Lecture 22:
Load balancing

David Bindel

15 Nov 2011

Logistics

- Proj 3 in!
  - Get it in by Monday with penalty.

Inefficiencies in parallel code

- Poor single processor performance
  - Typically in the memory system
  - Saw this in HW 1
- Overhead for parallelism
  - Thread creation, synchronization, communication
  - Saw this in HW 2-3
- Load imbalance
  - Different amounts of work across processors
  - Different speeds / available resources
  - Insufficient parallel work
  - All this can change over phases

Where does the time go?

- Load balance looks like high, uneven time at synchronization
- ... but so does ordinary overhead if synchronization expensive!
- And spin-locks may make synchronization look like useful work
- And ordinary time sharing can confuse things more
- Can get some help from tools like TAU (Timing Analysis Utilities)

Reminder: Graph partitioning

- Graph $G = (V, E)$ with vertex and edge weights
- Try to evenly partition while minimizing edge cut (comm volume)
- Optimal partitioning is NP complete – use heuristics
  - Spectral
  - Kernighan-Lin
  - Multilevel
- Tradeoff quality vs speed
- Good software exists (e.g. METIS)

The limits of graph partitioning

What if

- We don't know task costs?
- We don't know the communication pattern?
- These things change over time?

May want *dynamic* load balancing.

Basic parameters

- Task costs
  - Do all tasks have equal costs?
  - When are costs known (statically, at creation, at completion)?
- Task dependencies
  - Can tasks be run in any order?
  - If not, when are dependencies known?
- Locality
  - Should tasks be on the same processor to reduce communication?
  - When is this information known?

Task costs

- Easy: equal unit cost tasks
  - Branch-free loops
  - Much of HW 3 falls here!
- Harder: different, known times
  - Example: general sparse matrix-vector multiply
- Hardest: task cost unknown until after execution
  - Example: search

Q: Where does HW 2 fall in this spectrum?

Dependencies

- Easy: dependency-free loop (Jacobi sweep)
- Harder: tasks have predictable structure (some DAG)
- Hardest: structure changes dynamically (search, sparse LU)

Locality/communication

- Easy: tasks don't communicate except at start/end (embarrassingly parallel)
- Harder: communication is in a predictable pattern (elliptic PDE solver)
- Communication is unpredictable (discrete event simulation)

A spectrum of solutions

How much we can do depends on cost, dependency, locality

- Static scheduling
  - Everything known in advance
  - Can schedule offline (e.g. graph partitioning)
  - See this in HW 3
- Semi-static scheduling
  - Everything known at start of step (or other determined point)
  - Can use offline ideas (e.g. Kernighan-Lin refinement)
  - Saw this in HW 2
- Dynamic scheduling
  - Don't know what we're doing until we've started
  - Have to use online algorithms
  - Example: most search problems

Search problems

- Different set of strategies from physics sims!
- Usually require dynamic load balance
- Example:
  - Optimal VLSI layout
  - Robot motion planning
  - Game playing
  - Speech processing
  - Reconstructing phylogeny
  - ...

Example: Tree search

- Tree unfolds dynamically during search
- May be common subproblems along different paths (graph)
- Graph may or may not be explicit in advance

Search algorithms

Generic search:

Put root in stack/queue
while stack/queue has work
    remove node n from queue
    if n satisfies goal, return
    mark n as searched
    add viable unsearched children of n to stack/queue
        (Can branch-and-bound)

Variants: DFS (stack), BFS (queue), A^* (priority queue), ...

Simple parallel search

- Static load balancing: each new task on an idle processor until all have a subree
  - Not very effective without work estimates for subtrees!
  - How can we do better?

Centralized scheduling

Idea: obvious parallelization of standard search

- Shared data structure (stack, queue, etc) protected by locks
- Or might be a manager task

Teaser: What could go wrong with this parallel BFS?

```
Put root in queue
fork
  obtain queue lock
  while queue has work
    remove node n from queue
    release queue lock
    process n, mark as searched
    obtain queue lock
    add viable unsearched children of n to queue
  release queue lock
join
```

Centralized task queue

- Called *self-scheduling* when applied to loops
  - Tasks might be range of loop indices
  - Assume independent iterations
  - Loop body has unpredictable time (or do it statically)
- Pro: dynamic, online scheduling
- Con: centralized, so doesn't scale
- Con: high overhead if tasks are small

Variations on a theme

How to avoid overhead? Chunks! (Think OpenMP loops)

- Small chunks: good balance, large overhead
- Large chunks: poor balance, low overhead
- Variants:
  - Fixed chunk size (requires good cost estimates)
  - Guided self-scheduling (take $\lceil R/p \rceil$ work, R = tasks remaining)
  - Tapering (estimate variance; smaller chunks for high variance)
  - Weighted factoring (like GSS, but take heterogeneity into account)

Beyond centralized task queue

Basic *distributed* task queue idea:

- Each processor works on part of a tree
- When done, get work from a peer
- *Or* if busy, push work to a peer
- Requires asynch communication

Also goes by work stealing, work crews...

Implemented in Cilk, X10, CUDA, ...

Picking a donor

Could use:

- Asynchronous round-robin
- Global round-robin (keep current donor pointer at proc 0)
- Randomized – optimal with high probability!

Diffusion-based balancing

- Problem with random polling: communication cost!
  - But not all connections are equal
  - Idea: prefer to poll more local neighbors
- Average out load with neighbors $\implies$ diffusion!

Mixed parallelism

- Today: mostly coarse-grain *task* parallelism
- Other times: fine-grain *data* parallelism
- Why not do both?
- *Switched* parallelism: at some level switch from data to task