CSE524 Parallel Computation

Lawrence Snyder
www.cs.washington.edu/CSEp524

29 May 2007

Announcements

- Out of email contact from tonight to project turn-in
- Project turn-in is by email to me (and Nathan) in **.pdf** or .doc.
  - Deadline is 4 June 2007 @ 5:00 PM PDT
  - Including your code as plain text in an appendix is OK, but I don't want your dev directories
  - I will be reading these personally, so it may take a while

Course Reviews Tonight

Agenda

- Platform independent communication
- GPGPU
- Cell
- FPGA
- Transactional Memory
- One Last Cool Idea: Fetch & Add
- Wrap-up … what did you learn?

3

Insulating from Comm Mechanism

- Writing scalable programs is possible, but avoid locking in on a particular comm mech
- What is a suitable abstraction for machine independent communication?
  - Assignment (Load/store) is the basic mechanism
  - Use ZPL's Ironman as an analogy
- Plan:
  - Use global view, package comm behind procs
  - Instantiate procs to machine-specific mechanism

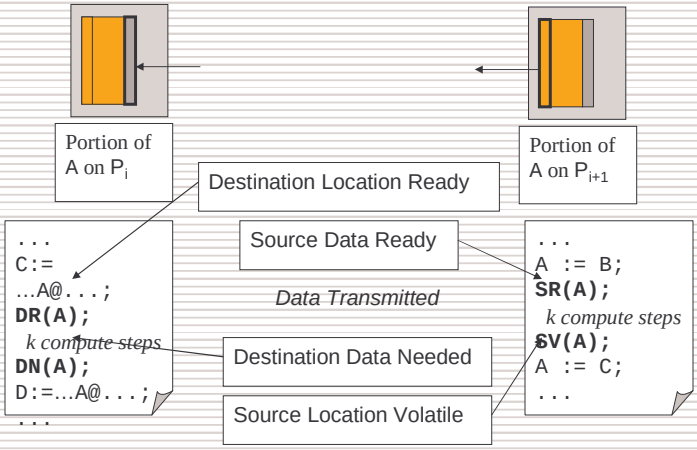
4

Ironman Basics

- Abstract comm as 4 functions protecting an assignment: D := S
  - Destination Ready DR() -- target location has been used for the last time before comm
  - Source Ready SR() -- source value to be X-mitted has been computed
  - Destination Needed DN() -- X-mitted value is now needed, and if not there, must stall
  - Source Volatile SV() -- source value location about to be reused

5

Ironman Basics



6

Binding Ironman Functions

- The machine's primitive communication facilities are bound to each function to implement the assignment

	Copying message passing (nCUBE, iPSC)	Asynchronous message passing MPI Asynch	Put-based 1-sided communication (Cray T3D, T3E)	Shared memory with coherency, SMPs
Destination Loc Ready		`mpi_irecv()`	`post_ready`	`post_ready`

Source Data Ready	`csend()`	`mpi_isend()`	`wait_ready`
			`shmem_put`
			`post_done`
				`wait_ready`
				`fluff:= A`
				`post_done`

Destination Data Needed	`crecf()`	`mpi_wait()`	`wait_done`	`wait_done`
Source Volatile		`mpi_wait()`		

7

Key Ideas of 'Personal Ironman'

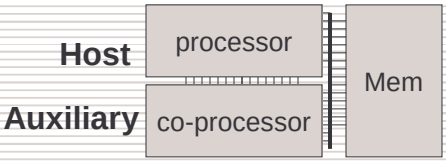
- The D := S assignment is a pair-wise communication between two processors
  - Functions protect it, so it "fires when ready"
  - Stalls are placed where needed, but are local
- Because communication is data-driven, the only delay in processing is based on either
  - The availability of the data
  - The availability of a place to put it

It's locally negotiated interaction

8

Other Parallel Architectures

- CMPs, SMPs, clusters, supercomputers do not exhaust parallel architectures
- Cell, General Purpose Graphics Processing Units (GPGPUs), processors with attached FPGAs are a popular parallel configuration



Note heterogeneous structure

9

Attached Processor Key Properties

- These engines benefit from Si advances more than standard processors
- They exploit …
  - Kernel processing … only speeding inner loop; leave all other processing/orchestrating to CPU
  - Data streaming from memory to get high throughput … moving lots of data fast permits cheap computations on each item
  - Specialized circuit designs at the expense of generality, making VLSI pay big

10

GPGPUs

- GPGPUs are hot! Without knowing what the table rows mean, we can see the advance eclipses standard processor improvements

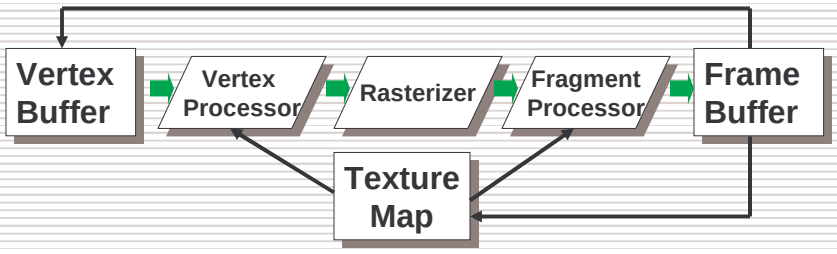
QuickTime™ and a
TIFF (LZW) decompressor
are needed to see this picture.

Thanks to David Blythe

11

Typical GPU Architecture

- GPGPUs have a standard pipelined graphics processing architecture



- Though GPGPUs have very general capabilities now, they retain this graphics terminology

Fragment processor is compute engine

12

Programming Model

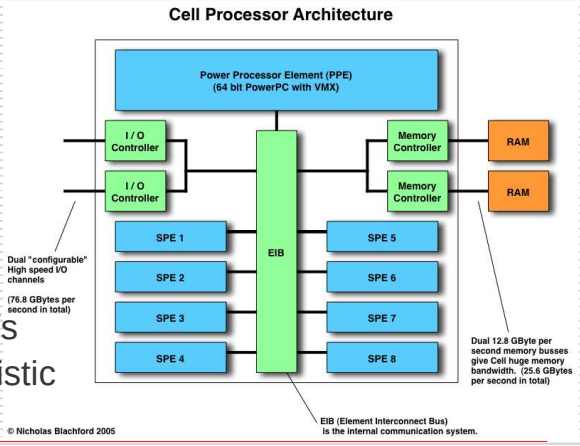
- GPGPUs are deeply pipelined
  - Computation must be element-wise over 2D data
  - Internal latencies favor interleaving operations
  - Huge data space amortizes pipe fill/drain
- Stream model most appropriate
- Programming tools retain strong graphics flavor -- need to translate to those terms

Accelerator is array language translating to GPGPUs

13

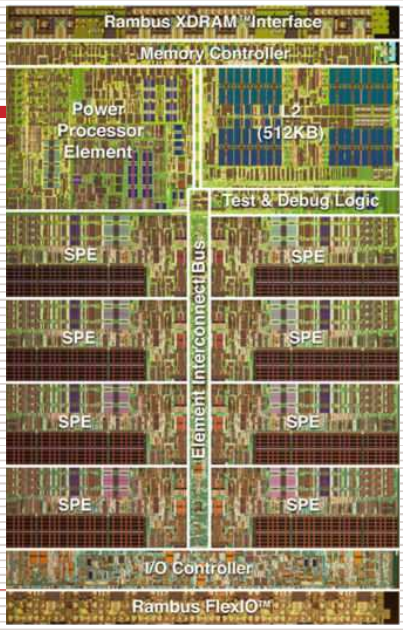
Cell

- Architecture
  - CPU
  - Synergistic Processing Elements
  - EIB rated at 307 GB/s but 204 GB/s is more realistic



14

Cell



Programming

- Cell is programmed with C/C++ under Linux using GCC
- Presently poorly supported by tools
- Primary goal is to decompose computation into streams and then orchestrate data flow into and out of the processor
- Everything must be “by hand” and profiling is fundamental to know what’s happening

The kind of computer an architect would think up!

16

Programming Matrix Multiplication

- Complexity seems dramatic

QuickTime™ and a
TIFF (LZW) decompressor
are needed to see this picture.

17

Attached FPGA

- Field Programmable Gate Array (Xilinx, Altera)
  - "Programmable hardware"
  - Ideal for ints, bit-twiddling, etc.
  - Very dramatic "regime change" between CPU and attached engine
  - FPGAs are supported "well" for circuit designers, but tools low level compared to IDE
  - Drop into "Opteron slot" for fast system build

Mostly for special purpose

18

Comparative Results

- All approaches improve on serial computers when the inner loop is compute intensive
- Matched Filter Computation

Implementation	Time per Sig	Speed-up
FPGA Cray XD-1	0.78 sec	3.91
GPU Nvidia 7900	1.00 sec	3.1
Cell	0.38 sec	8.0
CPU 3.2GHzXeon	3.05 sec	1.0

19

CSE524 Bottom Line

- Attached processors provide enormous power for low price, but they …
  - Use a different programming paradigm than our CTA-based view
  - Have no pretense of being general purpose
  - Should be programmed as though configuring hardware or building a hybrid machine
- Strengths are fast data streaming and leveraging VLSI
- Liabilities are programming challenges and rigidity

20

Transactional Memory-A Hot Idea

- You all know parallel programming is tough
- How to make it easier? Raise abstractions!
- Transactional Memory: Return of old idea
  - Databases concurrently manage external data consistently using multiple computers; well studied
  - Apply idea to concurrent management of the internal memory image
  - Transaction: Atomically change memory to new state or do nothing at all
  - Say the goal not how to achieve it

Idea: David Lomet in 1977

21

Atomic

An *easier-to-use* and *harder-to-implement* primitive

```
void deposit(int x){
synchronized(this){
  int tmp = balance;
  tmp += x;
  balance = tmp;
}}
```

```
void deposit(int x){
atomic {
  int tmp = balance;
  tmp += x;
  balance = tmp;
}}
```

lock acquire/release

(behave as if)
no interleaved computation;
no unfair starvation

22

Transactions

- The code executing atomically is everything (dynamically) between braces, including **`foo().`**

```
atomic {
  if (x != null) x.foo();
      y = true;
}
```

- Three choices: commit; abort; not terminate
- Optimistic: Little overhead if no conflict
- Avoid races and deadlocks due to lock acquisition

23

Transactions Advantages

- In DB transactions have ACID properties
  - A = atomicity … sequence of operations never interrupted or incomplete; commit or abort
  - C= consistency …changes leave memory in consistent state relative to application; for example new_balance==old_balance+deposit
  - I = isolation … transaction works correctly with any combination of other transactions
  - D = durability … result persists; not appropriate for multithreading memory case

24

DB & TM Transactions Are Different

- DBs use disks, meaning the SW support for DB transactions not time-critical; referencing memory is too brief (and frequent) to allow for heavy-weight protection
- TM need not be durable (last) since data doesn't outlast execution; simplifying
- TM must retrofit into a rich world of legacy code … must coexist with all other mechanisms; pervasive changes not feasible

25

Atomic Doesn't Solve Everything

- It's not difficult to mess up using `atomic`

Thread 1	Thread 2
`atomic {`	`atomic {`
`while (!flagA)`	`  flagA = true;`
`  flagB = true;`	`  while (!flagB);`
`}`	`}`

  flags have to be true at the start of block
- This code not serializable, i.e. there is no correct serial execution

26

TM Promising, But NRFPT

- Transactions are no panacea
  - Neither hardware nor software implementations have proved themselves
- Very Nice Monograph: Larus & Rajwar

http://www.morganclaypool.com/doi/abs/10.2200/S00070ED1V01Y200611CAC002

- A fundamental problem TM will not solve: How to scale shared memory computations to architectures with much larger λ, which are inevitable

27

An Alternate Concurrency Primitive

- Rather than using the Test&Set to guard shared data, use Fetch&Add
  - Fetch&Add is an atomic read-modify-write operation on memory -- requires special hardware, to be discussed
  - Use Fetch&Add as a semaphore and as a scheduler
- Operation: Fetch&Add(V,e)
  - V is a memory location
  - e is an integer expression
  - Contents of V are returned
  - New value of V is V+e
  - Operation is atomic

V: 0

Fetch&Add(V,1)

V: 1

0 is returned

28

Concurrent Fetch&Adds

- When multiple Fetch&Adds are executed simultaneously, they are serializable
- Assume Fetch&Add(V, e1), Fetch&Add(V, e2) execute simultaneously
  - Assuming an initial value of e0
  - Final value is e0+e1+e2
  - The 1st process receives either e0 or e0+e2, implying it was first (e0) or second (e0+e2)
  - The 2nd process receives either e0 or e0+e1, implying it was second (e0+e1) or first (e0)

Suppose both execute Fetch&Add(I,1), then one gets I back, the other I+1, and final is I+2

29

Break

30

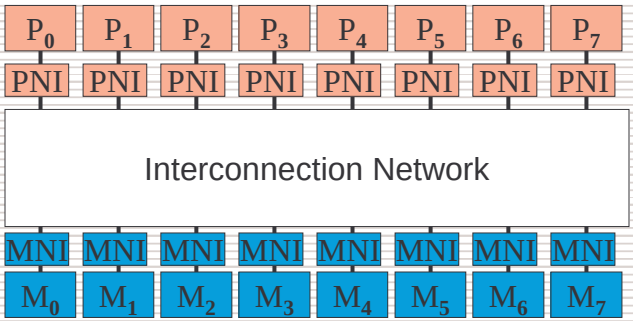
Fetch&Add Exploits Sharing

- Though earlier solutions attempt to reduce sharing to reduce the amount of invalidation and acknowledgment, Fetch&Add does better with greater sharing
- Sharing is used to schedule or allocate, which is then independent activity
  - Sharing is concentrated in a few variables
  - Fine grain size is possible
- Since load/store, Test&Set, etc. are implementable, it is a “sufficient” primitive

31

Implementing Fetch&Add

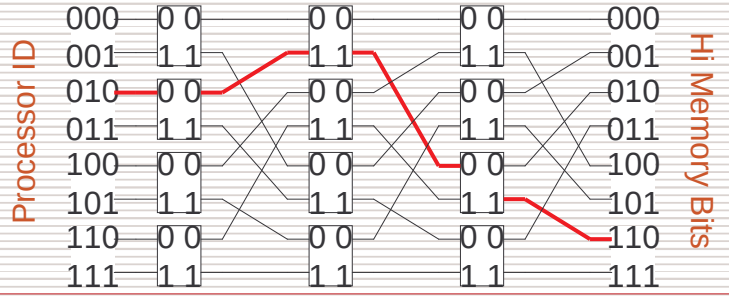
- Fetch&Add assumes a flat shared memory as implemented by a “dance hall architecture”



32

Omega Network

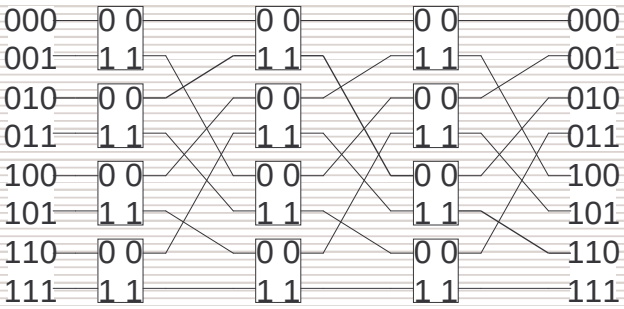
- The interconnection network is an Ω-network
  - Connection between 2 and 6 … follow bits to destination lsb to msb



33

Notice Details

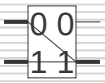
- The Ω-Network requires O(P log P) routers
- The given network uses 2x2 but $2^b x 2^b$ work
- Wiring is consistent at each stage



Long Wires Are Necessary

Routing In Ω-Network

- The network is pipelined
- There is a unique path between any processor and memory port pair
- Conflicts are possible because there exist permutations in which packets collide
- What happens when two packets collide at a router?
  - Packet is delayed, leaving its "file"
  - Pipelining is affected, here comes more

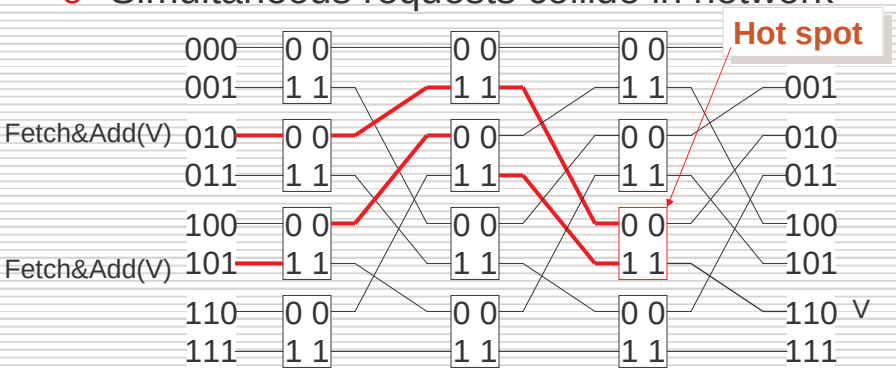


The separate packets must be serialized

35

Two Processors Do Fetch&Add(V,1)

- Simultaneous requests collide in network

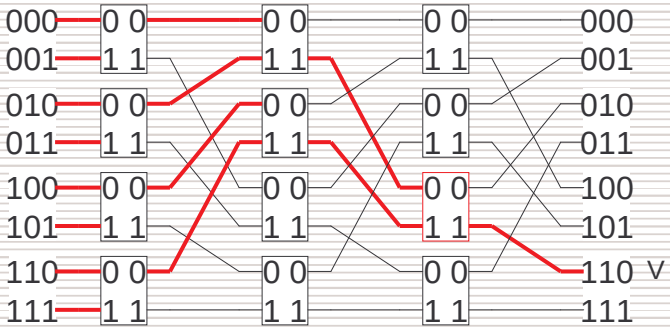


Fetch&Add increases potential for collisions

36

The Bright Idea: Combine Requests

Idea: Combine requests for same dest. In limit all nodes could reference same loc.



37

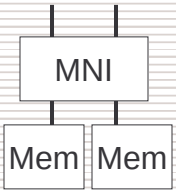
Loads & Stores

- At a switch combine loads and stores to a common location as follows
  - Load/Load -- forward one of the loads towards the memory, and when the value is returned, satisfy both
  - Load/Store -- forward the store, and when the ACK arrives back at the switch, return value to satisfy load
  - Store/Store -- forward one of the stores, and when the ACK arrives back at the switch, return it for both
- Processors are restricted to having only one outstanding request at a time to a given location

38

Implementing Fetch&Add

- Include an adder with the Memory Network Interface chips
- For Fetch&Add(V,e)
  - Fetch the value of V, say e0
  - Return e0 to processor requesting
  - Add e0+e
  - Store e0+e back into V
- It is probably necessary to do these concurrently

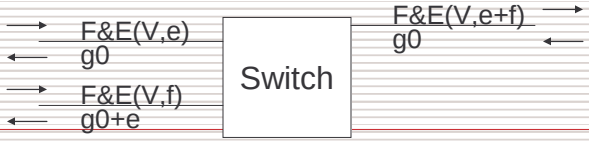


39

Combining Fetch&Adds at Switch

Suppose Fetch&Add(V,e), Fetch&Add(V,f) arrive at a switch together …

- Form the sum f+e
- Send Fetch&Add(V,f+e) on to the memory
- Store e locally
- When g0 is returned by the memory
  - Return g0 as response to Fetch&Add(V,e)
  - Return g0+e as response to Fetch&Add(V,f)



40

Combine F&A w/ other requests

- Combining can apply to all memory traffic to a location V
- Consider the following cases
  - Fetch&Add/Fetch&Add -- as just described
  - Fetch&Add/Load -- Treat Load as Fetch&Add(V,0)
  - Fetch&Add/Store -- If Fetch&Add(V,e) meets Store(V,f) send Store(V,e+f) to memory; when ACK is received, return f as value of F&A
- Conclusion -- it is possible to combine all requests to the same memory location

41

Just Thinking About, Will It Work?

- Potential Problems …
  - Network routing is driven entirely by performance, so a complicated switch is usually a problem
  - Routers typically forward non-blocked packets in <= 3 tix
  - Matching to recognize that two requests collide is an "add" operation
  - Combining is an "add" operation *after* the previous add
  - Combining relies on the requests getting to the switch simultaneously, or at worst, before the forwarded packet leaves … this is improbable
  - Most traffic is non-combinable -- head for different places
  - A combining router was created by Susan Dickey

42

A Backup Strategy

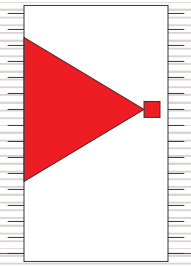
- If the network switch is too slow then …
  - Do not combine at every stage … so that some stages can be fast
  - Use two networks, one fast and one that does combining -- it can handle the sharing requests
  - Combine only like requests, e.g. loads/loads
  - Limit combining at a node to two requests

  - As it happened
    - Only like requests have ever been implemented in switch
    - IBM used the two network solution in the RP3

43

More Globally

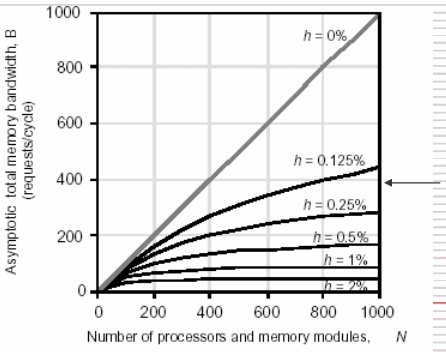
- Norton and Pfister discovered in simulation for the RP3 computer that the Ω-Network develops hot-spots
- It was thought that combining would remove the hot-spots … it seemed to for 64-way network
- The problem is that once a node becomes hot, a “back-up” tree forms “behind” the node



44

More Globally

- Lee, Kruskal, Kuck studied by simulation, analysis
  - LKK discovered and named the “back-up tree”
  - Showed in simulation that the 64-way network is lucky
  - Combining doesn’t help because it’s the other traffic



0.125% traffic directed at a hot spot

45

Assessment

- It was a good idea but it didn’t work
  - Good
    - Fetch&Add is clever -- a primitive with good properties
    - Shifting from protecting data to allocating work is better
    - Computation at memory is powerful, worth doing
  - Bad
    - Pipelined multistage networks probably just don’t work
    - Complexity in a switch is wrong -- speed is essential
    - Failed to exploit locality -- caching basically impossible

46

Wrap-Up on Parallel Computing

- What in your view was the high-order bit?

47

My List …

- Using parallel computers is tough
- Parallel computers generally behave like the CTA, so program to it … it won't disappoint
- Parallel algorithms often require fresh thinking -- sequential case may not be a good place to begin
- CMPs are sweet-spot now, but for how long?
- Reduce/Scan are basic building blocks

48

My List (continued) …

- Programming tools are all over the place
  - OpenMP is **very** simple, but not too expressive
  - Pthreads, a standard library, but very low level
  - MPI is universal, low level and abstraction-free
  - ZPL leaves all parallelism to compiler, but gives WYSIWYG as guide to writing good programs
  - Co-Array Fortran, UPC, Ti, Transactional Memory have promise perhaps, but NRFPT
- Hardware design is very volatile at moment

Log Out

- Submit project by 5:00 PM (PDT) 6/4
- Be sure to fill out a course evaluation