Parallel Programs

Why Bother with Programs?

It's what runs on the machines we design

- 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

- new degrees of freedom
- greater penalties for mismatch between program and architecture

Important for Whom?

Algorithm designers

- designing algorithms that will run well on real systems

Programmers

- understanding key issues and obtaining best performance

Architects

- understand workloads, interactions, important degrees of freedom

Next Three Sections of Class: Software

1. Parallel programs
   - process of parallelization
   - what parallel programs look like in major programming models
2. Programming for performance
   - key performance issues and architectural interactions
3. Workload-driven architectural evaluation
   - beneficial for architects and for users in procuring machines

Unlike on sequential systems, can't take workload for granted

- software base not mature, so need to open the box

Let's begin with parallel programs ...

Outline

Motivating Problems

Steps in creating a parallel program

What parallel programs look like: Grid solver example

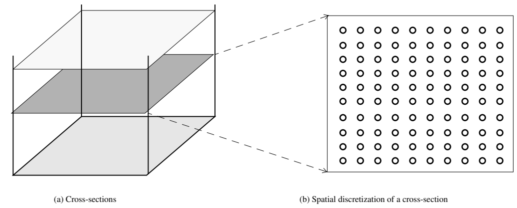
- what primitives must a system support?

Performance issues and architectural implications

- hardware-software interactions?
- what can/should architecture do?
- understanding parallel programs as workloads?

Case studies

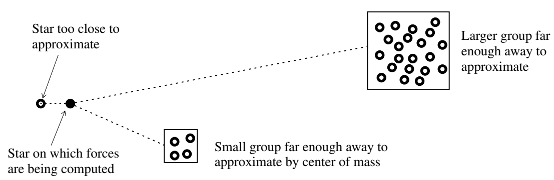
Simulating Ocean Currents



- Model as two-dimensional grids
- Discretize in space and time
- Many different computations per time step
  - set up and solve equations

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: $G\frac{m_1 m_2}{r^2}$



- 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

Creating a Parallel Program

Pieces of the job:

- Identify work that can be done in parallel
- Partition work and perhaps data among processes
- Manage data access, communication and synchronization

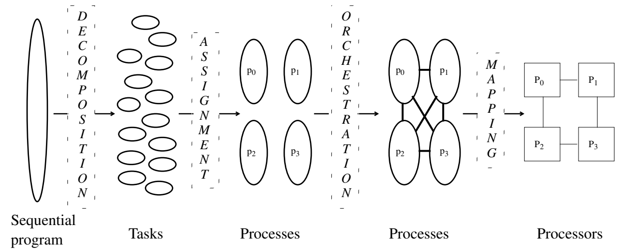
Main goal: Speedup

$$Speedup(p) = \frac{Performance(p)}{Performance(1)}$$

For a fixed problem:

$$Speedup(p) = \frac{Time(1)}{Time(p)}$$

Creating a Parallel Program



- *Task*: piece of undecomposed work in parallel computation
- *Process (thread)*: abstract entity that performs tasks
- *Processor*: physical engine on which process executes

4 steps: Decomposition, Assignment, Orchestration, Mapping

- Done by programmer or system software (compiler, runtime, ...)

Decomposition

Break up computation into tasks that are divided among procs

- Tasks may become available dynamically
- No. of available tasks may vary with time
- Identify concurrency and decide level at which to exploit it

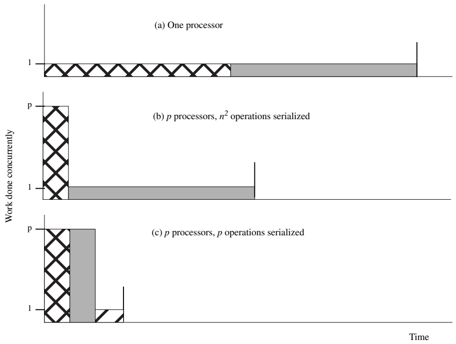
Goal: Enough tasks to keep procs busy, but not too many

- no. of tasks available at a time is upper bound on achievable speedup

Decomposition: Amdahl's Law

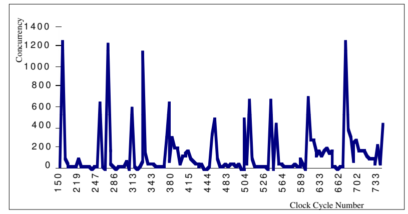
- If fraction s of seq execution is inherently serial, speedup <= 1/s
- Example: 2-phase calculation
  - sweep over n-by-n grid and compute
  - sweep again and add each value to global sum
- Time for first phase = n^2/p
- Second phase serialized at global variable, so time = n^2
- Speedup <= $\frac{2n^2}{\frac{n^2}{p}+n^2}$ or $\frac{2p}{p+1}$, 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 $n^2/p + n^2/p + p$, and speedup at best $p \times \frac{2n^2}{2n^2 + p^2}$

Pictorial Depiction



Concurrency Profiles

- Cannot usually divide into serial and parallel part



- Area under curve is total work done, or time with 1 proc
- Horizontal extent is lower bound on time (infinite processors)
- Speedup is the ratio. $Speedup(p) \le \frac{\sum_{k=1}^{\infty} f_k k}{\sum_{k=1}^{\infty} f_k \left\lceil \frac{k}{p} \right\rceil}$, base case $\frac{1}{s + \frac{1-s}{p}}$

Assignment

Determine mechanism to divide work up among processes

- Together with decomposition, also called *partitioning*
- Balance workload, reduce communication and management cost
- *Usually* independent of architecture or prog model

Structural approaches usually work well

- parallel loops or loop nests
- dynamic tasking

As programmers, we worry about *partitioning* first

As architects, we assume program does reasonable job

Orchestration

- Naming
- Communication structure
- Synchronization
- Also, temporal scheduling of tasks

Goals

- Reduce cost of comm and synch as seen by processors
- Preserve locality of data reference
- Schedule tasks to satisfy depedences early
- Reduce overhead of parallelism management

Closest to architecture; choices made depend most on

- comm abstraction, and efficiency of primitives

Mapping

Two aspects of mapping:

- which processes will run on same processor
  - if more processes than processors
- which process runs on which particular processor
  - mapping to a network topology

Not a major issue on modern multiprocessors

- number of procs; overhead may dominate; multiprogramming
- more attention to decomposition, assignment, orchestration

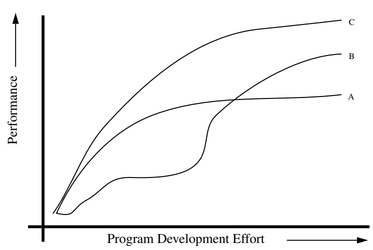
Usually adopt the view: process <-> processor

- multiprogramming, dynamic resource management?

High-level Goals

- High performance
- But low resource usage and development effort

Implications for algorithm designers and architects



Computation v. Data-oriented Views

Above view is centered around computation

- computation is decomposed and assigned (partitioned)

Partitioning Data is often a natural view too

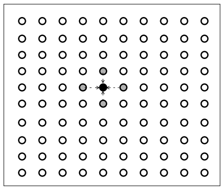
- computation follows data: *owner computes*
- grid example, High Performance Fortran (HPF)

But not general enough

- retain computation-centric view
- data access and communication is part of orchestration

What Parallel Programs Look Like

Grid Solver Example



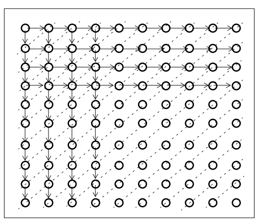
$A[i,j] = 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] + A[i,j+1] + A[i+1,j])$

- Simplified version of solver in Ocean simulation
- Gauss-Seidel iterations to convergence
  - accumulate partial sums into global sum at end of every iteration

```
int n;                          /* size of matrix: n-by-n elements*/
float **A, diff = 0;

main()
begin
  read(n) ;                     /* read input parameter: matrix size*/
  A ← malloc (a 2-d array of size n+2 by n+2 doubles);
  initialize(A);                /* initialize the matrix A somehow */
  Gauss-Seidel (A);             /* call the routine to solve equation*/
end main

procedure Gauss-Seidel(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 non-border 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
```

Decomposition

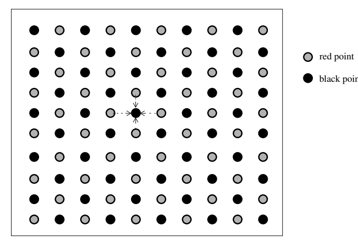
- Simple way to identify concurrency is to look at loop iterations
  - if not enough, then look further
- Not much concurrency here at this level
- Examine fundamental dependences



- Concurrency O(n) along anti-diagonals, serialization O(n)
- pt-to-pt synch, restructure loops, but load imbalance and too much synch

Exploit Application Knowledge

- Reorder grid traversal: red-black ordering



- The red and black traversals are each fully parallel: $\frac{n^2}{2}$
- Global synch between them is conservative
- Ocean uses red-black, we use simpler asynchronous one

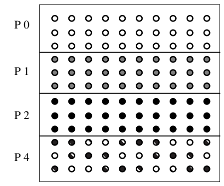
Decomposition Only

```
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
- to decompose into rows, make line 18 loop sequential; degree n
- assignment left to the system

Assignment

- Static assignments (given 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, $i+P$, and so on



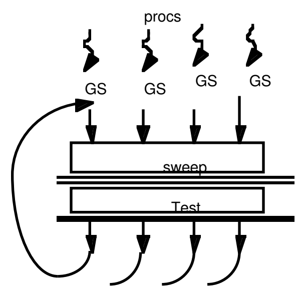
- Dynamic assignment
  - get a row index, work on the row, get a new row, and so on
- Static assignment into rows reduces the concurrency

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,*];
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 += 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)

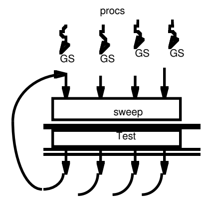


- assignment controlled by values of variables used as loop bounds

```
int n, nprocs;          /* matrix dimension and number of processors to be used */
float **A, diff;        /*A is global (shared) array representing the grid */
                        /* diff is global (shared) maximum difference in current sweep */
LOCKDEC(diff_lock);  /* declaration of lock to enforce mutual exclusion */
BARDEC (bar1);          /* barrier declaration for global synchronization between sweeps*/

main()
begin
  read(n); read(nprocs); /* read input matrix size and number of processes*/
  A ← G_MALLOC (a two-dimensional array of size n+2 by n+2 doubles);
  initialize(A);                          /* initialize A in an unspecified way*/
  CREATE (nprocs-1, Gauss-Seidel, A);
  Gauss-Seidel(A);          /* main process becomes a worker too*/
  WAIT_FOR_END;             /* wait for all child processes created to terminate */
end main

procedure Gauss-Seidel(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) */
 for i ← mymin to mymax do /* for each of my rows */
      for j ← 1 to n do      /* for all 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 have got 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
```

Notes on SAS Program

- SPMD



- assignment controlled by values of variables used as loop bounds
- unique pid per process
- done condition evaluated redundantly by all
- code that does the update identical to sequential program
- each process has private mydiff
- accumulations into shared diff have to be mutually exclusive

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:

P1	P2	
r1 ← diff		{P1 gets 0 in its r1}
	r1 ← diff	{P2 also gets 0}
r1 ← r1+r2		{P1 sets its r1 to 1}
	r1 ← r1+r2	{P2 sets its r1 to 1}
diff ← r1		{P1 sets cell_cost to 1}
	diff ← r1	{P2 also sets cell_cost to 1}

- Problem. Need the operations to be atomic (mutually exclusive)

Mutual Exclusion

Provided by LOCK-UNLOCK around critical section

- implementation of LOCK/UNLOCK must guarantee mutual excl.

Can lead to significant serialization if contended

- especially since expect non-local accesses in critical section

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
- conservative form of preserving dependences, but easy to use

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 to problem state	apply results to problem state	apply results to problem state
BARRIER (name, nprocs)	**BARRIER (name, nprocs)**	**BARRIER (name, nprocs)**

WAIT_FOR_END (nprocs-1)

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: flags (ordinary variables)

```
              P1                                          P2
a: while (flag is 0) do nothing;
                                                  b: flag = 1;
```

- *busy-waiting* or *spinning*

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

Message Passing Grid Solver

- Cannot declare A to be shared array any more
- Need to compose it logically from per-process private arrays
  - process assigned a set of rows allocates them locally
- Transfers of entire rows between traversals, so not asynch
- Program structurally similar, but data access/naming very different

```
1. int pid, n, nprocs; /* process id, matrix dimension and number of processors to be used */
2. float **myA;
3. main()
4. begin
5.   read(n);  read(nprocs);      /* read input matrix size and number of processes*/
8a.  CREATE (nprocs-1 processes that start at procedure Gauss-Seidel);
8b.  Gauss-Seidel();              /* main process becomes a worker too*/
8c.  WAIT_FOR_END;                /* wait for all child processes created to terminate */
9. end main

10. procedure Gauss-Seidel()
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 (pi = 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,*] */
17.  for i ← 1 to n' do                   /* for each of my rows */
18.      for j ← 1 to n do                /* for all 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 obtain 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(mydiff,sizeof(float),0,DONE);
25d. else
25e.     for i ← 1 to nprocs-1 do  /* for each of my rows */
25f.         RECEIVE(tempdiff,sizeof(float),*,DONE);
25g.         mydiff += tempdiff;    /* accumulate into total */
25h.     endfor
25i.     for i ← 1 to nprocs-1 do  /* for each of my rows */
25j.         SEND(done,sizeof(int),i,DONE);
25k.     endfor
25l. endif
26.      if (mydiff/(n*n) < TOL) then done = 1;
```

Notes on Message Passing Program

- Use of ghost rows
- Receive does not transfer data, send does
  - unlike SAS which is usually receiver-initiated
- Communication done at beginning of iteration, so no asynchrony
- Communication in whole rows, not element at a time
- Core similar, but indices 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;
26. endif
25k. BROADCAST(0,done,sizeof(int),DONE);
```

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
    - 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**

Requirements for performance are another story