Announcements

- Problems with shared-memory emulation layer on Beowulf cluster have been resolved (for now)
  - Send me e-mail if things break
  - For assignment 4, consider this emulation as implementing shared memory with page-sized (4096 bytes) coherence blocks
    - Additional details about underlying protocols in Lectures 9 and 10.
- Please start working on your project
  - Follow up with me (e-mail or office hours)
- Questions?

Outline

- So far in the course, we have talked about
  - Dominant architectures
    - Symmetric multiprocessors (small-scale shared memory)
    - Tightly coupled scalable parallel machines
      - With and without support for coherent caches
  - Dominant programming models
    - Data parallel
    - Message passing
    - Shared memory (all memory is shared, careful locality management)
- Growing trend: Use “clusters” of commodity PCs as parallel machines
  - Advantage: Economy of scale
  - Disadvantage: Loosely coupled
    - Lower communication performance
    - Rare to find hardware support for cache coherence
  - Performance optimizations in message passing and shared memory programming models become essential

Next Four Lectures

- Lecture 8: (Today)
  - Cluster network interfaces
    - Fast Ethernet, Myrinet, VIA
  - Optimizing message passing programs
    - Portable optimizations
    - Alternative messaging layers
      - Active Messages, Put/Get
- Lectures 9 and 10: (November 14 and 21)
  - Software support for shared memory models
    - Page based and object-based shared memory
- Lecture 11: (November 28)
  - High-level Parallel/Distributed Programming Models
  - Future directions
Cluster Architectures

- Only degree of freedom: Cluster network interface
  - Still needs to be relatively decoupled (i.e., on the I/O bus)
  - However, can provide features to improve performance

Cluster Network Interfaces (1): Fast Ethernet

Implications for communication software
- OS involvement is required to move data to/from host
  - System call overhead
  - Host informed of messages via interrupts
- Application-to-application latency depends on driver and API
  - ~50 µs (U-Net, MVIA): software implementation of VIA-like interface
  - ~150 µs (TCP/IP)
    - Protocol overheads + extra data copy from kernel to user memory

Cluster Network Interfaces (2): Myrinet

Implications for communication software
- Division of responsibility between Host/LANai processors
  - LANai offloads interrupt processing responsibility from host
    - DMAs to/from FIFOs into NIC memory, can also DMA directly into host memory
    - NIC memory can be mapped into process virtual address space
    - Host processor can use programmable I/O to write messages
  - Application-to-application latency of 9 µs (GM over Myrinet: a socket-like API)
- Problem: Expensive! ($1200/node)

Cluster Network Interfaces (3): VIA

- Virtual Interface Architecture
  - Intel, Compaq, Microsoft
  - Borrowed ideas from several research projects
    - Active Messages (Berkeley), Fast Messages (Illinois), U-Net (Cornell), VMMC (Princeton)
Clusters Network Interfaces (3): VIA (cont’d)

- VI PL library provides interfaces to post/query descriptors
- Data transfer to/from memory regions responsibility of VIA implementation
  - Hardware implementations (e.g., Giganet) see app-to-app latency of $\approx 5 \mu s$
  - Software implementations (e.g., M-VIA) achieve app-to-app latency of $\approx 50 \mu s$
- Note that API is no longer sockets

Optimizing Message Passing Programs

Message-Passing Programming Models

- Components of communication cost
  - synchronization (matching), overhead, latency, BW, contention

Message Passing: Inherent versus Artifactual Costs

$$\text{communication cost} = f \left( \text{sync} - \text{overlap} + o + l + \frac{n/m}{B} + t_c \right)$$

- Some part of cost due to inherent reasons
  - $f$: frequency of communication
  - $n/m$: average message size
- Remainder due to artifactual costs
  - determined by how communication is structured and implemented
  - three key components that are under implementation/programmer control
    - synchronization/matching: (sync – overlap)
      - Solutions: Pre-posted receives, message pipelining, multithreading
    - overhead: $o$
      - Solutions: Message aggregation, low-overhead messaging layers
    - contention: $t_c$
      - Solutions: Scheduling communication
Reducing Communication Costs: Synchronization

- Four mechanisms for reducing (sync – overlap)
  - reducing synchronization/matching wait time
    - pre-posted receives
    - message pipelining
  - increasing overlap
    - overlap communication with local computation
    - multithreading

Reducing Synchronization Costs (1):
Early Posting of Receives

- **Idea**: Ensure that the receive buffer is available prior the send
- **Advantages**
  - reduces synchronization/matching wait times
  - allows use of more efficient send primitives (e.g., MPI_RSend)
- **Example**: Jacobi iterations, row distribution

```c
/* send up, receive down */
MPI_Send( row[1], North );
MPI_Recv( row[n+1], South );
/* send down, receive up */
MPI_Send( row[n], South );
MPI_Recv( row[0], North );
/* receive down, up */
MPI_Irecv( row[n+1], South );
MPI_Irecv( row[0], North );
/* send up, down */
MPI_Isend( row[1], North );
MPI_Isend( row[n], South );
MPI_Waitall(4, ...);
/* receive down, up */
MPI_Irecv( row[n+1], South );
MPI_Irecv( row[0], North );
MPI_Barrier( ...);
/* send up, down */
MPI_Rsend( row[1], North );
MPI_Rsend( row[n], South );
MPI_Waitall(2, ...);
```

Reducing Synchronization Costs (2):
Message Pipelining

- **Idea**: Send message piece as soon as it becomes available
- **Advantage**: enables overlap of processing and communication

```c
MPI_Recv( row[0], North );
for (i=1, i<N/P; i++)
  for (j=0, j<N; j++)
    row[i][j] = row[i-1][j];
MPI_Send( row[N/P], South );
for (jj=0; jj<N; jj+=B) {
  MPI_Recv( &row[0][jj], North );
  for (i=1; i<=N/P; i++)
    for (j=0; j<B; j++)
      row[i][j] = row[i-1][j];
  MPI_Send( &row[N/P][jj], South);
}
```

Reducing Synchronization Costs (3):
Overlapping Communication with Local Operations

- **Idea**: Separate out sends and receives as much as possible
- **Advantage**: Overlap local synchronization time with computation
- **Example**
  - SPMD code for array-based computation
    - Eg., \( A[i] = 0.5 \times (A[i] + A[i-1]) \)
    - Owner-computes rule:
      - SEND for non-local READ
      - RECV for non-local READ
    - in general: processors also execute iterations which write non-local data
      - LocalIterSet
      - NonLocalReadIterSet
      - NonLocalWriteIterSet
Reducing Synchronization Costs (4):
Multithreading

- **Idea:** Mask synchronization times with computation in another thread
  - works if overhead of thread switching comparable to communication costs
- **Advantage:** Can continue to use “natural” programming style

![Diagram showing synchronization/matching]

- Requires additional support from languages and implementation
  - support for multithreaded programs
  - thread-safe and thread-aware communication libraries
    - fortunately, most recent MPI implementations are in this category

Reducing Communication Costs: Overhead

- Sources
  - handshaking: buffer management, tag matching
  - data transfer
- Two mechanisms for reducing overhead
  - message aggregation
    - amortize overhead over transmission of more data
  - low-overhead messaging layers
    - design communication abstractions which allow cost to be commensurate with required complexity

Messaging Layers: Sources of Overhead

**Messaging layer** = Software (either kernel-level or user-level) that implements the communication operations

- Consider actions required to implement non-blocking communication
  - **On the send side:**
    - allocate a structure to store message
    - send a SendRdy message to destination
    - upon receiving a RecvRdy message, find structure and initiate transfer
      - fragmentation into network packets
      - flow control and retransmission (if necessary)
  - **On the receive side:**
    - store receive buffer in a structure and perform a tag match
    - upon receiving a SendRdy message, perform a tag match
    - on a match, send RecvRdy message and wait for transfer
      - reassemble message from network packets
      - handle out-of-order packets -> buffering
Low-Overhead Messaging Layers (1):
Active Messages

- **Rationale:** Provide a communication abstraction much closer to level of a network transaction which eliminates/reduces overheads of
  - buffer management
  - tag matching
  - multi-phase transactions (implied by the semantics)

<table>
<thead>
<tr>
<th>Machine</th>
<th>Year</th>
<th>MFLOPs</th>
<th>T0 (µs)</th>
<th>T0 (cycles)</th>
<th>T0 (FLOPs)</th>
<th>B (MB/s)</th>
<th>nG</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>nCUBE/2</td>
<td>1990</td>
<td>2</td>
<td>160</td>
<td>3000</td>
<td>300</td>
<td>2</td>
<td>300</td>
</tr>
<tr>
<td>iPSC/860</td>
<td>1991</td>
<td>40</td>
<td>160</td>
<td>6400</td>
<td>6400</td>
<td>2</td>
<td>320</td>
</tr>
<tr>
<td>CM-5</td>
<td>1992</td>
<td>20</td>
<td>95</td>
<td>3135</td>
<td>1900</td>
<td>8</td>
<td>760</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>1994</td>
<td>50</td>
<td>30</td>
<td>1500</td>
<td>1500</td>
<td>175</td>
<td>7240</td>
</tr>
<tr>
<td>Paragon</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>

Active Messages: An Example Program (Jacobi)

```c
void store_row_from_down( void *r, int size ) {
    memcpy( row[n+1][i], r[i], size );
    num_received++ ;
}

void store_row_from_up( void *r, int size ) {
    memcpy( row[0][i], r[i], size );
    num_received++ ;
}
```

Active Messages: Performance

- Communication costs dominated by hardware access costs
  - Asynchronous interface
  - minimal buffering costs
    - sender waits till network can accept message
  - minimal parsing costs (matching/selection)
    - handler executes immediately on arrival in order of reception
Low-Overhead Messaging Layers (2):
Fast Messages

- **Rationale:** Active Messages does not provide high-level guarantees which simplify user code
  - in-order delivery
    - no effort made to reorder messages in adaptive networks
  - reliable delivery
    - sender only waits for network to accept messages
    - higher-level protocols must ensure that message data reaches destination
  - sender-receiver decoupling
    - sender blocks if receiver is not responsive enough in clearing the network
- Building these guarantees in user code results in 2x cost increase
- **Fast Messages:** Active Messages + above guarantees
  - same interface as Active Messages in FM, version 1
    - careful implementation keeps costs close to raw hardware level
  - FM, version 2 provides stream-based interface
    - efficient substrate for implementing higher-level layers (e.g., MPI)

Low-Overhead Messaging Layers (3):
Remote Put/Get

- Data transfer using traditional messaging primitives
  - Does not require sender to know destination address of transfer
  - Implications
    - Messaging layer needs to demultiplex data into posted buffers
    - Buffer management, tag matching, flow control costs …
- **Observation:** Send-side often knows the destination of data transfer (e.g., Jacobi, radix, …)
  - More efficient primitives possible, which allow direct writing to (put) and reading from (get) remote memory
    - Benefit when underlying hardware supports these primitives (e.g., VIA)

Reducing Communication Costs: Contention

- Two sources of contention (important only for large messages)
  - **network:** competition for network resources
  - **end-point:** lots of messages incident on the same processor
    - queuing delays
    - problem even when inherent communication pattern is balanced
      - because processors/network get out of synchronization
- Solution: Schedule messages to reduce contention

Reducing Contention: Message Scheduling

- **Idea:** Explicitly limit the number of messages in flight
  - between pairs of nodes
  - in the entire system
- **Advantage:** Network and end-point resources are never overcommitted
- Example:
  - all-pairs communication: each processor sends to every other processor
    - required for permutation communication patterns (e.g., in sorting)
  - break up pattern into multiple communication rounds
Summary

- This lecture
  - Cluster network interfaces
    - Fast Ethernet, Myrinet, VIA
  - Optimizing message passing programs
    - Portable optimizations
    - Alternative messaging layers
      - Active Messages, Put/Get

- Next lecture
  - Software support for shared memory models
    - Page based and object-based shared memory