Parallel Processing and Multiprocessors

why parallel processing?

types of parallel processors

cache coherence & memory consistency models

Why Parallel Processing

go past physical limits of uniprocessing (speed of light)

pros: performance

- cost-effectiveness (commodity parts)
- smooth upgrade path
- fault tolerance

cons: difficult to parallelize applications

- often uses compiler parallelization
- parallel program development
- IT IS THE SOFTWARE, stupid!

Amdahl's Law

speedup = $1/(frac_{enhanced}/speedup_{enhanced} + 1 - frac_{enhanced})$

speedup of 80 with 100 processors

=> $frac_{parallel} = 0.9975$

only 0.25% work can be serial

may help: problems where parallel parts scale faster than serial

- $O(n^2)$ parallel vs. $O(n)$ serial

challenge: long latencies (often several microsecs)

- achieve data locality in some fashion

Application Domains

Parallel Processing - true parallelism in one job

- data may be tightly shared

OS - large parallel program that runs a lot of time

- typically hand-crafted and fine-tuned
- data more loosely shared
- typically locked data structures at differing granularities

transaction processing - parallel among independent transactions

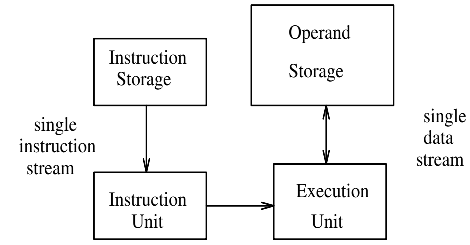
- throughput oriented parallelism

Types

Flynn Taxonomy

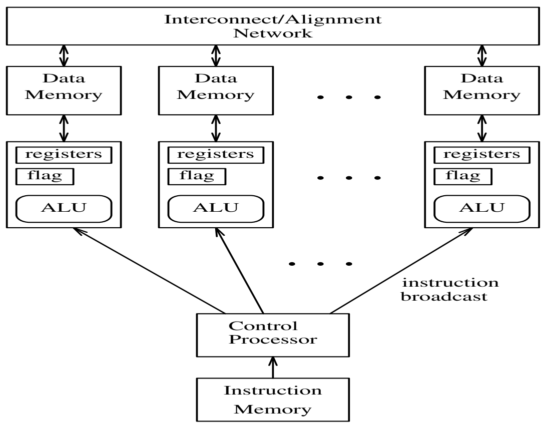
- 1966, not all encompassing but simple
- single or multiple instruction streams (SI or MI)
- single or mulitple data streams (SD or MD)
- SISD - uniprocessor
- SIMD - like vector
- MISD - few practical examples
- MIMD - multiprocessors - most common, very flexible

Single Instruction Single Data (SISD)



your basic uniprocessor

Single Instruction Multiple Data (SIMD)



Single Instruction Multiple Data (SIMD)

intrs and data usually separated

leads to data parallel programming model

works best for very regular, loop-oriented numerical problems

not for $1 million commercial database server

automatic parallelization can work

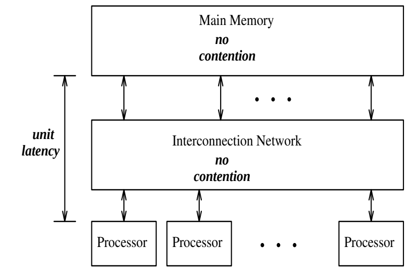
Multiple Instruction Multiple Data (MIMD)

most flexible and of most interest to us

may become the general purpose computer

automatic parallelization more difficult

Perfection: PRAM



parallel RAM - theoretical model

fully shared memory - unit latency

no contention, no need to exploit locality

Perfection not achievable

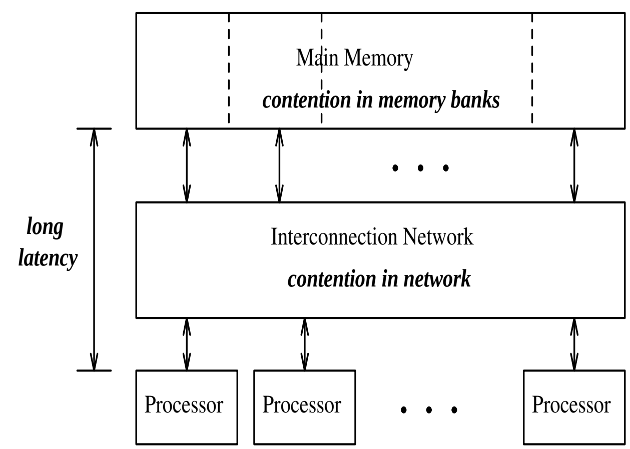
latencies grow as the system size grows

bandwidths restricted by memory organization and interconnect

dealing with reality leads to division between

- UMA and NUMA

UMA: uniform memory access



UMA: uniform memory access

latencies are the same

- but may be high

data placement unimportant

latency gets worse as system grows => scalability issues

typically used in small MPs only

contention restricts bandwidth

caches are often allowed in UMA systems

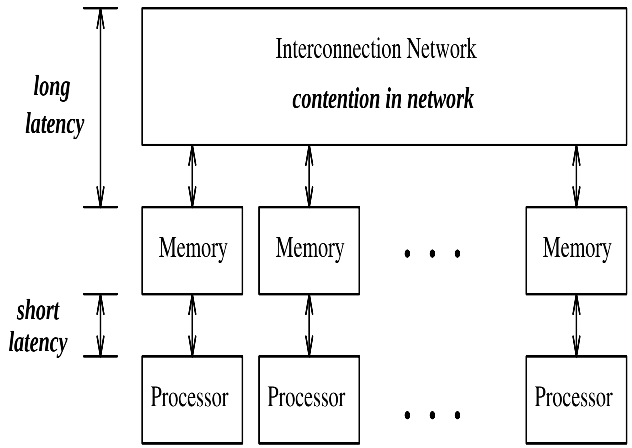
Caches

another way of tackling latency/bandwidth problems

holds recently used data

BUT cache coherence problems

NUMA: non-uniform memory access



NUMA: non-uniform memory access

latency low to local memory

latency much higher to remote memories

performance very sensitive to data placement

bandwidth to local may be higher

contention in network and for memories

NUMA Multiprocessors

shared memory

- one logical address space
- can be treated as shared memory
- use synchronization (e.g., locks) to access shared data

multicomputers (message passing)

- each processor has its own memory address space
- use message passing for communication

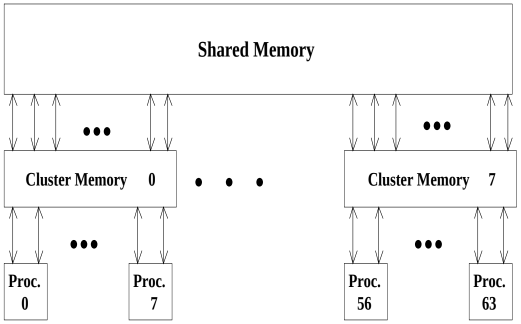
Clustered Systems

small UMA nodes in large UMA systems

hybrid of sorts

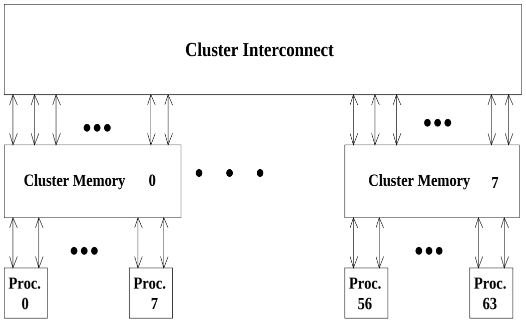
note: ambiguity of the term "cluster"

Cluster types



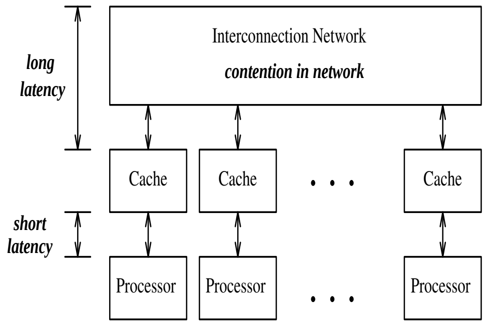
globally shared memoery - Illinois Cedar

Cluster types



No global memory - Stanford Dash, Wisconsin Typhoon

COMA: cache only memory architecture



effectively a form of NUMA

- caches only causes data to migrate naturally

Process communication

for most interesting parallel applications

- parallel processes (tasks) need to communicate

communication method leads to another division

- message passing
- shared memory

Shared memory vs message passing

shared memory

+ conceptually compatible with uniprocessors/small MPs
+ ease of programming if communication complex/dynamic
+ lower latency communicating small data items
+ hardware controlled sharing allows automatic data motion

- programs use loads/stores

Shared memory vs message passing

message passing

+ simpler hardware
+ communication pattern explicit and precise
+ but they MUST be explicit and precise
+ least common denominator

- shared memory MP can emulate message passing easily
- programs use sends/receives

Multicomputers: message passing

Architecture: communication via messages

- each node is a computer
- each computer has its own address space
- each node has a unique id

effect of latency

- messages pipelined through the network
- latency = time per node x #nodes + length/bandwidth
- long messages => first term irrelevant => large messages
- => infrequent communication, large message => coarse

Early Multicomputers

Cosmic Cube, IPSC/1, Ametek, N-Cube

interconnect - binary n-cube

- software controlled routing

problems

- n cubes scale theoretically but some practical problems
- software routing has high overhead
- store and forward routing
    - latency depends on lenght, uses storage at nodes
- higher dimension reduce #nodes but reduce link b/w

Second generation

use hardware routers

- overhead cut greatly
- lowers overhead per node
- => tends to make latency insensitive to distance
- => simplified programming

use lower dimension

- easy to construct
- helps link b/w
- wormhole routing => node overhead insensitive to length

(Not Too) Recent Parallel Systems

Machine	Comm mech	icn	#cpus	remote access (microsec
SPARCcenter	shared	bus	<=20	1
SGI Challenge	shared	bus	<= 32	1
Cray T3D	shared	3-d torus	64-1024	1
Convex SPP	shared	X-bar/ring	8-64	2
KSR-1	shared	bus/ring	32-	2-6
TMC CM-5	mesg	fat tree	64-1024	10
Intel Paragon	mesg	2-d mesh	32-2048	10-30
IBM SP-1	mesg	multistage	32-256	30-100

Shared memory MPs: cache coherence

e.g.,

- proc 1 reads A
- proc 2 reads A
- proc 1 writes A
- now proc 2 has stale data regardless of write-thru/-back

informally - method to make memory consistent despite caches

- caches can be viewed as large buffers
- with potentially very large delays

Shared memory MPs: cache coherence

cache coherence suggests absolute time scale

- not necessary
- what is required is appearance of coherence not absolute
- Bart Simpson's famous words -
- "nobody saw me do it so I didn't do it!"
- e.g. temporary incoherence between
    - writeback cache and memory ok

Causes of Incoherence

sharing of writeable data

- cause most commonly considered

process migration

- can occur even if independent jobs are executing

I/O

- can be fixed by OS cache flushes

Solutions to Coherence Problems

Disallow private caches - put caches next to memory

make shared-data non-cacheable - simplest software solution

have software flush at strategic times - e.g., after critical sections

use global table - does not scale

use bus-based snoopy cache - small scale e.g., Pentium Quads

use distributed memory-based directories - large scale

- like a phone book
- e.g., SGI Origin

Bus-based Coherence

typically write-back caches are used to minimize traffic

typical states

- invalid
- shared
- exclusive

consider state changes for

- read hit, read miss
- write hit, write miss

snooping: ALL caches monitor ALL traffic on the bus

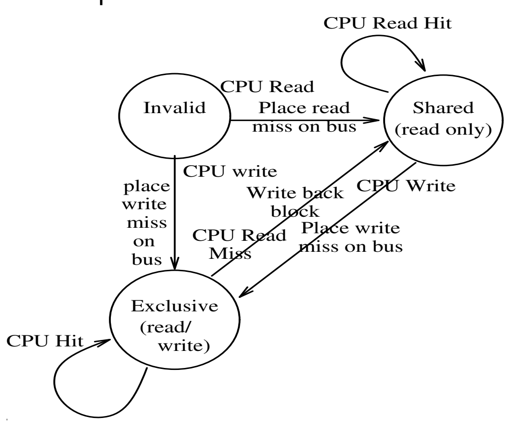
Bus-based Coherence

writes?

- invalidate copies in other caches
- => invalidation protocols
- OR
- update copies in other caches
- => update protocols

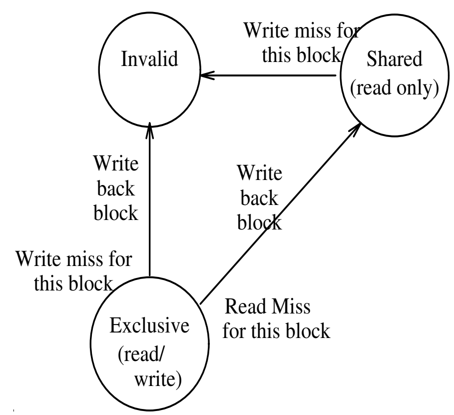
Typical Write Invalidate Protocol

State transitions for processor actions



Typical Write Invalidate Protocol

state transitions for cache actions



Bus-based Protocols: Performance

Misses: 3C model

- capacity
- compulsory
- conflict
- coherence

coherence misses: additional misses due to coherence protocol

as processors are added

- capacity misses decrease (total cache size increases)
- coherence misses increase (more communication)

Bus-based Protocols: Performance

as cache size is increased

- inherent coherence misses limit overall miss rate

as block size is increased

- less effective than a uniprocessor
  - less spatial locality
  - false sharing
- more bus traffic
  - problem in bus-based MP

Directory-based Cache Coherence

an alternative for large MPs

sharing info in memory directory

states of memory block

- shared - >=1 processor has data and memory is uptodate
- uncached - no processor has data
- exclusive - only one processor has data/memory is old

directory entry tracks which processors has data

- e.g., via a bit vector

Directory-based Cache Coherence

block in uncached state: memory current

- read miss - send data to requester; mark block shared; sharers <- requester
- write miss - send data to requester; mark block exclusive; sharers <- requester

block in shared: memory current

- read miss - send data to requester; sharers +<- requester (add to the list)
- write miss - send data to requester; invalidate all sharers; mark block exclusive; sharers <- requester (

Directory-based Cache Coherence

block in exclusive; memory stale

- read miss - fetch data from owner; update memory; mark data shared; sharers +<- requester (add to prev owner)
- data write-back - mark block uncached; sharers <- empty
- write miss - fetch data from owner; mark block exclusive; sharers <- requester (new owner)

Directory-based Cache Coherence

consider 3 processors - P1 P2 P3

sequence of events

- data in no caches
- all 3 processors do read
- P3 does write
- C3 hits; but does not have ownership

Directory-based Cache Coherence

- C3 makes write request
- memory sends invalidates to C1 and C2
- C1 and C2 invalidate and ack
- memory receives acks; sets line exclusive; sends write permission to C3
- C3 writes to cache; sets line dirty; P3 proceeds

Performance

divide misses into local and remote

miss rate decreases as caches grow larger

but coherence dampens the decrease in miss rate

latency effects

- cache hit: 3 cycles
- local miss: 75+ cycles
- remote miss (home): 150+ cycles
- remote miss (3-hop): 225+ cycles

Research: How do we make all misses look like hits or local miss?

Correctness: Memory Ordering

memory updates may become re-ordered by the memory system

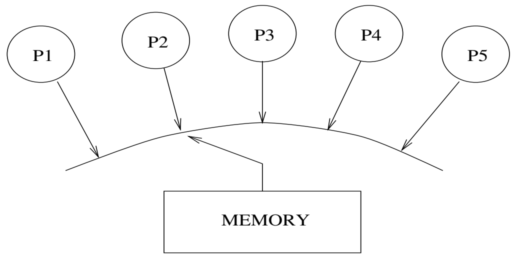
e.g.,

• Proc 1	Proc 2
• A = 0	B =0
• --	--
• A = 1	B =1
• L1: if (B== 0)	L2: if (A ==0)

intuitively impossible for both A and B to be 0

- BUT it can happen if memory operations are reordered

Correctness: Sequential Consistency



"system is sequentially consistent if the result of ANY execution is the same as if the operations of all processors were executed in SOME sequential order and the operations of each individual processor appears in this sequence in the order specified by the program"

Correctness: Memory Consistency Models

Everything in order

- Sequential Consistency (SC from SGI)

Allow stores to be delayed (e.g., write buffers)

- Processor Consistency (PC)
- Total Store Order (TSO from Sun)
- Intel Architecture-32 (IA-32 from Intel)

Allow loads and stores to be re-ordered

- Release Consistency (RC)
- Alpha Memory Order (from Compaq)