Announcements

- Course accounts
  - use “ssh” (secure shell) to connect to NCSA machines
    - “rlogin” on NYU CS machines is aliased to “ssh”
- Meeting to discuss project
  - please set up an appointment ASAP
- Lectures
  - No class on Nov. 5th and Nov. 12th
- Events of interest
  - November 13th: Jaswinder Pal Singh
    - Department colloquium: 1302 WWH, 11:30am-12:30pm
  - November 20th: Symposium on New Directions in Parallel and Concurrent Computing (PARCON’98)

Outline

- Last lecture: Programming for Performance
  - performance concerns common to all programming models
    - load-balance, synchronization, locality
  - data-parallel programming models
    - case study: Fortran D
    - data decomposition
    - compilation strategies
- This lecture: Programming for Performance (contd.)
  - message-passing programming models
    - components of communication cost
      - synchronization (matching), overhead, latency, BW, contention
    - synchronization: pre-posted receives, message pipelining, multithreading
    - overhead: message aggregation, low-overhead messaging layers
    - contention: scheduling communication

Message Passing: Communication Costs

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

- \(f\): frequency
- \(o\): overhead
- \(n_s/m\): avg msg size
- \(t_c\): contention
- \(B\): bandwidth

![Diagram showing communication cost components](attachment:image.png)
Inherent versus Artifactual Costs

\[ \text{communication cost} = f \left( \text{sync} - \text{overlap} + o + l + \frac{n_c}{B} + t_c \right) \]

- Reducing inherent costs
  - \( f \): frequency of communication
  - \( n_c/m \): 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: \((\text{sync} - \text{overlap})\)
      - overhead: \( o \)
      - contention: \( t_c \)

Reducing Communication Costs: Synchronization

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

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

```
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 );
```

Message Pipelining

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

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

Overlapping Communication with Local Operations

- **Idea**: Separate out sends and receives as much as possible
- **Advantage**: Overlap local synchronization time with computation

**Example (from Lecture 8)**

- SPMD code for non-local accesses in data-parallel programs
  - _owner-computes rule_

  ```plaintext
  SEND for non-local READ
  RECv for non-local READ
  compute IterSet
  SEND for non-local READ
  compute LocalIterSet
  RECV for non-local READ
  compute NonLocalReadIterSet
  
  in general: processors also execute iterations which write non-local data
  
  - LocalIterSet
  - NonLocalReadIterSet
  - NonLocalWriteIterSet
  ```

Multithreading

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

**Figure**

- 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

Message Aggregation

- **Idea**: Bundle together several small messages into a larger message
- **Advantage**: Amortize overhead over larger data transfers
  - works well if handshaking overhead dominates data transfer overhead (usually true)
- **Example**: Jacobi
  - sending a row at a time instead of single array elements
  
- Parallel computations with cross-processor dependencies
  - tradeoff between message aggregation (lower overhead) versus message pipelining (lower synchronization delay)
    - lot of research on good ways of making this decision
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

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>$T_0$ (µs)</th>
<th>$T_0$ (cycles)</th>
<th>$T_0$ (FLOPs)</th>
<th>B (MB/s)</th>
<th>num</th>
</tr>
</thead>
<tbody>
<tr>
<td>iPSC/2</td>
<td>1988</td>
<td>1</td>
<td>700</td>
<td>3600</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>2500</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>

Active Messages: An Example Program (Jacobi)

```
/* 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);

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++ ;
}

... 
num_received = 0;
/* send up */
AM_send( North, store_row_from_down, row[1] );
/* send down */
AM_send( South, store_row_from_up, row[n] );
/* wait until reqd messages have been received */
while ( num_received < 2 ) AM_extract();
... 
```
Active Messages: Performance

nCUBE/2 (vs. 150\(\mu\)s) vs. CM5 (vs. 95\(\mu\)s)

- Communication costs dominated by hardware access costs
  - minimal buffering costs
  - sender waits till network can accept message
  - minimal parsing costs (matching/selection)
  - handler executes immediately on arrival in order of reception

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)

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

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
Lecture Summary

- Programming for performance (contd.)
  - message-passing programming models
    - components of communication cost
      - synchronization (matching)
      - overhead
      - latency and BW
      - contention
    - reducing synchronization cost
      - pre-posted receives
      - message pipelining
      - multithreading
    - reducing overhead
      - message aggregation
      - low-overhead messaging layers: Active Messages, Fast Messages
    - reducing contention
      - scheduling communication

Next Lecture

- Programming for performance (contd.)
  - shared-memory models
    - dynamic task assignment
    - lock aggregation
    - layout to minimize cache conflicts
  - hardware-software tradeoffs
    - shared virtual address space
    - latency tolerance