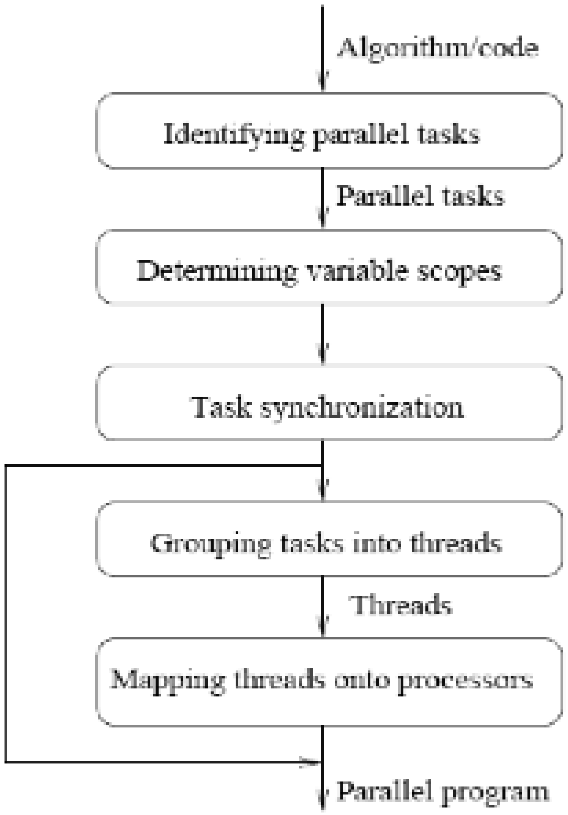
Shared-Memory Parallel Programming

[§3.1] Solihin identifies several steps in parallel programming.


Algorithm/code
Identifying parallel tasks
Parallel tasks
Determining variable scopes
Task synchronization
Grouping tasks into threads
Threads
Mapping threads onto processors
Parallel program

The first step is identifying parallel tasks. Can you give an example?

The next step is identifying variable scopes. What does this mean?

The next step is grouping tasks into threads. What factors need to be taken into account to do this?

Then threads must be synchronized. How have we seen this done in the last lecture?

What considerations are important in mapping threads to processors?

Solihin says that there are three levels of parallelism:

- program level
- algorithm level
- code level

Identifying loop-level parallelism

[§3.2] *Goal:* given a code, without knowledge of the algorithm, 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*, …]
- "*S*1 then *S*2" means that *S*1 *happens before S*2
- If *S*1 then *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

Example:

S1: *x* = 2;
S2: *y* = x;
S3: *y* = x + 4;
S4: *x* = *y*;

Exercise: Identify the dependences in the above code.

Loop-independent vs. loop-carried dependences

[§3.2] 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 still exists.

Example:

```
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

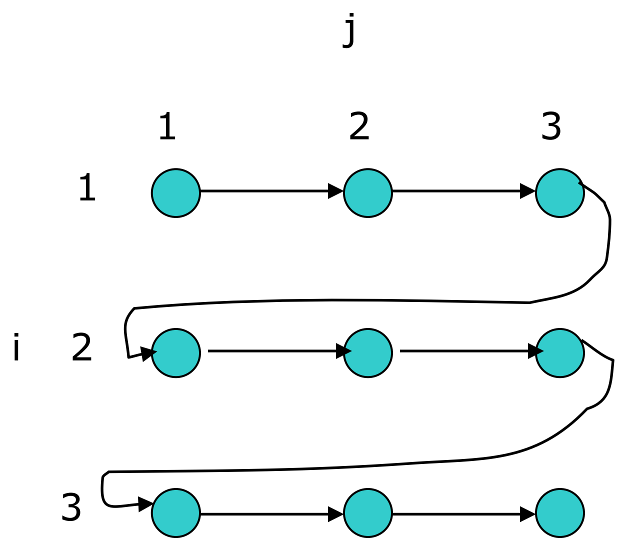
Iteration-space Traversal Graph (ITG)

[§3.2.1] The ITG shows graphically the order of traversal in the iteration space. This is sometimes called the *happens-before relationship*. In an ITG,

- A *node* represents a point in the iteration space
- A *directed edge* indicates 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;
```

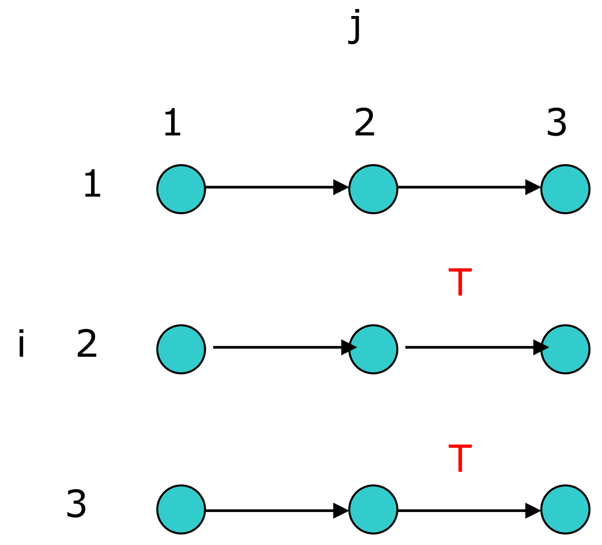


Loop-carried Dependence Graph (LDG)

- LDG shows the true/anti/output dependence relationship graphically.
- A *node* is a point in the iteration space.
- A *directed edge* represents the dependence.

Example:

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



Another example:

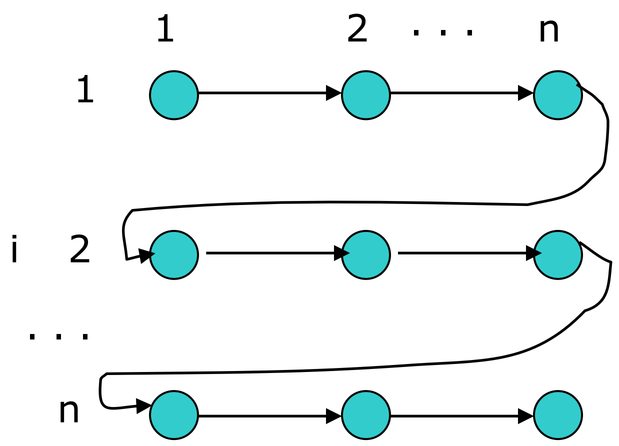
```
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++)
    S1: a[i][j] = a[i][j-1] + a[i][j+1] + a[i-1][j] + a[i+1][j];

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

Note that there are two “loop nests” in the code.

- The first involves S1.
- The other involves S2 and S3.

What do we know about the ITG for these nested loops?



Dependence relationships for Loop Nest 1

- True dependences:
  - `S1[i,j]` $\rightarrow$T `S1[i,j+1]`
  - `S1[i,j]` $\rightarrow$T `S1[i+1,j]`
- Output dependences:
  - None
- Anti-dependences:
  - `S1[i,j]` $\rightarrow$A `S1[i+1,j]`
  - `S1[i,j]` $\rightarrow$A `S1[i,j+1]`

Exercise: Suppose we dropped off the first half of S1, so we had

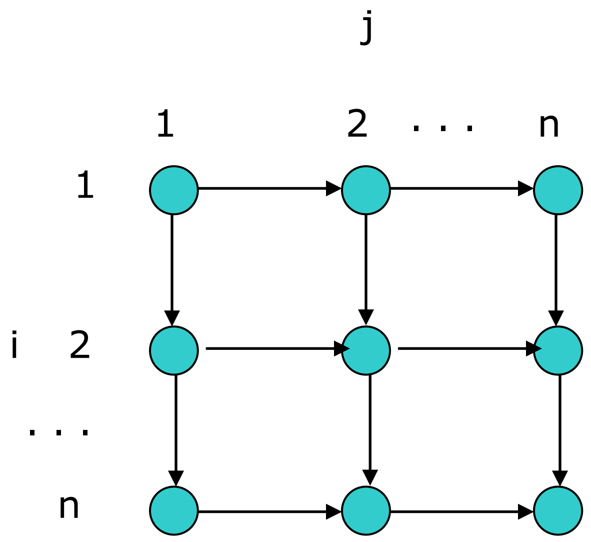
```
S1: a[i][j] = a[i-1][j] + a[i+1][j];
```

or the last half, so we had

```
S1: a[i][j] = a[i][j-1] + a[i][j+1];
```

Which of the dependences would still exist?

Draw the LDG for Loop Nest 1.

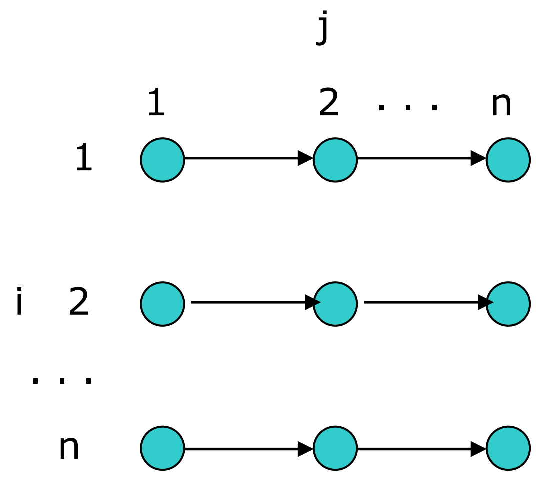


Note: each edge represents both true and anti-dependences

Dependence relationships 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)

Draw the LDG for Loop Nest 2.



Note: each edge represents only true dependences

Why are there no vertical edges in this graph? Answer here.

Why is the anti-dependence not shown on the graph?

Exercise: Consider this code sequence.

```
for (i = 3; i < n; i++) {
     for (j = 0; j < n - 3; j++) {
          S1: A[i][j] = A[i - 3][j] + A[i][j + 3];
          S2: B[i][j] = A[i][j] / 2;
     }
}
```

List the dependences, and say whether they are loop independent or loop carried. Then draw the ITG and LDG (you don't need to submit these).