Basics of Parallelization

- Dependence analysis
- Synchronization
  - Events
  - Mutual exclusion
- Parallelism patterns

1

When can 2 statements execute in parallel?

S1 and S2 can execute in parallel

iff

there are no dependences between S1 and S2

- true dependences
- anti-dependences
- output dependences

Some dependences can be removed.

2

Types of Dependences

- True (flow) dependence – RAW
- Anti-dependence – WAR
- Output dependence – WAW

3

Loop-Carried Dependence

- A loop-carried dependence is a dependence that is present between statements in two different iterations of a loop
- A loop-independent dependence is a dependence between two statements in the same loop
- Loop-carried dependences limit loop iteration parallelization

4

Synchronization

- Used to enforce dependences
- Control the ordering of events on different processors
  - Events – signal(x) and wait(x)
  - Fork-Join or barrier synchronization (global)
  - Mutual exclusion/critical sections

5

Eliminating Dependences

- Privatization or scalar expansion
- Reduction (common pattern)

6

Example: Scalar Expansion or Privatization

```
for (I = 0; I < 100; I++)
      T = A[I];
      A[I] = B[I];
      B[I] = T;
```

Loop-carried anti-dependence on T
Eliminate by converting T into an array or by making T private to each loop iteration

7

Example: Scalar Expansion

```
for (I = 0; I < 100; I++)
      T [I]= A[I];
      A[I] = B[I];
      B[I] = T[I];
```

Loop-carried anti-dependence eliminated

8

Removing Dependences: Reduction

```
sum = 0.0;
for( i=0; i<100; i++ ) sum += a[i];
```

- Loop-carried dependence on sum.
- Cannot be parallelized, but ...

9

Reduction (continued)

```
for( i=0; i<...; i++ ) sum[i] = 0.0;
fork();
for( j=…; j<…; j++ ) sum[i] += a[j];
join();
sum = 0.0;
for( i=0; i<...; i++ ) sum += sum[i];
```

Common pattern often with explicit support
e.g., sum = reduce (+, a, 0, 100)
CAVEAT: Operator must be commutative and associative

10

Steps in the Parallelization

- Decomposition into tasks
  - Expose concurrency
- Assignment to processes
  - Balancing load and maximizing locality
- Orchestration
  - Name and access data
  - Communicate (exchange) data
  - synchronization among processes
- Mapping
  - Assignment of processes to processors

11

Decomposition into Tasks

- Tasks may be
  - Identical computation
  - Different computation
  - Indeterminate size
- Tasks may be
  - Independent
  - Have non-trivial order

13

Decomposition into Tasks

- Conceptualize tasks and ordering as a task dependency DAG (for control dependency), along with a task interaction DAG (for data dependency)
  - Edges represent task serialization
  - Critical path – longest weighted path through graph (lower bound on parallel execution time)
- Measures of parallel performance: speedup, efficiency
- Tradeoff between
  - Degree of concurrency (number of tasks that can be processed in parallel)
  - Task granularity
  - Associated overheads

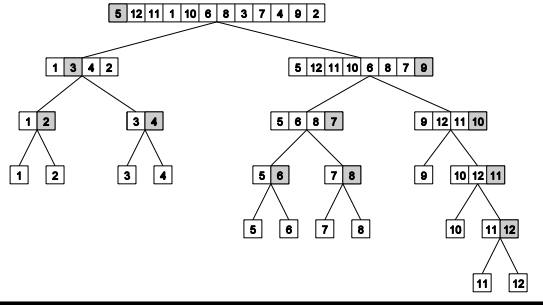
14

Recursive Decomposition

Suitable for problems solvable using divide-and-conquer

Example: Quicksort

1. Select a pivot
2. Partition set based on pivot
3. Recursively partition each subset in parallel



18

Patterns of Parallelism

- Decomposition views
  - Data (static) vs. recursive (dynamic) decomposition
  - Exploratory decomposition vs. speculative decomposition
    - Exploratory - Parallel formulation may perform different amounts of work resulting in super or sub-linear speedup
    - Speculative - Schedule tasks even when they may have dependencies
- Data parallelism: all processors do the same thing on different data.
  - Regular
  - Irregular
- Task parallelism: processors do different tasks or dynamically pick up data to compute on
  - Task queue
  - Pipelines

17

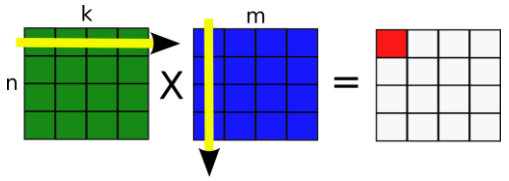
(Static) Data Parallelism

- Essential idea: each processor works on a different part of the data (usually in one or more arrays)
  - work partitioned based on "owner" computes rule, applied to either input, output, or intermediate data
- Regular or irregular data parallelism: using linear or non-linear indexing.
- Examples: MM (regular), SOR (regular), MD (irregular).

19

Matrix Multiplication

- Multiplication of two n by n matrices A and B into a third n by n matrix C



20

Matrix Multiply

```
for( i=0; i<n; i++ )
  for( j=0; j<n; j++ )
        c[i][j] = 0.0;
for( i=0; i<n; i++ )
  for( j=0; j<n; j++ )
        for( k=0; k<n; k++ )
              c[i][j] += a[i][k]*b[k][j];
```

21

Parallel Matrix Multiply

- No loop-carried dependences in i- or j-loop.
- Loop-carried dependence on k-loop.
- All i- and j-iterations can be run in parallel.

22

Parallel Matrix Multiply (contd.)

- If we have P processors, we can give n/P rows or columns to each processor.
- Or, we can divide the matrix in P squares, and give each processor one square.

23

SOR

- SOR implements a mathematical model for many natural phenomena, e.g., heat dissipation in a metal sheet
- Model is a partial differential equation
- Focus is on algorithm, not on derivation
- Discretized problem

24

Relaxation Algorithm

- For some number of iterations
  - for each internal grid point
    - compute average of its four neighbors
- Termination condition:
  - values at grid points change very little
  - (we will ignore this part in our example)

25

Discretized Problem Statement

```
/* Initialization */
for( i=0; i<n+1; i++ ) grid[i][0] = 0.0;
for( i=0; i<n+1; i++ ) grid[i][n+1] = 0.0;
for( j=0; j<n+1; j++ ) grid[0][j] = 1.0;
for( j=0; j<n+1; j++ ) grid[n+1][j] = 0.0;
for( i=1; i<n; i++ )
  for( j=1; j<n; j++ )
      grid[i][j] = 0.0;
```

26

Discretized Problem Statement

```
for some number of timesteps/iterations {
  for (i=1; i<n; i++ )
      for( j=1, j<n, j++ )
          temp[i][j] = 0.25 *
              ( grid[i-1][j] + grid[i+1][j]
              grid[i][j-1] + grid[i][j+1] );
  for( i=1; i<n; i++ )
      for( j=1; j<n; j++ )
          grid[i][j] = temp[i][j];
}
```

27

Parallel SOR

- No dependences between iterations of first (i,j) loop nest.
- No dependences between iterations of second (i,j) loop nest.
- Anti-dependence between first and second loop nest in the same timestep.
- True dependence between second loop nest and first loop nest of next timestep.

28

Parallel SOR Dependences

- First (i,j) loop nest can be parallelized.
- Second (i,j) loop nest can be parallelized.
- We must make processors wait at the end of each (i,j) loop nest.
- Natural synchronization: fork-join.

29

Parallel SOR Decomposition

- If we have P processors, we can give n/P rows or columns to each processor.
- Or, we can divide the array in P squares, and give each processor a square to compute.

30

Molecular Dynamics (MD)

- Simulation of a set of bodies under the influence of physical laws.
- Atoms, molecules, celestial bodies, ...
- Have same basic structure.

31

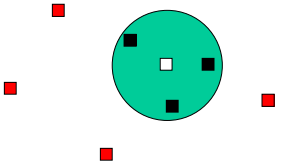
Molecular Dynamics (Skeleton)

```
for some number of timesteps {
    for all molecules i
        for all other molecules j
            force[i] += f( loc[i], loc[j] );
    for all molecules i
        loc[i] = g( loc[i], force[i] );
}
```

32

Molecular Dynamics (continued)

- To reduce amount of computation, account for interaction only with nearby molecules.



33

Molecular Dynamics (continued)

```
for some number of timesteps {
    for all molecules i
        for all nearby molecules j
            force[i] += f( loc[i], loc[j] );
    for all molecules i
        loc[i] = g( loc[i], force[i] );
}
```

34

Molecular Dynamics (continued)

```
for each molecule i
  number of nearby molecules count[i]
  array of indices of nearby molecules index[j]
   ( 0 <= j < count[i])
```

35

Molecular Dynamics (continued)

```
for some number of timesteps {
    for( i=0; i<num_mol; i++ )
        for( j=0; j<count[i]; j++ )
            force[i] += f(loc[i],loc[index[j]]);
    for( i=0; i<num_mol; i++ )
        loc[i] = g( loc[i], force[i] );
}
```

36

Molecular Dynamics (continued)

- No loop-carried dependence in first i-loop.
- Loop-carried dependence (reduction) in j-loop.
- No loop-carried dependence in second i-loop.
- True dependence between first and second i-loop.

37

Molecular Dynamics (continued)

- First i-loop can be parallelized.
- Second i-loop can be parallelized.
- Must make processors wait between loops.
- Natural synchronization: fork-join.

38

Molecular Dynamics (continued)

```
for some number of timesteps {
    for( i=0; i<num_mol; i++ )
        for( j=0; j<count[i]; j++ )
            force[i] += f(loc[i],loc[index[j]]);
    for( i=0; i<num_mol; i++ )
        loc[i] = g( loc[i], force[i] );
}
```

39

Irregular vs. regular data parallel

- In SOR, all arrays are accessed through linear expressions of the loop indices, known at compile time [regular].
- In MD, some arrays are accessed through non-linear expressions of the loop indices, some known only at runtime [irregular].

40

Irregular vs. regular data parallel

- No real differences in terms of parallelization (based on dependences)
- Will lead to fundamental differences in expressions of parallelism:
  - irregular difficult for parallelism based on data distribution
  - not difficult for parallelism based on iteration distribution.

41

Molecular Dynamics Decomposition

- Parallelization of first loop:
  - has a load balancing issue
  - some molecules have few/many neighbors
  - more sophisticated loop partitioning necessary

42

43

Patterns of Parallelism

- Decomposition views
  - Data (static) vs. recursive (dynamic) decomposition
  - Exploratory decomposition vs. speculative decomposition
    - Exploratory - Parallel formulation may perform different amounts of work resulting in super or sub-linear speedup
    - Speculative - Schedule tasks even when they may have dependencies
- Data parallelism: all processors do the same thing on different data.
  - Regular
  - Irregular
- Task parallelism: processors do different tasks or dynamically pick up data to compute on
  - Task queue
  - Pipelines

44

Task Parallelism

- Each process performs a different task.
- Two principal flavors:
  - pipelines
  - task queues
- Program Examples: PIPE (pipeline), TSP (task queue).

45

Pipeline

- Often occurs with image processing applications, where a number of images undergo a sequence of transformations.
- E.g., rendering, clipping, compression, etc.

46

Sequential Program

```
for( i=0; i<num_pic, read(in_pic[i]); i++ ) {
  int_pic_1[i] = trans1( in_pic[i] );
  int_pic_2[i] = trans2( int_pic_1[i]);
  int_pic_3[i] = trans3( int_pic_2[i]);
  out_pic[i] = trans4( int_pic_3[i]);
}
```

47

Parallelizing a Pipeline

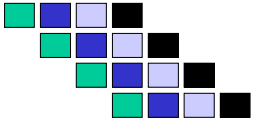
- For simplicity, assume we have 4 processors (i.e., equal to the number of transformations).
- Furthermore, assume we have a very large number of pictures (>> 4).

48

Sequential vs. Parallel Execution

- Sequential

- Parallel



(Color -- picture; horizontal line -- processor).

49

Parallelizing a Pipeline (part 1)

Processor 1:

```
for( i=0; i<num_pics, read(in_pic[i]); i++ ) {
  int_pic_1[i] = trans1( in_pic[i] );
  signal(event_1_2[i]);
}
```

50

Parallelizing a Pipeline (part 2)

Processor 2:

```
for( i=0; i<num_pics; i++ ) {
  wait( event_1_2[i] );
  int_pic_2[i] = trans2( int_pic_1[i] );
  signal(event_2_3[i] );
}
```

Same for processor 3

51

Parallelizing a Pipeline (part 3)

Processor 4:

```
for( i=0; i<num_pics; i++ ) {
  wait( event_3_4[i] );
  out_pic[i] = trans4( int_pic_3[i] );
}
```

52

Another Sequential Program

```
for( i=0; i<num_pic, read(in_pic); i++ ) {
  int_pic_1 = trans1( in_pic );
  int_pic_2 = trans2( int_pic_1);
  int_pic_3 = trans3( int_pic_2);
  out_pic = trans4( int_pic_3);
}
```

53

Can we use same parallelization?

Processor 2:

```
for( i=0; i<num_pics; i++ ) {
  wait( event_1_2[i] );
  int_pic_2 = trans1( int_pic_1 );
  signal(event_2_3[i] );
}
```

Same for processor 3

54

Can we use same parallelization?

- No, because of anti-dependence between stages, there is no parallelism
- Another example of privatization
- Costly in terms of memory

55

In-between Solution

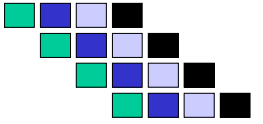
- Use n>1 buffers between stages.
- Block when buffers are full or empty

56

Perfect Pipeline

- Sequential

- Parallel



(Color -- picture; horizontal line -- processor).

57

Things are often not that perfect

- One stage takes more time than others
- Stages take a variable amount of time
- Extra buffers can provide some cushion against variability

58

Acknowledgements

Slides reflect content from Willy Zwaenepoel and from Grama/Gupta/Karypis/Kumar that accompany their corresponding course/textbooks and have been adapted to suit the content of this course

155