Federated Systems

Lecture 7
Federated Applications
Rich Martin
October 19th 1999

Today

- Applications on Federated Systems
  - I/O kernel evaluation, Arpaci-Dusseau
  - Porcupine (Mail server), Satio
  - Inktomi (Search Engine), Brewer
  - Scalable Distributed Data Structures, Gribble
- Why are these application interesting?
- What do they have in common?

Today (cont)

Common Themes

- Distributed State
- Parallel algorithms and data structures
  - partitioning, orchestration
  - locality management
- Novel (weak) consistency semantics
  - Sometimes, users just don't care that much

Application Areas

Application domains

- DataBases
  - sorting, scan
- Scientific
  - Transpose (FFT, Matrix Mul)
- Internet Services
  - Inverted Index
  - Mail

Ouline

Way too much material today

- Could spend a whole day on 1 system!
- Cover the big points only

Each of these systems deal with many complex and subtle issues

- Impossible do them justice in 2 hours

Walk away with better appreciation and ability to focus on similarities and differences

Architectural Evaluation

Big Idea: compare 3 machines on 3 common kernels (not full applications)

- Scan
- Sorting
- Transpose

What are the inherent strengths/weakness of these architectures (from a performance perspective)

EvaluationStrategy

Provide analytic models of kernels

Realize kernels on hardware (write code)

- Uniprocessor Workstation
- Cluster
- SMP

All platforms have same CPU and 1st level memory systems (167MHz Ultra1 & UPA)

Experiment Design

Examine resources (I/O bus, memory bus, CPU) in turn and discover:

- Modeled demand based on algorithm
- Actual demand of implementation
- Explain discrepancy

Experiment design is critical to systems research!

Experimentation

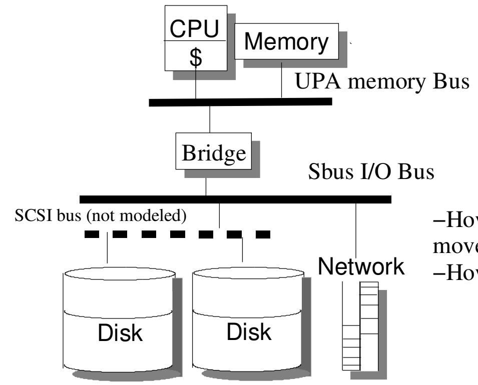
CS experiment culture is sloppy

Care in machine interface usage

- VIS memcpy()
- mmap() + madvise() for reading
- 64 KB write() system call for writing

Uses interfaces that naturally map to the kernels and also deliver high performance.

Building Block Model



–How much data must move across the mem bus?
–How much does?

–How much data must move across the I/O Bus?
–How much does?

Careful vs sloppy

Many things than can result in a 2x slowdown:

- Use of glibc vs. vendor libc
- Double buffering on read()/write() system calls
- Moving whole records vs. indexing
- compiler flags (inline, –O3 etc)
- Sum can result in 10X performance loss!

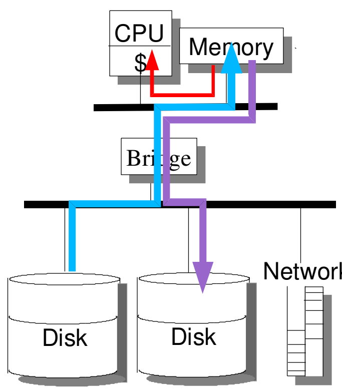
Unrealistic "Hyper–tuning" also possible!

- Still an interesting exercise, what's possible w. lots of effort vs. some effort vs. lazy.

Abstract Kernels

- Recall
  - Scan
  - Sort
  - Transpose

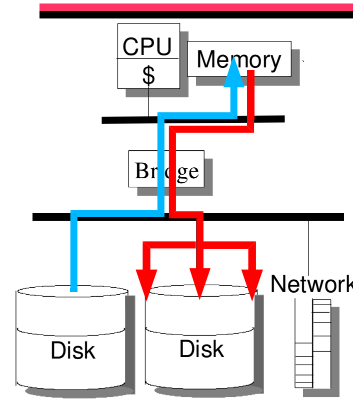
Database Scan



1. Read records from disk
2. Compare keys
   a. gather?

3. Write Records to disk

2 pass external sort



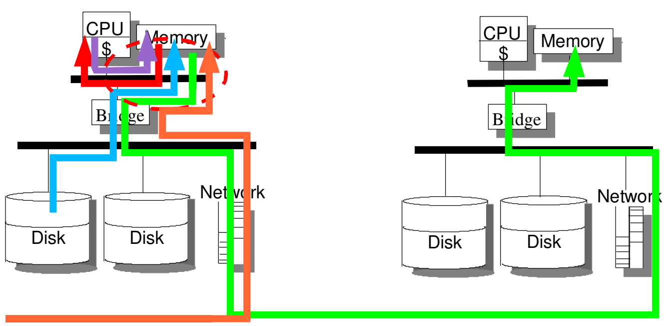
Pass1

A. Read records from disk
B. Create key/pointer pairs
C. Sort, create sorted runs
D. Write runs to disk

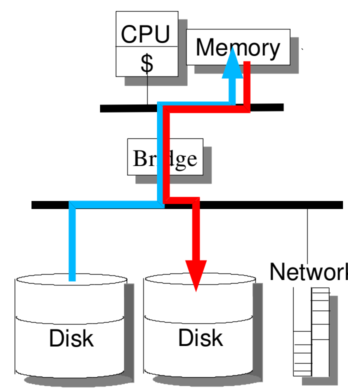
Pass2

A. Read tops of runs from disk
B. Select highest keyfrom each run
C. Write one long stream to disk

Parallel Sorting



Transpose



1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16

Cluster Data Movement

Much data movement in the system

- ~8D across the memory bus
- ~3D across the I/O bus

Sun's UPA can sink this easy

- What about PC memory Bus?
  - Serious arch. differences here
  - What is the box optimized for?

Technology trends as disk BW increases?

SMP Data Movement

Fundamentally less data movement

- if you are careful!

Tighter coupling of machine components

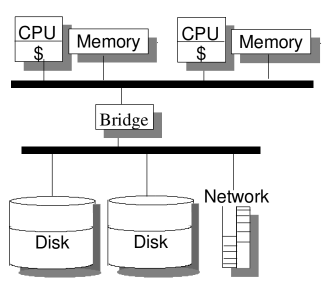
But....

Harder to balance!

- e.g. N cluster nodes = Nx more mem BW

Interent trade–off of SMP vs cluster?

Federated Potential?



Use disk processor to scatter data based on key

Can also build index/key pairs

Actively write runs (working ahead)?

Porcupine

- Big Idea: Implement an interesting internet service (Mail)
  - Discover systems issues along the way
  - Interesting system architecture
    - Lots of work to figure out from the paper what's going on
    - But it can be done

Porcupine Universe

System defined by a current group

- Hard state can't be lost.

Two kinds of hard objects:

  - Mail (in fragments)
  - User profiles (user name, passwd)

Everything else can be reconstructed

- Reconstruction happens on membership change

Tour

What happens on a reconfiguration?

- Q: How soft–state is reconstructed

1. Node notices membership change

- New epoch is started
- New membership list for epoch is generated
- Uses broadcast
  - Broadcast is a nice way out for hard systems problems

Broadcast Aside

- TRM protocols (and many membership protocols) use broadcast
  - What if you don't have it?
  - Can you synthesize in a reliable way from point–to–point?
- Rely on hardware (layer 2) broadcast
  - Scalability limit? 100? 500? 1000?
  - Fault–tolerant, fast broadcasts?

Tour (cont)

Coordinator generates new membership list

Broadcast to all nodes

- Also gets a new user list

Regenerate the user map

  - (user–> bucket–> node)

user map is broadcast to all nodes

Tour (cont)

All nodes now have updated user maps

Observe changes to hash buckets

Scan my state for changes

- What users don't I own?
- That have changes since last epoch?

Send info to owning node

- Owning node manages soft state

Porcupine Objects

Hard state is maintained in "Objects"

- DMBS–like semantics: must live on disk

Weaker semantics than a DBMS on when an object (record) is committed

- Might see an object twice

System will eventually converge

- Timestamps (in real time) used to resolve objects

Replication

Why replicate?

- Don't want to lose hard state (user profile + mail)

Replication complicates object protocol

- Reading is easy, writes are hard
- lower semantic level than ACID make a little easier

Also stresses performance

- More messages in internal interconnect

Architectural Comparison

- Porcupine system is disk starved
- Disk writes are the limiting factor!
  - Dismal performance shows more/disk per node needed
  - What would happen if there were
    - 5 disks per node? 10? 32?
    - Table

Porcupine: Cluster vs. SMP?

Build on a large SMP?

- Easier to scale?
- One giant multi–process application?

What about 2 large SMPs?

- Current system architecture for most large internet sites with hard state

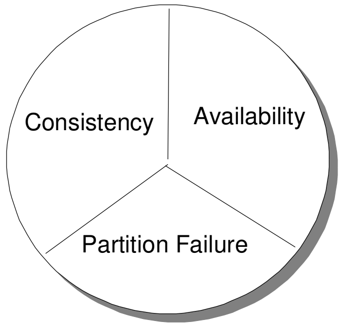
Inktomi

- CAP theorem
  - Big idea: DMBS semantics are too strong for this application
  - ➔critical performance loss due to semantics
- Inktomi on clusters
- Performance issues

CAP theorem

- Recall ACID
  - Atomic
  - Consistent
  - Integrity
  - Durability
- Recall BASE:
  - Basically Available, Soft state, Eventually Consistent

CAP



Pick any two

Examine P

Food for thought

- Partitioning is different from node failure

Node is dead is OK!

- Update one replica

When can you tell is something is dead vs. wires are down?

- Add more wires
- At some point, can't go on forever

Data Structures

- Two kind of tables
  - Table of all documents
  - Tables for each word
- Select Documents for each word
  - filtered by position, data, location
- Join with document table, present results
- Tables spread across whole cluster

Partitioning & Orchestration

- Partitioning
  - Tables spread across whole machine in disjoint manner
    - Lose a machine => lose data
    - easier to maintain
- Parallelized at "the bottom"
  - Initial part of the query scattered between nodes
  - That's where the hard work is, anyway

What happens on a failure?

"Bounce"

Like a reboot of whole system

Everything goes to a known state

Cleanup

- leaked sockets in TIME_WAIT
- memory leaks

10 second "hiccup"

- For a given app, what's acceptable response time? On average? Perturbations?

Porcupine & Inktomi

- Locality management
- CAP semantics

Strawman methodology

Identify semantic requirements on state

- What is ACID?
- What is BASE?

How much time is there to execute a

- query
- update
- How much state must be moved for each?

What are the interfaces to the above?

Locality Management

You have interface, + semantics.

How would you partition the DS for the application

- Between cluster nodes. Fed. Arch nodes?
- SMP processors + memory?

Build orchestration methods for the above DS for queries/updates/deletes.

- How long will these take?
- Synchronous or not?

Method (cont)

Construct model(s) of you application

- Use µ−benchmarks as building blocks
- Measure kernels on uniprocessors

Scaling under the model

- What is the appropriate scaling rule?
  - Time constrained? Resource constrained?
- How much data can it support?
- How many users?