Some Recent Medium-scale NUMA Multiprocessors (research machines)

- DASH (Stanford) multiprocessor.
  - “Cluster” = 4 processors on a shared-bus with a shared L2
  - Directory cache coherence on a cluster basis
  - Clusters (up to 16) connected through 2 2D-meshes (one for sending messages, one for acks)
- Alewife (MIT)
  - Dynamic pointer allocation directory (5 pointers)
  - On “overflow” of a directory entry, software takes over
  - Multithreaded. (Fast) Context-switch on a cache miss to a remote node
- FLASH (Stanford)
  - Use of a programmable protocol processor. Can implement different protocols (including message passing) depending on the application

Some Recent Medium-scale NUMA Multiprocessors (commercial machines)

- SGI Origin (follow-up on DASH)
  - 2 processors/cluster
  - Full directory
  - Hypercube topology up to 32 processors (16 nodes)
  - Then “fat hypercube” with a metarouter (up to 256 processors)
    - vertices of hypercubes connected to switches in metarouter
- Sequent NUMA-Q
  - SPM clusters of 4 processors + shared “remote” cache (caches only data not homed in cluster)
  - Clusters connected in a ring
  - SCI cache coherence via remote caches

Extending the range of SMP’s – Sun’s Starfire

- Use snooping buses (4 of them) for transmitting requests and addresses
  - One bus per each quarter of the physical address bus
- Up to 16 clusters of 4 processor/memory modules each
- Data is transmitted via a 16 x 16 cross-bar between clusters
- “Analysis” shows that up to 12 clusters, performance is limited by the data network; after that it’s by the snooping buses

Multiprogramming and Multiprocessing Imply Synchronization

- Locking
  - Critical sections
  - Mutual exclusion
  - Used for exclusive access to shared resource or shared data for some period of time
  - Efficient update of a shared (work) queue
- Barriers
  - Process synchronization -- All processes must reach the barrier before any one can proceed (e.g., end of a parallel loop).

Locking

- Typical use of a lock:

```
  while (!acquire (lock))    /*spin*/
     ;
/* some computation on shared data*/
release (lock)
```

- Acquire based on primitive: Read-Modify-Write
  - Basic principle: “Atomic exchange”
  - Test-and-set
  - Fetch-and-add

Test-and-set

- Lock is stored in a memory location that contains 0 or 1
- Test-and-set (attempt to *acquire*) writes a 1 and returns the value in memory
- If the value is 0, the process gets the lock; if the value is 1 another process has the lock.
- To *release*, just clear (set to 0) the memory location.

Atomic Exchanges

- Test-and-set is one form of atomic exchange
- Atomic-swap is a generalization of Test-and-set that allows values besides 0 and 1
- Compare-and-swap is a further generalization: the value in memory is not changed unless it is equal to the test value supplied

Fetch-and-Θ

- Generic name for fetch-and-add, fetch-and-store etc.
- Can be used as test-and-set (since atomic exchange) but more general. Will be used for barriers (see later)
- Introduced by the designers of the NYU Ultra where the interconnection network allowed combining.
  - If two fetch-and-add have the same destination, they can be combined. However, they have to be forked on the return path

Full/Empty Bits

- Based on producer-consumer paradigm
- Each memory location has a synchronization bit associated with it
  - Bit = 0 indicates the value has not been produced (empty)
  - Bit = 1 indicates the value has been produced (full)
- A write stalls until the bit is empty (0). After the write the bit is set to full (1).
- A read stalls until the bit is full and then empty it.
- Not all load/store instructions need to test the bit. Only those needed for synchronization (special opcode)
- First implemented in HEP and now in Tera.

Faking Atomicity

- Instead of atomic exchange, have an instruction pair that can be deduced to have operated in an atomic fashion
- Load locked (ll) + Store conditional (sc) (Alpha)
  - sc detects if the value of the memory location loaded by ll has been modified. If so returns 0 (locking fails) otherwise 1 (locking succeeds)
  - Similar to atomic exchange but does nor require read-modify-write
- Implementation
  - Use a special register (link register) to store the address of the memory location addressed by ll . On context-switch, interrupt or invalidation of block corresponding to that address (by another sc), the register is cleared. If on sc, the addresses match, the sc succeeds

Spin Locks

- Repeatedly: try to acquire the lock
- Test-and-Set in a cache coherent environment (invalidation-based):
  - Bus utilized during the whole read-modify-write cycle
  - Since test-and-set writes a location in memory, need to send an invalidate (even if the lock is not acquired)
  - In general loop to test the lock is short, so lots of bus contention
  - Possibility of "exponential back-off" (like in Ethernet protocol to avoid too many collisions)

Test and Test-and-Set

- Replace "test-and-set" with "test and test-and-set".
  - Keep the test (read) local to the cache.
  - First test in the cache (non atomic). If lock cannot be acquired, repeatedly test in the cache (no bus transaction)
  - On lock release (write 0 in memory location) all other cached copies of the lock are invalidated.
  - Still racing condition for acquiring a lock that has just been released. (O(n^2) bus transactions for n contending processes).
- Can use ll+sc but still racing condition when the lock is released

Queuing Locks

- Basic idea: a queue of waiting processors is maintained in shared-memory for each lock (best for bus-based machines)
  – Each processor performs an atomic operation to obtain a memory location (element of an array) on which to spin
  – Upon a release, the lock can be directly handed off to the next waiting processor

Software Implementation

```
lock struct {int Queue[P]; int Queuelast;} /*for P processors*/

ACQUIRE    myplace := fetch-and-add (lock->Queuelast);
           while (lock->Queue[myplace modP] = = 1; /* spin*/
           lock->Queue[myplace modP] :=  1;

RELEASE lock->Queue[myplace + 1 modP] :=  0;
```

– The Release should invalidate the cached value in the next processor that can then fetch the new value stored in the array.

Queuing Locks (hardware implementation)

- Can be done several ways via directory controllers
- Associate a syncbit (aka, full/empty bit) with each block in memory (a single lock will be in that block)
  – Test-and-set the syncbit for acquiring the lock
  – Unset to release
  – Special operation (QOLB) non-blocking operation that enqueues the processor for that lock if not already in the queue. Can be done in advance, like a prefetch operation.
- Have to be careful if process is context-switched (possibility of deadlocks)

Barriers

- All processes have to wait at a synchronization point
  – End of parallel do loops
- Processes don’t progress until they all reach the barrier
- Low-performance implementation: use a counter initialized with the number of processes
  – When a process reaches the barrier, it decrements the counter (atomically -- fetch-and-add (-1)) and busy waits
  – When the counter is zero, all processes are allowed to progress (broadcast)
- Lots of possible optimizations (tree, butterfly etc.)
  – Is it important? Barriers do not occur that often (Amdahl’s law….)