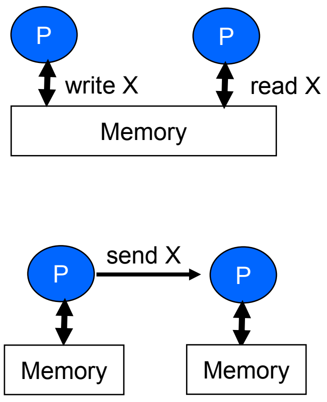
18-742
Lecture 3

Parallel Programming I

Spring 2005

Prof. Babak Falsafi

http://www.ece.cmu.edu/~ece742



Slides developed in part by Profs. Adve, Falsafi, Hill, Lebeck, Reinhardt, Smith, and Singh of University of Illinois, Carnegie Mellon University, University of Wisconsin, Duke University, University of Michigan, and Princeton University.

Homework 1

- **Due by Friday**
- **Homework 2 assigned on Friday**
  - **Parallel programming homework**

Readings

- **From Reader 2**
  - **Chapter 3**
  - **P. Ranganathan, K. Gharachorloo, S. V. Adve, and L. A. Barroso, *Performance of Database Workloads on Shared-Memory Systems with Out-of-Order Processors*, ASPLOS 1998.**
  - **A. R. Alameldeen, M. M. K. Martin, C. J. Mauer, K. E. Moore, M. Xu, D. J. Sorin, M. D. Hill, and D. A. Wood, *Simulating a $2M Commercial Server on a $2K PC*, IEEE Computer, February 2003.**
  - **A. R. Alameldeen and D. A. Wood, *Variability in Architectural Simulations of Multi-threaded Workloads*, HPCA 2003.**

Parallel Programming

To understand and evaluate
design decisions in a parallel machine,
we must get an idea of the software
that runs on a parallel machine.

--Introduction to Culler et al.'s Chapter 2,
beginning 192 pages on software

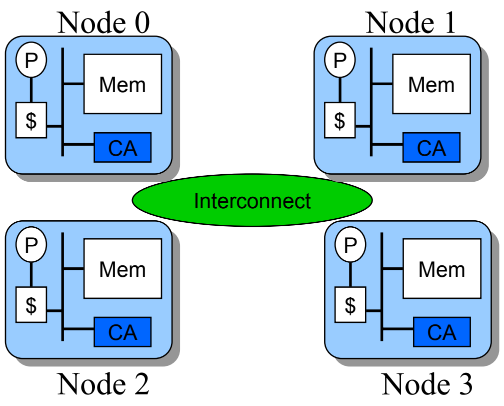
Outline

- Review
- Applications
- Creating Parallel Programs
- Programming for Performance
- Scaling

Review: Separation of Model and Architecture

- Shared Memory
  - Single shared address space
  - Communicate, synchronize using load / store
  - Can support message passing
- Message Passing
  - Send / Receive
  - Communication + synchronization
  - Can support shared memory
- Data Parallel
  - Lock-step execution on regular data structures
  - Often requires global operations (sum, max, min...)
  - Can support on either SM or MP

Review: A Generic Parallel Machine



- Separation of programming models from architectures
- All models require communication
- Node with processor(s), memory, communication assist

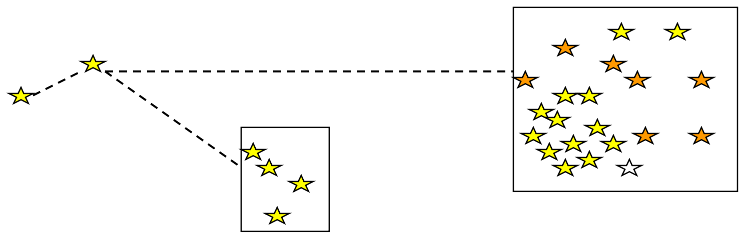
Review: Fundamental Architectural Issues

- Naming: How is communicated data and/or partner node referenced?
- Operations: What operations are allowed on named data?
- Ordering: How can producers and consumers of data coordinate their activities?
- Performance
  - Latency: How long does it take to communicate in a protected fashion?
  - Bandwidth: How much data can be communicated per second? How many operations per second?

Applications

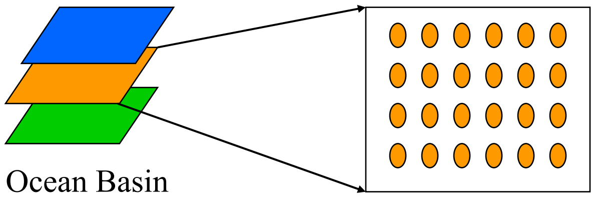
- N-Body Simulation: Barnes-Hut
- Ocean Current Simulation: Ocean
- VLSI Routing: Locus Route
- Ray Tracing
  – Shoot Ray through three dimensional scene (let it bounce off objects)
- Data Mining
  – finding associations
  – Consumers that are college students, and buy beer, tend to buy chips

Barnes-Hut



- Computing the mutual interactions of N bodies
  – n-body problems
  – stars, planets, molecules…
- Can approximate influence of distant bodies

Ocean

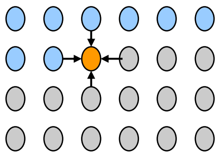


- Simulate ocean currents
- discretize in space and time

Creating a Parallel Program

- Can be done by programmer, compiler, run-time system or OS
- A Task is a piece of work
  - Ocean: grid point, row, plane
  - Raytrace: 1 ray or group of rays
- Task grain
  - small => fine-grain task
  - large => course-grain task
- Process (thread) performs tasks
  - According to OS: process = thread(s) + address space
- Process (threads) executed on processor(s)

Example: Ocean



- Equation Solver
  - kernel = small piece of important code (Not OS kernel…)
- Update each point based on NEWS neighbors
  - Gauss-Seidel (update in place)
- Compute average difference per element
- Convergence when diff small => exit

Equation Solver Decomposition

```
while !converged
    for
        for
```

- The loops are not independent!
- Exploit properties of problem
  - Don't really need up-to-date values (approximation)
  - May take more steps to converge, but exposes parallelism

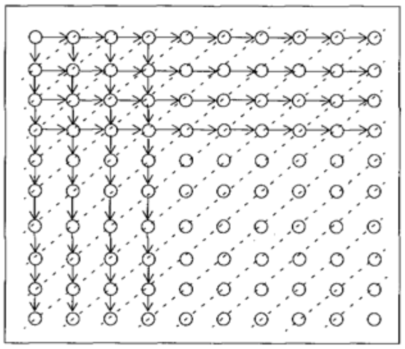
Sequential Solver

- **Recurrence in Inner Loop**

```
procedure Solve (A)                 /*solve the equation system*/
  float **A;                           /*A is an (n + 2)-by-(n + 2) array*/
begin
  int i, j, done = 0;
  float diff = 0, temp;
  while (!done) do                  /*outermost loop over sweeps*/
    diff = 0;                       /*initialize maximum difference to 0*/
    for i ← 1 to n do               /*sweep over nonborder points of grid*/
      for j ← 1 to n do
        temp = A[i,j];              /*save old value of element*/
        A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
          A[i,j+1] + A[i+1,j]);     /*compute average*/
        diff += abs(A[i,j] - temp);
      end for
    end for
    if (diff/(n*n) < TOL) then done = 1;
  end while
end procedure
```

Parallel Solver

- **Identical computation as Original Code**
- **Parallelism along anti-diagonal**
- **Different degrees of parallelism as diagonal grows/shrinks**



The FORALL Statement

```
while !converged
    forall
        forall
```

- Can execute the iterations in parallel
- Each grid point computation (n^2 parallelism)

```
while !converged
    forall
        for
```

- Computation for rows is independent (n parallelism)
  - less overhead

Asynchronous Parallel Solver

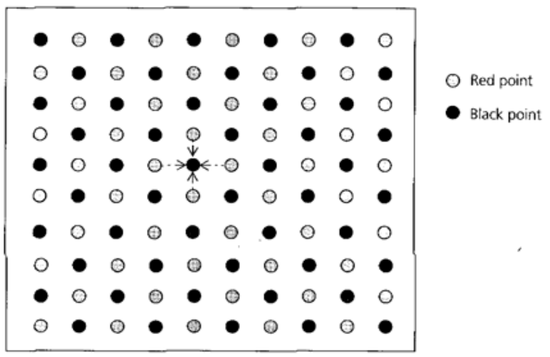
Each processor updates its region independent of other's values

- Global synch at end of iteration, to keep things somewhat up-to-date
- non-deterministic

```
while (!done) do                              /*a sequential loop*/
  diff = 0;
  for_all i ← 1 to n do                       /*a parallel loop nest*/
    for_all j ← 1 to n do
      temp = A[i,j];
      A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
        A[i,j+1] + A[i+1,j]);
      diff += abs(A[i,j] - temp);
    end for_all
  end for_all
  if (diff/(n*n) < TOL) then done = 1;
end while
```

Red-Black Parallel Solver

- Red-Black
  - like checkerboard update of Red point depends only on Black points
  - alternate iterations over red, then black
  - but convergence may change
  - deterministic



PARMACS

- **Macro Package, runtime system must implement**
  - **portability**

CREATE(p,proc,args)	Create p processes executing proc(args)
G_MALLOC(size)	Allocate shared data of size bytes
LOCK(name)	
UNLOCK(name)	
BARRIER(name,number)	Wait for number processes to arrive
WAIT_FOR_END(number)	Wait for number processes to terminate
WAIT(flag)	while(!flag);
SIGNAL(flag)	flag = 1;

Shared Memory Programming

```
procedure Solve(A)
  float **A;                              /*A is entire n+2-by-n+2 shared array,
                                          as in the sequential program*/
begin
  int i,j, pid, done = 0;
  float temp, mydiff = 0;                 /*private variables*/
  int mymin = 1 + (pid * n/nprocs);       /*assume that n is exactly divisible by*/
  int mymax = mymin + n/nprocs - 1        /*nprocs for simplicity here*/

  while (!done) do                  /*outer loop over all diagonal elements*/
    mydiff = diff = 0;              /*set global diff to 0 (okay for all to do it)*/
    BARRIER(bar1, nprocs);          /*ensure all reach here before anyone modifies diff*/
    for i ← mymin to mymax do /*for each of my rows*/
      for j ← 1 to n do             /*for all nonborder elements in that row*/
        temp = A[i,j];
        A[i,j] = 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
          A[i,j+1] + A[i+1,j]);
        mydiff += abs(A[i,j] - temp);
      endfor
    endfor
    LOCK(diff_lock);                /*update global diff if necessary*/
    diff += mydiff;
    UNLOCK(diff_lock);
    BARRIER(bar1, nprocs);          /*ensure all reach here before checking if done*/

    if (diff/(n*n) < TOL) then done = 1;       /*check convergence; all get
                                               same answer*/
    BARRIER(bar1, nprocs);
  endwhile
end procedure
```

Message Passing Primitives

Table 2.3 Some Basic Message-Passing Primitives

Name	Syntax	Function
CREATE	CREATE(procedure)	Create process that starts at procedure
SEND	SEND(src_addr, size, dest, tag)	Send size bytes starting at src_addr to the dest process, with tag identifier
RECEIVE	RECEIVE(buffer_addr, size, src, tag)	Receive a message with the tag identifier from the src process, and put size bytes of it into buffer starting at buffer_addr
SEND_PROBE	SEND_PROBE(tag, dest)	Check if message with identifier tag has been sent to process dest (only for asynchronous message passing, and meaning depends on semantics, as discussed in this section)
RECV_PROBE	RECV_PROBE(tag, src)	Check if message with identifier tag has been received from process src (only for asynchronous message passing, and meaning depends on semantics)
BARRIER	BARRIER(name, number)	Global synchronization among number processes: none gets past BARRIER until number have arrived
WAIT_FOR_END	WAIT_FOR_END(number)	Wait for number processes to terminate

Sends and Receives

- **Synchronous**
  - send: returns control to sending process after receive is performed
  - receive: returns control when data is written into address space
  - can deadlock on pairwise data exchanges
- **Asynchronous**
  - Blocking
    - » send: returns control when the message has been sent from source data structure (but may still be in transit)
    - » receive: like synchronous, but no ack is sent to sender
  - Non-Blocking
    - » send: returns control immediately
    - » receive: posts “intent” to receive
    - » probe operations determine completion

Message Passing Programming

- **Create separate processes**
- **Each process a portion of the array**
  - n/nprocs (+2) rows
  - boundary rows are passed in messages
    - » deterministic because boundaries only change between iterations
- **To test for convergence, each process computes mydiff and sends to proc 0**
  - synchronized via send/receive

Message Passing Programming 1 of 2

```
10.  procedure Solve()
11.  begin
13.    int i,j, pid, n' = n/nprocs, done = 0;
14.    float temp, tempdiff, mydiff = 0;       /*private variables*/
6.   myA ← malloc(a 2-d array of size [n/nprocs + 2] by n+2);
                                     /*my assigned rows of A*/
7.   initialize(myA);                /*initialize my rows of A, in an unspecified way*/

15.  while (!done) do
16.    mydiff = 0;                   /*set local diff to 0*/
16a.   if (pid != 0) then SEND(&myA[1,0],n*sizeof(float),pid-1,ROW);
16b.   if (pid = nprocs-1) then
           SEND(&myA[n',0],n*sizeof(float),pid+1,ROW);
16c.   if (pid != 0) then RECEIVE(&myA[0,0],n*sizeof(float),pid-1,ROW);
16d.   if (pid != nprocs-1) then
           RECEIVE(&myA[n'+1,0],n*sizeof(float), pid+1,ROW);
                                     /*border rows of neighbors have now been copied
                                     into myA[0,*] and myA[n'+1,*]*/
```

Message Passing Programming 2 of 2

```
17.    for i ← 1 to n' do            /*for each of my (nonghost) rows*/
18.      for j ← 1 to n do           /*for all nonborder elements in that row*/
19.        temp = myA[i,j];
20.        myA[i,j] = 0.2 * (myA[i,j] + myA[i,j-1] + myA[i-1,j] +
21.          myA[i,j+1] + myA[i+1,j]);
22.        mydiff += abs(myA[i,j] - temp);
23.      endfor
24.    endfor
                                     /*communicate local diff values and determine if
                                     done; can be replaced by reduction and broadcast*/
25a.   if (pid != 0) then                  /*process 0 holds global total diff*/
25b.     SEND(mydiff,sizeof(float),0,DIFF);
25c.     RECEIVE(done,sizeof(int),0,DONE);
25d.   else                                /*pid 0 does this*/
25e.     for i ← 1 to nprocs-1 do          /*for each other process*/
25f.       RECEIVE(tempdiff,sizeof(float),*,DIFF);
25g.       mydiff += tempdiff;             /*accumulate into total*/
25h.     endfor
25i      if (mydiff/(n*n) < TOL) then   done = 1;
25j.     for i ← 1 to nprocs-1 do          /*for each other process*/
25k.       SEND(done,sizeof(int),i,DONE);
25l.     endfor
25m.   endif
26.  endwhile
27.  end procedure
```

Programming for Performance

- Partitioning, Granularity, Communication, etc.
- Caches and Their Effects

Where Do Programs Spend Time?

- Sequential
  - Busy computing
  - Memory system stalls
- Parallel
  - Busy computing
  - Stalled for local memory
  - Stalled for remote memory (communication)
  - Synchronizing (load imbalance and operations)
  - Overhead
- Speedup (p) = time(1)/time(p)
  - Amdahl's Law
  - Superlinear

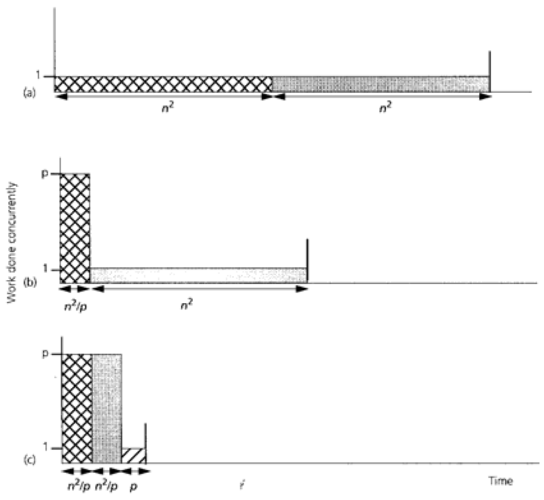
Speedup

$$\text{Speedup} \leq \frac{\text{Sequential Work}}{\mathit{max}(\text{Work on any processor})}$$

$$\text{Speedup} \leq \frac{\text{Sequential Work}}{\mathit{max}(\text{Work + Synch Wait + Communication})}$$

Concurrency Profile

- Plot number of concurrent tasks over time
- Example: operate on n^2 parallel data points, then sum them
  - sum sequentially
  - first sum blocks in parallel then sum p partial sums sequentially



Concurrency Profile: Speedup

- $Speedup \le \frac{Area\ under\ concurrency\ profile}{Horizontal\ extent\ of\ concurrency\ profile}$

  $f_k = fraction\ of\ work\ with\ concurrency\ k\ (\le p)$

  $p = number\ of\ processors$

  $$Speedup(p) \le \frac{\sum_{k=1}^{p} f_k}{\sum_{k=1}^{p} f_k \left(\frac{1}{k}\right)} = \frac{1}{\sum_{k=1}^{p} \frac{f_k}{k}}$$

- **Normalize total work to 1; make concurrency either serial or completely parallel ⇒ Amdahl's Law**

$$\frac{1}{s + \frac{1-s}{p}}$$

note: book has an *unusual* formulation of Amdahl's law (based on fraction of time rather than fraction of work)

Partitioning for Performance

- **Balance workload**
  - **reduce time spent at synchronization**
- **Reduce communication**
- **Reduce extra work**
  - **determining and managing good assignment**
- **These are at odds with each other**

Data vs. Functional Parallelism

- Data Parallelism
  - same ops on different data items
- Functional (control, task) Parallelism
  - pipeline
- Impact on load balancing?
- Functional is more difficult
  - longer running tasks

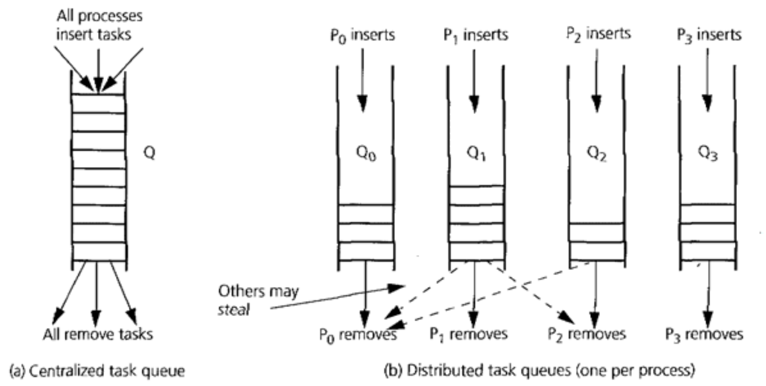
Impact of Task Granularity

- Granularity = Amount of work associated with task
- Large tasks
  - worse load balancing
  - lower overhead
  - less contention
  - less communication
- Small tasks
  - too much synchronization
  - too much management overhead
  - might have too much communication (affinity scheduling)

Impact of Concurrency

- Managing Concurrency
  - load balancing
- Static
  - Can not adpat to changes
- Dynamic
  - Can adapt
  - Cost of management increases
  - Self-scheduling (guided self-scheduling)
  - Centralized task queue
    - » contention
  - Distributed task queue
    - » Can steal from other queues
    - » Arch: Name data associated with stolen task

Task Queues



(a) Centralized task queue (b) Distributed task queues (one per process)

Dynamic Load Balancing

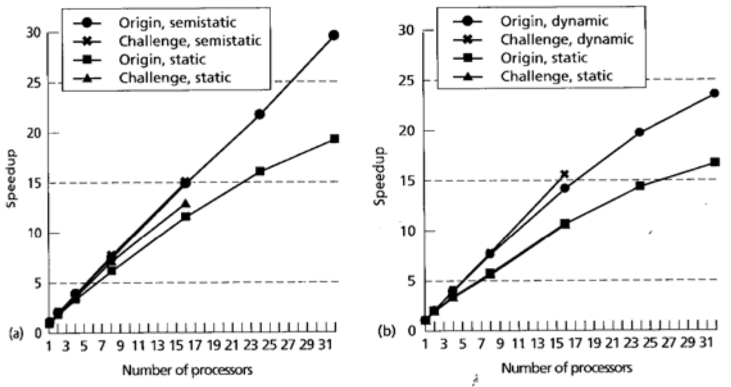


FIGURE 3.2 Illustration of the performance impact of dynamic partitioning for load balance. The graph in (a) shows the speedups of the Barnes-Hut application with and without semistatic partitioning, and the graph in (b) shows the speedups of Raytrace with and without dynamic tasking. Even in these applications that have a lot of parallelism, dynamic partitioning is important for improving load balance over static partitioning.

Impact of Synchronization and Serialization

- Too coarse synchronization
  - barriers instead of point-to-point synch
  - poor load balancing
- Too many synchronization operations
  - lock each element of array
  - costly operations
- Coarse grain locking
  - lock entire array
  - serialize access to array
- Architectural aspects
  - cost of synchronization operation
  - synchronization name space