

Parallel Programming

Instructor: Josep Torrellas
CS533

Problem Selection

- Can use parallel computation for 2 things:
 - Speed up an existing application
 - Improve quality of result for an appl (more compute power allows revolutionary change in algorithm)
- Appl should be compute intensive and unless significant speedup is achievable, parallelization is not worth effort

Fundamental Limits: Amdahl's Law

S = serial fraction of computation

C = parallelizable fraction

$$S + C = 1$$

T_p = execution time using P processors = $(S + C/P)T_1$

$$\text{Speedup} = T_1/T_p = 1/(S + C/P)$$

$$\text{Maximum speedup (infinite } P) = 1/S$$

example: $S = 0.05$, $\text{MaxSpeedup} = 20$ (perfect paral)

- Serial fractions have a way of appearing in non-obvious ways, e.g. profile:
 - input: 10%
 - compute setup: 15%
 - computation: 75%
- If only part 3 parallelized: $S_{\max} = 4$
- If only part 2 and 3 parallelized: $S_{\max} = 10$
- Have to live with S_{\max} if cannot change algo
- Honesty: have to compare to best sequential algo

Speedups with Scaled Pbm Sizes

- Speedup given constant problem size
- Speedup given constant wait time (assume perfect speedup and determine the problem size that can be computed given the same wait time)
- Linear scaling of problem (data set) with the number of Pes (amount of memory per processor is constant)..... What is the largest problem solvable on the machine?

Type of Parallelization

- User controlled vs automatic
- User controlled:
 - Programmer tells all procs what to do at all times
 - more freedom, but significant effort from programmer
 - several problems:
 - Programmer may not know the details of the machine as compiler
 - Repeated work done in task queue implementations
 - Sophistication needed to write programs with good locality & grain
- Automatic: compilers

Approaches: Exact vs Inexact

- Begin with sequential code or outline algorithm
- Exact parallelism:
 - Definition: all data dependences remain intact
 - Advantage: answer guaranteed to be the same as sequential implementation independent of the number of processors
 - Problem: unnecessary dependences causes inefficiency: e.g. relaxation

Approaches: Exact vs Inexact

- Inexact parallelism:
 - Definition: “relax” data dependences: allow “stale” data to be used, instead of most-up-to-date
 - Used in both numerical solution techniques and combinatorial optimization
 - Reduces synchronization overhead
 - Usually in context of iterative algorithms which still converge to right answer
 - may or may not be faster

Ex1: Circuit Simulation with Relaxation

- At every time-step, want to determine all node voltages (V_i)
- Non-relaxation: solve non-linear equations relating to all voltages
- Relaxation: solve local equations iteratively and wait for convergence
- Parallel relaxation: assign pieces of circuit to different processors and synchronize at every iteration
- Chaotic (inexact) relaxation: don't synchronize, less overhead but slower convergence; may only be appropriate in fine-grain massive parallelism

Ex1: Upper Atmosphere Wind Tunnel Simulation

- Can't use fluid flow equations, because too few particles
- Calculation position of every particle at every time step
- Check for collisions on a cell-by-cell basis, plus boundary collisions: if two particles are in the same cell or if a particle hits a boundary then it is considered a collision
- Distributed loop to process particles:
 - Processors pick up particles, calculate their new positions, and check for collisions
- Need to lock cell if changing
- Inexact: Don't lock, may lose a few collisions, but probability is small
- Speedup on 10 Pes: with locks: 7.4, without: 8.9

Other examples

- Parallel simulated annealing
- Standard cell routing
- ...

Interesting theoretical issue: Convergence

Speculative Parallelism

- Do more work than may need to be done
- Must be able to “back up” to correct state
- Examples:
 - The two sides of an IF statement
 - Parallel Alpha-Beta Search: may search portions of the tree that would have been eliminated in sequential search
 - Chaotic parallel logic simulation

Logic Simulation

- Determine binary values of signals over time for digital circuits
 - Event Driven: At every time step, pull logic transition events off queue, compute changes due to event, and generate event at later time
 - Parallel ED: Several procs process events
 - Chaotic PED:
 - Each element has a local clock
 - Simulation may proceed ahead even though new events on all inputs are not present
 - If event occurs in past time, and value is different from that assumed, then back up (otherwise it is a win)

Orthogonal Parallelism

- Think about parallelism like slicing an apple:
 - Make N cuts on X axis: N pieces
 - Make M cuts on Y axis: $N \times M$ pieces
 - Make K cuts on Z axis: $N \times M \times K$ pieces
- Because slice directions are orthogonal: get mpy
- A factor of 2-3 on each axis gets big payoff

Examples

- Nested loop
 - for I=1,10
 - for j = 1,5
 - indep work
- Ray tracing in Graphics:
 - Following light rays from light source and bounce off objects
 - Each ray in parallel
 - Inside each ray, computation can be parallelized
 - Speedup = #Rays * Speedup for each ray

Design Tradeoffs in Parallel Decomposition

1. Granularity vs Communication/Synchronization vs Load Balance

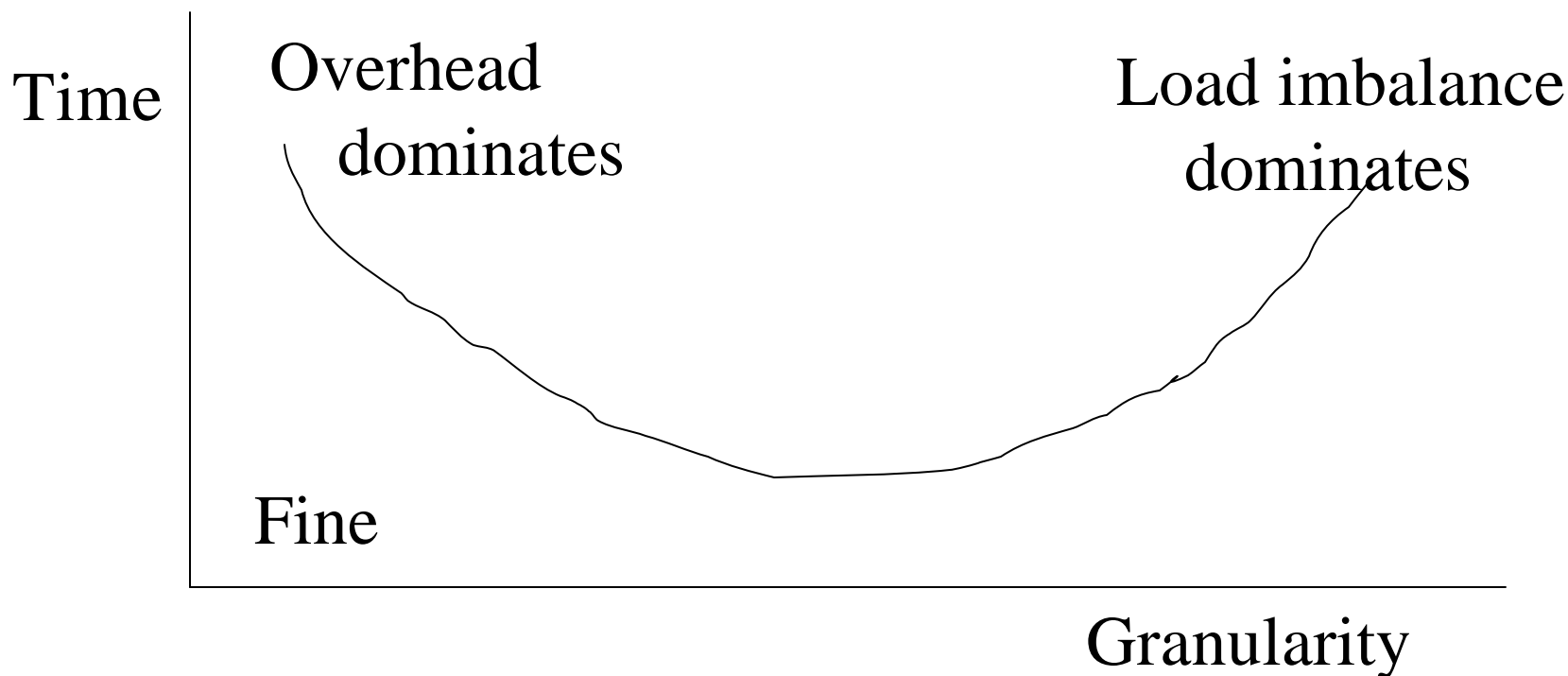
- Granularity:
 - Amount of computation between interprocess comm.
 - Tricky: interprocess communication = data transmission or synchronization. For shared-memory, data is cheap, not for message passing
 - Not necessarily “constant size”
- Grain size
 - fine: program chopped into many small pieces
 - coarse: fewer larger pieces

Impact

- Parallel decomposition overhead:
 - As granularity decreases, overhead increases
 - E.g. time taken by each process to obtain task (serialization if single task queue)
- Load balance:
 - As granularity decreases, get better balance
 - E.g. two processors, split into 3 tasks:
 - Speedup is not 2, but less than that

Graph

Typical graph of execution time using P processors
(if grain size can be varied)



Execution Time

$$E = S + (C/P)(1 + K_p) + O_p$$

K_p : cost due to load imbalance, communication

O_p : other overhead

Static vs Dynamic

- If grain size is constant and the number of tasks is known, then can statically assign tasks to processors (reduce overhead)
- If not, then need some dynamic assignment (task queue, self-scheduled loop)
- Possible even to have a dynamic decision about whether or not to spawn

Design Tradeoffs in Parallel Decomposition

2. Copying data vs recreating it

- Usually on non-shared-memory systems, where data transmission may be slow
- Can send out updated information, or just enough information to recalculate at each processor
- Example:
 - Each processor has copy of current state (TSP)
 - Each processor makes “moves” (swap two cities)
 - Could broadcast entire tour or just send move
 - Could use both; problem: recalculation is like serial calculation

Tuning a Parallel Program

- Profile to find out:
 - Time spent waiting for a lock (or barrier)
 - Time spent in lock critical section
 - Time spent in non-critical section, between barriers

Typical Bottlenecks: Task Queue

- One central queue may be a bottleneck
 - distributed task queues: p procs, $q \leq p$ queues
 - distributes contention across queues
 - Cost: if 1st queue processor checks is empty, it has to go and look at others
- Task insertion is an issue:
 - Number of tasks generated by each processor is uniform--> each processor is assigned a specific task queue for insertion
 - Task generation is non-uniform (e.g. one processor generates all tasks)--> tasks should be uniformly, randomly spread among queues
- Unequal-sized tasks: scheduling problem
 - Optimal scheduling is NP-hard
 - Even harder if we do not know time of generation of each tasks
- Priority queues: Give more important jobs higher priority --> get executed sooner

Typical Bottlenecks: Task Queue

- To speed up insertion and deletion: arrays of fixed memory rather than allocating and deallocating memory
- Termination condition for multiple task queues is tricky:
 - Must know that all processors are waiting (not generating more tasks)
 - Keep counts of number of processors waiting, and continuously check queues until counter = max number of processors

Typical Bottlenecks: Mem Allocation

- If central memory allocation is a bottleneck, use distributed free lists
- Each processor keeps its own list of available memory that has been freed
- Processor only goes to central allocation when that runs out
- Less efficiency in use of memory
- Can re-distribute sometimes

Typical Bottlenecks: Exclusive Access to Large Data Structs

- One lock for entire structure is bottleneck (for example, parallel simulated annealing for graph partitioning)
- One lock for every item may be an overkill (wastes space)
- For matrices: can lock 1 row, 1 column, or an area (lock address determined by index)
- Non-matrices: hash a key that points to lock

Producer-Consumer

- If single producer, single consumer, don't need locks, as long as single location is kept consistent
- Use a sync location (S)
 - Initialize $S=0$
 - Producer loads data into buffer, sets $S=1$
 - Consumer waits for $S=1$, takes data, sets $S=0$
 - $S=0$ signals producer it can refill the buffer