Distributed Memory Multiprocessors I: Building Scalable Systems

Last Time: Finished discussion about bus-based shared memory machines for small-to-moderate sized systems.

Today: Introduce scalable design techniques for massively parallel, large scale machines.

Topics:

- Scalability
  - Key Issues: Physical limitations, bandwidth, latency, cost
  - Integration to support scalability
- Realizing Programming Models
  - Network transactions
  - Specific programming model issues
  - Common issues

Administrative Stuff

Announcements
Exam 1 will be on Thursday, 2/3/05

Projects
Get started with simulators/tools ASAP (see me if you have questions).
I will email you about your proposals.

Distributed Memory Multiprocessors I: Building Scalable Systems

What Is Scalability?

A system is scalable if it is capable of being increased in size, or more accurately, capable of delivering an increase in performance proportional to an increase in size.


This applies not just to not just bandwidth, but many other performance criteria, as we will see.
Comparing Interconnection Networks: Bus vs LAN

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Bus</th>
<th>LAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical length</td>
<td>Feet</td>
<td>Miles</td>
</tr>
<tr>
<td>Number of Connections</td>
<td>Fixed</td>
<td>Variable (Many)</td>
</tr>
<tr>
<td>Maximum Bandwidth</td>
<td>Fixed</td>
<td>Large</td>
</tr>
<tr>
<td>Interface to Processor</td>
<td>Memory interface</td>
<td>Peripheral</td>
</tr>
<tr>
<td>Global Order</td>
<td>Arbitration</td>
<td>None</td>
</tr>
<tr>
<td>Protection</td>
<td>V/P Translation</td>
<td>Entirely OS</td>
</tr>
<tr>
<td>Trust</td>
<td>Complete</td>
<td>None</td>
</tr>
<tr>
<td>Operating System</td>
<td>Single</td>
<td>Independent</td>
</tr>
<tr>
<td>Communication Abstraction</td>
<td>Hardware</td>
<td>Mostly Software</td>
</tr>
</tbody>
</table>

Bandwidth Scalability

Fundamental limit to bandwidth: independent physical wires
To increase bandwidth, we need lots of them!
These individual wire segments are probably linked together by switches.
The number of outputs (or inputs) of a switch is known as its degree.

Scalable Machines

What would we like to see ideally in a scalable machine as we add processing power?
- Bandwidth should increase,
- Latency (per operation) should not increase,
- Cost increases slowly,
- Construction should make sense physically.

Dancehall Organization

How does the required network bandwidth scale with the number of processors?
What about latency?
Generic Distributed Memory Organization

How does the required network bandwidth scale with the number of processors?
What about latency?

Achieving Bandwidth Scalability

Basic requirement is a large number of independent communication paths between nodes.
This allows a larger number of concurrent transactions (each using different set of wires).

In order to achieve this, we need to make some sacrifices:
• Transactions will now be initiated independently,
• No global arbitration and explicit ordering,
• Transactions are no longer globally visible,
  – Only visible to some subset of nodes,
  – Broadcasts may now be expensive,
  – Transactions spawn more transactions,

Latency Scaling

Transfer in a scalable network is given by:
\[ T(n) = \text{Overhead} + \text{Channel Time} + \text{Routing Delay} \]

Where:
• \textit{overhead} - time to initiate a transfer
• \textit{channel time} - time to move through the communication medium (n/B where B is bandwidth of thinnest channel)
• \textit{routing delay} - time for data to move through individual steps of journey (function of steps or hops, H, and data size, n)

Ideally, we would like for each of the above components to remain small as the system is scaled.

Example: Latency Scaling

Calculate transfer time for a 128B message on a 64 and 1024 node system if:
• overhead is 1μs
• link bandwidth is 64MB/s
• link distance is \( \log_2 n \) for any transfer between two nodes
• router delay is 200ns per hop
Cost Scaling

For large, high-performance machines, cost obviously plays an important role:

\[
\text{Cost}(p,m) = \text{Fixed Cost} + \text{Incremental Cost}(p,m)
\]

We would like to limit the incremental cost for adding additional processors and memory (same is true for disk storage).

\[
\text{Speedup}(p) = \frac{\text{Time}(1)}{\text{Time}(p)}
\]

\[
\text{Costup}(p) = \text{Cost}(p)\text{Cost}(1)
\]

Cost-Effective if \(\text{Speedup}(p) > \text{Costup}(p)\)

In bus based systems, the fixed cost includes, the cabinet, power supply, and the bus (which could support a full configuration).

Would this be scalable, cost-wise?

Physical Scaling

Modularity is essential in scalable systems.

Beyond that, there is little consensus on what is ideal for physical scale.

Rule of thumb: Within a given technology, longer links (wires) are slower.

Important concern: Power density and cooling requirements.

The communication architecture can be integrated into the design at many levels to allow tradeoffs in packaging technology, cost, and performance:

- Chip-Level Integration
- Board-Level Integration
- System-Level Integration

Example: nCUBE/2

Good example of chip-level integration with densely packed nodes.

Example: CM-5

Moderately dense packing with standard processor components integrated at board level.
**Example: IBM SP-1/SP-2**

Less dense packaging but significantly less engineering time.

Nodes are almost complete systems.

---

**Realizing Programming Models**

Let's take another look at our beloved programming model diagram...

---

**Revisiting Programming Models**

Programming model is the abstraction of the machine seen/used by the programmer.

Important programming models that we have seen so far:

- Multiprogramming
- Shared Address Space
- Message Passing
- Data parallel

Scalable machines are historically more closely tied to the message passing model.

But through design choices communication abstraction there are many more options.

---

**Network Transactions**

The basic communication primitive is now the network transaction.

This is simply a one-way transfer of information from a source output buffer to a destination input buffer which:

- causes action at the destination
- does not appear immediately visible at the source

This is analogous to the bus transaction in SMP machines, but obviously they are not quite the same.
Communication requires a two-way request/response protocol. 
Reads: 1) Request of Data 2) Response with Data 
Writes: 1) Announce Write 2) Acknowledge Write 

Design Issues:  
- Length and format of transfer (fixed/variable)  
- Naming and addressing (virtual/physical)  
- Deadlock avoidance  
- Coherence and consistency

In general, meeting consistency and coherency guidelines is more challenging in scalable systems.
You don’t have a globally visible place to broadcast and/or snoop writes.
You don’t have a global bus arbitration to serialize writes.
We will revisit these topics when we look at distributed shared memory systems.

Fetch Deadlock

Like the bus, interconnection networks are shared resources.
The network itself and/or destination nodes might be unable to handle a new request.
This could lead to fetch deadlock (as we have seen before):  
Node A is waiting for its read request to complete.
Node B has data for A, but it is waiting for A to answer its own read request.
Each node is blocked, waiting on data from the other.
To avoid this, each node must be able to process replies and requests even while it has outstanding requests.

Message Passing

The send/receive operations are minimally one-way bulk-transfers.
There is an explicit event ordering between communicating parties.
However, there are semantic variations that allow tradeoffs in performance and complexity:  
- Synchronous Message Passing  
- Asynchronous Message Passing (Multiple Flavors)
Synchronous Message Passing

![Synchronous Message Passing Diagram]

Kind of nice, deterministic approach. What are the performance limitations of this scheme? What about data buffering?

Asynchronous Message Passing: Conservative

![Asynchronous Message Passing: Conservative Diagram]

A more robust alternative which can be further optimized.

Asynchronous Message Passing: Optimistic

![Asynchronous Message Passing: Optimistic Diagram]

This is a less restrictive model, but there are still problems:
- Can be expensive to do tag match, buffer allocation, store-and-forward.
- Possibility of overflowing input buffer.

Active Messages

While shared-memory and message passing are dominant programming models, there are others. Active Messages is a low-level communication abstraction that mimics the network transaction. Functions as Remote Procedure Calls (RPC).
Common Challenges
Regardless of the programming model, there are some new (and old) common challenges.

We are now talking about very large systems where:

- Each processor has very limited knowledge about the system.
- Many network transactions can be in progress simultaneously.
- Source and destination of transactions are decoupled.

Common Challenges: Fetch Deadlock
To avoid fetch deadlock, nodes must continue to accept messages even when they cannot source their own.

Possible solutions:
- Provide two logically independent networks for requests and replies (physical or virtual channels).
- Limit number of outstanding transactions to bound requests and reserve input buffer space.
- Send NACK when buffer is full.

Common Challenges: Input Buffer Overflow
What happens when many sources contend for buffer space at a single destination (simultaneously)?

Possible solutions:
- Make input buffer very large (reduces probability of overflow, but doesn’t completely solve the problem).
- Reserve space for each source and have senders track their outstanding balance.
- Refuse input when buffer is full (can lead to back pressure).
- Use reservations/acknowledgement.

Summary
Scalable systems allow us to build very large systems with reasonable cost and good performance.

Network transaction is key communication primitive.

Many design choices in realizing programming models.