Outline

- Last lecture: Small-scale Shared Memory Machines
  - communication: coherence and consistency
    - snooping-based coherence protocols
  - synchronization
- This lecture: Scalable Distributed Memory Machines
  - (continuation of Lecture 4)
    - synchronization (contd.)
    - case studies: SGI Challenge, Sun Enterprise
  - introduction
    - need for scalability
    - main design distinction: level of integration
  - realizing programming models
  - implications for software
  - case study: Cray T3E

Announcements

- Assignments
  - HW 2 due today
  - HW 3 assigned today, due 10/22
- Project
  - 2-page proposal due 10/15
  - if you want to meet, call me to set up an appointment

Outline

- Last lecture: Small-scale Shared Memory Machines
  - communication: coherence and consistency
    - snooping-based coherence protocols
  - synchronization
- This lecture: Scalable Distributed Memory Machines
  - (continuation of Lecture 4)
    - synchronization (contd.)
    - case studies: SGI Challenge, Sun Enterprise
  - introduction
    - need for scalability
    - main design distinction: level of integration
  - realizing programming models
  - implications for software
  - case study: Cray T3E

Point-to-Point Event Synchronization

- Software methods
  - interrupts
  - busy-waiting: use ordinary variables as flags
  - blocking: use semaphores
- Full hardware support:
  - full-empty bit with each word in memory
    - set when word is “full” with newly produced data (i.e. when written)
    - unset when word is “empty” due to being consumed (i.e. when read)
    - natural for word-level producer-consumer synchronization
      - producer: write if empty, set to full, consumer: read if full, set to empty
  - hardware preserves atomicity of bit manipulation with read or write
  - problem: flexibility
    - multiple consumers, or multiple writes before consumer reads?
    - needs language support to specify when to use
**Barriers**

- **Hardware barriers**
  - wired-AND line separate from address/data bus
  - set input high when arrive, wait for output to be high to leave
  - in practice, multiple wires to allow reuse
  - useful when barriers are global and very frequent
  - difficult to support arbitrary subset of processors
    - even harder with multiple processes per processor
    - difficult to dynamically change number and identity of participants
      - e.g. latter due to process migration
  - not common today on bus-based machines

- **Software algorithms**
  - implemented using locks, flags, counters

---

**A Simple Centralized Barrier**

- A shared counter maintains number of processes that have arrived
  - increment when arrive (lock), check until it reaches numprocs

```c
struct bar_type { int counter; struct lock_type lock; int flag = 0; } bar_name;
BARRIER (bar_name, p) {
  LOCK(bar_name.lock);
  if (bar_name.counter == 0)
    bar_name.flag = 0; /* reset flag if first to reach*/
    mycount = bar_name.counter++; /* mycount is private */
    UNLOCK(bar_name.lock);
    if (mycount == p) { /* last to arrive */
      bar_name.flag = 1; /* release waiters */
    }
    else while (bar_name.flag == 0) {}; /* busy wait for release */
}
```

**Problem?**

---

**A Working Centralized Barrier**

- Problem with the simple centralized barrier
  - consecutively entering the same barrier doesn’t work
  - must prevent a process from entering until all have left previous instance

- Solution: Sense reversal
  - wait for flag to take different value consecutive times
  - toggle this value only when all processes reach

```c
BARRIER (bar_name, p) {
  local_sense = !(local_sense); /* toggle private sense variable */
  LOCK(bar_name.lock);
  mycount = bar_name.counter++; /* mycount is private */
  if (bar_name.counter == p)
    UNLOCK(bar_name.lock);
    bar_name.flag = local_sense; /* release waiters */
  else {
    UNLOCK(bar_name.lock);
    while (bar_name.flag != local_sense) {};
  }
}
```

**Implications for Parallel Software**

- Load balance, inherent communication and extra work
  - issues same as before (unaffected by architecture)
  - assign so that only one processor writes a set of data
    - avoid write sharing
    - e.g. in RayTrace (read scene, write image): partition the image
  - Communication structure and mapping are not major issues
    - only require that processes do not migrate often (left to the OS)
  - Orchestration is the major issue
    - reduce cache misses and hence both latency and traffic
    - **temporal locality**: keep working sets tight enough to fit in cache
    - **spatial locality**: reduce fragmentation and false sharing

---

**Implications for Parallel Software**

- Load balance, inherent communication and extra work
  - issues same as before (unaffected by architecture)
  - assign so that only one processor writes a set of data
    - avoid write sharing
    - e.g. in RayTrace (read scene, write image): partition the image
  - Communication structure and mapping are not major issues
    - only require that processes do not migrate often (left to the OS)
  - Orchestration is the major issue
    - reduce cache misses and hence both latency and traffic
    - **temporal locality**: keep working sets tight enough to fit in cache
    - **spatial locality**: reduce fragmentation and false sharing
Bag of Tricks for Spatial Locality

- Reduce spatial interleaving of accesses
  - (task assignment) contiguous assignment of array elements
  - (data structuring) higher-dimensional arrays to keep partitions contiguous

- Contiguity in memory layout
  - Cache block straddles partition boundary
  - Cache block is within a partition boundary

(a) Two-dimensional array
(b) Four-dimensional array

Bag of Tricks for Spatial Locality (contd.)

- Beware of conflict misses
  - Typically a problem with “power of 2” caches and arrays
  - Leads to under-utilization of cache

- Use per-processor heaps for dynamic memory allocation
  - Ensures data structures use different cache blocks

- Copy data to increase locality
  - E.g., reuse of noncontiguous data
  - Must trade off against cost of copying

- Pad and align arrays
  - Can have false sharing versus fragmentation tradeoff

- Organize arrays of records for spatial locality
  - E.g., particles with fields: organize by particle or by field
    - In vector programs by field for unit-stride, in parallel often by particle
    - Phases of program may have different access patterns and needs

- These issues can have greater impact than inherent communication
  - Can cause us to revisit assignment decisions (e.g. row vs. block in grid)

Case Studies

- SGI Challenge with Powerpath bus

  36 R4400 (2.7 GFLOPS)
  1.2 GB/s bus (16 slots)
  128-byte cache line
  Split-transaction with
  8 outstanding requests
  All transactions take 5 cycles

- Uses Illinois MESI protocol
  - With cache-to-cache sharing

- 250ns access time from address on bus to data on bus
  - But overall latency seen by processor is 1000ns!
    - 300ns for request to get from processor to bus
    - 300ns more for data to get to processor chip
Case Studies (contd.)

- SUN Enterprise, with Gigaplane bus
  
  30 UltraSParc (9 GFLOPS)
  2.7 GB/s bus (16 slots)
  64-byte cache line
  split-transaction with
  112 outstanding transactions
  transactions take 11-18 cycles

- MOESI protocol
  - owned state for cache-to-cache sharing
- 300ns read miss latency
  - 11 cycle min bus protocol at 83.5 MHz is 130ns of this time
  - rest is path through caches and the DRAM access

Small-scale Shared Memory Machines: Summary

- Symmetric multiprocessors
  - category with the largest volume
  - bus-based architectures
  - coherence and consistency
    - simple write-through invalidation-based protocol
    - MSI write-back invalidation-based protocol
    - MESI write-back invalidation-based protocol
  - synchronization
    - hardware/software techniques for mutual exclusion, event synchronization
  - performance issues
    - bus bandwidth: exploit spatial and temporal locality

Why Scalable Parallel Machines?

Limitations of a shared bus architecture

- Hardware
  - length considerations
    - typically, less than 2 ft
    - determined by capacitive loading (signal quality) and power considerations
  - fixed number of slots
    - each device loads the bus
  - fixed maximum bandwidth
  - limited number of outstanding bus transactions
    - bus imposes global order

- Software
  - single operating system
    - if any processor fails, the system is rebooted

What Does Scalable Mean?

A scalable system attempts to avoid inherent design limits on the extent to which resources can be added to the system

- Four dimensions
  - bandwidth: concurrent transactions on independent wires
  - latency: $T(n) = \text{overhead} + \text{channel time} + \text{routing delay}$
  - cost of the system: fixed cost + incremental cost ($p, m$)
  - packaging of the system: modules, clock distribution, wires

- In an ideal scalable system
  - latency stays constant
  - bandwidth and cost of the system grow linearly
  - for system packaging, balance between
    - dense packing: to reduce fixed cost of wires, connectors, etc.
    - loose coupling: to reduce engineering effort and permit technology scaling
Generic Distributed-Memory Multiprocessor

2 levels of switches
- intranode switch is typically a bus
- internode switches provide independent communication paths between nodes so that bandwidth increases as nodes are added

Key Distinction: Level of Integration

- chip-level (processor bus): J-machine, Alewife, nCUBE/2
- board-level (memory bus): CM-5, Paragon, T3E, Origin
- system-level (I/O bus): SP-2, PCs/Myrinet

Realizing Programming Models

- A primitive network transaction
  - one way transfer + some action at the destination

- Differences from a bus transaction
  - source and destination of a transaction are uncoupled
    - no direct wires, no global arbitration of resources
  - no global information available to all modules at the same instant
  - huge number of concurrent transactions

Network Transactions: Design Issues

- **Protection**: who performs the check?
- **Format**: fixed vs. variable-sized packets, encapsulation
- **Output buffering**: staging area on source
- **Media arbitration**: global vs. per-link reservation
- **Destination name and routing**: logical to physical translation
- **Input buffering**: staging area on destination, N->1 merge
- **Action**: what happens as a result of the transaction?
- **Completion detection**: what is the source informed of?
- **Transaction ordering**: what ordering guarantees?
- **Deadlock avoidance**: affected by routing and end-point behavior
- **Delivery guarantees**: what happens when the destination buffer is full?
Supporting a Shared Address Space

- **Issues**
  - *input buffer overflow*: many nodes request from same address
  - *fetch deadlock*: a node must be able to sink requests
  - *delivery order*: important for implementing memory consistency models (more in Lectures 6 and 7)

Message Passing: Asynchronous Protocols

- **Problems with the optimistic protocol**
  - *store-and-forward delay* (if no posted receive buffer)
  - *input buffer overflow* (function of program behavior)

Supporting Message Passing: Synchronous Protocol

- Alternately, can be receiver-initiated (but, restricts tag matching)
  - tag-check on the sender: requires only two network transactions

Message Passing: Asynchronous Protocols (1+2-phase)

For small messages, a credit-based scheme can reduce handshake costs
Active Messages

- A message-passing abstraction closer to the level of the underlying network transaction
  
send( node, handler, buf, len)
  extract()

  - message send associated with a handler at the other end
    - handler integrates message into ongoing computation
    - allocates a buffer to store message if necessary
      - very often, this is not required (e.g., for looking up an address)
  - no explicit receive
    - reduces cost of tag matching and buffering
  - request-response transactions constitute a restricted RPC
    - response handlers cannot send messages to prevent deadlock
    - interface permits very efficient implementations
      - order of magnitude better than vendor MPI implementations

- More details in Lecture 11

HW Design Choices: Physical DMA

- HW does no interpretation on the information within a transaction
  - representative of early message-passing machines
    - nCUBE10, nCUBE/2, Intel iPSC, iPSC2, iPSC860, Delta, Ametek, SP-1
  - physical DMA to/from network
    - use of physical addresses
    - sending/receiving requires OS intervention

HW Design Choices: User-level Access

- HW distinguishes between system and user messages
  - user messages both injected and received without OS intervention
  - more recent machines: CM-5, Cray T3D/T3E, VIA

HW Design Choices: Dedicated Message Processing

- HW dedicates resources to interpret information in transactions
  - protocol processing can be off-loaded to communication processor
  - example machines: Meiko CS-2, Intel Paragon

  - differentiated by whether CP is a symmetric processor, or an embedded processor (with its own path to the network interface)
HW Design Choices: Shared Physical Address Space

- HW provides support for loads, stores, atomic operations
  - dancehall: NYU Ultracomputer, BBN Butterfly, IBM RP-3
  - distributed memory: Cray T3D, Cray T3E, Origin 2000

![Diagram of HW Design Choices]

Clusters and Networks of Workstations

- Building scalable machines using commodity components
  - systems: processors, memory, disk (e.g., a PC)
  - “killer” networks (e.g., Myrinet)
    - connects to the system on the peripheral bus
    - cost-effective, but needs to overcome disadvantages of loose coupling
    - very active area of research (mine!)

![Diagram of Clusters and Networks]

Implications for Parallel Software

- Network transaction performance: \( T(n) = T_5 + \frac{n}{B} \)

<table>
<thead>
<tr>
<th>Machine</th>
<th>Year</th>
<th>MFLOPs</th>
<th>( T_5 ) (µs)</th>
<th>( T_5 ) (cycles)</th>
<th>( T_5 ) (FLOPs)</th>
<th>B (MB/s)</th>
<th>( m_{ij} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>iPSC/2</td>
<td>1988</td>
<td>1</td>
<td>700</td>
<td>5600</td>
<td>700</td>
<td>2.7</td>
<td>1400</td>
</tr>
<tr>
<td>iPSC/660</td>
<td>1991</td>
<td>40</td>
<td>160</td>
<td>6400</td>
<td>2</td>
<td>320</td>
<td>8</td>
</tr>
<tr>
<td>SP-1</td>
<td>1993</td>
<td>100</td>
<td>50</td>
<td>2500</td>
<td>5000</td>
<td>25</td>
<td>1250</td>
</tr>
<tr>
<td>Meiko CS-2</td>
<td>1993</td>
<td>24</td>
<td>83</td>
<td>7470</td>
<td>1992</td>
<td>43</td>
<td>3560</td>
</tr>
<tr>
<td>Paragon</td>
<td>1994</td>
<td>50</td>
<td>30</td>
<td>1500</td>
<td>1500</td>
<td>175</td>
<td>7240</td>
</tr>
<tr>
<td>SP-2</td>
<td>1994</td>
<td>200</td>
<td>35</td>
<td>3960</td>
<td>7000</td>
<td>40</td>
<td>2400</td>
</tr>
<tr>
<td>T3D</td>
<td>1994</td>
<td>94</td>
<td>21</td>
<td>3150</td>
<td>1974</td>
<td>27</td>
<td>1502</td>
</tr>
<tr>
<td>NOW</td>
<td>1996</td>
<td>180</td>
<td>16</td>
<td>2672</td>
<td>2880</td>
<td>38</td>
<td>4200</td>
</tr>
</tbody>
</table>

- overlap possible between components of transaction
  - \( O_s \) and \( O_r \) (overhead at sender and receiver), and \( L \) (network latency)

Synchronization

- Message passing model
  - mutual exclusion comes for free
  - synchronization and data transfer are coupled
  - implementing group synchronization from point-to-point messages

- Shared address space model
  - need to extend lock implementations for scalable machines
  - details in Lecture 7

![Diagram of Synchronization]
Case Study: Cray T3E

- DEC Alpha 21164
- 300-450 MHz
- 8KB L1 instr and data
- 96 KB L2 unified
- 2 outstanding memory refs

Control Shell Logic
- Router, 3D interconnect
- 600 MB/s per link in each direction
- 128 bits wide
- 1.2 GB/s
- 300-450 MHz
- 4 x 4B x 75MHz = 1.2 GB/s

Local Memory
- 64MB to 2GB

Cray T3E: The “Glue” Logic

- 2048 processors, 3D Torus
- Stream Buffers
- Address Translation
- E-registers
- Messaging
- Atomic Ops

- Enhances local memory bandwidth
- 600 MB/s per link in each direction
- low-overhead synchronization
- 6 streams: 64-byte buffers monitor last 8 requests on repeat: allocate a stream

Cray T3E: Global Addressing

- 21164 Microprocessor
- Data Bus
- Address Bus
- E-Registers
- Centrifuge
- Virtual Address
- Virtual PE
- Global Xlation Buffer
- Segment Xlation Table
- Base PE
- GSEG
- Offset
- Physical Addr

- Network

Cray T3E: E-Registers

- Central mechanism of the “glue” logic
  - 512 user + 128 system registers (64-bit)
  - memory-mapped into non-cacheable I/O space
    - bit 39 of physical address distinguishes cacheable/noncacheable space
  - 2 types of operations
    - direct loads and stores
      - implicitly synchronized using full/empty flags
    - global E-register operations (gets and puts) use I/O space stores
      - address encodes command (put/get), and src/dest E-register
      - data encodes pointer to block of E-registers and address index
        - flexible data distribution
        - split-phase operations: 4 put/get operations every 2 (75 MHz) cycles
      - strided puts/gets

- Benefits
  - pipelined remote memory operations
  - high memory-to-memory bandwidth
    - 60 MB/s versus 340 MB/s for random gather
Cray T3E: Messaging

- Arbitrary number of user-level message queues
  - 64-byte messages
  - queues can be any size up to 128 MB
- Message queue control word (MQCW)
  - store MQCW at desired location of the queue
  - user responsible for buffer management
  - Threshold determines when processor is interrupted ($S$ is set)
- Sending a message
  - assemble message in block of 8 E-registers
  - issue a SEND command (address must point to valid MQCW)
  - flags on source E-registers indicate success/failure
- Performance
  - 5μs latency, ~350MB/s bandwidth

Cray T3E: Atomic Operations

- Atomic operations against arbitrary memory locations
- Build upon E-register support
  - specify AMO command on address bus
  - operands taken from aligned block of base/index E-registers (data bus)
  - result returned to E-register specified on address bus
- Operations
  - Fetch&Inc (hardware merge support)
  - Fetch&Add
  - Compare&Swap
  - Masked_Swap
- Performance
  - Sustained rate: 40ns per F&I, 222ns per op for the others
  - Latency: ~1.5μs

Cray T3E: Parallel Performance

[ from “Performance of the CRAY T3E Multiprocessor”, Supercomputing’97 ]

Performance per PE (MFLOP/s), NAS Parallel Benchmarks, Class C

<table>
<thead>
<tr>
<th>PEs</th>
<th>EP</th>
<th>MG</th>
<th>CG</th>
<th>FT</th>
<th>IS</th>
<th>LU</th>
<th>SP</th>
<th>BT</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>340</td>
<td>110</td>
<td>30</td>
<td>130</td>
<td>20</td>
<td>250</td>
<td>100</td>
<td>140</td>
</tr>
<tr>
<td>128</td>
<td>340</td>
<td>110</td>
<td>25</td>
<td>125</td>
<td>15</td>
<td>250</td>
<td>90</td>
<td>130</td>
</tr>
<tr>
<td>256</td>
<td>340</td>
<td>100</td>
<td>15</td>
<td>130</td>
<td>15</td>
<td>220</td>
<td>80</td>
<td>120</td>
</tr>
<tr>
<td>512</td>
<td>340</td>
<td>90</td>
<td>15</td>
<td>110</td>
<td>15</td>
<td>200</td>
<td>70</td>
<td>110</td>
</tr>
<tr>
<td>1024</td>
<td>340</td>
<td>85</td>
<td>15</td>
<td>70</td>
<td>10</td>
<td>160</td>
<td>60</td>
<td>100</td>
</tr>
</tbody>
</table>

Lecture Summary

- Scalable parallel machines
  - motivation
    - need for scalability
    - main design distinction: level of integration
  - realizing programming models
    - primitive network transactions
    - shared address space and message passing
  - hardware design choices
    - physical DMA, user-level access, dedicated message processing, shared physical address space
  - implications for software
  - case study: Cray T3E
Next Lecture

- Scalable Shared Memory Machines
  - directory-based coherence
  - relaxed memory consistency models
  - case study: SGI Origin 2000

- Tutorial: Programming with threads

Readings
  - Culler/Singh/Gupta: Chapter 8

- Reminder: 2-page project proposal due October 15!