Basics of Parallelization

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

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

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

Eliminating Dependences

- Privatization or scalar expansion
- Reduction (common pattern)

Decomposition into Tasks

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

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

Mapping/Assignment to Processes

- Optimal load balance
- Minimum communication (maximum locality)
  - Map independent tasks to different processes
  - Minimize interaction between processes
  - Assign tasks on critical path to processes ASAP

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
- Alternative views
  - Data vs. recursive decomposition (static task generation)
  - Exploratory decomposition vs. speculative decomposition (dynamic task generation)
    - 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

- 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).

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)

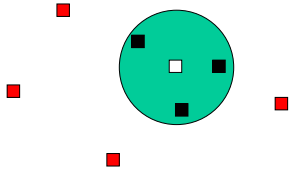
- Simulation of a set of bodies under the influence of physical laws.
- Atoms, molecules, celestial bodies, ...
- Have same basic structure.

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

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.

E.g. Molecular Dynamics

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

Task Parallelism

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

Pipeline

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

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]);
}
```

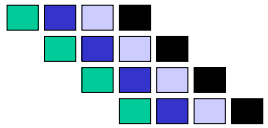
Parallelizing a Pipeline

- 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).

Sequential vs. Parallel Execution

- Sequential

- Parallel



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

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]);
}
```

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

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] );
}
```

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);
}
```

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

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

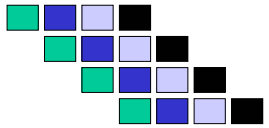
In-between Solution

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

Perfect Pipeline

- Sequential

- Parallel



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

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

TSP (Traveling Salesman)

- Goal:
  - given a list of cities, a matrix of distances between them, and a starting city,
  - find the shortest tour in which all cities are visited exactly once.
- Example of an NP-hard search problem.
- Algorithm: branch-and-bound.

Branching

- Initialization:
  - go from starting city to each of remaining cities
  - put resulting partial path into priority queue, ordered by its current length.
- Further (repeatedly):
  - take head element out of priority queue,
  - expand by each one of remaining cities,
  - put resulting partial path into priority queue.

Finding the Solution

- Eventually, a complete path will be found.
- Remember its length as the current shortest path.
- Every time a complete path is found, check if we need to update current best path.
- When priority queue becomes empty, best path is found.

Using a Simple Bound

- Once a complete path is found, we have a lower bound on the length of shortest path
- No use in exploring partial path that is already longer than the current lower bound
- Better bounding methods exist …

Sequential TSP: Data Structures

- Priority queue of partial paths.
- Current best solution and its length.
- For simplicity, we will ignore bounding.

Sequential TSP: Code Outline

```
init_q(); init_best();
while( (p=de_queue()) != NULL ) {
  for each expansion by one city {
        q = add_city(p);
        if( complete(q) ) { update_best(q) };
        else { en_queue(q) };
  }
}
```

Parallel TSP: Possibilities

- Have each process do one expansion
- Have each process do expansion of one partial path
- Have each process do expansion of multiple partial paths
- Issue of granularity/performance, not an issue of correctness.
- Assume: process expands one partial path.

Parallel TSP: Synchronization

- True dependence between process that puts partial path in queue and the one that takes it out.
- Dependences arise dynamically.
- Required synchronization: need to make process wait if q is empty.

Parallel TSP: First Cut (part 1)

```
process i:
    while( (p=de_queue()) != NULL ) {
      for each expansion by one city {
            q = add_city(p);
            if complete(q) { update_best(q) };
            else en_queue(q);
      }
    }
```

Parallel TSP: First cut (part 2)

- In de_queue: wait if q is empty
- In en_queue: signal that q is no longer empty

Parallel TSP

```
process i:
    while( (p=de_queue()) != NULL ) {
      for each expansion by one city {
            q = add_city(p);
            if complete(q) { update_best(q) };
            else en_queue(q);
      }
    }
```

Parallel TSP: More synchronization

- All processes operate, potentially at the same time, on q and best.
- This must not be allowed to happen.
- Critical section: only one process can execute in critical section at once.

Parallel TSP: Critical Sections

- All shared data must be protected by critical section.
- Update_best must be protected by a critical section.
- En_queue and de_queue must be protected by the same critical section.

Parallel TSP

```
process i:
    while( (p=de_queue()) != NULL ) {
      for each expansion by one city {
            q = add_city(p);
            if complete(q) { update_best(q) };
            else en_queue(q);
      }
    }
```

Termination condition

- How do we know when we are done?
- All processes are waiting inside de_queue.
- Count the number of waiting processes before waiting.
- If equal to total number of processes, we are done.

Programming Models

- Explicitly concurrent languages – e.g., Occam, SR, Java, Ada, UPC
- Compiler-supported extensions – e.g., HPF, Cilk
- Library packages outside the language proper – e.g., pthreads, MPI

Programming Models

- Standard models of parallelism
  - shared memory (Pthreads)
  - message passing (MPI)
  - data parallel (Fortran 90 and HPF)
  - shared memory + data parallel (OpenMP)
  - Global address space (UPC)

Thread Creation Syntax

- Properly nested (can share context)
  - Co-Begin (Algol 68, Occam, SR)
  - Parallel loops (HPF, Occam, Fortran90, SR)
  - Launch-at-Elaboration (Ada, SR)
- Fork/Join (pthreads, Ada, Modula-3, Java, SR, Cilk)
- Implicit Receipt (RPC systems, SR)
- Early Reply (SR)

Loops

- For – sequential
- Forall – each statement executed completely and in parallel
- Dopar – each iteration executed in parallel
- Dosingle – each variable assigned once, new value always used

Programming Models

- Standard models of parallelism
  - shared memory (Pthreads)
  - message passing (MPI)
  - data parallel (Fortran 90 and HPF)
  - shared memory + data parallel (OpenMP)
  - Remote procedure call
  - Global address space (UPC)