ADMINISTRIVIA

Project checks over the next two weeks

Reading for Wednesday (from the reader):

- Kay & Pasquale: TCP Control Overhead
- Clark & Tennenhouse: New Generation of Protocols

Optional reading: Split-C paper (online)

OUTLINE: LOW-LEVEL SOFTWARE

Review

Communication primitives

Split-C

Comparison of primitives

What are the right primitives?

FLOW CONTROL REVIEW

LB offers:

- No dropped packets
- Robust behavior if we get the rate wrong

RB offers:

- True congestion and flow control
- Reduces need for virtual channels/virtual circuits

Best performance requires both!

- Optimal overhead (just block-transfer overhead)
- Minimal congestion for given end-to-end rate

(only) example: *bandwidth matching*

BANDWIDTH MATCHING

Idea: statically limit sender to bottleneck rate:

- Eliminates congestion
- Avoids connections
- Zero overhead in the normal case
- Works well with adaptive routing
- Bubbles improve fairness

Cons:

- Requires one-on-one transfers (not really new)
- Requires polling not interrupts
- Requires knowledge of the bottleneck rate
  - Usually limited by the receiver
  - May also be the bisection bandwidth

Active Messages

Basic idea: building block for all protocols

The remote procedure is the *handler:*

- Handlers typically named by absolute address
- Handlers run atomically (and must be short)
- Handlers may send other active messages (with some restrictions)

Very simple:

- No buffering or ordering required
- Easy dispatch

Control vs. Data Transfer

Active messages execute at the receiver ⇒

- Correct process must be running
- Interacts with normal behavior of receiver
- Often involves some form of context switch

Most transfers are data only ⇒
avoid overhead of handler

Everything *can* use active messages,
but not everything should

Alternative: explicit access to remote memory

- Portals in PUMA
- Segments in TLL94

Remote Segments

Segment (portal) consists of:

- Global name
- Contiguous range of application address space

Locally segment is just a block of memory

Remote nodes access segment via *(name, offset):*

- ReadBlock(name, start, length, local incoming buffer)
- WriteBlock(name, start, length, local outgoing buffer)

Requires no cycles from remote process:

- Can be performed by kernel or by coprocessor

Managed by kernel, but read/write at user level

Segments vs. Active Messages

Segments are strictly less powerful:

- Can only move data
- Cannot even signal arrival or completion (directly)

Segments are faster for data transfer:

- AM with Polling ⇒ must wait for remote process
- AM with Interrupts ⇒ incur interrupt overhead
- Exploits DMA at both ends (to free up processors)
- Typically allows larger messages (lower overhead)

Send-Receive with segments (sender's view):

1. RPC for segment name, size
2. Write block transfer

Split-C

C extended for parallel programs

Global address space:

- Single coherent address space
- No cache-coherence hardware
- Each node owns part of the space (the *local* part)

Strong distinction between local and remote memory:

- Normal pointers can only access local memory
- Special *global pointers* for full access = *(node, address)*

Split-phase operations hide latency:

- First part: start remote operation, *get* or *put*
- Second part: wait for operations to complete, *sync*

Basic Split-C with Segments

Local memory is mapped as one giant segment:

- Segment name = node number
- Global pointers = *(node, address)* = segment's *(name, offset)*

Each *get*:

- Initiates read from remote segment
- Adds itself to list of pending gets

When data comes in:

- TLL94: optional handler
- Portals: mark bit in portal descriptor

Sync:

- Scan list of pending gets to make sure they're complete
- AM and TLL94 with handlers: can just use a counter

Split-Phase Put

How do we know when all *puts* are done?

AM solution:

- Count outstanding put bytes
  - Add to local count for each put
  - Reduce *local* count as each store arrives
- Sync for stores: global sum of counts must be zero

Problem for segments:

- Can't change count when data arrives (need a handler!)

Bottom line: segments are NOT sufficient for Split-C

How To Use Segments

Segments are only for data transfer

Poll for arrivals:

1. Add marker to each incoming buffer (at end!)
2. Initialize marker to *false*
3. Include marker=*true* with transmitted data
4. Poll until marker changes to *true*

Advantages over active messages:

- Can be done entirely in kernel and/or coprocessor
- Target process can even be asleep
- No handler overhead (especially bad for CM-5)

Problems...

SEGMENTS: PROBLEM 1

Can't act when a transfer is complete

Example: Split-C's use of counters

Worse example: which is the last incoming byte?
 I.e., where do we put the marker?

Radix sort: many nodes send to the same segment
 using different offsets

Can't tell when all of the data has arrived!
 Must scan entire buffer

AM solution: count the arriving bytes

SEGMENTS: PROBLEM 2

Expecting data from n nodes, first come first served

Problem: don't know which message will arrive next
 Must poll *all* buffers simultaneously (slow, complex)

Example: data-driven RPC

AM solution: messages are queued up

Segments work when data is *placed* somewhere

AM preferred when data is for the node in general

ALEWIFE HANDLERS

Combines handler with DMA

- Removes length restriction on AM data transfer
- Avoids copies

Sender:

*send_DMA(node, handler, arg1, ... argN, dest_base,
 local_base, length)*

Receiver:

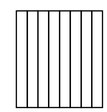
1. Executes *handler(arg1, ... argN)*
2. Simultaneously performs DMA into *dest_base*
3. Handler uses *wait_for_storeback()* to wait for DMA

Example: Split-C transfer with counter
 ⇒ DMA does transfer, handler updates counter

STRIDED ACCESS

Everything so far deals with contiguous blocks...

What about 2D arrays (or higher):



- One dimension is not contiguous
- Column major ⇒ strided rows

Current solution: separate DMA or AM for each block

- 100x100 array ⇒ 100 transfers per row, 1 per column

Gather/Scatter:

- *Gather* groups 100 blocks into one message
- *Scatter* breaks message back into strided blocks
- Lots of copying! Can we move this to hardware?

Strided Access in Hardware

Alewife does gather in hardware, scatter in software

Problems:

- Sender and receiver may use different strides
- Complicated interface:
  local base, block count, block length, local stride, remote base, remote stride
- This only handles 2D arrays, what about 3D, 4D, ...

Need one stride per *written* dimension:

- Any one dimension in 3D requires only 1 stride
- Writing a plane in 3D requires 2 strides

Settle for one dimension per message

Desired Properties

Remote access is common ⇒ specialize

- Kernel managed segments, or
- Specialized kernel/coprocessor handlers
- Exploit (existing?) protection hardware

Allow read/write even when target process unavailable

Split-phase access to hide latency

Strided block transfers with optional handlers

Provide handlers for generality,
but avoid them whenever possible

An Interface

Regular active messages for RPC & other protocols

Kernel-managed segments for user-level access

DMA-based block transfers with optional handlers

- Avoid copying and processor interaction
- Handlers support notification (such as counting)

Simple strided DMA (coprocessor?)

- Reduces overhead for multidimensional arrays
- Optional handler after all blocks (vs. once per block)