Sharing memory in MIMD machines

- Multiprocessors
  - UMA
  - NUMA
- Multiprocessors - distributed shared memory (DSM)
  - motivation
  - consistency models
  - implementation
    - implementation problems
      - granularity
      - page replacement
    - implementation approaches
  - thrashing

Multicomputers and Multiprocessors

- Multiprocessors
  - Any process can use usual load/store operations to access any memory word
  - Complex hardware — bus becomes a bottleneck with more than 10-20 CPUs
  - Simple communication between processes — shared memory locations
  - Synchronization is well-understood, uses classical techniques (semaphores, etc.)
- Multiprocessors (no shared memory)
  - Each CPU has its own private memory
  - Simple hardware with network connection
  - Complex communication — processes have to use message-passing, have to deal with lost messages, blocking, etc.
  - RPCs help, but aren't perfect (no globals, can't pass large data structures, etc.)

Bus-based multiprocessors

- Symmetric Multiprocessor (SMP)
  - Multiple CPUs (2–30), one shared physical memory, connected by a bus
  - Caches must be kept consistent
    - Each CPU has a “snooping” cache, which “snoops” what’s happening on the bus
      - On read hit, fetch data from local cache
      - On read miss, fetch data from memory, and store in local cache
        - Same data may be in multiple caches
      - (Write through) On write, store in memory and local cache
        - Other caches are snooping; if they have that word they invalidate their cache entry
        - After write completes, the memory is up-to-date and the word is only in one cache

NUMA Machines

- NUMA – Non-Uniform Memory Access
- NUMA multiprocessor
  - Multiple CPUs, each with its own memory
  - CPUs share a single virtual memory
  - Accesses to local memory locations are much faster (maybe 10x) than accesses to remote memory locations
  - No hardware caching, so it matters which data is stored in which memory
  - A reference to a remote page causes a hardware page fault, which traps to the OS, which decides whether or not to move the page to the local machine

Distributed Shared Memory (DSM) overview

- Basic idea (Kai Li, 1986)
  - Collection of workstations, connected by a LAN, all sharing a single paged, virtual address space
  - Each page is present on exactly one machine, and a reference to a local page is done in the usual fashion
  - A reference to a remote page causes a hardware page fault, which traps to the OS, which sends a message to the remote machine to get the page; the faulting instruction can then complete

To programmer, DSM machine looks like a conventional shared-memory machine
- Easy to modify old programs
  - Poor performance — lots of pages being sent back and forth over the network

Advantages of DSM

- Simpler abstraction - programmer does not have to worry about data movement, may be easier to implement than RPC since the address space is the same
- Easier portability - sequential programs can in principle be run directly on DSM systems
- Possibly better performance
  - Locality of data - data moved in large blocks which helps programs with good locality of reference
  - On-demand data movement
  - Larger memory space - no need to do paging on disk
- Flexible communication - no need for sender and receiver to exist, can join and leave DSM system without affecting the others
- Process migration simplified - one process can easily be moved to a different machine since they all share the address space
Comparison of shared memory systems

**SMP (Single Bus Multiprocessor)**
- Hardware access to all of memory, hardware caching of blocks
- Managed by MMU

**NUMA (Non-Uniform Memory Access)**
- Hardware access to all of memory, software caching of pages
- Managed by OS

**Distributed Shared Memory (DSM)**
- Software access to remote memory, software caching of pages
- Managed by language runtime system

Maintaining memory coherency

- DSM systems allow concurrent access to shared data
- Concurrency may lead to unexpected results - what if the read does not return the value stored by the most recent write (write did not propagate)?
- Memory is coherent if the value returned by the read operation is always the value the programmer expected
- To maintain coherency of shared data a mechanism that controls (and synchronizes) memory accesses is used.
- This mechanism only allows a restricted set of memory access orderings
- Memory consistency model - the set of allowable memory access orderings

Consistency models

- **Notation**: $ij[a]$ - operation number $i$ performed by process $P_j$ on memory element $a$
- **Strict consistency** (strongest model)
  - Value returned by a read operation is always the same as the value written by the most recent write operation
  - Writes become instantly available to all processes
  - Requires absolute global time to correctly order operations — not possible
- **Sequential consistency** (Lamport 1979)
  - All processes see all memory access operations in the same order
  - Interleaving of operations doesn’t matter, if all processes see the same ordering
  - Read operation may not return result of most recent write operation!
  - Running a program twice may give different results each time
  - If order matters, use semaphores!

Consistency models (cont.)

- **Processor consistency** (Goodman 1989)
  - PRAM +
  - Coherency on the same data item - all processes agree on the order of write operations to the same data item
  - Weak consistency (Dubois 1988)
    - Consistency need only apply to a group of memory accesses rather than individual memory accesses
    - Use synchronization variables to make all memory changes visible to all other processes (e.g., exiting critical section)
    - All access to synchronization variables must be sequentially consistent
    - Write operations are completed before access to synchronvar
    - Access to non-synchronvar is allowed only after synchronvar access is completed
- **Release consistency** (Gharachorloo 1990)
  - Two synchronization vars
    - Acquire - all changes to synchronized vars are propagated to the process
    - Release - all changes to synchronized vars are propagated to other processes
    - Programmer has to write accesses to these variables

Consistency models (cont.)

- **Causal consistency** (Hutto and Almasi 1990)
  - The processes are seen in the same order if they are potentially causally related
  - Read/write (two write) operations on the same item are causally related
  - Causality is transitive - if a process carries out an operation A that causally depends on the preceding op B then causally related to A even if they are on different items
- **PRAM consistency** (Lipton & Sandberg 1988)
  - All processes see only memory writes done by a single process in the same (correct) order
  - PRAM = pipelined RAM
    - Writes done by a single process can be pipelined; it doesn’t have to wait for one to finish before starting another
    - Writes by different processes may be seen in different order on a third process
  - Easy to implement — order writes on each processor independent of all others

Comparison of consistency models (cont.)

- Models differ in how restrictive they are, difficulty to implement, ease of implementation, and performance
- **Strict consistency** — most restrictive, but impossible to implement
- **Sequential consistency** — widely used, intuitive semantics, not much extra burden on programmer
- But does not allow much concurrency
- **Causal & PRAM consistency** — allow more concurrency, but have non-intuitive semantics, and put more of a burden on the programmer to avoid doing things that require more consistency
- **Weak and Release consistency** — intuitive semantics, but put extra burden on the programmer
Implementation issues

- how to keep track of the location of remote data
- how to overcome the communication delays and high overhead associated with execution of communication protocols
- how to make shared data concurrently accessible at several nodes to improve system performance

Granularity

- Granularity - size of shared memory unit
  - Page-based DSM
    - Single page — simple to implement
    - Multiple pages — take advantage of locality of reference, amortize network overhead over multiple pages
      - Disadvantage — false sharing
    - Shared-variable DSM
      - Share only those variables that are need by multiple processes
      - Updating is easier, can avoid false sharing, but puts more burden on the programmer
  - Object-based DSM
    - Retrieve not only data, but entire object — data, methods, etc.
    - Have to heavily modify old programs

Implementing sequential consistency on page-based DSM

- Can a page move? … be replicated?
- Nonreplicated, nonmigrating pages
  - All requests for the page have to be sent to the owner of the page
  - Easy to enforce sequential consistency — owner orders all access request
  - No concurrency
- Nonreplicated, migrating pages
  - All requests for the page have to be sent to the owner of the page
  - Each time a remote page is accessed, it migrates to the processor that accessed it
  - Easy to enforce sequential consistency — only processes on that processor can access the page
  - No concurrency

Implementing sequential consistency on page-based DSM (cont.)

- Replicated, migrating pages
  - All requests for the page have to be sent to the owner of the page
  - Each time a remote page is accessed, it’s copied to the processor that accessed it
  - Multiple read operations can be done concurrently
  - Hard to enforce sequential consistency — must invalidate (most common approach) or update other copies of the page during a write operation
- Replicated, nonmigrating pages
  - Replicated at fixed locations
  - All requests to the page have to be sent to one of the owners of the page
  - Hard to enforce sequential consistency — must update other copies of the page during a write operation

Thrashing

- Occurs when system spends a large amount of time transferring shared data blocks from one node to another (compared to time spent on useful computation)
  - interleaved data access by two nodes causes data block to move back and forth
  - read-only blocks invalidated as soon as they are replicated
- handling thrashing
  - application specifies when to prevent other nodes from moving block - has to modify application
  - “nailing” block after transfer for a minimum amount of time t - hard to select t, wrong selection makes inefficient use of DSM
    - adaptive nailing?
  - tailoring coherence semantics (Minin) - requires programmers