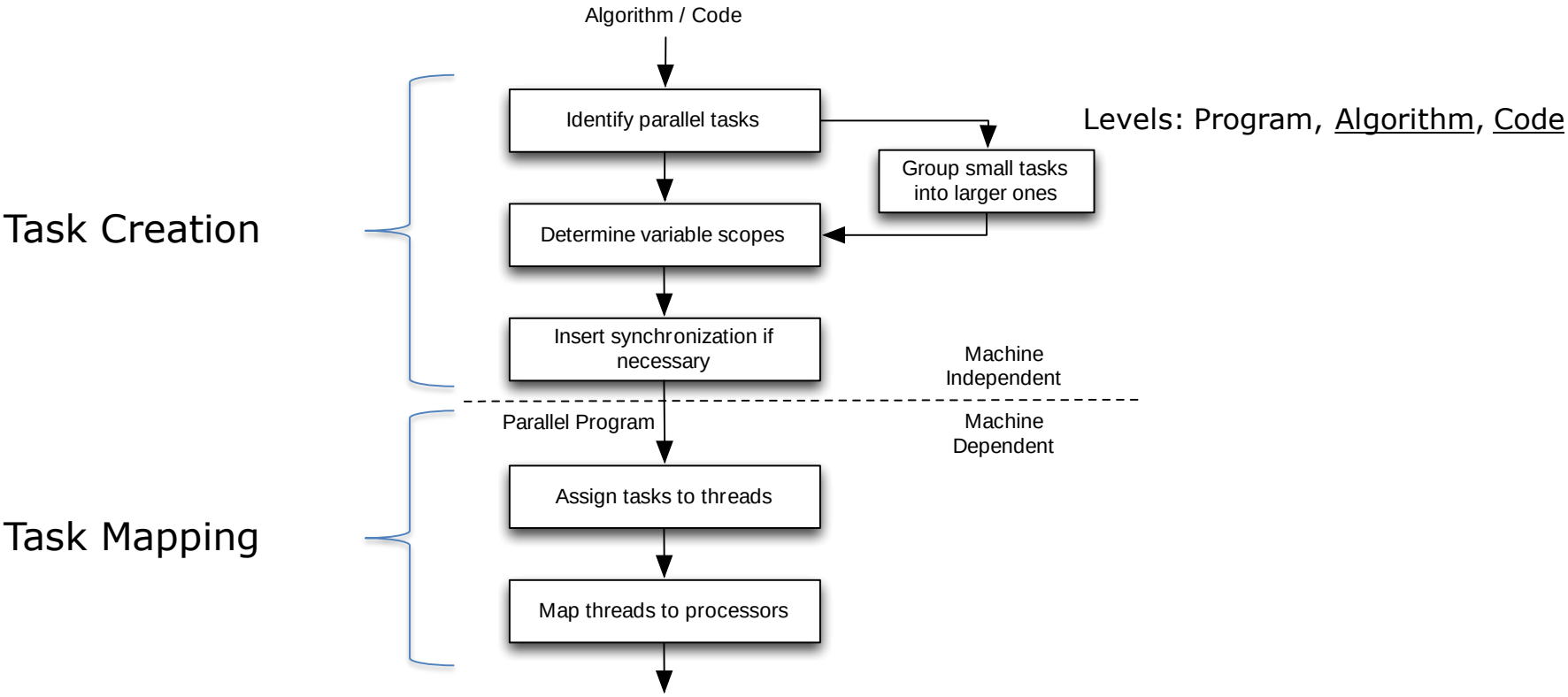
Chapter 3
Shared Memory Parallel Programming

Steps in Parallel Programming

Steps in Creating a Parallel Program



Task Creation: identifying parallel tasks, variable scopes, synchronization

=> Create tasks that can cooperate to perform a single computation

Task Mapping: grouping tasks, mapping to processors/memory

=> May be hidden from programmers

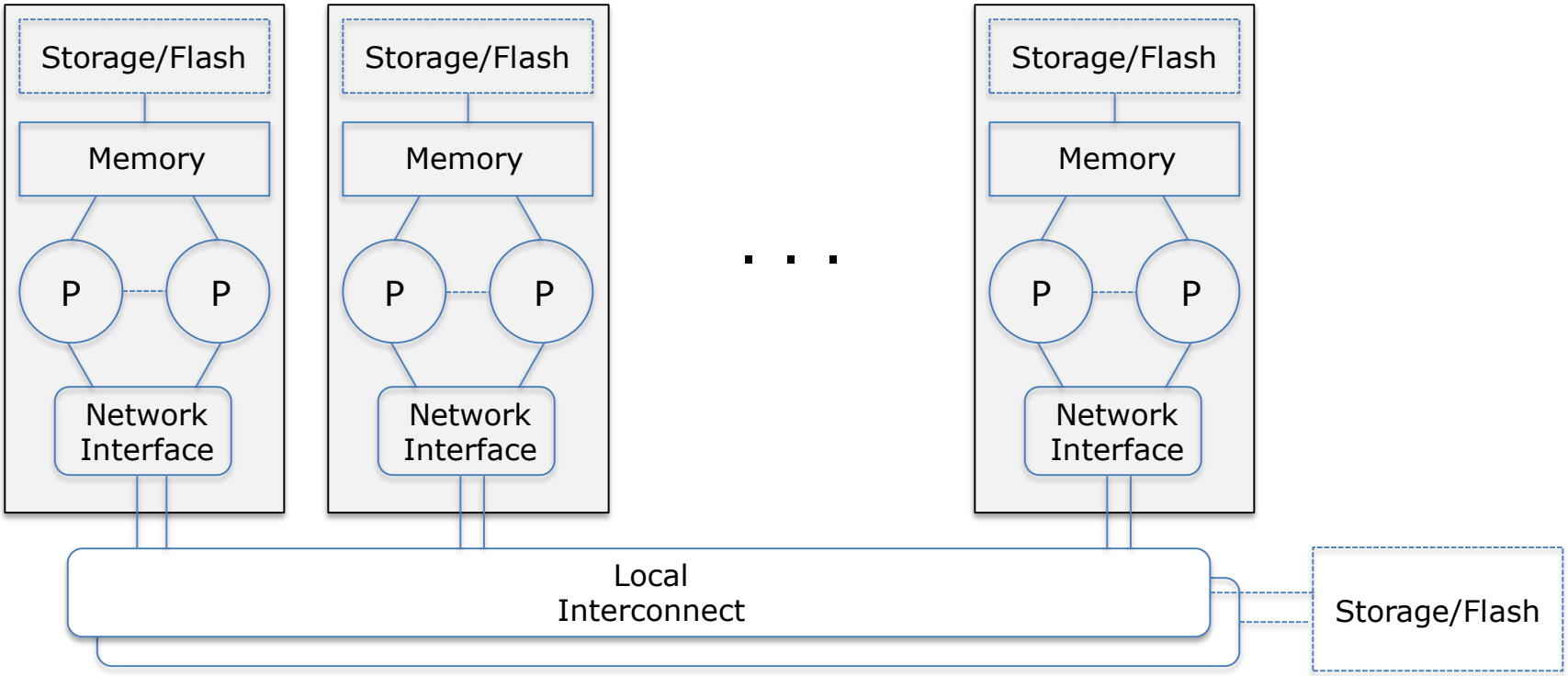
Multiprocessor Systems

Initial focus: Shared memory programming model

=> Concepts important to understand multicore architectures

=> It's a dominant programming model for multicore

=> Multicore processors (P) are the building block for base systems



A balanced system design necessary

Storage Rich Server Example

	Capacity Optimized	Performance Optimized	All Flash
CPU	1x Intel Xeon E5-2620v4 8cores	2x Intel Xeon E5-2620v4 8cores	2x Intel Xeon E5-2620v4 8cores
RAM	64GB RAM DDR4;	128GB RAM DDR4;	128GB RAM DDR4;
Storage	12x3.5" (6TB or 10TB)	24x2.5" 28TB+	24 x Intel 1200GB SATA SSD
Cache(R/W)	1xIntel® P3700/P4800 400GB NVMe	2x Intel P3700/P4800 400GB NVMe	3 x Intel® P4800X 375GB NVMe
HBA	Broadcom SAS 3008 IT-mode (HBA)		
NIC	Intel® X520 2x10GbE NIC SFP+	Intel® X520 2x10GbE NIC SFP+	2 x Mellanox ConnectX-4 25/100GbE
All Configurations require 2 switches: 1 for client connections, 1 for intra-cluster communication			

Spectrum NAS can run on any x86 storage rich server but the Items in purple must be the same family model/vendor as listed above

(See following page)

Parallelism and Amdahl's Law

The text states there are three levels of parallelism:

- program level => e.g., parallel compilation of file (not a focus)
- algorithm level
- code level

The limits of parallelism: Amdahl's law

Speedup is defined as

$$\frac{\text{time for serial execution}}{\text{time for parallel execution}}$$

or, more precisely, as

$$\frac{\text{time for serial execution of best serial algorithm}}{\text{time for parallel execution of our algorithm}}$$

Parallelism and Amdahl's Law

Taken Deeper

Consider the execution of a given program on a uniprocessor system with a total execution time of *T* minutes

Let's say the program has been parallelized or partitioned for parallel execution on a cluster of many processing nodes

Assume that a fraction α of the code must be executed sequentially, called the *sequential bottleneck*

Therefore, *(1 – α)* of the code can be compiled for parallel execution by *n* processors

The total execution time of the program is calculated by $\alpha T + (1 - \alpha)T/n$, where the first term is the sequential execution time on a single processor and the second term is the parallel execution on *n* processing nodes

Amdahl's Law states that the *speedup factor* of using the n-processor system over the use of a single processor is expressed by:

$$\text{Speedup} = S = T/[\alpha T + (1 - \alpha)T/n] = 1/[\alpha + (1 - \alpha)/n]$$

Note: All system or communication overhead is ignored. The I/O time or exception handling time is also not included.

Parallelism and Amdahl's Law
What do we learn?

$$\text{Speedup} = S = T/ [\alpha T + (1 - \alpha)T/n] = 1/ [\alpha + (1 - \alpha)/n]$$

The maximum speedup of n is achieved only if:

- The sequential bottleneck α is reduced to zero
- Or the code is fully parallelizable with α = 0

As the cluster becomes sufficiently large, that is, $n \rightarrow \infty$

S approaches $1/\alpha$, an upper bound

Hence, the benefit of optimizing for parallelization

Parallel Programming

- Task Creation
  - Identifying parallel tasks
    - **Code analysis**
    - Algorithm analysis
  - Variable partitioning
    - Shared vs. private vs. reduction
  - Synchronization
- Task mapping (performance)
  - Tasks to threads
    - Typically more tasks than processors, so multiple tasks assigned to threads
    - A goal may be to balance workload of the threads
  - Mapping threads to processors, including data layout in memory
    - Goal is to achieve communication locality between processors and data locality based on access

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 or group of statements in the source code
  - If there are two statements S_1 and S_2
  - $S_1 \rightarrow S_2$ means that S_1 *happens before* S_2
  - If $S_1 \rightarrow S_2$:
    - $S_1 \rightarrow^T S_2$ denotes true dependence, i.e. S_1 writes to a location that is read by S_2
    - $S_1 \rightarrow^A S_2$ denotes anti dependence, i.e. S_1 reads a location written by S_2
    - $S_1 \rightarrow^O S_2$ denotes output dependence, i.e. S_1 writes to the same location written by S_2
  - S[i,j] denotes a statement in the loop iteration [i,j]

Example

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

- Dependences:
  - $S_1 \rightarrow^{T} S_2$

    x is written in statement S_1 and read in statement S_2
  - $S_1 \rightarrow^{T} S_3$

    x is written in statement S_1 and read in statement S_3
  - $S_3 \rightarrow^{A} S_4$

    z is read in statement S_3 and written in statement S_4
  - $S_2 \rightarrow^{O} S_3$

    y is written in statement S_2 and also written in statement S_3

Statements can be Grouped

```
S12:
        x = 2;
        y = x;
S34:
        y = x + z;
        z = 6;
```

- Dependences:
  - $S_{12} \rightarrow^{T} S_{34}$

    Statement group S_{12} produces a value *x* that is read by statement group S_{34}

  - $S_{12} \rightarrow^{O} S_{34}$

    Statement group S_{12} writes onto *y* that is later written in statement group S_{34}

Loop-independent vs. loop-carried dependence

- Loop-carried dependence: dependence exists across iterations
  - i.e., if the loop is removed, the dependence no longer exists
- 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;
```

$S_1[i] \rightarrow^T S_1[i+1]$: loop-carried
$S_1[i] \rightarrow^T S_2[i]$: loop-independent

$S_3[i,j] \rightarrow^T S_3[i,j+1]$:
-loop-carried on for j loop
-No loop-carried dependence for i loop

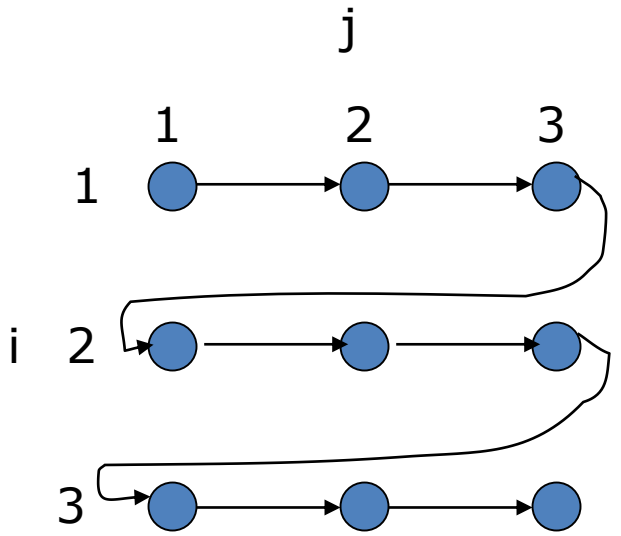
$S_4[i,j] \rightarrow^T S_4[i+1,j]$:
No loop-carried dependence for j loop
Loop-carried on for i loop

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;
```



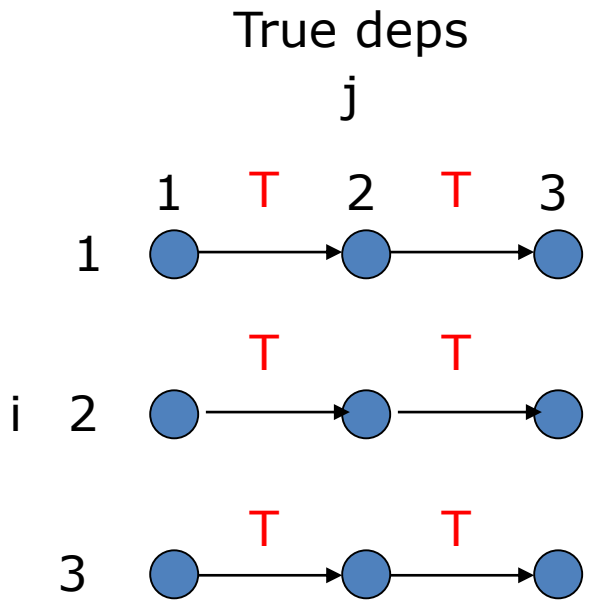
Loop-carried Dependence Graph (LDG)

- 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;
```

$S_3[i,j] \rightarrow^T S_3[i,j+1]$



Further example

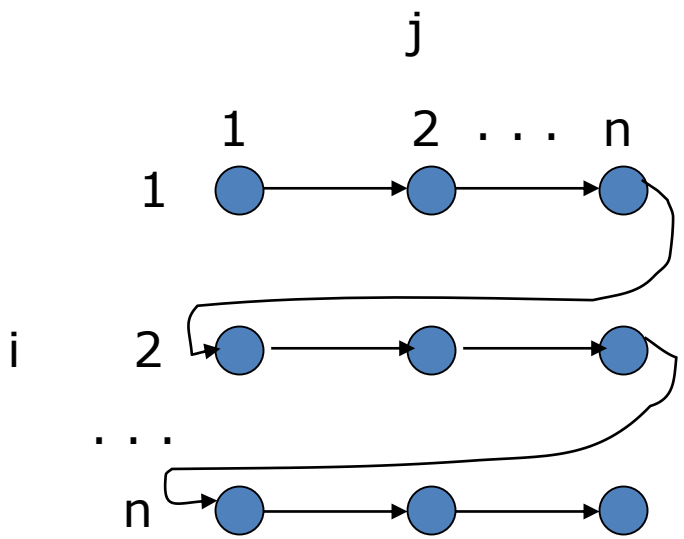
```
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++)
    S1: a[i][j] = a[i][j-1] + a[i][j+1] + a[i-1][j] + a[i+1][j];

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

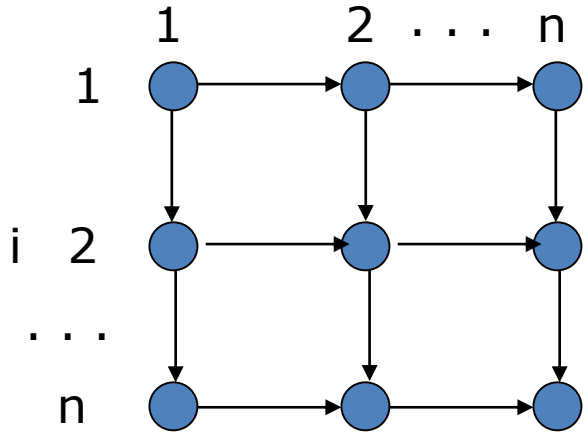



```
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++)
    S1: a[i][j] = a[i][j-1] + a[i][j+1] + a[i-1][j] + a[i+1][j];

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];
  }
```

Answer for Loop Nest 1

- True dependences:
  - $S_1[i,j] \rightarrow_T S_1[i,j+1]$
  - $S_1[i,j] \rightarrow_T S_1[i+1,j]$
- Output dependences:
  - None
- Anti dependences:
  - $S_1[i,j] \rightarrow_A S_1[i+1,j]$
  - $S_1[i,j] \rightarrow_A S_1[i,j+1]$
- LDG:



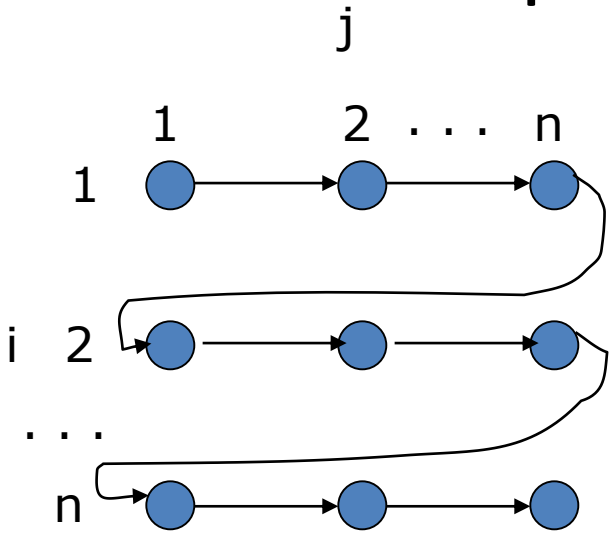
Note: each Edge represents both true, and anti dependences

```
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];
  }
```

Answer for Loop Nest 2

- ITG

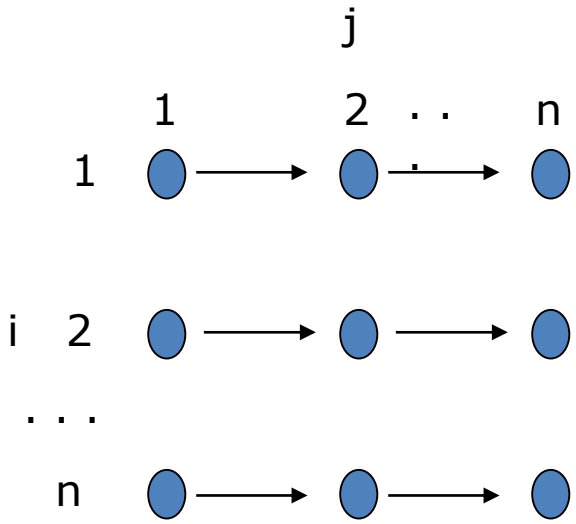



```
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++)
    S1: a[i][j] = a[i][j-1] + a[i][j+1] + a[i-1][j] + a[i+1][j];

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];
  }
```

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

```
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];
  }
```

Review Questions

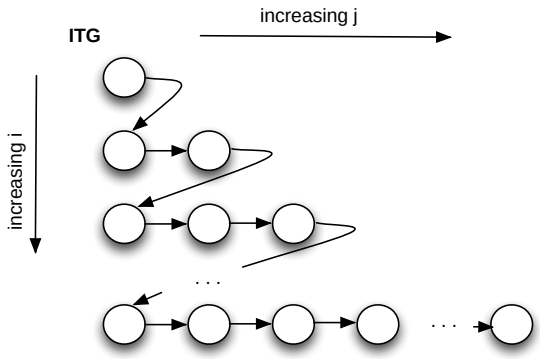
- What are three types of dependences?
- What are types of dependences related to loop structure?
- For the following code, show the ITG

```
...
for (i=1; i<=N; i++) {
  for (j=1; j<=i; j++) {  // note the index range!
    S1: a[i][j] = b[i][j] + c[i][j];
    S2: b[i][j] = a[i][j-1];
    S3: c[i][j] = a[i][j];
  }
}
```

Review Questions

- What are three types of dependences?
  - True (producer consumer), anti, and output
- What are types of dependences related to loop structure?
  - Loop carried (across loop iterations) and loop independent
- For the following code, show the ITG

```
...
for (i=1; i<=N; i++) {
  for (j=1; j<=i; j++) {  // note the index range!
    S1: a[i][j] = b[i][j] + c[i][j];
    S2: b[i][j] = a[i][j-1];
    S3: c[i][j] = a[i][j];
  }
}
```



Review Questions

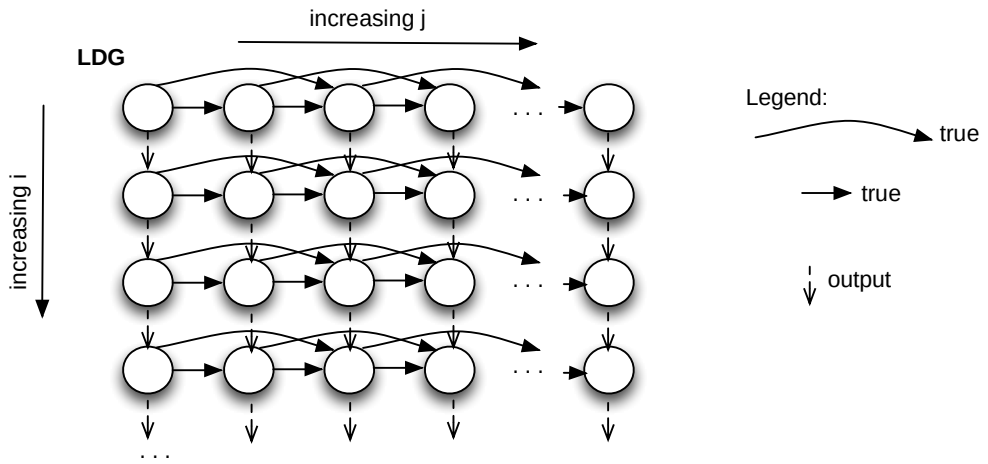
- For the following code, show the LDG

```
. . .
for (i=1; i<=N; i++) {
  for (j=2; j<=N; j++) {
    S1: a[i][j] = a[i][j-1] + a[i][j-2];
    S2: a[i+1][j] = a[i][j] * b[i-1][j];
    S3: b[i][j] = a[i][j];
  }
}
```

Review Questions

- For the following code, show the LDG

```
. . .
for (i=1; i<=N; i++) {
  for (j=2; j<=N; j++) {
    S1: a[i][j] = a[i][j-1] + a[i][j-2];
    S2: a[i+1][j] = a[i][j] * b[i-1][j];
    S3: b[i][j] = a[i][j];
  }
}
```

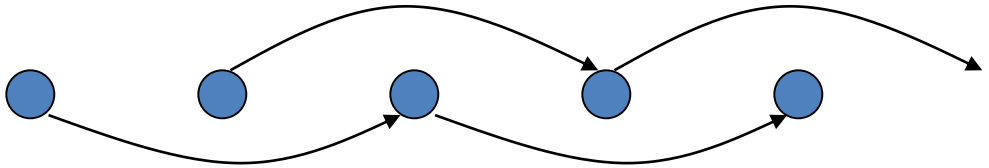


Finding parallel tasks across iterations

- 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):

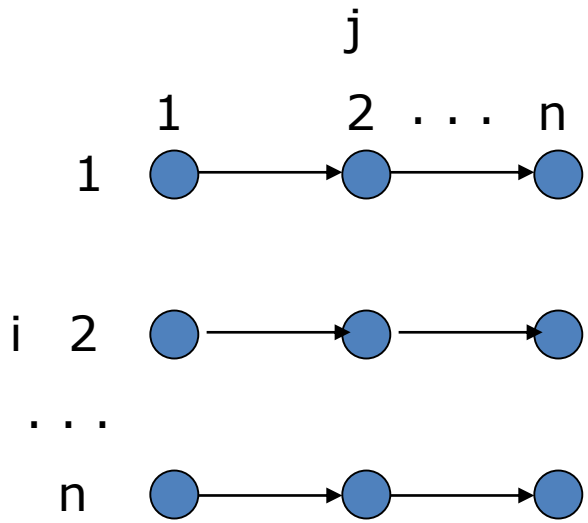
```
for (i=2; i<=n; i+=2)
  S: a[i] = a[i-2];
for (i=3; i<=n; i+=2)
  S: a[i] = a[i-2];
```

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;
```

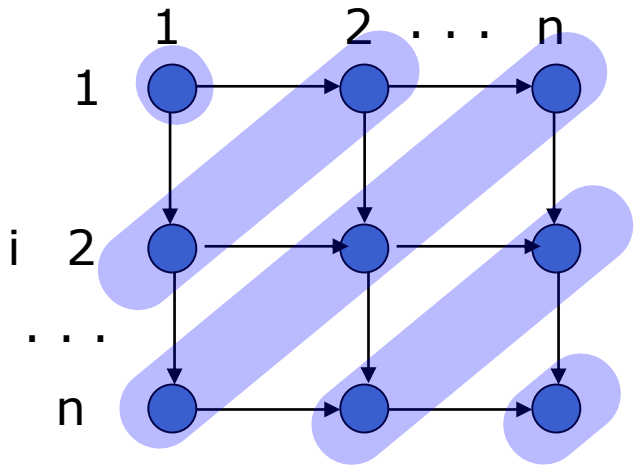
- LDG:



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

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] $\rightarrow^{T}$ 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 ($TS_1 + TS_2$) => $TS_1 + N \times TS_2$
- array temp[] introduces storage overhead
- Better solution?

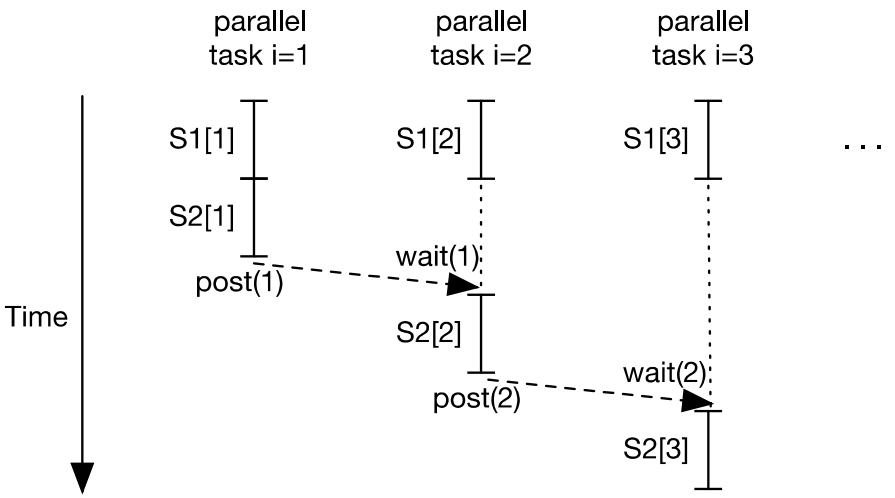
Post-Wait Synchronization Pair

- Post(unique name *x*)
  - Increments memory address m(x) that corresponds to name x
- Wait(x)
  - Blocks if m(x) <= 0, unblocks if m(x) > 0
  - Decrements m(x)
- Example
  - Post(flag), wait(flag)
  - Post(i+1), wait(i)
  - Post(i,j,k), wait(i,j,k)

DOACROSS Parallelism

DOACROSS Parallelism

```
Post(0);            //inserted to
                    //not block
                    //first iteration
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
$TS_1 + N \times TS_2$

And smaller storage overhead
- Temp is duplicated per thread, rather than per iteration

DOACROSS Parallelism

If assume TS1 = TS2 = T and N>>T

The **speedup ratio** becomes

$[N \times (TS_1 + TS_2)] / [TS_1 + (N \times TS_2)]$

$= 2NT / (N+1)T \approx 2$ times

DOPIPE Parallelism

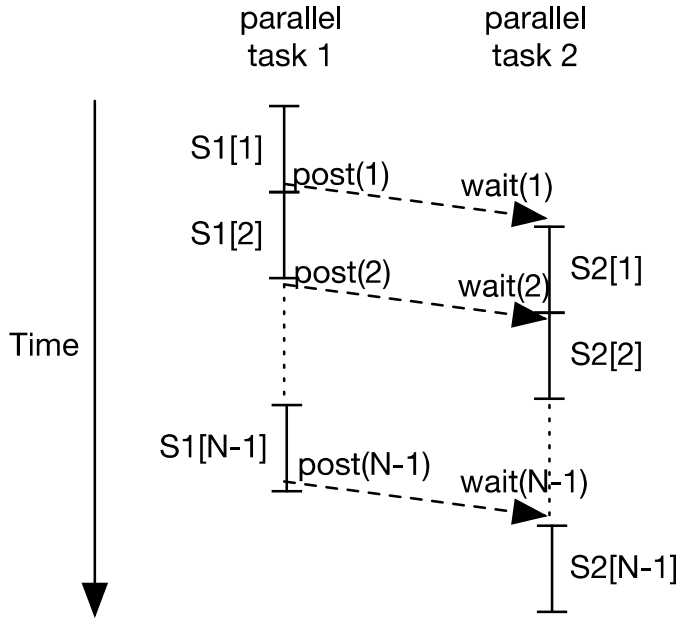
```
for (i=1; i<=N; i++) {
  S1: a[i] = a[i-1] + b[i];
  S2: c[i] = c[i] + a[i];
}
```

Loop-carried dependences:
  S1[i] $\rightarrow^{T}$ S1[i+1]
Loop independent dependence:
  S1[i] $\rightarrow^{T}$ S2[i]
So, where is the parallelism opportunity?

DOPIPE Parallelism

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

for (i=1; i<=N; i++) {
  wait(i);
  c[i] = c[i] + a[i];
}
```



i.e., Distribute the loop and introduce pipelined parallelism

Summary

- The main purpose of employing loop-level parallelism is to search and split sequential tasks of a program and convert them into parallel tasks without any prior information about the algorithm.
- Parts of data that are recurring and consume significant amount of execution time are good candidates for loop-level parallelism.
- Some common applications of loop-level parallelism are found in mathematical analysis that uses multiple-dimension matrices which are iterated in nested loops.
- There are different kind of parallelization techniques which are used on the basis of data storage overhead, degree of parallelization and data dependencies.

Summary

- **DOALL**
  - Technique used where we can parallelize each iteration of the loop without any interaction between the iterations.
  - Hence, the overall run-time gets reduced from N * T (for a serial processor, where T is the execution time for each iteration) to only T (since all the N iterations are executed in parallel).
- **DOACROSS**
  - Technique used wherever there is a possibility for data dependencies.
  - Hence, we parallelize tasks in such a manner that all the data independent tasks are executed in parallel, but the dependent ones are executed sequentially.
  - There is a degree of synchronization used to sync the dependent tasks across parallel processors.
- **DOPIPE**
  - Method to perform loop-level parallelism by pipelining the statements in a loop.
  - Pipelined parallelism may exist at different levels of abstraction like loops, functions and algorithmic stages.
  - The extent of parallelism depends upon the programmers' ability to make best use of this concept.
  - It also depends upon factors like identifying and separating the independent tasks and executing them in parallel

Parallel Programming

- Task Creation
  - Identifying parallel tasks
    - Code analysis
    - **Algorithm analysis**
  - Variable partitioning
    - Shared vs. private vs. reduction
  - Synchronization
- Task mapping (performance)
  - Tasks to threads
    - Typically more tasks than processors, so multiple tasks assigned to threads
    - A goal may be to balance workload of the threads
  - Mapping threads to processors, including data layout in memory
    - Goal is to achieve communication locality between processors and data locality based on access

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

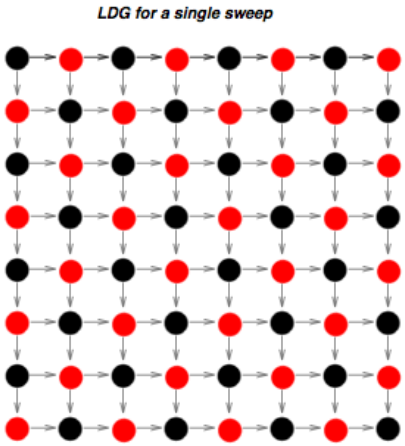
Algorithm:

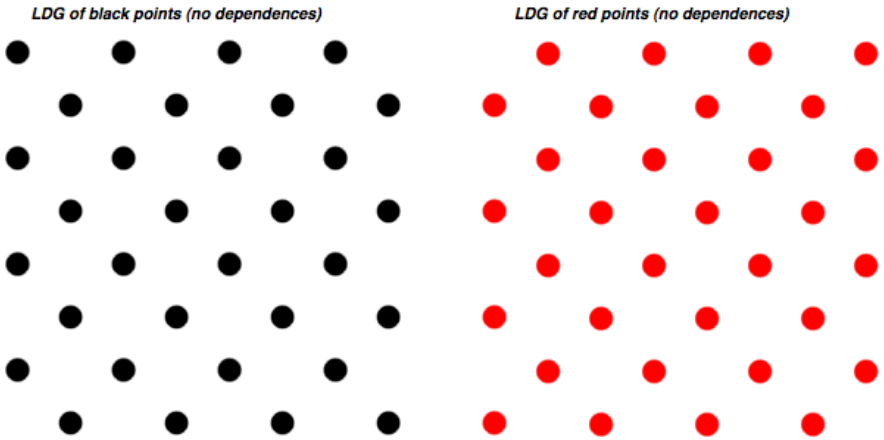
```
While not converging to a solution do:
  foreach timestep do:
    foreach cross section do a sweep:
      foreach point in a cross section do:
        compute the force interaction with its neighbors
```

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

Module Review Questions

- Describe the differences between DOALL, DACROSS, function, and DOPIPE parallelism

Module Review Questions

- Describe the differences between DOALL, DACROSS, function, and DOPIPE parallelism
  - DOALL: parallelism between independent loop iterations
  - DOACROSS: parallelism between dependent loop iterations guarded by synchronization
  - Function: parallelism between independent code sections. In the context of loops, the code sections come from different statements in a loop
  - DOPIPE: parallelism between dependent statements in a loop, guarded by synchronization

Parallel Programming

- Task Creation
  - Identifying parallel tasks
    - Code analysis
    - Algorithm analysis
  - Variable partitioning
    - **Shared vs. private vs. reduction**
  - Synchronization
- Task mapping (performance)
  - Tasks to threads
    - Typically more tasks than processors, so multiple tasks assigned to threads
    - A goal may be to balance workload of the threads
  - Mapping threads to processors, including data layout in memory
    - Goal is to achieve communication locality between processors and data locality based on access

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];
  }
```

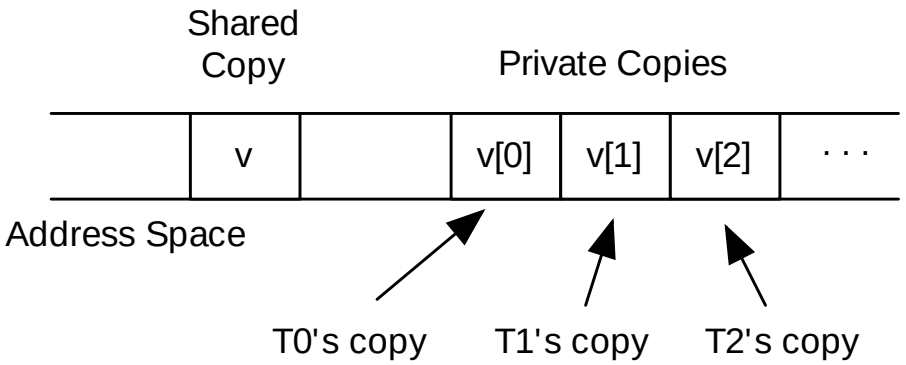
- Parallel task = each “for j” loop iteration
- Read-only:
  - n, i, c, d
- R/W Non-conflicting:
  - a, b, e
- R/W Conflicting:
  - j

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)

Illustration

- Privatization of “v”



One way to think of private copies as arrays of variables with the Task IDs used as the index for private copies

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

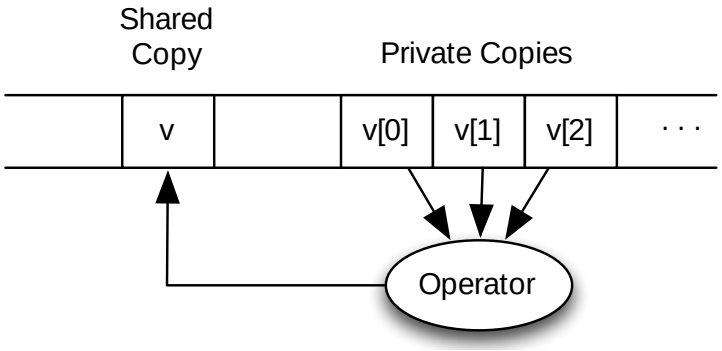
- Reduction Operation examples:
  - SUM (+), multiplication (*)
  - Logical (AND, OR, …)
- Reduction variable =
  - The scalar variable that is the result of a reduction operation
- Criteria for reducibility:
  - Reduction variable is updated by each task, and the order of update is not important
  -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

Illustration

- Reduction of “v”



Variable Scope Determination

- 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 Review Questions

- Describe various variable scope definitions

- Determine appropriate variable scope when we parallelize the for j loop:

```
for (i=0; i<n; i++) {
  for (j=0; j<p; j++) {
    x = 0;
    for (k=0; k<m; k++)
      x = x + A[i][k] * B[k][j];
    Y[i][j] = x + C[i][j];
  }
}
```

Module Review Questions

- Describe various variable scope definitions
  - Shared: all threads share one copy
  - Private: each thread has a copy, further divided into firstprivate and lastprivate in OpenMP
- Determine appropriate variable scope when we parallelize the for j loop:
  - Shared: i, n, m, A, B, C, Y
  - Private: j, k, x

```
for (i=0; i<n; i++) {
  for (j=0; j<p; j++) {
    x = 0;
    for (k=0; k<m; k++)
      x = x + A[i][k] * B[k][j];
    Y[i][j] = x + C[i][j];
  }
}
```

Parallel Programming

- Task Creation
  - Identifying parallel tasks
    - Code analysis
    - Algorithm analysis
  - Variable partitioning
    - Shared vs. private vs. reduction
  - **Synchronization**
- Task mapping (performance)
  - Tasks to threads
    - Typically more tasks than processors, so multiple tasks assigned to threads
    - A goal may be to balance workload of the threads
  - Mapping threads to processors, including data layout in memory
    - Goal is to achieve communication locality between processors and data locality based on access

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

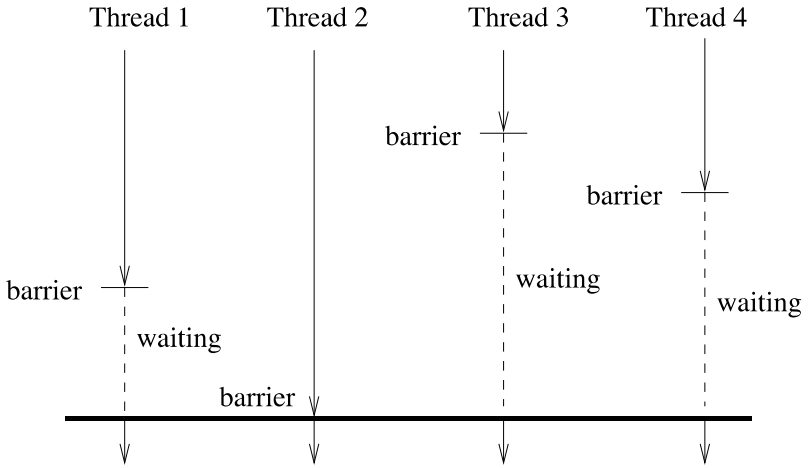
- What problem may arise here?

```
// 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
- Artificial load imbalance occurs with context switching => avoid by gang scheduling

Module Review Questions

- What common synchronization primitives are supported in shared memory system?
- Why is load balancing important when barriers are used?
- What problems are solved (and unsolved) with gang scheduling?

Module Review Questions

- What common synchronization primitives are supported in shared memory system?
  - Lock, barrier, and point-to-point (e.g. post-wait)
- Why is load balancing important when barriers are used?
  - Parallel region execution time is determined by the slowest thread
- What problems are solved (and unsolved) with gang scheduling?
  - GS prevents no-forward progress when a thread is switched out
  - GS frees up CPUs for other uses when a thread is switched out
  - GS does not remove load imbalance and its effects

Parallel Programming

- Task Creation
  - Identifying parallel tasks
    - Code analysis
    - Algorithm analysis
  - Variable partitioning
    - Shared vs. private vs. reduction
  - Synchronization
- Task mapping (performance)
  - **Tasks to threads**
    - Typically more tasks than processors, so multiple tasks assigned to threads
    - A goal may be to balance workload of the threads
  - Mapping threads to processors, including data layout in memory
    - Goal is to achieve communication locality between processors and data locality based on access

Tasks to Threads

- Typically, more tasks than threads, which leads to questions…
  - What tasks should be assigned to a thread?
  - How they should be assigned?
- Important Issues
  - Task management overheads (larger tasks incur lower overheads)
  - Load balance (larger tasks may reduce load balance)
  - Data locality (increasingly important with data growth trends)
- Static or Dynamic?
  - Static task mapping means tasks are preassigned to threads before execution
  - Dynamic task mapping means they're not preassigned prior to execution, but instead a task queue is created and maintained. i.e., whenever a thread becomes idle, it takes a task form the queue and executes it.

Parallel Programming

- Task Creation
  - Identifying parallel tasks
    - Code analysis
    - Algorithm analysis
  - Variable partitioning
    - Shared vs. private vs. reduction
  - Synchronization
- Task mapping (performance)
  - Tasks to threads
    - Typically more tasks than processors, so multiple tasks assigned to threads
    - A goal may be to balance workload of the threads
  - **Mapping threads to processors**, including data layout in memory
    - Goal is to achieve communication locality between processors and data locality based on access

Threads to Processors

- Aspects to consider…
  - Leave it to the OS thread scheduler to decide. i.e., do nothing
    - The OS thread scheduler decides when ready threads should run and which processors they should run on
    - It takes into account response time, fairness, thread priority, utilization of processors, and overheads of context switching
    - A challenge is that parallel threads need to synchronize (locks, barriers, etc.) and some OS schedulers offer gang scheduling (all or none)
  - Data Locality
    - Threads run on same node where data (especially large amounts) resides
      - e.g., a NUMA system has memory with varying access times
    - Another approach is to allocate or migrate data to the thread that accesses it
      - e.g., Page allocation and migration policy (LRU, First touch, round robin, etc.)