CS575: Parallel Processing Sanjay Rajopadhye CSU

Lecture 2: Parallel Computer Models

Course Topics

- Introduction, background
  - Complexity, orders of magnitude, recurrences
- Models of parallel computing & communication
- Performance, efficiency & speedup
  - Amdahl, Gustaffson, strong/weak scaling
- Parallel algorithms
  - Dense linear algebra, prefix sums, graph algorithms, FFT
- Slides/lectures complement the text and web resources

Course Organization

- Streamline 475-575 flow
- Focus on algorithms and analysis
  - Separate courses on distributed systems, networking
- Advanced CUDA programming
  - Performance tuning
  - Using roofline techniques
  - Guided by analysis
- Beyond CUDA

Sequential Algorithms

- Efficient Sequential Algorithms
  - Optimize for time or space (memory)
- Performance is portable
  - Efficient program on Pentium ~ Efficient program on Opteron
- *Algorithmic analysis enabled separation of concerns*
- Asymptotic analysis: problem size N

Parallel Algorithms

- Two independent parameters
  - Problem size *N* same as before
  - Processor count *P* also grows asymptotically
- Cost of parallelism:
  - Communication
  - Synchronization
- Efficient parallel algorithms (machine/model dependent)
  - Start with the best sequential algorithm
    - (almost) always the best strategy
- Recomputation (redundant computation) is sometimes better

Speedup & efficiency

- Definitions
- Bounds
- “superlinear”
- Why is that wrong
- Ideal speedup
- Isoefficency

// Programming Paradigms

- Sequential Paradigms: imperative, object oriented, declarative (functional, relational), …
- Parallel paradigms
  - Language style (same as seq)
  - Parallelism style:
    - Implicit parallelism
    - Explicit parallelism
      - Shared memory
      - Distributed memory

Implicit Parallelism

- Sequential Paradigms: Super compilers
  - Extract parallelism from sequential code
  - Programmer has to do nothing, compiler distributes data, creates and schedules tasks
    - Very limited success (only in niche domains)
- Implicit parallelism with declarative programs
    - Parallel logic languages
    - Parallel functional programming

Functional Languages

- No side effects, order of execution less constrained
- F (P(x,y), Q(y,z)) P and Q can be executed in parallel
- Simple single assignment memory model:
  - no pointers, no write after read or write after write hazards (dataflow semantics)
- FP was long doomed as too high level too inefficient, because the simple memory model causes lots of copies
- FP is coming back: MapReduce approach in data centers (Google) is a data parallel functional paradigm

CS575 lecture 1 9 1/24/14

Explicit Parallelism

- Multithreading:
  - OpenMP & CUDA
  - P(x,y), Q(y,z)) P and Q can be executed in parallel
- Message Passing (distributed memory)
  - MPI
- Programming becomes more complicated
  - Synchronization (semaphores, locks, messages)
  - creation, allocation, scheduling of processes
  - data partitioning

CS575 lecture 1 10 1/24/14

Background: algorithm analysis

- References:
  - *"Introduction to Algorithms,"* Cormen Rivest Leiserson Stein
  - Other texts and/or wiki
- Topics:
  - Intro, asymptotic growth of functions, summations recurrences
- Optional/advanced:
  - Average case analysis
  - Amortized analysis

Orders of magnitude

O, θ and Ω

- A function $f(n) = O(g(n))$ iff $\exists$ positive constants c and n_0 such that $\forall\ n \geq n_0$ (i.e., eventually/asymptotically) $f(n) < g(n)$ So, g is an upper bound on f
- A function $f(n) = \Omega(g(n))$ iff $\exists$ positive constants c and n_0 such that $\forall\ n \geq n_0$ (i.e., eventually/asymptotically) $f(n) > g(n)$ So, g is a lower bound on f
- A function $f(n) = \theta(g(n))$, i.e., g is a tight bound on f (and vice versa) iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

Algorithmic complexity

Complexity of

- some property (e.g., execution time, memory requirement, etc.)
- of algorithm(s) to solve a problem
  - specific algorithm (complexity of the algorithm)
  - lower bounds, quantified over all algorithms (universal quantifier) to solve that problem: complexity of the problem

A problem may be "closed" LB= θ (UB) or "have a gap"

Recurrence Relations

- Algorithmic complexity often described using recurrence relations:
  $f(n) = g(f(1), f(2), \ldots f(n-1))$
- Two common classes:
  - Linear:
    - constant number of occurrences of f and argument of each one is just a some constant less than n
    - g is a linear function, with possibly one additional term
  - D&C (divide and conquer)
    - constant number of occurrences of f and argument of each one is just a some constant factor of n
  - Covered in CS 420 (& CS420dl)

Repeated Substitution

- Simple recurrence relations (one recurrent term in the rhs) can sometimes be solved using repeated substitution
- Two types: Linear and D&C
  - $F(n) = a\,F(n\text{-}d) + g(n)$, base: $F(1)=v_1$
  - $F(n) = a\,F(n/d) + g(n)$, base: $F(1)=v_1$
- Two questions:
  - what is the pattern
  - how often is it applied until we hit the base case

CS575 lecture 1 15 1/23/14

Linear Example

$M(n) = 2M(n\text{-}1)+1,\ M(1)=1$ recognize this one?

$= 2(2M(n\text{-}2)+1)+1$

$= 4M(n\text{-}2)+2+1 = 4(2M(n\text{-}3)+1)+2+1$

$= 8M(n\text{-}3)+4+2+1 = \ldots$ inductive step …

$= 2^kM(n\text{-}k)+2^{k-1}+2^{k-2}+\ldots+2+1$

Hit the base case for $k = n\text{-}1$:

$= 2^{n-1}M(1)+2^{n-1}+2^{n-2}+\ldots+2+1$

$= 2^n\text{-}1$

CS575 lecture 1 16 1/23/14

D&C Example

Merge sort:

T(n) = 2T(n/2) + n, T(1)=1 (and n = 2k)

= 2(2(T(n/4) + n/2) + n

= 4T(n/4) + 2n

= 8T(n/8) + 3n … inductive step …

= 2kT(n/2k) + kn

hit base for k = log n

= n + kn = O(n log n)

Another one: binary search

G(n) = G(n/2) + c, G(1)=1 (and n = 2k)

= (G(n/4) + c) + c

= G(n/4) + 2c

= G(n/8) + 3c … inductive step …

= G(n/2k) + kc

hit base for k = log n

= G(1) + c log n = O(log n)

Master Method

- Cookbook solution, based on repeated substitution for a number of common cases
  $f(n) = c\, f(n/d) + k\, n^p$
  - if $C < d^p$ then $A_n = O(n^p)$
    e.g., $A_n = 3\, A_{n/2} + n^2$
  - if $C = d^p$ then $A_n = O(n^p \log(n))$
    e.g., $A_n = 2A_{n/2} + n$
  - if $C > d^p$ then $A_n = O(n^{\log_d c})$
    e.g., $A_n = 3\, A_{n/2} + n$
- Covered in CS 420 (& CS420dl)
- Do

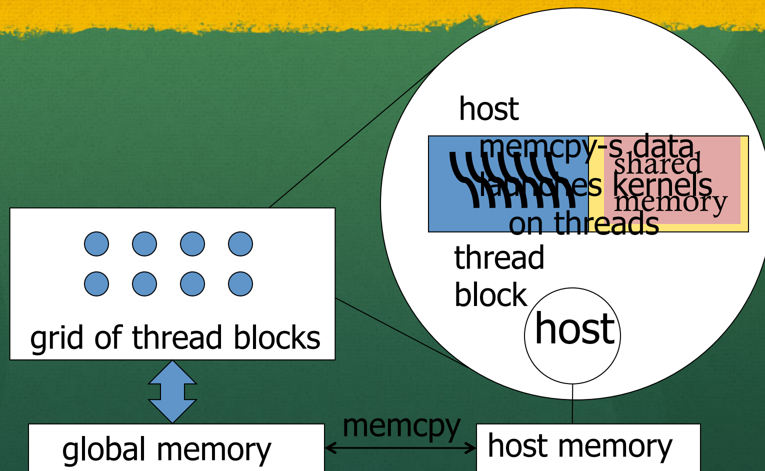
CS575 lecture 1 19 1/23/14

Examples

- Merge Sort
  $T(n) = 2T(n/2) + n,\ T(1)=1$
  C=? d=? p=? d^p=?
  T(n) = O(???)
- Binary Search
  $f(n) = f(n/2)+c \quad f(1)=1$
  C=? d=? p=? d^p=?
  f(n) = O(???)

CS575 lecture 1 20 1/23/14

GPU Programming



Questions

- How do threads and thread blocks get allocated to SMPs
- How do they synchronize/communicate
- How do they disambiguate memory addresses
  - Which thread writes/reads-from where?
  - What if the addresses are in conflict?
- How are things different at the two levels of memory?
- What about caches?

Thread allocation

Static allocation

- Program declares a number of (virtual) thread blocks – many more than number of SMs
- Run time system allocates them (details unspecified) to thread blocks – main idea non-preemptively scheduled, each TB runs through to completion
- Within a TB – program has a (virtual) number of threads each thread knows of two parameters – its thread id within the TB and the TBs id within the grid.
- Code is parametric, so
  - programmer's responsibility to write code so the algorithm is correctly implemented by this virtual collection of threads.

Thread Allocation

Static Allocation

- Program declares a (virtual)