Introduction

Copyright 2005 Mark D. Hill
University of Wisconsin-Madison

Slides are derived from work by Sarita Adve (Illinois), Babak Falsafi (CMU), Alvy Lebeck (Duke), Steve Reinhardt (Michigan), J. P. Singh (Princeton), and Jim Smith (Wisconsin). Thanks!

Outline

- Motivation & Applications
- Theory, History, & A Generic Parallel Machine
- UMA, NUMA, Examples, & More History
- Programming Models
  - Shared Memory
  - Message Passing
  - Data Parallel
- Issues in Programming Models
  - Function: naming, operations, & ordering
  - Performance: latency, bandwidth, etc.
Motivation

- CS/ECE 752 is about computers with one processor

- This course uses N processors in a computer to get
  - Higher Throughput via many jobs in parallel
  - Improved Cost-Effectiveness (e.g., adding 3 processors may yield 4X throughput for 2X system cost)
  - To get Lower Latency from shrink-wrapped software (e.g., databases and web servers today, but more tomorrow)
  - Lower latency through Parallelizing your application (but this is hard)

- Need faster than today’s microprocessor?
  - Wait for tomorrow’s microprocessor
  - Use many microprocessors in parallel

Applications: Science and Engineering

- Examples
  - Weather prediction
  - Evolution of galaxies
  - Oil reservoir simulation
  - Automobile crash tests
  - Drug development
  - VLSI CAD
  - Nuclear BOMBS!

- Typically model physical systems or phenomena
- Problems are 2D or 3D
- Usually requires “number crunching”
- Involves “true” parallelism
Applications: Commercial

• Examples
  – On-line transaction processing (OLTP)
  – Decision support systems (DSS)
  – “app servers”

• Involves data movement, not much number crunching
  – OTLP has many small queries
  – DSS has fewer large queries

• Involves throughput parallelism
  – inter-query parallelism for OLTP
  – intra-query parallelism for DSS

Applications: Multi-media/home

• Examples
  – speech recognition
  – data compression/decompression
  – 3D graphics

• Will become ubiquitous

• Involves everything (crunching, data movement, true parallelism, and throughput parallelism)
In Theory

- **Sequential**
  - Time to sum n numbers? $O(n)$
  - Time to sort n numbers? $O(n \log n)$
  - What model? RAM

- **Parallel**
  - Time to sum? Tree for $O(\log n)$
  - Time to sort? Non-trivially $O(\log n)$
  - What model?
    - PRAM [Fortune Willie STOC78]
    - P processors in lock-step
    - One memory (e.g., CREW for concurrent read exclusive write)

---

Perfection: the PRAM Model

- Parallel RAM
- Fully shared memory
- Unit latency
- No memory contention (unrestricted bandwidth)
- Data placement unimportant
Perfection is NOT Achievable

- Latencies grow as the system size grows
- Bandwidths are restricted by memory organizations and interconnection networks
- Dealing with reality leads to division between
  
  * UMA: Uniform Memory Access
  * NUMA: Non-Uniform Memory Access

But in Practice, How Do You

- Name a datum (e.g. all say a[I])?
- Communicate values?
- Coordinate and synchronize?
- Select processing node size (few-bit ALU to a PC)?
- Select number of nodes in system?
### Historical View

<table>
<thead>
<tr>
<th>Join At:</th>
<th>Program With:</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O (Network)</td>
<td>Message Passing</td>
</tr>
<tr>
<td>Memory</td>
<td>Shared Memory</td>
</tr>
<tr>
<td>Processor</td>
<td>(Dataflow/Systolic), Single-Instruction Single-Data (SIMD)</td>
</tr>
</tbody>
</table>

===> Data Parallel

### Historical View, cont.

- **Machine --> Programming Model**
  - Join at network so program with message passing model
  - Join at memory so program with shared memory model
  - Join at processor so program with SIMD or data parallel

- **Programming Model --> Machine**
  - Message-passing programs on message-passing machine
  - Shared-memory programs on shared-memory machine
  - SIMD/data-parallel programs on SIMD/data-parallel machine

- **But**
  - Isn’t hardware basically the same? Processors, memory, & I/O?
  - Why not have generic parallel machine & program with model that fits the problem?
A Generic Parallel Machine

- Separation of programming models from architectures
- All models require communication
- Node with processor(s), memory, communication assist

Today’s Parallel Computer Architecture

- Extension of traditional computer architecture to support communication and cooperation
  - Communications architecture

User Level
- Multiprogramming
- Shared Memory
- Message Passing
- Data Parallel

System Level
- Operating System Support
- Communication

Hardware/Software Boundary

Library and Compiler

(C) 2005 Mark D. Hill from Adve, Falsafi, Lebeck, Reinhardt, Smith & Singh
Simple Problem

for i = 1 to N
    A[i] = (A[i] + B[i]) * C[i]
    sum = sum + A[i]

• How do I make this parallel?

Simple Problem

for i = 1 to N
    A[i] = (A[i] + B[i]) * C[i]
    sum = sum + A[i]

• Split the loops
  » Independent iterations
  for i = 1 to N
    A[i] = (A[i] + B[i]) * C[i]
  for i = 1 to N
    sum = sum + A[i]

• Data flow graph?
Data Flow Graph

2 + N-1 cycles to execute on N processors
what assumptions?

Partitioning of Data Flow Graph

global synch
**Programming Model**

- Historically, Programming Model == Architecture
- Thread(s) of control that operate on data
- Provides a communication abstraction that is a contract between hardware and software (ala ISA)

**Current Models**
- Shared Memory
- Message Passing
- Data Parallel (Shared Address Space)
- (Data Flow)

**Shared (Physical) Memory**

- Communication, sharing, and synchronization with store / load on shared variables
- Must map virtual pages to physical page frames
- Consider OS support for good mapping
Shared (Physical) Memory on Generic MP

Node 0 0,N-1  
Node 1 N,2N-1  
Node 2 2N,3N-1  
Node 3 3N,4N-1

Keep private data and frequently used shared data on same node as computation

Return of The Simple Problem

```c
private int i, my_start, my_end, mynode;
shared float A[N], B[N], C[N], sum;
for i = my_start to my_end
    A[i] = (A[i] + B[i]) * C[i]
GLOBAL_SYNCH;
if (mynode == 0)
    for i = 1 to N
        sum = sum + A[i]
```

- Can run this on any shared memory machine
**Message Passing Architectures**

Node 0: \(0, N-1\)  
Node 1: \(0, N-1\)  
Node 2: \(0, N-1\)  
Node 3: \(0, N-1\)

- Cannot directly access memory on another node
- IBM SP-2, Intel Paragon
- Cluster of workstations

**Message Passing Programming Model**

- **User level send/receive abstraction**
  - local buffer \((x, y)\), process \((Q, P)\) and tag \(t\)
  - naming and synchronization
The Simple Problem Again

```c
int i, my_start, my_end, mynode;
float A[N/P], B[N/P], C[N/P], sum;
for i = 1 to N/P
    A[i] = (A[i] + B[i]) * C[i]
    sum = sum + A[i]
if (mynode != 0)
    send (sum,0);
if (mynode == 0)
    for i = 1 to P-1
        recv(tmp,i)
        sum = sum + tmp
• Send/Recv communicates and synchronizes
• P processors
```

Separation of Architecture from Model

• At the lowest level SM sends messages
  – HW is specialized to expedite read/write messages
• What programming model / abstraction is supported at user level?
• Can I have shared-memory abstraction on message passing HW?
• Can I have message passing abstraction on shared memory HW?
Data Parallel

- **Programming Model**
  - operations are performed on each element of a large (regular) data structure in a single step
  - arithmetic, global data transfer
- **Processor is logically associated with each data element**
- **Early architectures directly mirrored programming model**
  - Many, bit serial processors
- **Today we have FP units and caches on microprocessors**
- **Can support data parallel model on SM or MP architecture**

---

The Simple Problem Strikes Back

Assuming we have N processors

\[ A = (A + B) * C \]
\[ \text{sum} = \text{global}_\text{sum} (A) \]

- Language supports array assignment
- Special HW support for global operations
- CM-2 bit-serial
- CM-5 32-bit SPARC processors
  - Message Passing and Data Parallel models
  - Special control network
- Chapter 11.....
Data Flow Architectures

- Explicitly represent data dependencies (Data Flow Graph)
- No artificial constraints, like sequencing instructions!
  - Early machines had no registers or cache
- Instructions can “fire” when operands are ready
  - Remember tomasulo’s algorithm
- How do we know when operands are ready?
- Matching store
  - large associative search!
- Later machines moved to coarser grain (threads)
  - allowed registers and cache for local computation
  - introduced messages (with operations and operands)

Execute Data Flow Graph
No control sequencing
Review: Separation of Model and Architecture

- **Shared Memory**
  - Single shared address space
  - Communicate, synchronize using load / store
  - Can support message passing

- **Message Passing**
  - Send / Receive
  - Communication + synchronization
  - Can support shared memory

- **Data Parallel**
  - Lock-step execution on regular data structures
  - Often requires global operations (sum, max, min...)
  - Can support on either SM or MP

Review: A Generic Parallel Machine

- Separation of programming models from architectures
- All models require communication
- Node with processor(s), memory, communication assist

Let’s see UMA, NUMA, Examples, & More History
UMA: Uniform Memory Access

- Latencies are the same,
  - *but may be relatively high*
- Latencies get worse as system grows
  => scaling difficulties

Uniform Memory Access

- Data placement unimportant
- Typically used in small MPs only
- Contention restricts bandwidth
- Caches are often "allowed" in UMA systems
- Also called symmetric multiprocessors (SMP)
Example: Intel Pentium Pro Quad

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

NUMA: NonUniform Memory Access

- Latency low to local memory
- Latency much higher to remote memories
- Performance very sensitive to data placement
- Bandwidth to local memory may be higher
- Contention in network and for memories
NUMA Multiprocessors, contd.

- Distributed shared memory
  - One logical address space
  - Can be treated as shared memory
- Multicomputers
  - Each processor has its own memory address space
  - Use message passing for communication

Example: Cray T3E

- Scale up to 1024 processors, 480MB/s links
- Memory controller generates comm. request for nonlocal references
- No hardware mechanism for coherence (SGI Origin etc. provide this)
Historical Evolution: 1960s & 70s

- Early MPs
  - Mainframes
  - Small number of processors
  - crossbar interconnect
  - UMA

Historical Evolution: 1980s

- Bus-Based MPs
  - enabler: processor-on-a-board
  - economical scaling
  - precursor of today's SMPs
  - UMA
Historical Evolution: Late 80s, mid 90s

- Large Scale MPs (Massively Parallel Processors)
  - multi-dimensional interconnects
  - each node a computer (proc + cache + memory)
  - both shared memory and message passing versions
  - NUMA
  - not commercially viable
  - still used for “supercomputing”

Historical Evolution: Current

- Small to Mid-Scale SMPs
  - One module type: processor + caches + memory

- Clusters
  - Use high performance LAN to connect small SMPs

- Driven by economics
  - Smaller systems => higher volumes
  - Off-the-shelf components

- Driven by applications
  - Many more throughput applications (web servers)
  - Than parallel applications (weather prediction)
Historical Taxonomy (Flynn)

- **SISD: Single Instruction, Single Data**
- Operand and instruction storage may be the same
- Your basic uniprocessor

![Diagram of SISD]

- **SIMD: Single Instruction, Multiple Data**
  - Insts and data storage usually separated
  - Leads to "Data Parallel" programming model
  - Works better for loop-oriented numerical problems
  - Automatic parallelization can work

![Diagram of SIMD]
Historical Taxonomy (Flynn)

- **MIMD**: Multiple Instruction, Multiple Data
- **More flexible than SIMD and of more interest to us**
- **Important for general purpose computing**
- **Automatic parallelization more difficult**

![Diagram of a computer's instruction streams, execution units, and storage units.](image)

Programming Model Design Issues

- **Naming**: How is communicated data and/or partner node referenced?
- **Operations**: What operations are allowed on named data?
- **Ordering**: How can producers and consumers of data coordinate their activities?
- **Performance**
  - **Latency**: How long does it take to communicate in a protected fashion?
  - **Bandwidth**: How much data can be communicated per second? How many operations per second?
**Issue: Naming**

- **Single Global Linear-Address-Space** (shared memory)
- **Single Global Segmented-Name-Space** (global objects)
- **Multiple Local Address/Name Spaces** (message passing)
- **Naming strategy affects**
  - Programmer / Software
  - Performance
  - Design Complexity

**Issue: Operations**

- **Uniprocessor RISC**
  - ld/st and atomic operations on memory
  - arithmetic on registers
- **Shared Memory Multiprocessor**
  - ld/st and atomic operations on local/global memory
  - arithmetic on registers
- **Message Passing Multiprocessor**
  - send/receive on local memory
  - broadcast
- **Data Parallel**
  - ld/st
  - Global operations (add, max, etc.)
Issue: Ordering

- **Uniprocessor**
  - programmer sees order as program order (even if implementation executes things out of order)

- **Data Parallel**
  - Total order (one logical program counter)

- **Shared Memory**
  - What is order among several threads accessing shared data?
  - Memory consistency model (e.g., sequential consistency)

- **Message Passing**
  - Partial Order
  - A<B<C, D<E<F, B<D
  - Implies: A<B<D<F

Issue: Order/Synchronization

- **Coordination mainly takes three forms:**
  - mutual exclusion (e.g., spin-locks)
  - event notification
    - point-to-point (e.g., producer-consumer)
    - global (e.g., end of phase indication, all or subset of processes)
  - global operations (e.g., sum)

- **Issues:**
  - synchronization name space (entire address space or portion)
  - granularity (per byte, per word, ... => overhead)
  - low latency, low serialization (hot spots)
Performance Issue: Latency

- Must deal with latency when using fast processors
- Options:
  - Reduce frequency of long latency events
    - algorithmic changes, computation and data distribution
  - Reduce latency
    - cache shared data, network interface design, network design
  - Tolerate latency
    - message passing overlaps computation with communication (program controlled)
    - SM overlaps access completion and computation using consistency model and prefetching

Performance Issue: Bandwidth

- Private and global bandwidth requirements
- Private bandwidth requirements can be supported by:
  - distributing main memory among PEs
  - application changes, local caches, memory system design
- Global bandwidth requirements can be supported by:
  - scalable interconnect technology
  - distributed main memory and caches
  - efficient network interfaces
  - avoiding contention (hot spots) through application changes
Cost of Communication

Cost = Frequency x (Overhead + Latency + Xfer size/BW - Overlap)

- **Frequency = # communications per unit work**
  - algorithm, placement, replication, bulk data transfer
- **Overhead = processor cycles spent handling**
  - protection checks, status, buffer mgmt, copies, events
- **Latency = time to move bits from source to dest**
  - comm assist, topology, routing, congestion
- **Transfer time = time through bottleneck**
  - comm assist, links, congestions
- **Overlap = portion overlapped with useful work**
  - comm assist, comm operations, processor design

Summary

- **Motivation & Applications**
- **Theory, History, & A Generic Parallel Machine**
- **UMA, NUMA, Examples, & More History**
- **Programming Models**
  - Shared Memory
  - Message Passing
  - Data Parallel
- **Issues in Programming Models**
  - Function: naming, operations, & ordering
  - Performance: latency, bandwidth, etc.