Parallel Programming
Todd C. Mowry
CS 740
October 16 & 18, 2000

Topics
• Motivating Examples
• Parallel Programming for High Performance
• Impact of the Programming Model
• Case Studies
  - Ocean simulation
  - Barnes-Hut N-body simulation

Motivating Problems

Simulating Ocean Currents
• Regular structure, scientific computing

Simulating the Evolution of Galaxies
• Irregular structure, scientific computing

Rendering Scenes by Ray Tracing
• Irregular structure, computer graphics
• Not discussed here (read in book)

Simulating Ocean Currents
• Model as two-dimensional grids
• Discretize in space and time
  - finer spatial and temporal resolution => greater accuracy
• Many different computations per time step
  - set up and solve equations
• Concurrency across and within grid computations

Simulating Galaxy Evolution
• Simulate the interactions of many stars evolving over time
• Computing forces is expensive
• $O(n^2)$ brute force approach
• Hierarchical Methods take advantage of force law: $G \frac{m_1 m_2}{r^2}$

• Many time-steps, plenty of concurrency across stars within one
Rendering Scenes by Ray Tracing

- Shoot rays into scene through pixels in image plane
- Follow their paths
  - they bounce around as they strike objects
  - they generate new rays: ray tree per input ray
- Result is color and opacity for that pixel
- Parallelism across rays

All case studies have abundant concurrency

Parallel Programming Task

Break up computation into tasks
- assign tasks to processors

Break up data into chunks
- assign chunks to memories

Introduce synchronization for:
- mutual exclusion
- event ordering

Steps in Creating a Parallel Program

4 steps: Decomposition, Assignment, Orchestration, Mapping
- Done by programmer or system software (compiler, runtime, ...)
- Issues are the same, so assume programmer does it all explicitly

Partitioning for Performance

Balancing the workload and reducing wait time at synch points
Reducing inherent communication
Reducing extra work

Even these algorithmic issues trade off:
- Minimize comm. => run on 1 processor => extreme load imbalance
- Maximize load balance => random assignment of tiny tasks => no control over communication
- Good partition may imply extra work to compute or manage it

Goal is to compromise
- Fortunately, often not difficult in practice
Load Balance and Synch Wait Time

Limit on speedup: $\frac{\text{Speedup}_{\text{problem}(p)}}{\text{Max Work on any Processor}}$

- Work includes data access and other costs
- Not just equal work, but must be busy at same time

Four parts to load balance and reducing synch wait time:
1. Identify enough concurrency
2. Decide how to manage it
3. Determine the granularity at which to exploit it
4. Reduce serialization and cost of synchronization

Deciding How to Manage Concurrency

Static versus Dynamic techniques

Static:
- Algorithmic assignment based on input; won’t change
- Low runtime overhead
- Computation must be predictable
- Preferable when applicable (except in multiprogrammed/heterogeneous environment)

Dynamic:
- Adapt at runtime to balance load
- Can increase communication and reduce locality
- Can increase task management overheads

Dynamic Assignment

Profile-based (semi-static):
- Profile work distribution at runtime, and repartition dynamically
- Applicable in many computations, e.g. Barnes-Hut, some graphics

Dynamic Tasking:
- Deal with unpredictability in program or environment (e.g. Raytrace)
  - computation, communication, and memory system interactions
  - multiprogramming and heterogeneity
  - used by runtime systems and OS too
- Pool of tasks; take and add tasks until done
- E.g. “self-scheduling” of loop iterations (shared loop counter)

Dynamic Tasking with Task Queues

Centralized versus distributed queues

Task stealing with distributed queues
- Can compromise comm and locality, and increase synchronization
- Whom to steal from, how many tasks to steal, ...
- Termination detection
- Maximum imbalance related to size of task

![Dynamic Tasking with Task Queues Diagram]
Determining Task Granularity

Task granularity: amount of work associated with a task

General rule:
- Coarse-grained => often less load balance
- Fine-grained => more overhead; often more communication and contention

Communication and contention actually affected by assignment, not size
- Overhead by size itself too, particularly with task queues

Reducing Serialization

Careful about assignment and orchestration (including scheduling)

Event synchronization
- Reduce use of conservative synchronization
  - e.g. point-to-point instead of barriers, or granularity of pt-to-pt
  - But fine-grained synch more difficult to program, more synch ops.

Mutual exclusion
- Separate locks for separate data
  - e.g. locking records in a database: lock per process, record, or field
  - lock per task in task queue, not per queue
  - finer grain => less contention/serialization, more space, less reuse
- Smaller, less frequent critical sections
  - don't do reading/testing in critical section, only modification
  - e.g. searching for task to dequeue in task queue, building tree
  - Stagger critical sections in time

Reducing Inherent Communication

Communication is expensive!
Measure: communication to computation ratio
Focus here on inherent communication
- Determined by assignment of tasks to processes
- Later see that actual communication can be greater

Assign tasks that access same data to same process
Solving communication and load balance NP-hard in general case
But simple heuristic solutions work well in practice
- Applications have structure!

Domain Decomposition

Works well for scientific, engineering, graphics, ... applications
Exploits local-biased nature of physical problems
- Information requirements often short-range
- Or long-range but fall off with distance

Simple example: nearest-neighbor grid computation

Perimeter to Area comm-to-comp ratio (area to volume in 3D)
- Depends on n,p: decreases with n, increases with p
Reducing Extra Work

Common sources of extra work:
- Computing a good partition
  - e.g. partitioning in Barnes-Hut or sparse matrix
- Using redundant computation to avoid communication
- Task, data and process management overhead
  - applications, languages, runtime systems, OS
- Imposing structure on communication
  - coalescing messages, allowing effective naming

Architectural Implications:
- Reduce need by making communication and orchestration efficient

Speedup ≤ \frac{\text{Sequential Work}}{\text{Max (Work + Synch Wait Time + Comm Cost + Extra Work)}}

Summary of Tradeoffs

Different goals often have conflicting demands
- Load Balance
  - fine-grain tasks
  - random or dynamic assignment
- Communication
  - usually coarse grain tasks
  - decompose to obtain locality: not random/dynamic
- Extra Work
  - coarse grain tasks
  - simple assignment
- Communication Cost:
  - big transfers: amortize overhead and latency
  - small transfers: reduce contention

Impact of Programming Model

Example: LocusRoute (standard cell router)

while (route_density_improvement > threshold)
{
  for (i = 1 to num_wires) do
    - rip old wire route out
    - explore new routes
    - place wire using best new route
}

Shared-Memory Implementation

Shared memory algorithm:
- Divide cost-array into regions (assign regions to PEs)
- Assign wires to PEs based on the region in which center lies
- Do load balancing using stealing when local queue empty

Good points:
- Good load balancing
- Mostly local accesses
- High cache-hit ratio
Message-Passing Implementations

Solution-1:
- Distribute wires and cost-array regions as in sh-mem implementation
- Big overhead when wire-path crosses to remote region
  - send computation to remote PE, or
  - send messages to access remote data

Solution-2:
- Wires distributed as in sh-mem implementation
- Each PE has copy of full cost array
  - one owned region, plus potentially stale copy of others
  - send frequent updates so that copies not too stale
- Consequences:
  - waste of memory in replication
  - stale data => poorer quality results or more iterations

=> In either case, lots of thinking needed on the programmer’s part

Case Studies

Simulating Ocean Currents
- Regular structure, scientific computing

Simulating the Evolution of Galaxies
- Irregular structure, scientific computing

Case 1: Simulating Ocean Currents
- Model as two-dimensional grids
- Discretize in space and time
  - finer spatial and temporal resolution => greater accuracy
- Many different computations per time step
  - set up and solve equations
- Concurrency across and within grid computations

Time Step in Ocean Simulation
Partitioning

Exploit data parallelism
• Function parallelism only to reduce synchronization
Static partitioning within a grid computation
• Block versus strip
  - inherent communication versus spatial locality in communication
  - Load imbalance due to border elements and number of boundaries
Solver has greater overheads than other computations

Two Static Partitioning Schemes

Strip
Block

Which approach is better?

Impact of Memory Locality

algorithmic = perfect memory system; No Locality = dynamic assignment of columns to processors; Locality = static subgrid assignment (infinite caches)

Impact of Line Size & Data Distribution

no-alloc = round-robin page allocation; otherwise, data assigned to local memory. L = cache line size
Case 2: Simulating Galaxy Evolution

- Simulate the interactions of many stars evolving over time
- Computing forces is expensive
- \( O(n^2) \) brute force approach
- Hierarchical Methods take advantage of force law: \( G \frac{m_1 m_2}{r^2} \)

Hierarchical Methods:
- \( n \) time-steps, \( n \) concurrency across stars within one

Barnes-Hut

Locality Goal:
- particles close together in space should be on same processor

Difficulties:
- nonuniform, dynamically changing

Application Structure

- Main data structures: array of bodies, of cells, and of pointers to them
  - Each body/cell has several fields: mass, position, pointers to others
  - Pointers are assigned to processes

Partitioning

Decomposition: bodies in most phases, cells in computing moments

Challenges for assignment:
- Nonuniform body distribution => work and comm. nonuniform
  - Cannot assign by inspection
- Distribution changes dynamically across time-steps
  - Cannot assign statically
- Information needs fall off with distance from body
  - Partitions should be spatially contiguous for locality
- Different phases have different work distributions across bodies
  - No single assignment ideal for all
  - Focus on force calculation phase
- Communication needs naturally fine-grained and irregular
Load Balancing

- Equal particles = equal work.
  - Solution: Assign costs to particles based on the work they do

- Work unknown and changes with time-steps
  - Insight: System evolves slowly
  - Solution: Count work per particle, and use as cost for next time-step.

Powerful technique for evolving physical systems

A Partitioning Approach: ORB

Orthogonal Recursive Bisection:
- Recursively bisect space into subspaces with equal work
  - Work is associated with bodies, as before
  - Continue until one partition per processor

- High overhead for large number of processors

Another Approach: Costzones

Insight: Tree already contains an encoding of spatial locality.

- Costzones is low-overhead and very easy to program

Barnes-Hut Performance

- Speedups on simulated multiprocessor
  - Extra work in ORB is the key difference