Lecture 8: Parallel Processing

Prof. Fred Chong
ECS 250A Computer Architecture
Winter 1999

(Adapted from Culler CS258 and Dally EE282)

Parallel Programming
- Motivating Problems (application case studies)
- Process of creating a parallel program
- What a simple parallel program looks like
  - three major programming models
  - What primitives must a system support?

Simulating Ocean Currents
- Model as two-dimensional grids
  - Discretize in space and time
  - finer spatial and temporal resolution  => greater accuracy
- Many different computations per time step
  - set up and solve equations
  - Concurrency across and within grid computations
- Static and regular

Simulating Galaxy Evolution
- Simulate the interactions of many stars evolving over time
- Computing forces is expensive
  - Hierarchical Methods take advantage of force law: $G \frac{m_1 m_2}{r^2}$
  - Many time-steps, plenty of concurrency across stars within one

Rendering Scenes by Ray Tracing
- Shoot rays into scene through pixels in image plane
- Follow their paths
  - they bounce around as they strike objects
  - they generate new rays: ray tree per input ray
- Result is color and opacity for that pixel
- Parallelism across rays
- How much concurrency in these examples?

Creating a Parallel Program
- Pieces of the job:
  - Identify work that can be done in parallel
  - work includes computation, data access and I/O
  - Partition work and perhaps data among processes
  - Manage data access, communication and synchronization
Definitions

- **Task**:
  - Arbitrary piece of work in parallel computation
  - Executed sequentially; concurrency is only across tasks
  - E.g., a particle/cell in Barnes-Hut, a ray or ray group in Raytrace
  - Fine-grained versus coarse-grained tasks

- **Process (thread)**:
  - Abstract entity that performs the tasks assigned to processes
  - Processes communicate and synchronize to perform their tasks

- **Processor**:
  - Physical engine on which process executes
  - Processes virtualize machine to programmer
    - write program in terms of processes, then map to processors

---

4 Steps in Creating a Parallel Program

- **Decomposition** of computation in tasks
- **Assignment of tasks to processes**
- **Orchestration** of data access, comm., synch.
- **Mapping** processes to processors

---

Decomposition

- Identify concurrency and decide level at which to exploit it
- Break up computation into tasks to be divided among processes
  - Tasks may become available dynamically
  - No. of available tasks may vary with time
- Goal: Enough tasks to keep processes busy, but not too many
  - Number of tasks available at a time is upper bound on achievable speedup

---

Limited Concurrency: Amdahl’s Law

- Most fundamental limitation on parallel speedup
- If fraction of seq execution is inherently serial, speedup < T/S
- Example: 2-phase calculation
  - sweep over n-by-n grid and do some independent computation
  - sweep again and add each value to global sum
- Time for first phase = \( n^2/p \)
- Second phase serialized at global variable, so time = \( n^2 \)
- Speedup = \( \frac{n^2}{n^2 + n^2/p} \), or at most 2
- Trick: divide second phase into two
  - accumulate into private sum during sweep
  - add per-process private sum into global sum
- Parallel time is \( n^2/p + n^2 + p \), and speedup at best \( \frac{2n^2}{2n^2 + n^2/p} \)

---

Understanding Amdahl’s Law

- Area under curve is total work done, or time with 1 processor
- Horizontal extent is lower bound on time (infinite processors)
- Amdahl’s law applies to any overhead, not just limited concurrency

---

Concurrency Profiles

- Area under curve is total work done, or time with 1 processor
- Horizontal extent is lower bound on time (infinite processors)
- Amdahl’s law applies to any overhead, not just limited concurrency
Orchestration
- Naming data
- Structuring communication
- Synchronization
- Organizing data structures and scheduling tasks temporally

- Goals
  - Reduce cost of communication and synch.
  - Preserve locality of data reference
  - Schedule tasks to satisfy dependences early
  - Reduce overhead of parallelism management

- Choices depend on Prog. Model., comm. abstraction, efficiency of primitives
- Architects should provide appropriate primitives efficiently

Mapping
- Two aspects:
  - Which process runs on which particular processor?
  - mapping to a network topology
  - Will multiple processes run on same processor?

  - space-sharing
    - Machine divided into subsets, only one app at a time in a subset
    - Processes can be pinned to processors, or left to OS

- System allocation
- Real world
  - User specifies desires in some aspects, system handles some
  - Usually adopt the view: process <-> processor

Parallelizing Computation vs. Data
- Computation is decomposed and assigned (partitioned)
  - Partitioning Data is often a natural view too
    - Computation follows data: owner computes
  - Grid example; data mining;

  - Distinction between comp. and data stronger in many applications
    - Barnes-Hut
    - Raytrace

Architect’s Perspective
- What can be addressed by better hardware design?
- What is fundamentally a programming issue?

High-level Goals

<table>
<thead>
<tr>
<th>Step</th>
<th>Architecture-Dependent?</th>
<th>Major Performance Goals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decomposition</td>
<td>Mostly no</td>
<td>Expose enough concurrency but not too much</td>
</tr>
<tr>
<td>Assignment</td>
<td>Mostly no</td>
<td>Balance workload</td>
</tr>
<tr>
<td>Orchestration</td>
<td>Yes</td>
<td>Reduce non-inherent communication via data locality</td>
</tr>
<tr>
<td>Reduce communication and synchronization cost as seen by the processor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reduce serialization at shared resources</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Schedule tasks to satisfy dependences early</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reduce overhead of parallelism management</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reduce cost of communication and synchronization</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Mapping | Yes | Put related processes on the same processor if necessary |
| Exploit locality in network topology |

What Parallel Programs Look Like

- High performance (speedup over sequential program)
- But low resource usage and development effort
- Implications for algorithm designers and architects?
Example: iterative equation solver

- Simplified version of a piece of Ocean simulation
- Illustrate program in low-level parallel language
  - C-like pseudocode with simple extensions for parallelism
  - Expose basic comm. and synch. primitives
  - State of most real parallel programming today

\[
A_{i,j} = 0.2 \times (A_{i,j} + A_{i,j-1} + A_{i-1,j} + A_{i,j+1} + A_{i+1,j})
\]

Expr ession for updating each interior point:

Grid Solver

- Gauss-Seidel (near-neighbor) sweeps to convergence
  - interior n-by-n points of (n+2)-by-(n+2) updated in each sweep
  - updates done in-place in grid
  - difference from previous value computed
  - accumulate partial diffs into global diff at end of every sweep
  - check if has converged
    » to within a tolerance parameter

Sequential Version

```
1. n ← 10, \text{init} = 0.
2. \text{main}()
3. \begin{align*}
   \text{begin} & \\
   \text{read}(n) & ; \text{/*read input parameter: matrix size*/} & \\
   \text{A} & \gets \text{malloc (a 2-d array of size n + 2 by n + 2 doubles);} & \\
   \text{initialize(A);} & \text{/*initialize the matrix A somehow*/} & \\
   \text{Solve(A);} & \text{/*call the routine to solve equation*/} & \\
   \text{end main} & \\
\end{align*}
```

```
11. \text{begin} & \\
12. \text{int i, j, done = 0;} & \\
13. \text{float diff = 0, temp;} & \\
14. \text{while (!done) do} & \text{/*outermost loop over sweeps*/} & \\
15. \text{diff = 0;} & \text{/*initialize maximum difference to 0*/} & \\
16. \text{for i ← 1 to n do} & \text{/*sweep over nonborder points of grid*/} & \\
17. \text{for j ← 1 to n do} & \\
18. \text{temp} & \text{\gets A[i,j];} & \\
20. \text{diff += abs(A[i,j] - temp);} & \\
21. \text{end for} & \\
22. \text{end for} & \\
23. \text{if ( \text{diff/(n*n)} < \text{TOL}) then done = 1;} & \\
24. \text{end while} & \\
25. \text{end procedure} & \\
```

Decomposition

- Simple way to identify concurrency is to look at loop iterations
  - dependence analysis; if not enough concurrency, then look further
- Not much concurrency here at this level (all loops sequential)
- Examine fundamental dependences

- Concurrency O(n) along anti-diagonals, serialization O(n) along diag.
- Retain loop structure, use pt-to-pt synch; Problem: too many synch ops.
- Restructure loops, use global synch; imbalance and too much\text{serial}

Exploit Application Knowledge

- Reorder grid traversal: red-black ordering
  - Different ordering of updates: may converge quicker or slower
  - Red sweep and black sweep are each fully parallel:
    - Global synch between them (conservative but convenient)
  - Ocean uses red-black
    - We use simpler, asynchronous one to illustrate
    - no red-black, simply ignore dependences within sweep
    - parallel program nonterministic

Decomposition

- Decomposition into elements: degree of concurrency n²
- Decompose into rows?
Assignment

• Static assignment: decomposition into rows
  - block assignment of rows: Row i is assigned to process i
  - cyclic assignment of rows: process i assigned rows i, i+p, ...

• Dynamic assignment
  - get a row index, work on the row, get a new row, ...

What is the mechanism?
- Concurrency? Volume of Communication?

Generating Threads

1. int n;
2. b.
3. a. float **A, diff; /*A is global (shared) array representing the grid*/

Data Parallel Solver

1. int n, nprocs; /*grid size (n+2 by n+2) and number of processes*/
2. float **A; /*A is entire n+2-by-n+2 shared array, as in the sequential program*/
3. procedure Solve(A)
4. Solve(A); /*call the routine to solve equation*/
5. initialize(A); /*initialize the matrix A somehow*/
6. float **A; /*A is an (n + 2-by-n + 2) array*/
7. main()
8. Solve (A); /*main process becomes a worker too*/
9. read(n); read (doubles);
10. begin
11. float **A; /*A is an entire n+2-by-n+2 shared array, as in the sequential program*/

Assignment Mechanism

1. procedure Solve(A)
2. Solve(A); /*main process becomes a worker too*/
3. G_MALLOC (nprocs); /*read input grid size and number of processes*/
4. read(n); read (doubles);
5. begin
6. float **A; /*A is entire n+2-by-n+2 shared array, as in the sequential program*/
7. procedure Solve(A)
8. Solve(A); /*main process becomes a worker too*/
9. G_MALLOC (nprocs); /*read input grid size and number of processes*/
10. read(n); read (doubles);
11. begin
12. float **A; /*A is entire n+2-by-n+2 shared array, as in the sequential program*/

Shared Address Space Solver

Single Program Multiple Data (SPMD)

1. procedure Solve(A)
2. Solve(A); /*main process becomes a worker too*/
3. G_MALLOC (nprocs); /*read input grid size and number of processes*/
4. read(n); read (doubles);
5. begin
6. float **A; /*A is entire n+2-by-n+2 shared array, as in the sequential program*/
7. procedure Solve(A)
8. Solve(A); /*main process becomes a worker too*/
9. G_MALLOC (nprocs); /*read input grid size and number of processes*/
10. read(n); read (doubles);
11. begin
12. float **A; /*A is entire n+2-by-n+2 shared array, as in the sequential program*/

SAS Program

• SPMD: not lockstep. Not necessarily same instructions
• Assignment controlled by values of variables used as loop bounds
  - unique pid per process, used to control assignment
• done condition evaluated redundantly by all
• Code that does the update identical to sequential program
  - each process has private mydiff variable
• Most interesting special operations are for synchronization
  - accumulations into shared diff have to be mutually exclusive
  - why the need for all the barriers?
• Good global reduction?
  - Utility of this parallel accumulate???
Mutual Exclusion

• Why is it needed?
  • Provided by LOCK-UNLOCK around critical section
    – Set of operations we want to execute atomically
      – Implementation of LOCK/UNLOCK must guarantee mutual excl.

• Serialization?
  – Contention?
    – Non-local accesses in critical section?
    – use private mydiff for partial accumulation!

Global Event Synchronization

• BARRIER(nprocs): wait here till nprocs processes get here
  – Built using lower level primitives
  – Global sum example: wait for all to accumulate before using sum
  – Often used to separate phases of computation

• Process P_i
  – set up eqn system
  – solve eqn system
  – apply results
  – Barrier (name, nprocs)
  – WAIT_FOR_END (nprocs-1)

Group Event Synchronization

• Subset of processes involved
  – Can use flags or barriers (involving only the subset)
    – Concept of producers and consumers

• Major types:
  – Single-producer, multiple-consumer
  – Multiple-producer, single-consumer

Message Passing Grid Solver

• Cannot declare A to be global shared array
  – compose it logically from per-process private arrays
  – usually allocated in accordance with the assignment of work
  – process assigned a set of rows allocates them locally

• Transfers of entire rows between traversals

• Structurally similar to SPMD SAS

• Orchestration different
  – data structures and data access/naming
  – communication
  – synchronization

• Ghost rows

Pt-to-pt Event Synch (Not Used Here)

• One process notifies another of an event so it can proceed
  – Concurrent programming on uniprocessor: semaphores
  – Shared address space parallel programs: semaphores, or use ordinary variables as flags

  \[ a = 0; \]
  \[ b = 0; \]
  \[ \text{while} (flag = 0) \text{ do nothing;} \]
  \[ b = 1; \]
  \[ \text{print } A \]

Data Layout and Orchestration

Data partition allocated per processor
- Add ghost rows to hold boundary data
- Send edges to neighbors
- Receive into ghost rows
- Compute as in sequential program
Notes on Message Passing Program

- Use of ghost rows
- Receive does not transfer data, send does
- sas which is usually receiver-initiated (read touches data)
- Communication done at beginning of iteration, so no asynchrony
- Communication in whole rows, not element at a time
- Core similar, but indices/bounds in local rather than global space
- Synchronization through sends and receives
  - Update of global diff and event synch for done condition
  - Could implement locks and barriers with messages
- Can use REDUCE and BROADCAST library calls to simplify code
  
  /* communicate local diff values and determine if done, using reduction and broadcast*/
  Send and Receive Alternatives

  Can extend functionality: stride, scatter-gather, groups
  Semantic flavors: based on when control is returned
  Aid when data structures or buffers can be reused at either end
  - Affect when data structures or buffers can be reused at either end
  - Affect ease of programming and performance
  - Synchronous messages provide built-in synch. through match
  - Separate event synchronization needed with asynch. messages
  - With synch. messages, our code is deadlocked. Fix?

Orchestration: Summary

- Shared address space
  - Shared and private data explicitly separate
  - Communication explicit in access patterns
  - No correctness need for data distribution
  - Synchronization via atomic operations on shared data
  - Synchronization explicit and distinct from data communication
- Message passing
  - Data distribution among local address spaces needed
  - No explicit shared structures (implicit in comm. patterns)
  - Communication is explicit
  - Synchronization implicit in communication (at least in synch. case)
  - Mutual exclusion by fiat

Correctness in Grid Solver Program

<table>
<thead>
<tr>
<th>SAS</th>
<th>MPI/Passing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exploit global data structure?</td>
<td>Yes</td>
</tr>
<tr>
<td>Assignment overlap data layout?</td>
<td>Yes</td>
</tr>
<tr>
<td>Communication</td>
<td>Implicit</td>
</tr>
<tr>
<td>Synchronization</td>
<td>Explicit</td>
</tr>
<tr>
<td>Explicit replication of border rows?</td>
<td>Yes</td>
</tr>
</tbody>
</table>

History of Parallel Architectures

- Parallel architectures tied closely to programming models
  - Divergent architectures, with no predictable pattern of growth
  - Mid 80s renaissance

## Send/Receive Alternatives

- Synchronous
- Asynchronous

## Send/Receive

- Blocking
- Nonblocking

## Application Software

- System Software
- Application Software

## Architecture

- System Software
- Application Software

## SIMD

- System Software
- Application Software

## Message Passing

- System Software
- Application Software

## Shared Memory

- System Software
- Application Software

## Dataflow

- System Software
- Application Software
**Convergence**

- Look at major programming models
  - where did they come from?
  - The 80s architectural renaissance!
  - What do they provide?
  - How have they converged?
- Extract general structure and fundamental issues
- Reexamine traditional camps from new perspective

**Programming Model**

- **Conceptualization of the machine that programmer uses in coding applications**
  - How parts cooperate and coordinate their activities
  - Specifies communication and synchronization operations
- **Multiprogramming**
  - no communication or synch. at program level
- **Shared address space**
  - like bulletin board
- **Message passing**
  - like letters or phone calls, explicit point to point
- **Data parallel:**
  - more regimented, global actions on data
  - Implemented with shared address space or message passing

**Structured Shared Address Space**

- Add hoc parallelism used in system code
- Most parallel applications have structured SAS
- Same program on each processor
  - shared variable X means the same thing to each thread

**Engineering: Intel Pentium Pro Quad**

- All coherence and multiprocessing glue in processor module
- Highly integrated, targeted at high volume
- Low latency and bandwidth

**Engineering: SUN Enterprise**

- Proc + mem card - I/O card
  - 16 cards of either type
  - All memory accessed over bus, so symmetric
  - Higher bandwidth, higher latency bus

**Scaling Up**

- Problem is interconnect: cost (crossbar) or bandwidth (bus)
- Dance-hall: bandwidth still scalable, but lower cost than crossbar
- Latencies to memory uniform, but uniformly large
- Distributed memory or non-uniform memory access (NUMA)
- Construct shared address space out of simple message transactions across a general-purpose network (e.g., read-request, read-response)
- Caching shared (particularly nonlocal) data?
Engineering: Cray T3E

- Scale up to 1024 processors, 480MB/s links
- Memory controller generates request message for non-local references
- No hardware mechanism for coherence
  - SGI Origin etc. provide this

Message Passing Architectures

- Complete computer as building block, including I/O
  - Communication via explicit I/O operations
- Programming model
  - Direct access only to private address space (local memory)
  - Communication via explicit messages (send/receive)
- High-level block diagram
  - Communication integration?
  - Mem, I/O, LAN, Cluster
  - Easier to build and scale than SAS
- Programming model more removed from basic hardware operations
  - Library or OS intervention

Message-Passing Abstraction

- Send specifies buffer to be transmitted and receiving process
- Recv specifies sending process and application storage to receive into
- Memory to memory copy, but need to name processes
- Optional tag on send and matching rule on receive
- User process names local data and entities in process/tag space too
- In simplest form, the send/receive match achieves pairwise synch event
  - Other variants too
- Many overheads: copying, buffer management, protection

Evolution of Message-Passing Machines

- Early machines: FIFO on each link
  - HW close to prog. Model;
  - Synchronous ops
  - Topology-central (hypercube algorithms)

Diminishing Role of Topology

- Shift to general links
  - DMA, enabling non-blocking ops
  - Buffered by system at destination until recv
  - Store/forward routing
- Diminishing role of topology
  - Any-to-any pipelined routing
  - Node-network interface dominates communication time
    - $H_{top} = N_{edge}$ vs $T_{top} + T_{edge} = N_{edge}$
  - Simplifies programming
  - Allows richer design space
    - Grids vs hypercubes

Example Intel Paragon

- Intel iPSC/1 -> iPSC/2 -> iPSC/860
- 4-way interleaved DRAM
- 8 bits, 175 MHz, bidirectional 2D grid network with processing node attached to every switch
- Intel Paragon node
- Intel’s new Paragon IP’s based hypercube
- Single processor node with hypercube connection to 8 x Paragon IP nodes
Building on the mainstream: IBM SP-2

- Made out of essentially complete RS6000 workstations
- Network interface integrated in I/O bus (limited by I/O bus)

Berkeley NOW

- 100 Sun Ultra2 workstations
- Intelligent network interface
  - proc + mem
- Myrinet Network
  - 160 MB/s per link
  - 300 ns per hop

Toward Architectural Convergence

- Evolution and role of software have blurred boundary
  - Send/recv supported on SAS machines via buffers
  - Can construct global address space on MP (GA --> P | LA)
  - Page-based (or finer-grained) shared virtual memory
- Hardware organization converging too
  - Tighter NI integration even for MP (low-latency, high-bandwidth)
  - Hardware SAS passes messages
- Even clusters of workstations/SMPs are parallel systems
  - Emergence of fast system area networks (SAN)
- Programming models distinct, but organizations converging
  - Nodes connected by general network and communication assists
  - Implementations also converging, at least in high-end machines

Programming Models Realized by Protocols

- Multiprogramming
- Parallel applications
- Communication abstractions

Shared Address Space Abstraction

- Fundamentally a two-way request/response protocol
  - writes have an acknowledgment
- Issues
  - fixed or variable length (bulk) transfers
  - remote virtual or physical address, where is action performed?
  - deadlock avoidance and input buffer full
  - coherent? consistent?

The Fetch Deadlock Problem

- Even if a node cannot issue a request, it must sink network transactions.
- Incoming transaction may be a request, which will generate a response.
- Closed system (finite buffering)
Consistency
- write-atomicity violated without caching

Key Properties of Shared Address Abstraction
- Source and destination data addresses are specified by the source of the request
- a degree of logical coupling and trust
- no storage logically “outside the address space”
- may employ temporary buffers for transport
- Operations are fundamentally request response
- Remote operation can be performed on remote memory
- logically does not require intervention of the remote processor

Message passing
- Bulk transfers
- Complex synchronization semantics
  - more complex protocols
  - More complex action
- Synchronous
  - Send completes after matching recv and source data sent
  - Receive completes after data transfer completes from matching send
- Asynchronous
  - Send completes after send buffer may be reused

Synchronous Message Passing
- Constrained programming model.
- Deterministic!
- Destination contention very limited.

Asynch. Message Passing: Optimistic
- More powerful programming model
- Wildcard receive => non-deterministic
- Storage required within msg layer?

Asynch. Msg Passing: Conservative
- Where is the buffering?
- Contention control? Receiver initiated protocol?
- Short message optimizations
Key Features of Msg Passing Abstraction

- Source knows send data address, dest. knows receive data address
- after handshake they both know both
- Arbitrary storage “outside the local address spaces”
  - may post many sends before any receives
  - non-blocking asynchronous sends reduces the requirement to an arbitrary number of descriptors
  - fine print says these are limited too
- Fundamentally a 3-phase transaction
  - includes a request / response
  - can use optimistic 1-phase in limited “Safe” cases
    - credit scheme

Active Messages

- User-level analog of network transaction
  - transfer data packet and invoke handler to extract it from the network and integrate with on-going computation
- Request/Reply
- Event notification: interrupts, polling, events?
- May also perform memory-to-memory transfer

Common Challenges

- Input buffer overflow
  - N-1 queue over-commit => must slow sources
  - reserve space per source (credit)
  - when available for reuse?
    - Ack at higher level
  - Refuse input when full
    - backpressure in reliable network
  - deadlock free
    - what happens to traffic not bound for congested dest?
  - Reserve ack back channel
  - drop packets
    - Utilize higher-level semantics of programming model

Challenges (cont)

- Fetch Deadlock
  - For network to remain deadlock free, nodes must continue accepting messages, even when cannot source image
  - what if incoming transaction is a request?
    - Each may generate a response, which cannot be sent!
  - What happens when internal buffering is full?
- logically independent request/reply networks
  - physical networks
    - virtual channels with separate input/output queues
  - bound requests and reserve input buffer space
    - K(P-1) requests + K responses per node
  - service discipline to avoid fetch deadlock?
- NACK on input buffer full
  - NACK delivery?

Challenges in Realizing Prog. Models in the Large

- One-way transfer of information
- No global knowledge, nor global control
  - barriers, scms, reduce, global-OR give fuzzy global state
- Very large number of concurrent transactions
  - Management of input buffer resources
    - many sources can issue a request and over-commit destination before any see the effect
  - Latency is large enough that you are tempted to “take risks”
    - optimistic protocols
  - large transfers
    - dynamic allocation
  - Many many more degrees of freedom in design and engineering of these system

Network Transaction Processing

- Key Design Issue:
  - How much interpretation of the message?
  - How much dedicated processing in the Comm. Assist?
**Spectrum of Designs**

- **None**: Physical bit stream
  - IBM, physical DMA
- **User/System**
  - User-level port
  - User-level handler
  - Monsoon, ...
- **Remote virtual address**
  - Processing, translation
  - CS-2
- **Global physical address**
  - Proc + Memory controller
  - RPS, BBN, T3D
- **Cache-to-cache**
  - Cache controller
  - Dash, KSR, Flash

Increasing HW Support, Specialization, Intrusiveness, Performance (???)

**Net Transactions: Physical DMA**

- DMA controlled by regs, generates interrupts
- **Physical ↔ OS initiates transfers**
  - Send-side
    - construct system “envelope” around user data in kernel area
  - Receive
    - must receive into system buffer, since no interpretation in CA

**nCUBE Network Interface**

- independent DMA channel per link direction
  - leave input buffers always open
  - segmented messages
- routing interprets envelope
  - dimension-order routing on hypercube
  - bit-serial with 36 bit cut-through

**Conventional LAN NI**

User Memory

- Host Memory
- NIC Controller
- DMA
- Addr
- Len
- Status
- Next
- Data

**User Level Ports**

- initiate transaction at user level
- deliver to user without OS intervention
- network port in user space
- User/system flag in envelope
  - protection check, translation, routing, media access in src CA
  - user/sys check in dest CA, interrupt on system

**User Level Network ports**

- Appears to user as logical message queues plus status
- What happens if no user push?
Example: CM-5

- Input and output FIFO for each network
- 2 data networks
- Tag per message
  - Index NI mapping table
- Context switching?
- *T integrated NI on chip
- iWARP also

User Level Handlers

- Hardware support to vector to address specified in message
  - Message ports in registers

J-Machine: Msg-Driven Processor

- Each node a small msg driven processor
- HW support to queue msgs and dispatch to msg handler task

Communication Comparison

- Message passing (active messages)
  - Interrupts (int-mp)
  - Polling (poll-mp)
  - Bulk transfer (bulk)
- Shared memory (sequential consistency)
  - Without prefetching (sm)
  - With prefetching (pre-sm)

Motivation

- Comparison over a range of parameters
  - Latency and bandwidth emulation
  - Hand-optimized code for each mechanism
  - 5 versions of 4 applications

The Alewife Multiprocessor
**Alewife Mechanisms**

- Int-mp -- 100-200 cycles Send/Rec ovhd
- Poll-mp -- saves 50-170 cycles Rec ovhd
- Bulk -- gather/scatter
- Sm -- 42-63 cycles + 1.6 cycles/hop
- Pre-sm -- 2 cycles, 16 entry buffer

**Applications**

- Irregular Computations
- Little data re-use
- Data driven

**Application Descriptions**

- EM3D 3D electromagnetic wave
- ICCG irreg sparse matrix solver
- Unstruc 3D fluid flow
- Moldyn molecular dynamics

**Performance Breakdown**

![Performance Breakdown Chart]

**Performance Summary**

![Performance Summary Chart]

**Traffic Breakdown**

![Traffic Breakdown Chart]
Traffic Summary

Effects of Bandwidth
- Lower bisection by introducing cross-traffic

Bandwidth Emulation
- Lower bisection by introducing cross-traffic

Sensitivity to Bisection

Effects of Latency
- Clock variation
  - processor has tunable clock
  - network is asynchronous
  - results in variations in relative latency
- Context switch on miss
  - add delay

Latency Emulation
Sensitivity to Latency

- Low overhead in shared memory performs well even with:
  - irregular, data-driven applications
  - little re-use

Communication Comparison Summary

- Low overhead in shared memory performs well even with:
  - irregular, data-driven applications
  - little re-use
- Bisection and latency can cause crossovers

Future Technology

- Technology changes the cost and performance of computer elements in a non-uniform manner
  - logic and arithmetic is becoming plentiful and cheap
  - wires are becoming slow and scarce
- This changes the tradeoffs between alternative architectures
  - superscalar doesn’t scale well
  - global control and data
- So what will the architectures of the future be?

Single-Chip Multiprocessors

- Build a multiprocessor on a single chip
  - linear increase in peak performance
  - advantage of fast interaction between processors
- But
  - memory bandwidth problem multiplied

Exploiting fine-grain threads

- Where will the parallelism come from to keep all of these processors busy?
  - ILP - limited to about 3
  - Outer-loop parallelism
    - e.g., domain decomposition
    - requires big problems to get lots of parallelism
- Fine threads
  - make communication and synchronization very fast (1 cycle)
  - break the problem into smaller pieces
  - more parallelism
**Processor with DRAM (PIM)**

- Put the processor and the main memory on a single chip
  - much lower memory latency
  - much higher memory bandwidths

- But
  - need to build systems with more than one chip

**Reconfigurable processors**

- Adapt the processor to the application
  - special function units
  - special wiring between function units

- Builds on FPGA technology
  - FPGAs are inefficient
    - a multiplier built from an FPGA is about 100x larger and 10x slower than a custom multiplier.
  - Need to raise the granularity
    - configure ALUs, or whole processors

- Memory and communication are usually the bottleneck
  - not addressed by configuring a lot of ALUs

**Summary**

- Parallelism is inevitable
  - ILP
  - Medium
  - Massive

- Commodity forces
  - SMPs
  - NOWs, CLUMPs

- Technological trends
  - MP chips
  - Intelligent memory

**EPIC - explicit (instruction-level) parallelism aka VLIW**

- Compiler schedules instructions
- Encodes dependencies explicitly
  - saves having the hardware repeatedly rediscover them

- Support speculation
  - speculative load
  - branch prediction

- Really need to make communication explicit too
  - still has global registers and global instruction issue