Announcements

- Assignment 1 is now due on September 26th
- Assignments 4 and 5 will be combined into one three-week assignment
- Schedule of lectures, assignments will shift down by a week
- Class account information will be given out on September 26th
- Please subscribe to the class mailing list
  - Information on the course home page

Outline

- Last lecture
  - central and mainstream nature of parallel computing
    - application demands, technology and architecture trends, economics
  - convergence in parallel architectures
    - shared address-space architectures

- This lecture
  - convergence in parallel architectures (cont’d)
    - message-passing architectures
    - Others: data-parallel, data flow, systolic
  - parallel programs
    - key steps: decomposition, assignment, orchestration, mapping
    - case studies: Ocean, Raytrace

(Review) Evolution of Architectural Models

- Rationale
  - historically: machines tailored to programming models
    - architecture = prog. model + comm. abstraction + machine organization
  - now: separation of programming models and architectures
    - tracing the evolution helps identify core concepts, understand convergence

- Five architectures of interest
  - dominant: shared address space, message passing
  - others: data parallel, dataflow, systolic arrays

- Last lecture: Shared address space architectures
  - Multiple processes in a single address space
    - Communication via memory loads and stores
    - Architectures reflect this natural extension over uniprocessors
      - Multiple processors on memory bus
      - More scalable architectures trap remote load/store requests and realize them using explicit communication operations
(Review) Message Passing Architectures

- Programming model
  - directly access only private address space (local memory), communicate via explicit messages (send/receive)
  - in simplest form, achieves pair-wise synchronization
  - model is removed from basic hardware operations
    - library or OS intervention for copying, buffer management, protection

Example of a Message Passing Machine: IBM SP-2

- Made out of essentially complete RS/6000 workstations
- Network interface integrated in I/O bus (bw limited by I/O bus)

Evolution of Message-Passing Machines

- Early machines: FIFO on each link
  - hw close to programming model
    - synchronous (blocking) operations
  - replaced by DMA
    - enables non-blocking ops
    - buffered by software at destination

- Now: diminishing role of topology
  - topology important for store-and-forward routing
  - introduction of pipelined (cut-through) routing made it less so
    - Virtual cut through, wormhole routing
  - cost is in node-network interface
Toward Architectural Convergence

- Evolution and role of software have blurred boundary between programming models
  - send/recv on shared address space (SAS) architectures: buffers
  - global address space on message passing (MP) architectures
    - hashing and page-based (or finer-grained) shared virtual memory
  - Convergence in hardware organizations as well
    - tighter NI integration even for MP
    - hardware SAS passes messages at lower level
    - even clusters of workstations/SMPs are parallel systems
      - emergence of fast system area networks (SAN)

- Programming models distinct, but organizations converging
  - nodes connected by general network and communication assists
  - implementations also converging, at least in high-end machines

Other Architectures (1): Data Parallel

- Programming model
  - operations performed in parallel on each element of data structure
  - logically single thread of control, performs sequential or parallel steps
    - conceptually, a processor associated with each data element

- Architectural model
  - array of many simple cheap processors, each with little memory
  - a control processor issues instructions
  - specialized and general communication, cheap global synchronization

- Original motivations
  - matches simple differential equation solvers
  - centralize high cost of instruction fetch/sequencing

Applications of Data Parallelism

- Each PE contains an employee record with his/her salary
  
  ```
  if salary > 100K then
      salary = salary * 1.05
  else
      salary = salary * 1.10
  ```

- Other examples
  - finite differences, linear algebra
  - document searching, graphics, image processing

- Popular architecture in the late 1980s and early 1990s:
  - Thinking Machines CM-1, CM-2 (and CM-5)
  - Maspar MP-1 and MP-2

Evolution and Convergence

- Rigid control structure popular in the 1960s
  - cost savings of centralized sequencer high
  - Flynn taxonomy: SIMD (SISD: uniprocessor, MIMD: multiprocessor)

- Replaced by vector machines in mid-70s
  - more flexible w.r.t. memory layout and easier to manage

- Revived in mid-80s
  - when only 32-bit datapath slices would fit on chip

- Other reasons for demise
  - simple, regular applications have good locality, can do well anyway
    - On any programming model/architecture
    - hardwiring data parallelism in architecture limits applications

- Lasting contributions
  - programming model converges with SPMD (single program multiple data)
  - need for fast global synchronization, structured global address space
Other Architectures (2): Dataflow

- Represent computation as a graph of essential dependences
  - logical processor at each node, activated by availability of operands
    - message (tokens) carrying tag of next instruction sent to next processor
    - tag compared with others in matching store; match fires execution

Dataflow graph

Network

Other Architectures (3): Systolic

- Replace single processor with array of regular processing elements
  - orchestrate data flow for high throughput with less memory access

Systolic Architectures (contd.)

- Differences from other organizations
  - pipelining: nonlinear structure, multi-directional data flow,
    each PE may have (small) local instruction and data memory
  - SIMD: each PE may do something different

- Original motivations
  - VLSI enables inexpensive special-purpose chips/modules
  - represent algorithms directly by chips connected in regular pattern

Evolution and Convergence

- Characteristics
  - ability to name operations, synchronization, dynamic scheduling

- Problems
  - operations have locality: useful to group together
  - complexity of matching store and memory units
  - too much parallelism (?)

- Converged to use conventional processors and memory
  - differences
    - support for large, dynamic set of threads to map to processors
    - integration of communication and thread creation
  - eventually, separation of programming model from hardware

- Lasting contributions:
  - tightly integrated communication and fine-grained synchronization
  - remains a useful concept for software (compilers etc.)

Example realization: iWARP

- used quite general processors: variety of algorithms on same hardware
- but dedicated interconnect channels: register-to-register data transfer

Specialized, and ran into same problems as SIMD

- general purpose systems work well for same algorithms (locality etc.)
- current day manifestation: Embedded digital signal processors (DSPs)
Convergence: Generic Parallel Architecture

- A generic modern multiprocessor

![Diagram of network and communication assist]

- Node: processor(s), memory system, plus communication assist
  - network interface and communication controller
- Scalable network

Convergence allows lots of innovation, now within framework
  - integration of assist with node, what operations, how efficiently...

Outline

- Last lecture
  - central and mainstream nature of parallel computing
  - application demands, technology and architecture trends, economics
  - convergence in parallel architectures
  - shared address-space architectures

- This lecture
  - convergence in parallel architectures (cont’d)
    - message-passing architectures
    - Others: data parallel, data flow, systolic
  - parallel programs
    - key steps: decomposition, assignment, orchestration, mapping
    - case studies: Ocean, Raytrace

[Culler/Singh/Gupta: Chapters 1–2; Andrews: Chapter 1]

Why Bother with Programs?

- They’re what run on the machines
  - helps make design decisions
  - helps evaluate systems tradeoffs
- Led to the key advances in uniprocessor architectures
  - caches and instruction set design
- More important in multiprocessors
  - additional degrees of freedom
  - greater penalties for mismatch between program and architecture
- Important for
  - algorithm designers: designing algorithms to run well on real systems
  - programmers: understanding key issues and obtaining best performance
  - OS and compiler writers: deciding which optimizations are worthwhile
  - architects: understanding workloads, interactions, degrees of freedom

Creating a Parallel Program

- Assumption: Sequential algorithm is given
  - sometimes need very different algorithm, but beyond scope of this course
- Key steps:
  - identify work that can be done in parallel
  - partition work and perhaps data among processes
  - manage data access, communication and synchronization
  - note: work includes computation, data access and I/O
- Primary goal: Speedup
  \[
  \text{speedup}(p) = \left( \frac{\text{performance}(1)}{\text{performance}(p)} \right) \left( \frac{\text{time}(1)}{\text{time}(p)} \right)
  \]
- Secondary goals: Low programming effort, low resource needs
Steps in Creating a Parallel Program

Some Important Concepts

- **Task**
  - arbitrary piece of work in a parallel computation
  - executed sequentially: concurrency is only across tasks
  - fine-grained versus coarse-grained tasks
    (with respect to communication/data access requirements)

- **Process (thread)**
  - abstract entity that performs one or more tasks
  - processes communicate and synchronize to perform tasks

- **Processor**
  - physical engine on which a process executes

- **Parallel program**
  - Computation ➔ Tasks ➔ Processes ➔ Processors

Step 1: Decomposition

- Break up computation into tasks to be divided among processes
  - identify concurrency: the work that can be done in parallel
  - decide appropriate level at which to exploit concurrency
    - too much: high overheads of management
    - too few: variation in work performed across tasks
  - in general: need to respect precedence relationships between tasks

- Characteristics
  - task creation: static versus dynamic
  - task granularity: uniform versus non-uniform,
    fine-grained versus coarse-grained
  - concurrency: number of available tasks

- 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

- Fundamental limitation on parallel speedup
  - if fraction $s$ of sequential execution is inherently serial
    $$\text{speedup} \leq \frac{1}{s}$$
  - assuming ideal speedup for the non-serial part
    $$\text{speedup} = \frac{1}{\frac{1-s}{P} + s} \leq \frac{1}{s}$$
  - assumes that the problem size remains the same
    - the serial part may contribute different fractions at different problem sizes
    - limits on speedup not quite as restrictive!
Limited Concurrency: Example

2-phase calculation
- phase 1: sweep over $N$-by-$N$ grid and do some independent computation
- phase 2: sweep again and add each value to global sum
- times for each phase (assuming $P$ processors)
  - first phase: $N^2/P$ (each grid point can be done in parallel)
  - second phase: $N^2$ (since serialized at the global variable)

\[
speedup = \frac{2N^2}{\frac{N^2}{P} + N^2} \leq 2
\]

- workaround: reduce serial part by splitting phase 2 into two subparts
  - first accumulate partial sums on a per-processor basis ($N^2/P$)
  - then, add these sums ($P$)

\[
speedup = \frac{2N^2}{\frac{N^2}{P} + \frac{N^2}{P} + P} = \frac{P}{1 + \frac{P^2}{N^2}} \leq P
\]

Concurrency Profiles

- area under curve is total work done (time with 1 processor)
- horizontal extent is lower bound on time (infinite processors)
\[
speedup(p) \leq \frac{\sum_{i=1}^{k} f_i k}{\sum_{i=1}^{k} f_i} \leq \frac{1}{\frac{k}{p} + s}
\]

- Amdahl’s law applies to any overhead, not just limited concurrency

Example: Pictorial Depiction

Step 2: Assignment

- Specifying mechanism to divide work up among processes
  - e.g. which process performs which task
  - together with decomposition, also called partitioning
  - goal: 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

- Division of responsibility between programmer and architecture
  - programmers worry about partitioning first
  - usually independent of architecture or programming model
  - however, cost and complexity of parallel constructs may affect decisions
  - architecture assumes program is reasonably partitioned
  - cannot do anything if this is not the case!
Step 3: Orchestration

- **Issues**
  - naming data
  - structuring communication and synchronization
    assignment of tasks produces need for inter-process interactions
  - organizing data structures and scheduling tasks over time

- **Goals**
  - reduce communication and synchronization costs (visible to processors)
  - preserve locality of data reference
  - schedule tasks to satisfy task dependencies early
  - reduce overhead of parallelism management

- **Orchestration choices depend upon**
  - available primitives (programming models/languages)
  - efficiency of these primitives (architecture)

Step 4: Mapping

- Controls execution of processes by processors
  - implicit (OS) versus explicit (programmer)

- Degrees of control
  - which processes will run on same processor (collocation)
  - which process runs on which particular processor (placement)

- **Alternatives**
  - space-sharing
    - machine divided into subsets, only one program at a time in a subset
    - processes can be pinned to processors, or left to OS
  - complete resource management control left to OS
    - OS tries to achieve better resource sharing and utilization, and speed up execution of a parallel program (gang- or affinity-scheduling)
  - real world: user specifies some aspects, system takes care of rest

- Mapping in multiprogrammed systems is an active research area

Parallelizing Computation versus Data

- So far: Parallelization view centered around computation
  - computation is decomposed and assigned (partitioned)
  - data partitioning (if present) arises from how tasks access data

- Alternate view: Partition data
  - very natural perspective in data parallel models
    - same operation on each element of a data structure
    - computation follows data: owner computes
  - High Performance Fortran (HPF), other languages
    - e.g., grid-based computations (grid subset), data mining (part of database)

- Problem: not general enough
  - stronger distinction between computation and data in some applications
    - e.g., Raytrace (more details in a few slides)
  - retain computation-centric view
    - data access and communication is viewed as part of orchestration

High-level Goals

- High speedup but with low resource usage and development effort

<table>
<thead>
<tr>
<th>Step</th>
<th>Architecture-Dependent?</th>
<th>Major Performance Goals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decomposition</td>
<td>Mostly no</td>
<td>Expose enough concurrency but not too much</td>
</tr>
<tr>
<td>Assignment</td>
<td>Mostly no</td>
<td>Balance workload</td>
</tr>
<tr>
<td>Orchestration</td>
<td>Yes</td>
<td>Reduce communication via data locality</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reduce communication and synchronization cost as seen by the processor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reduce serialization at shared resources</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Schedule tasks to satisfy dependences early</td>
</tr>
<tr>
<td>Mapping</td>
<td>Yes</td>
<td>Put related processes on the same processor if necessary</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Exploit locality in network topology</td>
</tr>
</tbody>
</table>

- Implications
  - algorithm: high-performance, low resource needs
  - architecture: high-performance, low cost, reduced programming effort
Examples of Parallel Programs

[From Culler/Singh/Gupta: Chapter 2]

- Ocean: Simulating ocean currents
  - regular structure, scientific computing
  - amenable to both data- and computation-oriented partitioning
- Raytrace: Rendering scenes by ray tracing
  - irregular structure, computer graphics

[Other examples described in the text]

- Culler/Singh/Gupta
  - Barnes-hut, Data mining
- Andrews
  - Matrix multiplication, Adaptive quadrature, …

Ocean: Simulating Ocean Currents

- Model: Set of two-dimensional grids (cross sections)
  - discretize in space and time
    - each variable (pressure, velocity, various currents) has a value per grid point
    - finer spatial and temporal resolution => greater accuracy
  - many different computations per time step
    - set up and solve equations of motion
  - concurrency across and within grid computations

Ocean: Steps in the Parallelization Process

- Decomposition: Parallelism across and within grid computations
  - several choices of task granularity
    - coarse-grained: compute all grid-points in a cross section
    - medium-grained: compute one row of grid points
    - fine-grained: compute a single grid point
  - same task computes all time steps vs. new tasks created each time step
  - interaction because computation affected by neighboring grid points

- Assignment (assuming row decomposition)
  - static:
    - block: row i is assigned to process \( \left\lfloor \frac{i}{p} \right\rfloor \)
    - cyclic: process \( j \) is assigned rows \( j, j+p, … \)
  - dynamic:
    - get a row index, work on row, get next row, …

Ocean: Parallelization Steps (contd.)

- Orchestration
  - communication: values of neighboring grid points
    - from up/down cross sections
    - from neighbors in the same grid
  - synchronization: dependencies between tasks
    - e.g., a grid point cannot proceed to the next time step unless all its neighbors have completed the current time step
  - load-balance: not a problem since same work at each point

- Mapping
  - topology insensitive: leave it to the OS
  - topology sensitive
    - ensure processes that contain contiguous row blocks are mapped to neighboring processors
    - important for optimizing bandwidth
    - less so for latency
Raytrace: Rendering Scenes by Ray Tracing

- Scene: Set of objects in three-dimensional space
- Output: Two-dimensional array of pixels
  - color, brightness, opacity values for each pixel
  - as seen from a specific viewpoint (position of the eye)
- Approach: 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

Raytrace: Steps in the Parallelization Process

- Decomposition
  - ray-oriented approach: parallelism across rays
    - several choices of task granularity
      - coarse: sub-portion of image place (e.g., a 4x4 block of pixels)
      - fine: a single pixel
    - tasks do not need to interact: read-only access to scene objects
  - scene-oriented approach: parallelism across scene objects
    - partition three-dimensional scene into subspaces
    - task handles ray transport within subspace
    - interaction between tasks when a ray crosses over into the next subspace
  - both approaches: wide variations in task granularity
- Assignment (ray-oriented approach)
  - if scene can be replicated: load-balance is the primary issue
  - else, allocate tasks which access the same set of objects (reuse)
    - scatter decomposition: more on this in Lectures 8-10

Raytrace: Parallelization Steps (contd.)

- Orchestration
  - locality: need to ensure that tasks both
    - reuse the same set of scene objects (spatial locality)
      - achieved by a block assignment
    - reuse a particular scene objects as many times as possible (temporal locality)
      - harder to achieve, given the irregular nature of the computation
  - synchronization
    - none required between tasks

- Mapping
  - given irregular nature, not much that can be done explicitly

What do Parallel Programs Look Like?

- Examine a simplified version of a piece of Ocean simulation
  - iterative equation solver
    - using a finite differencing method
  - do each step in detail
- Illustrate parallel program in low-level parallel language
  - C-like pseudocode with simple extensions for parallelism
  - exposes basic communication and synchronization primitives
    - in each of the programming models
  - state of most real parallel programming today
Grid Solver Example

Gauss-Seidel (near-neighbor) sweeps to convergence
- interior n-by-n points of (n+2)-by-(n+2) grid updated in each sweep
  - updates done in-place
  - keep track of difference from previous value
- accumulate partial differences into global difference at end of every sweep
- do another sweep if error has not converged

Grid Solver Example: Decomposition

- Option 1: Look at loop iterations
  - dependence analysis; if not enough concurrency, then look further
  - not much concurrency here at this level (both loops are sequential)
- Option 2: Examine fundamental dependences, ignoring loop structure
  - concurrency $O(n)$ along anti-diagonals (southwest to northeast)
  - serialization $O(n)$ along diagonal

Grid Solver Example: Decomposition (contd.)

But, how to exploit this parallelism
- Option 1: Retain loop structure
  - synchronize between each pair of grid points
  - problem: too many synchronization operations
- Option 2: Restructure loops to loop over anti-diagonals
  - global synchronization between iterations
  - problem: imbalance and still too much synchronization
- Option 3: Exploit application knowledge
  - reorder grid traversal using red-black ordering (chess-board pattern)
Grid Solver Example: Red-Black Ordering

- Left-to-right, top-to-bottom ordering not fundamental to Gauss-Seidel
- Red-black ordering
  - decompose grid into two sets of points (as in a chess-board)
  - different ordering of updates: may converge quicker or slower
  - red sweep and black sweep are each fully parallel
  - global synchronization between them (conservative but convenient)
- Exploit additional asynchrony not present in the sequential algorithm

Grid Solver Example: Code for Decomposition

- Simpler example of asynchrony: no dependencies

```
15. while (!done) do /* a sequential loop */
16.    diff = 0;
17.    for_all i ← 1 to n do /* a parallel loop nest */
18.        for_all j ← 1 to n do
19.            temp = A[i,j];
21.            diff += abs(A[i,j] - temp);
22.        end for_all
23.    end for_all
24.    if (diff/(n*n) < TOL) then done = 1;
25.    end while
```

Grid Solver Example: Assignment

- Static assignment (given decomposition into rows)
  - block: row $i$ is assigned to process $P_i$
  - cyclic: process $j$ is assigned rows $j, j+p, ...$
  - dynamic assignment
    - get a row index, work on the row, get a new row, and so on

Grid Solver Example: Orchestration

- Requirements
  - communication
    - values of neighboring grid points must be available
  - synchronization
    - next iteration (or alternating red-black sweeps) cannot proceed until all grid points have been evaluated in the current sweep
- Language support for orchestration is programming model specific
  - data parallel
    - concurrent loops, computation and data structure decomposition, collective operations
  - shared memory
    - process creation, mutual exclusion, global synchronization, post-wait
  - message passing
    - process creation, synchronous and asynchronous send/receive, global synchronization
Data Parallel Models: Orchestration Support

- Dynamic allocation of shared data
  - `G_MALLOC` (global malloc)

- Concurrent loops
  - `for_all`
  - parallel processes are implicitly active: only within `for_all` body

- Decomposition of data and computation
  - `DECOMP arr[block, *, nprocs]`
  - specifies assignment of data elements to processes
  - `owner-computes`: specifies assignment of iterations to processes

- Collective operations
  - `REDUCE`, others such as broadcast, etc.
  - all-to-all operations, implemented efficiently by the underlying system

```
1. int n, nprocs; /*grid size (n + 2-by-n + 2) and number of processes*/
2. float **A, diff = 0;
3. main()
4. begin
5. read(n); read(nprocs); /*read input grid size and number of processes*/
6. A ← `G_MALLOC` (a 2-d array of size n+2 by n+2 doubles);
7. initialize(A); /*initialize the matrix A somehow*/
8. Solve (A); /*call the routine to solve equation*/
9. end main
10. procedure Solve(A) /*solve the equation system*/
11. float **A; /*A is an (n + 2-by-n + 2) array*/
12. begin
13. int i, j, done = 0;
14. float mydiff = 0, temp;
15. while (!done) do /*outermost loop over sweeps*/
16. mydiff = 0; /*initialize maximum difference to 0*/
17. `for_all` i ← 1 to n do /*sweep over non-border points of grid*/
18. `for_all` j ← 1 to n do
19. temp = A[i,j]; /*save old value of element*/
22. mydiff += abs(A[i,j] - temp);
23. `end for_all`
24. `for_all` j ← 1 to n do
24a. `REDUCE` (mydiff, diff, ADD);
25. if (diff/(n*n) < TOL) then done = 1;
26. `end while`
27. `end procedure`
```

Shared Memory Models: Orchestration Support

- Process creation and termination
  - `CREATE(p, proc, args)`
  - `WAIT_FOR_END(number)`

- Dynamic allocation of shared data
  - `G_MALLOC`

- Mutual exclusion
  - `LOCK(name):` acquire mutually exclusive access
  - `UNLOCK(name):` release access

- Global synchronization
  - `BARRIER(name, number)`
  - no process gets past barrier until `number` have arrived

- Point-to-point synchronization
  - `WAIT(flag)`: wait for flag to be set (spin or block)
  - `SIGNAL(flag)`: set flag, wake up waiting processes
  - producer-consumer sharing, semaphores

Shared Memory Model: Grid Solver Example

- Grid declared as a shared array
  - all processes can access it just as in sequential program

- Single program multiple data (SPMD) style
  - assignment controlled by values of variables used as loop bounds

```
Processes
Solve Solve Solve Solve
Sweep

Producers
```
1. int n, nprocs; /*matrix dimension and number of processors to be used*/
2a. float **A, diff; /*A is global (shared) array representing the grid*/ /*diff is global (shared) maximum difference in current sweep*/
2b. LOCKDEC(diff_lock); /*declaration of lock to enforce mutual exclusion*/
2c. BARDEC(bar1); /*barrier declaration for global synchronization between sweeps*/
3. main()
4. begin
5. read(n); read(nprocs); /*read input matrix size and number of processes*/
6. A ← G_MALLOC(a two-dimensional array of size n+2 by n+2 doubles);
7. initialize(A, nprocs, Solve, A);
8. Solve(A);
8b. WAIT_FOR_END(nprocs–1); /*wait for all child processes created to terminate*/
9. end main
10. procedure Solve(A)
11. float **A; /*A is entire n+2 by n+2 shared array, as in the sequential program*/
12. begin
13. int i,j, pid, done = 0;
14. float temp, mydiff = 0; /*private variables*/
14a. int mymin = 1 + (pid * n/nprocs); /*assume that n is exactly divisible by nprocs for simplicity here*/
14b. int mymax = mymin + n/nprocs - 1
15. while (!done) do /*outer loop over all diagonal elements*/
16. mydiff = 0; /*set global diff to 0 (okay for all to do it)*/
16a. BARRIER(bar1, nprocs); /*ensure all reach here before anyone modifies diff*/
17. for i ← mymin to mymax do /*for each of my rows*/
18. for j ← 1 to n do /*for all nonborder elements in that row*/
19. temp = A[i,j];
21. mydiff += abs(A[i,j] - temp);
22. endfor
23. endfor
24a. LOCK(diff_lock); /*update global diff if necessary*/
25. diff += mydiff; /*read for simplicity here*/
25b. UNLOCK(diff_lock);
25c. BARRIER(bar1, nprocs); /*ensure all reach here before checking if done*/
25d. if (diff/(n*n) < TOL) then done = 1; /*check convergence; all get same answer*/
26. endwhile
27. end procedure

Shared Memory Model: Grid Solver (contd.)

- Single program multiple data
  - not lockstep or even necessarily same instructions
  - assignment of iterations controlled by values of variables
  - done condition evaluated redundantly by all
  - code that does the update identical to sequential program
    - since grid array is in shared address space
    - each process has private mydiff variable

- Most interesting special operations are for synchronization
  - accumulations into shared diff have to be mutually exclusive
  - why all the barriers?

Grid Solver: Need for Mutual Exclusion

- Code each process executes
  - load the value of diff into register r1
  - store the value of register r1 into diff
  - store the value of register r1 into diff

- A possible interleaving
  \[ P_1 \]
  \[ r_1 \leftarrow \text{diff} \]
  \[ r_1 \leftarrow \text{diff} \]

  \[ \{ P_1 \text{ gets 0 in its } r_1 \} \]
  \[ \{ P_2 \text{ also gets 0} \} \]

  \[ r_1 \leftarrow r_1 + r_2 \]
  \[ r_1 \leftarrow r_1 + r_2 \]

  \[ \{ P_2 \text{ sets its } r_1 \text{ to 1} \} \]
  \[ \{ P_2 \text{ sets } r_1 \text{ to 1} \} \]

  \[ \text{diff} \leftarrow \text{diff} \]
  \[ \text{diff} \leftarrow \text{diff} \]

  \[ \{ P_2 \text{ also sets } \text{diff} \text{ to 1} \} \]

- need the sets of operations to be atomic (mutually exclusive)
- use of mydiff reduces contention for the lock
Message Passing Model: Orchestration Support

- Process creation and termination
  - CREATE
  - WAIT_FOR_END

- Communication: data-transfer + synchronization
  - SEND(src_addr, size, dest, tag)
    send size bytes from src_addr to dest process, with tag identifier
  - RECEIVE(buffer_addr, size, src, tag)
    receive a message of size from src process with tag identifier, and store it in buffer_addr
  - SEND_ASYNC, SEND_PROBE
  - RECEIVE_ASYNC, RECEIVE_PROBE

- Global synchronization
  - BARRIER

Message Passing Model: Grid Solver Example

- Structurally similar to shared memory program (still SPMD), but differs significantly in orchestration
  - data structures and data access/naming
    - cannot declare grid to be a shared array any more
    - need to 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
  - communication
    - transfers of entire rows between traversals
  - synchronization

Message Passing Model: Grid Solver (contd.)

- Private portions of grid array
  - use of ghost rows: to store neighbor values
- Core similar, but indices/bounds in local rather than global space
- Communication
  - receive does not transfer data, send does
  - at beginning of iteration (no asynchrony), whole rows at a time
- Synchronization
  - using sends and receives
    - update of global diff and event synchronization for done condition
    - could implement locks and barriers with messages
  - can use REDUCE and BROADCAST library calls to simplify code

/*communicate local diff values and determine if done, using reduction and broadcast*/
25b. REDUCE(0, mydiff, sizeof(float), ADD);
25c. if (pid == 0) then
25i. if (mydiff/(n*n) < TOL) then done = 1;
25k. endif
25m. BROADCAST(0, done, sizeof(int), DONE);
Send and Receive Alternatives

- Can extend functionality
  - stride, scatter-gather, groups
- Semantic flavors: based on when control is returned after call
  - affect when data structures or buffers can be reused at either end
  - affect event synchronization
    - synchronous messages provide synchronization through match
    - separate event synchronization needed with asynchronous messages
  - affect ease of programming and performance
- with synchronous messages, our code is deadlocked! Fix?

Orchestration: Summary

- Data parallel
  - decomposition of data structures (implicit assignment of tasks)
- Shared address space
  - shared and private data explicitly separated
  - no correctness need for data distribution
  - communication implicit in access patterns
- Synchronization via atomic operations on shared data
  - synchronization explicit and distinct from data communication
- Message passing
  - data distribution among local address spaces needed
  - no explicit shared structures
  - communication is explicit
  - synchronization implicit in communication
    - with synchronous SEND/RECEIVE primitives
    - mutual exclusion for free: only one process updating each address space

Grid Solver Program

- Decomposition and assignment (partitioning) similar in all three programming models
- Orchestration is different
  - data structures, data access/naming, communication, synchronization

<table>
<thead>
<tr>
<th></th>
<th>Data Parallel</th>
<th>Shared Memory</th>
<th>Message Passing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explicit global data structure?</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Assignment independent of data layout?</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Communication</td>
<td>Implicit</td>
<td>Implicit</td>
<td>Explicit</td>
</tr>
<tr>
<td>Synchronization</td>
<td>Implicit</td>
<td>Explicit</td>
<td>Implicit</td>
</tr>
<tr>
<td>Explicit replication of border rows?</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Next Lecture

- Models of Parallel Computation
  (or, what determines the performance of a parallel program)
  - analytical
    - PRAM, LogP
  - operational
    - message passing, shared memory
  - common issues
    - naming, synchronization, latency, bandwidth
- Tutorial
  - threads programming

Readings
- Culler/Singh/Gupta: Chapter 1 (1.3), 3
- LogP paper: follow link on course web page