inst.eecs.berkeley.edu/~cs61c

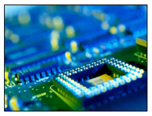
CS61C : Machine Structures

Lecture #30 Parallel Computing

2007-8-15

Scott Beamer, Instructor

Ion Wind Cooling Developed by Researchers at Purdue



www.bbc.co.uk

Review of Software Parallelism

- Parallelism is necessary
  - It looks like the future of computing…
  - It is unlikely that serial computing will ever catch up with parallel computing
- Software parallelism
  - Grids and clusters, networked computers
  - Two common ways to program:
    - Message Passing Interface (lower level)
    - MapReduce (higher level, more constrained)
- Parallelism is often difficult
  - Speedup is limited by serial portion of code and communication overhead

A New Hope: Google's MapReduce

- Remember CS61A?

```
(reduce + (map square '(1 2 3)) ⇒
(reduce + '(1 4 9)) ⇒
14
```

- We told you "the beauty of pure functional programming is that it's easily parallelizable"
  - Do you see how you could parallelize this?
  - What if the `reduce` function argument were associative, would that help?
- Imagine 10,000 machines ready to help you compute anything you could cast as a MapReduce problem!
  - This is the abstraction Google is famous for authoring (but their reduce not the same as the CS61A's or MPI's reduce)
    - Builds a reverse-lookup table
  - It hides LOTS of difficulty of writing parallel code!
  - The system takes care of load balancing, dead machines, etc.

MapReduce Programming Model

Input & Output: each a set of key/value pairs

Programmer specifies two functions:

```
map (in_key, in_value) →
    list(out_key, intermediate_value)
```

- Processes input key/value pair
- Produces set of intermediate pairs

```
reduce (out_key, list(intermediate_value)) →
    list(out_value)
```

- Combines all intermediate values for a particular key
- Produces a set of merged output values (usu just one)

code.google.com/edu/parallel/mapreduce-tutorial.html

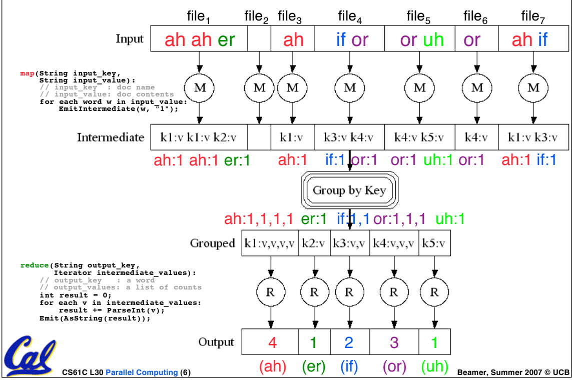
MapReduce Code Example

```
map(String input_key,
    String input_value):
    // input_key  : document name
    // input_value: document contents
    for each word w in input_value:
        EmitIntermediate(w, "1");

reduce(String output_key,
       Iterator intermediate_values):
    // output_key    : a word
    // output_values: a list of counts
    int result = 0;
    for each v in intermediate_values:
        result += ParseInt(v);
    Emit(AsString(result));
```

- "Mapper" nodes are responsible for the `map` function
- "Reducer" nodes are responsible for the `reduce` function
- Data on a distributed file system (DFS)

MapReduce Example Diagram


file1 file2 file3 file4 file5 file6 file7
Input
ah ah er | | ah | if or | or uh | or | ah if
M M M M M M M
Intermediate
k1:v k1:v k2:v | | k1:v | k3:v k4:v | k4:v k5:v | k4:v | k1:v k3:v
ah:1 ah:1 er:1 ah:1 if:1 or:1 or:1 uh:1 or:1 ah:1 if:1
Group by Key
ah:1,1,1,1 er:1 if:1,1 or:1,1,1 uh:1
Grouped
k1:v,v,v,v | k2:v | k3:v,v | k4:v,v,v | k5:v
R R R R R
Output
4 | 1 | 2 | 3 | 1
(ah) (er) (if) (or) (uh)

```
map(String input_key,
    String input_value):
    // input_key  : doc name
    // input_value: doc contents
    for each word w in input_value:
        EmitIntermediate(w, "1");
```

```
reduce(String output_key,
       Iterator intermediate_values):
    // output_key    : a word
    // output_values: a list of counts
    int result = 0;
    for each v in intermediate_values:
        result += ParseInt(v);
    Emit(AsString(result));
```

MapReduce Advantages/Disadvantages

- Now it's easy to program for many CPUs
  - Communication management effectively gone
    - I/O scheduling done for us
  - Fault tolerance, monitoring
    - machine failures, suddenly-slow machines, other issues are handled
  - Can be much easier to design and program!
- But… it further restricts solvable problems
  - Might be hard to express some problems in a MapReduce framework
  - Data parallelism is key
    - Need to be able to break up a problem by data chunks
  - MapReduce is closed-source – Hadoop!

Introduction to Hardware Parallelism

- Given many threads (somehow generated by software), how do we implement this in hardware?
- Recall the performance equation:

Execution Time = (Inst. Count)(CPI)(Cycle Time)

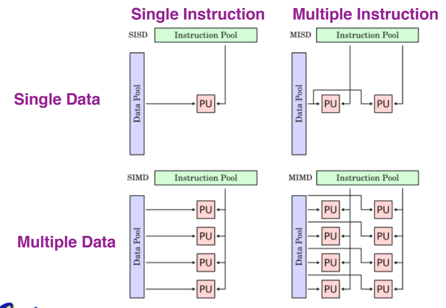
- Hardware Parallelism improves:
  - Instruction Count - If the equation is applied to each CPU, each CPU needs to do less
  - CPI - If the equation is applied to system as a whole, more is done per cycle
  - Cycle Time - Will probably be made worse in process

Disclaimers

- Please don't let today's material confuse what you have already learned about CPU's and pipelining
- When *programmer* is mentioned today, it means whoever is generating the assembly code (so it is probably a compiler)
- Many of the concepts described today are *difficult* to implement, so if it sounds easy, think of possible hazards

Flynn's Taxonomy

- Classifications of parallelism types

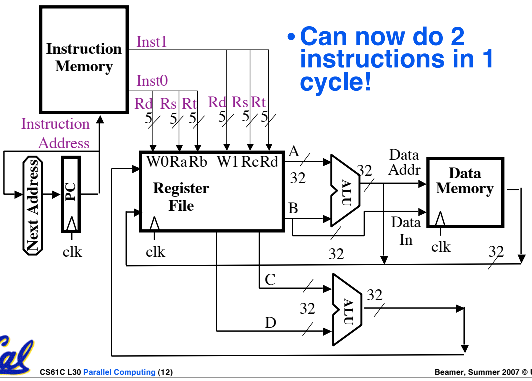


www.wikipedia.org

Superscalar

- Add more functional units or pipelines to CPU
- Directly reduces CPI by doing more per cycle
- Consider what if we:
  - Added another ALU
  - Added 2 more read ports to the RegFile
  - Added 1 more write port to the RegFile

Simple Superscalar MIPS CPU



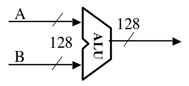
- Can now do 2 instructions in 1 cycle!

Simple Superscalar MIPS CPU (cont.)

- Considerations
  - ISA now has to be changed
  - Forwarding for pipelining now *harder*
- Limitations
  - Programmer must explicitly generate parallel code
  - Improvement only if other instructions can fill slots
  - Doesn't scale well

Single Instruction Multiple Data (SIMD)

- Often done in a *vector* form, so all data has the same operation applied to it
- Example: AltiVec (like SSE)
  - 128bit registers can hold:
    - 4 floats, 4 ints, 8 shorts, 16 chars, etc.
  - Processes whole vector



Superscalar in Practice

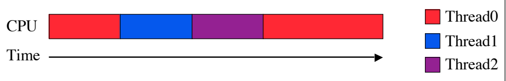
- ISA's have extensions for these *vector* operations
- One thread, that has parallelism internally
- Performance improvement depends on program and programmer being able to fully utilize all slots
- Can be parts other than ALU (like load)
- Usefulness will be more apparent when combined with other parallel techniques

Administrivia

- Put in regrade requests now for any assignment past HW2
- Final: Thursday 7-10pm @ 10 Evans
  - NO backpacks, cells, calculators, pagers, PDAs
  - 2 writing implements (we'll provide write-in exam booklets) – pencils ok!
  - Two pages of notes (both sides) 8.5"x11" paper
  - One green sheet
- Course Survey last lecture, 2pts for doing it

Thread Review

- A *Thread* is a single stream of instructions
  - It has its own registers, PC, etc.
  - Threads from the same process operate in the same virtual address space
  - Are an easy way to describe/think about parallelism
- A single CPU can execute many threads by *Time Division Multipexing*

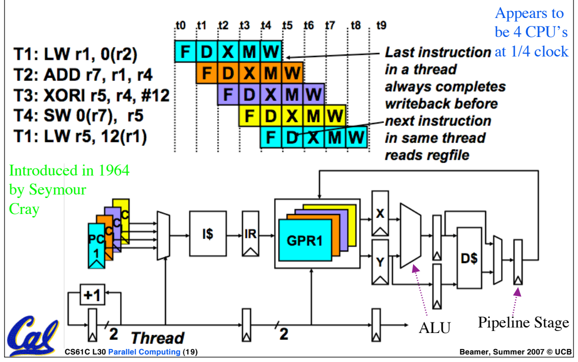


Multithreading

- Multithreading is running multiple threads through the same hardware
- Could we do *Time Division Multipexing* better in hardware?
- Consider if we gave the OS the abstraction of having 4 physical CPU's that share memory and each executes one thread, but we did it all on 1 physical CPU?

Static Multithreading Example

Interleave 4 threads, T1-T4, on non-bypassed 5-stage pipe


Appears to be 4 CPU's at 1/4 clock
t0 t1 t2 t3 t4 t5 t6 t7 t8 t9
T1: LW r1, 0(r2)
T2: ADD r7, r1, r4
T3: XORI r5, r4, #12
T4: SW 0(r7), r5
T1: LW r5, 12(r1)
F D X M W
Last instruction in a thread always completes writeback before next instruction in same thread reads regfile
Introduced in 1964 by Seymour Cray
PC 1
I$
IR
GPR1
X
Y
D$
+1
2
Thread
ALU
Pipeline Stage

Static Multithreading Example Analyzed

- Results:
  - 4 Threads running in hardware
  - Pipeline hazards reduced
    - No more need to forward
    - No control issues
    - Less structural hazards
  - Depends on being able to fully generate 4 threads evenly
    - Example if 1 Thread does 75% of the work
      - Utilization = (% time run)(% work done)
        = (.25)(.75) + (.75)(.25) = .375
        = 37.5%

Dynamic Multithreading

- Adds flexibility in choosing time to switch thread
- Simultaneous Multithreading (SMT)
  - Called Hyperthreading by Intel
  - Run multiple threads at the same time
  - Just allocate functional units when available
  - Superscalar helps with this

Dynamic Multithreading Example

One thread, 8 units

Cycle	M	M	FX	FX	FP	FP	BR	CC
1	🟨							🟨
2	🟨	🟨					🟨	
3				🟨	🟨			
4								
5								
6								
7	🟨			🟨		🟨		
8		🟨			🟨			
9				🟨				

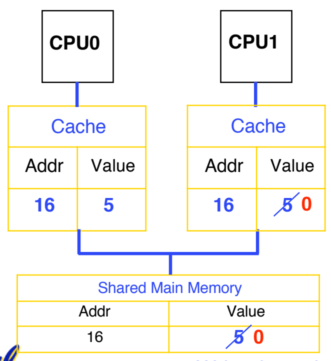
Two threads, 8 units

Cycle	M	M	FX	FX	FP	FP	BR	CC
1	🟨	🟦	🟦					🟨
2	🟨	🟨	🟦			🟦	🟨	
3	🟦			🟨	🟨			
4	🟦	🟦				🟦		
5		🟦						🟦
6								
7	🟨		🟦	🟨	🟦	🟨		
8		🟨		🟦	🟨	🟦		
9	🟦	🟦		🟨		🟦		

Multicore

- Put multiple CPU's on the same die
- Why is this better than multiple dies?
  - Smaller
  - Cheaper
  - Closer, so lower inter-processor latency
  - Can share a L2 Cache (details)
  - Less power (power ~ freq^2)
- Cost of multicore: complexity and slower single-thread execution

Two CPUs, two caches, shared DRAM ...



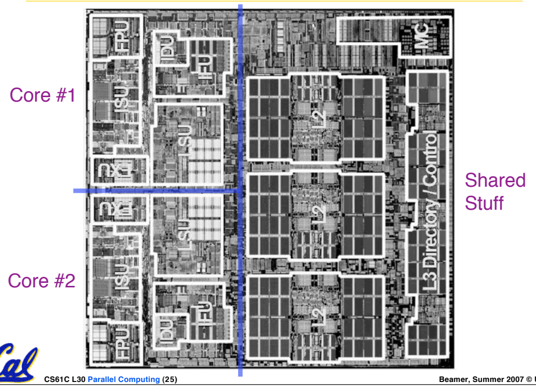
CPU0:
```
LW R2, 16(R0)
```
CPU1:
```
LW R2, 16(R0)
```
CPU1:
```
SW R0,16(R0)
```

View of memory no longer "coherent".

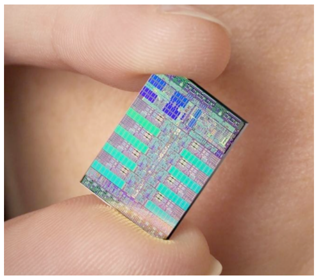
Loads of location 16 from CPU0 and CPU1 see different values!

Write-through caches

Multicore Example (IBM Power5)



Real World Example 1: Cell Processor



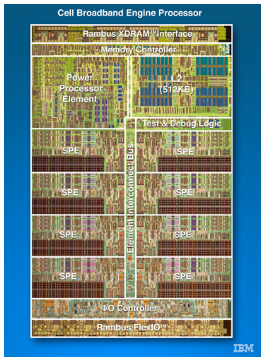
- Multicore, and more….

Real World Example 1: Cell Processor

- 9 Cores (1PPE, 8SPE) at 3.2GHz
- Power Processing Element (PPE)
  - Supervises all activities, allocates work
  - Is multithreaded (2 threads)
- Synergystic Processing Element (SPE)
  - Where work gets done
  - Very Superscalar
  - No Cache, only Local Store

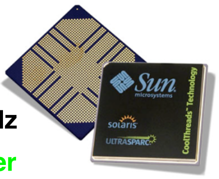
Real World Example 1: Cell Processor

- Great for other multimedia applications such as HDTV, cameras, etc…
- Really dependent on programmer use of SPE's and Local Store to get the most out of it



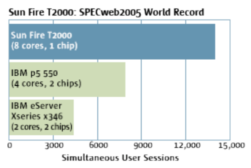
Real World Example 2: Niagara Processor

- Multithreaded and Multicore
- 32 Threads (8 cores, 4 threads each) at 1.2GHz
- Designed for low power
- Has simpler pipelines to fit more on
- Maximizes thread level parallelism
- Project Blackbox



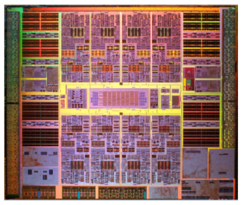
Real World Example 2: Niagara Processor

- Each thread runs slower (1.2GHz), and there is less number crunching ability (no FP unit), but tons of threads
- This is great for webservers, where there are typically many simple requests, and many data stalls
- Can beat "faster" and more expensive CPU's, while using less power



Rock: Niagara's Successor

- Released last week
- 64 Threads (8 cores, 8 threads each)
- 8 FPU's, 8 Crypto Co-processors
- Integrated 10GbE and PCIe hardware
- Supports 64 Logical Domains (for 64 virtual OS's)
- Only 20 months later



Peer Instruction

1. The majority of PS3's processing power comes from the Cell processor
2. A computer that has max utilization can get more done multithreaded
3. Current multicore techniques can scale well to many (32+) cores

	ABC
1:	FFF
2:	FFT
3:	FTF
4:	FTT
5:	TFF
6:	TFT
7:	TTF
8:	TTT

Summary

- Superscalar: More functional units
- Multithread: Multiple threads executing on same CPU
- Multicore: Multiple CPU's on the same die
- The gains from all these parallel hardware techniques relies heavily on the programmer being able to map their task well to multiple threads
- Hit up CS150, CS152, CS162, 194-3, 198-5 and wikipedia for more info