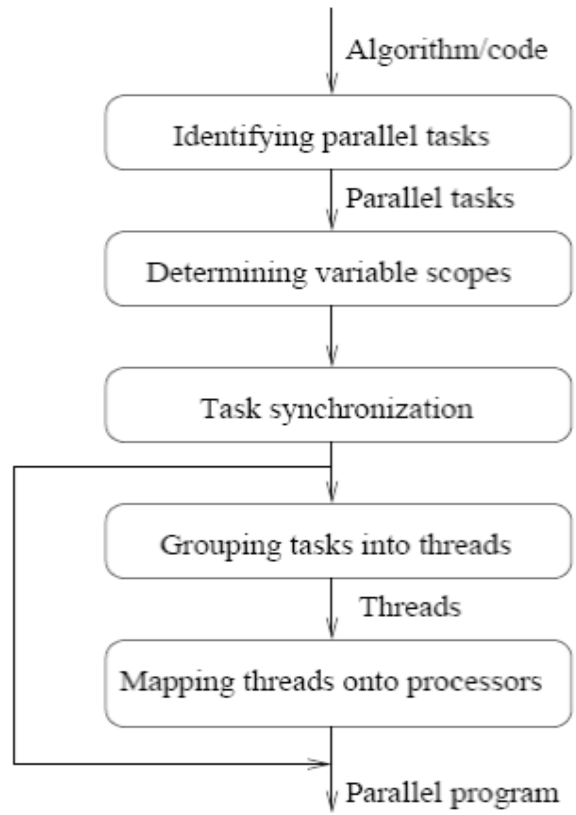
Chapter 3
Shared Memory Parallel Programming

Module 3.1: Parallel Programming Techniques 1

Steps in Creating a Parallel Program



Task Creation: identifying parallel tasks, variable scopes, synchronization
Task Mapping: grouping tasks, mapping to processors/memory

Parallel Programming

- Task Creation (correctness)
  - Finding parallel tasks
    - **Code analysis**
    - Algorithm analysis
  - Variable partitioning
    - Shared vs. private vs. reduction
  - Synchronization
- Task mapping (performance)
  - Static vs. dynamic
  - Block vs. cyclic
  - Dimension mapping: column-wise vs. row-wise
  - Communication and data locality considerations

Code Analysis

- Goal: given a code, without the knowledge of the algorithms, find parallel tasks
- Focus on loop dependence analysis
- Notations:
  - S is a statement in the source code
  - S[i,j,…] denotes a statement in the loop iteration [i,j,…]
  - S1 <u>then</u> S2 means that S1 *happens before* S2
  - If S1 <u>then</u> S2:
    - S1 →T S2 denotes true dependence, i.e. S1 writes to a location that is read by S2
    - S1 →A S2 denotes anti dependence, i.e. S1 reads a location written by S2
    - S1 →O S2 denotes output dependence, i.e. S1 writes to the same location written by S2

Example

```
S1: x = 2;
S2: y = x;
S3: y = x + z;
S4: z = 6;
```

- Dependences:
  - S1 → T S2
  - S1 → T S3
  - S3 → A S4
  - S2 → O S3

Loop-independent vs. loop-carried dependence

- Loop-carried dependence: dependence exists across iterations, i.e., if the loop is removed, the dependence <u>no longer exists</u>
- Loop-independent dependence: dependence exists within an iteration . i.e., if the loop is removed, the dependence exists

```
for (i=1; i<n; i++) {
  S1: a[i] = a[i-1] + 1;
  S2: b[i] = a[i];
}

for (i=1; i<n; i++)
  for (j=1; j< n; j++)
    S3: a[i][j] = a[i][j-1] + 1;

for (i=1; i<n; i++)
  for (j=1; j< n; j++)
    S4: a[i][j] = a[i-1][j] + 1;
```

S1[i] →T S1[i+1]: loop-carried
S1[i] →T S2[i]: loop-independent

S3[i,j] →T S3[i,j+1]:
-loop-carried on for j loop
-No loop-carried dependence in for i loop

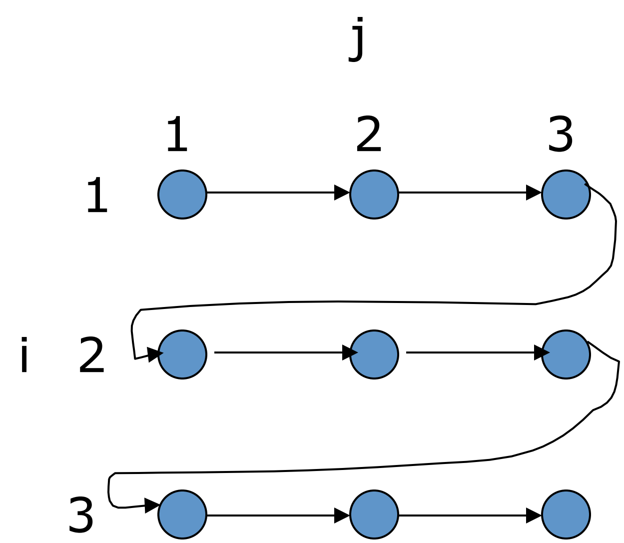
S4[i,j] →T S4[i+1,j]:
No loop-carried dependence in for j loop
Loop-carried on for i loop

Iteration-space Traversal Graph (ITG)

- ITG shows graphically the order of traversal in the iteration space (happens-before relationship)
- Node = a point in the iteration space
- Directed Edge = the next point that will be encountered after the current point is traversed

Example:

```
for (i=1; i<4; i++)
  for (j=1; j<4; j++)
    S3: a[i][j] = a[i][j-1] + 1;
```



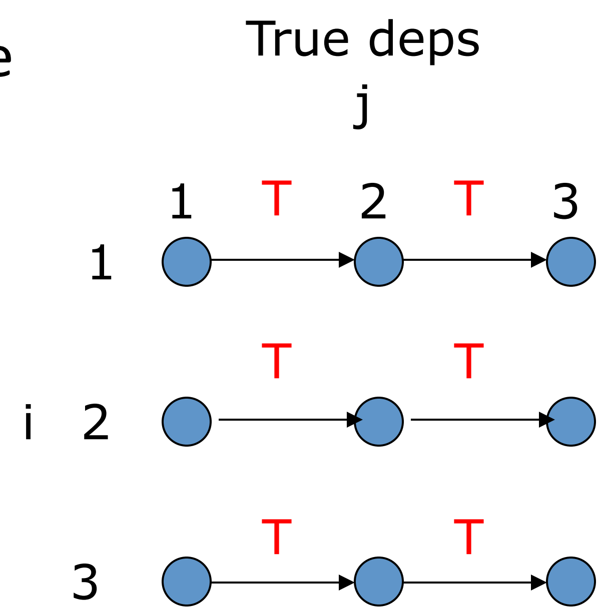
Loop-carried Dependence Graph (LDG)

- LDG shows the true/anti/output dependence relationship graphically
- Node = a point in the iteration space
- Directed Edge = the dependence

Example:

```
for (i=1; i<4; i++)
  for (j=1; j<4; j++)
    S3: a[i][j] = a[i][j-1] + 1;
```

S3[i,j] →T S3[i,j+1]



Further example

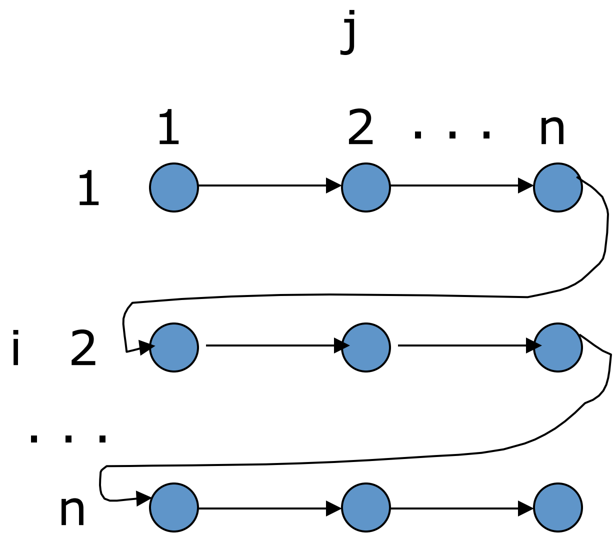
```
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++)
    S1: a[i][j] = a[i][j-1] + a[i][j+1] + a[i-1][j] + a[i+1][j];

for (i=1; i<=n; i++)
  for (j=1; j<=n; j++) {
    S2: a[i][j] = b[i][j] + c[i][j];
    S3: b[i][j] = a[i][j-1] * d[i][j];
  }
```

- Draw the ITG
- List all the dependence relationships
- Draw the LDG

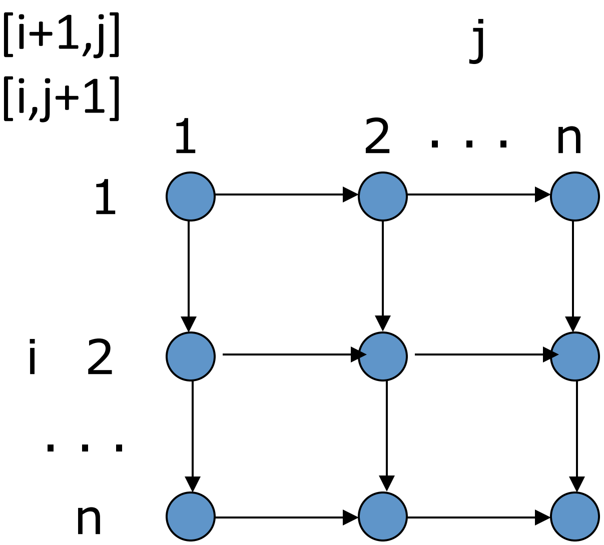
Answer for Loop Nest 1

- ITG



Answer for Loop Nest 1

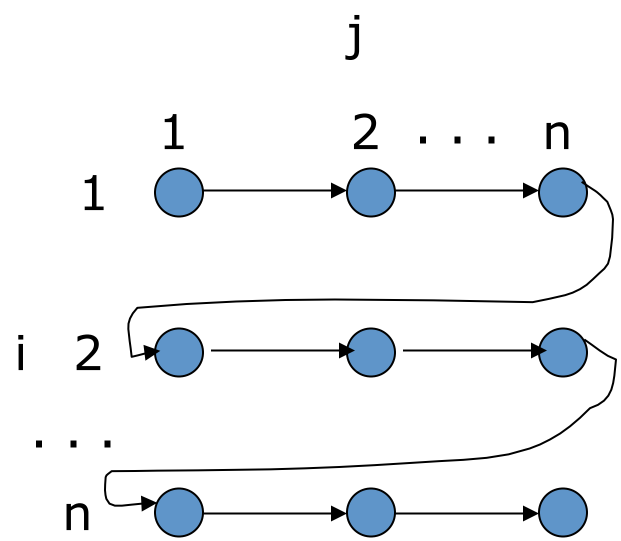
- True dependences:
  - S1[i,j] →T S1[i,j+1]
  - S1[i,j] →T S1[i+1,j]
- Output dependences:
  - None
- Anti dependences:
  - S1[i,j] →A S1[i+1,j]
  - S1[i,j] →A S1[i,j+1]
- LDG:



Note: each Edge represents both true, and anti dependences

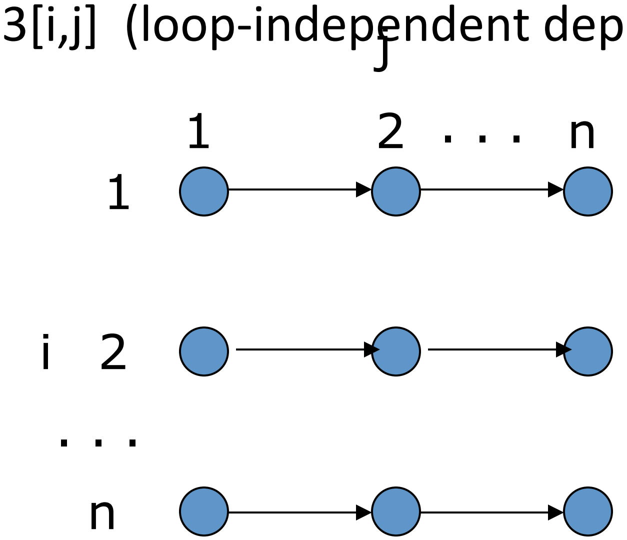
Answer for Loop Nest 2

- ITG



Answer for Loop Nest 2

- True dependences:
  - S2[i,j] →T S3[i,j+1]
- Output dependences:
  - None
- Anti dependences:
  - S2[i,j] →A S3[i,j] (loop-independent dependence)
- LDG:



Note: each edge represents only true dependences

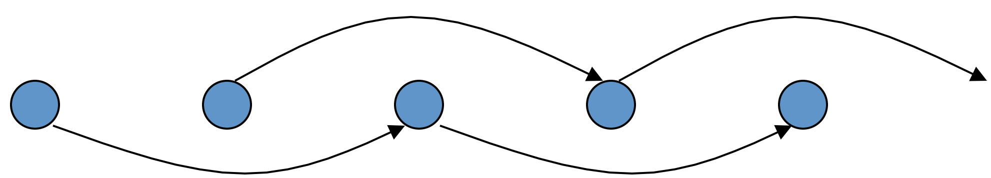
Module 3.2: Parallel Programming Techniques 2

Finding parallel tasks across iterations

- Analyze loop-carried dependences:
  - Dependence must be obeyed (esp. true dependences)
  - There are opportunities when some dependences are missing
- Example 1:

```
for (i=2; i<=n; i++)
  S: a[i] = a[i-2];
```

- LDG:



- Can divide the loop into two parallel tasks (one with odd iterations and another with even iterations):

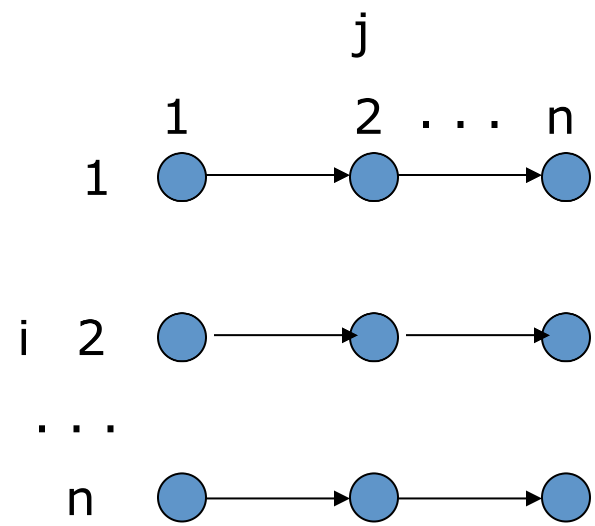
```
for (i=2; i<=n; i+=2)
  S: a[i] = a[i-2];
for (i=3; i<=n; i+=2)
  S: a[i] = a[i-2];
```

Example 2

- Example 2:

```
for (i=0; i<n; i++)
  for (j=0; j< n; j++)
    S3: a[i][j] = a[i][j-1] + 1;
```

- LDG:

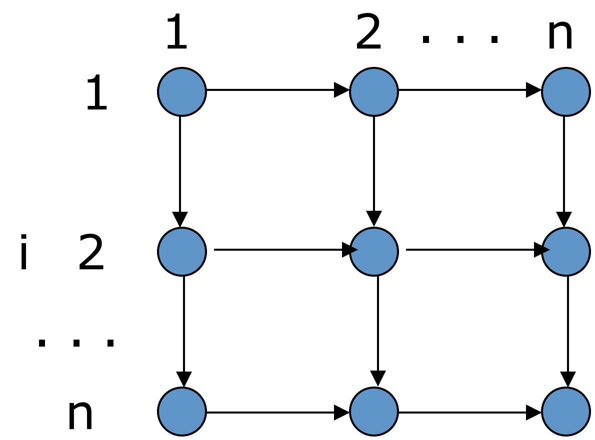


- There are n parallel tasks (one task per i iteration)

Further example

```
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++)
    S1: a[i][j] = a[i][j-1] + a[i][j+1] + a[i-1][j] + a[i+1][j];
```

- LDG:

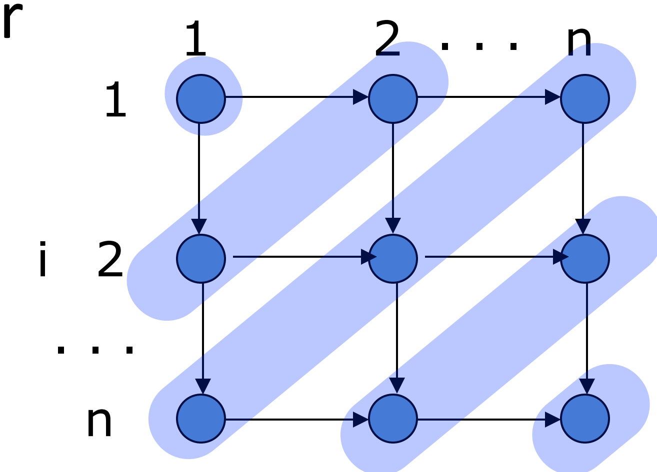


Note: each Edge represents both true, and anti dependences

- Where are the parallel tasks?

Example 3

- Identify which nodes are not dependent on each other
- In each anti-diagonal, the nodes are independent of each other



Note: each Edge represents both true, and anti dependences

- Need to rewrite the code to iterate over anti-diagonals

Structure of Rewritten Code

- Iterate over anti-diagonals, and over elements within an anti-diagonal:

```
Calculate number of anti-diagonals
Foreach anti-diagonal do:
  calculate number of points in the current anti-diagonal
  For each point in the current anti-diagonal do:
    compute the current point in the matrix
```

- Parallelize the highlighted loop
- Write the code…

DOACROSS Parallelism

```
for (i=1; i<=N; i++) {
  S: a[i] = a[i-1] + b[i] * c[i];
}
```

Opportunity for parallelism?

S[i] →T S[i+1]

So it has loop-carried dependence

But, notice that the b[i] * c[i] part has no Loop-carried dependence

Can change to:

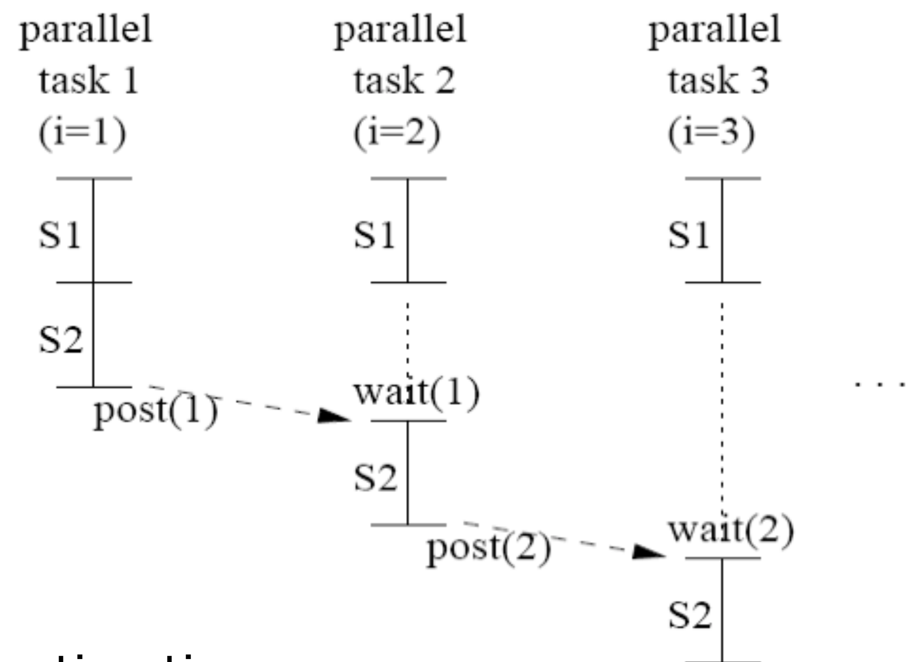
```
for (i=1; i<=N; i++) {
  S1: temp[i] = b[i] * c[i];
}
for (i=1; i<=N; i++) {
  S2: a[i] = a[i-1] + temp[i];
}
```

- Now the first loop is parallel, but the second one is not
- Execution time N x (TS1 + TS2)
- array temp[] introduces storage overhead
- Better solution?

DOACROSS Parallelism

DOACROSS Parallelism

```
Post(0);
for (i=1; i<=N; i++) {
  S1: temp = b[i] * c[i];
  wait(i-1);
  S2: a[i] = a[i-1] + temp;
  post(i);
}
```



Execution time now
TS1 + N x TS2

Small storage overhead

Finding Parallel Tasks in Loop Body

- Identify dependences in a loop body
- If there are independent statements, can split/distribute the loops

```
for (i=0; i<n; i++) {
  S1: a[i] = b[i+1] * a[i-1];
  S2: b[i] = b[i] * coef;
  S3: c[i] = 0.5 * (c[i] + a[i]);
  S4: d[i] = d[i-1] * d[i];
}
```

Loop-carried dependences:
S1[i] →A S2[i+1]
Loop-indep dependences:
S1[i] →T S3[i]

- Note that S4 has no dependences with other statements
- "S1[i] →A S2[i+1]" implies that S2 at iteration i+1 must be executed after S1 at iteration i. Hence dependence not violated if all S2's executed after all S1's

After loop distribution

```
for (i=0; i<n; i++) {
  S1: a[i] = b[i+1] * a[i-1];
  S2: b[i] = b[i] * coef;
  S3: c[i] = 0.5 * (c[i] + a[i]);
  S4: d[i] = d[i-1] * d[i];
}
```

```
for (i=0; i<n; i++) {
  S1: a[i] = b[i+1] * a[i-1];
  S2: b[i] = b[i] * coef;
  S3: c[i] = 0.5 * (c[i] + a[i]);
}

for (i=0; i<n; i++) {
  S4: d[i] = d[i-1] * d[i];
}
```

-Each loop is a parallel task
-Referred to as **function parallelism**

- More distribution possible (refer to textbook)

Identifying Concurrency (contd.)

- Function parallelism:
  - modest degree, does not grow with input size
  - ***difficult to load balance***
  - pipelining, as in video encoding/decoding, or polygon rendering

- Most scalable programs are data parallel
  - use both when data parallelism is limited

DOPIPE Parallelism

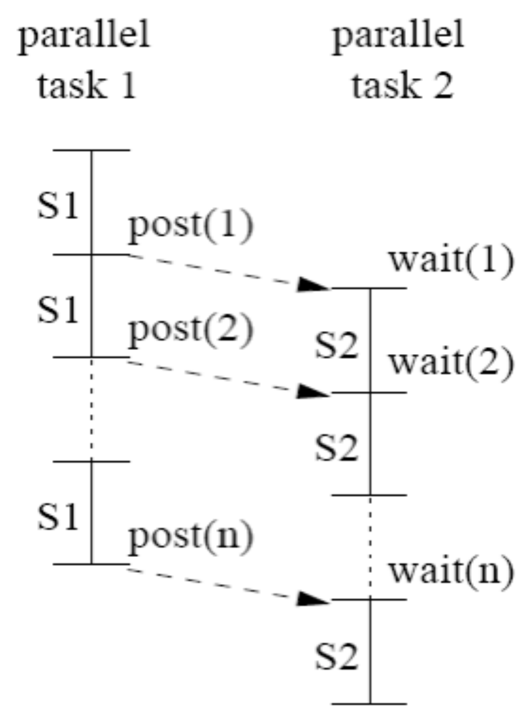
```
for (i=2; i<=N; i++) {
  S1: a[i] = a[i-1] + b[i];
  S2: c[i] = c[i] + a[i];
}
```

Loop-carried dependences:
S1[i-1] →T S1[i]
Loop independent dependence:
S1[i] →T S2[i]
So, where is the parallelism opportunity?

DOPIPE Parallelism

```
for (i=2; i<=N; i++) {
  a[i] = a[i-1] + b[i];
  post(i);
}

for (i=2; i<=N; i++) {
  wait(i);
  c[i] = c[i] + a[i];
}
```



What is the max speedup?
see textbook

Parallel Programming

- Task Creation (correctness)
  - Finding parallel tasks
    - Code analysis
    - **<u>Algorithm analysis</u>**
  - Variable partitioning
    - Shared vs. private vs. reduction
  - Synchronization
- Task mapping (performance)
  - Static vs. dynamic
  - Block vs. cyclic
  - Dimension mapping: column-wise vs. row-wise
  - Communication and data locality considerations

Task Creation: Algorithm Analysis

- Goal: code analysis misses parallelization opportunities available at the algorithm level
- Sometimes, the ITG introduces unnecessary serialization

- Consider the "ocean" algorithm
  - Numerical goal: at each sweep, compute how each point is affected by its neighbors
  - Hence, any order of update (within a sweep) is an approximation
  - Different ordering of updates: may converge quicker or slower
  - Change ordering to improve parallelism
  - Partition iteration space into red and black points
  - Red sweep and black sweep are each fully parallel

Example 3: Simulating Ocean Currents

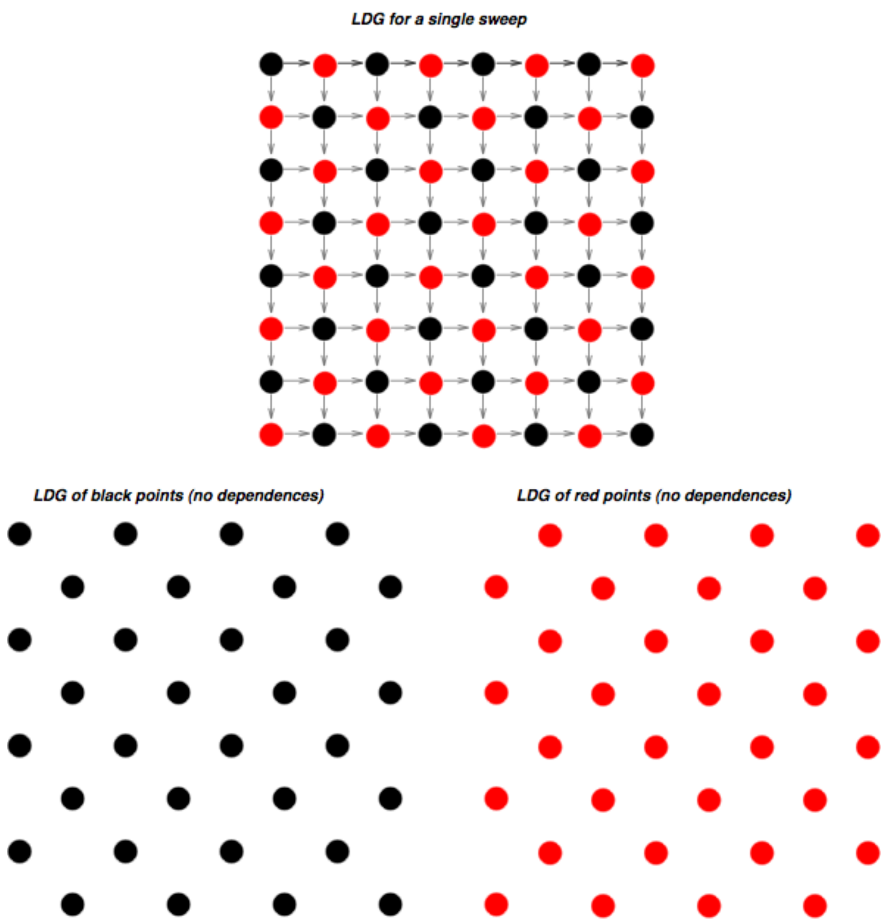
```
Algorithm:

While not converging to a solution do:
  foreach timestep do:
    foreach cross section do a sweep:
      foreach point in a cross section do:
        compute the force interaction with its neighbors


compare with the code that implements the algorithm:

for (I=1; I<=N; I++) {
  for (j=1; j<=N; j++) {
    S1: temp = A[I][j];
    S2: A[I][j] = 0.2 * (A[I][j]+A[I][j-1]+A[I-1][j]+
                  +A[I][j+1]+A[I+1][j]);
    S3: diff += abs(A[I][j] - temp);
  }
}
```

Red-Black Coloring



in one sweep:
- no dependence between black and red points

restructured algorithm:

```
While not converging to a solution do:
  foreach timestep do:
    foreach cross section do:
      foreach red point do: //red sweep
        compute the force interaction
      wait until red sweep
      foreach black point do: //blk sweep
        compute the force interaction
```

see textbook for code

Task Creation: Further Algorithm Analysis

- Can algorithm tolerate *asynchronous* execution?
  - simply ignore dependences within a sweep
  - parallel program *nondeterministic* (timing-dependent!)

```
for (I=1; I<=N; I++) {
  for (j=1; j<=N; j++) {
    S1: temp = A[I][j];
    S2: A[I][j] = 0.2 * (A[I][j]+A[I][j-1]+A[I-1][j]+
                  +A[I][j+1]+A[I+1][j]);
    S3: diff += abs(A[I][j] - temp);
  }
}
```

Module 3.3: Parallel Programming Techniques 3

Parallel Programming

- Task Creation (correctness)
  - Finding parallel tasks
    - Code analysis
    - Algorithm analysis
  - Variable partitioning
    - **Shared vs. private vs. reduction**
  - Synchronization
- Task mapping (performance)
  - Static vs. dynamic
  - Block vs. cyclic
  - Dimension mapping: column-wise vs. row-wise
  - Communication and data locality considerations

Determining Variable Scope

- This step is specific to shared memory programming model
- Analyze how each variable may be used across parallel tasks:
  - Read-only:
    - variable is only read by all tasks
  - R/W non-conflicting:
    - variable is read, written, or both by only one task
  - R/W Conflicting:
    - variable written by one task may be read by another

Example 1

```
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++) {
    S2: a[i][j] = b[i][j] + c[i][j];
    S3: b[i][j] = a[i][j-1] * d[i][j];
  }
```

- Define a parallel task as each “for i” loop iteration
- Read-only:
  - n, c, d
- R/W non-conflicting:
  - a, b
- R/W conflicting:
  - i, j

Example 2

```
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++) {
    S1: a[i][j] = b[i][j] + c[i][j];
    S2: b[i][j] = a[i-1][j] * d[i][j];
    S3: e[i][j] = a[i][j];
  }
```

- Parallel task = each "for j" loop iteration
- Read-only:
  - n, i, c, d
- R/W Non-conflicting:
  - a, b, e
- R/W Conflicting:
  - j

Privatization

- Privatization = converting a shared variable into a private variable in order to remove conflicts
  - Goal: R/W Conflicting ➔ R/W Non-conflicting
- A conflicting variable is privatizable if
  - In program order, the variable is always defined (=written) by a task before use (=read) by the same task
  - The values for different parallel tasks are known ahead of time (hence, private copies can be initialized to the known values)
- Consequence
  - Conflicts disappear when the variable is "privatized"
- Privatization
  - involves making private copies of a shared variable
  - One private copy per *thread* (not per task)
  - How is this achieved in shared memory abstraction?

Example 1

```
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++) {
    S2: a[i][j] = b[i][j] + c[i][j];
    S3: b[i][j] = a[i][j-1] * d[i][j];
  }
```

- Define a parallel task as each "for i" loop iteration
- Read-only:
  - n, c, d
- R/W non-conflicting:
  - a, b
- R/W conflicting **but privatizable**:
  - i, j
  - After privatization: i[ID], j[ID]

Example 2

```
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++) {
    S1: a[i][j] = b[i][j] + c[i][j];
    S2: b[i][j] = a[i-1][j] * d[i][j];
    S3: e[i][j] = a[i][j];
  }
```

- Parallel task = each "for j" loop iteration
- Read-only:
  - n, i, c, d
- R/W Non-conflicting:
  - a, b, e
- R/W Conflicting **but privatizable**:
  - j
  - After privatization: j[ID]

Reduction

- Reduction
  – A special case of privatization, where:
  – Results are accumulated by each thread to a private copy
  – Private copies are merged into the shared copy at the end of computation
- Example: summing up array elements
  – Each thread works on its part of the array and accumulates its sum to a private sum
  – Private sums are accumulated into the shared sum at the end of computation

Reduction Variables and Operations

- Reduction Operation examples:
  - SUM (+), multiplication (*)
  - Logical (AND, OR, …)
- Reduction variable =
  - The scalar variable that is the result of a reduction operation
- Criteria for reducibility:
  - Reduction variable is updated by each task, and the order of update is not important
  - Hence, the reduction operation must be **commutative** and **associative**

Reduction Operation

- Compute:
  - y = y_init *op* x1 *op* x2 *op* x3 ... *op* xn
- *op* is a reduction operator if it is *commutative*
  - u *op* v = v *op* u
- and *associative*
  - (u *op* v) *op* w = u *op* (v *op* w)
- Certain operations can be transformed into reduction operations (see Homeworks)

Variable Partitioning

- Should be declared private:
  - Privatizable variables
- Should be declared shared:
  - Read-only variables
  - R/W Non-conflicting variables
- Should be declared reduction:
  - Reduction variables
- Other R/W Conflicting variables:
  - Privatization possible? If so, privatize them
  - Otherwise, declare as shared, but protect with synchronization

Example 1

```
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++) {
    S2: a[i][j] = b[i][j] + c[i][j];
    S3: b[i][j] = a[i][j-1] * d[i][j];
  }
```

- Declare as shared:
  - n, c, d, a, b
- Declare as private:
  - i, j

Example 2

```
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++) {
    S1: a[i][j] = b[i][j] + c[i][j];
    S2: b[i][j] = a[i-1][j] * d[i][j];
    S3: e[i][j] = a[i][j];
  }
```

- Parallel task = each “for j” loop iteration
- Declare as shared:
  - n, i, c, d, a, b, e
- Declare as private:
  - j

Module 3.4: Parallel Programming Techniques 4

Parallel Programming

- Task Creation (correctness)
  - Finding parallel tasks
    - Code analysis
    - Algorithm analysis
  - Variable partitioning
    - Shared vs. private vs. reduction
  - **Synchronization**
- Task mapping (performance)
  - Static vs. dynamic
  - Block vs. cyclic
  - Dimension mapping: column-wise vs. row-wise
  - Communication and data locality considerations

Synchronization Primitives

- Point-to-point
  - a pair of post() and wait()
  - a pair of send() and recv() in message passing
- Lock
  - ensures mutual exclusion, only one thread can be in a locked region at a given time
- Barrier
  - a point where a thread is allowed to go past it only when all threads have reached the point.

Lock

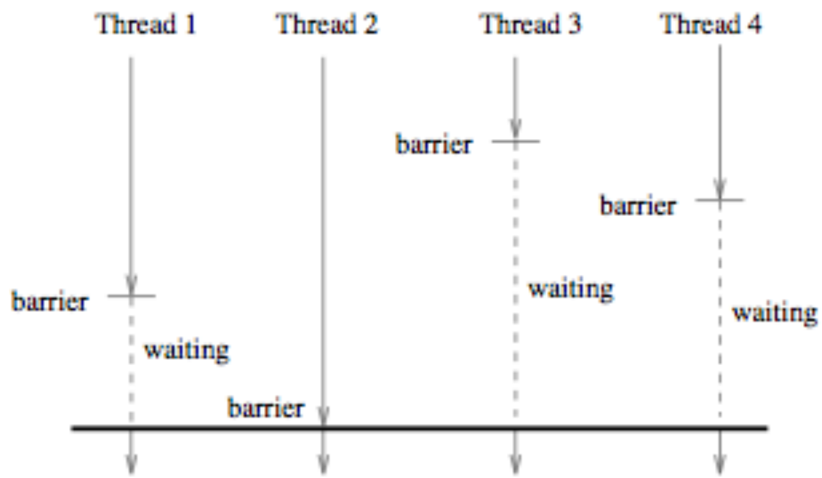
- What problem may arise here?

```
// inside a parallel region
for (i=start_iter; i<end_iter; i++)
  sum = sum + a[i];
```

- Lock ensures only one thread inside the locked region

```
// inside a parallel region
for (i=start_iter; i<end_iter; i++) {
    lock(x);
    sum = sum + a[i];
    unlock(x);
}
```

Barrier: Global Event Synchronization



- Load balance important
- Execution time dependent on the slowest thread
  - One reason for gang scheduling and avoiding time sharing and context switching