Lecture 8: Parallel Processing

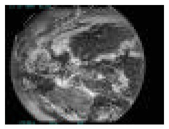
Prof. Fred Chong
ECS 250A Computer Architecture
Winter 1999

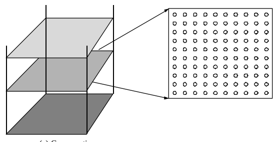
(Adapted from Culler CS258 and Dally EE282)

Parallel Programming

- Motivating Problems (application case studies)
- Process of creating a parallel program
- What a simple parallel program looks like
  - three major programming models
  - What primitives must a system support?

Simulating Ocean Currents





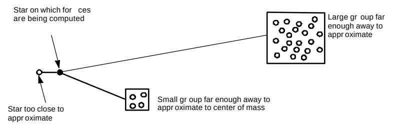
(a) Cross sections

(b) Spatial discretization of a cross section

- Model as two-dimensional grids
  - Discretize in space and time
  - finer spatial and temporal resolution => greater accuracy
- Many different computations per time step
  - » set up and solve equations
  - Concurrency across and within grid computations
- Static and regular

Simulating Galaxy Evolution

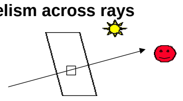
- Simulate the interactions of many stars evolving over time
- Computing forces is expensive
  - O(n^2) brute force approach
  - Hierarchical Methods take advantage of force law: $G\frac{m_1 m_2}{r^2}$


Star on which for ces are being computed
Large gr oup far enough away to appr oximate
Small gr oup far enough away to appr oximate to center of mass
Star too close to appr oximate

•Many time-steps, plenty of concurrency across stars within one

Rendering Scenes by Ray Tracing

- Shoot rays into scene through pixels in image plane
- Follow their paths
  - they bounce around as they strike objects
  - they generate new rays: ray tree per input ray
- Result is color and opacity for that pixel
- Parallelism across rays



- How much concurrency in these examples?

Creating a Parallel Program

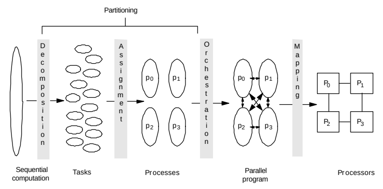
- Pieces of the job:
  - Identify work that can be done in parallel
    - » work includes computation, data access and I/O
  - Partition work and perhaps data among processes
  - Manage data access, communication and synchronization

Definitions

- ***Task***:
  - Arbitrary piece of work in parallel computation
  - Executed sequentially; concurrency is only across tasks
  - E.g. a particle/cell in Barnes-Hut, a ray or ray group in Raytrace
  - Fine-grained versus coarse-grained tasks
- ***Process (thread)***:
  - Abstract entity that performs the tasks assigned to processes
  - Processes communicate and synchronize to perform their tasks
- ***Processor***:
  - Physical engine on which process executes
  - Processes virtualize machine to programmer
    - » write program in terms of processes, then map to processors

FTC.W99 7

4 Steps in Creating a Parallel Program



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

FTC.W99 8

Decomposition

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

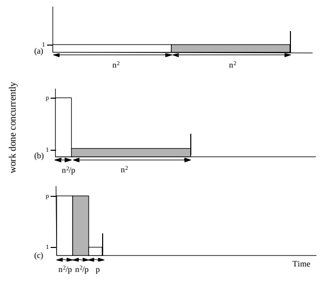
FTC.W99 9

Limited Concurrency: Amdahl's Law

- Most fundamental limitation on parallel speedup
- If fraction *s* of seq execution is inherently serial, speedup <= *1/s*
- Example: 2-phase calculation
  - sweep over *n*-by-*n* grid and do some independent computation
  - sweep again and add each value to global sum
- Time for first phase = n^2/p
- Second phase serialized at global variable, so time = n^2
- Speedup <= $\dfrac{2n^2}{\frac{n^2}{p} + n^2}$ or at most 2
- Trick: divide second phase into two
  - accumulate into private sum during sweep
  - add per-process private sum into global sum
- Parallel time is n^2/p + *n2/p* + p, and speedup at best $\dfrac{2n^2}{2n^2 + p^2}$

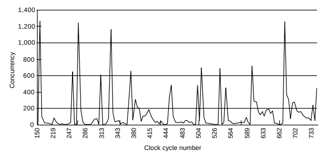
FTC.W99 10

Understanding Amdahl's Law



FTC.W99 11

Concurrency Profiles



- Area under curve is total work done, or time with 1 processor
- Horizontal extent is lower bound on time (infinite processors)

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

FTC.W99 12

Orchestration

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

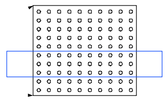
- Goals
  - Reduce cost of communication and synch.
  - Preserve locality of data reference
  - Schedule tasks to satisfy dependences early
  - Reduce overhead of parallelism management
- Choices depend on Prog. Model., comm. abstraction, efficiency of primitives
- Architects should provide appropriate primitives efficiently

Mapping

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

Parallelizing Computation vs. Data

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



Architect's Perspective

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

High-level Goals

Table 2.1 Steps in the Parallelization Process and Their Goals

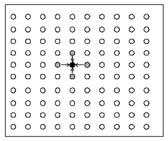
Step	Architecture-Dependent?	Major Performance Goals
Decomposition	Mostly no	Expose enough concurrency but not too much
Assignment	Mostly no	Balance workload Reduce communication volume
Orchestration	Yes	Reduce noninherent communication via data locality Reduce communication and synchronization cost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

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

What Parallel Programs Look Like

Example: iterative equation solver

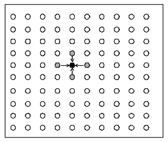
- Simplified version of a piece of Ocean simulation
- Illustrate program in low-level parallel language
  - C-like pseudocode with simple extensions for parallelism
  - Expose basic comm. and synch. primitives
  - State of most real parallel programming today



Expression for updating each interior point:

A[i,j] = 0.2 × (A[i,j] + A[i,j – 1] + A[i – 1, j] +
A[i,j + 1] + A[i + 1, j])

Grid Solver



Expression for updating each interior point:

A[i,j] = 0.2 × (A[i,j] + A[i,j – 1] + A[i – 1, j] +
A[i,j + 1] + A[i + 1, j])

- **Gauss-Seidel (near-neighbor) sweeps to convergence**
  - interior n-by-n points of (n+2)-by-(n+2) updated in each sweep
  - updates done in-place in grid
  - difference from previous value computed
  - accumulate partial diffs into global diff at end of every sweep
  - check if has converged
    - » to within a tolerance parameter

Sequential Version

```
1. int n;                                      /*size of matrix: (n + 2-by-n + 2) elements*/
2. float **A, diff = 0;

3. main()
4. begin
5.    read(n) ;                                /*read input parameter: matrix size*/
6.    A ← 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 diff = 0, temp;
15.   while (!done) do                         /*outermost loop over sweeps*/
16.      diff = 0;                             /*initialize maximum difference to 0*/
17.      for i ← 1 to n do                     /*sweep over nonborder points of grid*/
18.         for j ← 1 to n do
19.            temp = A[i,j];                  /*save old value of element*/
20.            A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21.               A[i,j+1] + A[i+1,j]);        /*compute average*/
22.            diff += abs(A[i,j] - temp);
23.         end for
24.      end for
25.      if (diff/(n*n) < TOL) then done = 1;
26.   end while
27.end procedure
```

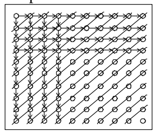
Decomposition

•Simple way to identify concurrency is to look at loop iterations

–*dependence analysis*; if not enough concurrency, then look further

•Not much concurrency here at this level (all loops *sequential*)

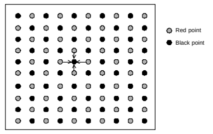
•Examine fundamental dependences



- **Concurrency *O(n)* along anti-diagonals, serialization *O(n)* along diag.**
- **Retain loop structure, use pt-to-pt synch; Problem: too many synch ops.**
- **Restructure loops, use global synch; imbalance and too much synch**

Exploit Application Knowledge

•Reorder grid traversal: red-black ordering



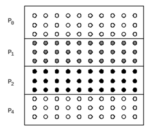
- **Different ordering of updates: may converge quicker or slower**
- **Red sweep and black sweep are each fully parallel:**
- **Global synch between them (conservative but convenient)**
- **Ocean uses red-black**
- **We use simpler, asynchronous one to illustrate**
  - » **no red-black, simply ignore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

Decomposition

```
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];
20.        A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21.          A[i,j+1] + A[i+1,j]);
22.        diff += abs(A[i,j] - temp);
23.      end for_all
24.    end for_all
25.    if (diff/(n*n) < TOL) then done = 1;
26.  end while
```

- **Decomposition into elements: degree of concurrency *n^2***
- **Decompose into rows?**

Assignment



- Static assignment: decomposition into rows
  - block assignment of rows: Row *i* is assigned to process $\left\lfloor \frac{i}{p} \right\rfloor$
  - cyclic assignment of rows: process *i* is assigned rows *i*, *i+p*, ...
- Dynamic assignment
  - get a row index, work on the row, get a new row, ...
- What is the mechanism?
- Concurrency? Volume of Communication?

Data Parallel Solver

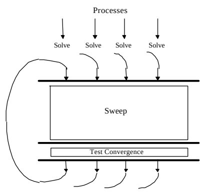
```
1.   int n, nprocs;                              /*grid size (n + 2-by-n + 2) and number of processes*/
2.   float **A, diff = 0;

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;
14a.     DECOMP A[BLOCK,*, nprocs];
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*/
20.          A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21.            A[i,j+1] + A[i+1,j]);             /*compute average*/
22.          mydiff[i,j] = abs(A[i,j] - temp);
23.        end for_all
24.      end for_all
24a.       REDUCE (mydiff, diff, ADD);
25.      if (diff/(n*n) < TOL) then done = 1;
26.    end while
27.  end procedure
```

Shared Address Space Solver

Single Program Multiple Data (SPMD)



- Assignment controlled by values of variables used as loop bounds

Generating Threads

```
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);             /*initialize A in an unspecified way*/
8a.      CREATE (nprocs–1, Solve, A);
8.       Solve(A);                  /*main process becomes a worker too*/
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. ----
27. end procedure
```

Assignment Mechanism

```
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*/
14b.       int mymax = mymin + n/nprocs - 1          /*nprocs for simplicity here*/

15.        while (!done) do                          /*outer loop sweeps*/
16.          mydiff = diff = 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];
20.                 A[i,j] = 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21.                         A[i,j+1] + A[i+1,j]);
22.                 mydiff += abs(A[i,j] - temp);
23.             endfor
24.           endfor
25a.          LOCK(diff_lock);                              /*update global diff if necessary*/
25b.            diff += mydiff;
25c.          UNLOCK(diff_lock);
25d.          BARRIER(bar1, nprocs);                 /*ensure all reach here before checking if done*/
25e.          if (diff/(n*n) < TOL) then done = 1;   /*check convergence; all get
                                                        same answer*/
25f.          BARRIER(bar1, nprocs);
26.        endwhile
27.   end procedure
```

SAS Program

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

Mutual Exclusion

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

FTC.W99 31

Global Event Synchronization

- BARRIER(nprocs): wait here till nprocs processes get here
  - Built using lower level primitives
  - Global sum example: wait for all to accumulate before using sum
  - Often used to separate phases of computation

Process P_1	*Process P_2*	*Process P_nprocs*
set up eqn system	set up eqn system	set up eqn system
Barrier **(name, nprocs)**	Barrier **(name, nprocs)**	Barrier **(name, nprocs)**
solve eqn system	solve eqn system	solve eqn system
Barrier **(name, nprocs)**	Barrier **(name, nprocs)**	Barrier **(name, nprocs)**
apply results	apply results	apply results
Barrier **(name, nprocs)**	Barrier **(name, nprocs)**	Barrier **(name, nprocs)**

  - Conservative form of preserving dependences, but easy to use
- WAIT_FOR_END (nprocs-1)

FTC.W99 32

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

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

P_1 | P_2

```
                                          A = 1;
a: while (flag is 0) do nothing;          b: flag = 1;
   print A;
```

•*Busy-waiting* or *spinning*

FTC.W99 33

Group Event Synchronization

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

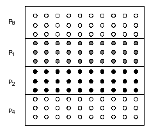
FTC.W99 34

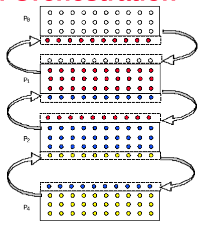
Message Passing Grid Solver

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

FTC.W99 35

Data Layout and Orchestration





Data partition allocated per processor

Add ghost rows to hold boundary data

Send edges to neighbors

Receive into ghost rows

Compute as in sequential program

FTC.W99 36

```
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);
                                                    /*initialize my rows of A, in an unspecified way*/

15.  while (!done) do
16.     mydiff = 0;                                   /*set local diff to 0*/
     /* Exchange border rows of neighbors into myA[0,*] and myA[n'+1,*]*/
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);
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
```

Notes on Message Passing Program

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

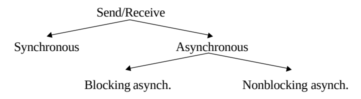
```
        /*communicate local diff values and determine if done, using reduction and broadcast*/
25b.       REDUCE(0,mydiff,sizeof(float),ADD);
25c.       if (pid == 0) then
25i.          if (mydiff/(n*n) < TOL) then done = 1;
25k.       endif
25m           BROADCAST(0,done,sizeof(int),DONE);
```

Send and Receive Alternatives

Can extend functionality: stride, scatter-gather, groups

Semantic flavors: based on when control is returned

Affect when data structures or buffers can be reused at either end



- Affect event synch (mutual excl. by fiat: only one process touches data)
- Affect ease of programming and performance

- Synchronous messages provide built-in synch. through match
  - Separate event synchronization with asynch. messages
- With synch. messages, our code is deadlocked. Fix?

Orchestration: Summary

- Shared address space
  - Shared and private data explicitly separate
  - Communication implicit in access patterns
  - No *correctness* need for data distribution
  - 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 (implicit in comm. patterns)
  - Communication is explicit
  - Synchronization implicit in communication (at least in synch. case)
    - » mutual exclusion by fiat

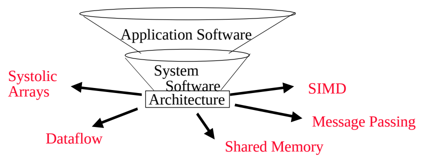
Correctness in Grid Solver Program

	SAS	Msg-Passing
Explicit global data structure?	Yes	No
Assignment indept of data layout?	Yes	No
Communication	Implicit	Explicit
Synchronization	Explicit	Implicit
Explicit replication of border rows?	No	Yes

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

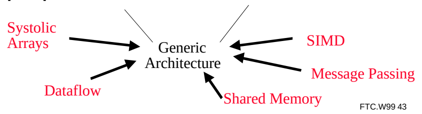
History of Parallel Architectures

- Parallel architectures tied closely to programming models
  - Divergent architectures, with no predictable pattern of growth.
  - Mid 80s rennaisance



Convergence

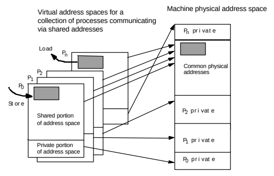
- Look at major programming models
  - where did they come from?
  - The 80s architectural rennaisance!
  - What do they provide?
  - How have they converged?
- Extract general structure and fundamental issues
- Reexamine traditional camps from new perspective



Programming Model

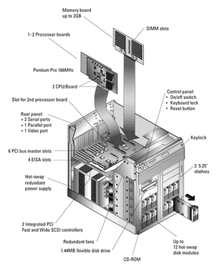
- *Conceptualization of the machine that programmer uses in coding applications*
  - How parts cooperate and coordinate their activities
  - Specifies communication and synchronization operations
- Multiprogramming
  - no communication or synch. at program level
- *Shared address space*
  - like bulletin board
- *Message passing*
  - like letters or phone calls, explicit point to point
- *Data parallel*:
  - more regimented, global actions on data
  - Implemented with shared address space or message passing

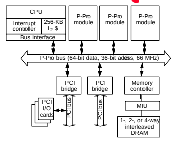
Structured Shared Address Space


Virtual address spaces for a collection of processes communicating via shared addresses
Machine physical address space
Load
Store
P_n
P_2
P_1
P_0
Shared portion of address space
Private portion of address space
P_n private
Common physical addresses
P_2 private
P_1 private
P_0 private

- Add hoc parallelism used in system code
- Most parallel applications have structured SAS
- Same program on each processor
  - shared variable X means the same thing to each thread

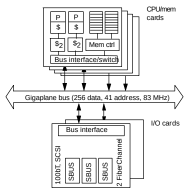
Engineering: Intel Pentium Pro Quad





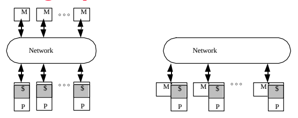
- All coherence and multiprocessing glue in processor module
- Highly integrated, targeted at high volume
- Low latency and bandwidth

Engineering: SUN Enterprise



- Proc + mem card - I/O card
  - 16 cards of either type
  - All memory accessed over bus, so symmetric
  - Higher bandwidth, higher latency bus

Scaling Up

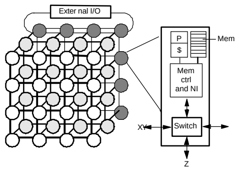


"Dance hall"

Distributed memory

- Problem is interconnect: cost (crossbar) or bandwidth (bus)
- Dance-hall: bandwidth still scalable, but lower cost than crossbar
  - » latencies to memory uniform, but uniformly large
- Distributed memory or non-uniform memory access (NUMA)
  - » Construct shared address space out of simple message transactions across a general-purpose network (e.g. read-request, read-response)
- Caching shared (particularly nonlocal) data?

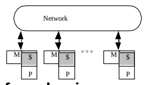
Engineering: Cray T3E



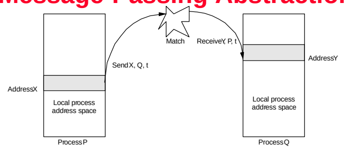
- Scale up to 1024 processors, 480MB/s links
- Memory controller generates request message for non-local references
- No hardware mechanism for coherence
  - » SGI Origin etc. provide this

Message Passing Architectures

- Complete computer as building block, including I/O
  - Communication via explicit I/O operations
- Programming model
  - direct access only to private address space (local memory),
  - communication via explicit messages (send/receive)
- High-level block diagram
  - Communication integration?
    - » Mem, I/O, LAN, Cluster
  - Easier to build and scale than SAS
- Programming model more removed from basic hardware operations
  - Library or OS intervention



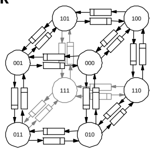
Message-Passing Abstraction



- Send specifies buffer to be transmitted and receiving process
- Recv specifies sending process and application storage to receive into
- Memory to memory copy, but need to name processes
- Optional tag on send and matching rule on receive
- User process names local data and entities in process/tag space too
- In simplest form, the send/recv match achieves pairwise synch event
  - » Other variants too
- Many overheads: copying, buffer management, protection

Evolution of Message-Passing Machines

- Early machines: FIFO on each link
  - HW close to prog. Model;
  - synchronous ops
  - topology central (hypercube algorithms)



CalTech Cosmic Cube (Seitz, CACM Jan 95)

Diminishing Role of Topology

- Shift to general links
  - DMA, enabling non-blocking ops
    - » Buffered by system at destination until recv
  - Store&forward routing
- Diminishing role of topology
  - Any-to-any pipelined routing
  - node-network interface dominates communication time

$$H \times (T_0 + n/B)$$

vs

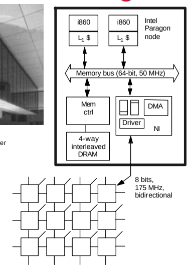
$$T0 + H\Delta + n/B$$

  - Simplifies programming
  - Allows richer design space
    - » grids vs hypercubes

Intel iPSC/1 -> iPSC/2 -> iPSC/860

Example Intel Paragon

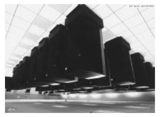
Sandia's Intel Paragon XP/S-based Supercomputer

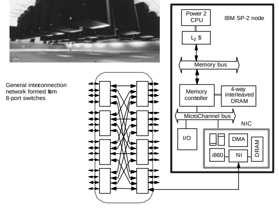


2D grid network with processing node attached to every switch

Building on the mainstream: IBM SP-2

- Made out of essentially complete RS6000 workstations
- Network interface integrated in I/O bus (bw limited by I/O bus)





FTC.W99 55

Berkeley NOW

- 100 Sun Ultra2 workstations
- Inteligent network interface
  - proc + mem
- Myrinet Network
  - 160 MB/s per link
  - 300 ns per hop

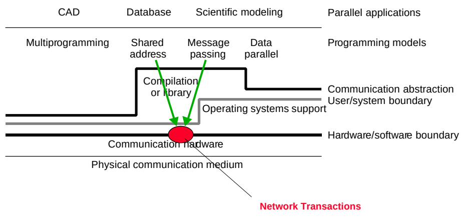
FTC.W99 56

Toward Architectural Convergence

- Evolution and role of software have blurred boundary
  - Send/recv supported on SAS machines via buffers
  - Can construct global address space on MP (GA -> P | LA)
  - Page-based (or finer-grained) shared virtual memory
- Hardware organization converging too
  - Tighter NI integration even for MP (low-latency, high-bandwidth)
  - Hardware SAS passes messages
- 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

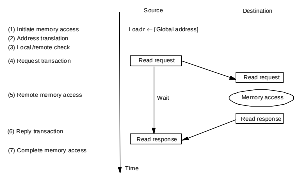
FTC.W99 57

Programming Models Realized by Protocols



FTC.W99 58

Shared Address Space Abstraction



- Fundamentally a two-way request/response protocol
  - writes have an acknowledgement
- Issues
  - fixed or variable length (bulk) transfers
  - remote virtual or physical address, where is action performed?
  - deadlock avoidance and input buffer full
- coherent? consistent?

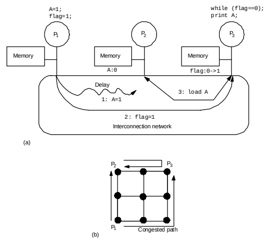
FTC.W99 59

The Fetch Deadlock Problem

- Even if a node cannot issue a request, it must sink network transactions.
- Incoming transaction may be a request, which will generate a response.
- Closed system (finite buffering)

FTC.W99 60

Consistency



- write-atomicity violated without caching

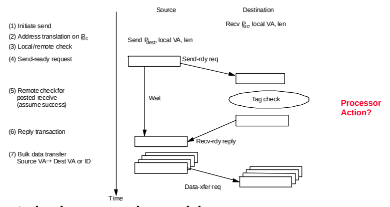
Key Properties of Shared Address Abstraction

- Source and destination data addresses are specified by the source of the request
  - a degree of logical coupling and trust
- no storage logically “outside the address space”
  - » may employ temporary buffers for transport
- Operations are fundamentally request response
- Remote operation can be performed on remote memory
  - logically does not require intervention of the remote processor

Message passing

- Bulk transfers
- Complex synchronization semantics
  - more complex protocols
  - More complex action
- Synchronous
  - Send completes after matching recv and source data sent
  - Receive completes after data transfer complete from matching send
- Asynchronous
  - Send completes after send buffer may be reused

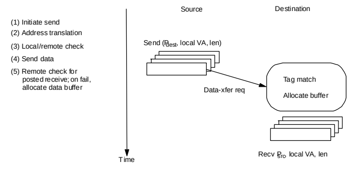
Synchronous Message Passing



Source
Destination
(1) Initiate send
(2) Address translation on P_{src}
(3) Local/remote check
(4) Send-ready request
(5) Remote check for posted receive (assume success)
(6) Reply transaction
(7) Bulk data transfer Source VA → Dest VA or ID
Recv P_{src}, local VA, len
Send P_{dest}, local VA, len
Send-rdy req
Wait
Tag check
Processor Action?
Recv-rdy reply
Data-xfer req
Time

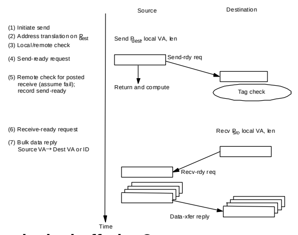
- Constrained programming model.
- Deterministic!
- Destination contention very limited.

Asynch. Message Passing: Optimistic



- More powerful programming model
- Wildcard receive => non-deterministic
- Storage required within msg layer?

Asynch. Msg Passing: Conservative



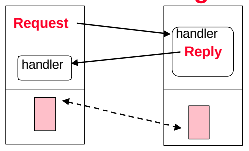
Source
Destination
(1) Initiate send
(2) Address translation on P_{dest}
(3) Local/remote check
(4) Send-ready request
(5) Remote check for posted receive (assume fail); record send-ready
(6) Receive-ready request
(7) Bulk data reply Source VA → Dest VA or ID
Send P_{dest}, local VA, len
Send-rdy req
Return and compute
Tag check
Recv P_{src}, local VA, len
Recv-rdy req
Data-xfer reply
Time

- Where is the buffering?
- Contention control? Receiver initiated protocol?
- Short message optimizations

Key Features of Msg Passing Abstraction

- Source knows send data address, dest. knows receive data address
  - after handshake they both know both
- Arbitrary storage "outside the local address spaces"
  - may post many sends before any receives
  - non-blocking asynchronous sends reduces the requirement to an arbitrary number of descriptors
    - » fine print says these are limited too
- Fundamentally a 3-phase transaction
  - includes a request / response
  - can use optimisitic 1-phase in limited "Safe" cases
    - » credit scheme

Active Messages



- User-level analog of network transaction
  - transfer data packet and invoke handler to extract it from the network and integrate with on-going computation
- Request/Reply
- Event notification: interrupts, polling, events?
- May also perform memory-to-memory transfer

Common Challenges

- Input buffer overflow
  - N-1 queue over-commitment => must slow sources
  - reserve space per source (credit)
    - » when available for reuse?
      - Ack or Higher level
  - Refuse input when full
    - » backpressure in reliable network
    - » tree saturation
    - » deadlock free
    - » what happens to traffic not bound for congested dest?
  - Reserve ack back channel
  - drop packets
  - Utilize higher-level semantics of programming model

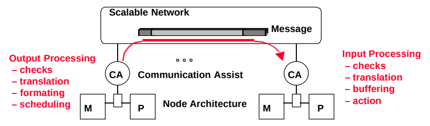
Challenges (cont)

- Fetch Deadlock
  - For network to remain deadlock free, nodes must continue accepting messages, even when cannot source msgs
  - what if incoming transaction is a request?
    - » Each may generate a response, which cannot be sent!
    - » What happens when internal buffering is full?
- logically independent request/reply networks
  - physical networks
  - virtual channels with separate input/output queues
- bound requests and reserve input buffer space
  - K(P-1) requests + K responses per node
  - service discipline to avoid fetch deadlock?
- NACK on input buffer full
  - NACK delivery?

Challenges in Realizing Prog. Models in the Large

- One-way transfer of information
- No global knowledge, nor global control
  - barriers, scans, reduce, global-OR give fuzzy global state
- Very large number of concurrent transactions
- Management of input buffer resources
  - many sources can issue a request and over-commit destination before any see the effect
- Latency is large enough that you are tempted to "take risks"
  - optimistic protocols
  - large transfers
  - dynamic allocation
- Many many more degrees of freedom in design and engineering of these system

Network Transaction Processing



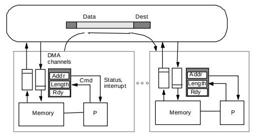
- Key Design Issue:
- How much interpretation of the message?
- How much dedicated processing in the Comm. Assist?

Spectrum of Designs

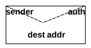
- None: Physical bit stream
  - blind, physical DMA — nCUBE, iPSC, . . .
- User/System
  - User-level port — CM-5, *T
  - User-level handler — J-Machine, Monsoon, . . .
- Remote virtual address
  - Processing, translation — Paragon, Meiko CS-2
- Global physical address
  - Proc + Memory controller — RP3, BBN, T3D
- Cache-to-cache
  - Cache controller — Dash, KSR, Flash

Increasing HW Support, Specialization, Intrusiveness, Performance (???)

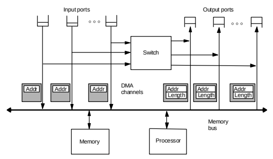
Net Transactions: Physical DMA



- DMA controlled by regs, generates interrupts
- Physical => OS initiates transfers
- Send-side
  - construct system "envelope" around user data in kernel area
- Receive
  - must receive into system buffer, since no interpretation in CA



nCUBE Network Interface

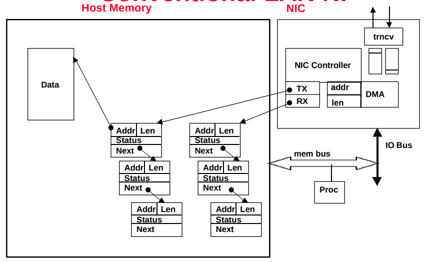


- independent DMA channel per link direction
  - leave input buffers always open
  - segmented messages
- routing interprets envelope
  - dimension-order routing on hypercube
  - bit-serial with 36 bit cut-through

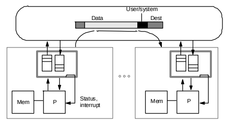
Os	16 ins	260 cy	13 us
Or	18	200 cy	15 us

- includes interrupt

Conventional LAN NI

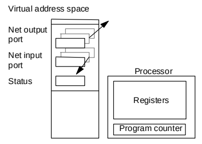


User Level Ports



- initiate transaction at user level
- deliver to user without OS intervention
- network port in user space
- User/system flag in envelope
  - protection check, translation, routing, media access in src CA
  - user/sys check in dest CA, interrupt on system

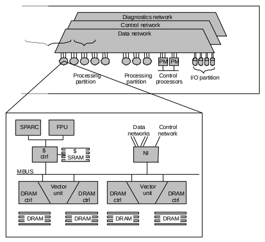
User Level Network ports



- Appears to user as logical message queues plus status
- What happens if no user pop?

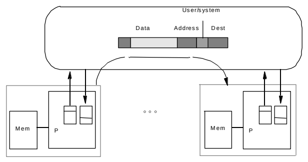
Example: CM-5

- Input and output FIFO for each network
- 2 data networks
- tag per message
  - index NI mapping table
- context switching?
- *T integrated NI on chip
- iWARP also



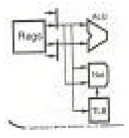
Os	50 cy	1.5 us
Or	53 cy	1.6 us
interrupt		10us

User Level Handlers



- Hardware support to vector to address specified in message
  - message ports in registers

J-Machine: Msg-Driven Processor



- Each node a small msg driven processor
- HW support to queue msgs and dispatch to msg handler task

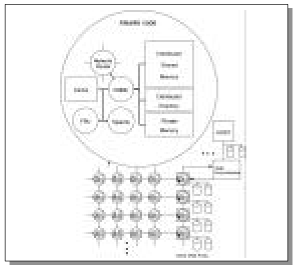
Communication Comparison

- Message passing (active messages)
  - interrupts (int-mp)
  - polling (poll-mp)
  - bulk transfer (bulk)
- Shared memory (sequential consistency)
  - without prefetching (sm)
  - with prefetching (pre-sm)

Motivation

- Comparison over a range of parameters
  - latency and bandwidth emulation
  - hand-optimized code for each mechanism
    - » 5 versions of 4 applications

The Alewife Multiprocessor



Alewife Mechanisms

- Int-mp -- 100-200 cycles Send/Rec ovrhd
- Poll-mp -- saves 50-170 cycles Rec ovrhd
- Bulk -- gather/scatter
- Sm -- 42--63 cycles + 1.6 cycles/hop
- Pre-sm -- 2 cycles, 16 entry buffer

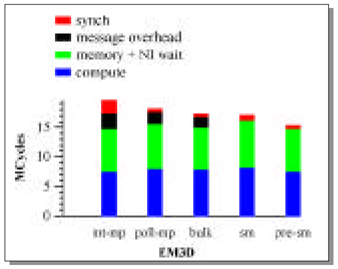
Applications

- Irregular Computations
- Little data re-use
- Data driven

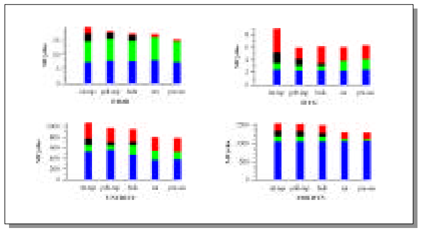
Application Descriptions

EM3D	3D electromagnetic wave
ICCG	irreg sparse matrix solver
Unstruc	3D fluid flow
Moldyn	molecular dynamics

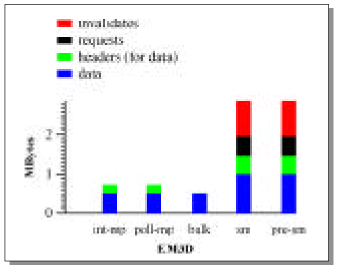
Performance Breakdown



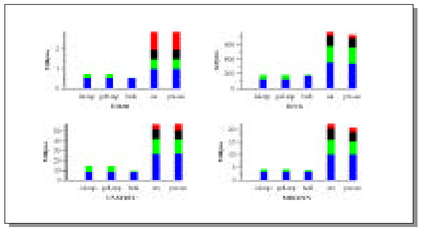
Performance Summary



Traffic Breakdown

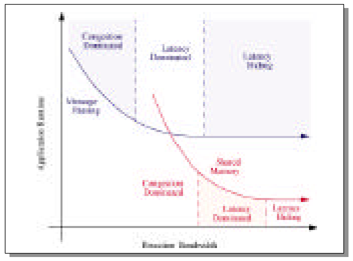


Traffic Summary



FTC.W99 91

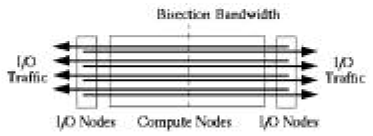
Effects of Bandwidth



FTC.W99 92

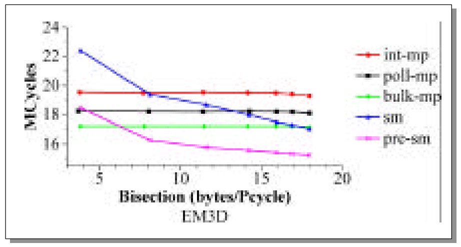
Bandwidth Emulation

- Lower bisection by introducing cross-traffic



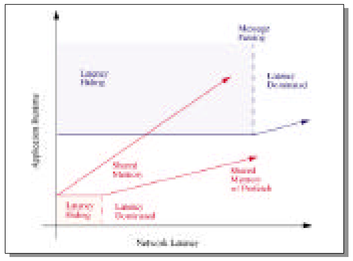
FTC.W99 93

Sensitivity to Bisection



FTC.W99 94

Effects of Latency



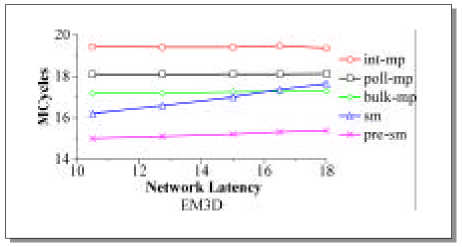
FTC.W99 95

Latency Emulation

- Clock variation
  - processor has tunable clock
  - network is asynchronous
  - results in variations in *relative* latency
- Context switch on miss
  - add delay

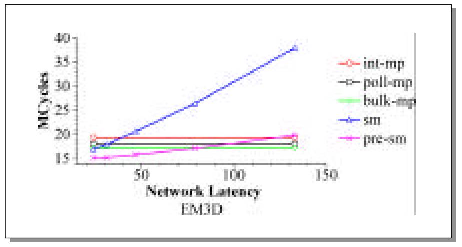
FTC.W99 96

Sensitivity to Latency



FTC.W99 97

Sensitivity to Higher Latencies



FTC.W99 98

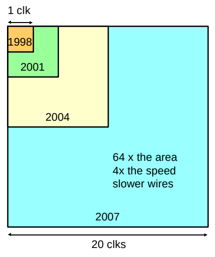
Communication Comparison Summary

- Low overhead in shared memory performs well even with:
  - irregular, data-driven applications
  - little re-use
- Bisection and latency can cause crossovers

FTC.W99 99

Future Technology

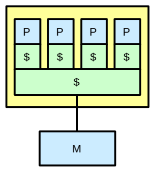
- Technology changes the cost and performance of computer elements in a non-uniform manner
  - logic and arithmetic is becoming plentiful and cheap
  - wires are becoming slow and scarce
- This changes the tradeoffs between alternative architectures
  - superscalar doesn't scale well
    - » global control and data
- So what will the architectures of the future be?



FTC.W99 100

Single-Chip Multiprocessors

- Build a multiprocessor on a single chip
  - linear increase in *peak* performance
  - advantage of fast interaction between processors
- But
  - memory bandwidth problem multiplied



FTC.W99 101

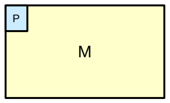
Exploiting fine-grain threads

- Where will the parallelism come from to keep all of these processors busy?
  - ILP - limited to about 5
  - Outer-loop parallelism
    - » e.g., domain decomposition
    - » requires big problems to get lots of parallelism
- Fine threads
  - make communication and synchronization very fast (1 cycle)
  - break the problem into smaller pieces
  - more parallelism

FTC.W99 102

Processor with DRAM (PIM)

- Put the processor and the main memory on a single chip
  - much lower memory latency
  - much higher memory bandwidth
- But
  - need to build systems with more than one chip



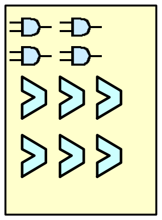
64Mb SDRAM Chip
Internal - 128 512K subarrays
4 bits per subarray each 10ns
51.2 Gb/s

External - 8 bits at 10ns, 800Mb/s

1 Integer processor ~ 100KBytes DRAM
1 FP processor ~ 500KBytes DRAM

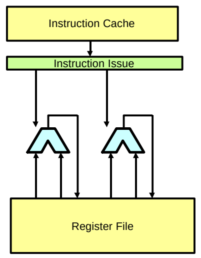
Reconfigurable processors

- Adapt the processor to the application
  - special function units
  - special wiring between function units
- Builds on FPGA technology
  - FPGAs are inefficient
    - » a multiplier built from an FPGA is about 100x larger and 10x slower than a custom multiplier.
  - Need to raise the granularity
    - » configure ALUs, or whole processors
  - Memory and communication are usually the bottleneck
    - » not addressed by configuring a lot of ALUs



EPIC - explicit (instruction-level) parallelism aka VLIW

- Compiler schedules instructions
- Encodes dependencies explicitly
  - saves having the hardware repeatedly rediscover them
- Support speculation
  - speculative load
  - branch prediction
- Really need to make communication explicit too
  - still has global registers and global instruction issue



Summary

- Parallelism is inevitable
  - ILP
  - Medium
  - Massive
- Commodity forces
  - SMPs
  - NOWs, CLUMPs
- Technological trends
  - MP chips
  - Intelligent memory