CS533: Introduction to Parallel Programming

Josep Torrellas

University of Illinois in Urbana-Champaign

Parallel Programming: Problem Selection

- Can use parallel computation for 2 things:
    - Speed up an existing application
    - Improve quality of result for an application (more computer power allows revolutionary change in algorithm)
- Application 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 = $(S + \frac{C}{P}) * T_1$
- $Speedup = T_1 / T_p = 1/(S + \frac{C}{P})$
- Maximum speedup (∞ P) = $\frac{1}{S}$
- example: S = 0.05, MaxSpeedup = 20 (perfect parallelism)

Fundamental Limits: Amdahl's Law

- Serial fractions have a way of appearing in non-obvious ways, e.g. profile:
  1. input: 10%
  2. compute setup: 15%
  3. 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 algorithm
- Honesty: have to compare to sequential algorithm

Speedups with Scaled Problem Sizes

- Scaling modes:
  - Strong scaling: keep the problem size constant
  - Weak scaling: increase the problem size
- Speedups:
  - Speedup given constant problem size
  - Speedup given constant wait time (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 data per processor is constant)

Exact vs Inexact Parallelism

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

Example: 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

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 processors 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 multiplication
- 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

Granularity vs Communication/Synchronization vs Load Balance

Granularity:

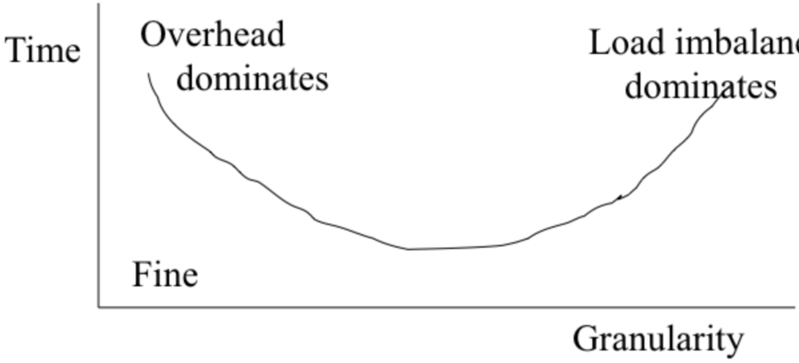
- Amount of computation between interprocess communication
- Tricky: interprocess communication, e.g. data transmission or synchronization. For shared-memory, data is cheap, not for message passing
- Not necessarily "constant size"

Grain size:

- Fine: program chopped into many small pieces
- Coarse: fewer larger pieces

Graph

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



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

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

Typcial Bottlenecks: Task Queue

- One central queue may be a bottleneck
  - ditributed task queues: p procs, $q \leq p$ queues
  - distributed 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 $\rightarrow$ each processor is assigned a specific task queue for insertion
  - Task generation is non-uniform (e.g. one processor generates all tasks) $\rightarrow$ 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 $\rightarrow$ get executed sooner

Typical Bottlenecks: Task Queue (continued)

- 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: Memory 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 redistribute sometimes

Typical Bottlenecks: Exclusive Access to Large Data Structures

- 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, use a volatile flag.
- How the flag F is used:
    - Initialize $F = 0$
    - Producer loads data into buffer, sets $F = 1$
    - Consumer waits for $F = 1$, takes data, sets $F = 0$
    - $F = 0$ signals producer it can refill the buffer