ADMINISTRIVIA

Problem Set 1, due *now*

Reading for Monday:

- Walker94: The MPI Standard (reader)

OUTLINE

User-level protocols

Shared memory vs. message passing

Polling vs. interrupts

Coprocessor summary

SHARED-MEMORY VS. MESSAGE PASSING

General claims for shared memory:

- Simpler than message passing
- Avoids complicated planning and data layout
- Faster for fine-grain access (one instruction)

When is shared-memory simpler?

When is it faster?

PRODUCER-CONSUMER SHARING

A produces a value, v, used by B

Message passing: send one active message to B

Shared memory: (common case)

- B reads v before it is ready
- A writes $v \Rightarrow$ invalidate B, wait for ack
- B reads v again $\Rightarrow$ second round trip

Prefetching does not help

One solution: *update* protocols

Update Protocols

Idea: A writes v in place at B

B always has the local copy

Problems:

- ◆ Still can't tell when data reaches B
  Requires separate synchronization (e.g. barrier)
- ◆ Update protocols can lead to weaker consistency
  (but not for single consumer)
- ◆ Want most writes to invalidated remote copies
  ⇒ two kinds of write instructions

Typhoon's EM3D Protocol

Idea: new user-level protocol for EM3D

Advantages:

- ◆ Uses explicit update commands
  ⇒ Exactly 1 write from A to B
- ◆ Uses presence counters at receiver for synchronization

Nearly identical to message passing code, except:

- ◆ Message passing code is simpler
- ◆ SM requires 10x more memory for caching
- ◆ MP is slightly faster (with better cache behavior!)

When are user-level protocols helpful?
(vs. System protocols + message passing)

User-Level Protocols

User-Level protocols are helpful when:

- ◆ You know the *nature* of the sharing...
  - Temporal locality
  - Producer/Consumer
- ◆ but not the exact layout or dependencies

Message passing requires precise layout...

Regular protocols handle a specific kind of sharing

Application Classes

Static layout:

- ◆ Iterative PDE, FFTs, dense matrices, APPBT
- ◆ Message passing is works great

Unknown but stable layout:

- ◆ Iterative sparse matrix solvers, EM3D
- ◆ Tempest: first iteration identifies the consumers
- ◆ MP: preprocessing step identifies the layout
- ◆ Rest of the iterations are done with MP

Dynamic layout:

- ◆ Barnes-Hut: layout changes slowly over time
- ◆ Tempest: level of indirection to track bodies
- ◆ MP: software caching (essentially the same)

Do you need User-Level Protocols?

Claim: a few good protocols (+ MP) will get:

- All of the benefit on most applications
- Most of the benefit on some applications
- Perform noticeably worse on very few applications

The Tempest argument:

- User-level protocols are required for some applications
- ... and are worth the complexity
- There will always be applications that *require* a new protocol for good performance

Unknowns:

- How good is software shared memory?
- How much harder is writing protocols than MP programs?

Synchronization

Fundamental problem with shared memory:
How do you know when data arrives?

MP: handler tells you

- Receiver looks in one “place” for *all* incoming data

SM: data just arrives at its final destination

Approaches:

- Protocol counts arriving data: Tempest EM3D, split-C
- Barriers ⇒ all data has arrived (tricky)
- Mark each incoming data structure *full* or *empty*
  - Receiver either: polls all incoming data
  - or accesses data in a fixed order and spins

Shared Memory and NOW

Should NOWs support shared memory?

Challenges:

- How to define the address space (variable machine size)
- Fault tolerance
- Multiple users
- Deadlock
- Not much to leverage...

What about indirect support?

- Global address space (without caching)
- Page-level shared memory
- Fine-grain access control (as in Tempest)
  ⇒ can build fine-grain software shared memory

Active Messages Revisited

Tempest supports user-level AM, FLASH does not

We can build user-level AM given only enqueue

- Send AM = enqueue packet
- Receive AM = dequeue packet and execute
  polling = check the queue

Trade-offs:

+ Arbitrary handlers, *NO restrictions*
+ No user-level code on coprocessor
– No user-level code on coprocessor
+ True multiprogramming
+ Checking queue is faster than polling
+ Optional interrupts

Polling vs. Interrupts

When is polling faster?

- Very frequent arrivals ⇒ no difference
  (Interrupt handler polls after the first arrival)
- Somewhat frequent or small bursts ⇒
  polling is faster; hard to amortize interrupt overhead
- Infrequent ⇒ interrupts is faster because of wasted polling
- Phases (Bimodal) ⇒ only poll during communication phases
  ⇒ polling is faster

Polling allows bandwidth matching (limited receiving)

Fault model:

- Polling error ⇒ deadlock ⇒ can figure it out with debugger
- Interrupt error ⇒ race condition ⇒ nondeterministic bug!

Atomicity

Interrupts introduce race conditions

- ⇒ handlers need to access locks
- ⇒ additional overhead for lock/unlock

What happens when a handler can't get a lock?

- Blocking would cause deadlock
- Tempest solution:
  - Put handler aside and set flag
  - Lock holder checks flag on release,
    then executes the handler
  - What if lock acquisition is in the middle of a handler?

Polling automatically excludes handlers during critical sections (not quite true!?)

What should a coprocessor provide?

One opinion!

Shared memory support

Block transfers & strided block transfers

- Interleaving of transfers
- Exploit DMA

Optional interrupts

Enqueue + specialized handlers:

- Read/Write
- Write plus increment counter
- Atomic operations: compare and swap
- Barriers, scan operations