Perspective on Parallel Programming

CS 258, Spring 99
David E. Culler
Computer Science Division
U.C. Berkeley

Outline for Today

• Motivating Problems (application case studies)
• Process of creating a parallel program
• What a simple parallel program looks like
  – three major programming models
  – What primitives must a system support?
• Later: Performance issues and architectural interactions

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
• Static and regular

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:
    \[ \frac{G m_1 m_2}{r^2} \]
• Many time-steps, plenty of concurrency across stars within one step

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
• How much concurrency in these examples?

Creating a Parallel Program

• Pieces of the job:
  – Identify work that can be done in parallel
  – Work includes computation, data access and I/O
  – Partition work and perhaps data among processes
  – Manage data access, communication and synchronization
**Definitions**

- **Task:**
  - Arbitrary piece of work in parallel computation
  - Executed sequentially; concurrency is only across tasks
  - E.g. a particle/cell in Barnes-Hut, a ray or ray group in Raytrace
  - Fine-grained versus coarse-grained tasks

- **Process (thread):**
  - Abstract entity that performs the tasks assigned to processes
  - Processes communicate and synchronize to perform their tasks

- **Processor:**
  - Physical engine on which process executes
  - Processes virtualize machine to programmer
  - write program in terms of processes, then map to processors

**4 Steps in Creating a Parallel Program**

- Decomposition of computation in tasks
- Assignment of tasks to processes
- Orchestration of data access, comm, synch.
- Mapping processes to processors

**Decomposition**

- Identify concurrency and decide level at which to exploit it
- Break up computation into tasks to be divided among processes
  - Tasks may become available dynamically
  - No. of available tasks may vary with time
- Goal: Enough tasks to keep processes busy, but not too many
  - Number of tasks available at a time is upper bound on achievable speedup

**Limited Concurrency: Amdahl’s Law**

- Most fundamental limitation on parallel speedup
- If fraction $s$ of seq execution is inherently serial, speedup $\leq \frac{1}{s}$
- Example: 2-phase calculation
  - sweep over n-by-n grid and do some independent computation
  - sweep again and add each value to global sum
- Time for first phase $= \frac{n^2}{p}$
- Second phase serialized at global variable, so time $= n^2$
- Speedup $= \frac{\frac{n^2}{p} + n^2}{\frac{n^2}{p}}$ or at most 2
- Trick: divide second phase into two
  - accumulate into private sum during sweep
  - add per-process private sum into global sum
- Parallel time is $\frac{n^2}{p} + \frac{n^2}{p} + p$, and speedup at best $\frac{2n^2}{2n^2 + p^2}$

**Understanding Amdahl’s Law**

- Area under curve is total work done, or time with 1 processor
- Horizontal extent is lower bound on time (infinite processors)
- Speedup is the ratio: $\frac{\sum_{k=1}^{\infty} f_k}{\sum_{k=1}^{\infty} \frac{f_k}{P}}$ base case: $\frac{1}{s + \frac{1}{1-s}}$
- Amdahl’s law applies to any overhead, not just limited concurrency
Assignment

- Specify mechanism to divide work up among processes
  - E.g. which process computes forces on which stars, or which rays
  - Balance workload, reduce communication and management cost
- Structured approaches usually work well
  - Code inspection (parallel loops) or understanding of application
  - Well-known heuristics
  - Static versus dynamic assignment
- As programmers, we worry about partitioning first
  - Usually independent of architecture or prog model
  - But cost and complexity of using primitives may affect decisions

Orchestration

- Name data
- Structuring communication
- Synchronization
- Organizing data structures and scheduling tasks temporally

• Goals
  - Reduce cost of communication and synch.
  - Preserve locality of data reference
  - Schedule tasks to satisfy dependences early
  - Reduce overhead of parallelism management

• Choices depend on Prog. Model., comm. abstraction, efficiency of primitives
• Architects should provide appropriate primitives efficiently

Mapping

- Two aspects:
  - Which process runs on which particular processor?
  - mapping to a network topology
  - Will multiple processes run on same processor?
- space-sharing
  - Machine divided into subsets, only one app at a time in a subset
  - Processes can be pinned to processors, or left to OS
- System allocation
- Real world
  - User specifies desires in some aspects, system handles some
  - Usually adopt the view: process <-> processor

Parallelizing Computation vs. Data

- Computation is decomposed and assigned (partitioned)
- Partitioning Data is often a natural view too
  - Computation follows data: owner computes
  - Grid example; data mining;
- Distinction between comp. and data stronger in many applications
  - Barnes-Hut
  - Raytrace

Architect’s Perspective

- What can be addressed by better hardware design?
- What is fundamentally a programming issue?

High-level Goals

- High performance (speedup over sequential program)
- But low resource usage and development effort
- Implications for algorithm designers and architects?
What Parallel Programs Look Like

Example: iterative equation solver
- Simplified version of a piece of Ocean simulation
- Illustrate program in low-level parallel language
  - C-like pseudocode with simple extensions for parallelism
  - State of most real parallel programming today

\[
A_{i,j} = 0.2 \times (A_{i,j} + A_{i,j-1} + A_{i-1,j} + A_{i,j+1} + A_{i+1,j})
\]

Expression for updating each interior point:

Grid Solver
- Gauss-Seidel (near-neighbor) sweeps to convergence
  - Interior n-by-n points of (n+2)-by-(n+2) updated in each sweep
  - Updates done in-place in grid
  - Difference from previous value computed
  - Accumulate partial diffs into global diff at end of every sweep
  - Check if has converged
    - to within a tolerance parameter

Sequential Version

\[
\text{Procedure Solve} (A)
\]

Exploit Application Knowledge
- Reorder grid traversal: red-black ordering
  - Different ordering of updates: may converge quicker or slower
  - Red sweep and black sweep are each fully parallel
  - Global synch between them (convenient but conservative)
  - Ocean uses red-black
  - We use simpler, asynchronous one to illustrate
    - No red-black, simply ignore dependences within sweep
    - Parallel program nondeterministic

Decomposition
- Simple way to identify concurrency is to look at loop iterations
  - Dependence analysis: if not enough concurrency, then look further
- Not much concurrency here at this level (all loops sequential)
- Examine fundamental dependences
  - Concurrency O(n) along anti-diagonals, serialization O(n) along diag.
Decomposition

• Decomposition into elements: degree of concurrency \( n^2 \)
• Decomposition into rows? Degree?
• for_all assignment ??

Assignment

• Static assignment: decomposition into rows
  - block assignment of rows: Row \( i \) is assigned to process \( i \) + \( p \) \( \mod \) \( n \) for simplicity here
  - assume that \( n \) is exactly divisible by \( n \) processes

• Dynamic assignment
  - get a row index, work on the row, get a new row, ...
  - What is the mechanism?
  - Concurrency? Volume of Communication?

Data Parallel Solver

Shared Address Space Solver

Single Program Multiple Data (SPMD)

Generating Threads

Assignment Mechanism

• procedure Solve(A)
• BEGIN
• \( \text{for all } i \) \( \text{do} \)
• \( \text{mydiff}[i,j] = \text{abs}(A[i,j] - \text{temp}) \)
• \( \text{temp} = A[i,j] \)
• \( \text{for all } j \) \( \text{do} \)
• \( \text{diff += abs}(A[i,j] - \text{temp}) \)
• \( \text{end for_all} \)
• \( \text{if } \text{diff} / (n \times n) < \text{TOL} \) then \( \text{done} = 1 \)
• \( \text{end while} \)
• \( \text{end procedure} \)

• procedure Solve(A)
• BEGIN
• \( \text{for all } i \) \( \text{do} \)
• \( \text{mydiff}[i,j] = \text{abs}(A[i,j] - \text{temp}) \)
• \( \text{temp} = A[i,j] \)
• \( \text{for all } j \) \( \text{do} \)
• \( \text{diff += abs}(A[i,j] - \text{temp}) \)
• \( \text{end for_all} \)
• \( \text{if } \text{diff} / (n \times n) < \text{TOL} \) then \( \text{done} = 1 \)
• \( \text{end while} \)
• \( \text{end procedure} \)
**SAS Program**

- SPMD: not lockstep. Not necessarily same instructions
- Assignment controlled by values of variables used as loop bounds
  - unique pid per process, used to control assignment
- done condition evaluated redundantly by all
- Code that does the update identical to sequential program
  - each process has private mydiff variable
- Most interesting special operations are for synchronization
  - accumulations into shared diff have to be mutually exclusive
  - why the need for all the barriers?
- Good global reduction?
  - Utility of this parallel accumulate???

**Global Event Synchronization**

- BARRIER(nprocs): wait here till nprocs processes get here
  - Built using lower level primitives
  - Global sum example: wait for all to accumulate before using sum
  - Often used to separate phases of computation
- Process P_1
- Process P_2
- Process P_nprocs
  - Barrier (name, nprocs) Barrier (name, nprocs) Barrier (name, nprocs)
  - solve eqn system solve eqn system solve eqn system
  - apply results apply results apply results
  - Barrier (name, nprocs) Barrier (name, nprocs) Barrier (name, nprocs)
  - Conservative form of preserving dependences, but easy to use
- WAIT_FOR_END (nprocs-1)

**Mutual Exclusion**

- Why is it needed?
- Provided by LOCK-UNLOCK around critical section
  - Set of operations we want to execute atomically
  - Implementation of LOCK/UNLOCK must guarantee mutual excl.
- Serialization?
  - Contention?
  - Non-local accesses in critical section?
  - use private mydiff for partial accumulation!

**Group Event Synchronization**

- Subset of processes involved
  - Can use flags or barriers (involving only the subset)
  - Concept of producers and consumers
- Major types:
  - Single-producer, multiple-consumer
  - Multiple-producer, single-consumer
  - Multiple-producer, single-consumer

**Pt-to-pt Event Synch (Not Used Here)**

- One process notifies another of an event so it can proceed
  - Common example: producer-consumer (bounded buffer)
  - Concurrent programming on uniprocessor: semaphores
  - Shared address space parallel programs: semaphores, or use ordinary variables as flags

**Message Passing Grid Solver**

- Cannot declare A to be global shared array
  - compose it logically from per-process private arrays
  - usually allocated in accordance with the assignment of work
  - process assigned a set of rows allocates them locally
- Transfers of entire rows between traversals
  - Structurally similar to SPMD SAS
- Orchestration different
  - data structures and data access/naming
  - communication
  - synchronization
- Ghost rows
Data Layout and Orchestration

- Data partition allocated per processor
- Add ghost rows to hold boundary data
- Send edges to neighbors
- Receive into ghost rows
- Compute as in sequential program

Notes on Message Passing Program

- Use of ghost rows
- Receive does not transfer data, send does
  - Unlike SAS which is usually receiver-initiated (load fetches data)
- Communication done at beginning of iteration, so no asynchrony
- Communication in whole rows, not element at a time
- Core similar, but indices/bounds in local rather than global space
- Synchronization through sends and receives
  - Update of global diff and event sync for done condition
  - Could implement locks and barriers with messages
- Can use REDUCE and BROADCAST library calls to simplify

Send and Receive Alternatives

Can extend functionality: stride, scatter, gather, groups

Semantic flavors: based on when control is returned

- Affect when data structures or buffers can be reused at either end
  - Synchronous: Provides built-in synchronization
  - Asynchronous: Requires explicit synchronization

Orchestration: Summary

- Shared address space
  - Shared and private data explicitly separate
  - No correctness need for data distribution
  - Synchronization via atomic operations on shared data
  - Message passing
    - Communication explicit and distinct from data communication

Correctness in Grid Solver Program

<table>
<thead>
<tr>
<th></th>
<th>SAS</th>
<th>Msg-Passing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explicit global data structure?</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Assignment indep of data layout?</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Communication</td>
<td>Implicit</td>
<td>Explicit</td>
</tr>
<tr>
<td>Synchronization</td>
<td>Explicit</td>
<td>Implicit</td>
</tr>
<tr>
<td>Explicit replication of border rows?</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

- Decomposition and Assignment similar in SAS and message-passing
- Orchestration is different
  - Data structures, data access/naming, communication, synchronization
  - Performance?