Announcements

- Fill up form for accounts on NCSA Exemplar and Origin machines
  - only those of you taking the course for credit!
  - login information will be handed out on 09/24

Outline

- Last lecture
  - central and mainstream nature of parallel computing
    - application demands, technology and architecture trends, economics
  - convergence in parallel architectures
    - programming models: data-parallel, message passing, shared memory
    - architectures: small-scale shared memory, large-scale distributed memory, and large-scale shared memory

- This lecture
  - parallel programs
    - key steps: decomposition, assignment, orchestration, mapping
    - case studies: Ocean, Raytrace
  - parallel constructs in different programming models

[Culler/Singh/Gupta: Chapter 2, Almasi/Gottlieb: Chapter 2]

Why Bother with Programs?

- They’re what runs on the machines
  - helps make design decisions
  - helps evaluate systems tradeoffs

- Led to the key advances in uniprocessor architecture
  - 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
  \[
  \text{speedup}(p) = \frac{\text{performance}(1)}{\text{performance}(p)} \left(\frac{\text{time}(1)}{\text{time}(p)}\right)
  \]

- Secondary goals: Low programming effort, low resource needs

Some Important Concepts

- Task
  - arbitrary piece of undecomposed work in 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

\[\text{Computation} \Rightarrow \text{Tasks} \Rightarrow \text{Processes} \Rightarrow \text{Processors}\]

Steps in Creating a Parallel Program

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: 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

- Most 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
    $$\text{speedup} = \left( \frac{1}{\frac{1-s}{P} + s} \right) \leq \frac{1}{s}$$
  - 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)

$$\text{speedup} = \frac{2N^2}{\left( \frac{N^2}{P} + N^2 \right)} \leq \frac{P}{N^2}$$

- trick: 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 ($P$)

$$\text{speedup} = \frac{2N^2}{\left( \frac{N^2}{P} + N^2 + P \right)} \leq \frac{P}{1 + \frac{P^2}{N^2}}$$

Example: Pictorial Depiction

- area under curve is total work done (time with 1 processor)
- horizontal extent is lower bound on time (infinite processors)

Concurrency Profiles

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

- 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 noninherent 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

• Almasi/Gottlieb
  – Weather code, Seismic migration, Transactions, State space searching

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 $j, j+p, \ldots$
  – 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

Expression for updating each interior point:

\[
A(i,j) = 0.2 \times (A(i,j) + A(i-1,j) + A(i+1,j) + A(i,j-1) + A(i,j+1))
\]

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: 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.     end for_all
  22.   end for_all
  23.   if (diff/(n*n) < TOL) then done = 1;
  24. end while

  – for_all leaves assignment to the system
    • but implicit global synch. at end of for_all loop
    – as shown: a task is a single grid point, so O(n²) tasks
    – to decompose into rows: make line 18 loop sequential; O(n) tasks

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: Assignment

• Static assignment (given decomposition into rows)
  – block: row i is assigned to process $p_i$
  – cyclic: process $j$ is assigned rows $j, j+p, ...$
  – block assignment reduces communication requirement

• 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

**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

```
1. int n, nprocs; /*matrix dimension and number of processors to be used*/
2a. float **A, diff; /*A is a global shared array representing the grid*/
2b. LOCK(diff_lock); /*declaration of lock to enforce mutual exclusivity*/
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); /*variables A is in an unspecified way*/
8a. CREATE (nprocs–1, Solve, A); /*main process becomes a worker too*/
8. Solve(A);
9. end main
10. procedure Solve(A)
11. float **A; /*same as in the sequential program*/
12. begin
13. int i,j, pid, done = 0;
14a. int mymin = 1 + (pid * n/nprocs); /*assume that n is exactly divisible by*/
14b. int mymax = mymin + n/nprocs - 1 /*nprocs for simplicity here*/
15. while (!done) do
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
25a. LOCK(diff_lock);
25b. diff += mydiff; /*update global diff if necessary*/
25c. UNLOCK(diff_lock);
25d. BARRIER(bar1, nprocs); /*wait for all child processes created to terminate*/
25e. 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
  - add the register r2 to register r1
  - store the value of register r1 into diff

- A possible interleaving
  - need the sets of operations to be atomic (mutually exclusive)
  - use of mydiff reduces contention for the lock
Grid Solver: Need for Barriers

- Line 25d
  - ensures that all processes have updated $diff$
  - needed for ensuring that the tolerance check is correct
- Line 25f
  - ensures that each process waits for all others to get done before exiting
  - not strictly required
- Line 16a
  - ensures that there is no race condition between Lines 16 and 25b
  - else, an arbitrary slow process can reset $diff$ after a faster process has updated it for the next iteration

- Barriers are a form of many-to-many synchronization
  - WAIT/SIGNAL is an example of one-to-many synchronization
  - WAIT_FOR_END is an example of many-to-one synchronization

Message Passing Model: Orchestration Support

- Process creation and termination
  - CREATE
  - WAIT_FOR_END
- Communication: data-transfer + synchronization
  - SEND(src_addr, size, dest, tag)
  - RECEIVE(buffer_addr, size, src, tag)
  - 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

```c
int pid, n, b; /* process id, matrix dimension and number of processors to be used */
float **myA; /* my assigned rows of A */
main()
begin
read(n); read(nprocs); /* read input matrix size and number of processes */
CREATE(nprocs-1, Solve); /* main process becomes a worker too */
WAIT_FOR_END(nprocs-1); /* wait for all child processes created to terminate */
procedure Solve()
begin
  int i, j, pid, n' = n/nprocs, done = 0;
  float temp, tempdiff, mydiff = 0; /* private variables */
  myA ← malloc(a 2-d array of size [n/nprocs + 2] by n + 2); /* my assigned rows of A */
  initialize(myA); /* initialize my rows of A, in an unspecified way */
  while (!done) do
    mydiff = 0; /* set local diff to 0 */
    if (pid != 0) then SEND(&myA[1,0], n*sizeof(float), pid-1, ROW);
    if (pid = nprocs-1) then SEND(&myA[n',0], n*sizeof(float), pid+1, ROW);
    if (pid != 0) then RECEIVE(&myA[0,0], n*sizeof(float), pid-1, ROW);
    if (pid != nprocs-1) then RECEIVE(&myA[n'+1,0], n*sizeof(float), pid+1, ROW);
    for i ← 1 to n' do /* for each of my (nonghost) rows */
      for j ← 1 to n do /* for all nonborder elements in that row */
        temp = myA[i,j];
        mydiff += abs(myA[i,j] - temp);
    endfor
    endfor
    if (pid != 0) then SEND(mydiff, sizeof(float), 0, DIFF); /* process 0 holds global total diff */
    if (pid = 0) then RECEIVE(myA[0,0], n*sizeof(float), 0, ROW);
    if (pid = 0) then RECEIVE(myA[n'+1,0], n*sizeof(float), 0, ROW);
    for i ← 1 to nprocs-1 do /* for each other process */
      RECEIVE(&myA[i,0], n*sizeof(float), i, ROW);
      mydiff += tempdiff; /* accumulate into total */
    endfor
    if (mydiff/(n*n) < TOL) then done = 1;
    for i ← 1 to nprocs-1 do /* for each other process */
      SEND(done, sizeof(int), i, DONE);
    endfor
    endwhile
end procedure
```
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

```c
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

! communicate local diff values and determine if done, using reduction and broadcast/

Orchestration: Summary

- Data parallel
  - decomposition of data structures (implicit assignment of tasks)
- Shared address space
  - shared and private data explicitly separate
  - 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>
Lecture Summary

• Parallel programs
  – key steps:
    • decomposition: identify concurrent tasks
    • assignment: decide which task is performed by which process
    • orchestration: handle required communication and synchronization
    • mapping: decide which process executes on which processor
  – case studies: Ocean, Raytrace

• Parallel constructs in different programming models
  – data parallel: array decomposition, concurrent loops, collective operations
  – message passing: process creation, send/receive, global synchronization
  – shared memory: process creation, mutual exclusion, global and point-to-point synchronization

• What about performance?

Next Lecture

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

• Tutorial
  – data-parallel programming

Readings
  – Culler/Singh/Gupta: Chapter 1 (1.3), 3
  – Almasi/Gottlieb: Chapter 4
  – LogP paper: follow link on course web page