-to-p mappings only for data that are local.
    - Remote data accesses incur page faults; brought in via page fault handlers. OS/system software support needed
    - Same programming model, different hardware requirements and cost model.

3 **Or through compilers or runtime, so above sys/user interface** In user space
    - shared objects, instrumentation of shared accesses, compiler support.

Lower Layers Support of Naming and Operations For Message Passing Model

Example: Implementing Message Passing

1 **<u>Direct support at hardware interface:</u>**

- **But message matching and buffering benefit from the added flexibility provided by software.**

2 **<u>Support at sys/user interface or above in software (almost always)</u>**

- **Hardware interface provides basic data transport (well suited).**
- **Send/receive built in software for flexibility (matching, protection, buffering, etc.).**
- **<u>Choices at user/system interface:</u>**
  - **All messages go through OS each time: expensive**
  - **OS sets up once/infrequently, then little software involvement each time for simple data transfer operations.** **To reduce OS involvement**

3 **Or lower interfaces provide SAS, and send/receive built on top with buffers and loads/stores.**

- **Need to examine the issues and tradeoffs at every layer**
  - **Frequencies and types of operations, costs.**

Lower Layers Support of Ordering

- **Message passing:** No assumptions on orders across processes except those imposed by send/receive pairs.
- **SAS:** How processes see the order of other processes' references defines semantics of SAS:
  - Ordering is very important and subtle.
  - Uniprocessors play tricks with orders to gain parallelism or locality. e.g out of order execution, buffering
  - These are more important in multiprocessors.
  - Need to understand which old tricks are valid, and learn new ones.
  - How programs behave, what they rely on, and hardware implications.

Lower Layers Support of Replication

- **Very important for reducing data transfer/communication.**

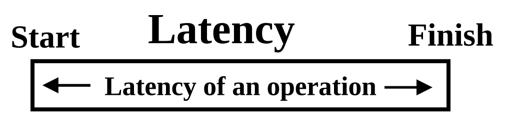
to improve parallel performance (lower communication cost)

- **Again, depends on naming model.**
- **Uniprocessor: caches do it automatically**
  - **Reduce communication with memory.**
- **Message Passing naming model at an interface:**
  - **A receive replicates data, giving a new name (renames) in private address space; subsequently use new name.**
  - **Replication is explicit in software above that interface.**
- **SAS naming model at an interface:**
  - **A load brings in data transparently (from shared space), so can replicate transparently (i.e in local node cache).**
  - **Hardware caches do this, e.g. in shared physical address space.**
  - **OS can do it at page level in shared virtual address space, or objects.**
  - **No explicit renaming, many copies for same name: *coherence* problem**
    - **In uniprocessors, "coherence" of copies is natural in memory hierarchy (what about write-back cache?).**

Thus in SAS, cache coherence protocols are needed to ensure data consistency of the various cached data copies

Communication Performance

- Performance characteristics determine usage of operations at a layer:
  - Programmer, compilers etc. make choices based on this
- Fundamentally, three characteristics:
  - ***Latency*:** time taken for an operation.
  - ***Bandwidth*:** rate of performing operations (or throughput).
  - ***Cost*:** impact on execution time of program.

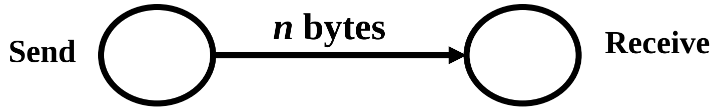


- If processor (or system component, network etc..) does one thing at a time: bandwidth is proportional to 1/latency

i.e no operation overlap or pipelining

  - But actually more complex in modern systems due to overlapping of operations/pipelining.
- Characteristics apply to overall operations, as well as individual components of a system, however small
- We'll focus on communication or data transfer across nodes (over the network).

Linear Model of Data Transfer Latency



$$\textit{Transfer time (n)} = T_0 + n/B$$

T_0 = *Start-up cost* B = Transfer rate (or bandwidth)   n = Amount of data

- Useful for message passing, memory access, etc.
- As *n* increases, bandwidth approaches asymptotic rate *B*
- How quickly it approaches depends on T_0
- Size needed for half bandwidth (half-power point):

$$n_{1/2} = T_0 / B$$

- But the linear model is not enough:
  - When can next transfer be initiated? Can cost be overlapped?
  - Need to know how the transfer is performed.

Communication Cost Model

Bottleneck component delay (e.g at CA)

Comm Time per message(n) = Overhead + Occupancy + Network Delay

= Overhead + Occupancy + Network Latency + Size/Bandwidth + Contention

$$= o_v + o_c + l + n/B + T_c$$

Overhead, o_v = Time for the processor to initiate the transfer.

Occupancy, o_c = The time it takes data to pass through the slowest(bottleneck) component on the communication path. Limits frequency of communication operations. e.g Communication Assist (CA)

$l + n/B + T_c$ = *Total* Network Delay, can be hidden by overlapping with other processor operations.

Shown next slide

- **Overhead and assist occupancy may be *f(n)* or not.**
- **Each component along the way has occupancy and delay**
  - **Overall delay is sum of delays.**
  - **Overall occupancy (1/bandwidth) is biggest of occupancies**

n = size of message in bytes

Communication Cost Model (Continued)

Communication Cost = Time added to parallel execution time as a result of communication

Added to parallel execution time

Comm Cost = frequency * (Comm time - overlap)

Communication time per message (from last slide)

Frequency of Communication:

- The number of communication operations (or messages) per unit of work in the program. Or total per program
- Depends on many program and hardware factors.
  - Hardware may limit transfer size increasing comm. Frequency.
- Also affected by degree of hardware data replication and migration.

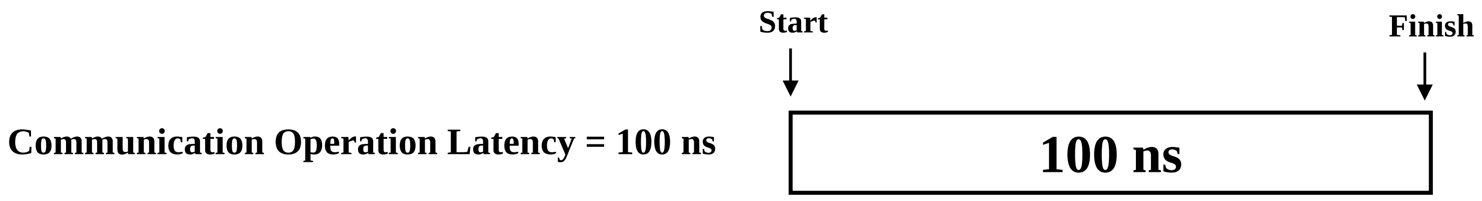
The Overlap:

- The portion of the communication operation time performed concurrently with other useful work including computation and other useful work. $\rightarrow$ To hide long communication latency/time
- Reduction of effective communication cost is possible because much of the communication work is done by components other than the processor including:
  - Communication assist, bus, the network, remote processor or memory.

How?

Simple Communication Cost Example

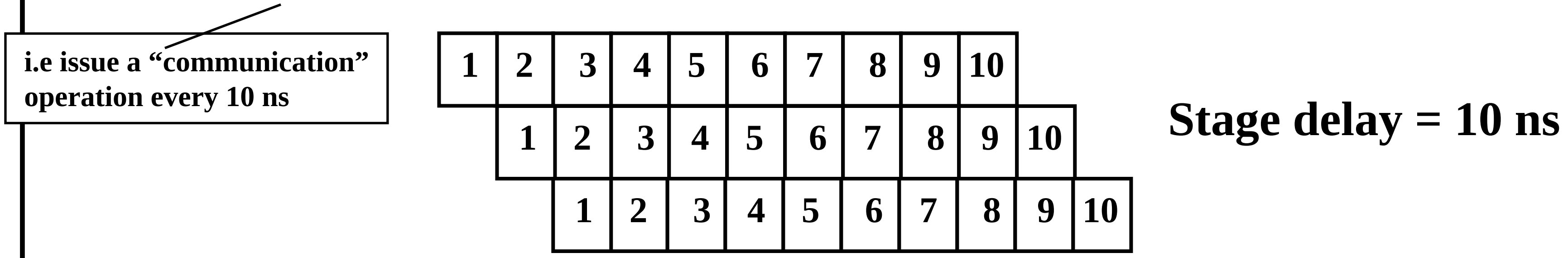
- **Component (or network) performs an operation in 100ns (latency).**



  – **Simple bandwidth without pipelining or overlap (one communication operation at a time):**

Throughput or Bandwidth = 1/ Latency = 10 Million Operations/sec (Mops)

- **If component (or network) is pipelined with 10 stages**



- **Peak bandwidth = 1 /stage delay = 1/10 ns = 100 Mops**
  – **Rate determined by slowest stage of pipeline, not overall latency.**

Simple Communication Cost Example (Continued)

- Delivered bandwidth to application depends on initiation frequency.

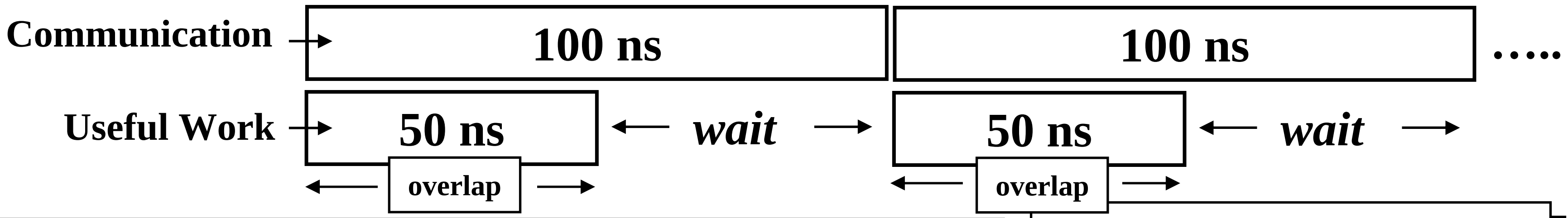
or effective throughput

- Suppose application performs a total of 100 million communication operations on this component. What is the range of cost of these operations to the application?
  - Op count * Op latency gives 100/10 = 10 sec (upper bound)
    - Assume no overlap with useful work.
  - Op count / peak op rate gives 1 sec (lower bound)
    - Assumes full overlap of latency with useful work, so just issue cost.
  - If application can do 50 ns of useful work before depending on result of Op, cost to application is the other 50ns of latency
    - Total cost to application = 5 sec

i.e time added to execution time

i.e delay of one pipeline stage = 10 ns

i.e 50 ns is overlapped with useful work



Communication Cost = Time added to execution time as a result of communication

Summary of Design Issues

- Functional and performance issues apply at all layers
- <u>Functional:</u> Naming, operations and ordering.
- <u>Performance:</u> Replication (to reduce communication), Organization, latency, bandwidth, overhead, occupancy.
- Replication and communication are deeply related:
  - Management depends on naming model.
- <u>Goal of architects:</u> design against frequency and type of operations that occur at communication abstraction, constrained by tradeoffs from above (parallel applications, programming models) or below (lower layers, hardware architecture).
  - Hardware/software tradeoffs.