CSE524 Parallel Computation

Lawrence Snyder
www.cs.washington.edu/CSEp524

10 April 2007

Announcements

- Homework submission issues
- Final project due Monday, 4 June 2007
- Next HW assigned next week

We will discuss homework shortly

Unanswered Question from Last Time

- Question on topic of “no standard parallel model”: Sequential computers were quite different originally, before a machine (IBM 701) gained widespread use. Won’t the widespread use of Intel (or AMD) CMPs have that same affect?

3

Review

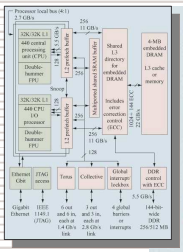
- High-level logic of last week’s lecture:
  - Parallel architectures are diverse (looked at 5)
  - Key difference: Memory structure
  - In sequential programming, we use simple RAM model
  - In parallel programming, PRAM misdirects us
  - CTA abstracts machines; captures …
    - Parallelism of multiple (full service) processors
    - Local vs nonlocal memory reference costs
    - Vague memory structure details; no shared memory
  - Different mechanisms impl. nonlocal memory reference
    - Shared, message passing, one-sided

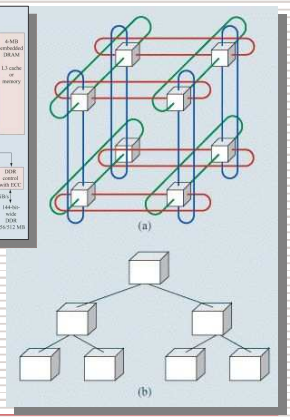
4

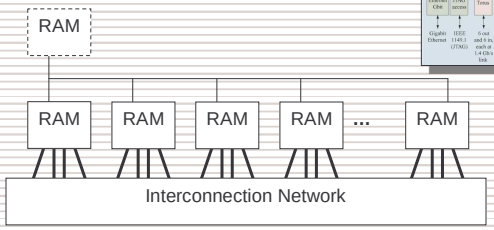
CTA Abstracts BlueGene/L

- Consider BlueGene/L as a CTA machine

$\lambda > 5000$





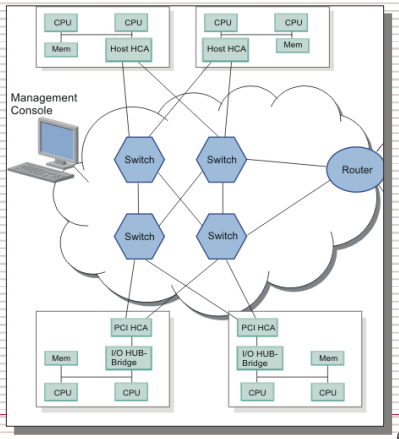


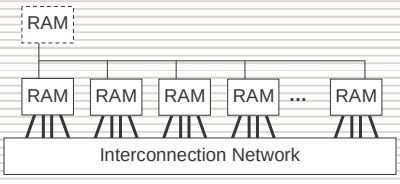
5

CTA Abstracts Clusters

- Consider a cluster as a CTA

$\lambda > 4000$



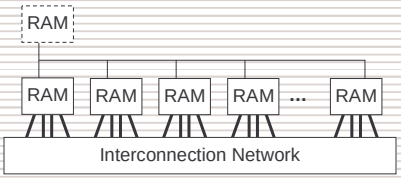


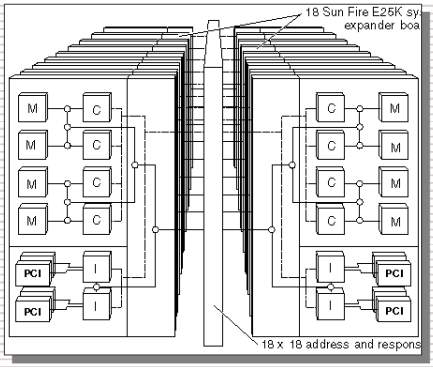
6

CTA Abstracts X-bar SMPs

- Consider the SunFire E25K as a CTA

λ ~ 600



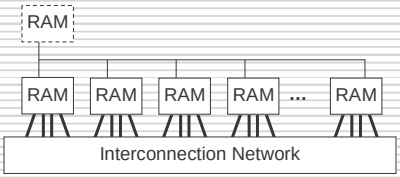


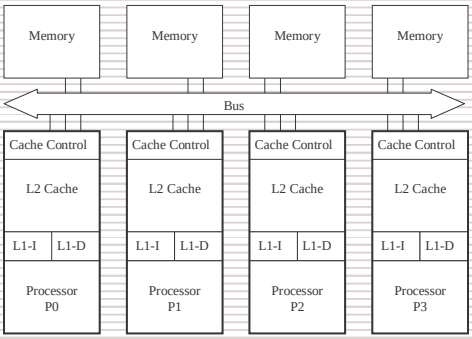
7

CTA Abstracts Bus SMPs

- Consider Bus-based SMPs as CTAs

λ ~ 100s

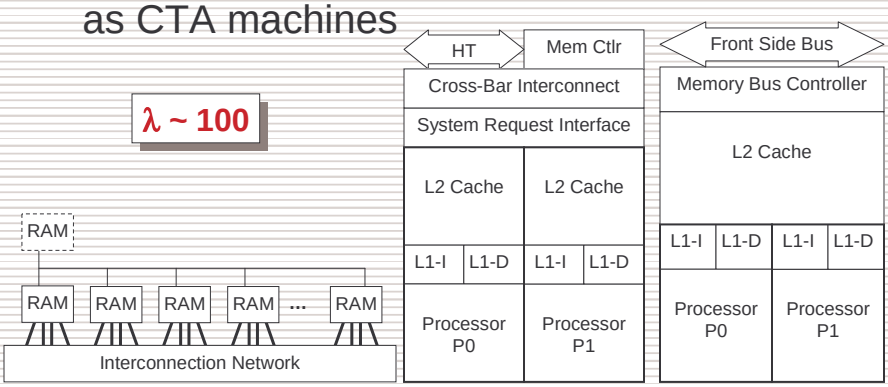




8

CTA Abstracts CMPs

- Consider Core Duo & Dual Core Opteron as CTA machines



9

CTA Abstracts Machines: Summary

- Naturally, the "match" between the CTA & a given ||-architecture differs from all others
- Two main differences--
  - Controller--not particularly essential--can be efficiently emulated
  - Nonlocal reference time--is smaller for small machines, larger for large machines, implying λ increases as *P* increases … need it for scaling

Though λ is "too large" for small machines, the "error" forces programs towards more efficient solutions: more locality!

10

Shared Memory and the CTA

- The CTA has no shared memory - meaning no guarantee of hardware implementing shared memory => cannot depend on it
  - Some machines have shared memory, which is effectively their communication facility
  - Some machines have no shared memory, meaning there's another form of communication
- Either way, assume it is expensive relative to local computation to communicate

11

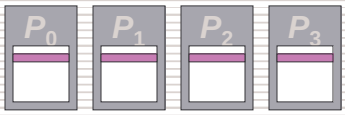
Assignment from last week

- Homework Problem: Analyze the complexity of the Odd/Even Interchange Sort: Given array A[n], exchange o/e pairs if not ordered, then exchange e/o pairs if not ordered, then repeat until sorted
- Analyze in CTA model (i.e. for P, λ, d), and charge the o/e-e/o pair c time if operands are local; ignore all other local computation

12

O/E - E/O Sort

- The array is assigned to memories



One Step:
- get end neighbor values: λ
- O/E half step: (n/P)c
- get end neighbor values: λ
- E/O half step: (n/P)c
- And-reduce over done_?:λ log P

No. Steps: n/2 in worst case

13

Parallelism vs Performance

- Naïvely, many reason that applying *P* processors to a *T* time computation will result in *T/P* time performance
- Wrong!

The Intuition: The serial and parallel solutions differ

- **More or fewer instructions must be executed**
- **The hardware is different**
- **Parallel solution has difficult-to-quantify costs that the serial solution does not have, etc.**

Consider Each Reason

14

More Instructions Needed

- To implement parallel computations requires overhead that sequential computations do not need
  - All costs associated with communication are overhead: locks, cache flushes, coherency, message passing protocols, etc.
  - All costs associated with thread/process setup
  - Lost optimizations -- many compiler optimizations not available in parallel setting
    - Global variable register assignment

15

More Instructions (Continued)

- Redundant execution can avoid communication -- a parallel optimization

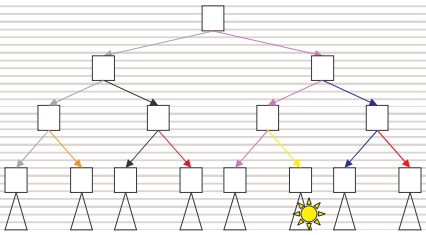
New random number needed for loop iteration:
(a) Generate one copy, have all threads ref it … requires communication
(b) Communicate seed once, then each thread generates its own random number … removes communication and gets parallelism, but by increasing instruction load

A common (and recommended) programming trick

16

Fewer Instructions

- Searches illustrate the possibility of parallelism requiring fewer instructions



- Independently searching subtrees means an item is likely to be found faster than sequential

17

Threads

- A thread consists of program code, a program counter, call stack, and a small amount of thread-specific data
  - Threads share access to memory (and the file system) with other threads
  - Threads communicate through the shared memory
  - The native memory model of computers does not automatically accommodate safe concurrent memory reference

Shared memory parallel programming

18

Processes

- A process is a thread in its own private address space
  - Processes do not communicate through shared memory, but need another mechanism like message passing
  - Key issue: How is the problem divided among the processes, which includes data and work
  - Processes (logically subsume) threads

Message-passing parallel programming

19

Compare Threads & Processes

- Both have code, PC, call stack, local data
  - Threads -- One address space
  - Processes -- Separate address spaces
- Weight and Agility
  - Threads: lighter weight, faster to setup, tear down, perform communication
  - Processes: heavier weight, setup and tear down more time consuming, communication is slower

20

Terminology

- Terms used to refer to a unit of parallel computation include: thread, process, processor, …
  - Technically, thread and process are SW, processor is HW
  - Usually, it doesn't matter

Most frequently the term *processor* is used

21

Parallelism vs Performance

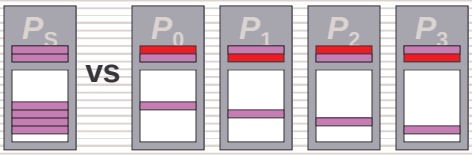
- Sequential hardware ≠ parallel hardware
  - There is more parallel hardware, e.g. memory
  - There is more cache on parallel machines
  - Sequential computer $\stackrel{?}{\neq}$ 1 processor of || computer, because of cache coherence hw
    - Important in multicore context
  - Parallel channels to disk, possibly

These differences *tend* to favor || machine

22

Superlinear Speed up

- Additional cache is an advantage of ||ism

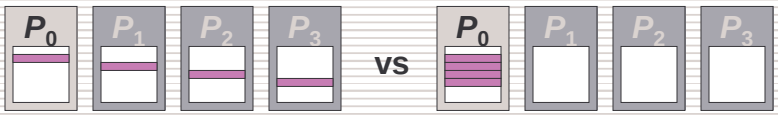


- The effect is to make execution time < *T*/*P* because data (& program) reference faster
- Cache-effects help mitigate other || costs

23

"Cooking" The Speedup Numbers

- The sequential computation should not be charged for **any** || costs … consider



- If referencing memory in other processors takes time (λ) and data is distributed, then one processor solving the problem results in greater *t* compared to true sequential

This complicates methodology for large problems

24

Other Parallel Costs

- Wait: All computations must wait at points, but serial computation waits are well known
- Parallel waiting …
  - For serialization to assure correctness
  - Congestion in communication facilities
    - Bus contention; network congestion; etc.
  - Stalls: data not available/recipient busy
- These costs are generally time-dependent, implying that they are highly variable

25

Bottom Line …

- Applying *P* processors to a problem with a time *T* (serial) solution can be either …

  better or worse …

  it's up to programmers to exploit the advantages and avoid the disadvantages

26

Break

Two kinds of performance

- Latency -- time required before the result available
  - Latency, measured in seconds; called *transmit time* or *execution time* or just *time*
- Throughput -- amount of work completed in a given amount of time
  - Throughput, measured in “work”/sec, where “work” can be bits, instructions, jobs, etc.; also called *bandwidth* in communication

Both terms apply to computing and communications

Latency

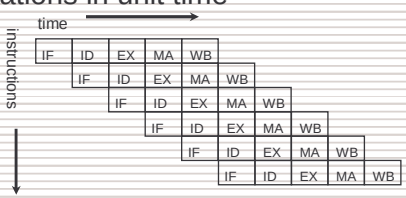
- Reducing latency (execution time) is a principal goal of parallelism
- There is upper limit on reducing latency
  - Speed of light, esp. for bit transmissions
  - (Clock rate) x (issue width), for instructions
  - Diminishing returns (overhead) for problem instances

Hitting the upper limit is rarely a worry

29

Throughput

- Throughput improvements are often easier to achieve by adding hardware
  - More wires improve bits/second
  - Use processors to run separate jobs
  - Pipelining is a powerful technique to execute more (serial) operations in unit time



Better throughput often hyped as better latency

30

Digress: Inherently Sequential

- As an artifact of *P*-completeness theory, we have the idea of *Inherently Sequential* -- computations not appreciably improved by parallelism

Circuit Value Problem: Given a circuit α over Boolean input values values b_1, …, b_n and designated output value y, is the circuit true for y?

- Probably not much of a limitation

31

Latency Hiding

- Reduce wait times by switching to work on different operation
  - Old idea, dating back to Multics
  - In parallel computing it's called *latency hiding*
- Idea most often used to lower λ costs
  - Have many threads ready to go …
  - Execute a thread until it makes nonlocal ref
  - Switch to next thread
  - When nonlocal ref is filled, add to ready list

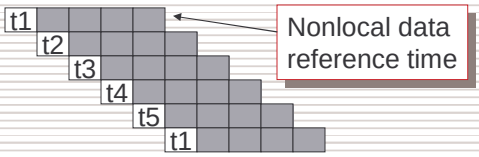
Tera MTA did this at instruction level

32

Latency Hiding (Continued)

- Latency hiding requires …
  - Consistently large supply of threads ~ λ/e

    where e = average # cycles between nonlocal refs
  - Enough network throughput to have many requests in the air at once



- Latency hiding has been claimed to make shared memory feasible with large λ

There are difficulties

33

Latency Hiding (Continued)

- Challenges to supporting shared memory
  - Threads must be numerous, and the shorter the interval between nonlocal refs, the more
    - Running out of threads stalls the processor
  - Context switching to next thread has overhead
    - Many hardware contexts -- or --
    - Waste time storing and reloading context
  - Tension between latency hiding & caching
    - Shared data must still be protected somehow
  - Other technical issues

34

Amdahl's Law

- If 1/*S* of a computation is inherently sequential, then the maximum performance improvement is limited to a factor of *S*

$$T_P = 1/S \times T_S + (1\text{-}1/S) \times T_S / P$$

T_S=sequential time
T_P=parallel time
P =no. processors

- Amdahl's Law, like the Law of Supply and Demand, is a fact

Gene Amdahl -- IBM Mainframe Architect

35

Interpreting Amdahl's Law

- Consider the equation

$$T_P = 1/S \times T_S + (1\text{-}1/S) \times T_S / P$$

- With no charge for || costs, let $P \rightarrow \infty$ then $T_P \rightarrow 1/S \times T_S$

The best parallelism can do to is to eliminate the parallelizable work; the sequential remains

- Amdahl's Law applies to problem *instances*

Parallelism seemingly has little potential

36

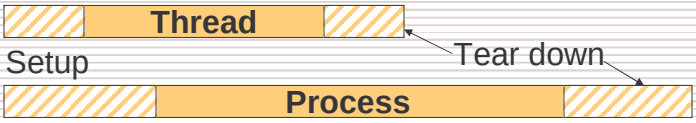
More On Amdahl's Law

- Amdahl's Law assumes a fixed problem instance: Fixed *n*, fixed input, perfect speedup
  - The algorithm can change to become more ||
  - Problem instances grow implying proportion of work that is sequential may reduce
  - … Many, many realities including parallelism in 'sequential' execution imply analysis is simplistic
- *Amdahl is a fact; it's not a show-stopper*

37

Performance Loss: Overhead

- Threads and processes incur overhead



- Obviously, the cost of creating a thread or process must be recovered through parallel performance:

$(t + o_s + o_{td} + \text{cost}(t))/2 < t$

$\therefore\ o_s + o_{td} + \text{cost}(t) < t$

t = execution time
o_s = setup, o_{td} = tear down
cost(t) = all other || costs

38

Performance Loss: Contention

- Contention, the action of one processor interferes with another processor's actions, is an elusive quantity
  - Lock contention: One processor's lock stops other processors from referencing; they must wait
  - Bus contention: Bus wires are in use by one processor's memory reference
  - Network contention: Wires are in use by one packet, blocking other packets
  - Bank contention: Multiple processors try to access a memory simultaneously

Contention is very time dependent, that is, variable

39

Performance Loss: Load Imbalance

- Load imbalance, work not evenly assigned to the processors, underutilizes parallelism
  - The assignment of *work*, not data, is key
  - Static assignments, being rigid, are more prone to imbalance
  - Because dynamic assignment carries overhead, the quantum of work must be large enough to amortize the overhead
  - With flexible allocations, load balance can be solved late in the design programming cycle

40

The Best Parallel Programs …

- Performance is maximized if processors execute continuously on local data without interacting with other processors
  - To unify the ways in which processors could interact, we adopt the concept of dependence
  - A *dependence* is an ordering relationship between two computations
    - Dependences are usually induced by read/write
    - Dependences that cross processor boundaries induce a need to synchronize the threads

Dependences are well-studied in compilers

41

Dependences

- Dependences are orderings that must be maintained to guarantee correctness
  - Flow-dependence: read after write **True**
  - Anti-dependence: write after read **False**
  - Output-dependence: write after write **False**
- True dependences affect correctness
- False dependences arise from memory reuse

42

Example of Dependences

- Both true and false dependences

```
1.    sum = a + 1;
2.    first_term = sum * scale1;
3.    sum = b + 1;
4.    second_term = sum * scale2;
```

43

Example of Dependences

- Both true and false dependences

```
1.    sum = a + 1;
2.    first_term = sum * scale1;
3.    sum = b + 1;
4.    second_term = sum * scale2;
```

- Flow-dependence read after write; must be preserved for correctness
- Anti-dependence write after read; can be eliminated with additional memory

44

Removing Anti-dependence

- Change variable names

```
1.    sum = a + 1;
2.    first_term = sum * scale1;
3.    sum = b + 1;
4.    second_term = sum * scale2;
```

```
1.    first_sum = a + 1;
2.    first_term = first_sum * scale1;
3.    second_sum = b + 1;
4.    second_term = second_sum * scale2;
```

45

Granularity

- Granularity is used in many contexts…here *granularity* is the amount of work between cross-processor dependences
  - Important because interactions usually cost
  - Generally, larger grain is better
    - \+ fewer interactions, more local work
    - \- can lead to load imbalance
  - Batching is an effective way to increase grain

46

Locality

- The CTA motivates us to maximize locality
  - Caching is the traditional way to exploit locality … but it doesn't translate directly to ||ism
  - Redesigning algorithms for parallel execution often means repartitioning to increase locality
  - Locality often requires redundant storage and redundant computation, but in limited quantities they help

47

Measuring Performance

- Execution time … what's time?
  - 'Wall clock' time
  - Processor execution time
  - System time
- Paging and caching can affect time
  - Cold start vs warm start
- Conflicts w/ other users/system components
- Measure kernel or whole program

48

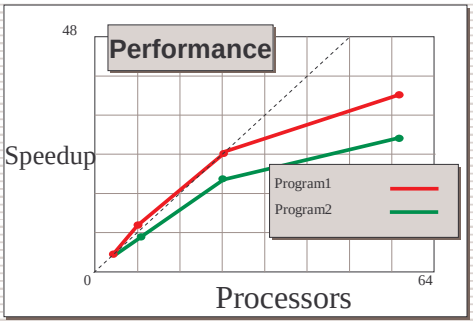
FLOPS

- Floating Point Operations Per Second is a common measurement for scientific pgms
  - Even scientific computations use many ints
  - Results can often be influenced by small, low-level tweaks having little generality: mult/add
  - Translates poorly across machines because it is hardware dependent
  - Limited application

49

Speedup and Efficiency

- Speedup is the factor of improvement for *P* processors: T_S/T_P



Efficiency = Speedup/*P*

50

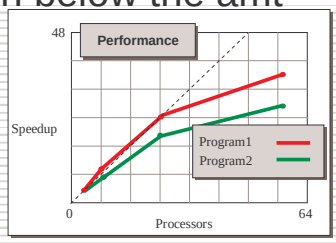
Issues with Speedup, Efficiency

- Speedup is best applied when hardware is constant, or for family within a generation
  - Need to have computation, communication is same ratio
  - Great sensitivity to the T_S value
    - T_S should be time of best sequential program on 1 processor of ||-machine
    - $T_{P=1} \neq T_S$ Measures *relative speedup*

51

Scaled v. Fixed Speedup

- As *P* increases, the amount of work per processor diminishes, often below the amt needed to amortize costs
- Speedup curves bend dn
- Scaled speedup keeps the work per processor constant, allowing other affects to be seen
- Both are important



If not stated, speedup is **fixed** speedup

52

Assignment

- Read Chapter 4