

## Basics of Parallelization

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

## Steps in the Parallelization

- Decomposition into tasks
- Assignment to processes
- Orchestration – communication of data, synchronization among processes

## Why is Parallel Computing Hard?

- Amdahl's law – insufficient available parallelism –
  - $Speedup = 1 / (fraction\_enhanced / speedup\_enhanced + (1 - fraction\_enhanced))$
- Overhead of communication and coordination
- Portability – knowledge of underlying architecture often required

## 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.

## Types of Dependences

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

## Loop-Carried Dependence

- A loop-carried dependence is a dependence that is present only if the statements occur in two different instances of a loop
- Otherwise, we call it a loop-independent dependence
- Loop-carried dependences limit loop iteration parallelization

## 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

## Example 1: Creating Parallelism by Enforcing Dependences

```
for( i=1; i<100; i++ ) {  
    a[i] = ...;  
    ...;  
    ... = a[i-1];  
}
```

- Loop-carried dependence, not parallelizable

## Synchronization Facility

- Suppose we had a set of primitives, signal(x) and wait(x).
- wait(x) blocks unless a signal(x) has occurred.
- signal(x) does not block, but causes a wait(x) to unblock, or causes a future wait(x) not to block.

## Example 1: Enforcing Dependencies (continued)

```
for( i=...; i<...; i++ ) {  
    a[i] = ...;  
    signal(e_a[i]);  
    ...;  
    wait(e_a[i-1]);  
    ... = a[i-1];  
}
```

## Example 1 (continued)

- Note that here it matters which iterations are assigned to which processor.
- It does not matter for correctness, but it matters for performance.
- Cyclic assignment is probably best.

## Example 2: Enforcing Dependences

```
for( i=0; i<100; i++ ) a[i] = f(i);  
x = g(a);  
for( i=0; i<100; i++ ) b[i] = x + h( a[i] );
```

- First loop can be run in parallel.
- Middle statement is sequential.
- Second loop can be run in parallel.

### Example 2 (continued)

- We will need to make parallel execution stop after first loop and resume at the beginning of the second loop.
- Two (standard) ways of doing that:
  - fork() - join()
  - barrier synchronization

### Fork-Join Synchronization

- fork() causes a number of processes to be created and to be run in parallel.
- join() causes all these processes to wait until all of them have executed a join().

### Example 2 (continued)

```
fork();
for( i=...; i<...; i++ ) a[i] = f(i);
join();
x = g(a);
fork();
for( i=...; i<...; i++ ) b[i] = x + h( a[i] );
join();
```

### Eliminating Dependences

- Privatization or scalar expansion
- Reduction (common pattern)

### 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

### 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

## Removing Dependences: Reduction

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

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

## 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

## Patterns of Parallelism

- Data parallelism: all processors do the same thing on different data.
  - Regular
  - Irregular
- Task parallelism: processors do different tasks.
  - Task queue
  - Pipelines

## Data Parallelism

- Essential idea: each processor works on a different part of the data (usually in one or more arrays).
- Regular or irregular data parallelism: using linear or non-linear indexing.
- Examples: MM (regular), SOR (regular), MD (irregular).

## Matrix Multiplication

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

## 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];
```

## 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.

## 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.

## 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 as in first lecture

## 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)

## 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;
```

## 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];
}
```

## 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.

## Parallel SOR (continued)

- 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.

## Parallel SOR (continued)

- 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.

## Molecular Dynamics (MD)

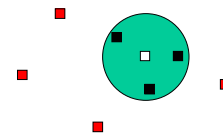
- Simulation of a set of bodies under the influence of physical laws.
- Atoms, molecules, celestial bodies, ...
- Have same basic structure.

## 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] );  
}
```

## Molecular Dynamics (continued)

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



## 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] );
}
```

## 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] )
```

## 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] );
}
```

## 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.

## 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.

## 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] );
}
```

### 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].

### 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.

### Molecular Dynamics (continued)

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