Fall 2023 CS 462: OpenMP and Parallel Software

Gregory D. Peterson
gdp@utk.edu

Min H. Kao Dept.
of Electrical Engineering
and Computer Science

September 14, 2023

THE UNIVERSITY OF TENNESSEE KNOXVILLE

1

Last Class on OpenMP

- Cache coherence and consistency
- Synchronization
- False sharing

2

Evolving Challenges with Programs

- They're what runs on the machines we design
  - Helps clarify best design decisions
  - Helps evaluate systems tradeoffs
- Understanding programs 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

3

Three Concerns for Parallel 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; evolves with architectures for performance
  - So need to open the box
- Let's begin with parallel programs ...

4

Roadmap for Today

- Motivating Problems (application case studies)
- Steps in creating a parallel program
- What a simple parallel program looks like
  - In the three major programming models
  - Ehat primitives must a system support?
- *Later*: Performance issues and architectural interactions

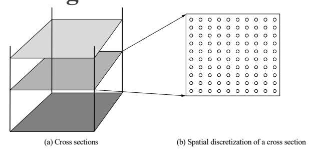
5

Motivating Problems

- Simulating Ocean Currents
  - Regular structure, scientific computing
- Simulating the Evolution of Galaxies
  - Irregular structure, scientific computing
- Rendering Scenes by Ray Tracing
  - Irregular structure, computer graphics
- Data Mining
  - Irregular structure, information processing

6

Simulating Ocean Currents

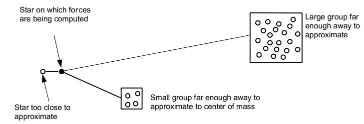


- Model as two-dimensional grids
- Discretize in space and time
  - finer spatial and temporal resolution => greater accuracy
- Many different computations per time step
  - set up and solve equations
- Concurrency across and within grid computations

7

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, lots of concurrency across stars within one

8

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
- Abundant concurrency

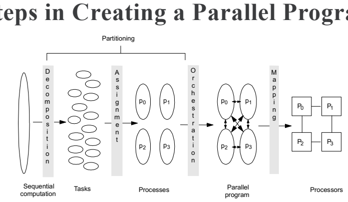
9

Creating a Parallel Program

- Assumption: Sequential algorithm is given
  - Sometimes need very different algorithm, but beyond scope
- 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
  - *Note*: work includes computation, data access and I/O

10

Steps in Creating a Parallel Program



- 4 steps: Decomposition, Assignment, Orchestration, Mapping
  - Done by programmer or system software (compiler, runtime, ...)
  - Issues are the same, so assume programmer does it all explicitly

11

Some Important Concepts

- *Task*:
  - Arbitrary piece of undecomposed 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
    - first write program in terms of processes, then map to processors

12

Decomposition

- Break up computation into tasks to be divided among processes
  - Tasks may become available dynamically
  - Number of available tasks may vary with time
- Identify concurrency and decide level at which to exploit it
- Goal: Enough tasks to keep processes busy, but not too many
  - Number of tasks available at a time is upper bound on achievable speedup

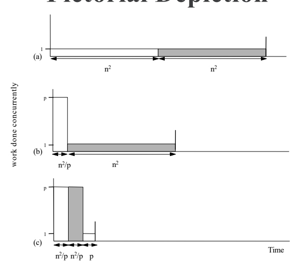
13

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 <= $\frac{2n^2}{\frac{n^2}{p} + n^2}$ 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 $n^2/p + n^2/p + p$, and speedup at best $\frac{2n^2p}{2n^2 + p^2}$

14

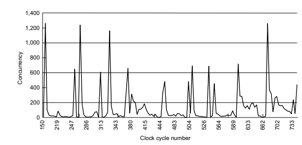
Pictorial Depiction



15

Concurrency Profile

- Cannot usually divide into serial and parallel part



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

16

Assignment

- Specifying mechanism to divide work up among processes
  - E.g. which process computes forces on which stars, or which rays
  - Together with decomposition, also called *partitioning*
  - Balance workload, reduce communication and management cost
- Structured approaches usually work well
  - Code inspection (parallel loops) or understanding of application
  - Well-known heuristics
  - *Static* versus *dynamic* assignment
- As programmers, we worry about partitioning first
  - *Usually* independent of architecture or programming model
  - But cost and complexity of using primitives may affect decisions
- As architects, we assume program does reasonable job of it

17

Orchestration

- Naming data
- Structuring communication
- Synchronization
- Organizing data structures and scheduling tasks temporally
- Goals
  - Reduce cost of communication and synch. as seen by processors
  - Reserve locality of data reference (incl. data structure organization)
  - Schedule tasks to satisfy dependences early
  - Reduce overhead of parallelism management
- Closest to architecture (and programming model & language)
  - Choices depend a lot on comm. abstraction, efficiency of primitives
  - Architects should provide appropriate primitives efficiently

18

Mapping

- After orchestration, already have parallel program
- Two aspects of mapping:
  - Which processes will run on same processor, if necessary
  - Which process runs on which particular processor
    - mapping to a network topology
- One extreme: *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
- Another extreme: complete resource management control to OS
  - OS uses the performance techniques we will discuss later
- Real world is between the two
  - User specifies desires in some aspects, system may ignore
- Usually adopt the view: process <-> processor

19

Parallelizing Computation vs. Data

- 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; data mining
- But not general enough
  - Distinction between computation and data stronger in many applications
    - Barnes-Hut, Raytrace
  - Retain computation-centric view
  - Data access and communication is part of orchestration

20

What Parallel Programs Look Like

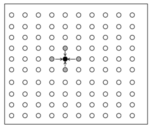
21

Parallelization of Example Program

- Motivating problems all lead to large, complex programs
- Examine a simplified version of a piece of Ocean simulation
  - Iterative equation solver
- Illustrate parallel program in low-level parallel language
  - C-like pseudocode with simple extensions for parallelism
  - Expose basic comm. and synch. primitives that must be supported
  - State of most real parallel programming today

22

Grid Solver Example



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])

- Simplified version of solver in Ocean simulation
- 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, and diff. from prev. value computed
  - accumulate partial diffs into global diff at end of every sweep
  - check if error has converged (to within a tolerance parameter)
  - if so, exit solver; if not, do another sweep

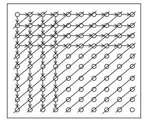
23

```
int n;                                  /*size of matrix: (n + 2-by-n + 2) 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*/
  Solve (A);                            /*call the routine to solve equation*/
end main

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
```

24

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, ignoring loop structure

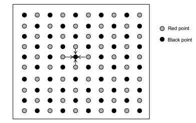


- Concurrency *O(n)* along anti-diagonals, serialization *O(n)* along diagonal
- Retain loop structure, use pt-to-pt synch; Problem: too many synch ops
- Restructure loops, use global synch; imbalance and too much synch

25

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
  - sequential order same as original, parallel program *nondeterministic*

26

Decomposition Only

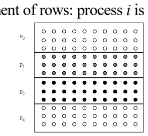
```
15. while (!done) do                       /*a sequential loop*/
16.   diff = 0;
17.   for_all i ← 1 to n do                /*a parallel loop nest*/
18.     for_all j ← 1 to n do
19.       temp = A[i,j];
20.       A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21.         A[i,j+1] + A[i+1,j]);
22.       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
- `for_all` leaves assignment left to system
  - but implicit global synch. at end of `for_all` loop

27

Assignment

- Static assignments (given decomposition into rows)
  - block assignment of rows: Row i is assigned to process $\lfloor \frac{i}{p} \rfloor$
  - cyclic assignment of rows: process i is assigned rows i, $i+p$, etc



- Dynamic assignment
  - get a row index, work on the row, get a new row, and so on
- Static assignment into rows reduces concurrency (from n to p)
  - block assign. reduces communication by keeping adjacent rows together
- Let's dig into orchestration under three programming models

28

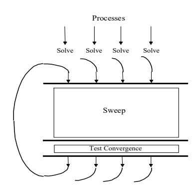
Data Parallel Solver

```
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 += 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
```

29

Shared Address Space Solver

Single Program Multiple Data (SPMD)



Assignment controlled by values of variables used as loop bounds

30

```
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.      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 over all diagonal elements*/
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
```

31

Notes on SAS Program

- SPMD: not lockstep or even 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?

32

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}
```

- Need the sets of operations to be atomic (mutually exclusive)

33

Mutual Exclusion

- Provided by LOCK-UNLOCK around *critical section*
  - Set of operations we want to execute atomically
  - Implementation of LOCK/UNLOCK must guarantee mutual excl.
- Can lead to significant serialization if contended
  - Especially since expect non-local accesses in critical section
  - Another reason to use private mydiff for partial accumulation

34

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)

35

Correctness in Grid Solver Program

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

	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 ...

36