Parallel Programming Models and Architecture

CS 740
September 22, 2014

Seth Goldstein
Carnegie Mellon University

One Definition of Parallel Architecture

A parallel computer is a collection of processing elements that cooperate to solve large problems fast

Some broad issues:

- **Resource Allocation**:
  - how large a collection?
  - how powerful are the elements?
  - how much memory?
- **Data access, Communication and Synchronization**
  - how do the elements cooperate and communicate?
  - how are data transmitted between processors?
  - what are the abstractions and primitives for cooperation?
- **Performance and Scalability**
  - how does it all translate into performance?
  - how does it scale?

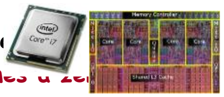
Why Study Parallel Architecture and Programming?

The Answer from 10 Years Ago:

- **Mostly, Because it allows you to achieve performance beyond what we get with CPU clock frequency scaling**
  - important for applications with high performance demands
- **Rarely, Exploit concurrency for programmability**

The Answer Today:

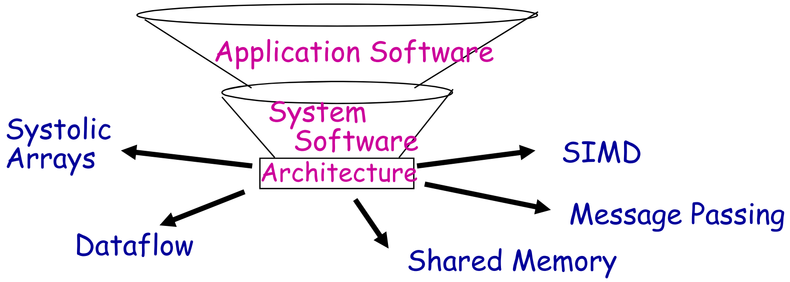
- **Because it is the *only way* to achieve higher performance in the foreseeable future**
- **CPU clock rates are no longer increasing!**
- **Instruction-level-parallelism is not increasing eith**
- **Without parallel programming, performance becom[illegible] game.**
- **Improved dependability**
- **Reduce Complexity of hardware design**
- **Reduce power (remember: $P = ½CV^2F$ and $V \propto F \rightarrow P \propto CF^3$)**



History

Historically, parallel architectures tied to programming models

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



Uncertainty of direction paralyzed parallel software development!

Types of Parallelism

Instruction Level Parallelism

- Different instructions within a stream can be executed in parallel
- Pipelining, out-of-order execution, speculative execution, VLIW
- Dataflow

Data Parallelism

- Different pieces of data can be operated on in parallel
- SIMD: Vector processing, array processing
- Systolic arrays, streaming processors

Task Level Parallelism

- Different "tasks/threads" can be executed in parallel
- Multithreading
- Multiprocessing (multi-core)

Flynn's Taxonomy of Computers

Mike Flynn, "Very High-Speed Computing Systems," 66

SISD: Single instruction operates on single data element

SIMD: Single instr operates on multiple data elements

- Array processor
- Vector processor

MISD: Multiple instrs operate on single data element

- Closest form?: systolic array processor, streaming processor

MIMD: Multiple instructions operate on multiple data elements (multiple instruction streams)

- Multiprocessor
- Multithreaded processor

Today

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 (hw or sw)

Compilers, libraries and OS are crucial bridges

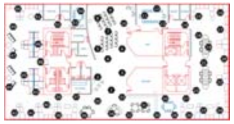
Convergence crosses parallel architectures to include what historically were distributed systems.

Concurrent Systems

Embedded-Physical Distributed

Claytronics

Sensor Networks



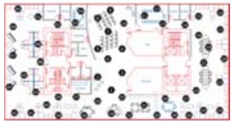
Geographically Distributed

Concurrent Systems

Embedded-Physical Distributed

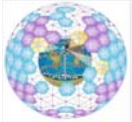
Claytronics

Sensor Networks



Geographically Distributed

Internet



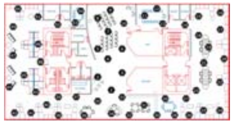
Power Grid

Concurrent Systems

Embedded-Physical Distributed

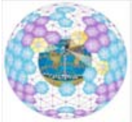
Claytronics

Sensor Networks



Geographically Distributed

Internet



Power Grid

Cloud Computing

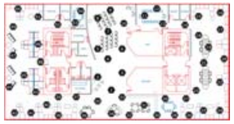
EC2
Tashi

Concurrent Systems

Embedded-Physical Distributed

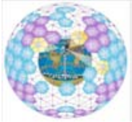
Claytronics

Sensor Networks



Geographically Distributed

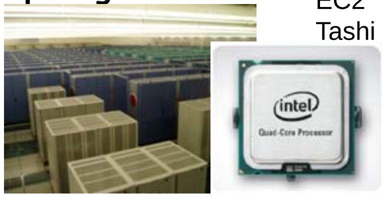
Internet



Power Grid

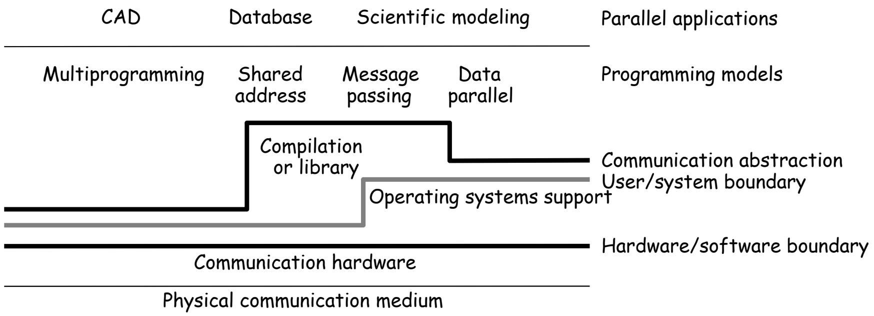
Cloud Computing

Parallel



EC2
Tashi

Modern Layered Framework



CAD
Database
Scientific modeling
Parallel applications

Multiprogramming
Shared address
Message passing
Data parallel
Programming models

Compilation or library
Communication abstraction
User/system boundary
Operating systems support

Hardware/software boundary
Communication hardware
Physical communication medium

Programming Model

What programmer uses in coding applications

Specifies communication and synchronization

Examples:

- **Multiprogramming**: no communication or synch. at program level
- ***Shared address space***: like bulletin board
- ***Message passing***: like letters or phone calls, explicit point to point
- ***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 hw, or via OS, or via user sw

Lot of debate about what to support in sw and gap between layers

Today:

- Hw/sw 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:

- Comm. primitives exposed to user-level by hw and system-level sw

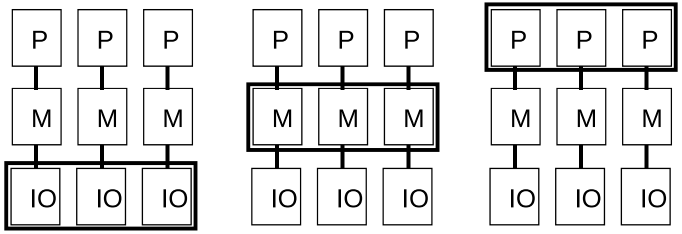
Implementation:

- Organizational structures that implement the primitives: hw or OS
- How optimized are they? How integrated into processing node?
- Structure of network

Goals:

- Performance
- Broad applicability
- Programmability
- Scalability
- Low Cost

Where Communication Happens



Join At:	I/O (Network)	Memory	Processor
Program With:	Message Passing	Shared Memory	(Dataflow/Systolic), Single-Instruction Multiple-Data (SIMD) ==> Data Parallel

Evolution of Architectual Models

Historically, machines tailored to programming models

- Programming model, communication abstraction, and machine organization lumped together as the "architecture"

Evolution helps understand convergence

- Identify core concepts

Most Common Models:

- Shared Address Space, Message Passing, Data Parallel

Other Models:

- Dataflow, Systolic Arrays

Examine programming model, motivation, intended applications, and contributions to convergence

Shared Address Space Architectures

Any processor can directly reference any memory location

- Communication occurs implicitly as result of loads and stores

Convenient:

- Location transparency
- Similar programming model to time-sharing on uniprocessors
  - Except processes run on different processors
  - Good throughput on multiprogrammed workloads

Naturally provided on wide range of platforms

- History dates at least to precursors of mainframes in early 60s
- Wide range of scale: few to hundreds of processors

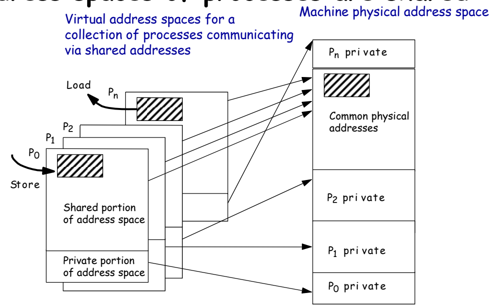
Popularly known as *shared memory* machines or model

- Ambiguous: memory may be physically distributed among processors

Shared Address Space Model

Process: virtual address space plus one or more threads of control

Portions of address spaces of processes are shared

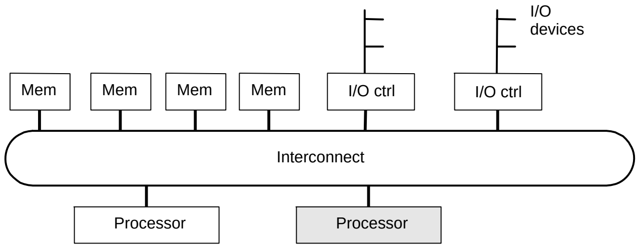

Machine physical address space
Virtual address spaces for a collection of processes communicating via shared addresses
Load
P_n
P_2
P_1
P_0
Store
Shared portion of address space
Private portion of address space
P_n private
Common physical addresses
P_2 private
P_1 private
P_0 private

- Writes to shared address visible to other threads, processes
- Natural extension of uniprocessor model: conventional memory operations for comm.; special atomic operations for synchronization
- OS uses shared memory to coordinate processes

Communication Hardware

Also a natural extension of a uniprocessor

Already have processor, one or more memory modules and I/O controllers connected by hardware interconnect of some sort



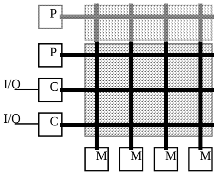
Memory capacity increased by adding modules, I/O by controllers

- Add processors for processing!
- For higher-throughput multiprogramming, or parallel programs

History

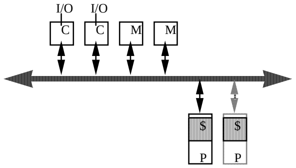
"Mainframe" approach:

- Motivated by multiprogramming
- Extends crossbar used for mem bw and I/O
- Originally processor cost limited to small scale
  - later, cost of crossbar
- Bandwidth scales with *p*
- High incremental cost; use multistage instead

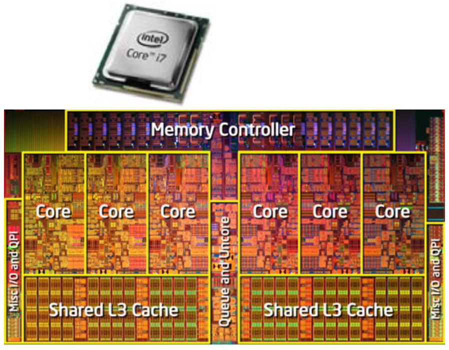


"Minicomputer" approach:

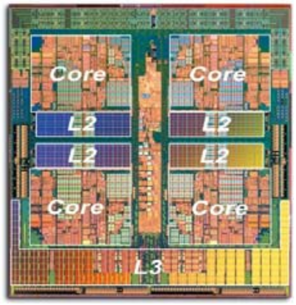
- Almost all microprocessor systems have bus
- Motivated by multiprogramming, TP
- Used heavily for parallel computing
- Called symmetric multiprocessor (SMP)
- Latency larger than for uniprocessor
- Bus is bandwidth bottleneck
  - caching is key: *coherence problem*
- Low incremental cost



Recent x86 Examples



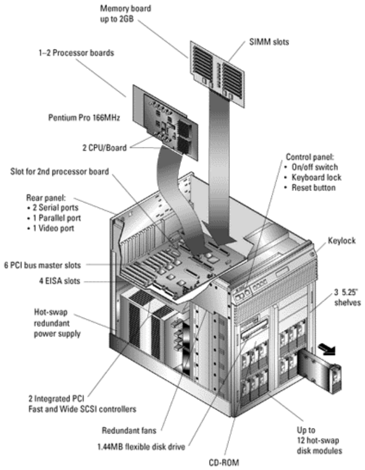
Intel's Core i7-980X

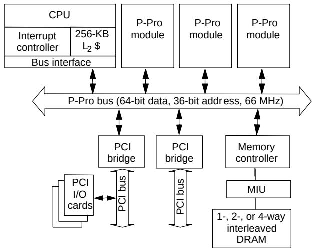


AMD's Quad-Core Phenom II

- Highly integrated, commodity systems
- On-chip: low-latency, high-bandwidth communication via shared cache
- Current scale = 4-6 processors

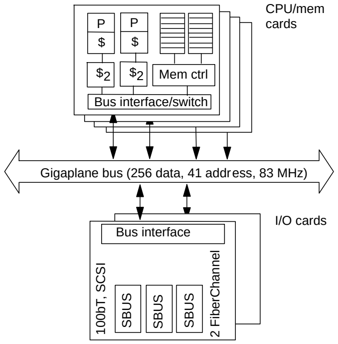
Example: Intel Pentium Pro Quad





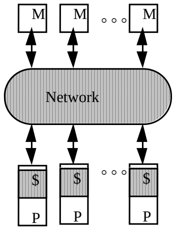
- All coherence and multiprocessing glue in processor module
- Highly integrated, targeted at high volume
- Low latency and bandwidth

Example: SUN Enterprise

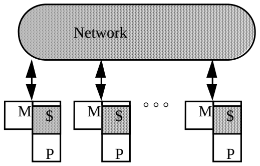


- 16 cards of either type: processors + memory, or I/O
- All memory accessed over bus, so symmetric
- Higher bandwidth, higher latency bus

Scaling Up



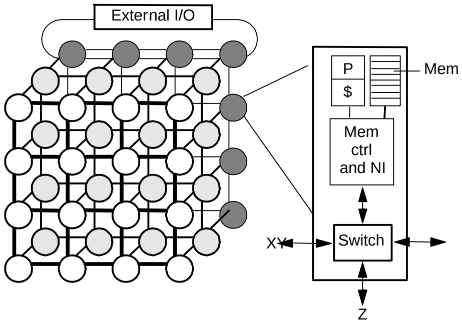
"Dance hall"



Distributed memory

- **Problem is interconnect: cost (crossbar) or bandwidth (bus)**
- **Dance-hall: bandwidth still scalable, but lower cost than crossbar**
  - latencies to memory uniform, but **uniformly large**
- **Distributed memory or non-uniform memory access (NUMA)**
  - Construct shared address space out of simple message transactions across a general-purpose network (e.g. read-request, read-response)
- **Caching shared (particularly nonlocal) data?**

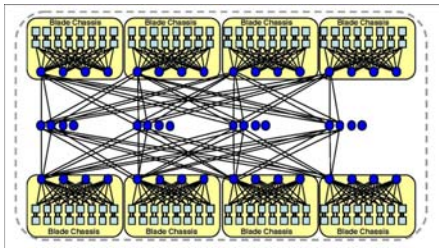
Example: Cray T3E



- **Scale up to 1024 processors, 480MB/s links**
- **Memory controller generates comm. request for nonlocal references**
- **No hardware mechanism for coherence (SGI Origin etc. provide this)**

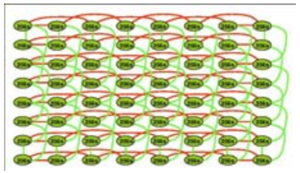
Example: SGI Altix UV 1000

Blacklight at the PSC (4096 cores)



256 socket (2048 core) fat-tree
(this size is doubled in Blacklight via a torus)

- **Scales up to 131,072 cores**
- **15GB/sec links**
- **Hardware cache coherence**



8x8 torus

Message Passing Architectures

Complete computer as building block, including I/O

- **Communication via explicit I/O operations**

Programming model:

- **directly access only private address space (local memory)**
- **communicate via explicit messages (send/receive)**

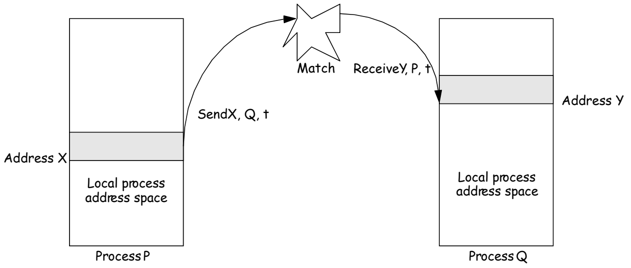
High-level block diagram similar to distributed-mem SAS

- **But comm. integrated at IO level, need not put into memory system**
- **Like networks of workstations (clusters), but tighter integration**
- **Easier to build than scalable SAS**

Programming model further from basic hardware ops

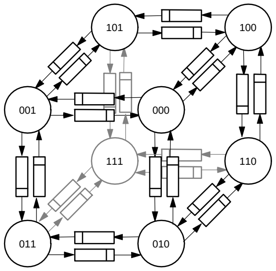
- **Library or OS intervention**

Message Passing Abstraction



- **Send specifies buffer to be transmitted and receiving process**
- **Recv specifies sending process and application storage to receive into**
- **Memory to memory copy, but need to name processes**
- **Optional tag on send and matching rule on receive**
- **User process names local data and entities in process/tag space too**
- **In simplest form, the send/recv match achieves pairwise synch event**
  - Other variants too
- **Many overheads: copying, buffer management, protection**

Evolution of Message Passing



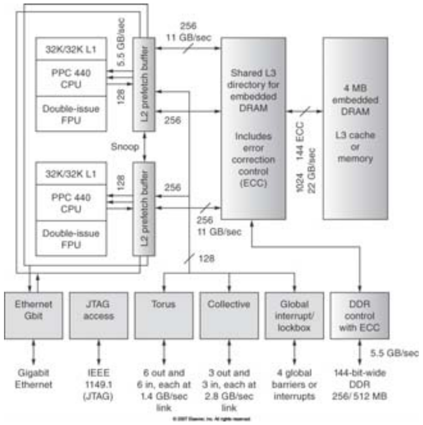
Early machines: FIFO on each link

- **Hardware close to programming model**
  - synchronous ops
- **Replaced by DMA, enabling non-blocking ops**
  - Buffered by system at destination until recv

Diminishing role of topology

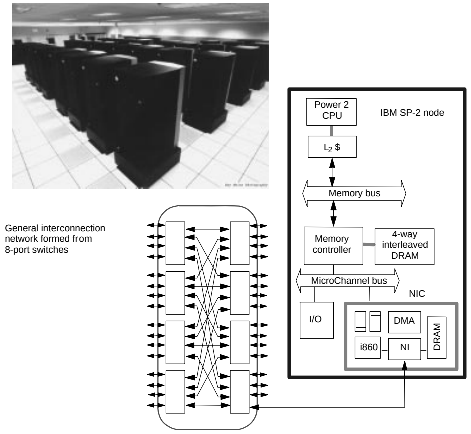
- **Store & forward routing: topology important**
- **Introduction of pipelined routing made it less so**
- **Cost is in node-network interface**
- **Simplifies programming**

Example: IBM Blue Gene/L



Nodes: 2 PowerPC 400s; everything except DRAM on one chip

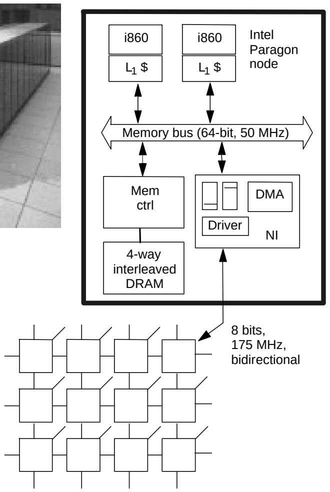
Example: IBM SP-2



- **Made out of essentially complete RS6000 workstations**
- **Network interface integrated in I/O bus (bw limited by I/O bus)**

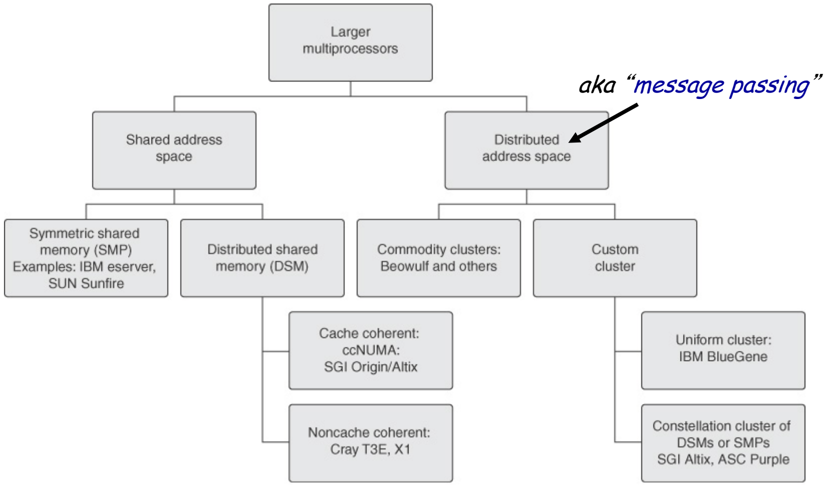
Example: Intel Paragon

Sandia' s Intel Paragon XP/S-based Supercomputer



2D grid network with processing node attached to every switch

Taxonomy of Common Large-Scale SAS and MP Systems



Larger multiprocessors
Shared address space
Distributed address space
aka "message passing"
Symmetric shared memory (SMP) Examples: IBM eserver, SUN Sunfire
Distributed shared memory (DSM)
Commodity clusters: Beowulf and others
Custom cluster
Cache coherent: ccNUMA: SGI Origin/Altix
Noncache coherent: Cray T3E, X1
Uniform cluster: IBM BlueGene
Constellation cluster of DSMs or SMPs SGI Altix, ASC Purple

© 2007 Elsevier, Inc. All rights reserved.

Toward Architectural Convergence

Evolution and role of software have blurred boundary

- Send/recv supported on SAS machines via buffers
- Can construct global address space on MP using hashing
- Page-based (or finer-grained) shared virtual memory

Hardware organization converging too

- Tighter NI integration even for MP (low-latency, high-bandwidth)
- At lower level, even hardware SAS passes hardware messages

Even clusters of workstations/SMPs are parallel systems

- Emergence of fast system area networks (SAN)

Programming models distinct, but organizations converging

- Nodes connected by general network and communication assists
- Implementations also converging, at least in high-end machines

Data Parallel Systems

Programming model:

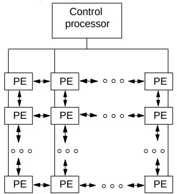
- Operations performed in parallel on each element of data structure
- Logically single thread of control, performs sequential or parallel steps
- Conceptually, a processor associated with each data element

Architectural model:

- Array of many simple, cheap processors with little memory each
  - Processors don't sequence through instructions
- Attached to a control processor that issues instructions
- Specialized and general communication, cheap global synchronization

Original motivation:

- Matches simple differential equation solvers
- Centralize high cost of instruction fetch & sequencing



Application of Data Parallelism

- Each PE contains an employee record with his/her salary

```
If salary > 100K then
   salary = salary *1.05
else
   salary = salary *1.10
```

- Logically, the whole operation is a single step
- Some processors enabled for arithmetic operation, others disabled

Other examples:

- Finite differences, linear algebra, ...
- Document searching, graphics, image processing, ...

Some examples:

- Thinking Machines CM-1, CM-2 (and CM-5)
- Maspar MP-1 and MP-2,

Evolution and Convergence

Rigid control structure (SIMD in Flynn taxonomy)

- SISD = uniprocessor, MIMD = multiprocessor

Popular when cost savings of centralized sequencer high

- 60s when CPU was a cabinet; replaced by vectors in mid-70s
- Revived in mid-80s when 32-bit datapath slices just fit on chip
- No longer true with modern microprocessors

Other reasons for demise

- Simple, regular applications have good locality, can do well anyway
- Loss of applicability due to hardwiring data parallelism
  - MIMD machines as effective for data parallelism and more general

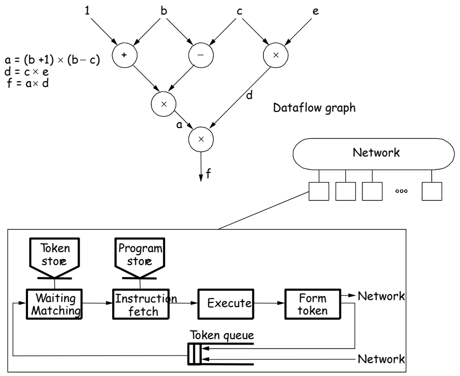
Programming model converges to SPMD (single program multiple data)

- Contributes need for fast global synchronization
- Structured global address space, implemented with either SAS or MP

Dataflow Architectures

Represent computation as a graph of essential dependences

- Logical processor at each node, activated by availability of operands
- Message (tokens) carrying tag of next instruction sent to next processor
- Tag compared with others in matching store; match fires execution



Evolution and Convergence

Key characteristics:

- Ability to name operations, synchronization, dynamic scheduling

Problems:

- Operations have locality across them, useful to group together
- Handling complex data structures like arrays
- Complexity of matching store and memory units
- Exposes too much parallelism (?)

Converged to use conventional processors and memory

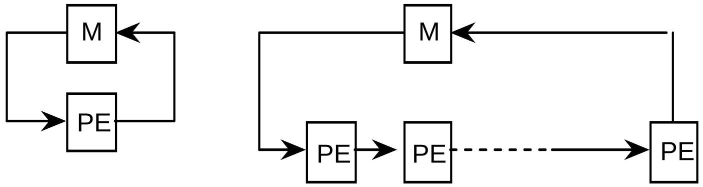
- Support for large, dynamic set of threads to map to processors
- Typically shared address space as well
- But separation of programming model from hardware (like data parallel)

Lasting contributions:

- Integration of communication with thread (handler) generation
- Tightly integrated communication and fine-grained synchronization
- Remained useful concept for software (compilers etc.)

Systolic Architectures

- Replace single processor with array of regular processing elements
- Orchestrate data flow for high throughput with less memory access



Different from pipelining:

- Nonlinear array structure, multidirection data flow, each PE may have (small) local instruction and data memory

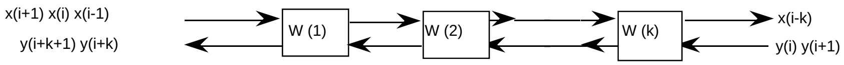
Different from SIMD: each PE may do something different

Initial motivation: VLSI enables inexpensive special-purpose chips

Represent algorithms directly by chips connected in regular pattern

Systolic Arrays (Cont)

Example: Systolic array for 1-D convolution

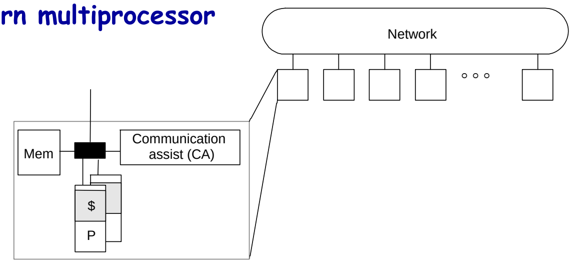


$$y(i) = \sum_{j=1}^{k} w(j)*x(i-j)$$

- Practical realizations (e.g. iWARP) use quite general processors
  - Enable variety of algorithms on same hardware
- But dedicated interconnect channels
  - Data transfer directly from register to register across channel
- Specialized, and same problems as SIMD
  - General purpose systems work well for same algorithms (locality etc.)

Convergence: General 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...

Fundamental Design Issues

Understanding Parallel Architecture

Traditional taxonomies not very useful

Programming models not enough, nor hardware structures

- Same one 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 progr. 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?
- *Replication:* How are data replicated to reduce communication?
- *Communication Cost:* Latency, bandwidth, overhead, occupancy

Understand at programming model first, since that sets requirements

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 dependences 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

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 (discussed later)

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 via *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
- Send and receive can provide pt-to-pt synch between processes
- Mutual exclusion inherent

Can construct global address space:

- Process number + address within process address space
- But no direct operations on these names

Design Issues Apply at All Layers

Programming model's position provides constraints/goals for 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

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 v-to-p mappings, no software layers

Hardware supports independent physical address spaces

- Can provide SAS through OS, so 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

Naming and Operations (Cont)

Example: Implementing Message Passing

Direct support at hardware interface

- But match and buffering benefit from more flexibility

Support at system/user interface or above in software (almost always)

- Hardware interface provides basic data transport (well suited)
- Send/receive built in software for flexibility (protection, buffering)
- Choices at user/system interface:
  - OS each time: expensive
  - OS sets up once/infrequently, then little software involvement each time
- 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

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 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

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 $\propto$ 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 will focus on communication or data transfer across nodes

Communication Cost Model

Communication Time per Message

= *Overhead* + *Assist Occupancy* + *Network Delay* + *Size*/*Bandwidth* + *Contention*

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

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

Comm Cost = frequency * (Comm time - overlap)

General model for data transfer: applies to cache misses too

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

Are We Asking Right Questions?

- **Programming model:**
  - SAS/MP/DP?
  - Is this what should be exposed to the programmer?
- **Design issues:**
  - Naming/operations/ordering/replication/communication
  - Should any of this be exposed to programmer?

- **Alternative Approach?**

Holy grail is to design a system that

- Is easy to program
- Yields good performance (and efficiency)
- Can easily scale (adding more resources improves performance)

Are we ready for declarative programming languages?

Recap

Exotic designs have contributed much, but given way to convergence

- Push of technology, cost and application performance
- Basic processor-memory architecture is the same
- Key architectural issue is in communication architecture

Fundamental design issues:

- Functional: naming, operations, ordering
- Performance: organization, replication, performance characteristics

Design decisions driven by workload-driven evaluation

- Integral part of the engineering focus

Performance Metrics

Parallel Speedup

Time to execute the program with 1 processor
divided by
Time to execute the program with N processors

Parallel Speedup Example

$a4x^4 + a3x^3 + a2x^2 + a1x + a0$

Assume each operation 1 cycle, no communication cost, each op can be executed in a different processor

How fast is this with a single processor?

- Assume no pipelining or concurrent execution of instructions

How fast is this with 3 processors?

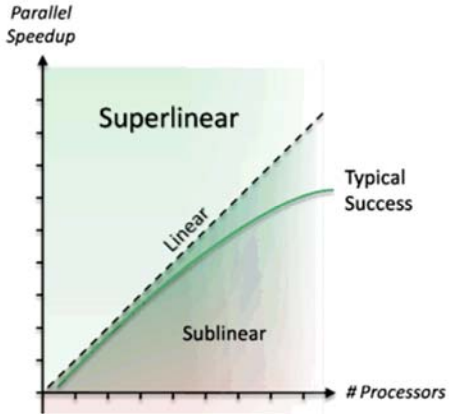
Takeaway

To calculate parallel speedup fairly you need to use the best known algorithm for each system with N processors

If not, you can get superlinear speedup

Superlinear Speedup

Can speedup be greater than P with P processing elements?



Consider:

- Cache effects
- Memory effects
- Working set

Happens in two ways:

- Unfair comparisons
- Memory effects

Utilization, Redundancy, Efficiency

Traditional metrics

- Assume all P processors are tied up for parallel computation

Utilization: How much processing capability is used

- U = (# Operations in parallel version) / (processors x Time)

Redundancy: how much extra work is done

- R = (# of operations in parallel version) / (# operations in best uni-processor algorithm version)

Efficiency

- E = (Time with 1 processor) / (processors x Time with P procs)
- E = U/R

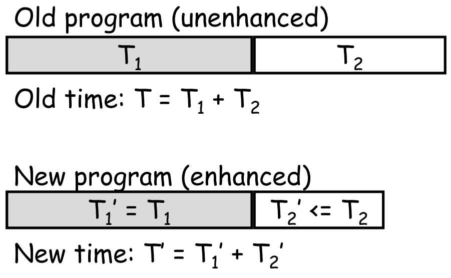
Amdahl's law

You plan to visit a friend in Normandy France and must decide whether it is worth it to take the Concorde SST ($3,100) or a 747 ($1,021) from NY to Paris, assuming it will take 4 hours Pgh to NY and 4 hours Paris to Normandy.

	time NY->Paris	total trip time	speedup over 747
747	8.5 hours	16.5 hours	1
SST	3.75 hours	11.75 hours	1.4

Taking the SST (which is 2.2 times faster) speeds up the overall trip by only a factor of 1.4!

Amdahl's law (cont)



Old program (unenhanced)

T_1 | T_2

Old time: $T = T_1 + T_2$

New program (enhanced)

$T_1' = T_1$ | $T_2' <= T_2$

New time: $T' = T_1' + T_2'$

T_1 = time that can NOT be enhanced.

T_2 = time that can be enhanced.

T_2' = time after the enhancement.

Speedup: $S_{overall} = T / T'$

Amdahl's law (cont)

Two key parameters:

$F_{enhanced} = T_2 / T$ (fraction of original time that can be improved)
$S_{enhanced} = T_2 / T_2'$ (speedup of enhanced part)

$$\begin{aligned} T' &= T_1' + T_2' = T_1 + T_2' = T(1-F_{enhanced}) + T_2' \\ &= T(1-F_{enhanced}) + (T_2/S_{enhanced}) && \text{[by def of } S_{enhanced}\text{]} \\ &= T(1-F_{enhanced}) + T(F_{enhanced} / S_{enhanced}) && \text{[by def of } F_{enhanced}\text{]} \\ &= T((1-F_{enhanced}) + F_{enhanced}/S_{enhanced}) \end{aligned}$$

Amdahl's Law:

$$S_{overall} = T / T' = 1/((1-F_{enhanced}) + F_{enhanced}/S_{enhanced})$$

Key idea: Amdahl's law quantifies the general notion of diminishing returns. It applies to any activity, not just computer programs.

Amdahl's law (cont)

Trip example: Suppose that for the New York to Paris leg, we now consider the possibility of taking a rocket ship (15 minutes) or a handy rip in the fabric of space-time (0 minutes):

	time NY->Paris	total trip time	speedup over 747
747	8.5 hours	16.5 hours	1
SST	3.75 hours	11.75 hours	1.4
rocket	0.25 hours	8.25 hours	2.0
rip	0.0 hours	8 hours	2.1

Amdahl's law (cont)

Useful corollary to Amdahl's law:

- $1 <= S_{overall} <= 1 / (1 - F_{enhanced})$

$F_{enhanced}$	Max $S_{overall}$	$F_{enhanced}$	Max $S_{overall}$
0.0	1	0.9375	16
0.5	2	0.96875	32
0.75	4	0.984375	64
0.875	8	0.9921875	128

Moral: It is hard to speed up a program.

Moral++ : It is easy to make premature optimizations.

Caveats of Parallelism (I): Amdahl's Law

Amdahl's Law

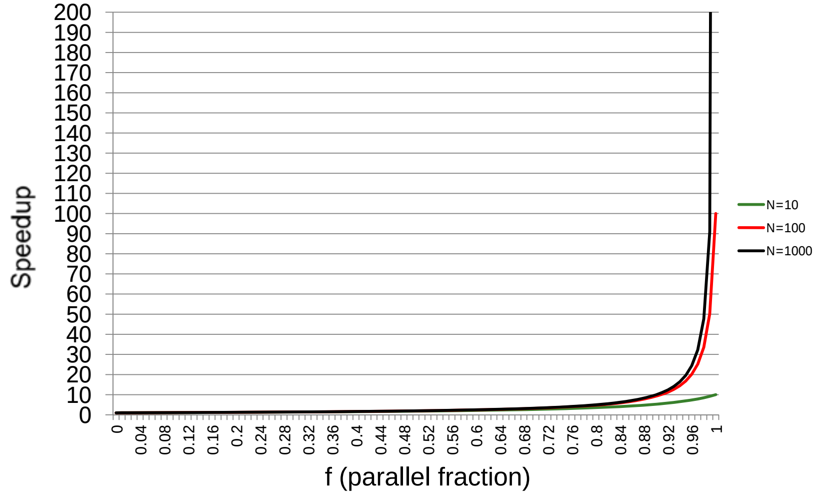
- **f: Parallelizable fraction of a program**
- **P: Number of processors**

$$\text{Speedup} = \frac{1}{1 - f + \frac{f}{P}}$$

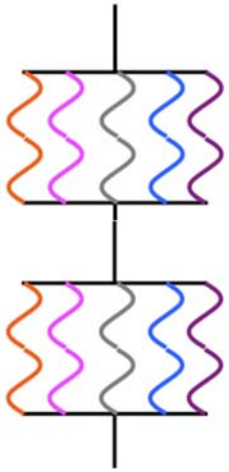
- **Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," AFIPS 1967.**

Maximum speedup limited by serial portion: Serial bottleneck

Sequential Bottleneck



Why the Sequential Bottleneck?



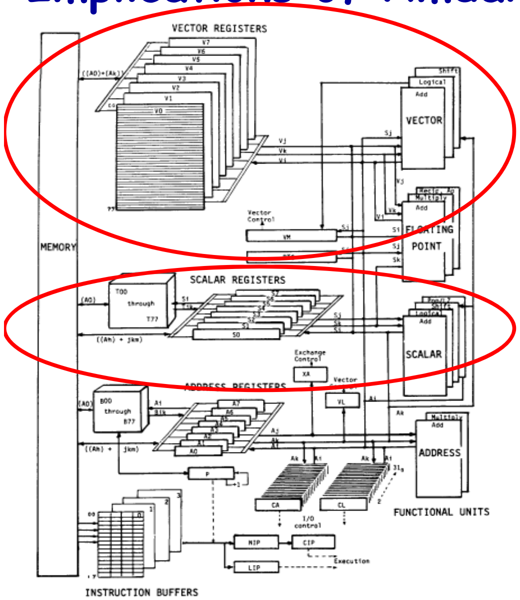
Parallel machines have the sequential bottleneck

Main cause: Non-parallelizable operations on data (e.g. non-parallelizable loops)

```
for ( i = 0 ; i < N; i++)
    A[i] = (A[i] + A[i-1]) / 2
```

Single thread prepares data and spawns parallel tasks (usually sequential)

Implications of Amdahl's Law on Design



- CRAY-1
- Russell, "The CRAY-1 computer system," CACM 1978.
- Well known as a fast vector machine
  - 8 64-element vector registers
- The fastest SCALAR machine of its time!
  - Reason: Sequential bottleneck!

Caveats of Parallelism (II)

Amdahl's Law

- f: Parallelizable fraction of a program
- P: Number of processors

$$\text{Speedup} = \frac{1}{1 - f + \frac{f}{P}}$$

- Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," AFIPS 1967.

Maximum speedup limited by serial portion: Serial bottleneck

Parallel portion is usually not perfectly parallel

- Synchronization overhead (e.g., updates to shared data)
- Load imbalance overhead (imperfect parallelization)
- Resource sharing overhead (contention among N processors)

Bottlenecks in Parallel Portion

Synchronization: Operations manipulating shared data cannot be parallelized

- Locks, mutual exclusion, barrier synchronization
- Communication: Tasks may need values from each other
- Causes thread serialization when shared data is contended

Load Imbalance: Parallel tasks may have different lengths

- Due to imperfect parallelization or microarchitectural effects
- Reduces speedup in parallel portion

Resource Contention: Parallel tasks can share hardware resources, delaying each other

- Replicating all resources (e.g., memory) expensive
- Additional latency not present when each task runs alone

Difficulty in Parallel Programming

Little difficulty if parallelism is natural

- "Embarrassingly parallel" applications
- Multimedia, physical simulation, graphics
- Large web servers, databases?

Big difficulty is in

- Harder to parallelize algorithms
- Getting parallel programs to work correctly
- Optimizing performance in the presence of bottlenecks

Much of parallel computer architecture is about

- Designing machines that overcome the sequential and parallel bottlenecks to achieve higher performance and efficiency
- Making programmer's job easier in writing correct and high-performance parallel programs

Bottlenecks in the Parallel Portion

Amdahl's Law does not consider these

How do synchronization (e.g., critical sections), and load imbalance, resource contention affect parallel speedup?

Can we develop an intuitive model (like Amdahl's Law) to reason about these?

Need better analysis of critical sections in real programs