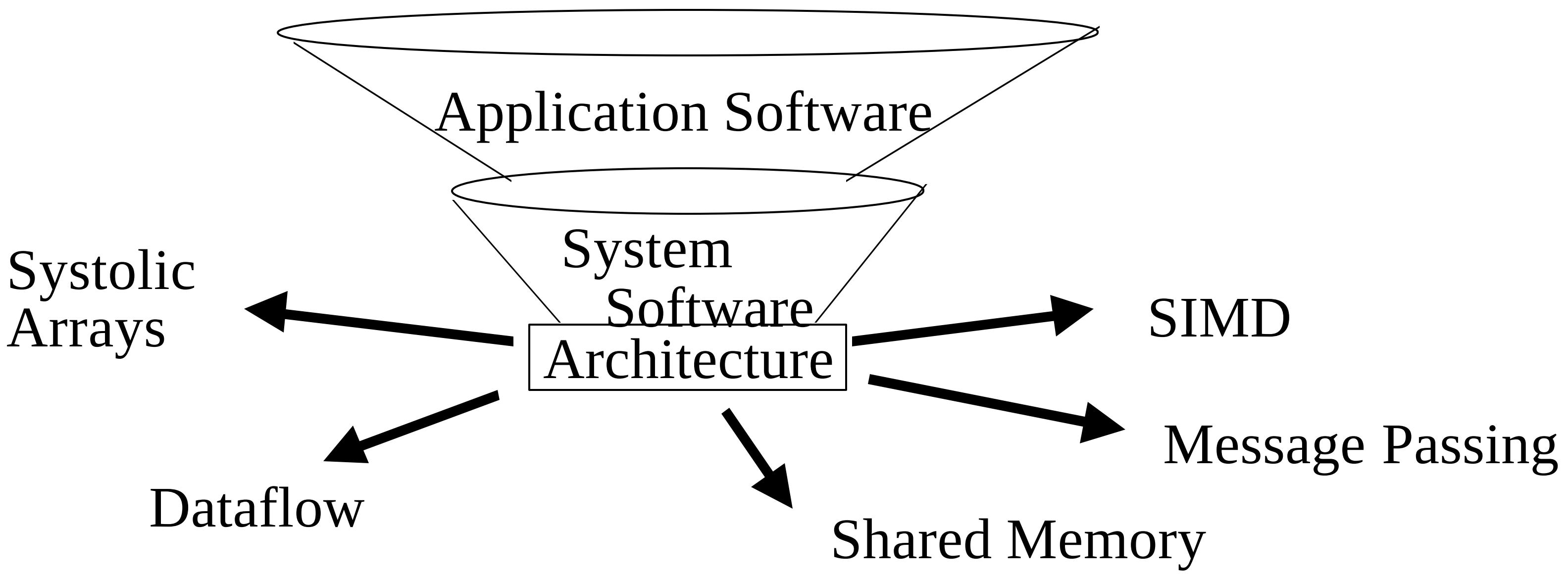
Parallel Architectures History

Historically, parallel architectures tied to programming models

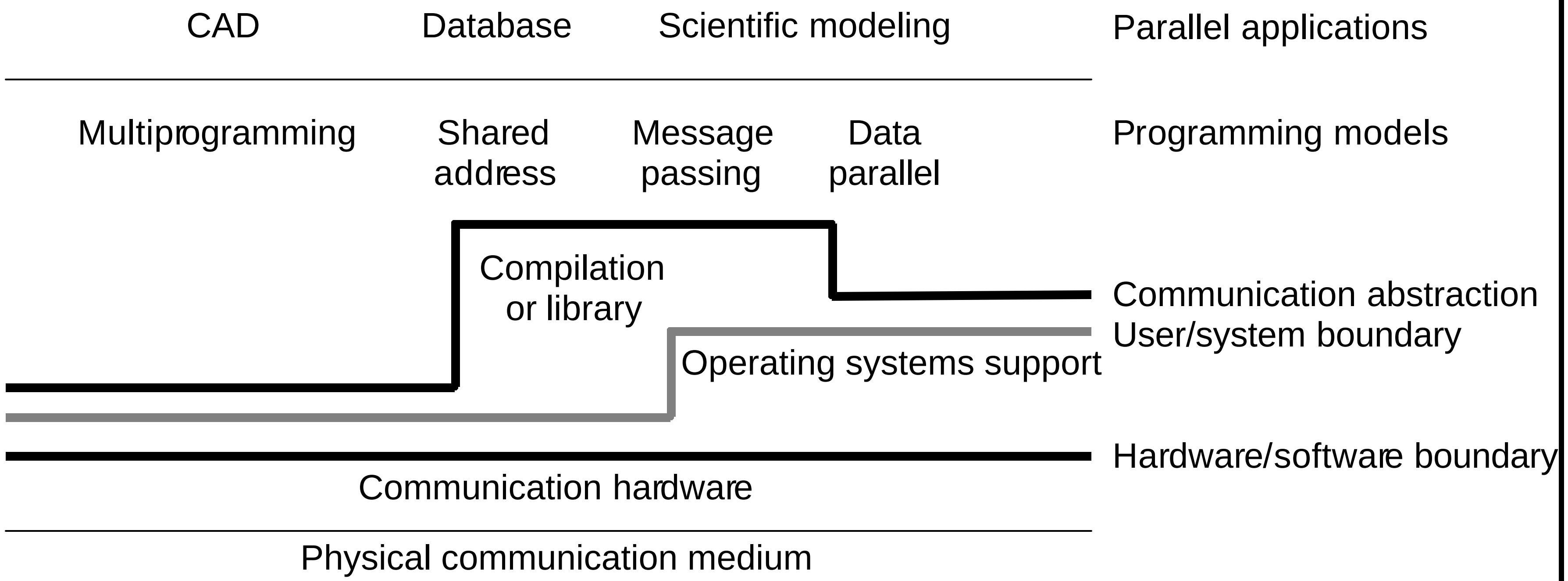
- **Divergent architectures, with no predictable pattern of growth**.



Current Trends In Parallel Architectures

- **The extension of "computer architecture" to support communication and cooperation:**
  - **OLD: Instruction Set Architecture.**
  - **NEW: *Communication Architecture.***
- **Defines:**
  - **Critical abstractions, boundaries, and primitives (interfaces).**
  - **Organizational structures that implement interfaces (hardware or software).**
- **Compilers, libraries and OS are important bridges today**

Modern Parallel Architecture Layered Framework



Programming Models

- **Programming methodology used in coding applications**
- **Specifies communication and synchronization**

- **Examples:**
  - **Multiprogramming:**

    No communication or synchronization at program level
  - ***Shared memory address space:***
  - ***Message passing*:**

    Explicit point to point communication
  - ***Data parallel*:**

    More regimented, global actions on data
    - **Implemented with shared address space or message passing**

Communication Abstraction

- **User-level communication primitives provided**
  - **Realizes the programming model.**
  - **Mapping exists between language primitives of programming model and these primitives**
- **Supported directly by hardware, or via OS, or via user software.**
- **Lot of debate about what to support in software and gap between layers.**
- **Today:**
  - **Hardware/software interface tends to be flat, i.e. complexity roughly uniform.**
  - **Compilers and software play important roles as bridges today.**
  - **Technology trends exert strong influence**
- **Result is convergence in organizational structure**
  - **Relatively simple, general purpose communication primitives.**

Communication Architecture

= *User/System Interface + Implementation*

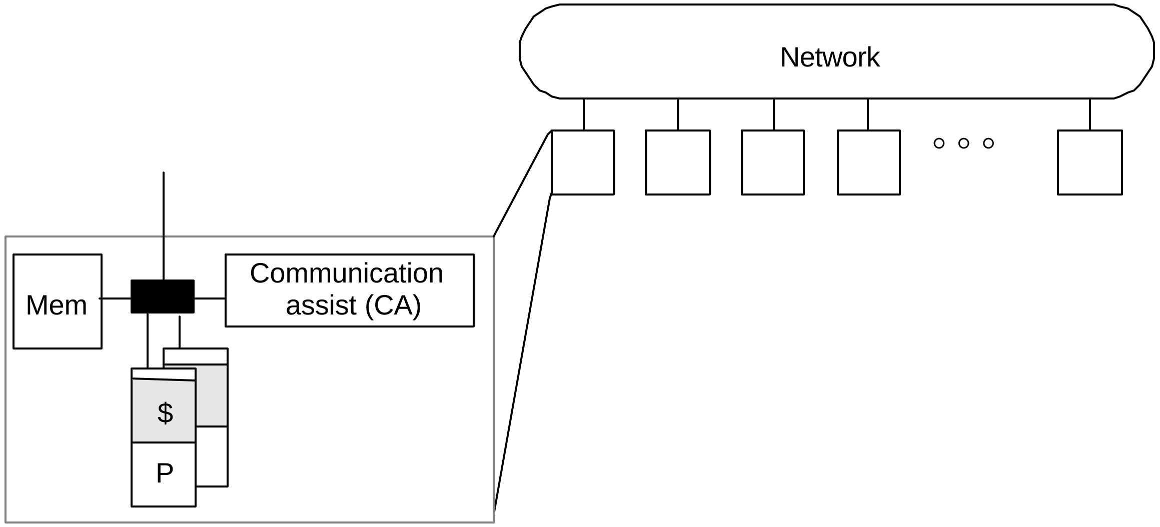
- **User/System Interface:**
  - **Communication primitives exposed to user-level by hardware and system-level software.**
- **Implementation:**
  - **Organizational structures that implement the primitives: hardware or OS.**
  - **How optimized are they? How integrated into processing node?**
  - **Structure of network.**
- **Goals:**
  - **Performance**
  - **Broad applicability**
  - **Programmability**
  - **Scalability**
  - **Low Cost**

Toward Architectural Convergence

- Evolution and role of software have blurred boundary:
  - Send/receive supported on SAS machines via buffers.
  - Can construct global address space on massively parallel (MP) message-passing machines by carrying along pointers specifying the process and local virtual address space.
  - 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 MP (low-latency, high-bandwidth):
    - 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 distinct, but organizations converging:
  - Nodes connected by general network and communication assists.
  - Implementations also converging, at least in high-end machines.

Convergence of Scalable Parallel Machines:
Generic Parallel Architecture

- **A generic modern multiprocessor:**



Node: processor(s), memory system, plus *communication assist:*

  - **Network interface and communication controller.**

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

Understanding Parallel Architecture

- Traditional taxonomies not very useful.
- Programming models are not enough, nor hardware structures.
  - Can be supported by radically different architectures.
- ***Architectural distinctions that affect software***
  - 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.

Fundamental Design Issues

- **At any layer, interface (contract) aspect and performance aspects:**
  - ***Naming*: How are logically shared data and/or processes referenced?**
  - ***Operations*: What operations are provided on these data.**
  - ***Ordering*: How are accesses to data ordered and coordinated to satisfy program threads dependencies?**
  - **Replication: How are data replicated to reduce communication overheads?**
  - **Communication Cost: Latency, bandwidth, overhead, occupancy.**
- **Understand at programming model level first, since that sets requirements from lower layers.**
- **Other issues:**
  - **Node Granularity: How to split between processors and memory?**
  - **...**

Sequential Programming Model

Contract

- **Naming: Can name any variable in virtual address space**
  - **Hardware (and perhaps compilers) does translation to physical addresses.**
- **Operations: Loads and Stores.**
- **Ordering: Sequential program order.**

Performance

- **Rely on dependencies on single location (mostly): *dependence order.***
- **Compilers and hardware violate other orders without getting caught.**
- **Compiler: reordering and register allocation**
- **Hardware: out of order, pipeline bypassing, write buffers**
- **Transparent replication in caches**

SAS Programming Model

- **Naming: Any process can name any variable in shared space.**
- **Operations: loads and stores, plus those needed for ordering and thread synchronization.**
- **Simplest Ordering Model:**
  - **Within a process/thread: sequential program order.**
  - **Across threads: some interleaving (as in time-sharing).**
  - **Additional orders through synchronization.**
  - **Again, compilers/hardware can violate orders without getting caught.**
  - **Different, more subtle ordering models also possible.**

Synchronization

Mutual exclusion (locks):

- Ensure certain operations on certain data can be performed by only one process at a time.
- Room that only one person can enter at a time.
- No ordering guarantees.

Event synchronization:

- Ordering of events to preserve dependences
  - e.g. producer —> consumer of data
- 3 main types:
  - point-to-point
  - global
  - group

Message Passing Programming Model

- **Naming: Processes can name private data directly.**
  - **No shared address space.**
- **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.**
- **Ordering:**
  - **Program order within a process.**
  - **Blocking send and receive can provide point to point synchronization between 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.**

Design Issues Apply at All Layers

- Prog. model's position provides 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.
  - Replication.
  - Communication performance.
- Any set of positions can be mapped to any other by software.
- Let's see issues across layers:
  - How lower layers can support contracts of programming models.
  - Performance issues.

Lower Layers Support of Naming and Operations

- **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

- **Hardware interface supports *shared physical address space***
  - **Direct support by hardware through virtual-to-physical mappings, no software layers.**
- **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.**
    - **Same programming model, different hardware requirements and cost model.**
  - **Or through compilers or runtime, so above sys/user interface**
    - **shared objects, instrumentation of shared accesses, compiler support.**

Lower Layers Support of Naming and Operations

Example: Implementing Message Passing

- Direct support at hardware interface:
  - But message matching and buffering benefit from the added flexibility provided by software.
- Support at sys/user interface or above in software (almost always)
  - Hardware interface provides basic data transport (well suited).
  - Send/receive built in sw for flexibility (protection, buffering).
  - Choices at user/system interface:
    - All messages go through OS each time: expensive
    - OS sets up once/infrequently, then little software involvement each time for simple data transfer operations.
  - 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.
  - 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.**
- **Again, depends on naming model.**
- **Uniprocessor: caches do it automatically**
  - **Reduce communication with memory.**
- **Message Passing naming model at an interface:**
  - **A receive replicates, giving a new name; subsequently use new name.**
  - **Replication is explicit in software above that interface**
- **SAS naming model at an interface**
  - **A load brings in data transparently, so can replicate transparently**
  - **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.**

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.**
  - ***Cost*: impact on execution time of program.**
- **If processor does one thing at a time: bandwidth µ 1/latency**
  - **But actually more complex in modern systems.**
- **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.**

Simple Communication Cost Example

- Component performs an operation in 100ns.
- Simple bandwidth: 10 M operations
- Internally pipeline depth 10 => bandwidth 100 Mops
  - Rate determined by slowest stage of pipeline, not overall latency.
- Delivered bandwidth on application depends on initiation frequency.
- Suppose application performs 100 M operations. What is cost?
  - op count * op latency gives 10 sec (upper bound)
  - 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

Linear Model of Data Transfer Latency

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

T_0 = *Start-up cost* **B = Transfer rate** **n = Amount of data**

- **useful for message passing, memory access, vector ops 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

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 = Time for the processor to initiate the transfer.

Occupancy = The time it takes data to pass through the slowest component on the communication path. Limits frequency of communication operations.

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

- **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**

Communication Cost Model

Comm Cost = frequency * (Comm time - overlap)

Frequency of Communication:

- The number of communication operations per unit of work in the 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.
- 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.

Summary of Design Issues

- Functional and performance issues apply at all layers
- Functional: Naming, operations and ordering.
- Performance: Organization, latency, bandwidth, overhead, occupancy.
- Replication and communication are deeply related:
  - Management depends on naming model.
- Goal of architects: design against frequency and type of operations that occur at communication abstraction, constrained by tradeoffs from above or below.
  - Hardware/software tradeoffs.