Fundamental Design Issues

for Parallel Architecture

Todd C. Mowry
CS 418
January 13, 2011

Understanding Parallel Architecture

Traditional taxonomies not very useful
Programming models not enough, nor hardware structures
• Some one can be supported by radically different architectures

Architectural distinctions that affect software
• Compilers, libraries, programs
Design of user/system and hardware/software interface
• Constrained from above by programming models and below by technology

Guiding principles provided by layers
• What primitives are provided at communication abstraction
• How programming models map to these
• How they are mapped to hardware

Fundamental Design Issues

At any layer, interface (contract) aspect and performance aspects
• Naming: How are logically shared data and/or processes referenced?
• Operations: What operations are provided on these data
• Ordering: How are accesses to data ordered and coordinated?
• Replication: How are data replicated to reduce communication?
• Communication Cost: Latency, bandwidth, overhead, occupancy

Understand at programming model first, since that sets requirements

Other issues:
• Node Granularity: How to split between processors and memory?
• Heterogeneity: Are all processors equally powerful?

Sequential Programming Model

Contract
• Naming: Can name any variable in virtual address space
  • Hardware (and perhaps compilers) does translation to physical addresses
• Operations: Loads and Stores
• Ordering: Sequential program order

Performance
• Rely on dependences on single location (mostly): dependence order
• Compilers and hardware violate other orders without getting caught
• Compiler: reordering and register allocation
• Hardware: out of order, pipeline bypassing, write buffers
• Transparent replication in caches
**Shared Address Space Programming Model**

**Naming:**
- Any process can name any variable in shared space

**Operations:**
- Loads and stores, plus those needed for ordering

**Simplest Ordering Model:**
- Within a process/thread: sequential program order
- Across threads: some interleaving (as in time-sharing)
- Additional orders through synchronization
- Again, compilers/hardware can violate orders without getting caught
  - Different, more subtle ordering models also possible (discussed later)

**Synchronization**

**Mutual exclusion (locks):**
- Ensure certain operations on certain data can be performed by only one process at a time
- Analogy: like a room that only one person can enter at a time
- No ordering guarantees

**Event synchronization:**
- Ordering of events to preserve dependences
  - e.g. producer → consumer of data
- 3 main types:
  - point-to-point
  - global
  - group

**Message Passing Programming Model**

**Naming:** Processes can name private data directly.
- No shared address space

**Operations:** Explicit communication via send and receive
- Send transfers data from private address space to another process
- Receive copies data from process to private address space
- Must be able to name processes

**Ordering:**
- Program order within a process
- Send and receive can provide pt-to-pt synch between processes
- Mutual exclusion inherent

**Can construct global address space:**
- Process number + address within process address space
- But no direct operations on these names

**Design Issues Apply at All Layers**

Programming model’s position provides constraints/goals for system
In fact, each interface between layers supports or takes a position on:
- Naming model
- Set of operations on names
- Ordering model
- Replication
- Communication performance

Any set of positions can be mapped to any other by software
Let’s see issues across layers:
- How lower layers can support contracts of programming models
- Performance issues
Naming and Operations

Naming and operations in programming model can be directly supported by lower levels, or translated by compiler, libraries or OS.

Example: Shared virtual address space in programming model

1. Hardware interface supports shared physical address space
   - Direct support by hardware through v-to-p mappings, no software layers

2. Hardware supports independent physical address spaces
   - Can provide SAS through OS, so in system/user interface
     - v-to-p mappings only for data that are local
     - remote data accessed incur page faults; brought in via page fault handlers
     - same programming model, different hardware requirements and cost model
   - Or through compilers or runtime, so above sys/user interface
     - shared objects, instrumentation of shared accesses, compiler support

Naming and Operations (Cont)

Example: Implementing Message Passing

1. Direct support at hardware interface
   - But match and buffering benefit from more flexibility

2. Support at system/user interface or above in software (almost always)
   - Hardware interface provides basic data transport (well suited)
   - Send/receive built in software for flexibility (protection, buffering)
   - Choices at user/system interface:
     - OS each time: expensive
     - OS sets up once/infrequently, then little software involvement each time
   - Or lower interfaces provide SAS, and send/receive built on top with buffers and loads/stores

Need to examine the issues and tradeoffs at every layer
   - Frequencies and types of operations, costs

Ordering

Message passing: no assumptions on orders across processes except those imposed by send/receive pairs

SAS: How processes see the order of other processes' references defines semantics of SAS
   - Ordering very important and subtle
   - Uniprocessors play tricks with orders to gain parallelism or locality
   - These are more important in multiprocessors
   - Need to understand which old tricks are valid, and learn new ones
   - How programs behave, what they rely on, and hardware implications

Replication

Very important for reducing data transfer/communication
Again, depends on naming model

Uniprocessor: caches do it automatically
   - Reduce communication with memory

Message Passing naming model at an interface
   - A receive replicates, giving a new name; subsequently use new name
   - Replication is explicit in software above that interface

SAS naming model at an interface
   - A load brings in data transparently, so can replicate transparently
   - Hardware caches do this: e.g., in shared physical address space
   - OS can do it at page level in shared virtual address space, or objects
   - No explicit renaming, many copies for same name: coherence problem
     - in uniprocessors, "coherence" of copies is natural in memory hierarchy
**Communication Performance**

Performance characteristics determine usage of operations at a layer
- Programmer, compilers etc make choices based on this

Fundamentally, three characteristics:
- **Latency**: time taken for an operation
- **Bandwidth**: rate of performing operations
- **Cost**: impact on execution time of program

If processor does one thing at a time: bandwidth $\propto 1$/latency
- But actually more complex in modern systems

Characteristics apply to overall operations, as well as individual components of a system, however small
We will focus on communication or data transfer across nodes

**Communication Cost Model**

Communication Time per Message

$= \text{Overhead} \times \text{Assist Occupancy} \times \text{Network Delay} \times \text{Size/Bandwidth} \times \text{Contention}$

$= o_o \times o_o \times n/B \times T_c$

Overhead and assist occupancy may be $f(n)$ or not

Each component along the way has occupancy and delay
- Overall delay is sum of delays
- Overall occupancy (1/bandwidth) is biggest of occupancies

Comm Cost = frequency * (Comm time - overlap)
General model for data transfer: applies to cache misses too

**Summary of Design Issues**

Functional and performance issues apply at all layers

Functional: Naming, operations and ordering

Performance: Organization, latency, bandwidth, overhead, occupancy

Replication and communication are deeply related
- Management depends on naming model

Goal of architects: design against frequency and type of operations that occur at communication abstraction, constrained by tradeoffs from above or below
- Hardware/software tradeoffs

**Recap**

Parallel architecture is now mainstream

Exotic designs have contributed much, but given way to convergence
- Push of technology, cost and application performance
- Basic processor-memory architecture is the same
- Key architectural issue is in communication architecture

Fundamental design issues:
- Functional: naming, operations, ordering
- Performance: organization, replication, performance characteristics

Design decisions driven by workload-driven evaluation
- Integral part of the engineering focus