UPC Ph.D. Course on
Parallel Computer Architecture

Parallel Programming (Chapters 2, 3, & 4)

University of Wisconsin-Madison

Slides are derived from work by
Sarita Adve (Illinois), Babak Falsafi (CMU),
Alvy Lebeck (Duke), Steve Reinhardt (Michigan),
and J. P. Singh (Princeton). Thanks!

Parallel Programming

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

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

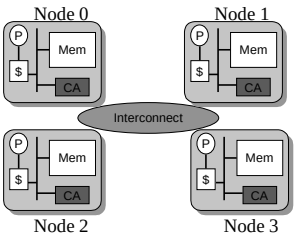
Outline

- **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?

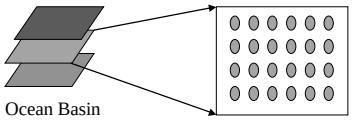
Applications

- 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

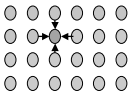
- Computing the mutual interactions of N bodies
  - n-body problems
  - stars, planets, molecules…
- Can approximate influence of distant bodies

Ocean



- Simulate ocean currents
- discretize in space and time

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
- Red-Black
  - like checkerboard update of Red point depends only on Black points
  - alternate iterations over red, then black
- Asynchronous
  - Each processor updates its region independent of other's values
  - Global synch at end of iteration, to keep things somewhat up-to-date

The FORALL Statement

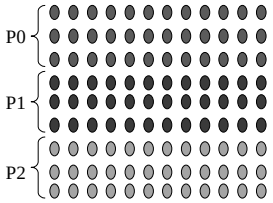
```
while !converged
    forall
        forall
```

- 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

Equation Solver Assignment



- **Each process gets a contiguous block of rows**

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;

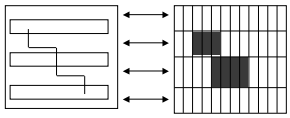
Equation Solver: The Ugly Code

```
main()
   parse_arguments();
   A = G_MALLOC(size of big array);
   CREATE(nprocs-1,Solve, A);
   Solve(A)
   WAIT_FOR_END;
end main
Solve(A)
   while !done
      for i = my_start to my_end
         for j = 1 to n
            mydiff += abs(A[i,j] - temp);
   LOCK(diff_lock);
      diff += mydiff;
   UNLOCK(diff_lock);
   if (convergence_test) done =1
   BARRIER
```

Example: Programmer / Software

- **LocusRoute (standard cell router)**

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



Shared Memory Implementation

- **Shared memory algorithm**
  - Divide cost array into regions
    - » logically assign regions to PEs
  - Assign wires to PEs based on the region in which center lies
  - Do load balancing using stealing when local queue empty
- **Pros:**
  - Good load balancing
  - Mostly local accesses
  - High cache hit ratio
- **Cons:**
  - non-deterministic
  - potential hot spots
  - amount of parallelism

Message Passing Implementations

- **Method 1:**
  - Distribute wires and cost array regions as in SM implementation
  - When wire-path crosses to remote region
    - » send computation to remote PE, or
    - » send message to access remote data
- **Method 2:**
  - Distribute only wires as in SM implementation
  - Fully replicate cost array on each PE
    - » one owned region, and potential stale copy of others
    - » send updates so copies are not too stale
  - Consequences:
    - » waste of memory in replication
    - » stale data => poorer quality results or more iterations
- **Both methods require lots of thought for programmer**

Programming for Performance

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

Aside on Cost-Effective Computing

- Isn't Speedup(P) < P inefficient?
- If only throughput matters, use P computers instead?
- But much of a computer's cost is NOT in the processor [Wood & Hill, IEEE Computer 2/95]
- Let Costup(P) = Cost(P)/Cost(1)
- Parallel computing cost-effective:
- Speedup(P) > Costup(P)
- E.g. for SGI PowerChallenge w/ 500MB:
- Costup(32) = 8.6

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

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

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

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)

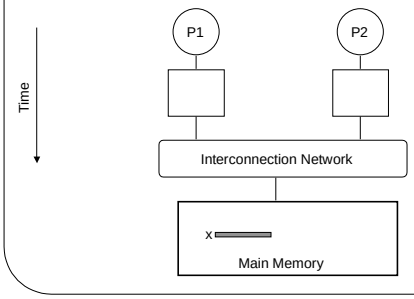
Impact of Concurrency

- 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

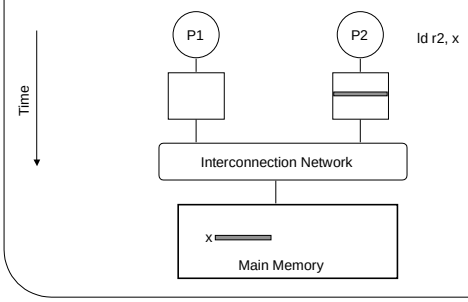
Impact of Synchronization and Serialization

- 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

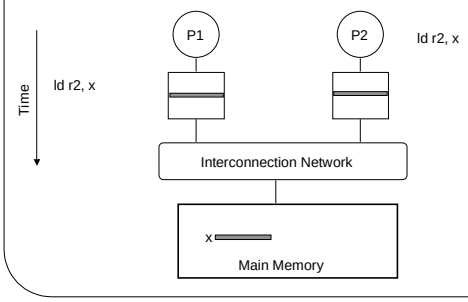
Cache Coherent Shared Memory



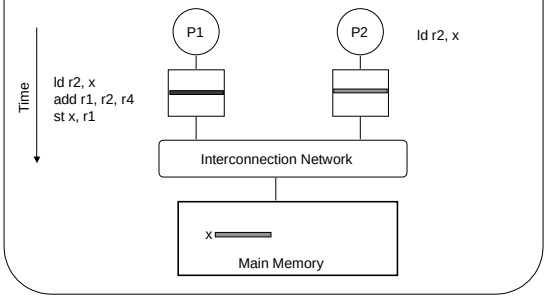
Cache Coherent Shared Memory



Cache Coherent Shared Memory



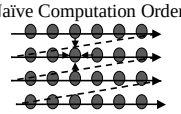
Cache Coherent Shared Memory



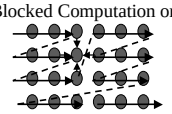
Reorder Computation for Caches

- **Exploit Temporal and Spatial Locality**
  - **Temporal locality affects replication**
  - **Touch too much data == capacity misses**
- **Computation Blocking**

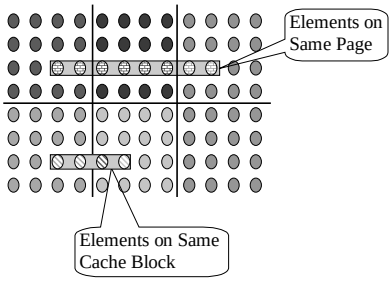
Naïve Computation Order



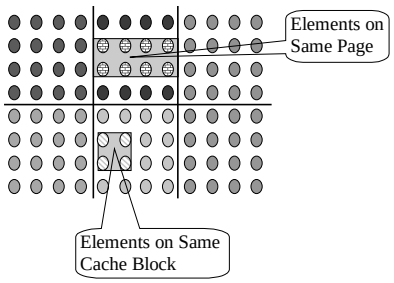
Blocked Computation order



Reorder Data for Caches: Before



Reorder Data for Caches: With Blocking



Scaling: Why Talk About it?

- **Speedup: change in performance as system parameter is scaled (e.g., number of processors, P)**
- **New problems on new machines**
  - **problem scaling**
  - **data set size**
  - **algorithmic complexity**
- **Scaling is natural when simulating physical phenomena**
  - **Space is grid**
  - **Refine grid size**
  - **Larger grid**

Questions in Scaling

- **Fundamental Question:**

  Why to real users actually do when they get access to larger parallel machines?

- **Constant Problem Size**
  - **Just add more processors**
- **Memory Constrained**
  - **Scale data size linearly with # of processors**
  - **Can significantly increase execution time**
- **Time Constrained**
  - **Keep same wall clock time as processors are added**
  - **Solve largest problem in same amount of time**

Problem Constrained Scaling

- **User wants to solve same problem, only faster**
  - **E.g., Video compression & VLSI routing**
- $$Speedup_{PC}(p) = \frac{Time(1)}{Time(p)}$$
- ***Assessment***
  - ***Good: easy to do & explain***
  - ***May not be realistic***
  - ***Doesn't work well for much larger machine (c.f., Amdahl's Law)***

Time Constrained Scaling

- **Execution time is kept fixed as system scales**
  - **User has fixed time to use machine or wait for result**
- **Performance = Work/Time as usual, and time is fixed, so**

$$SpeedupTC(p) = \frac{Work(p)}{Work(1)}$$

- **Assessment**
  - **Often realistic (e.g., best weather forecast over night)**
  - **Must understand application to scale meaningfully (would scientist scale grid, time step, error bound, or combination?)**
  - **Execution time on a single processor can be hard to get (no uniprocessor may have enough memory)**

Memory Constrained Scaling

- **Scale so memory usage per processor stays fixed**
- **Scaled Speedup: Is Time(1) / Time(p)?**

$$Speedup_{MC}(p) = \frac{Work(p)}{Time(p)} \times \frac{Time(1)}{Work(1)} = \frac{Increase\ in\ Work}{Increase\ in\ Time}$$

- **Assessment**
  - **Realistic for memory-constrained programs (e.g., grid size)**
  - **Can lead to large increases in execution time if work grows faster than linearly in memory usage**
  - **e.g. matrix factorization**
    - » **10,000-by 10,000 matrix takes 800MB and 1 hour on uniprocessor**
    - » **With 1,000 processors, can run 320K-by-320K matrix**
    - » **but ideal parallel time grows to 32 hours!**

Scaling Down

- Scale down to shorten evaluation time on hardware and especially on simulators
- "Scale up" issues apply in reserve
- Must watch out if problem size gets too small
  - Communication dominates computation (e.g., all boundary elements)
  - Problem size gets too small for realistic caches, yielding too many cache hits
    - » Scale caches down considering application working sets
    - » E.g., if a on a realistic problem a realistic cache could hold a matrix row but not whole matrix
    - » Scale cache so it hold only row or scaled problem's matrix

The SPLASH2 Benchmarks

- Kernels
  - Complex 1D FFT
  - Blocked LU Factorization
  - Blocked Sparse Cholesky Factorization
  - Integer Radix Sort
- Applications
  - Barnes-Hut: interaction of bodies
  - Adaptive Fast Multipole (FMM): interaction of bodies
  - Ocean Simulation
  - Hierarchical Radiosity
  - Ray Tracer (Raytrace)
  - Volume Renderer (Volrend)
  - Water Simulation with Spatial Data Structrue (Water-Spatial)
  - Water Simulation without Spatial Data Structure (Water-Nsquared)