Load Balancing Part 1: Dynamic Load Balancing

Kathy Yelick
yelick@cs.berkeley.edu
www.cs.berkeley.edu/~yelick/cs194f07
Implementing Data Parallelism

• Why didn’t data parallel languages like NESL, *LISP, pC++, HPF, ZPL take over the world in the last decade?
  
• 1) parallel machines are made from commodity processors, not 1-bit processors; the compilation problem is nontrivial (not necessarily impossible) and users were impatient.

• 2) data parallelism is not a good model when the code has lots of branches (recall “turn off processors” model).

![Logical execution of statement](image)

![Mapping to bulk-synchronous execution](image)
Load Imbalance in Parallel Applications

The primary sources of inefficiency in parallel codes:
• Poor single processor performance
  • Typically in the memory system
• Too much parallelism overhead
  • Thread creation, synchronization, communication
• Load imbalance
  • Different amounts of work across processors
    • Computation and communication
  • Different speeds (or available resources) for the processors
    • Possibly due to load on the machine
• How to recognizing load imbalance
  • Time spent at synchronization is high and is uneven across processors, but not always so simple …
Measuring Load Imbalance

• Challenges:
  • Can be hard to separate from high synch overhead
  • Especially subtle if not bulk-synchronous
  • “Spin locks” can make synchronization look like useful work
  • Note that imbalance may change over phases
  • Insufficient parallelism always leads to load imbalance
  • Tools like TAU can help (acts.nersc.gov)
Tough Problems for Data Parallelism

- Hierarchical parallelism
  - E.g., Loosely connected “cities” of life variation of HW2
  - List of grids representation; nested data parallelism might work
  - Corresponds to real “Adaptive Mesh Refinement” algorithms

- Divide and conquer parallelism
  - E.g., Quicksort relies on either nested data parallelism or tasks

- Branch-and-bound search
  - Game tree search: consider possible moves, search recursively
  - Problem: amount of work depends on computed values; not a function only of input size

- Event-driven execution
  - Actor model for multi-player games, asynchronous circuit simulation, etc.

Load balancing is a significant problem for all of these
Load Balancing Overview

Load balancing differs with properties of the tasks (chunks of work):

• **Tasks costs**
  • Do all tasks have equal costs?
  • If not, when are the costs known?
    • Before starting, when task created, or only when task ends

• **Task dependencies**
  • Can all tasks be run in any order (including parallel)?
  • If not, when are the dependencies known?
    • Before starting, when task created, or only when task ends

• **Locality**
  • Is it important for some tasks to be scheduled on the same processor (or nearby) to reduce communication cost?
  • When is the information about communication known?
Outline

• Motivation for Load Balancing
• Recall graph partitioning as load balancing technique
• Overview of load balancing problems, as determined by
  • Task costs
  • Task dependencies
  • Locality needs
• Spectrum of solutions
  • Static - all information available before starting
  • Semi-Static - some info before starting
  • Dynamic - little or no info before starting
• Survey of solutions
  • How each one works
  • Theoretical bounds, if any
  • When to use it
Task Cost Spectrum

Schedule a set of tasks under one of the following assumptions:

*Easy:* The tasks all have equal (unit) cost.

\[ \begin{array}{c}
\text{n items} \\
\hline
\end{array} \quad \begin{array}{c}
\text{p bins} \\
\hline
\end{array} \]

branch-free loops

*Harder:* The tasks have different, but known, times.

\[ \begin{array}{c}
\text{n items} \\
\hline
\end{array} \quad \begin{array}{c}
\text{p bins} \\
\hline
\end{array} \]

sparse matrix-vector multiply

*Hardest:* The task costs unknown until after execution.

\[ \begin{array}{c}
\text{n items} \\
\hline
\end{array} \quad \begin{array}{c}
\text{p bins} \\
\hline
\end{array} \]

GCM, circuits, search
Task Dependency Spectrum

Schedule a graph of tasks under one of the following assumptions:

**Easy:** The tasks can execute in any order. dependence
free loops

**Harder:** The tasks have a predictable structure.
wave-front  out-tree  in-tree  general dag
balanced or unbalanced

**Hardest:** The structure changes dynamically (slowly or quickly) search, sparse LU
Task Locality Spectrum (Communication)

Schedule a set of tasks under one of the following assumptions:

**Easy:** The tasks, once created, do not communicate.

**Harder:** The tasks communicate in a predictable pattern.

![Regular and irregular graphs](image)

**Hardest:** The communication pattern is unpredictable.

- *embarrassingly parallel*
- *PDE solver*
- *discrete event simulation*
Spectrum of Solutions

A key question is when certain information about the load balancing problem is known. Many combinations of answer leads to a spectrum of solutions:

- **Static scheduling.** All information is available to scheduling algorithm, which runs before any real computation starts.
  - *Off-line* algorithms make decisions before execution time
- **Semi-static scheduling.** Information may be known at program startup, or the beginning of each timestep, or at other well-defined points.
  - Offline algorithms may be used, between major steps.
- **Dynamic scheduling.** Information is not known until mid-execution.
  - *On-line* algorithms make decisions mid-execution
Dynamic Load Balancing

- Motivation for dynamic load balancing
  - Search algorithms as driving example
- Centralized load balancing
  - Overview
  - Special case for schedule independent loop iterations
- Distributed load balancing
  - Overview
  - Engineering
  - Theoretical results

- Example scheduling problem: mixed parallelism
  - Demonstrate use of coarse performance models
Search

• Search problems are often:
  • Computationally expensive
  • Have very different parallelization strategies than physical simulations.
  • Require dynamic load balancing

• Examples:
  • Optimal layout of VLSI chips
  • Robot motion planning
  • Chess and other games (N-queens)
  • Speech processing
  • Constructing phylogeny tree from set of genes
Example Problem: Tree Search

• In Tree Search the tree unfolds dynamically
• May be a graph if there are common sub-problems along different paths
• Graphs unlike meshes which are precomputed and have no ordering constraints
Sequential Search Algorithms

- **Depth-first search (DFS)**
  - Simple backtracking
    - Search to bottom, backing up to last choice if necessary
  - Depth-first branch-and-bound
    - Keep track of best solution so far ("bound")
    - Cut off sub-trees that are guaranteed to be worse than bound
  - Iterative Deepening
    - Choose a bound on search depth, d and use DFS up to depth d
    - If no solution is found, increase d and start again
    - Iterative deepening A* uses a lower bound estimate of cost-to-solution as the bound

- **Breadth-first search (BFS)**
  - Search across a given level in the tree
Depth vs Breadth First Search

• DFS with Explicit Stack
  • Put root into Stack
    • Stack is data structure where items added to and removed from the top only
  • While Stack not empty
    • If node on top of Stack satisfies goal of search, return result, else
      – Mark node on top of Stack as “searched”
      – If top of Stack has an unsearched child, put child on top of Stack, else remove top of Stack

• BFS with Explicit Queue
  • Put root into Queue
    • Queue is data structure where items added to end, removed from front
  • While Queue not empty
    • If node at front of Queue satisfies goal of search, return result, else
      – Mark node at front of Queue as “searched”
      – If node at front of Queue has any unsearched children, put them all at end of Queue
      – Remove node at front from Queue
Parallel Search

• Consider simple backtracking search
• Try static load balancing: spawn each new task on an idle processor, until all have a subtree

Load balance on 2 processors

Load balance on 4 processors

• We can and should do better than this …
Centralized Scheduling

- Keep a queue of task waiting to be done
  - May be done by manager task
  - Or a shared data structure protected by locks
Centralized Task Queue: Scheduling Loops

- When applied to loops, often called self scheduling:
  - Tasks may be range of loop indices to compute
  - Assumes independent iterations
  - Loop body has unpredictable time (branches) or the problem is not interesting

- Originally designed for:
  - Scheduling loops by compiler (or runtime-system)
  - Original paper by Tang and Yew, ICPP 1986

- This is:
  - Dynamic, online scheduling algorithm
  - Good for a small number of processors (centralized)
  - Special case of task graph – independent tasks, known at once
**Variations on Self-Scheduling**

- Typically, don’t want to grab smallest unit of parallel work, e.g., a single iteration
  - Too much contention at shared queue
- Instead, choose a chunk of tasks of size $K$.  
  - If $K$ is large, access overhead for task queue is small
  - If $K$ is small, we are likely to have even finish times (load balance)

- (at least) Four Variations:
  1. Use a fixed chunk size
  2. Guided self-scheduling
  3. Tapering
  4. Weighted Factoring
Variation 1: Fixed Chunk Size

• Kruskal and Weiss give a technique for computing the optimal chunk size

• Requires a lot of information about the problem characteristics
  • e.g., task costs as well as number

• Not very useful in practice.
  • Task costs must be known at loop startup time
  • E.g., in compiler, all branches be predicted based on loop indices and used for task cost estimates
Variation 2: Guided Self-Scheduling

• Idea: use larger chunks at the beginning to avoid excessive overhead and smaller chunks near the end to even out the finish times.
  • The chunk size $K_i$ at the $i^{th}$ access to the task pool is given by $\text{ceiling}(R_i/p)$
  • where $R_i$ is the total number of tasks remaining and
  • $p$ is the number of processors

Variation 3: Tapering

• Idea: the chunk size, $K_i$ is a function of not only the remaining work, but also the task cost variance
  • variance is estimated using history information
  • high variance $\Rightarrow$ small chunk size should be used
  • low variance $\Rightarrow$ larger chunks OK

  • Gives analysis (based on workload distribution)
  • Also gives experimental results -- tapering always works at least as well as GSS, although difference is often small
Variation 4: Weighted Factoring

- If hardware is heterogeneous (some processors faster than others)
- Idea: similar to self-scheduling, but divide task cost by computational power of requesting node

- Also useful for shared resource clusters, e.g., built using all the machines in a building
  - as with Tapering, historical information is used to predict future speed
  - “speed” may depend on the other loads currently on a given processor

- See Hummel, Schmit, Uma, and Wein, SPAA ‘96
  - includes experimental data and analysis
When is Self-Scheduling a Good Idea?

Useful when:

- A batch (or set) of tasks without dependencies
  - can also be used with dependencies, but most analysis has only been done for task sets without dependencies
- The cost of each task is unknown
- Locality is not important
- Shared memory machine, or at least number of processors is small – centralization is OK
Distributed Task Queues

• The obvious extension of task queue to distributed memory is:
  • a distributed task queue (or “bag”)
  • Doesn’t appear as explicit data structure in message-passing
  • Idle processors can “pull” work, or busy processors “push” work

• When are these a good idea?
  • Distributed memory multiprocessors
  • Or, shared memory with significant synchronization overhead or very small tasks which lead to frequent task queue accesses
  • Locality is not (very) important
  • Tasks that are:
    • known in advance, e.g., a bag of independent ones
    • dependencies exist, i.e., being computed on the fly
  • The costs of tasks is not known in advance
Distributed Dynamic Load Balancing

- Dynamic load balancing algorithms go by other names:
  - Work stealing, work crews, …
- Basic idea, when applied to tree search:
  - Each processor performs search on disjoint part of tree
  - When finished, get work from a processor that is still busy
  - Requires asynchronous communication

![Diagram showing the process of distributed dynamic load balancing.]

1. Service pending messages
2. Select a processor and request work
3. Do fixed amount of work
4. Service pending messages
5. Got work
6. No work found
7. Got work
8. Idle
How to Select a Donor Processor

• Three basic techniques:
  1. Asynchronous round robin
     • Each processor $k$, keeps a variable “target$_k$”
     • When a processor runs out of work, requests work from target$_k$
     • Set target$_k$ = (target$_k$ +1) mod procs
  2. Global round robin
     • Proc 0 keeps a single variable “target”
     • When a processor needs work, gets target, requests work from target
     • Proc 0 sets target = (target + 1) mod procs
  3. Random polling/stealing
     • When a processor needs work, select a random processor and request work from it

• Repeat if no work is found
How to Split Work

• First parameter is number of tasks to split off
  • Related to the self-scheduling variations, but total number of tasks is now unknown

• Second question is which one(s)
  • Send tasks near the bottom of the stack (oldest)
  • Execute from the top (most recent)
  • May be able to do better with information about task costs
Theoretical Results (1)

Main result: A simple randomized algorithm is optimal with high probability

- Karp and Zhang [88] show this for a tree of unit cost (equal size) tasks
  - Parent must be done before children
  - Tree unfolds at runtime
  - Task number/priorities not known a priori
  - Children “pushed” to random processors

- Show this for independent, equal sized tasks
  - “Throw balls into random bins”: $\Theta \left( \frac{\log n}{\log \log n} \right)$ in largest bin
  - Throw $d$ times and pick the smallest bin: $\frac{\log \log n}{\log d} = \Theta (1)$ [Azar]
  - Extension to parallel throwing [Adler et all 95]
  - Shows $p \log p$ tasks leads to “good” balance
Theoretical Results (2)

Main result: A simple randomized algorithm is optimal with high probability

- Blumofe and Leiserson [94] show this for a fixed task tree of variable cost tasks
  - their algorithm uses task pulling (stealing) instead of pushing, which is good for locality
  - i.e., when a processor becomes idle, it steals from a random processor
  - also have bounds on the total memory required
- Chakrabarti et al [94] show this for a dynamic tree of variable cost tasks
  - uses randomized pushing of tasks instead of pulling: worse for locality, but faster balancing in practice
  - works for branch and bound, i.e. tree structure can depend on execution order
Distributed Task Queue References

- Introduction to Parallel Computing by Kumar et al (text)
- Multipol library (See C.-P. Wen, UCB PhD, 1996.)
  - Part of Multipol (www.cs.berkeley.edu/projects/multipol)
  - Try to push tasks with high ratio of cost to compute/cost to push
    - Ex: for matmul, ratio = $2n^3 \text{cost(flop)} / 2n^2 \text{cost(send a word)}$
- Goldstein, Rogers, Grunwald, and others (independent work) have all shown
  - advantages of integrating into the language framework
  - very lightweight thread creation
- CILK (Leiserson et al) (supertech.lcs.mit.edu/cilk)
  - Space bound on task stealing
- X10 from IBM
Diffusion-Based Load Balancing

• In the randomized schemes, the machine is treated as fully-connected.
• Diffusion-based load balancing takes topology into account
  • Locality properties better than prior work
  • Load balancing somewhat slower than randomized
  • Cost of tasks must be known at creation time
  • No dependencies between tasks
Diffusion-based load balancing

- The machine is modeled as a graph
- At each step, we compute the weight of task remaining on each processor
  - This is simply the number if they are unit cost tasks
- Each processor compares its weight with its neighbors and performs some averaging
  - Analysis using Markov chains
- See Ghosh et al, SPAA96 for a second order diffusive load balancing algorithm
  - takes into account amount of work sent last time
  - avoids some oscillation of first order schemes
- Note: locality is still not a major concern, although balancing with neighbors may be better than random
Load Balancing Summary

- Techniques so far deal with
  - Unpredictable loads → online algorithms
- Two scenarios
  - Fixed set of tasks with unknown costs: self-scheduling
  - Dynamically unfolding set of tasks: work stealing
- Little concern over locality, except
  - Stealing (pulling) is better than pushing (sending work away)
  - When you steal, steal the oldest tasks which are likely to generate a lot of work
- What if locality is very important?
  - Load balancing based on data partitioning
  - If equal amounts of work per grid point, divide grid points evenly
  - This is what you’re doing in HW3
  - Optimize locality by minimizing surface area (perimeter in 2D) where communication occurs; minimize aspect ratio of blocks
- What if we know the task graph structure in advance?
- More algorithms for these other scenarios
Project Discussion
Project outline

• Select an application or algorithm (or set of algorithms) Choose something you are personally interested in that has potential to need more compute power
  • Machine learning (done for GPUs in CS267)
  • Algorithm from “physics” game, e.g., collision detection
  • Sorting algorithms
  • Parsing html (ongoing project)
  • Speech or image processing algorithm
  • What are games, medicine, SecondLife, etc. limited by?

• Select a machine (or multiple machines)
  • Preferably multicore/multisocket SMP, GPU, Cell (>= 8 cores)

• Proposal (due Fri, Oct 19): Describe problem, machine, predict bottlenecks and likely parallelism (~1-page)
Project continued

Project steps:
• Implement a parallel algorithm on machine(s)
• Analyze performance (!); develop performance model
  • Serial work
  • Critical path in task graph (can’t go faster)
  • Memory bandwidth, arithmetic performance, etc.
• Tune performance
• We will have preliminary feedback sessions in class!
• Write up results with graphs, models, etc.
  • Length is not important, but think of 8-10 pages
• Note: what is the question you will attempt to answer?
  • X machine is better than Y for this algorithm (and why)
  • This algorithm will scale linearly on X (for how many procs?)
  • This algorithm is entirely limited by memory bandwidth